Static Single Assignment Form in the COINS Compiler Infrastructure

Masataka Sassa

(Tokyo Institute of Technology)

Background

Static single assignment (SSA) form facilitates compiler optimizations
Compiler infrastructure facilitates compiler development.

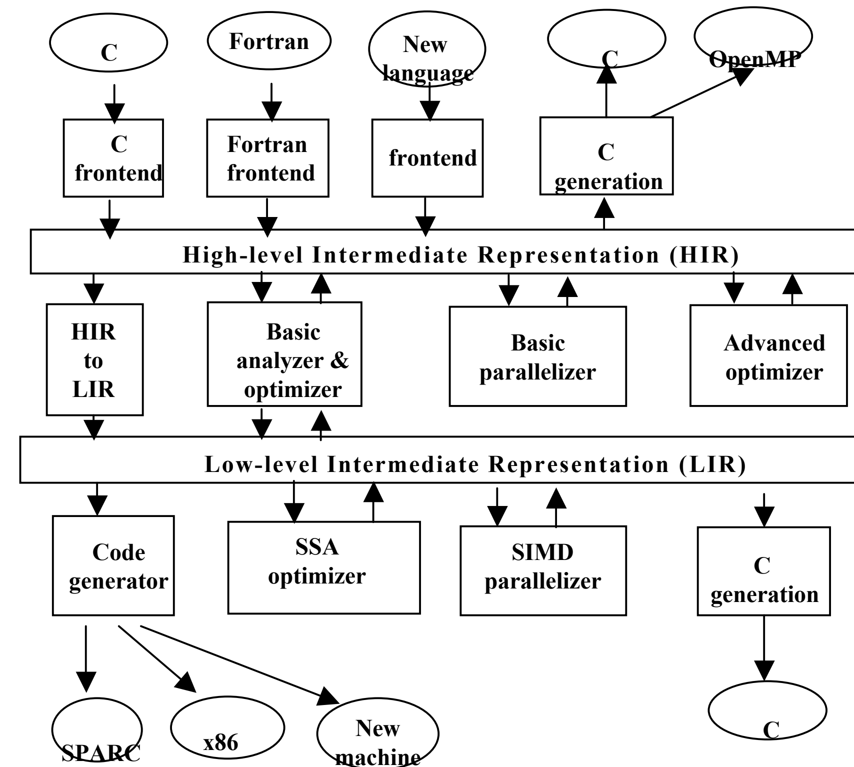
Outline

0. COINS infrastructure and the SSA form
1. SSA optimization module in COINS
2. A comparison of two major algorithms for translating from normal form into SSA form
3. A comparison of two major algorithms for translating back from SSA form into normal form
4. Examples of SSA form optimization
5. Preliminary experimental results

0. COINS infrastructure and the static single assignment form (SSA form)

COINS compiler infrastructure

- Multiple source languages
- Retargetable
- Two intermediate forms, HIR and LIR
- Optimizations
- Parallelization
- C generation, source-to-source translation
- Written in Java
- 2000~ developed by Japanese institutions under Grant of the Ministry



Static single assignment (SSA) form

```
1: a = x + y
2: a = a + 3
3: b = x + y
```

(a) Normal (conventional) form (source program or internal form)

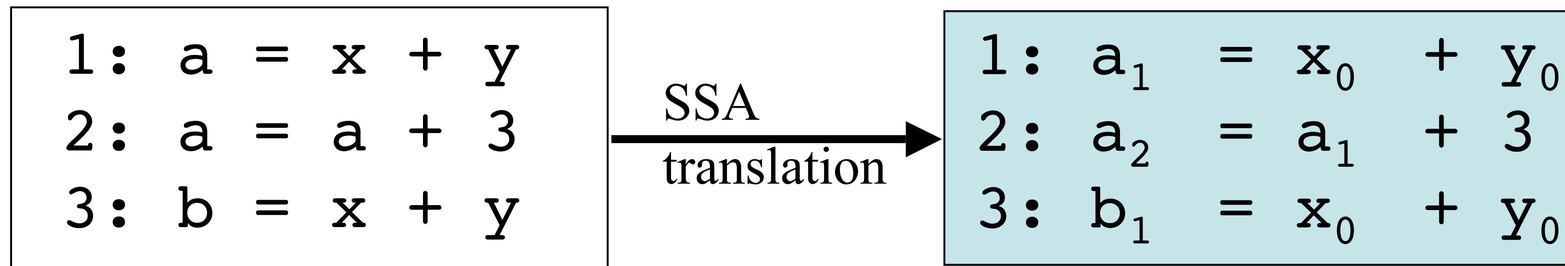
$1{:}\ a_1 = x_0 + y_0$
$2{:}\ a_2 = a_1 + 3$
$3{:}\ b_1 = x_0 + y_0$

(b) SSA form

SSA form is a recently proposed internal representation where each use of a variable has a single definition point.

Indices are attached to variables so that their definitions become unique.

Optimization in static single assignment (SSA) form



(a) Normal form

(b) SSA form

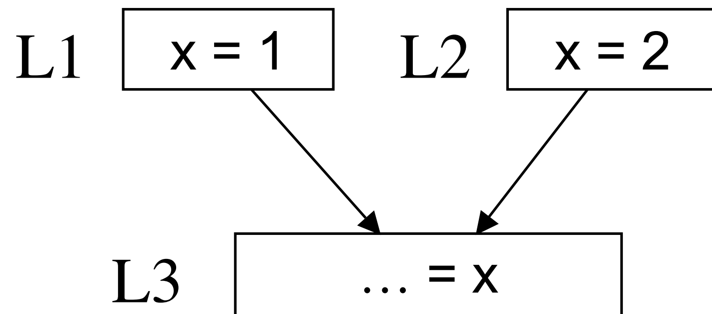
Optimization in SSA form (common subexpression elimination)

(c) After SSA form optimization

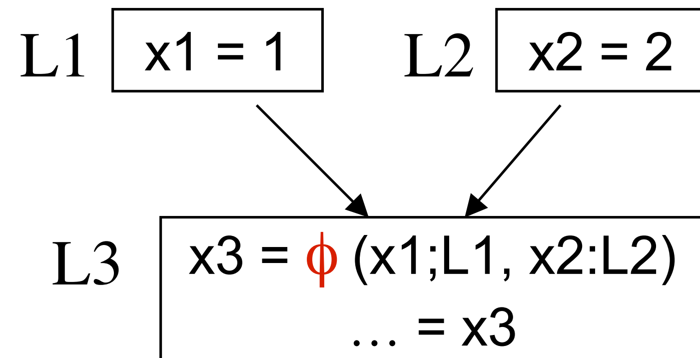
(d) Optimized normal form

SSA form is becoming increasingly popular in compilers, since it is suited for clear handling of dataflow analysis and optimization.

Translating into SSA form (SSA translation)

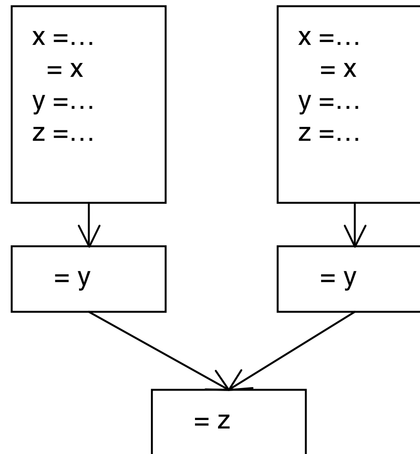


(a) Normal form

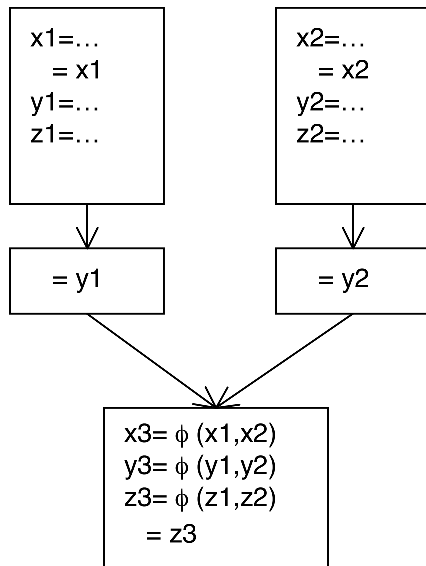


(b) SSA form

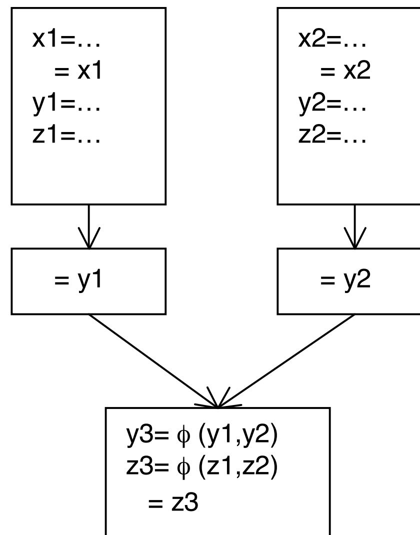
Translating into SSA form (SSA translation)



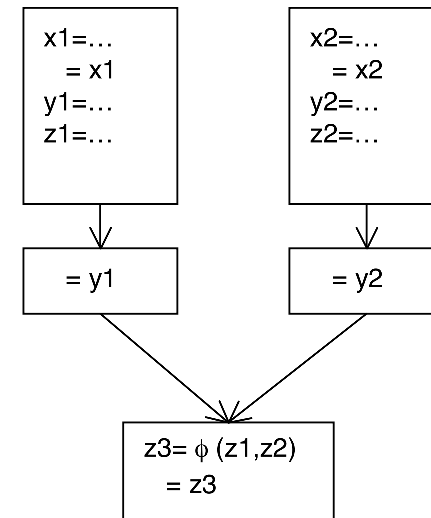
Normal form



Minimal SSA form

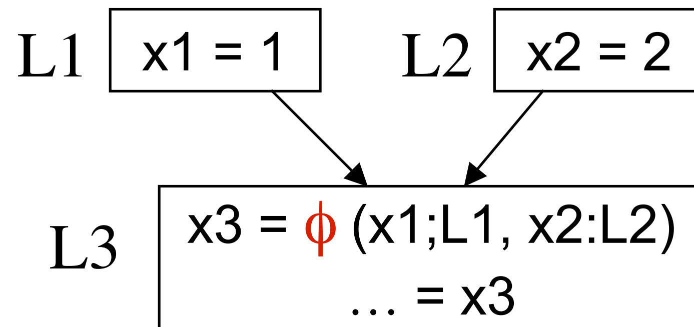


Semi-pruned SSA form

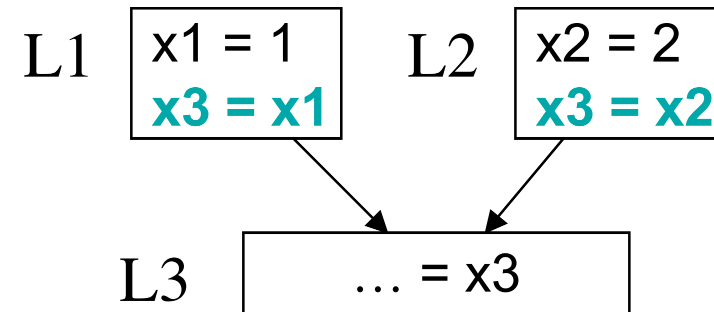


Pruned SSA form

Translating back from SSA form (SSA back translation)



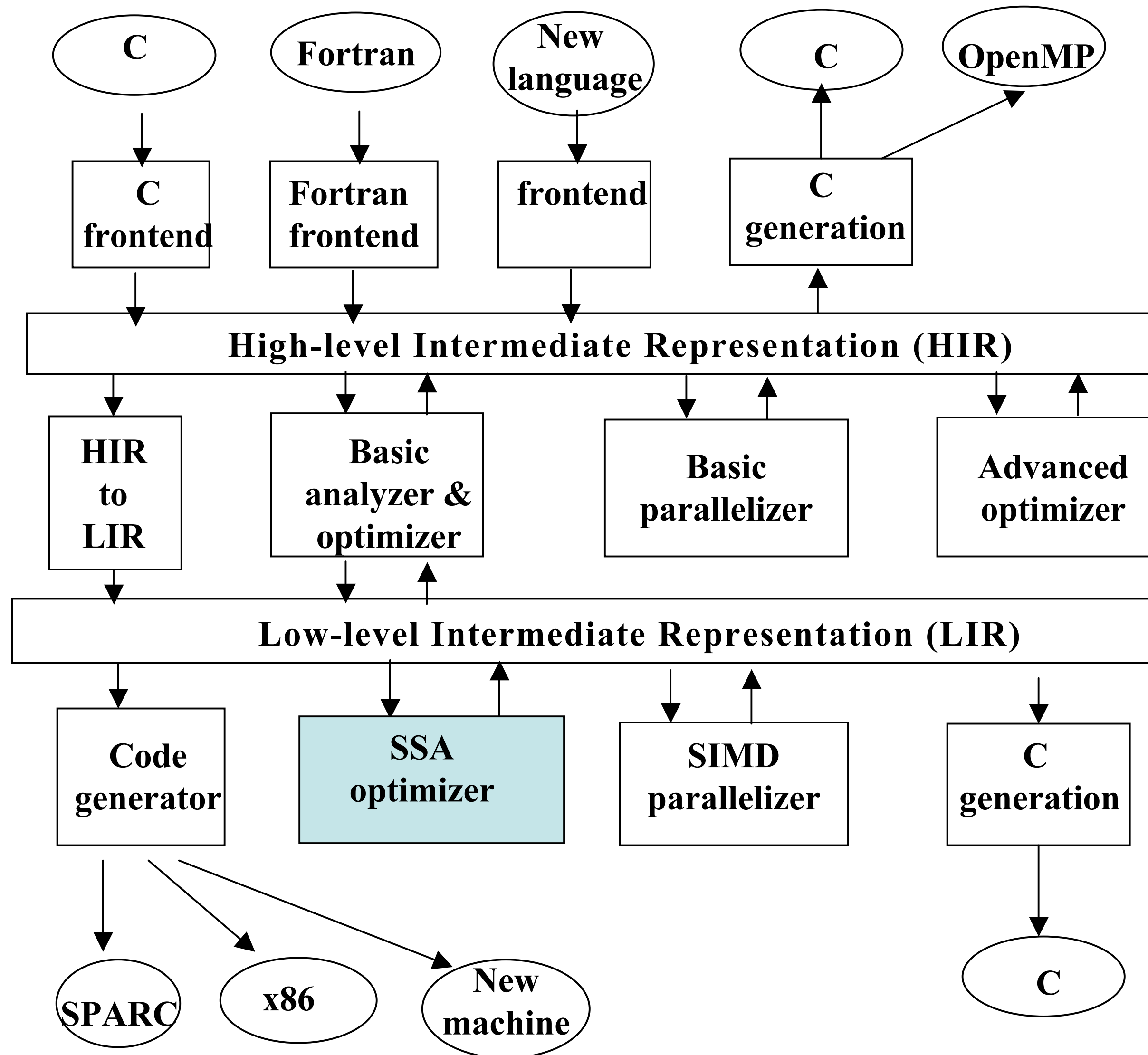
(a) SSA form



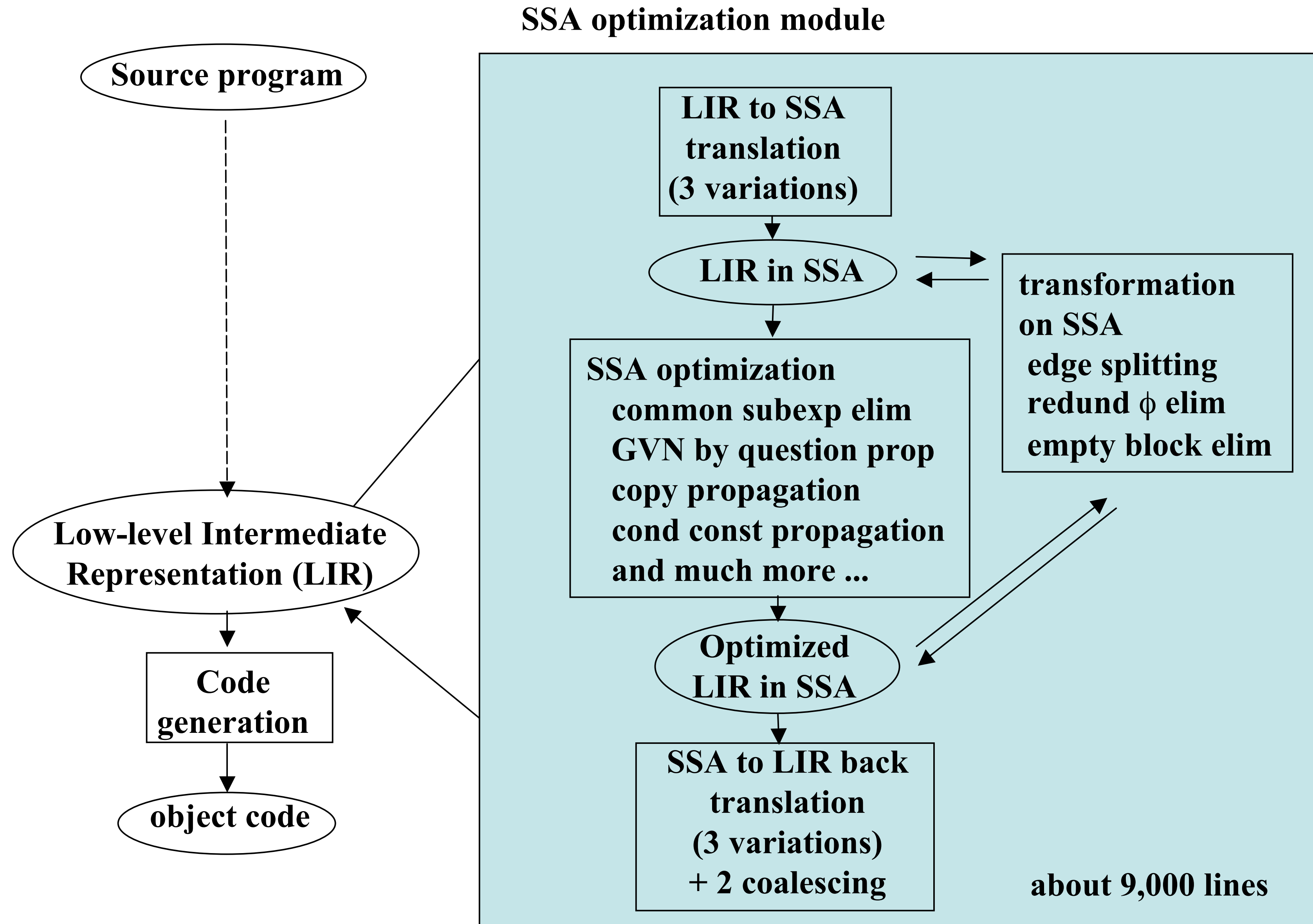
(b) Normal form

1. SSA optimization module in COINS

COINS compiler infrastructure



SSA optimization module in COINS



Outline of SSA module in COINS(1)

- Translation into and back from SSA form on Low-level Intermediate Representation (LIR)
  - SSA translation
    - Use dominance frontier [Cytron et al. 91]
    - 3 variations: translation into Minimal , Semi-pruned and Pruned SSA forms
  - SSA back translation [Sreedhar et al. 99]
    - 3 variations: Method I, II, and III
  - Coalescing
    - SSA-based coalescing during SSA back translation [Sreedhar et al. 99]
    - Chaitin's style coalescing after back translation
  - Each variation and coalescing can be specified by options

Outline of SSA module in COINS (2)

- Several optimization on SSA form:

  dead code elimination, copy propagation, common subexpression elimination, global value numbering based on efficient query propagation, conditional constant propagation, loop invariant code motion, operator strength reduction for induction variable and linear function test replacement, empty block elimination, copy folding at SSA translation time ...

- Useful transformation as an infrastructure for SSA form optimization:

  critical edge removal on control flow graph, loop transformation from 'while' loop to 'if-do-while' loop, redundant phi-function elimination, making SSA graph …

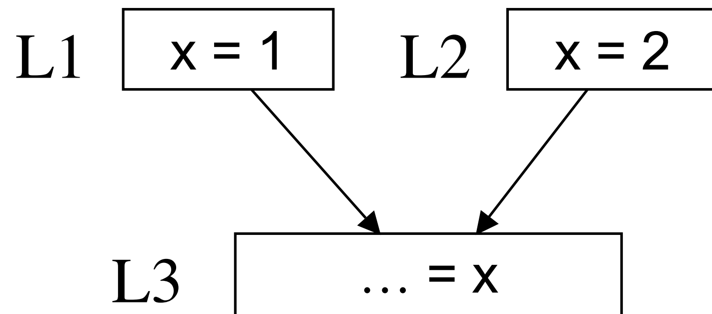
- Each variation, optimization and transformation can be made selectively by specifying options

2. A comparison of two major algorithms for SSA translation

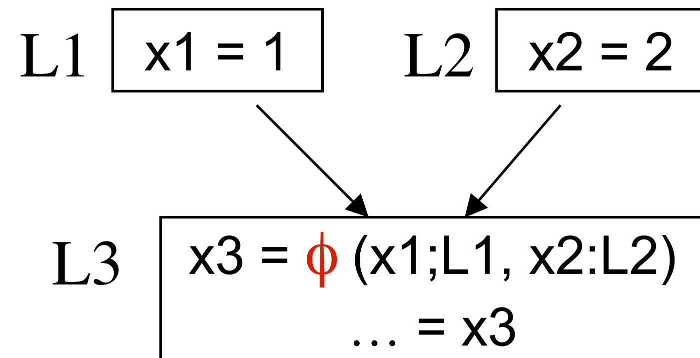
- Algorithm by Cytron [1991]
  - Dominance frontier
- Algorithm by Sreedhar [1995]
  - DJ-graph

Comparison made to decide the algorithm to be included in COINS

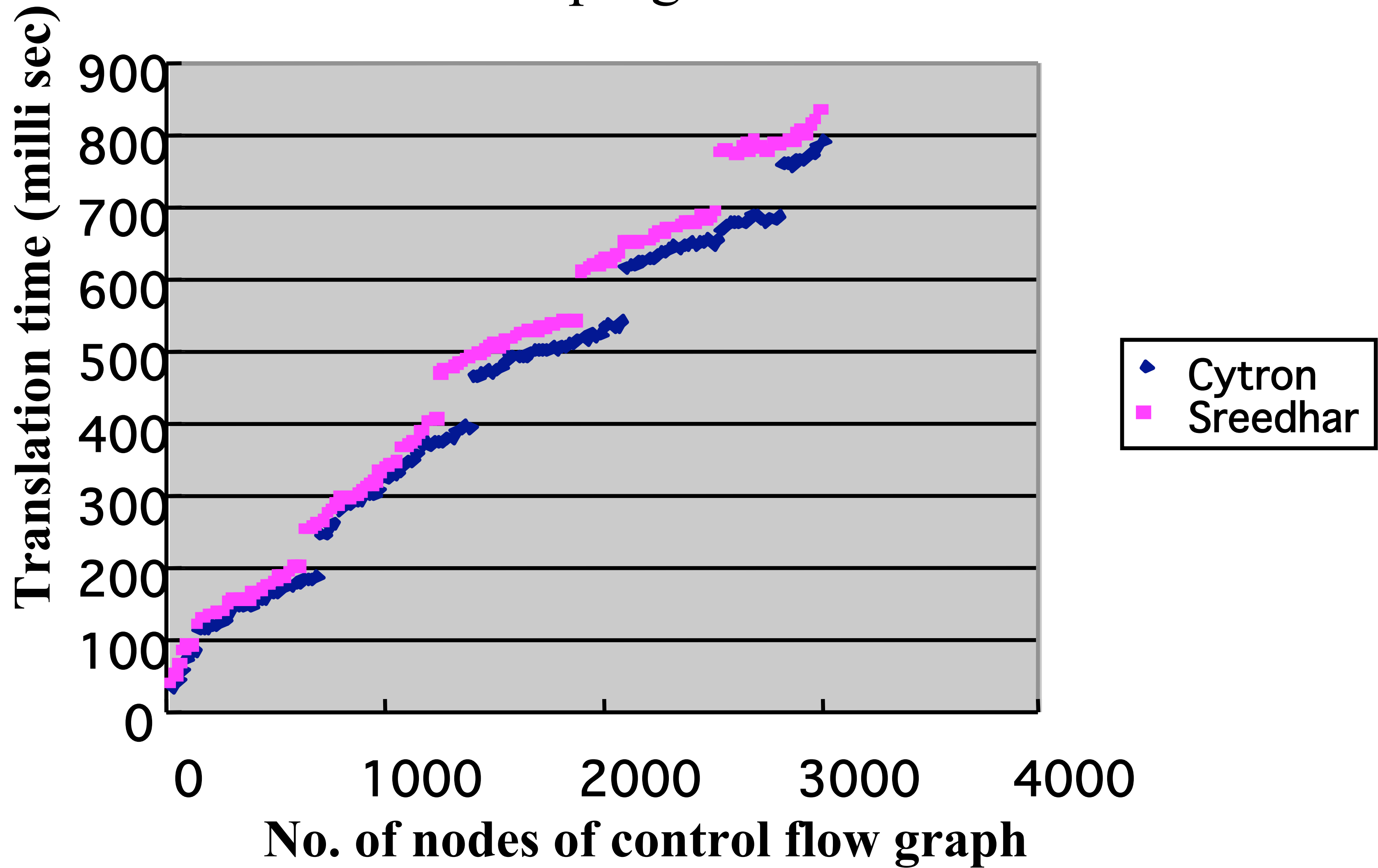
Translating into SSA form (SSA translation)



(a) Normal form

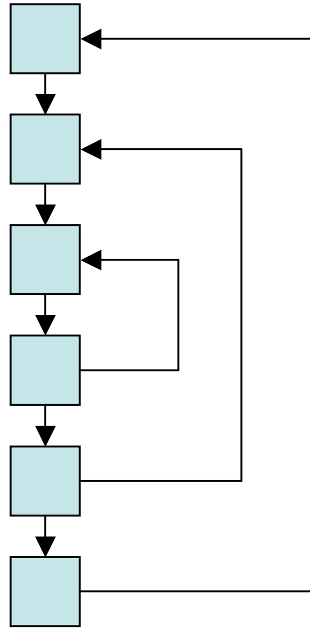


(b) SSA form

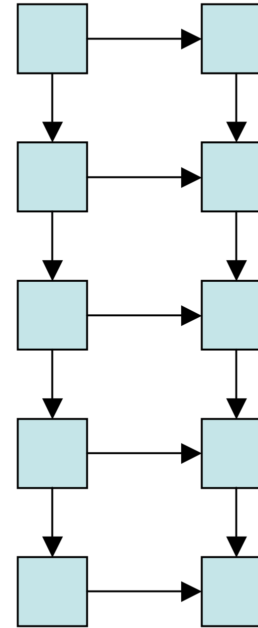


(The gap is due to the garbage collection)

Peculiar programs



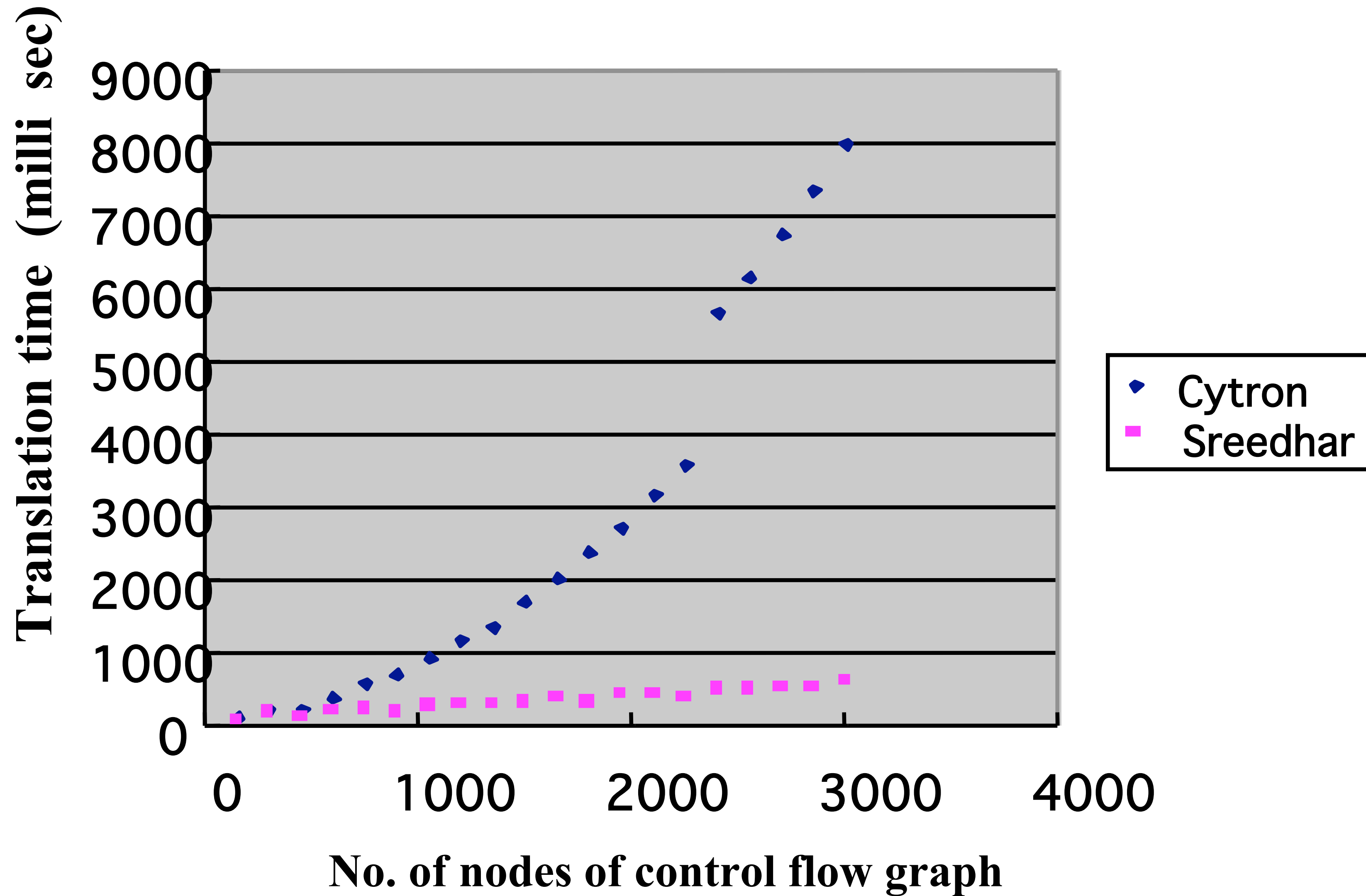
(a) nested loop

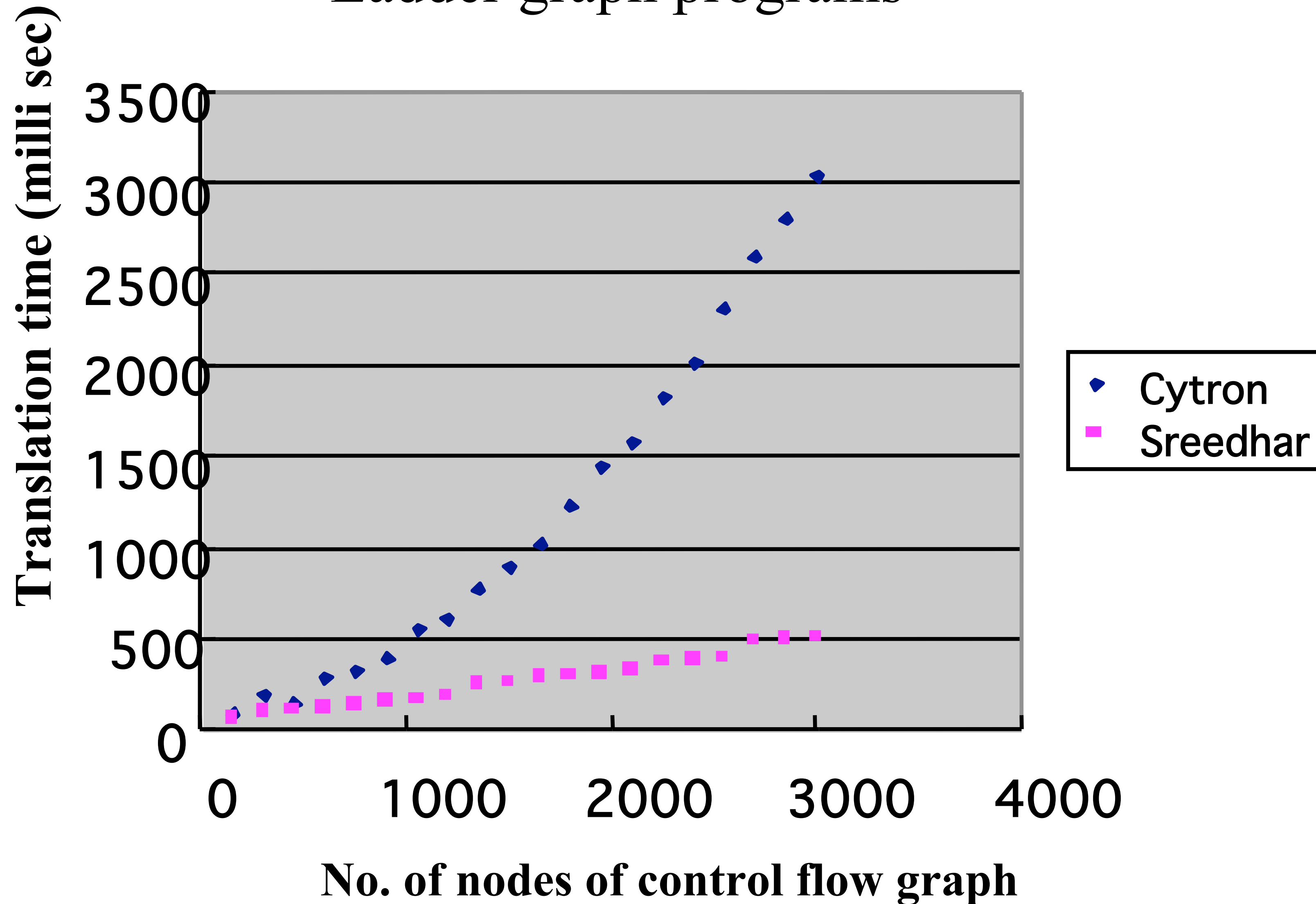


(b) ladder graph

Translation time

Nested loop programs





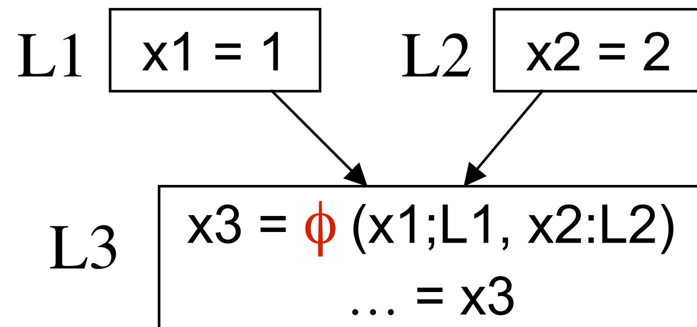
Translation time
Ladder graph programs
Translation time (milli sec)
3500
3000
2500
2000
1500
1000
500
0
0
1000
2000
3000
4000
No. of nodes of control flow graph
Cytron
Sreedhar

3. A comparison of two major algorithms for SSA back translation

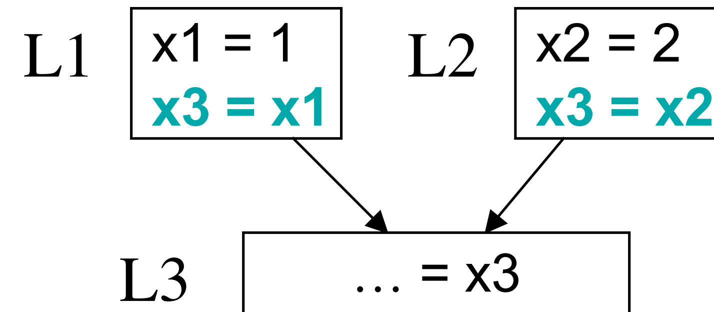
- Algorithm by Briggs [1998]
  Insert copy statements
- Algorithm by Sreedhar [1999]
  Eliminate interference

There have been no studies of comparison
Comparison made on COINS

Translating back from SSA form (SSA back translation)



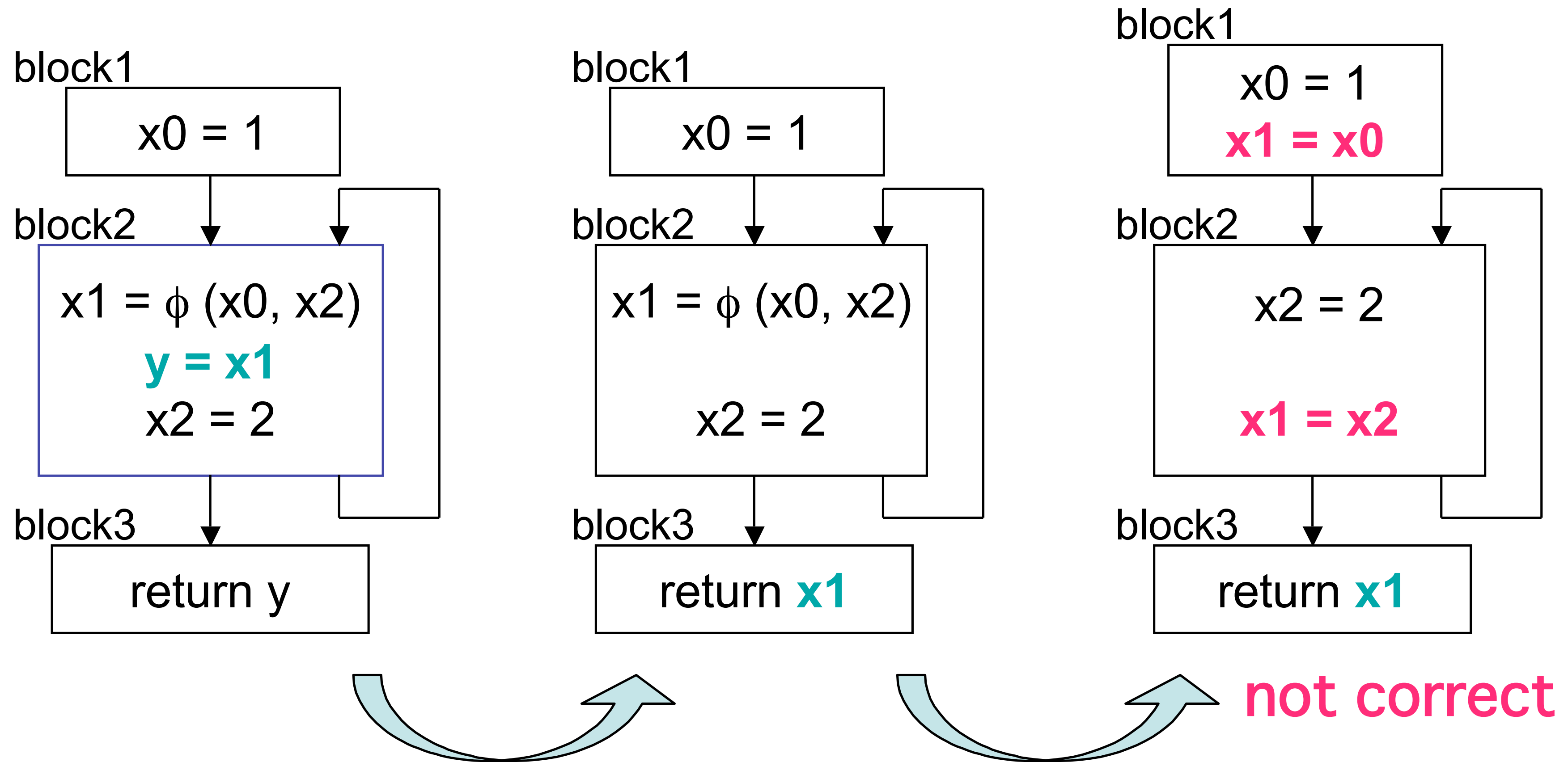
(a) SSA form



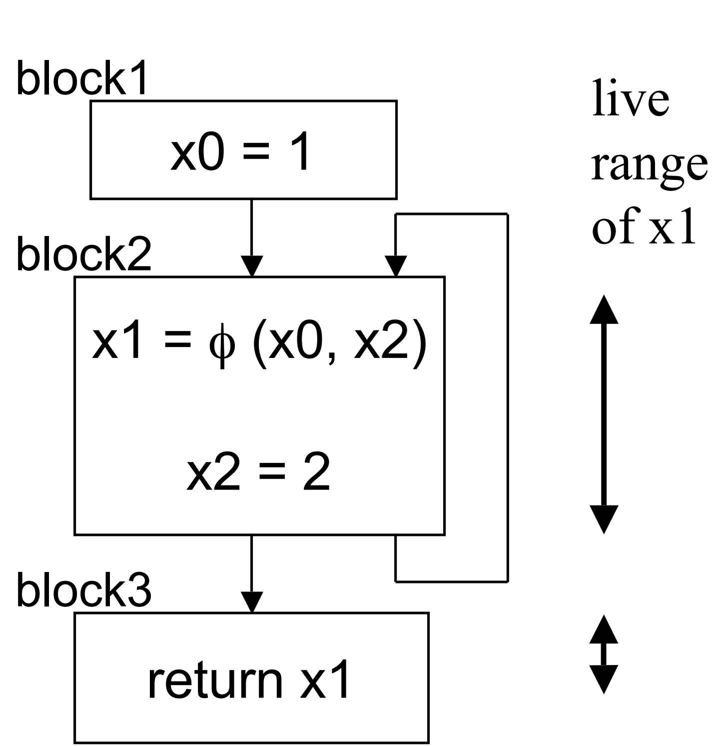
(b) Normal form

Problems of naïve SSA back translation

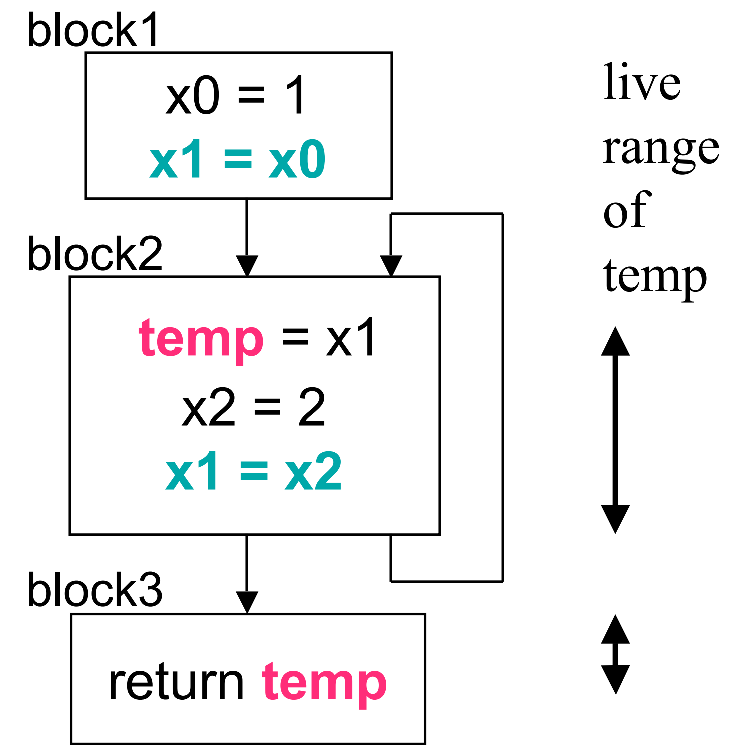
(lost copy problem)



To remedy these problems...
(i) SSA back translation algorithm by Briggs

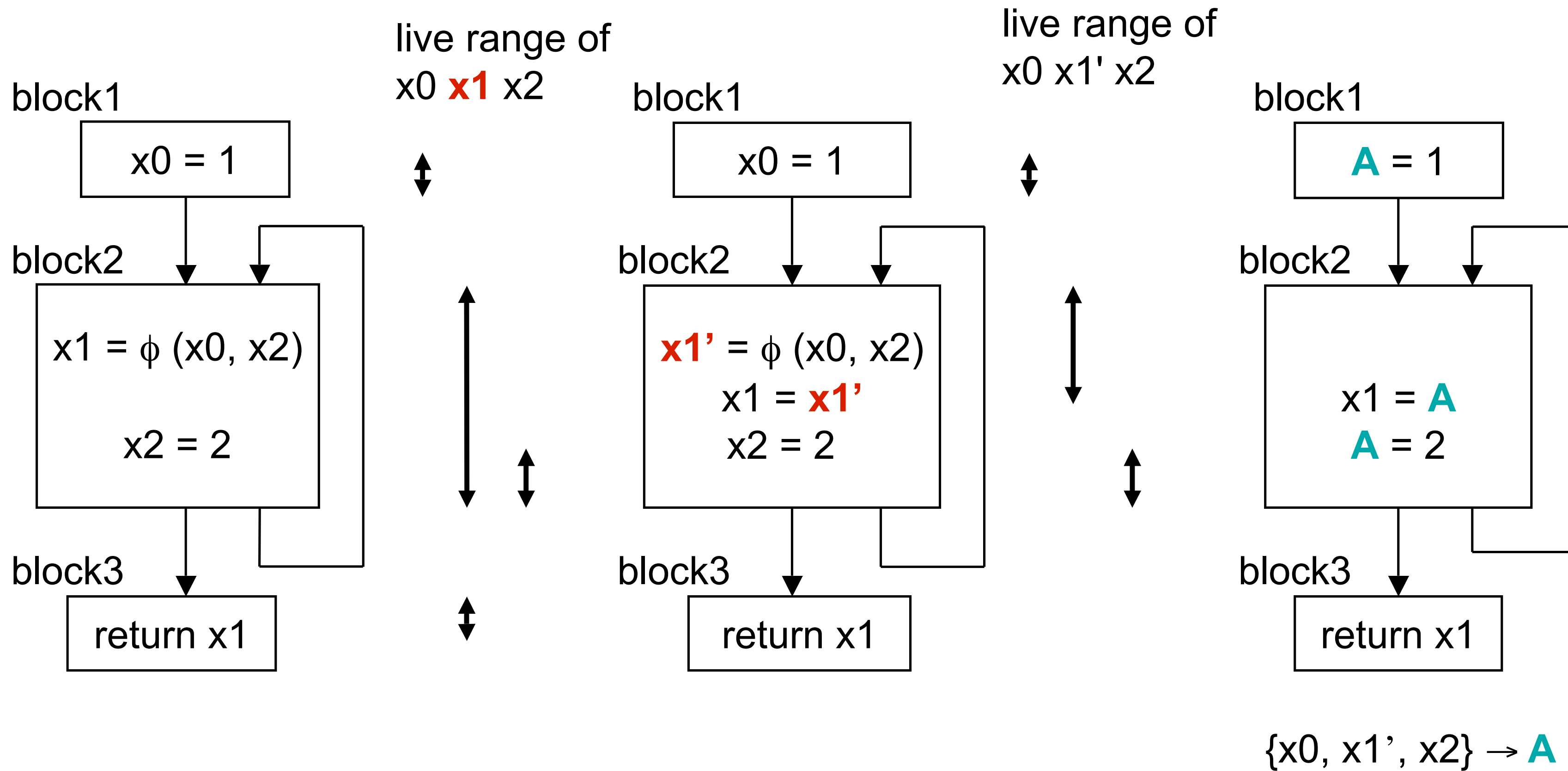


(a) SSA form



(b) normal form after back translation

(ii) SSA back translation algorithm by Sreedhar



{x0, x1', x2} → A

(a) SSA form

(b) eliminating interference

(c) normal form after back translation

Empirical comparison of SSA back translation

No. of copies (no. of copies in loops)

	SSA form	Briggs	Briggs + Coalescing	Sreedhar
Lost copy	0	3	1 (1)	1 (1)
Simple ordering	0	5	2 (2)	2 (2)
Swap	0	7	5 (5)	3 (3)
Swap-lost	0	10	7 (7)	4 (4)
do	0	9	6 (4)	4 (2)
fib	0	4	0 (0)	0 (0)
GCD	0	9	5 (2)	5 (2)
Selection Sort	0	9	0 (0)	0 (0)
Hige Swap	0	8	3 (3)	4 (4)

4. Examples of SSA form optimization

Conditional constant propagation

Example source program

```
/* from Appel, A.: Modern compiler implementation in Java, 2nd ed.,
   2002, Fig. 19.4 */
int main() {
  int i, j, k;
  i = 1;
  j = 1;
  k = 0;
  while (k < 100) {
    if (j < 20) {
      j = i;
      k = k + 1;
    } else {
      j = k;
      k = k + 2;
    }
  }
  printf("%d¥n",j);
}
```

Conditional constant propagation

By carefully analyzing the source program, we know that ‘j’ is always 1.

```
i = 1;
j = 1;
k = 0;
while (k < 100) {
  if (j < 20) {
    j = i;
    k = k + 1;
  } else {
    j = k;
    k = k + 2;
  }
}
printf("%d¥n",j);
```

after conditional constant propagation

```
k = 0;
L3:
if (k >= 100) go to L8;
k = k + 1;
goto L3;
L8:
printf("%d¥n",1);
```

(it is actually in SSA fom intermediate representation, but shown here in C style without ϕ)

Conditional constant propagation

By carefully analyzing the source program, we know that ‘j’ is always 1.

```
i = 1;
j = 1;
k = 0;
while (k < 100) {
  if (j < 20) {
    j = i;
    k = k + 1;
  } else {
    j = k;
    k = k + 2;
  }
}
printf("%d¥n",j);
```

(same as before)

```
k = 0;
while (k < 100) {
  k = k + 1;
}
printf("%d¥n",1);
```

(in structured program style)

We see that the else part of the while statement is deleted.

Operator strength reduction of loop induction variables and linear function test replacement

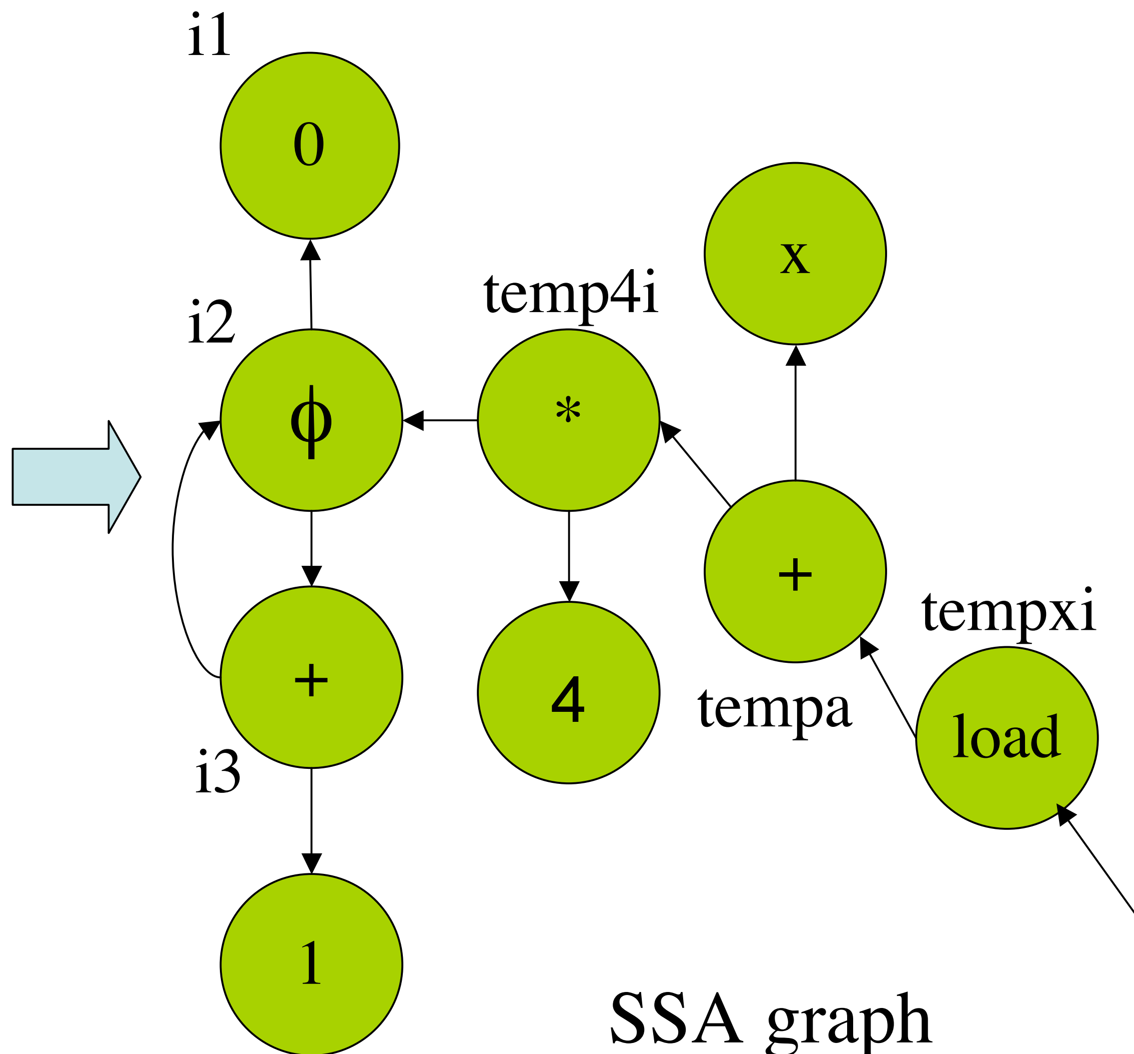
Example source program

```
/* addvectosr.c -- add vector for osr example */
void addvect(int z[], int x[], int y[], int n) {
  int i;
  for (i = 0; i < n; i++) {
     z[i] = x[i] + y[i];
  }
}
```

Operator strength reduction of loop induction variables and linear function test replacement

```
  i1 = 0
L1:
  i2 = φ(i1, i3)
  if (i2 >= n1) goto L6
   temp4i = 4 * i2
   tempa = x + temp4i
   tempxi = * tempa
   ...
   i3 = i2 + 1
   goto L1
L6:
```

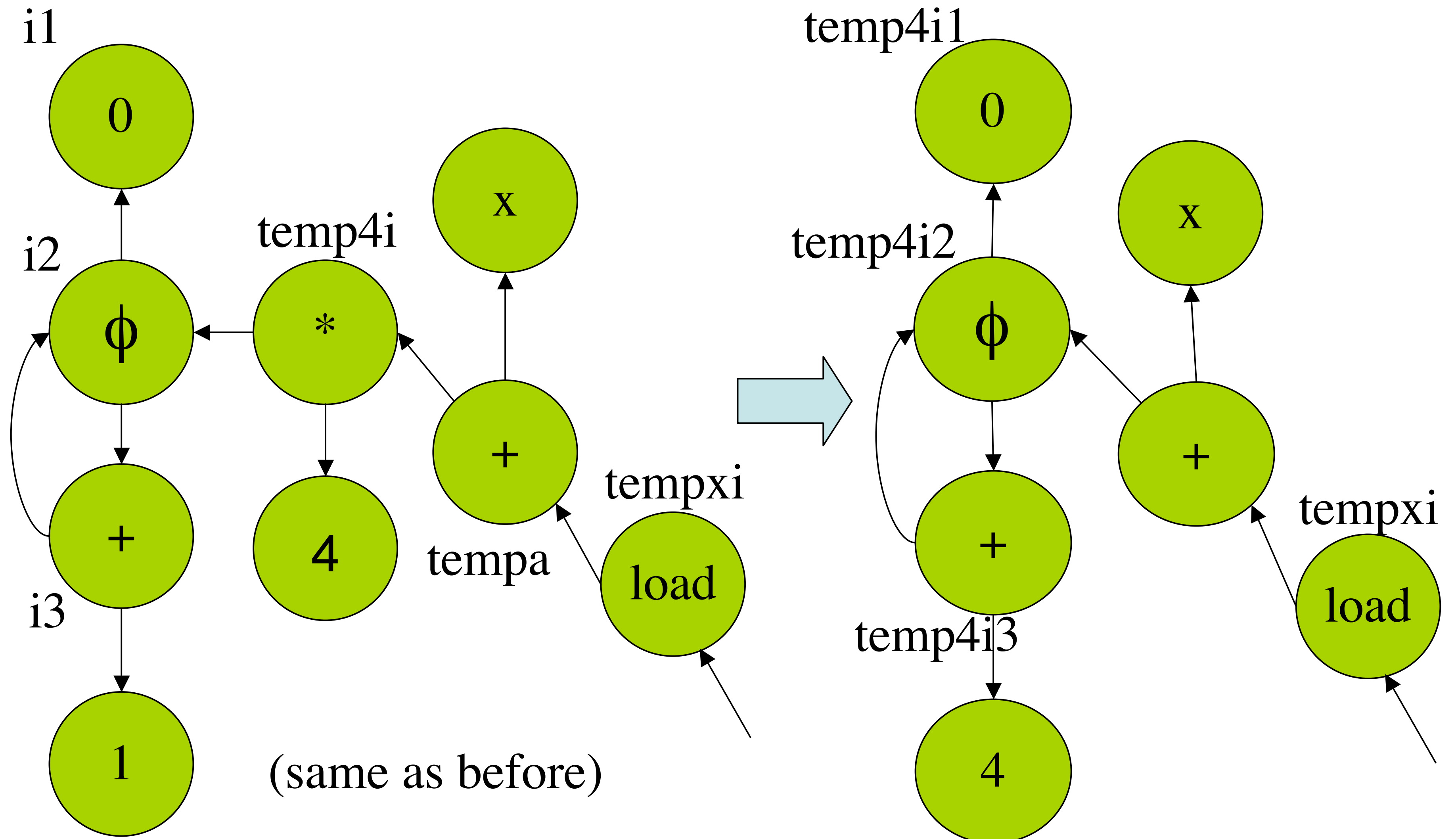
SSA form intermediate representation shown in C style



SSA graph

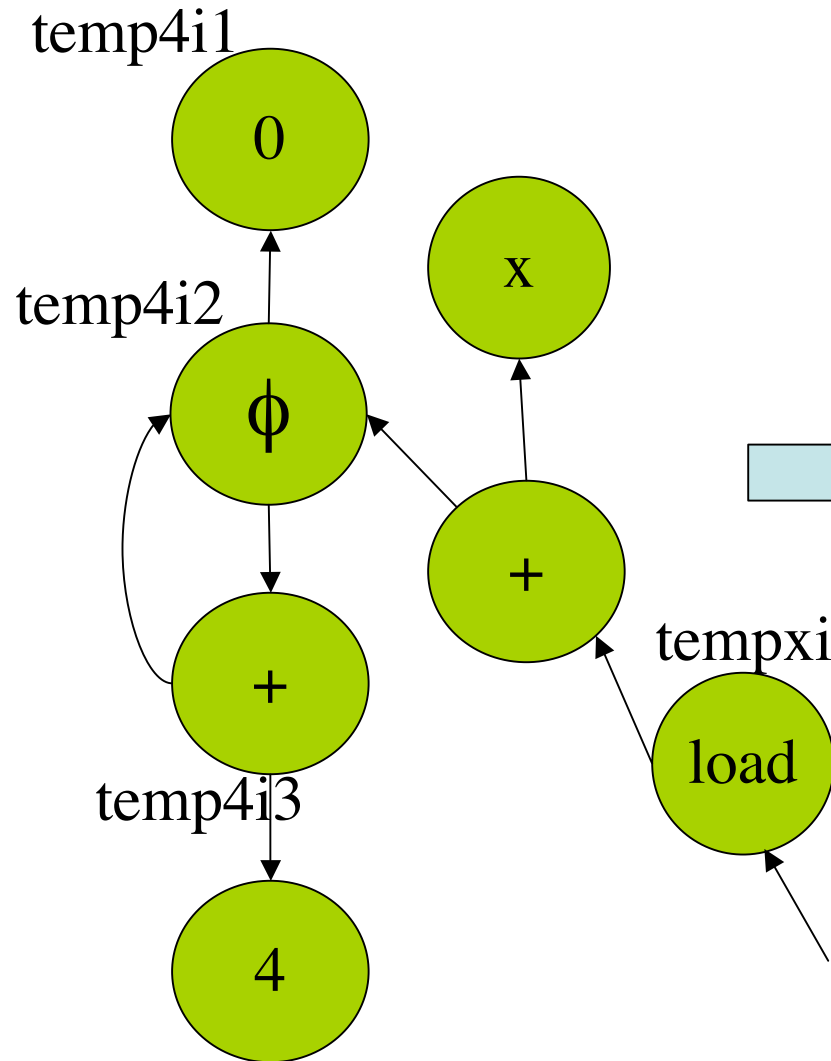
('load' corresponds to prefix '*' in C)

Operator strength reduction of loop induction variables and linear function test replacement



transformation of SSA graph - strength reduction

Operator strength reduction of loop induction variables and linear function test replacement



SSA graph (same as before)

```
  temp4i1 = 0
L1:
  temp4i2 = ϕ(temp4i1, temp4i3)
  ...
  tempxi = * (x + temp4i2)
  ...
  temp4i3 = temp4i2 + 4
  ...
```

SSA form

reconstruct SSA form

Operator strength reduction of loop induction variables and linear function test replacement

Example source program

```
void addvect (int z[],
int x[], int y[], int n) {
  int i;
  for (i = 0; i < n; i++) {
     z[i] = x[i] + y[i];
  }
}
```

after strength reduction of ind var

```
  ...
  temp4n = n << 2;
  ...
  temp4i = 0;
  i = 0;
  if (i >= n) goto L6;
L4:
  tempxi = * (x + temp4i);
  tempyi = * (y + temp4i);
  * (z + temp4i) = tempxi +
                     tempyi;
 i = i + 1;
 temp4i = temp4i + 4;
 if (temp4i < temp4n) goto L4;
L6:
```

In array accesses, 'multiply by 4' disappears. (Actually it is in SSA form intermediate representation, but shown here in C style without ϕ.)

Operator strength reduction of loop induction variables and linear function test replacement

after strength reduction of ind var

```
 ...
 temp4n = n << 2;
 ...
 temp4i = 0;
 i = 0;
 if (i >= n) goto L6;
L4:
 tempxi = * (x + temp4i);
 tempyi = * (y + temp4i);
 * (z + temp4i) = tempxi +
                  tempyi;
 i = i + 1;
 temp4i = temp4i + 4;
 if (temp4i < temp4n) goto L4;
L6:
```

(same as before)

after all optimization

```
 ...
 temp4n = n << 2;
 if (0 >= n) goto L6;
 temp4i = 0;
L4:
 tempxi = * (x + temp4i);
 tempyi = * (y + temp4i);
 * (z + temp4i) = tempxi +
                  tempyi;
 temp4i = temp4i + 4;
 if (temp4i < temp4n) goto L4;
L6:
```

Test of induction variable 'i' is replaced by that of 'temp4i', and 'i' disappears.

Common subexpression elimination

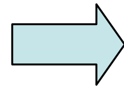
Example source program

```
/* cse_test.c */
...
a = 1;
b = 2;
x = 3;
y = a + b;
c = x + y;
if (a < 10)
    z = a + b;
else
    z = a + b;
d = x + z;
printf("%d¥n",d);
```

Common subexpression elimination

source program

```
a = 1;
b = 2;
x = 3;
y = a + b;
c = x + y;
if (a < 10)
    z = a + b;
else
    z = a + b;
d = x + z;
printf("%d¥n", d);
```



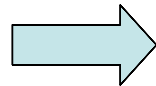
after analysis

```
a = 1;
b = 2;
x = 3;
y = a + b;
c = x + y;
if (a < 10)
  (z IS y) // a + b is a common subexpr
           // and z is equal to y.
else
  (z IS y) // a + b is a common subexpr
           //and z is equal to y.
(z IS y IS a+b) // therefore, z is equal to y
                // or a+b after if statement.
(d IS x+z) // d is equal to x+z
printf("%d¥n", d);
```

Common subexpression elimination

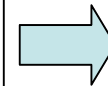
```
a = 1;
b = 2;
x = 3;
y = a + b;
c = x + y;
if (a < 10)
  (z IS y)
else
   (z IS y)
(z IS y IS a+b)
(d IS x+z)
printf("%d¥n", d);
```

after analysis
(same as before)



```
a = 1;
b = 2;
x = 3;
z = a + b;
printf("%d¥n",
         x+z);
```

after common
subexpr elim,
dead code elim,
and coalescing



```
printf("%d¥n", 6);
```

after all
optimizations

5. Preliminary experimental results

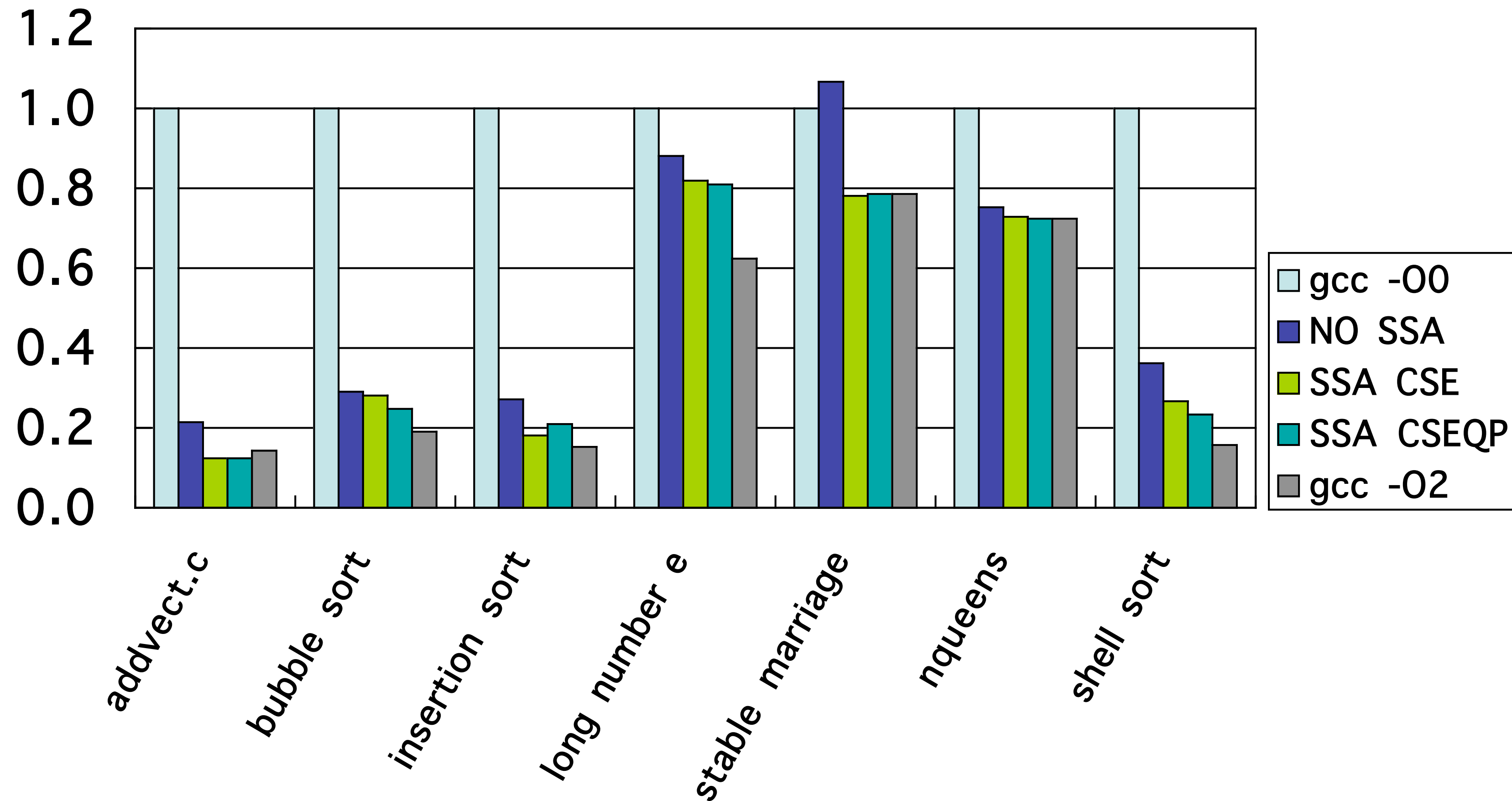
- Coins C-to-SPARC compiler
- Measured on Sun Blade 1000
  - Ultra SPARC III
  - Superscalar SPARC V9
  - 750 MHz * 2 CPU
  - L1 cache : 64 kB data, 32 kB instruction
  - memory: 1 GB
  - SunOS 5.8

Execution time of object code

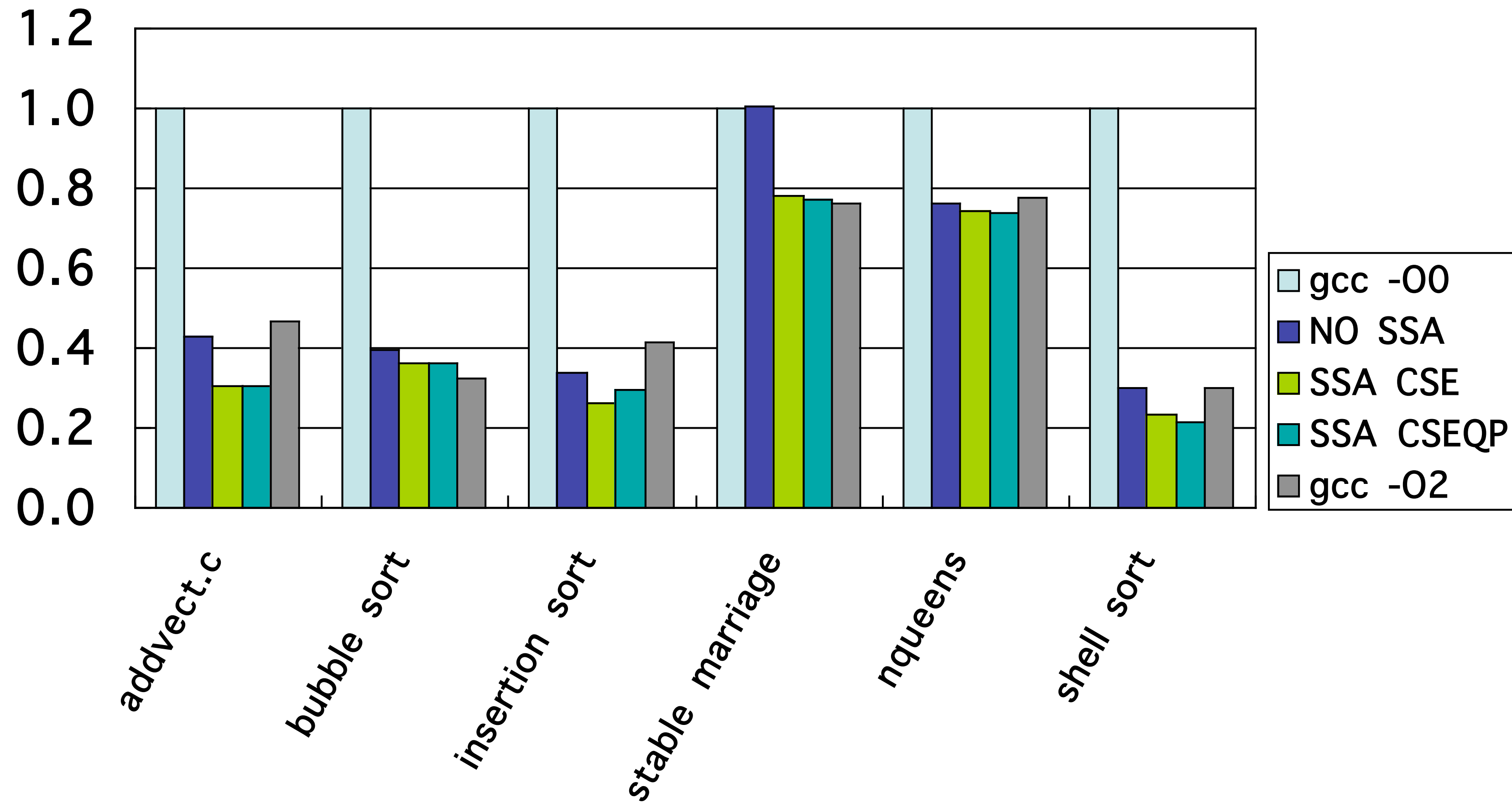
Legend

- gcc -O0
- NO SSA: Coins baseline compiler (without SSA optimization)
- SSA CSE: constant propagation, hoisting loop invariant, SSA graph, common subexpression elimination, copy propagation, constant propagation, dead code elimination, operator strength reduction, constant propagation, common subexpression elimination, redundant phi elimination, dead code elimination, empty block elimination, in this order (pruned SSA, Sreedhar's method 3 back translation, no memory alias analysis, loop transformation)
- SSA CSEQP: constant propagation, hoisting loop invariant, SSA graph, common subexpression elimination by EQP, copy propagation, constant propagation, dead code elimination, operator strength reduction, constant propagation, common subexpression elimination by EQP, redundant phi elimination, dead code elimination, empty block elimination, in this order (pruned SSA, Sreedhar's method 3 back translation and Chaitin's coalescing, loop transformation)
- gcc -O2

Execution time of object code (small integer programs)

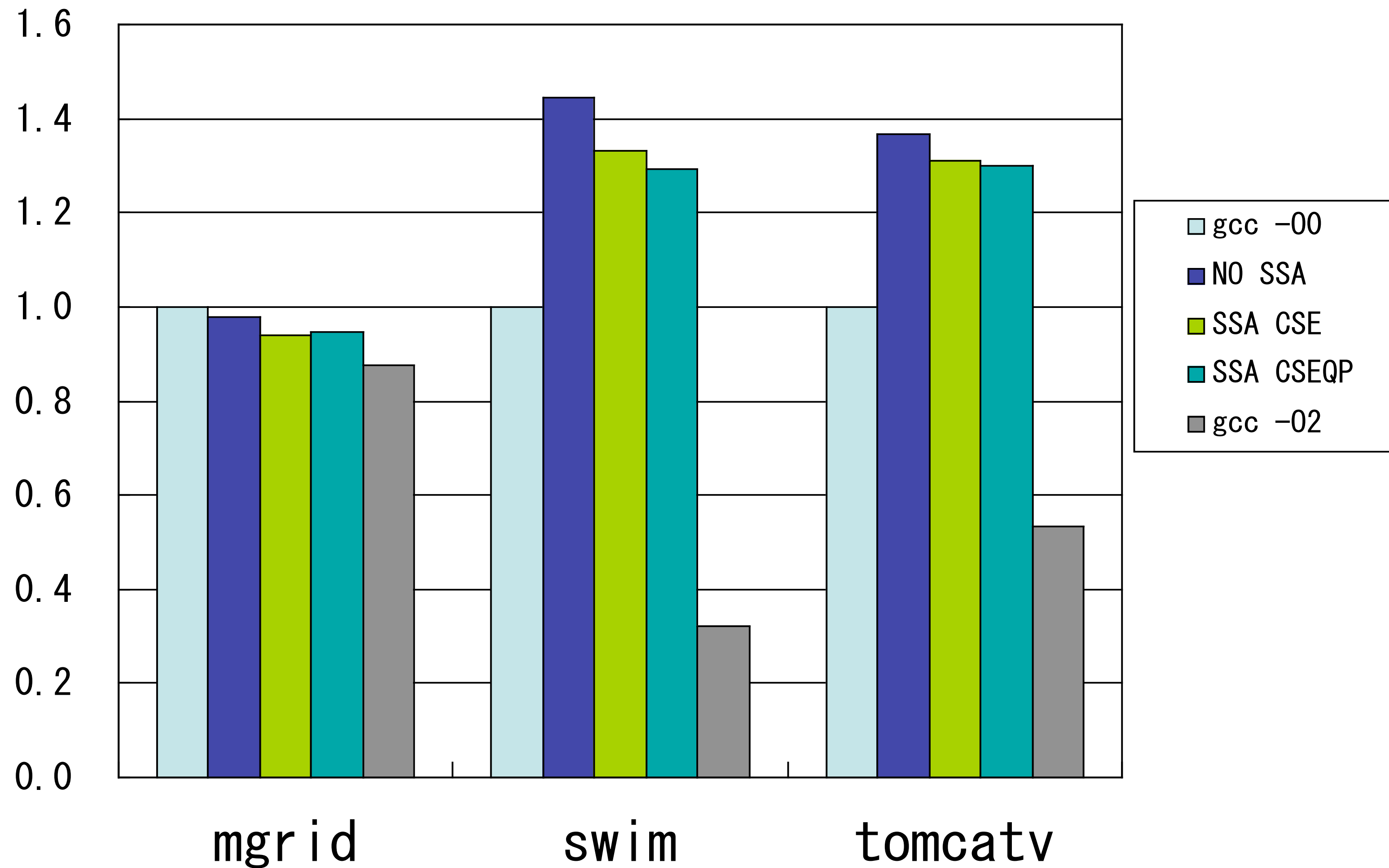


Execution time of object code (small floating point programs)

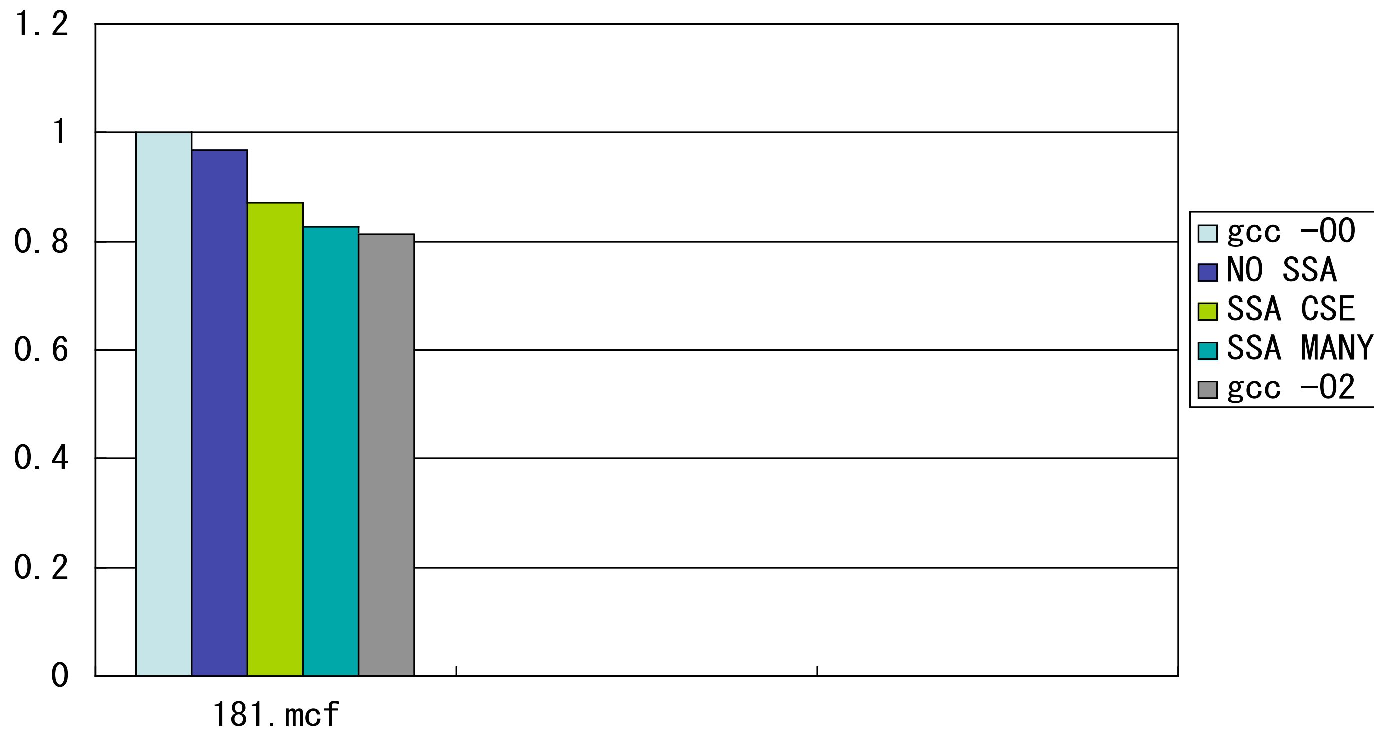


Execution time of object code

(SPEC 95 Fortran benchmarks rewritten to C)



Execution time of object code
(SPEC 2000 benchmark)



SSA CSE: hoist loop inv, common subexpression elimination

SSA MANY: hoist loop inv, const prop, com subexp elim, dead code elim, coalescing

Related work: SSA form in compiler infrastructures

- SUIF (Stanford Univ.): no SSA form
- machine SUIF (Harvard Univ.): only one optimization in SSA form
- Scale (Univ. Massachusetts): a couple of SSA form optimizations. But it generates only C programs, and cannot generate machine codes like in COINS.
- GCC: some attempts but experimental

Only COINS will have full support of SSA form as a compiler infrastructure

Summary

- SSA form module of the COINS infrastructure
- Empirical comparison of algorithms for SSA translation gave criterion to make a good choice
- Empirical comparison of algorithms for SSA back translation suggests Sreedhar et al.'s method is recommended.
- A rich set of SSA form optimization modules
- Preliminary experimental results

Hope COINS and its SSA module help the compiler writer to compare/evaluate/add optimization methods

the following slides are unnecessary

After Conditional Constant Propagation (cont.)

```
k = 0;
L3:
if (k >= 100) go to L8;
k = k + 1;
goto L3;
L8:
printf("%d¥n",1);
```

=

```
k = 0;
while (k < 100) {
  k = k + 1;
}
printf("%d¥n",1);
```

(in structure program style)

We see that the else part of the while statement is deleted.