Karlsruhe Institute of Technology

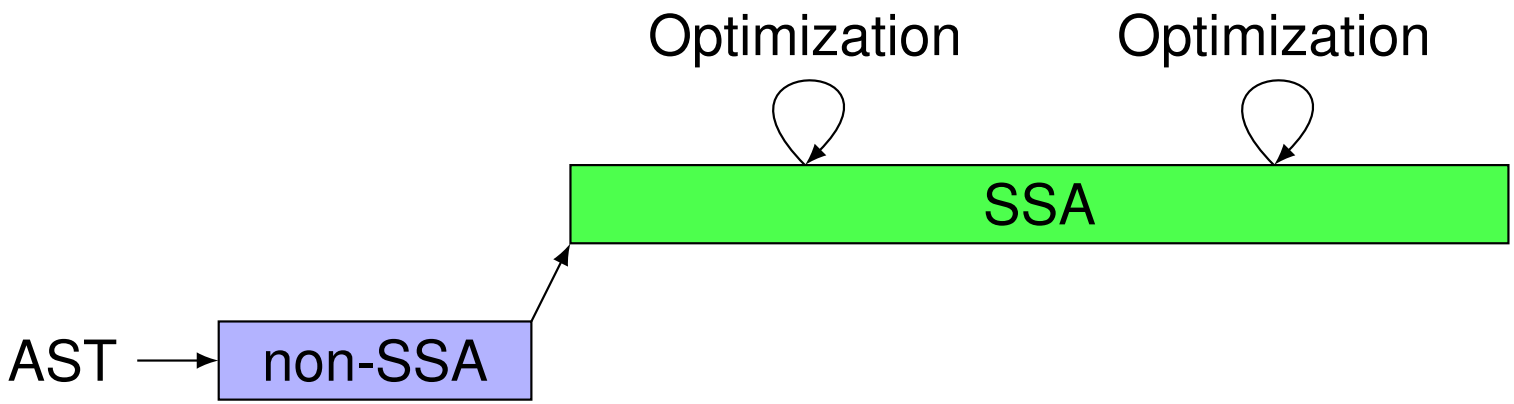
SAARLAND UNIVERSITY
COMPUTER SCIENCE

Simple and Efficient Construction of Static Single Assignment Form

Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon and Andreas Zwinkau

Institute for Program Structures and Data Organization, Karlsruhe Institute of Technology (KIT)

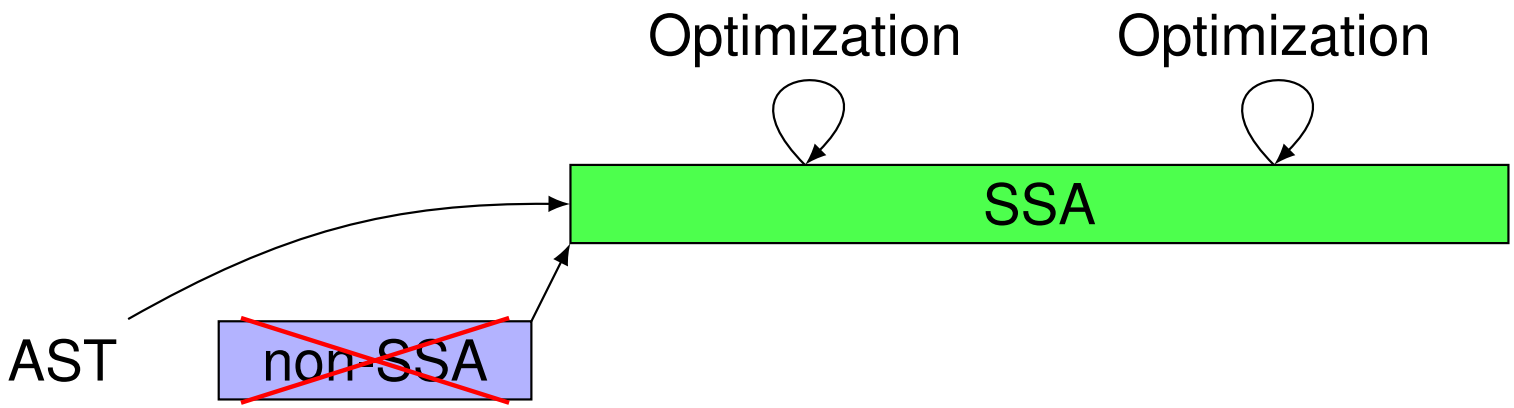
Modern Intermediate Representations



Two-phase approach:

- Construct non-SSA intermediate representation from AST
  - Compute dominance tree
  - Compute liveness
- Construct SSA form

Simple and Efficient SSA Construction



Our SSA construction algorithm

- Requires no prior analysis
- Constructs SSA form directly from AST

Straight-Line Code

Data structure:
For each basic block: Mapping Variables $\rightarrow$ Values

Code:

a = 42;
b = a;
a = a + b;

IR:

v_0: 42

Mapping:

"a" $\mapsto$ v_0

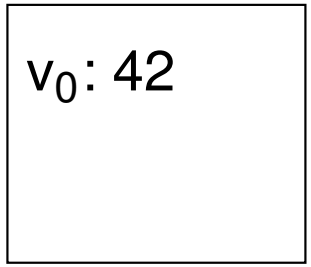
Straight-Line Code

Data structure:
For each basic block: Mapping Variables $\rightarrow$ Values

Code:

a = 42;
b = a;
a = a + b;

IR:



Mapping:

"a" $\mapsto$ v_0
"b" $\mapsto$ v_0

Straight-Line Code

Data structure:
For each basic block: Mapping Variables $\rightarrow$ Values

Code:

```
a = 42;
b = a;
a = a + b;
```

IR:

v_0: 42

v_1: v_0 + v_0

Mapping:

"a" $\mapsto$ v_0
"b" $\mapsto$ v_0

Straight-Line Code

Data structure:
For each basic block: Mapping Variables $\rightarrow$ Values

Code:

```
a = 42;
b = a;
a = a + b;
```

IR:

v_0: 42

v_1: v_0 + v_0

Mapping:

"a" $\mapsto$ v_1
"b" $\mapsto$ v_0

Properties of Basic Blocks

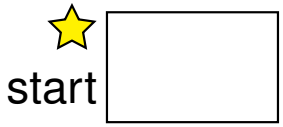
Filled

- All code of this block has already been constructed
- Existing mapping will not change

Sealed ☆

- The block is connected to all its direct control flow predecessors
- All direct control flow predecessors are filled

Handling Multiple Basic Blocks

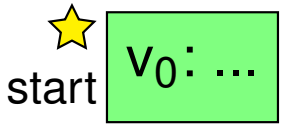



```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

Filled: Inserted all instructions

☆ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks

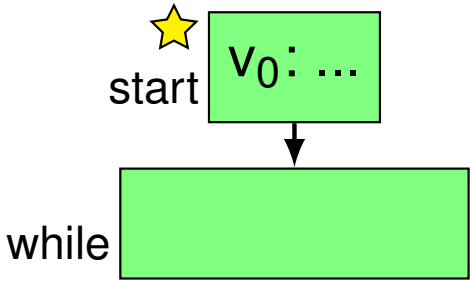



```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

Filled: Inserted all instructions

☆ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks



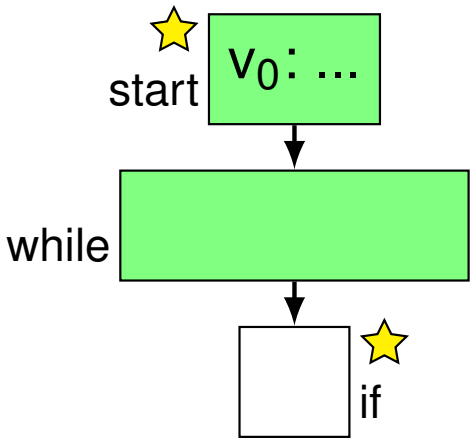

```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

Filled: Inserted all instructions

☆ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks

```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

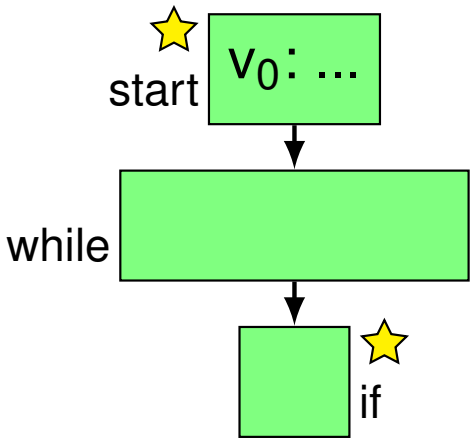


Filled: Inserted all instructions

☆ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks

```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

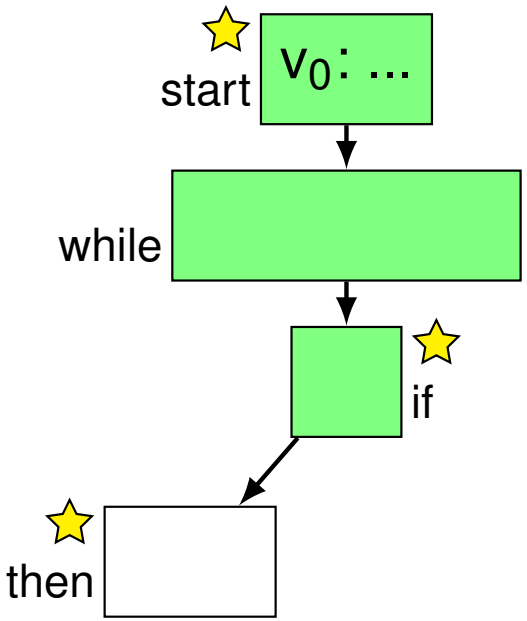


Filled: Inserted all instructions

☆ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks

```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

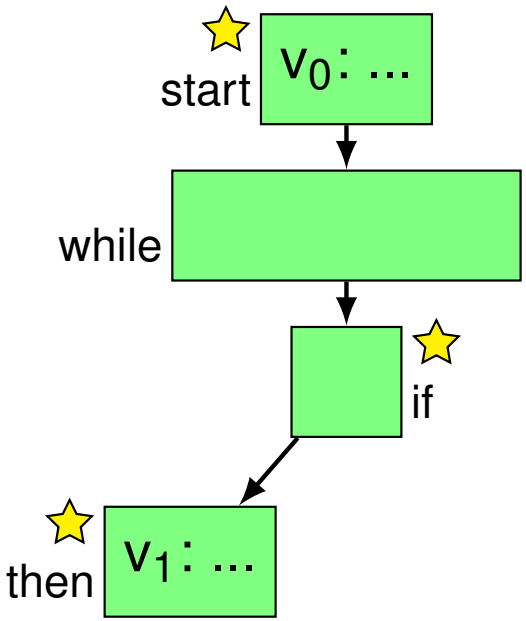


Filled: Inserted all instructions

☆ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks

```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

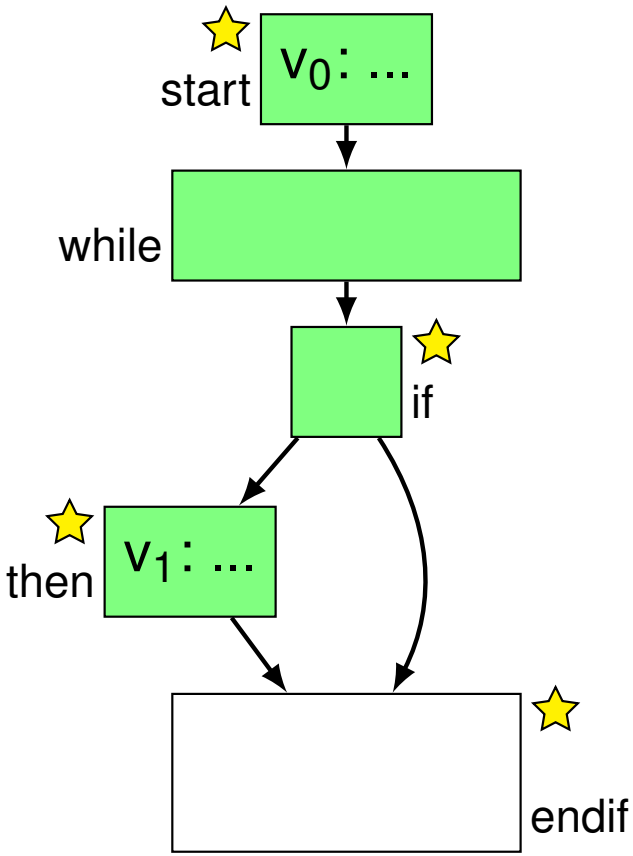


Filled: Inserted all instructions

☆ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks

```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

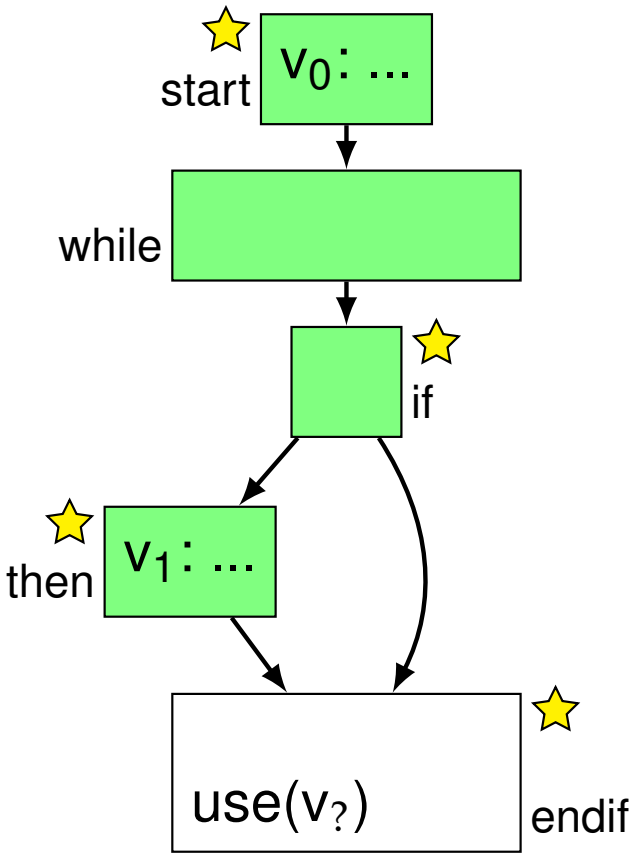


Filled: Inserted all instructions

☆ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks

```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

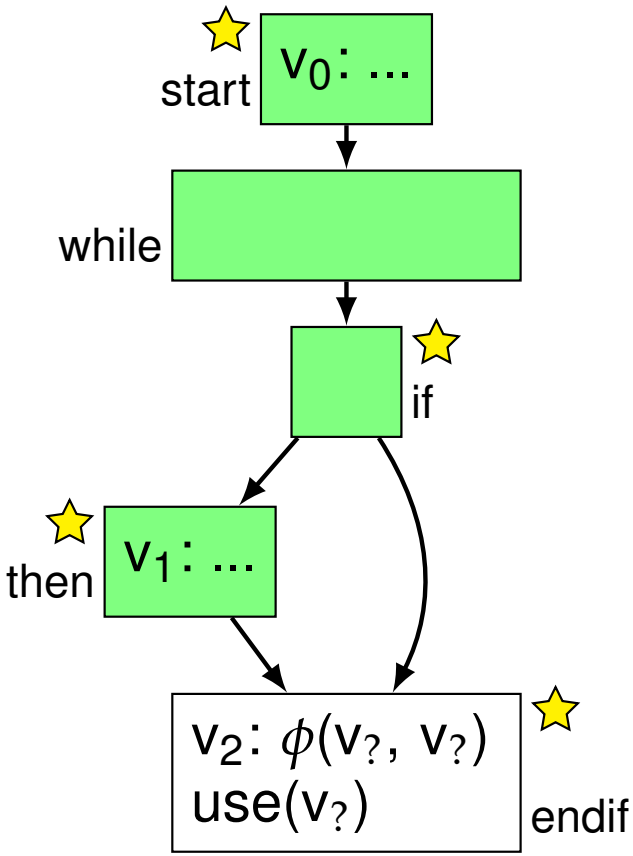


Filled: Inserted all instructions

☆ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks

```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

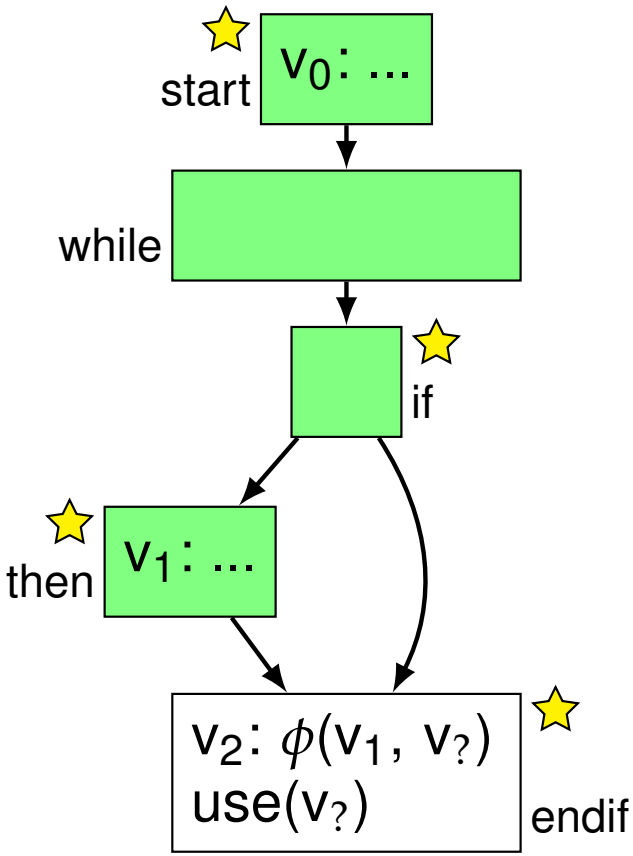


Filled: Inserted all instructions

☆ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks

```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

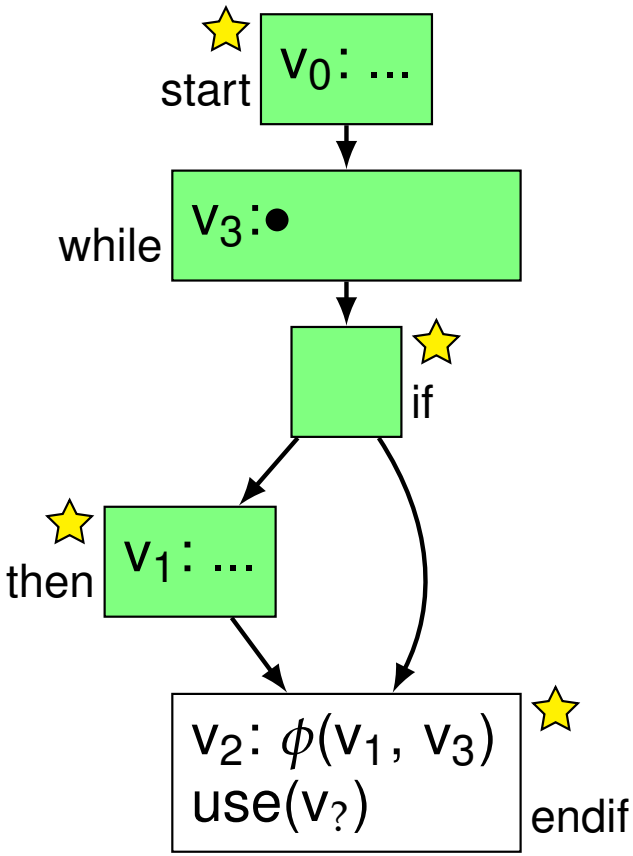


Filled: Inserted all instructions

☆ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks

```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

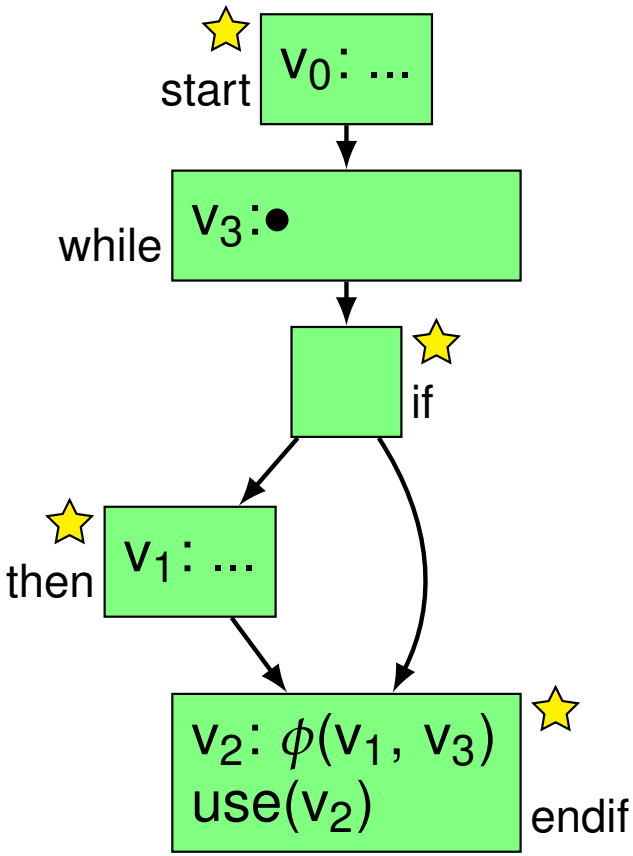


Filled: Inserted all instructions

☆ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks

```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

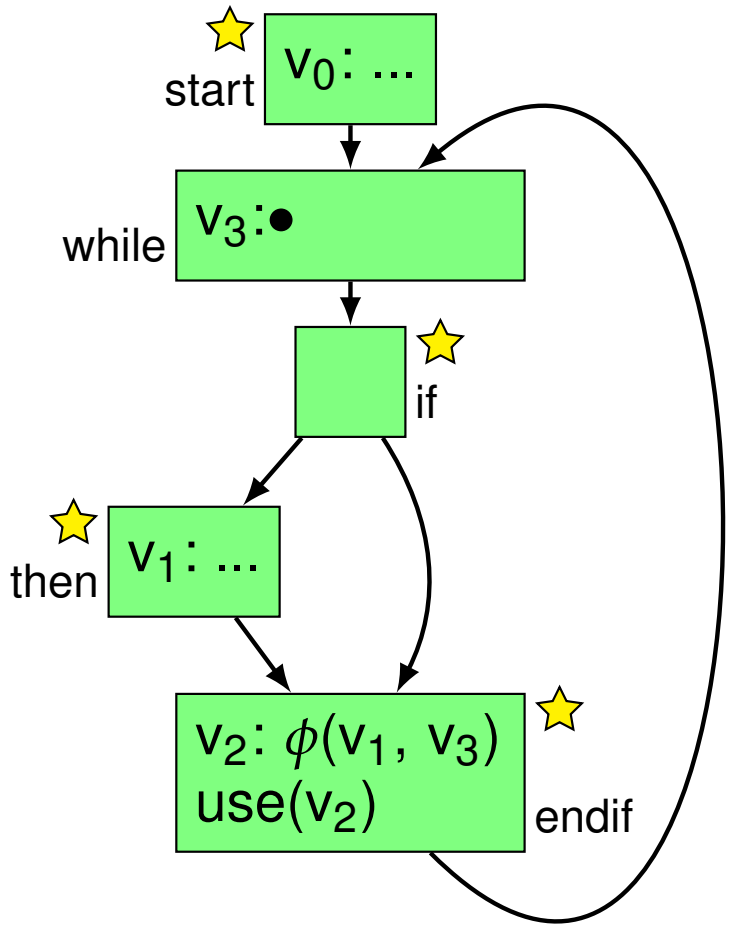


Filled: Inserted all instructions

★ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks

```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

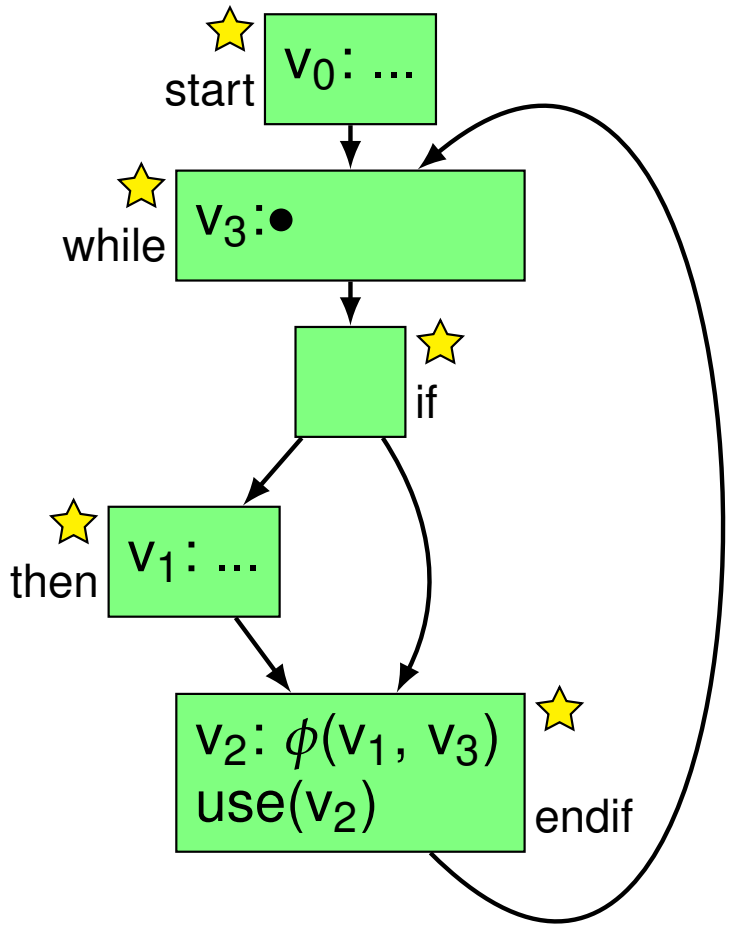


Filled: Inserted all instructions

★ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks

```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

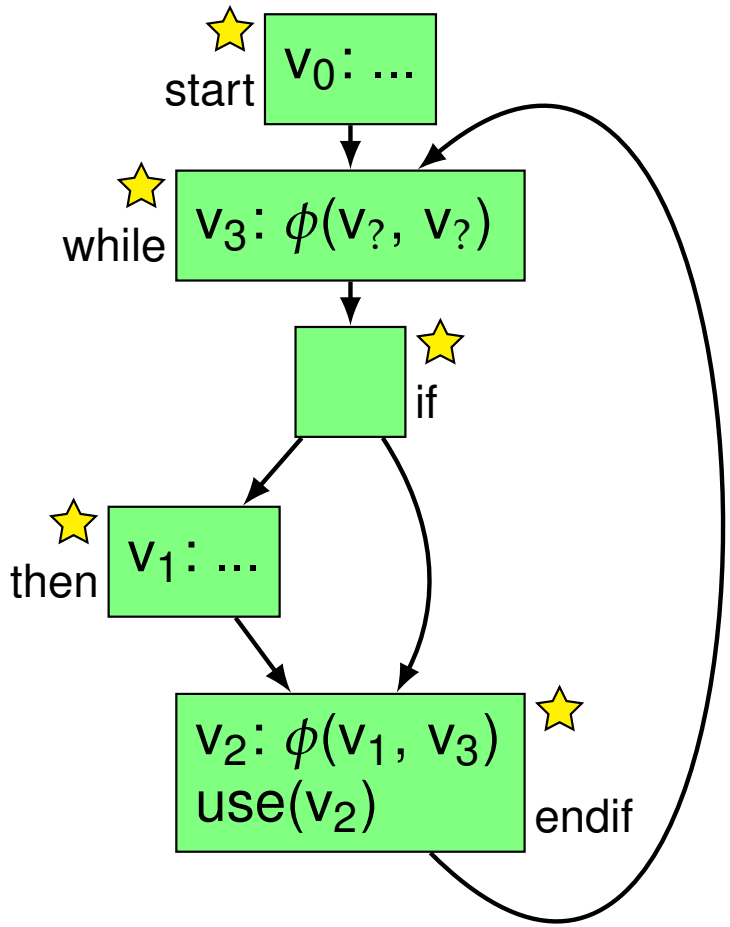


Filled: Inserted all instructions

★ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks

```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

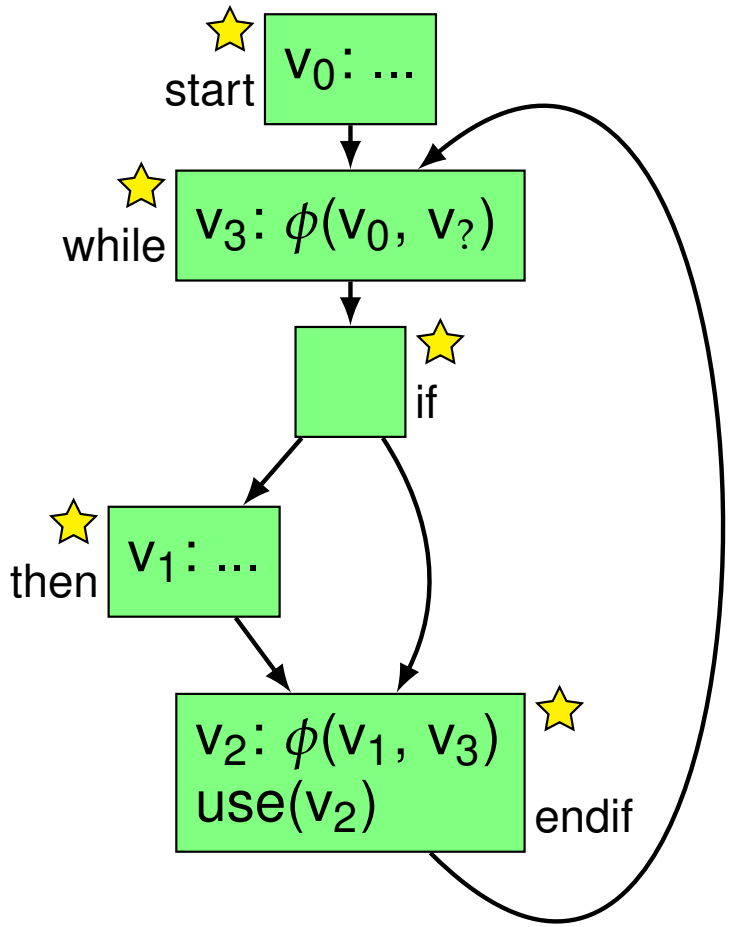


Filled: Inserted all instructions

☆ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks

```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

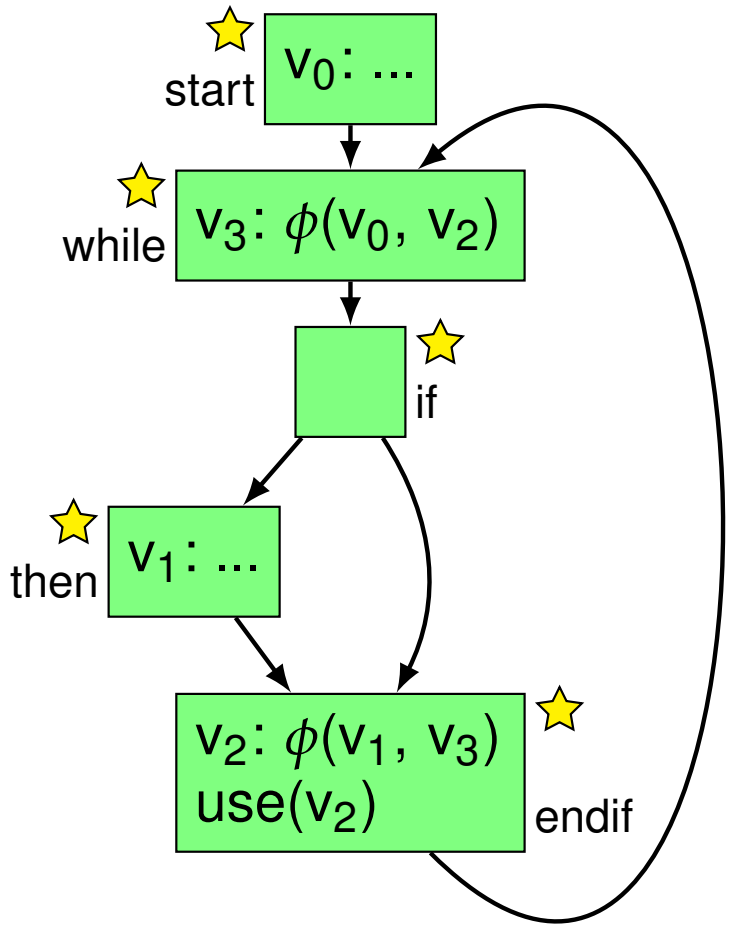


Filled: Inserted all instructions

☆ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks

```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

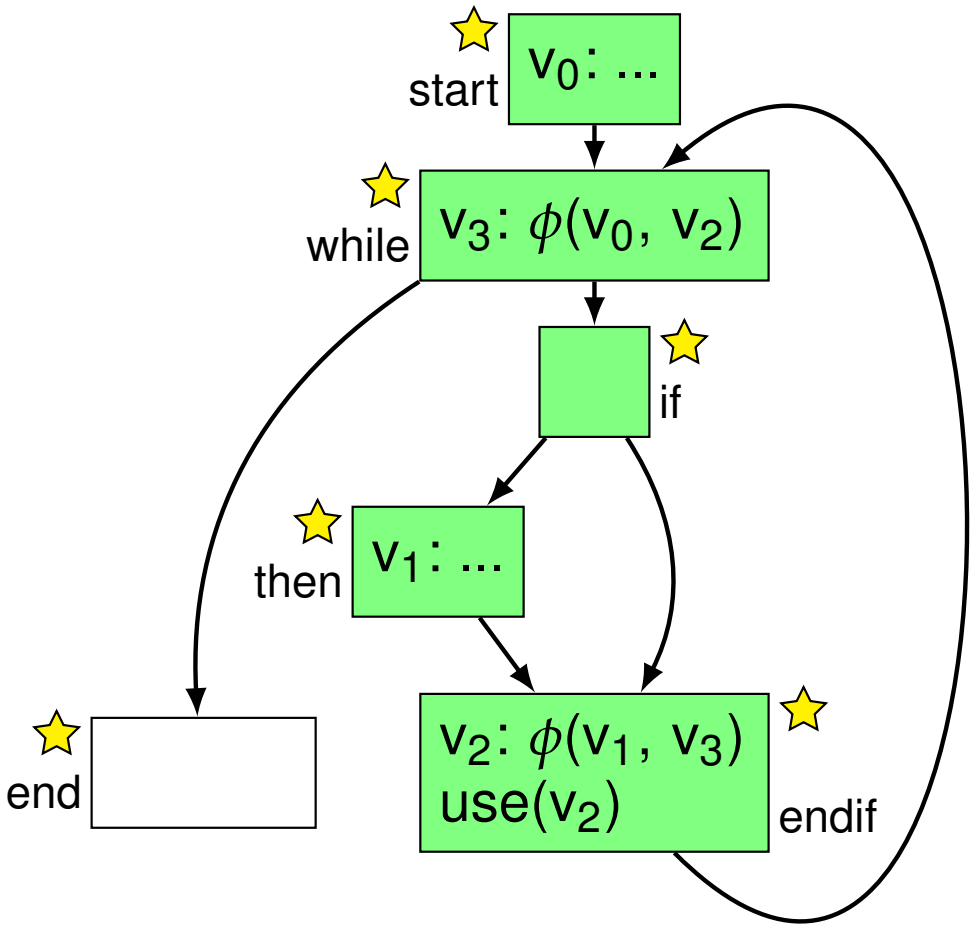


Filled: Inserted all instructions

☆ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks

```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

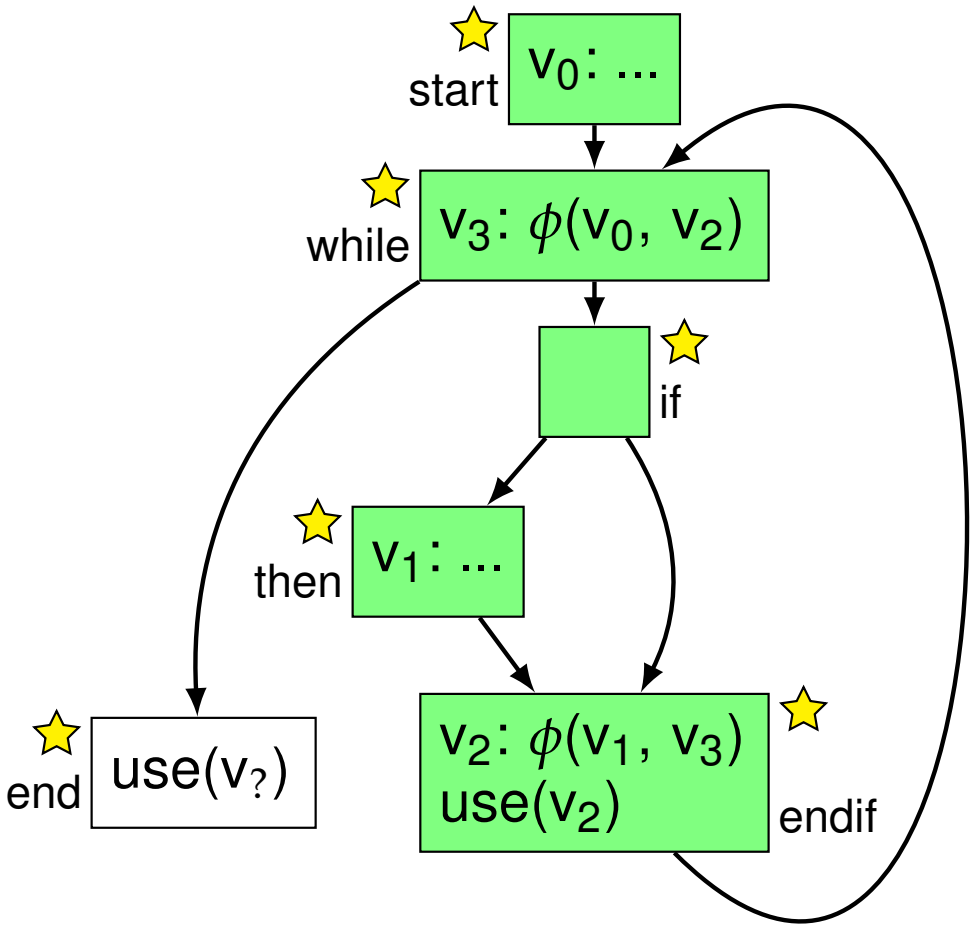


Filled: Inserted all instructions

★ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks

```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```

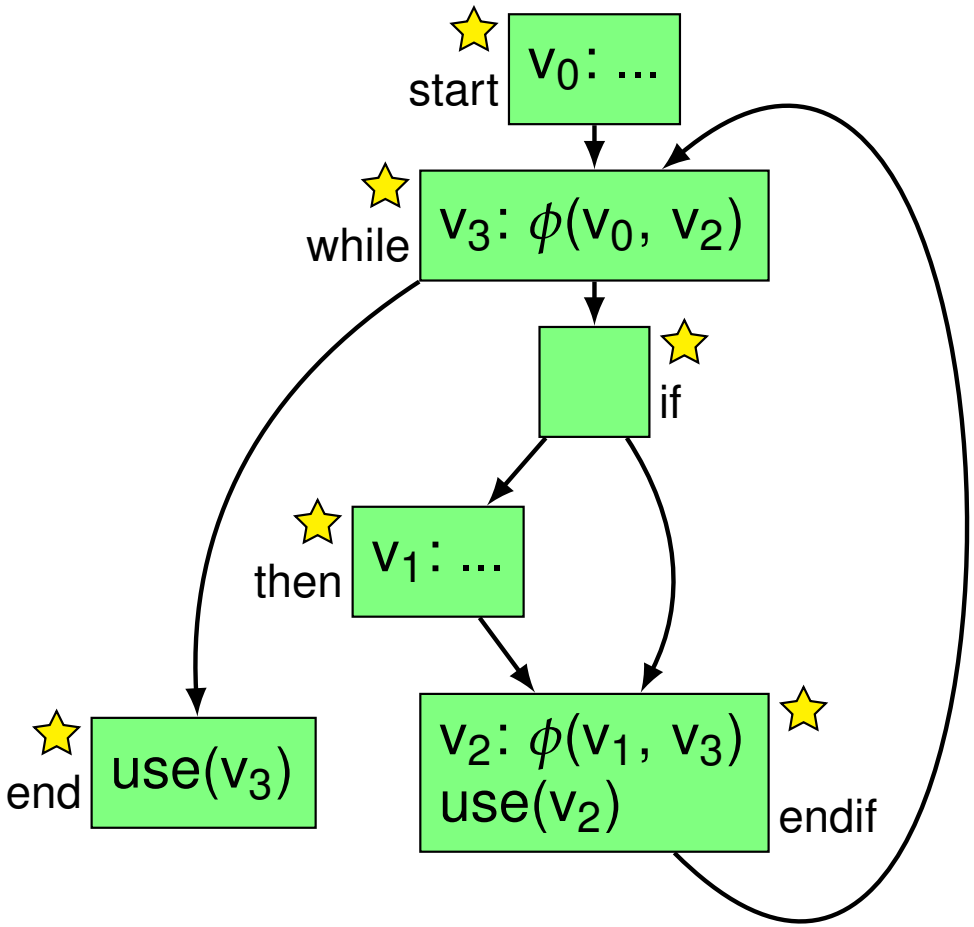


Filled: Inserted all instructions

☆ Sealed: Connected to all direct predecessors

Handling Multiple Basic Blocks

```
void foo(void) {
    x = ...
    while (...) {
        if (...) {
            x = ...
        }
        use(x)
    }
    use(x)
}
```



Filled: Inserted all instructions

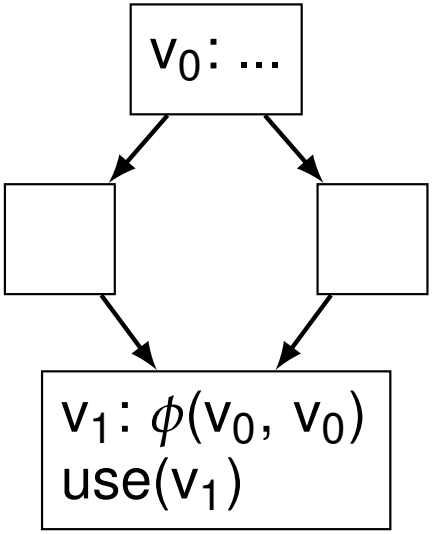
★ Sealed: Connected to all direct predecessors

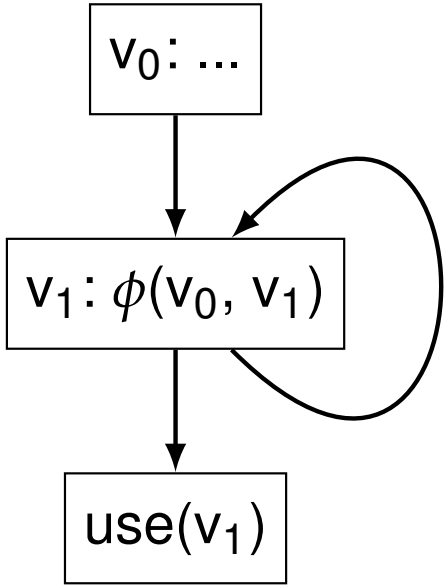
Guarantees for Constructed SSA Form

Pruned SSA form
No dead ϕ functions. ✓

Minimal SSA form (Cytron et al.)
ϕ functions only occur at iterated dominance frontiers. ?

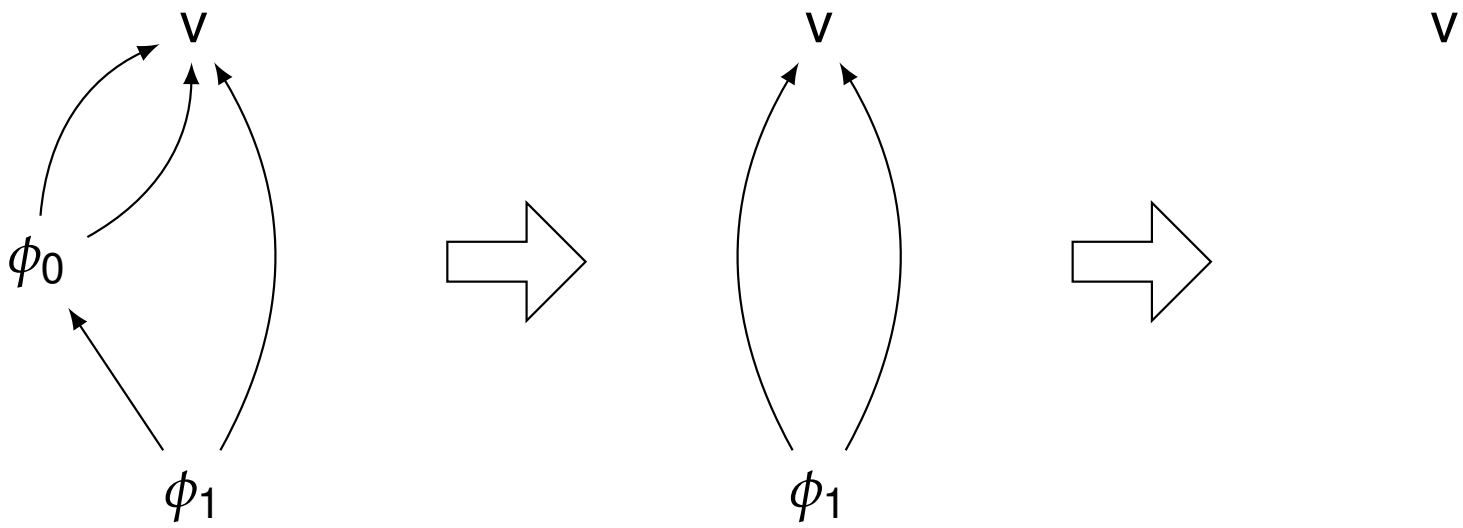
Trivial ϕ Functions





Trivial ϕ function
Just references one value (except itself)

Optimization of Trivial ϕ Functions



Optimization of trivial ϕ functions

- Replace trivial ϕ function by its unique operand
- Iterative approach to remove all trivial ϕ functions

Guarantees for Constructed SSA Form

Karlsruhe Institute of Technology

Theorem

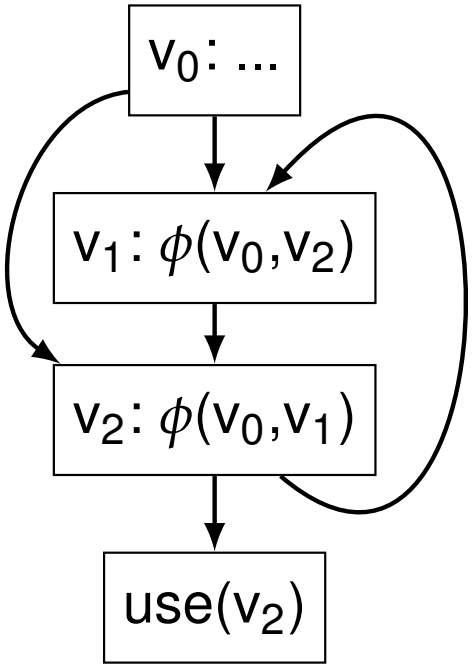
A program in SSA form with a reducible control flow graph without any trivial ϕ functions is in minimal SSA form.

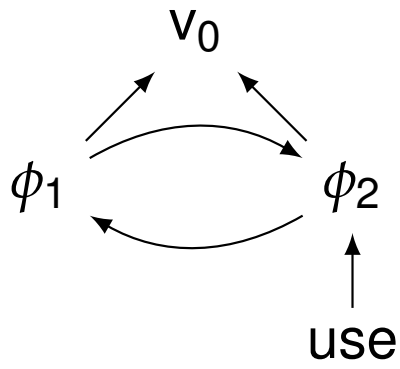
Pruned SSA form
No dead ϕ functions. ✓

Minimal SSA form (Cytron et al.)
ϕ functions only occur at iterated dominance frontiers. ✓*

$*$ for Java and most C programs

Irreducible Control Flow

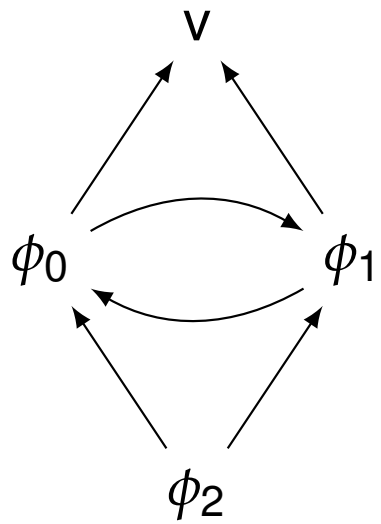




Redundant set of ϕ functions

- Generalization of trivial ϕ functions
- Just reference one value (except themselves)

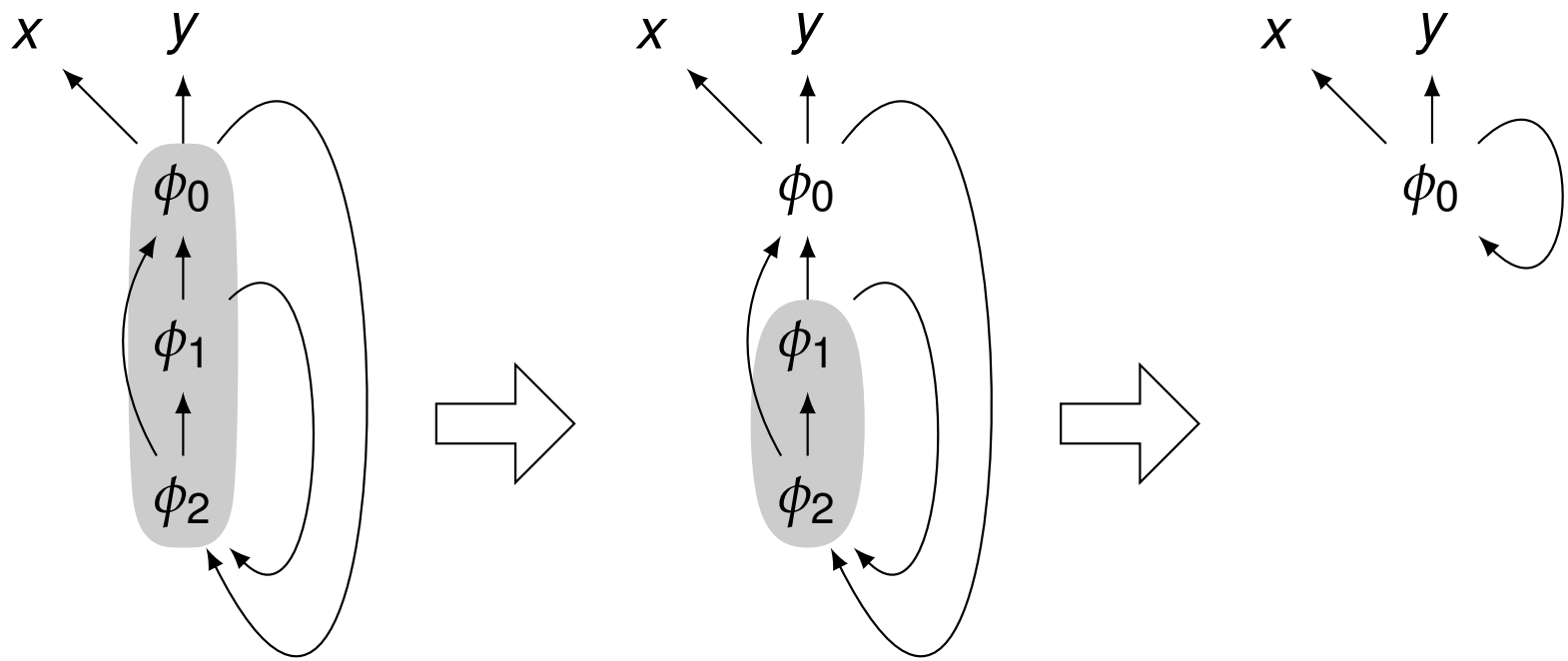
Redundant ϕ Functions and SCCs



Lemma

Let P be a redundant set of ϕ functions. Then there is a strongly connected component $S \subseteq P$ that is also redundant.

Optimization of Redundant ϕ Functions



Algorithm to find all redundant ϕ functions

- Use Tarjan's algorithm to find ϕ-SCCs of maximum size
- If SCC is not redundant: Check for redundant inner SCC

New Criterion for Minimal SSA Form

Karlsruhe Institute of Technology

Definition (Minimal SSA form (SCC))

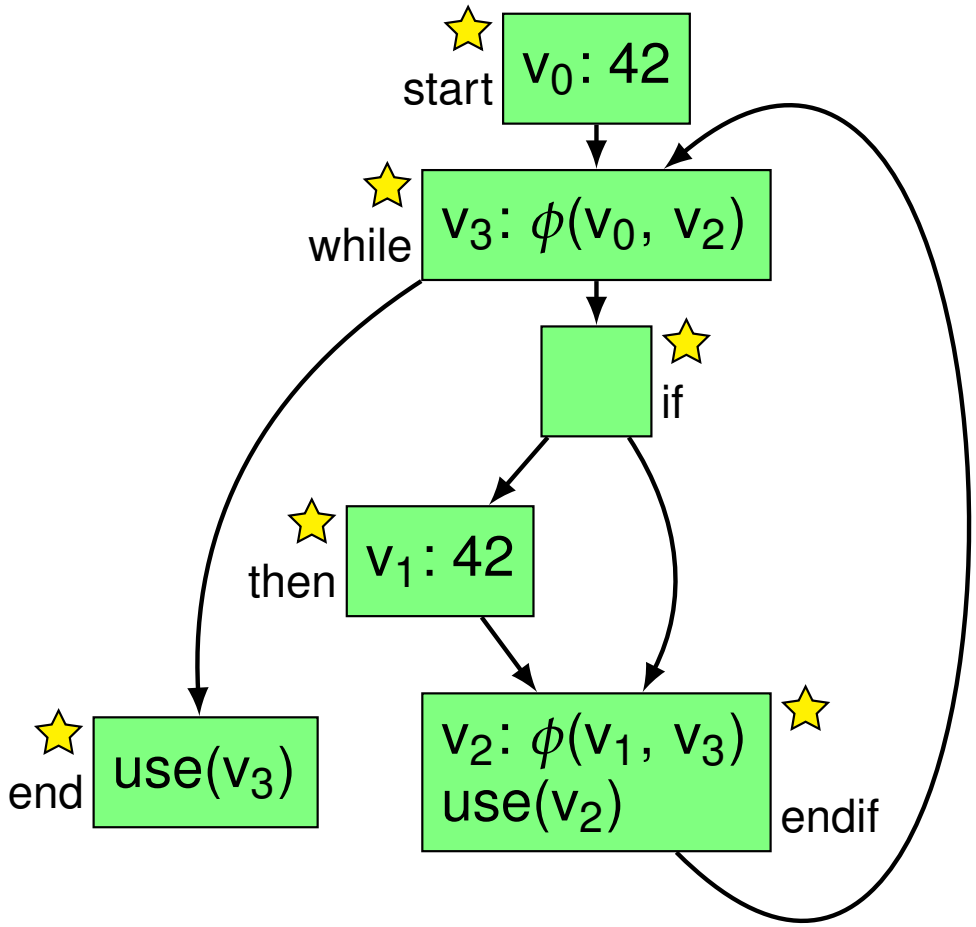
A program is in minimal SSA form if it contains no redundant ϕ-SCCs.

Advantages

- Independent of the source program
- Implies Cytron et al.'s criterion

Optimizations and ϕ-SCCs

```
void foo(void) {
    x = 42
    while (...) {
        if (...) {
            x = 42
        }
        use(x)
    }
    use(x)
}
```



Guarantees for Constructed SSA Form

Pruned SSA form
No dead ϕ functions. ✓

Minimal SSA form (Cytron et al.)
ϕ functions only occur at iterated dominance frontiers. ✓

Minimal SSA form (SCC)
No redundant ϕ-SCC. ✓

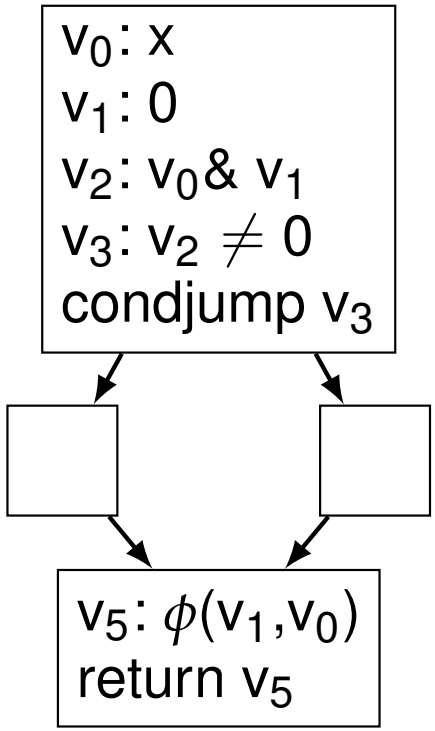
Can we have even fewer ϕ functions?

On-the-Fly Optimizations

```
int bar(int x) {
    int mask = 0;
    int res;

    if (x & mask) {
        res = 0;
    } else {
        res = x;
    }

    return res;
}
```

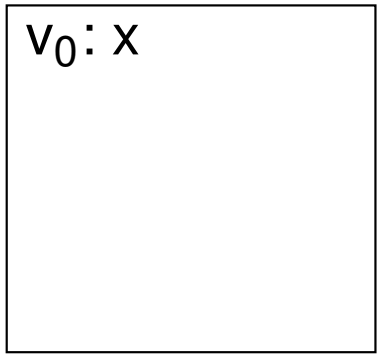


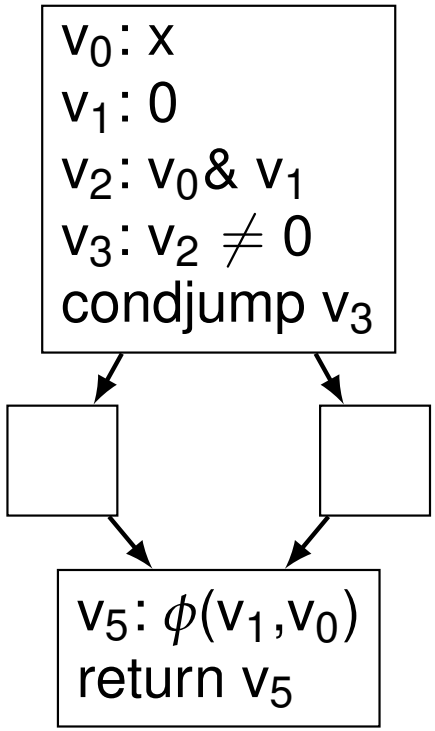
On-the-Fly Optimizations

```
int bar(int x) {
    int mask = 0;
    int res;

    if (x & mask) {
        res = 0;
    } else {
        res = x;
    }

    return res;
}
```



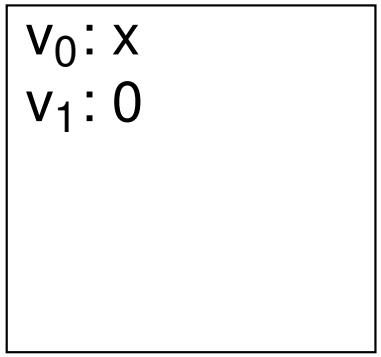


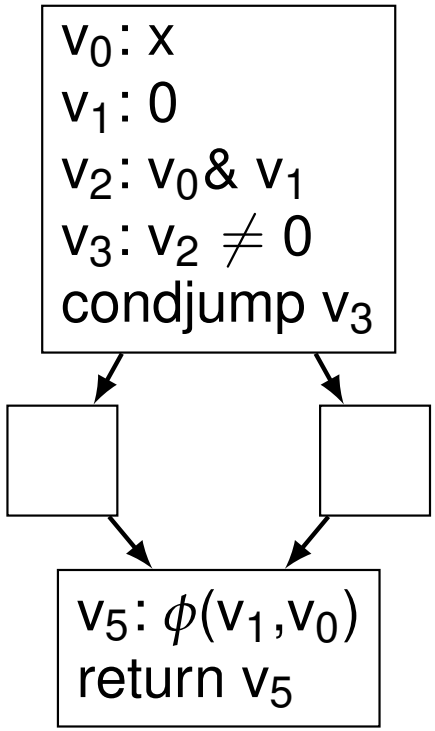
On-the-Fly Optimizations

```
int bar(int x) {
    int mask = 0;
    int res;

    if (x & mask) {
        res = 0;
    } else {
        res = x;
    }

    return res;
}
```



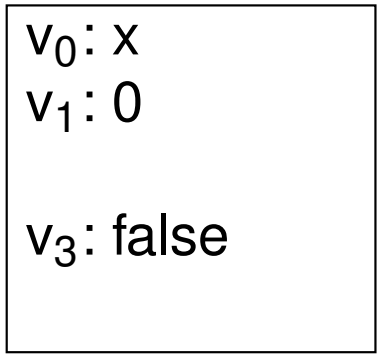


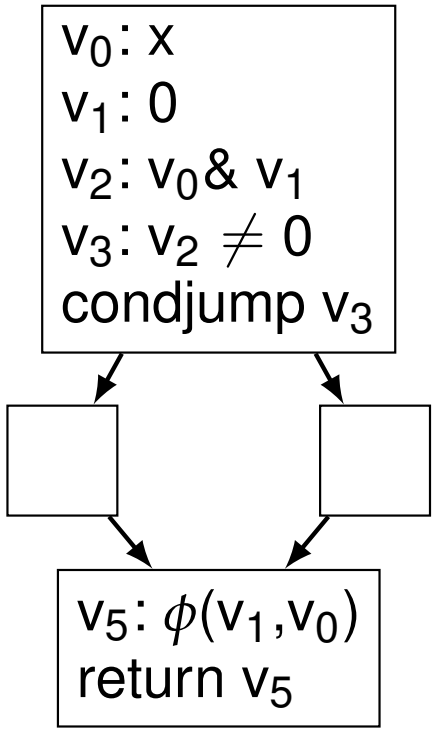
On-the-Fly Optimizations

```
int bar(int x) {
    int mask = 0;
    int res;

    if (x & mask) {
        res = 0;
    } else {
        res = x;
    }

    return res;
}
```



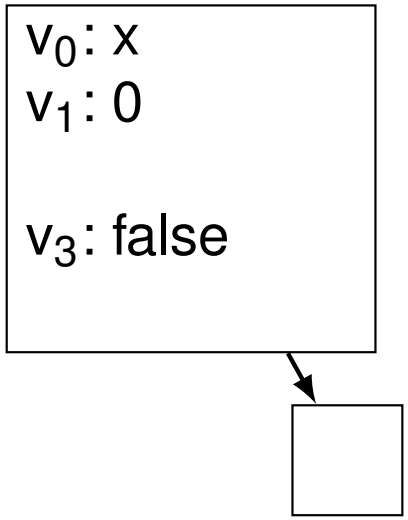


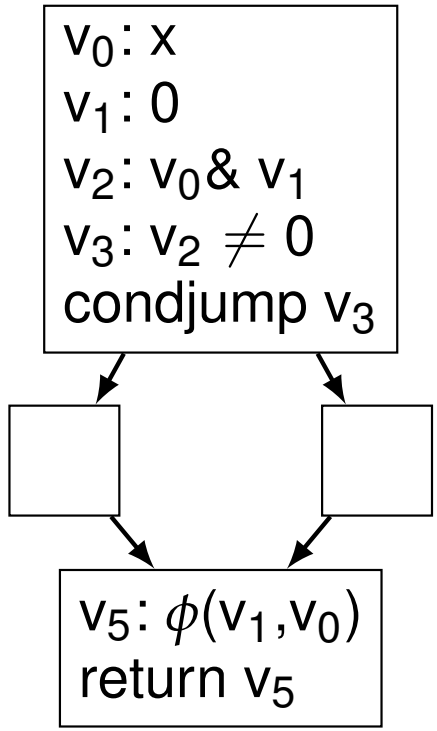
On-the-Fly Optimizations

```
int bar(int x) {
    int mask = 0;
    int res;

    if (x & mask) {
        res = 0;
    } else {
        res = x;
    }

    return res;
}
```



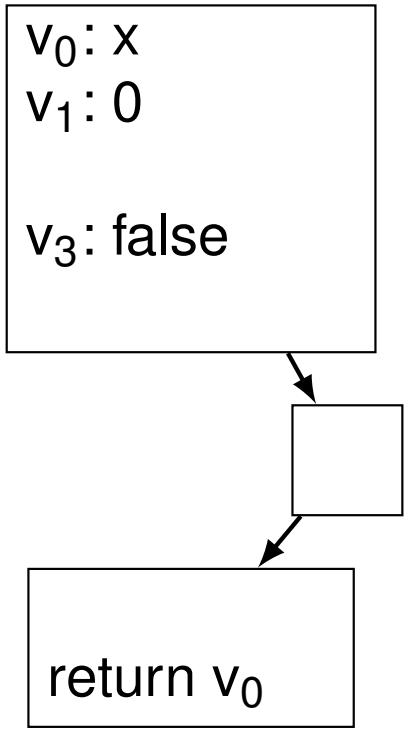


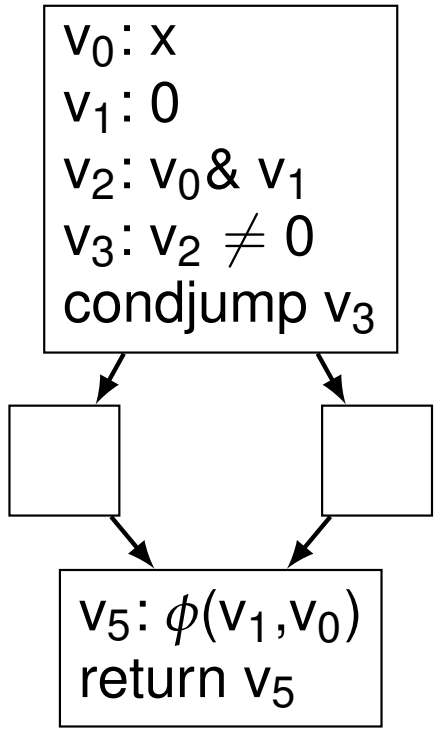
On-the-Fly Optimizations

```
int bar(int x) {
    int mask = 0;
    int res;

    if (x & mask) {
        res = 0;
    } else {
        res = x;
    }

    return res;
}
```





Evaluation Setup

Setup to evaluate our SSA construction algorithm

- Full-fledged implementation in cparser/libFirm
- Proof-of-concept implementation in clang/LLVM
- Comparison against existing SSA construction algorithm
  - Sreedhar and Gao
  - Better than Cytron et al.
  - Highly-tuned implementation
- C programs of SPEC CINT2000 benchmark

Number of Constructed IR Instructions

Benchmark	LLVM		non-SSA / SSA
	non-SSA	SSA	
164.gzip	12,038	9,187	131%
175.vpr	40,701	27,155	150%
176.gcc	516,537	395,652	131%
181.mcf	3,988	2,613	153%
186.crafty	44,891	36,050	125%
197.parser	30,237	20,485	148%
253.perlbmk	185,576	140,489	132%
254.gap	201,185	149,755	134%
255.vortex	126,097	88,257	143%
256.bzip2	8,605	6,012	143%
300.twolf	76,078	58,737	130%
Average			138%

Quality Comparison

Benchmark	Number of ϕ functions			Redundant ϕ-SCCs	
	LLVM	Our	Δ	Optimization	Irreducible
164.gzip	594	594	0	0	0
175.vpr	1,201	1,201	0	0	0
176.gcc	12,904	12,910	6	2	3
181.mcf	154	154	0	0	0
186.crafty	1,466	1,466	0	0	0
197.parser	1,243	1,243	0	0	0
253.perlbmk	5,840	5,857	17	0	5
254.gap	9,325	9,326	1	0	0
255.vortex	2,739	2,737	2	1	0
256.bzip2	359	359	0	0	0
300.twolf	2,849	2,849	0	0	0
Sum	38,674	38,696	22	3	8

Compilation Speed

Benchmark	LLVM	Our	$\frac{\text{Our}}{\text{LLVM}}$
164.gzip	969,233,677	967,798,047	99.85%
175.vpr	3,039,801,575	3,025,286,080	99.52%
176.gcc	25,935,984,569	26,009,545,723	100.28%
181.mcf	722,918,540	722,507,455	99.94%
186.crafty	3,653,881,430	3,632,605,590	99.42%
197.parser	2,084,205,254	2,068,075,482	99.23%
253.perlbmk	12,246,953,644	12,062,833,383	98.50%
254.gap	8,358,757,289	8,339,871,545	99.77%
255.vortex	7,841,416,740	7,845,699,772	100.05%
256.bzip2	569,176,687	564,577,209	99.19%
300.twolf	6,424,027,368	6,408,289,297	99.76%
Average			99.59%

Conclusion

Karlsruhe Institute of Technology

Summary

- Simple SSA construction algorithm: 375 LOC vs. 1141 LOC
- Guarantees minimal and pruned SSA form
- Can reuse conservative optimizations
- New criterion for minimal SSA form
- A lot more in the paper

Give it a try!