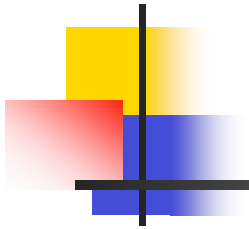
Comparison and Evaluation of Back Translation Algorithms for Static Single Assignment Form

Masataka Sassa[#], Masaki Kohama[+] and Yo Ito[#]

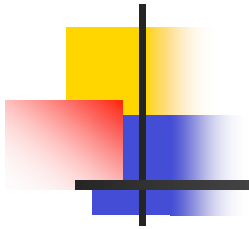
[#] Dept. of Mathematical and Computing Sciences,
Tokyo Institute of Technology

[+] Fuji Photo Film Co., Ltd



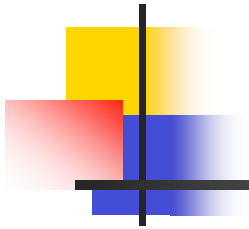
Background

- SSA form (static single assignment form)
  - Good representation for optimizing compilers
  - Cannot be trivially translated into object code such as machine code
- SSA back translation (translation to normal form)
  - Several algorithms exist, whose translation results differ
  - No research for their comparison and evaluation
  - Algorithm by Briggs et al. published earlier is often adopted without much consideration



Outline

- Comparison of SSA back translation algorithms
  - Algorithm by Briggs et al.
  - Algorithm by Sreedhar et al.
- A proposal for improving Briggs' algorithm
- Comparison by experiments
  - Changing the no. of registers and combination of optimizations
  - Execution time difference is not negligible (by 7%~10%)
- Give a criterion in selecting SSA back translation algorithm



Contents

1. SSA form
2. SSA back translation
3. Two major algorithms for SSA back translation
4. Improvement of Briggs' algorithm
5. Experimental Results
6. Conclusion

1 Static single assignment form (SSA form)

Static single assignment form (SSA form)

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

(a) Normal form

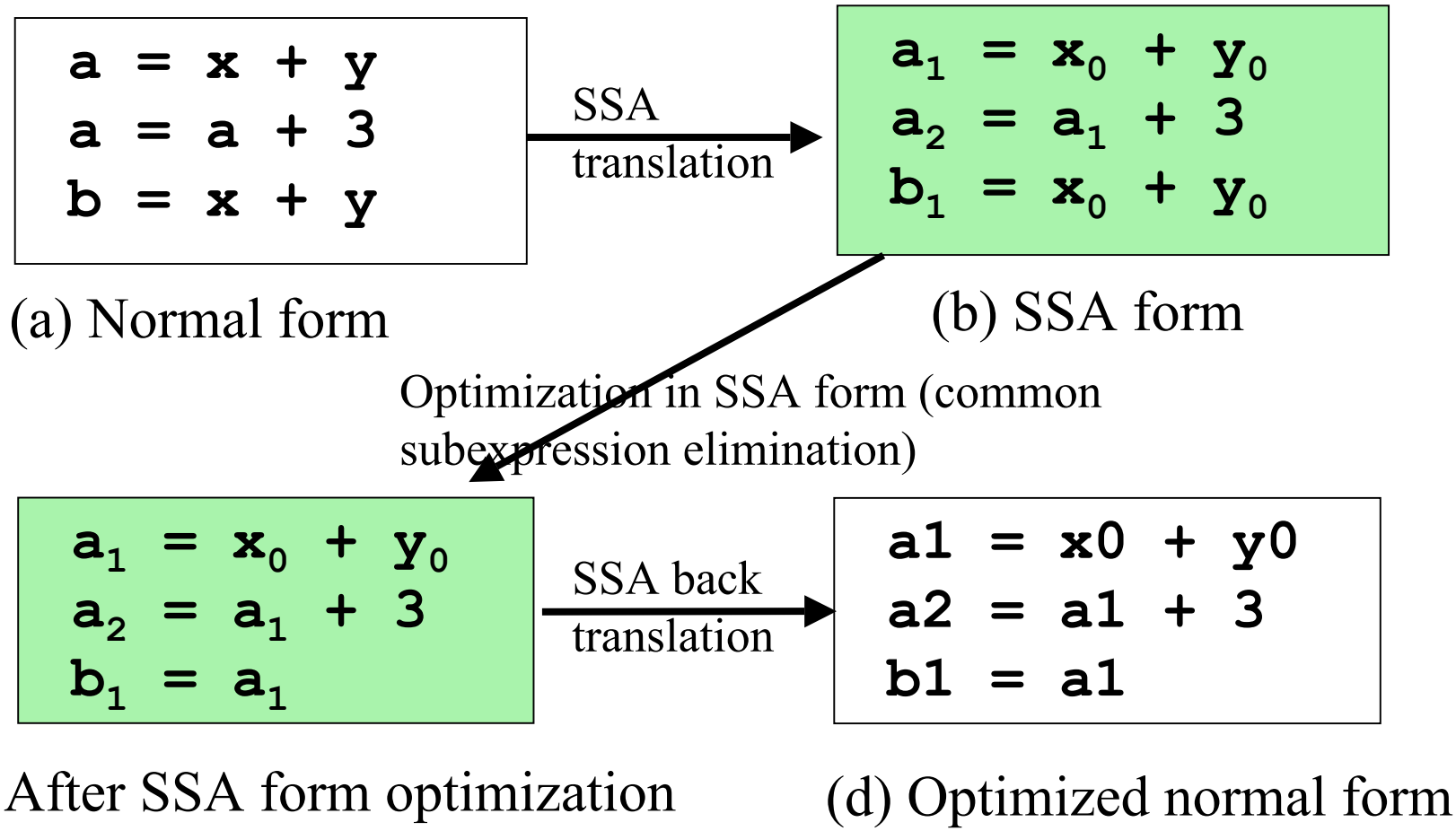
```
x1 = 1
y1 = 2
a1 = x1 + y1
a2 = a1 + 3
b1 = x1 + y1
```

(b) SSA form

Only one definition for each variable.

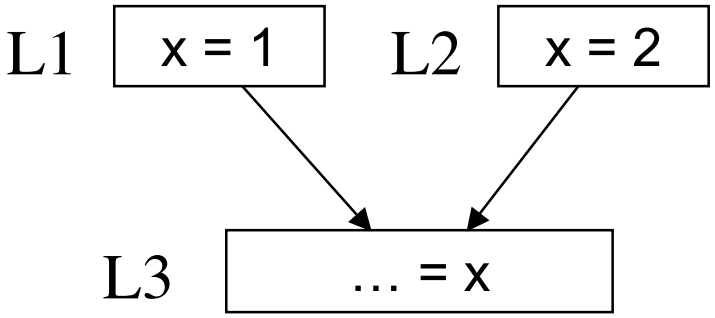
For each use of variable, only one definition reaches.

Optimization in static single assignment (SSA) form

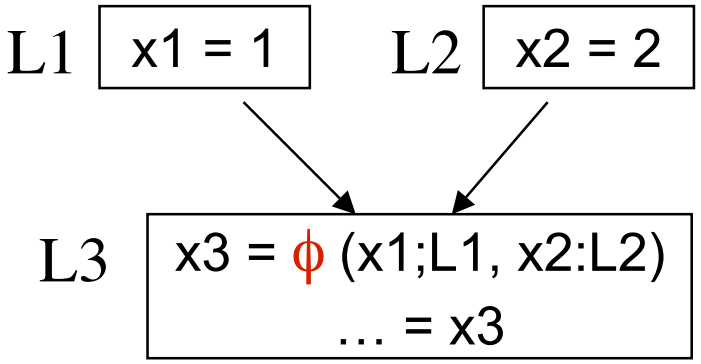


SSA form is becoming increasingly popular in compilers, since it is suited for clear handling of dataflow analysis (definition and use) and optimization.

Translation into SSA form (SSA translation)



(a) Normal form

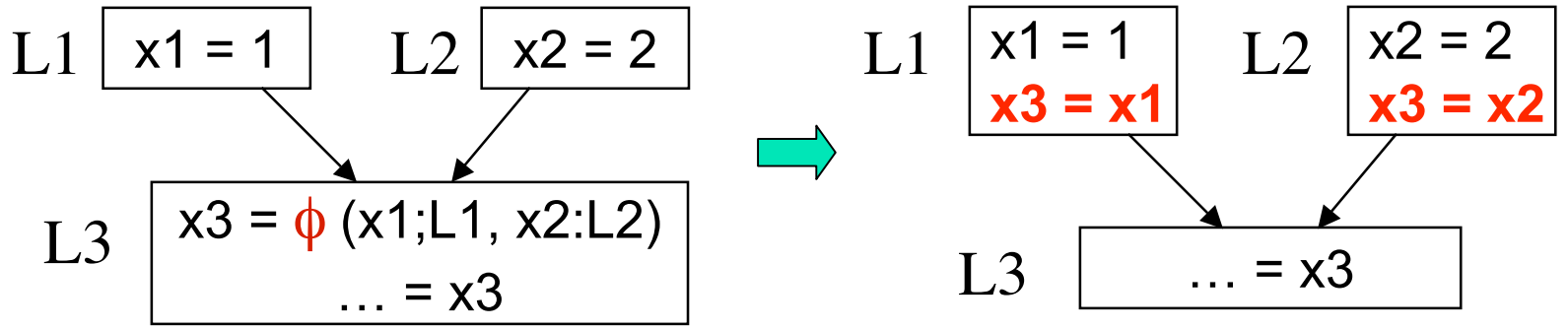


(b) SSA form

ϕ-function is a hypothetical function to make definition point unique

2 Back translation from SSA form into normal form (SSA back translation)

Back translation from SSA form into normal form (SSA back translation)



(a) SSA form

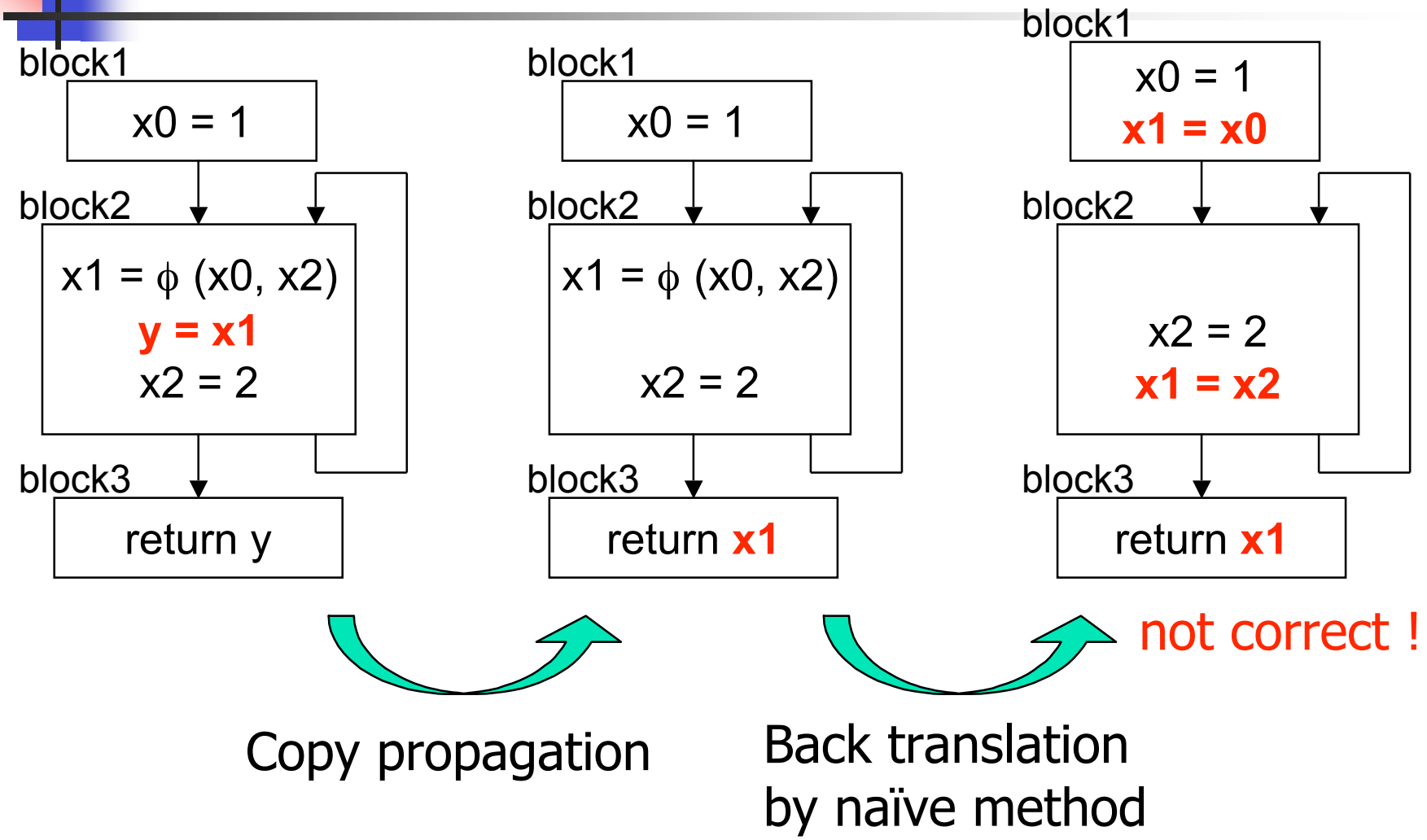
(b) Normal form

ϕ-function must be deleted before code generation.

Insert copy statements in the predecessor blocks of ϕ-function and delete ϕ.

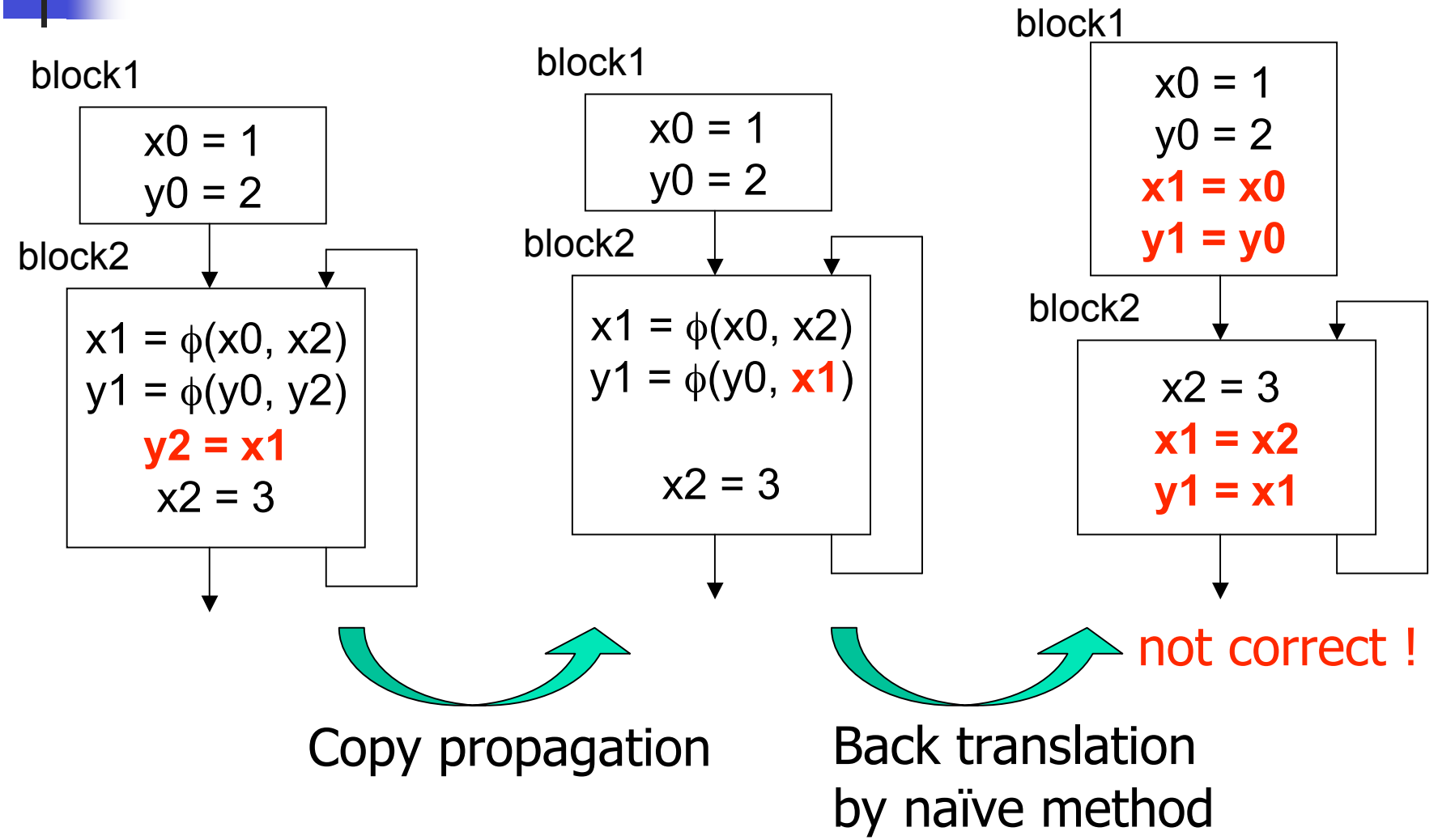
Problems of naïve SSA back translation

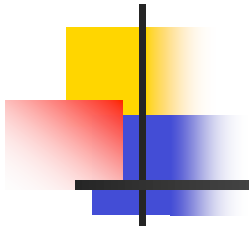
(i) Lost copy problem



Problems of naïve SSA back translation

(ii) Simple ordering problem:
simultaneous assignments to ϕ-functions





3 Two major algorithms for SSA back translation

To remedy these problems...

SSA back translation algorithms by

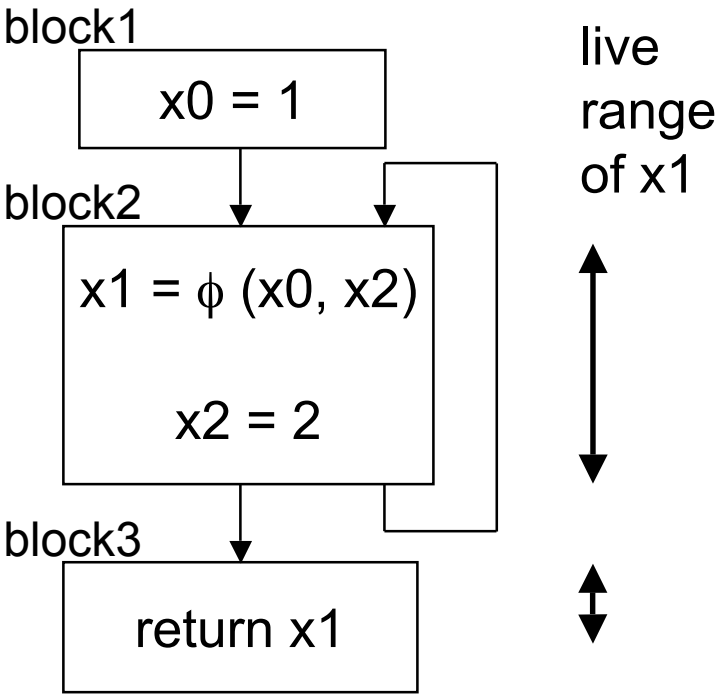
(i) Briggs et al. [1998]

Insert copy statements

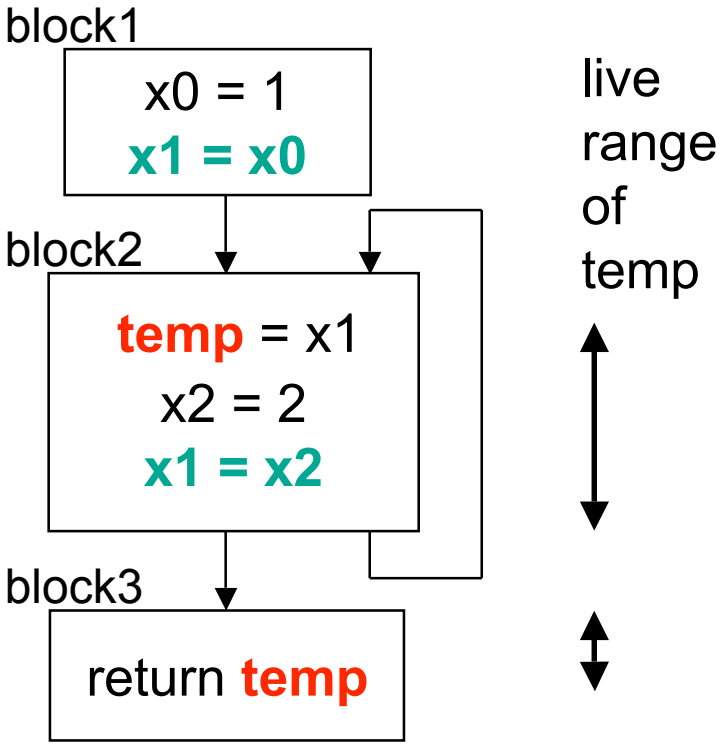
(ii) Sreedhar et al. [1999]

Eliminate interference

(i) SSA back translation algorithm by Briggs (lost copy problem)



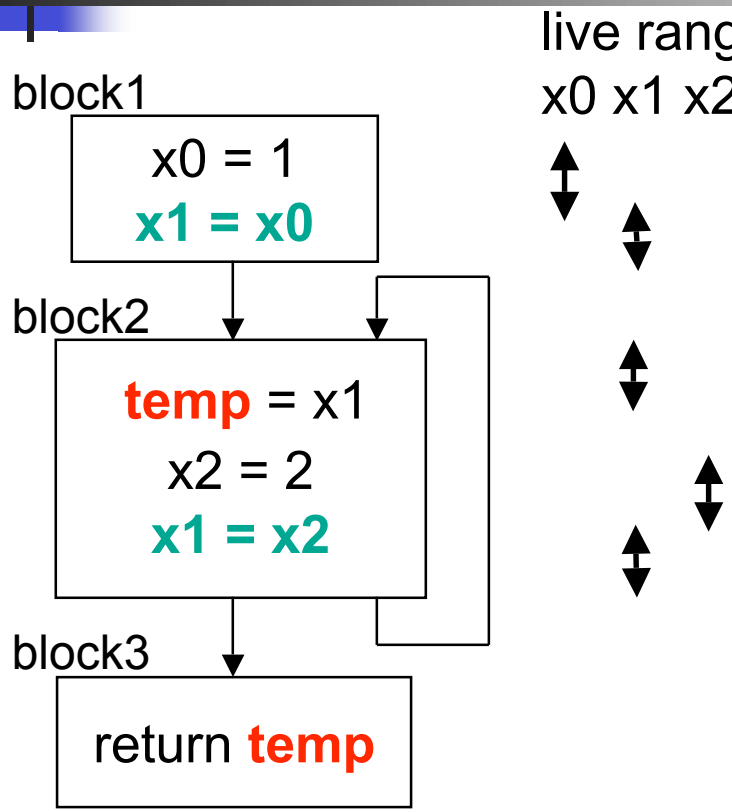
(a) SSA form



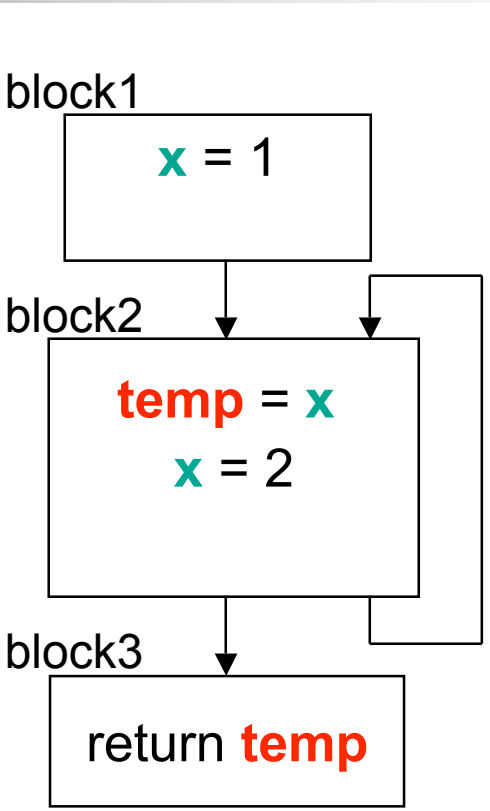
(b) normal form after back translation

Many copies inserted. Is is OK?

(i) SSA back translation algorithm by Briggs (cont)



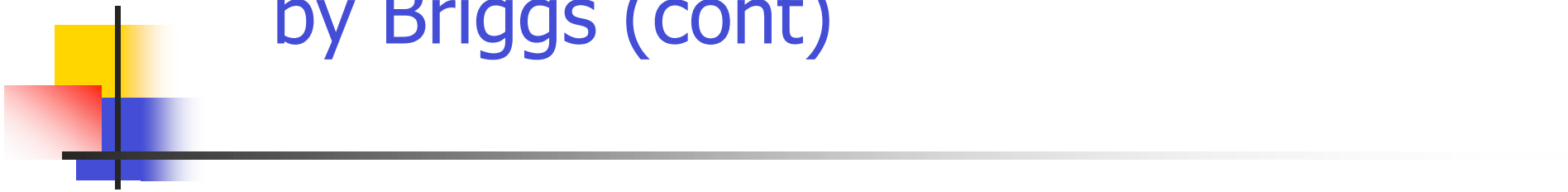
live range of
x0 x1 x2



(c) normal form after back translation (same as (b))

coalesce {x0, x1, x2} → x
(delete relevant copies)
(d) after coalescing

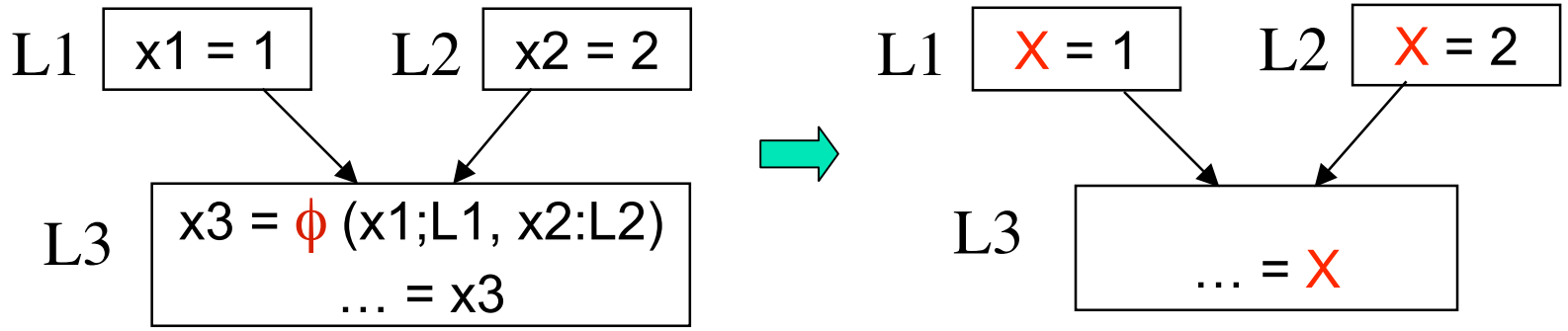
(i) SSA back translation algorithm by Briggs (cont)



Possible problems:

- Many copies are inserted.
  - They claim most copies can be coalesced.
  - Actually there are many copies which interfere, thus cannot be coalesced.
  - This increases register pressure (demand for registers).
- Problems when processing ϕ-functions within loops
  - Causes copies that cannot be coalesced
  - We propose an improvement

(ii) SSA back translation algorithm by Sreedhar - principle

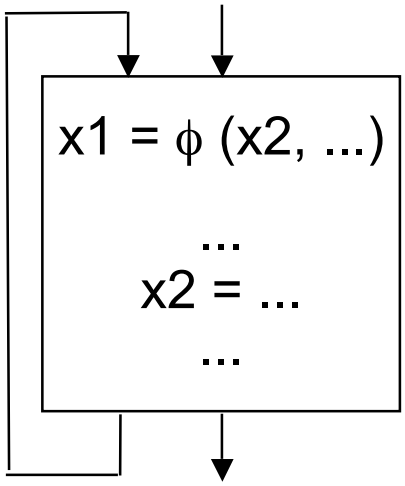


(a) SSA form

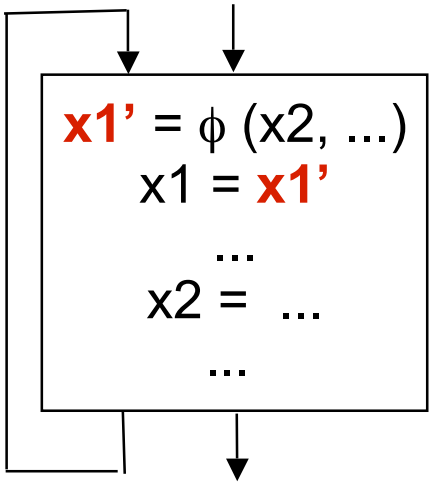
(b) Normal form

If the live ranges of variables in ϕ-function (x3, x1, x2) do not interfere, then coalesce them into a single variable (X), and delete the ϕ-function.

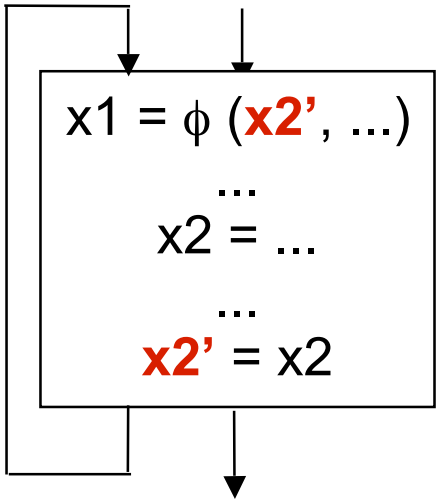
(ii) SSA back translation algorithm by Sreedhar - rewriting



(a) SSA form



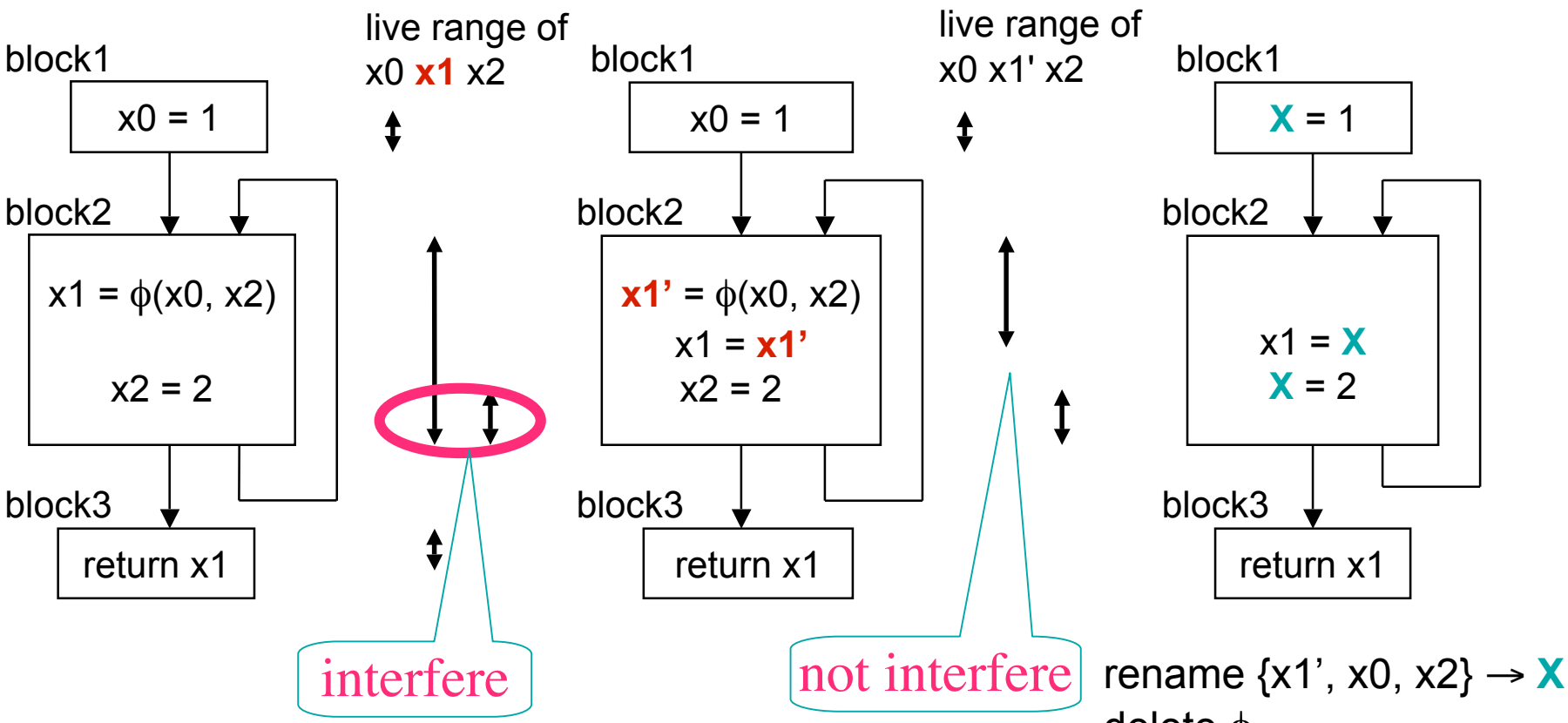
(b) rewrite target



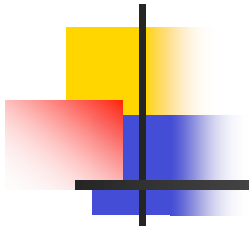
(c) rewrite parameter

If variables in ϕ-function interfere, rewrite the target or the parameter.

(ii) SSA back translation algorithm by Sreedhar (lost copy problem)



(ii) SSA back translation algorithm by Sreedhar (cont)



Benefit:

- Copies are few

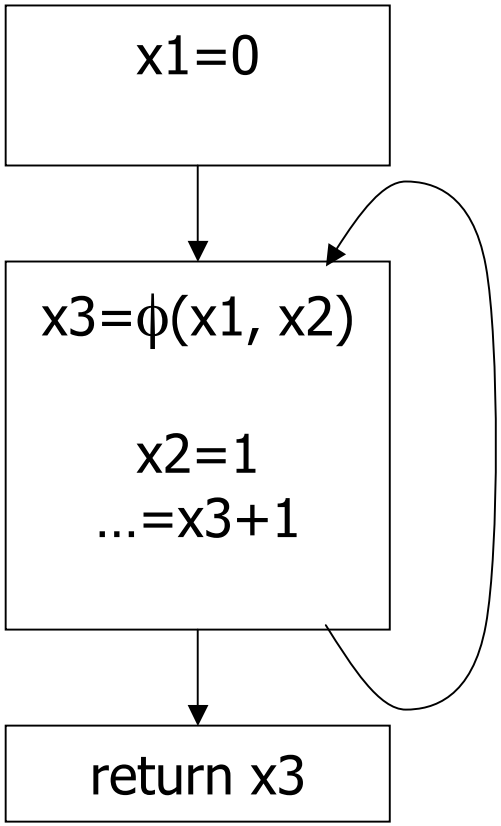
Possible problems:

- Live range of variables are long
- May increase register pressure

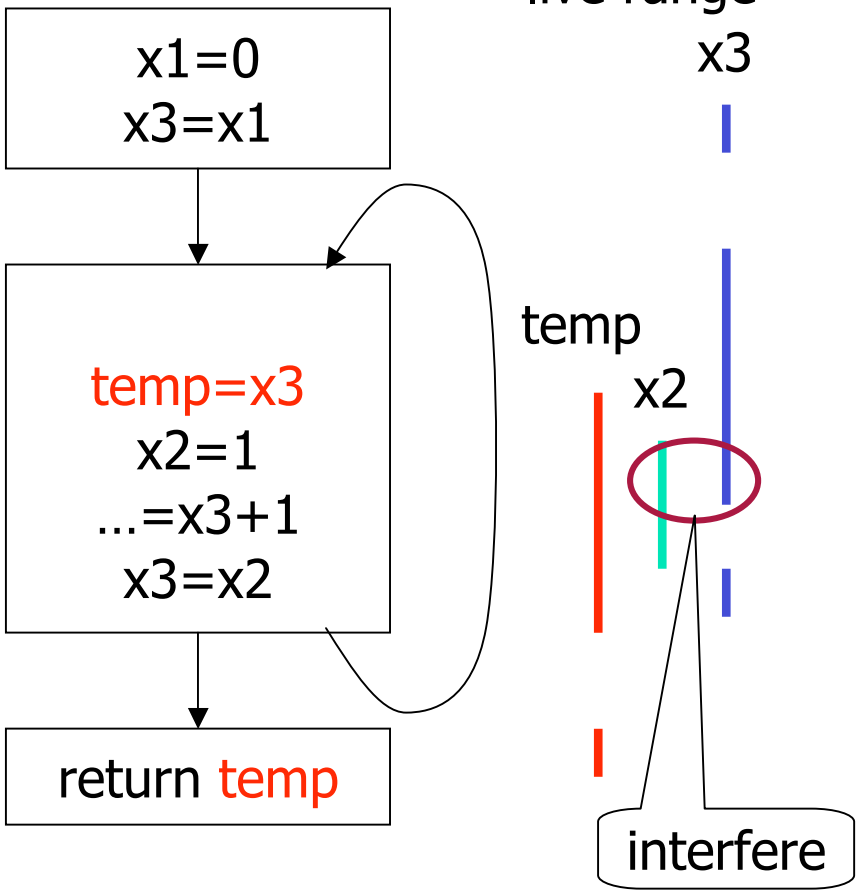
4 Problems of Briggs' algorithm and proposal for improvement

Problems of Briggs' algorithm - long live range

(ϕ-function in loop -> insert copies -> copy "x3=x2" is inserted in the live range of x3 -> need temporary temp -> copies interfere -> copies cannot be coalesced)



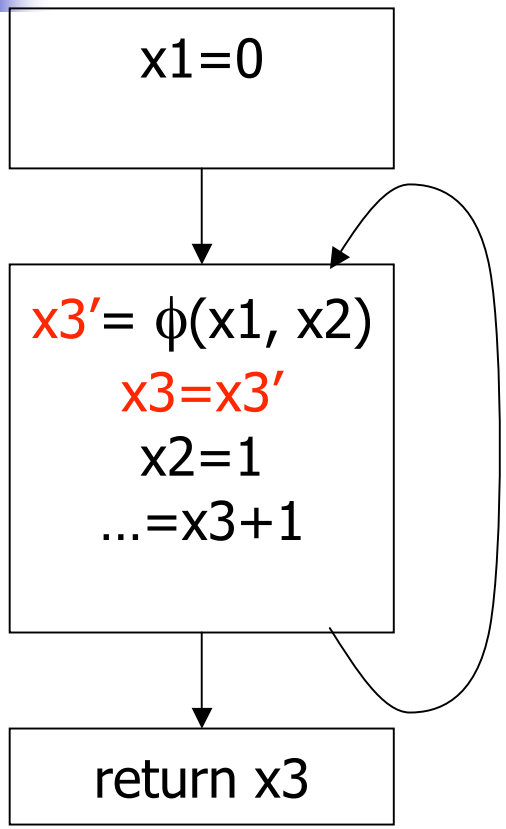
(a) SSA form



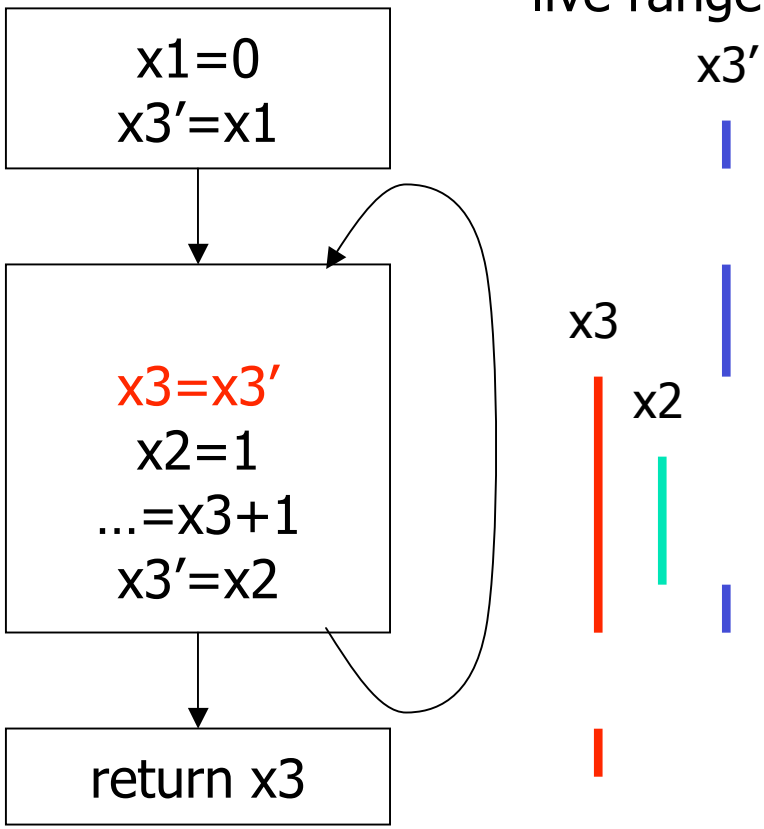
(b) Briggs' back translation

Improvement of Briggs' algorithm (1)

(first rewrite ϕ-function à la Sreedhar's method -> remove interference between x2 and x3 -> back translation -> x2 and x3' do not interfere and can be coalesced -> better than Briggs')



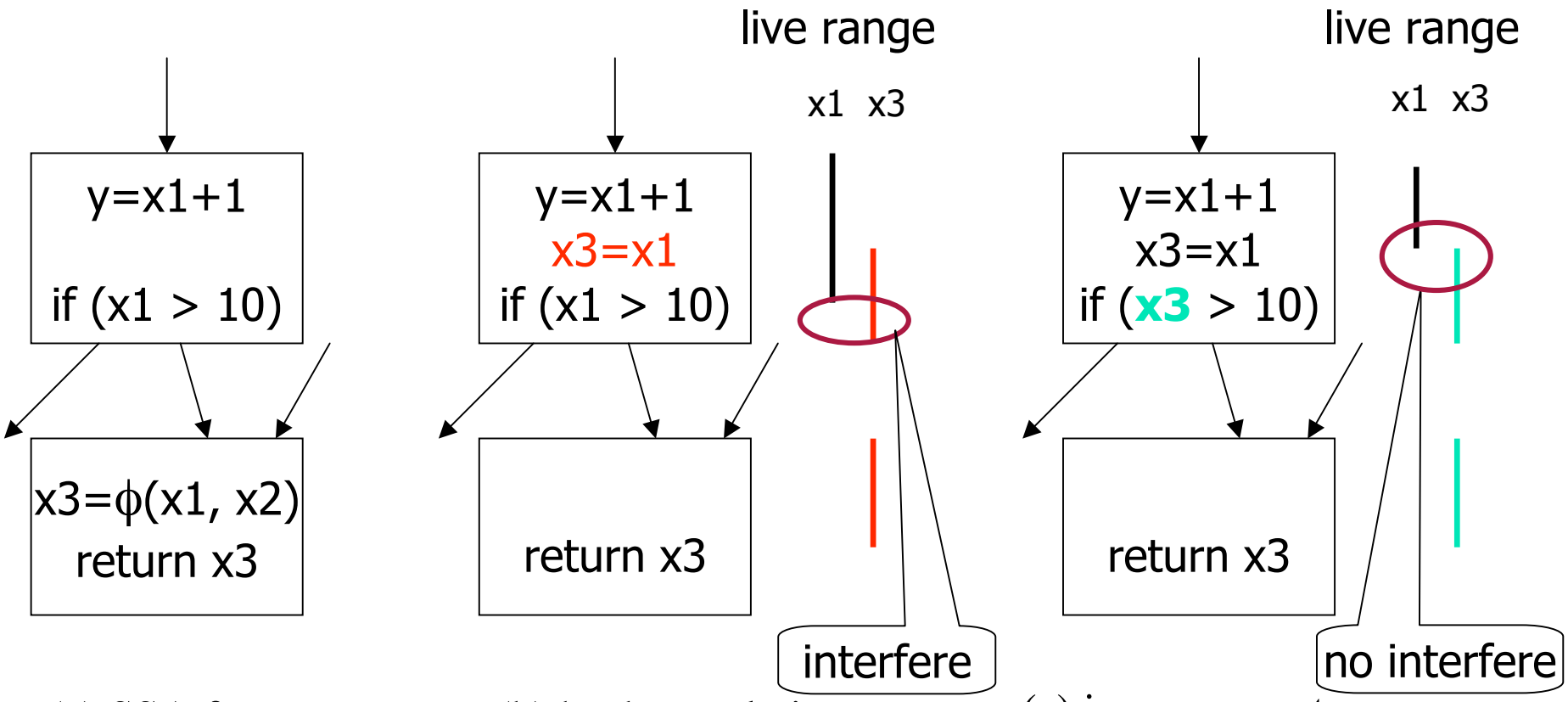
(c) rewrite ϕ



(x2 and x3' can be coalesced)

(d) after back translation

Improvement of Briggs' algorithm (2) - conditional branch

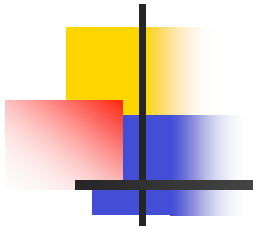


(a) SSA form

(b) back translation by Briggs

(c) improvement (x1 and x3 can be coalesced)

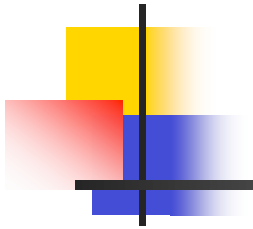
Implementation



- COINS compiler infrastructure
  - Adopt Sreedhar's SSA back translation algorithm
  - Iterated coalescing [George & Appel et al. 1996]
  - Implemented in Java
- SSA optimization module of COINS
  - Briggs's algorithm and the proposed improvement are additionally implemented for measurement
  - Implemented in Java

5 Experimental results using SPEC benchmarks

Consideration up to now



- Briggs's algorithm
  - many interfering copies
    - enough no. of registers -> disadvantageous
    - small no. of registers
      - register pressure increases due to interfering copies
- Sreedhar's algorithm
  - interfering copies are few, but live ranges of variables are long
    - enough no. of registers -> advantageous
    - small no. of registers
      - register pressure increases due to long live ranges of variables

Experiments

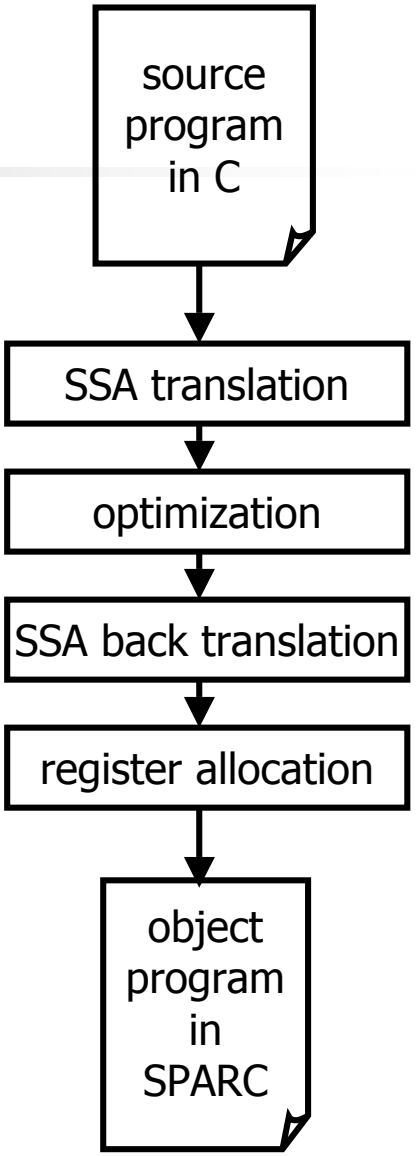
- Sun-Blade-1000

architecture	SPARC V9
processor	UltraSPARC-III 750MHz x 2
L1 cache	64KB(data), 32KB(instruction)
L2 cache	8MB
Memory	1GB
OS	SunOS 5.8

- Benchmark (C language)
  - SPECint2000 mcf, gzip-1.2.4 (simply called gzip)
- Combination of optimization
  - Optimization 1: copy propagation
  - Optimization 2: copy propagation, dead code elimination, common subexpression elimination
  - Optimization 3: copy propagation, loop-invariant code motion
- No. of registers
  - 8 (e.g. x86) and 20 (e.g. RISC machines)

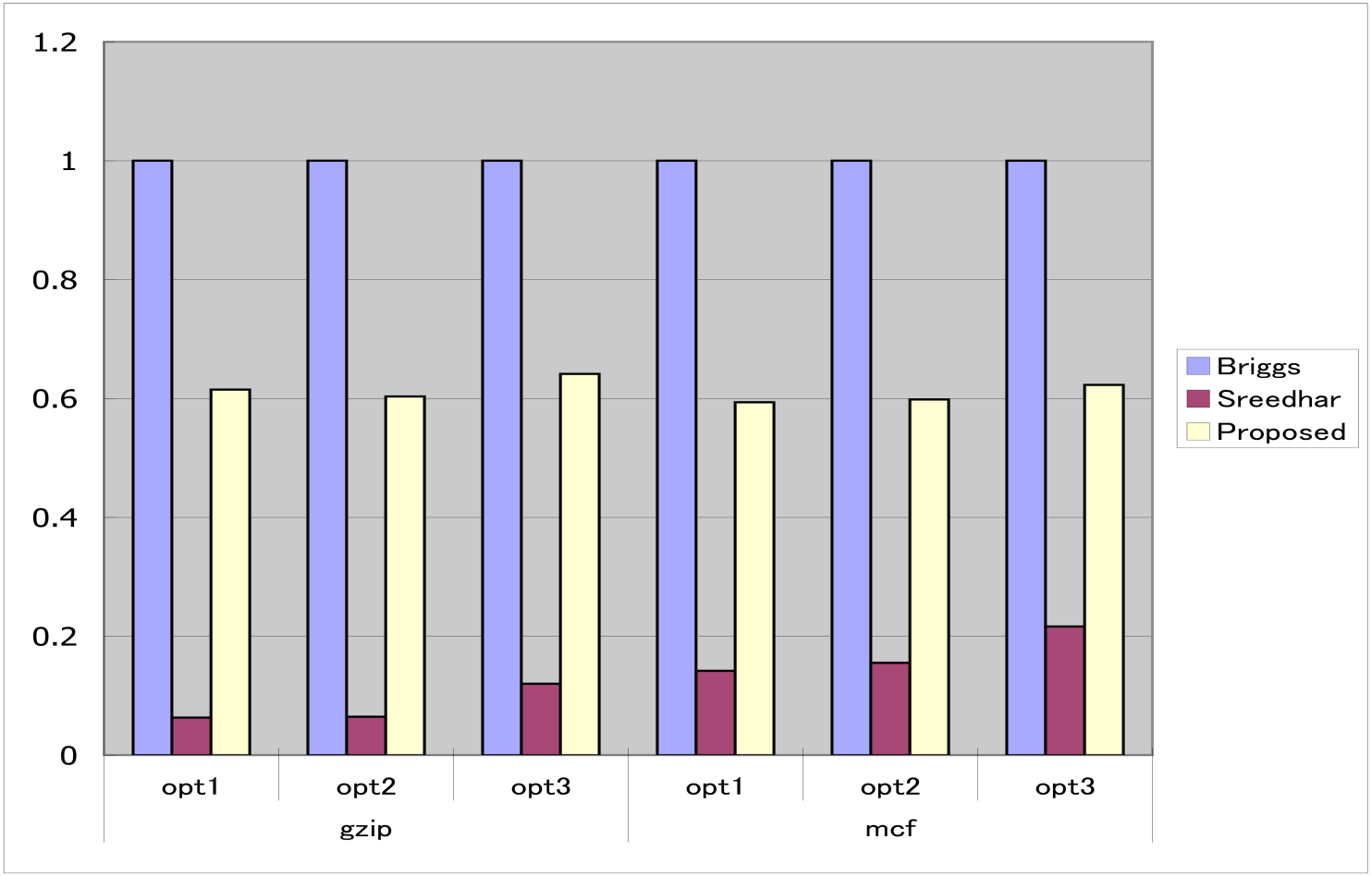
Experiments
- viewpoints of evaluation

- Static Count
  - No. of copies that cannot be coalesced
  - No. of variables spilled in register allocation
- Dynamic Count
  - No. of executed copies
  - No. of executed load and store instructions
  - Execution time



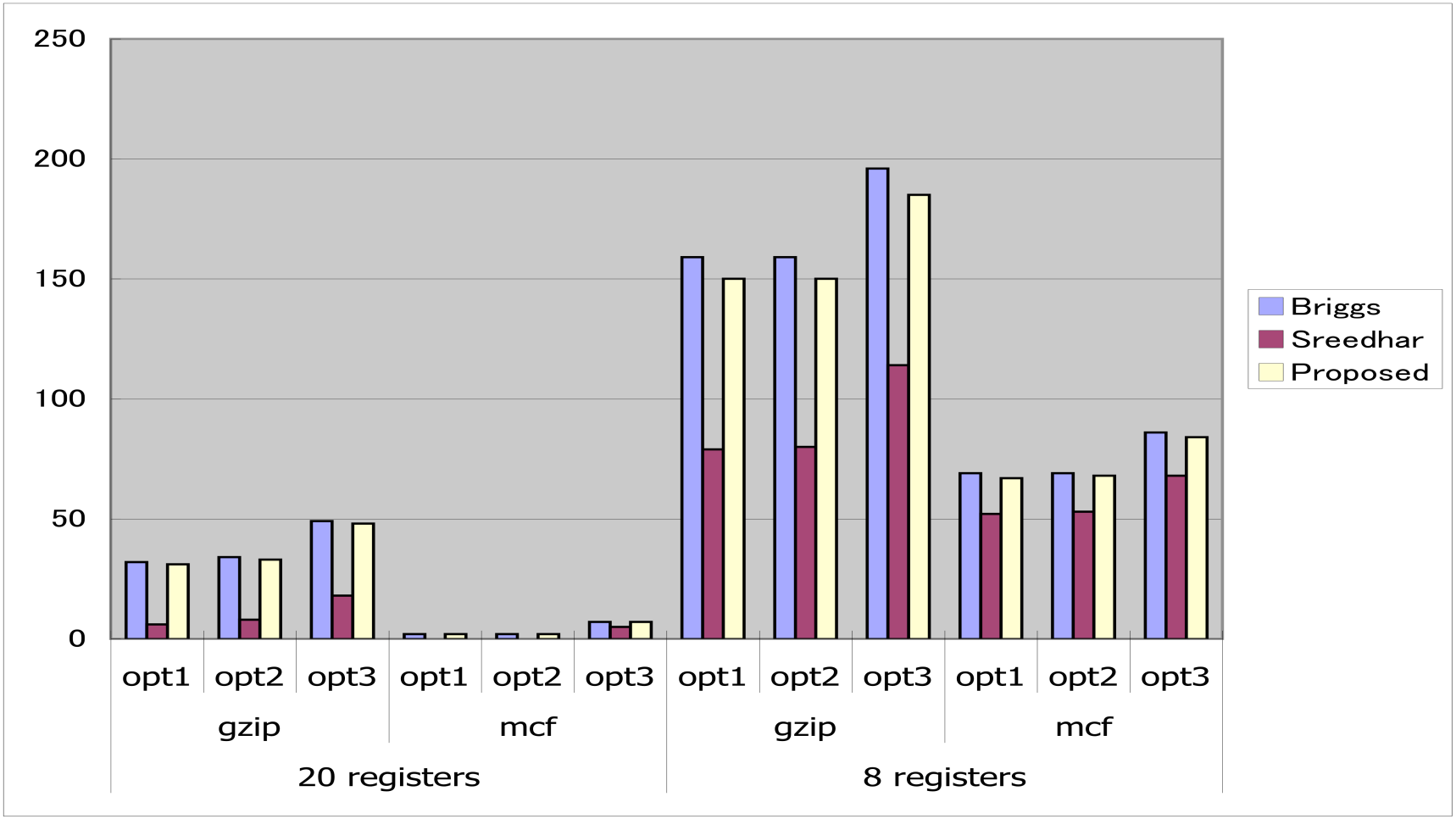
No. of copies that cannot be coalesced (static count)

(relative value: Briggs = 1, small is better)



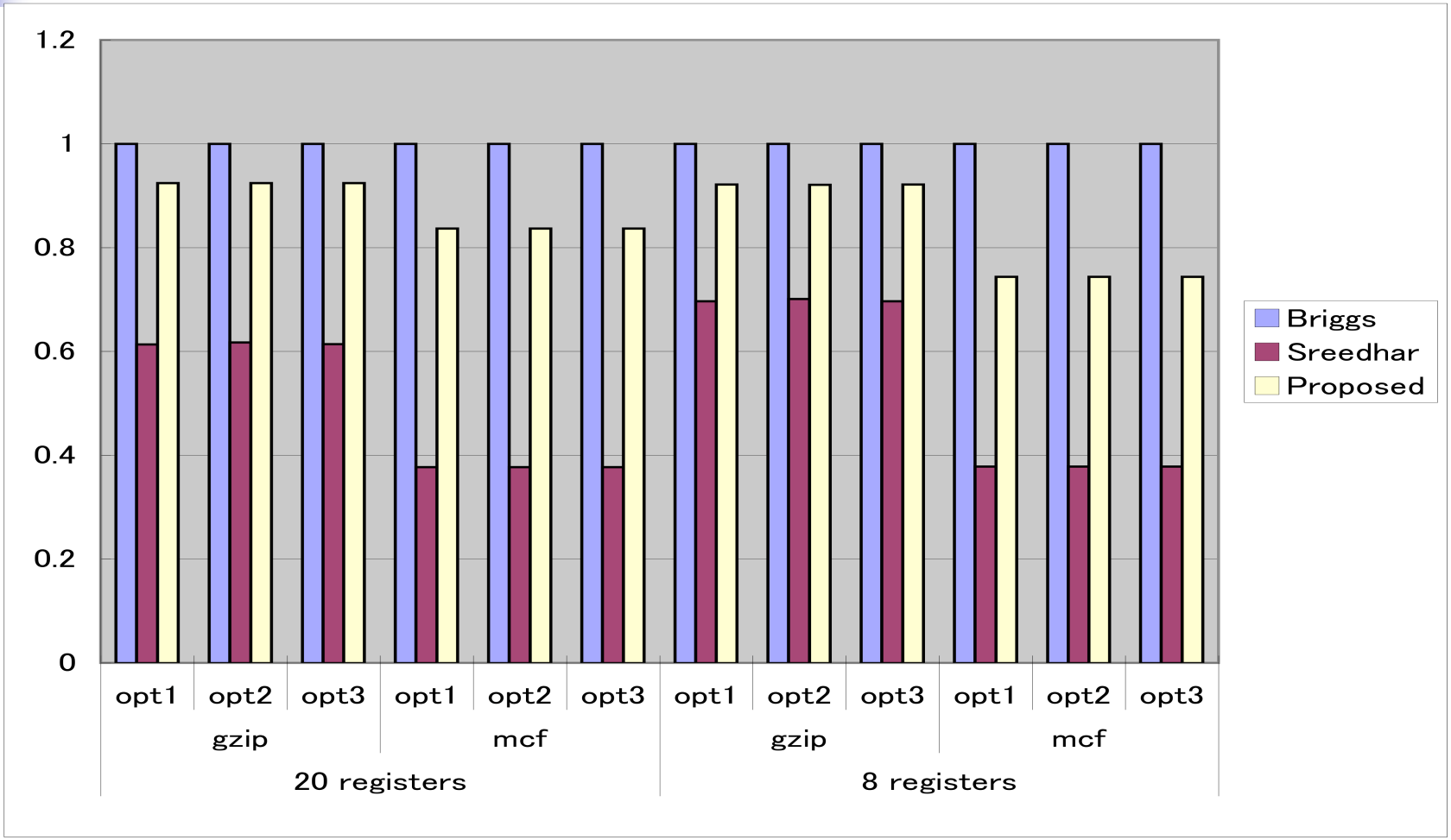
No. of variables spilled (static count)

(unit: absolute value, small is better)



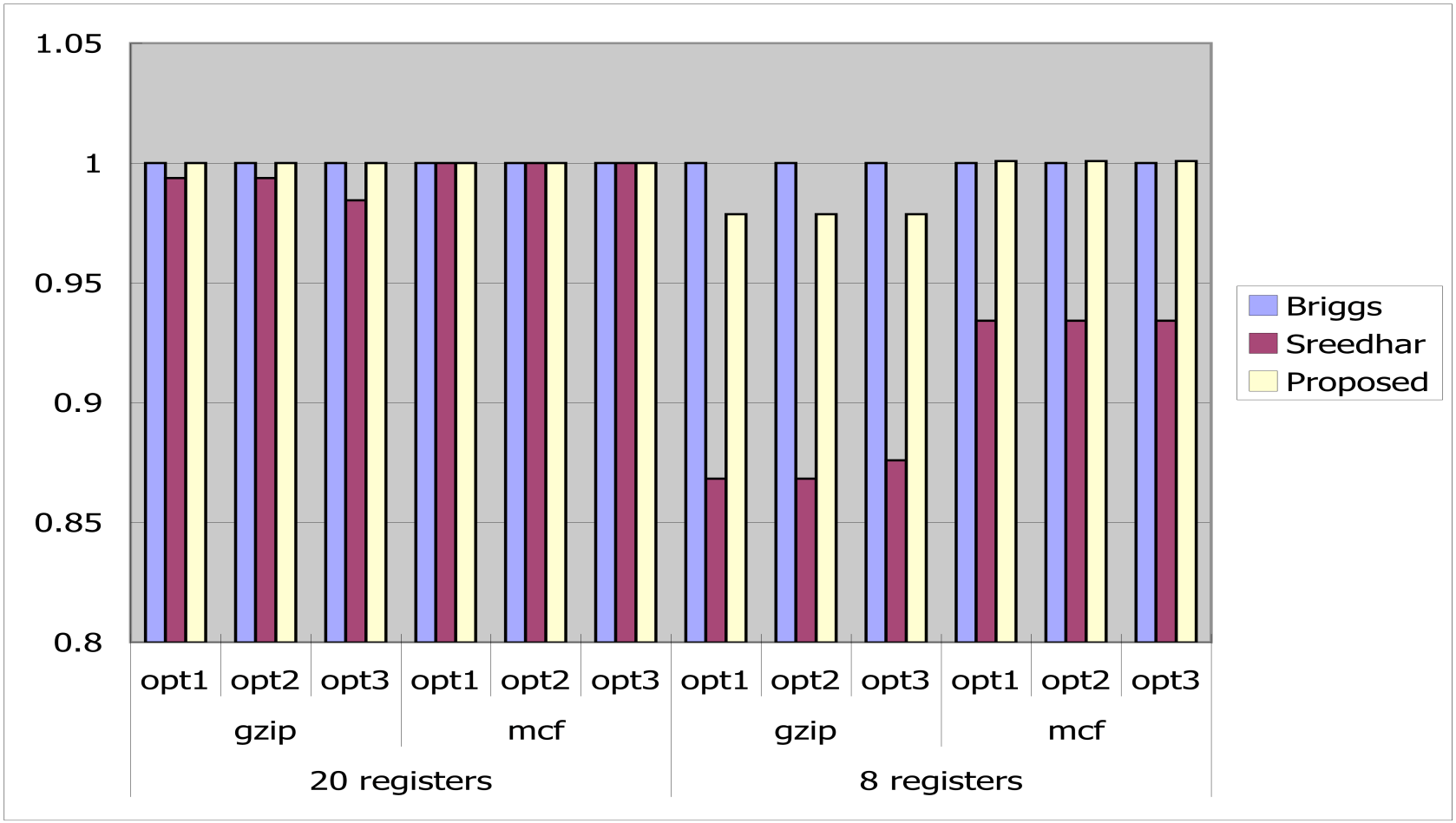
No. of executed copies (dynamic count)

(relative value: Briggs = 1, small is better)

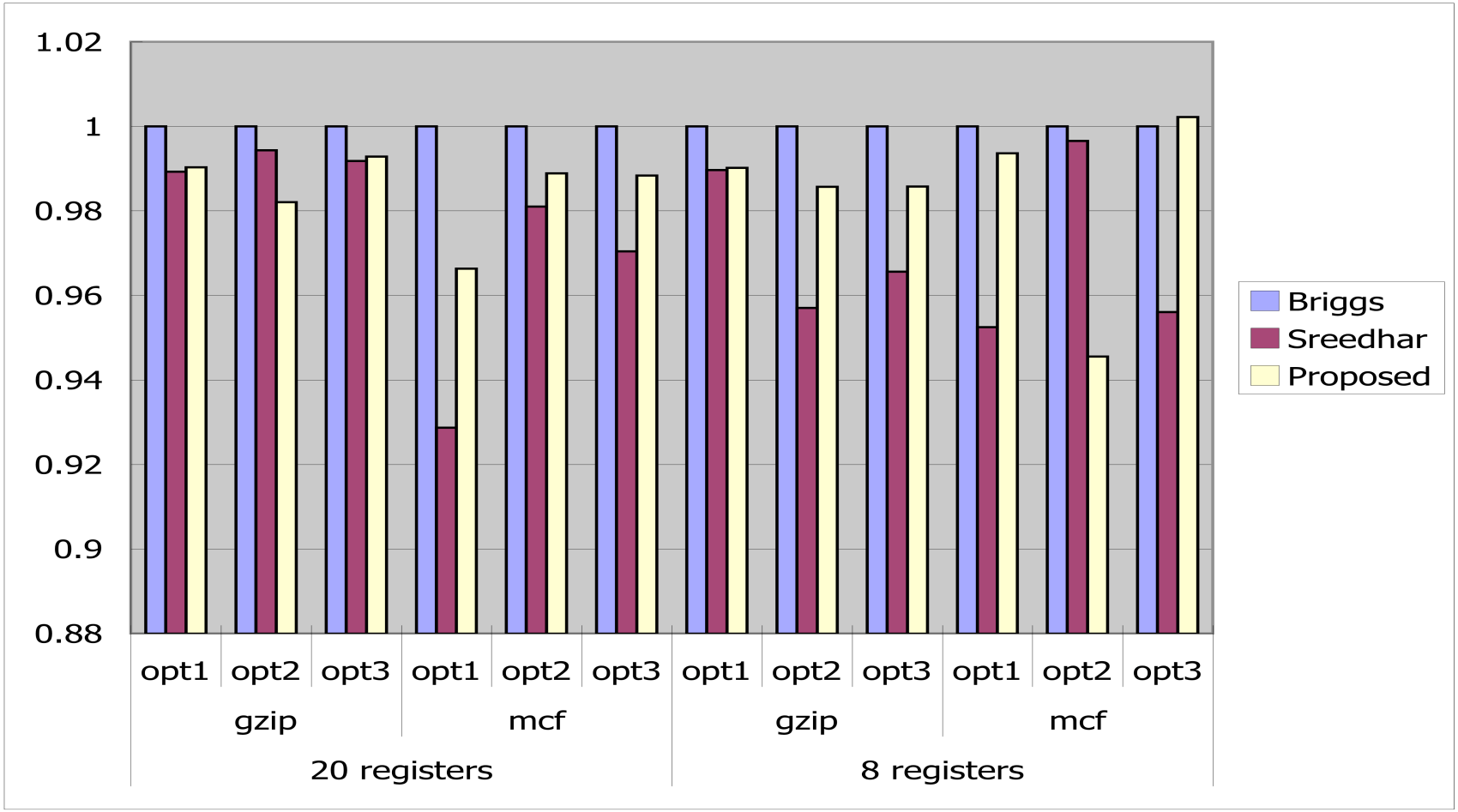


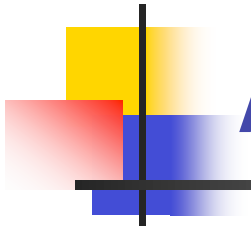
No. of executed load/store instructions (dynamic count)

(relative value: Briggs = 1, small is better)



Execution time
(relative value: Briggs = 1, small is better)

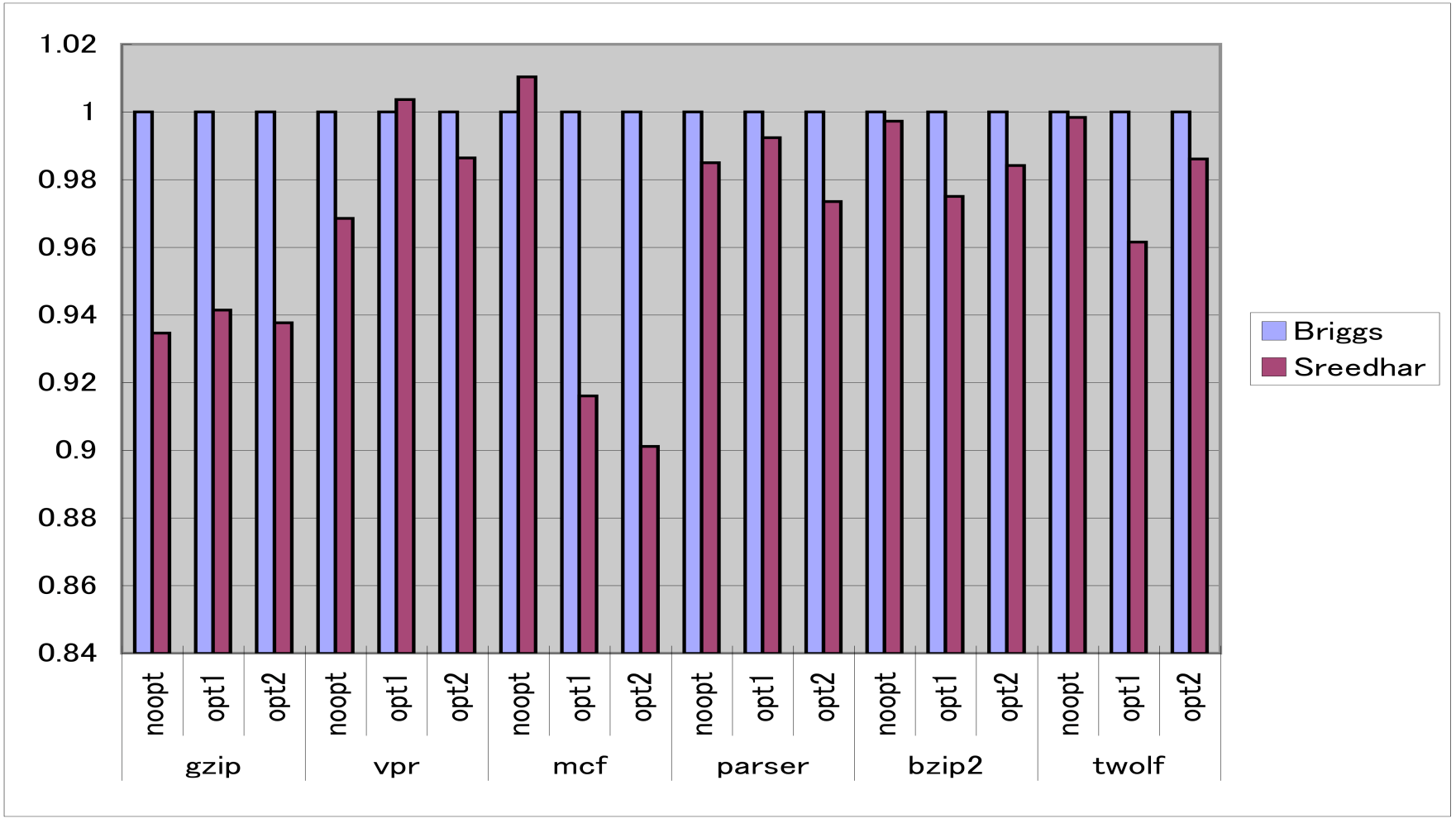


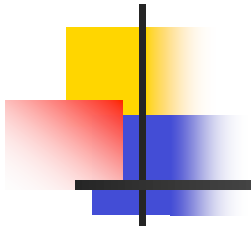


Additional experiments

- More benchmarks is executed on more general optimizations for Briggs' and Sreedhar's algorithms
- Benchmarks
  - SPECint2000 gzip, vpr, mcf, parser, bzip2, twolf
- Combination of optimizations
  - No optimization
  - Optimization 1: copy propagation, constant propagation, dead code elimination
  - Optimization 2: loop-invariant motion, constant propagation, common subexpression elimination, copy propagation, dead code elimination
- Execution time is measured

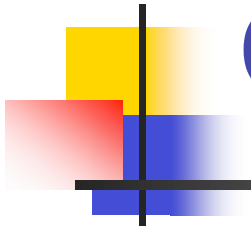
Execution time
(relative value: Briggs = 1, small is better)





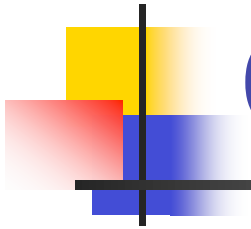
Summary of experiments

- Briggs's algorithm gave many copies that cannot be coalesced, which is beyond our expectation
- Case of 20 registers
  - Sreedhar's algorithm is advantageous due to the dynamic count of copies
  - It gives better execution time by around 1%~10%
- Case of 8 registers
  - Sreedhar's algorithm is advantageous due to both the dynamic count of copies, and the dynamic count of load/store instructions
  - It gives better execution time by around 1%~5%
- Effect of SSA back translation algorithms does not depend so much to the combination of optimizations
- Our proposed improvement has some effect over Briggs', but does not surpass Sreedhar's algorithm



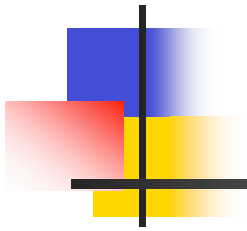
Conclusion and future work

- Our contribution
  - Has shown that difference in SSA back translation algorithms influences execution time by up to 10%
  - Experiments in a variety of situations
    - Clarified advantage and disadvantage of different algorithms
    - Proposed an improvement of Briggs' algorithm
    - Sreedhar's algorithm is superior from experiments (up to 10%)
- Future work
  - Further experiments measuring several facet using more benchmarks
  - Consider coalescing algorithm in register allocation



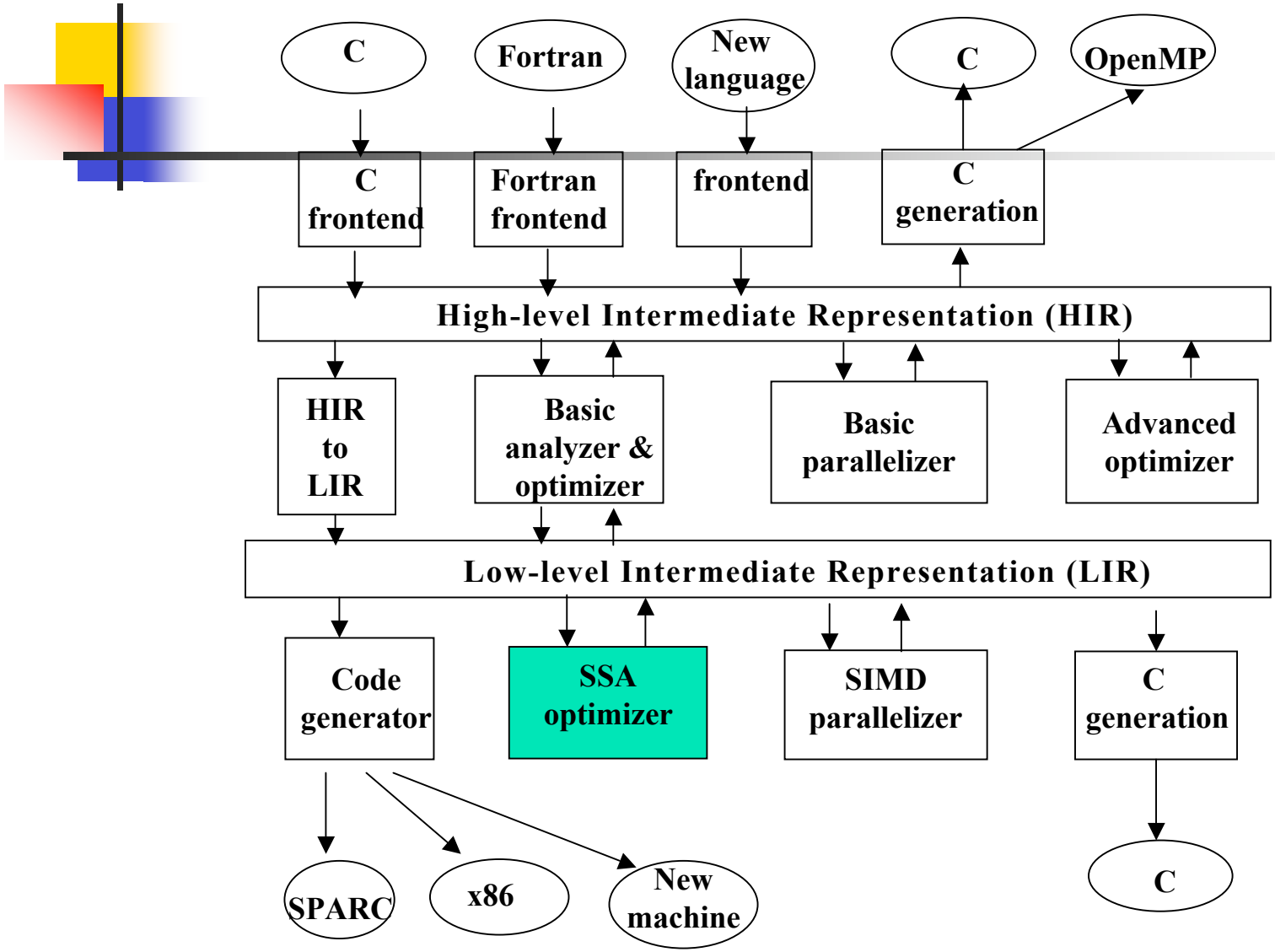
Conclusion

- The experimental result shows that **selecting a good back translation algorithm is quite important** since it reduces the execution time of the object code by up to 10%, which is equal to applying a middle-level global optimization.

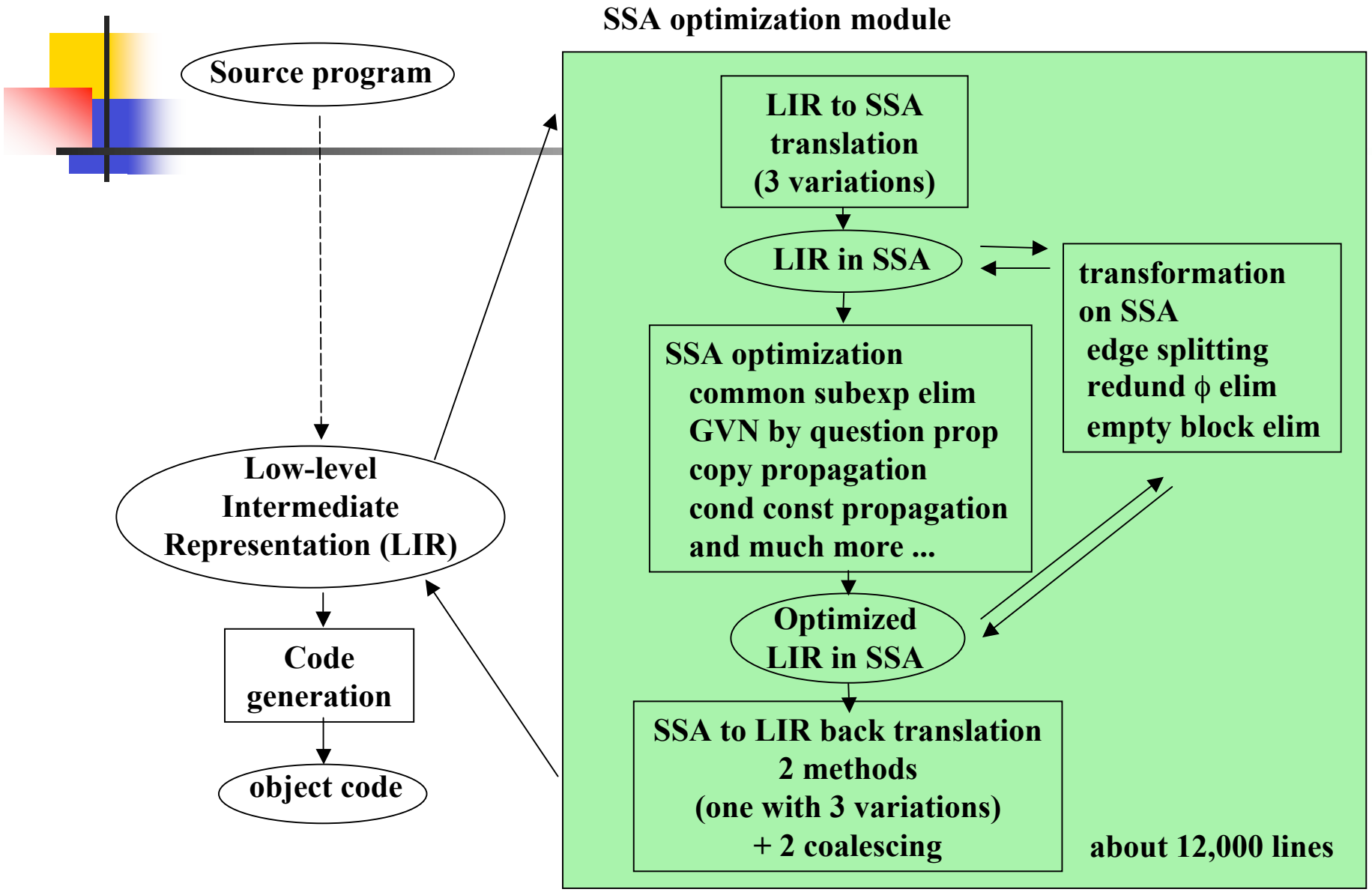


Appendix
SSA optimization module in COINS

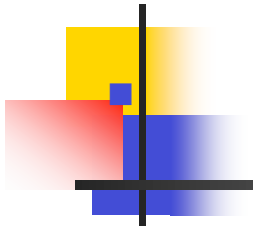
COINS compiler infrastructure



SSA optimization module in COINS



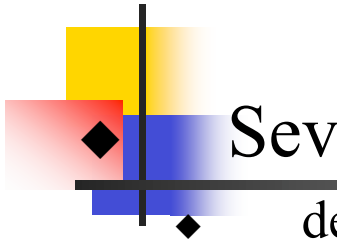
Outline of SSA module in COINS (1)



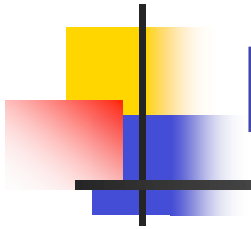
Translation into and back from SSA form on Low-level Intermediate Representation (LIR)

- 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.'s method [Sreedhar et al. 99]
    - 3 variations: Method I, II, and III
  - Briggs et al.'s method [Briggs et al. 98] (under development)
- 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)



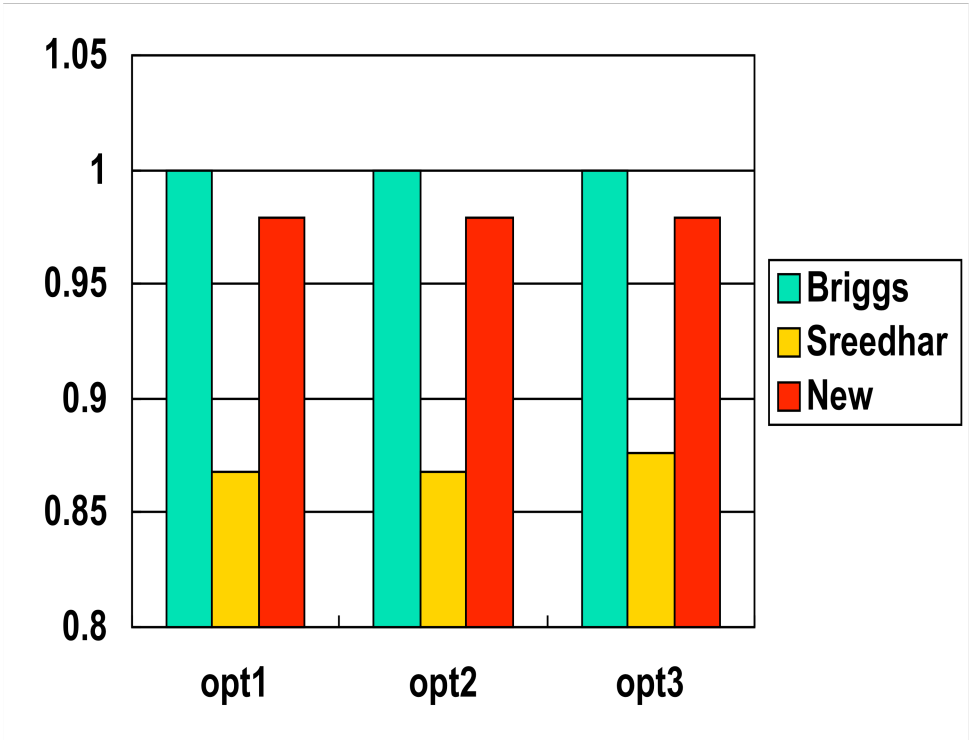
- Several optimization on SSA form:
  - dead code elimination, copy propagation, common subexpression elimination, global value numbering based on efficient query propagation, conditional constant propagation, loop invariant code motion, operator strength reduction for induction variable and linear function 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



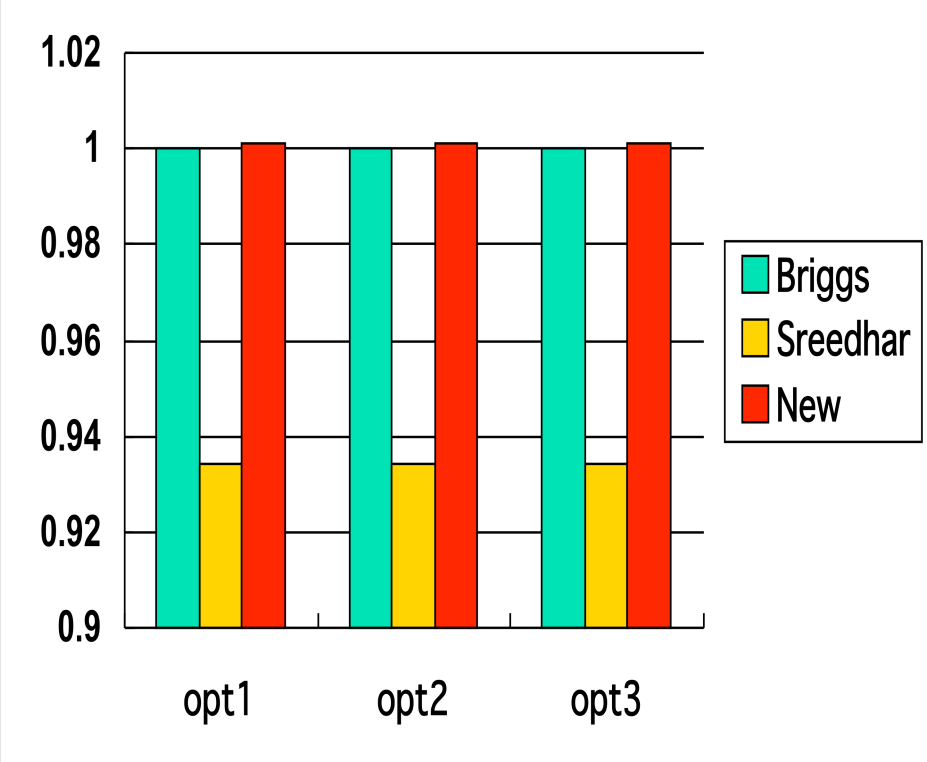
References

- COINS
  - http://www.coins-project.org/
- SSA module in COINS
  - http://www.is.titech.ac.jp/~sassa/coins-www-ssa/english/index.html

Relative dynamic count of load and store
Number of registers = 8

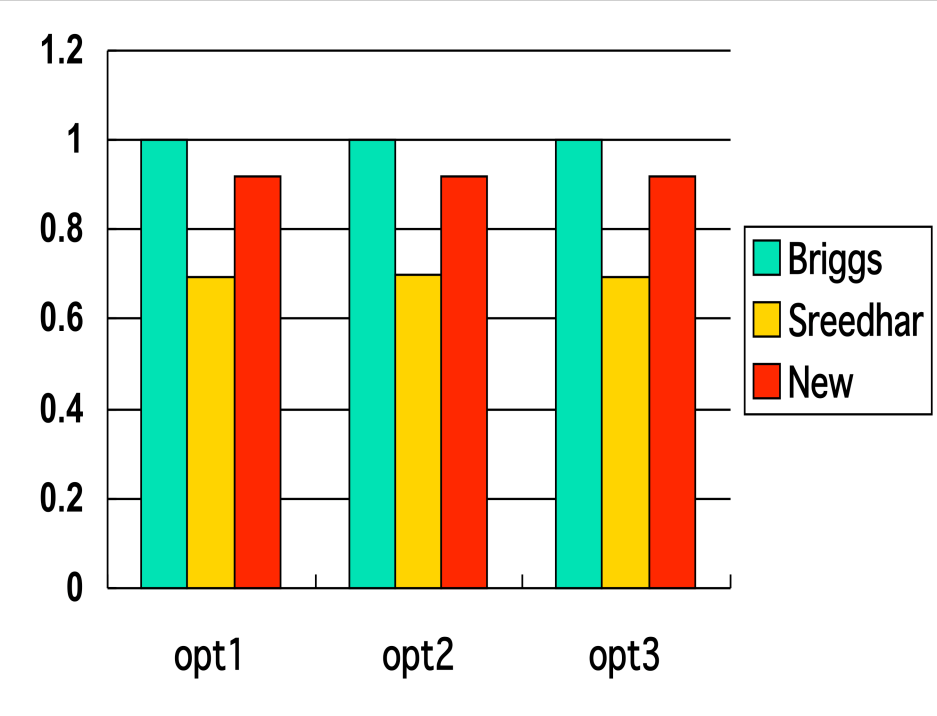


gzip

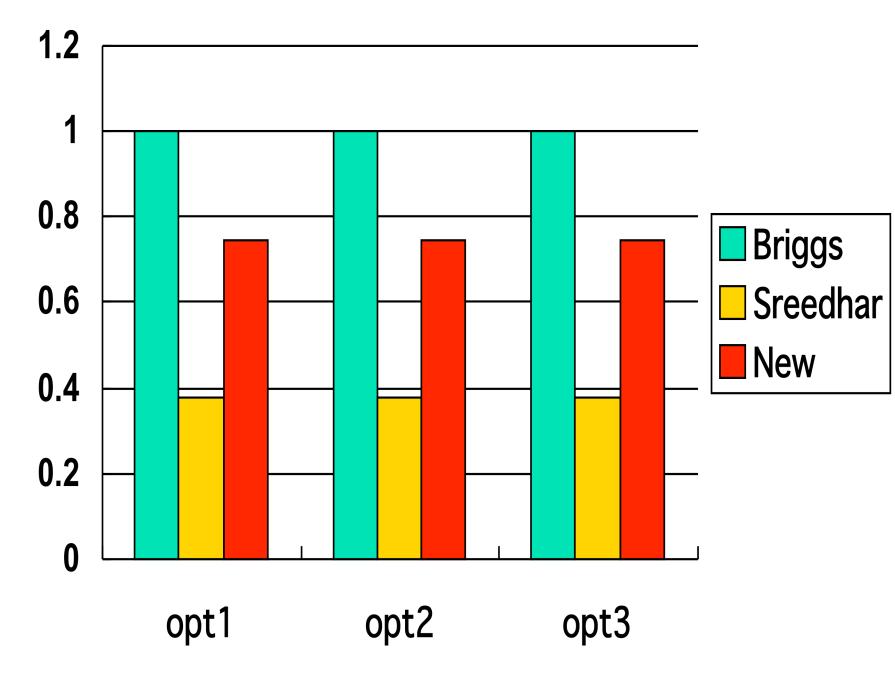


mcf

Relative dynamic count of copies
Number of registers = 8

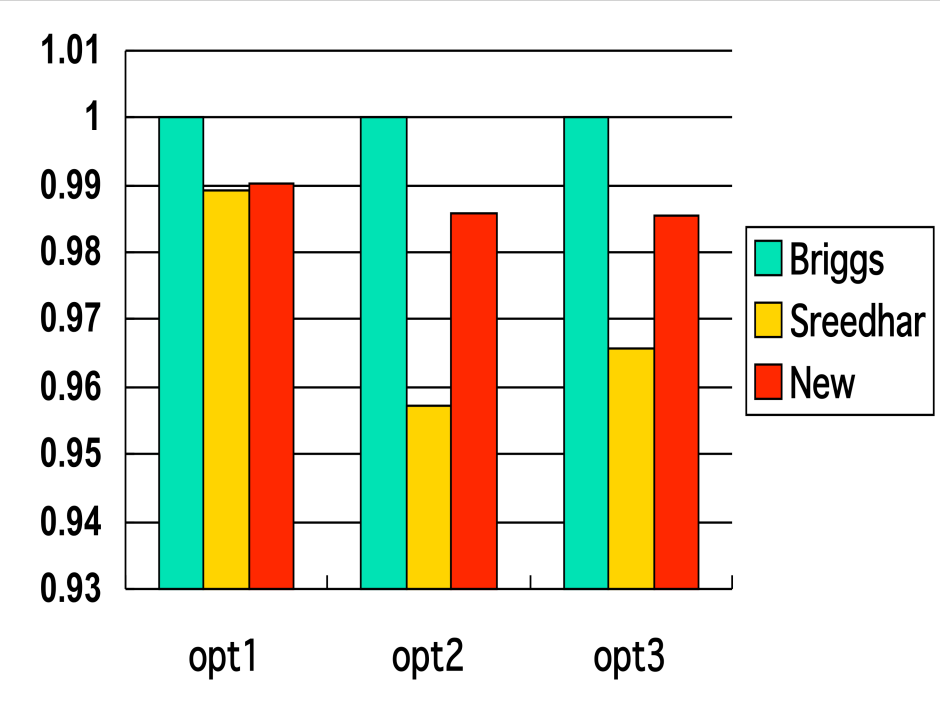


gzip

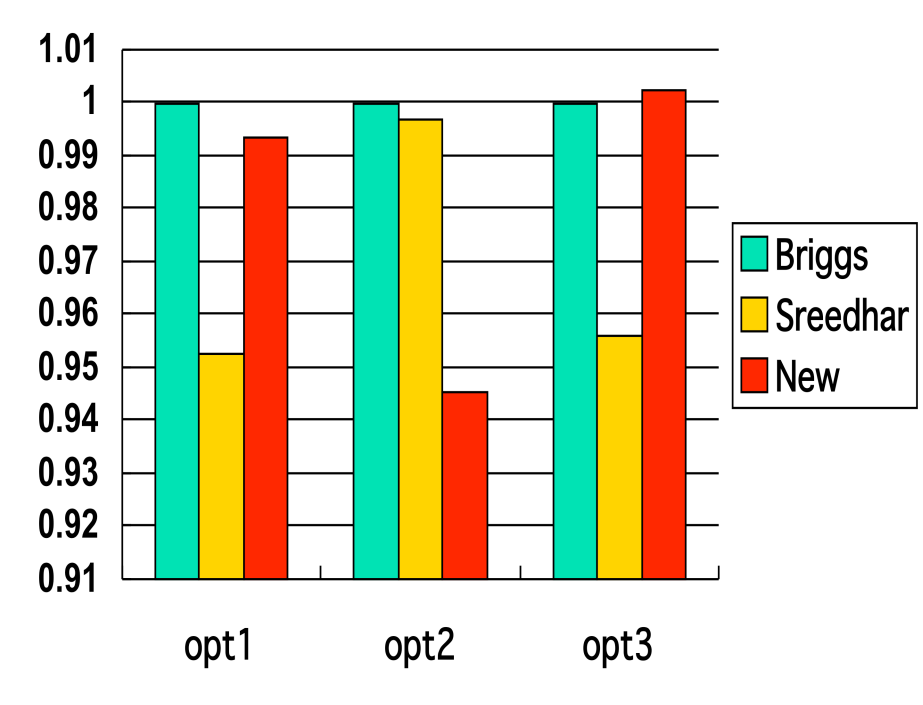


mcf

Relative execution time
Number of registers = 8

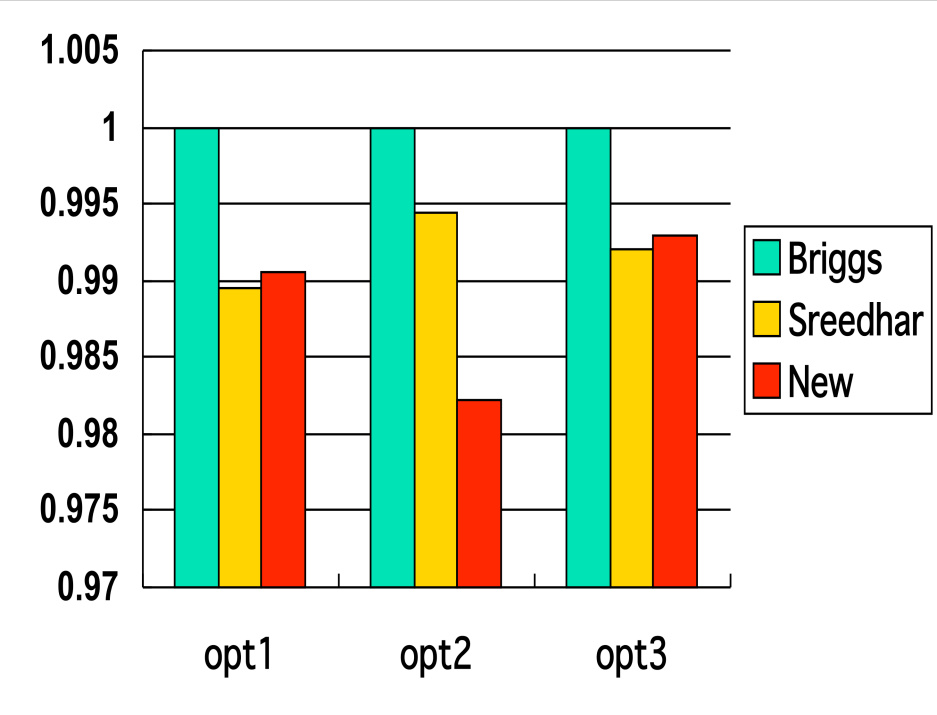


gzip

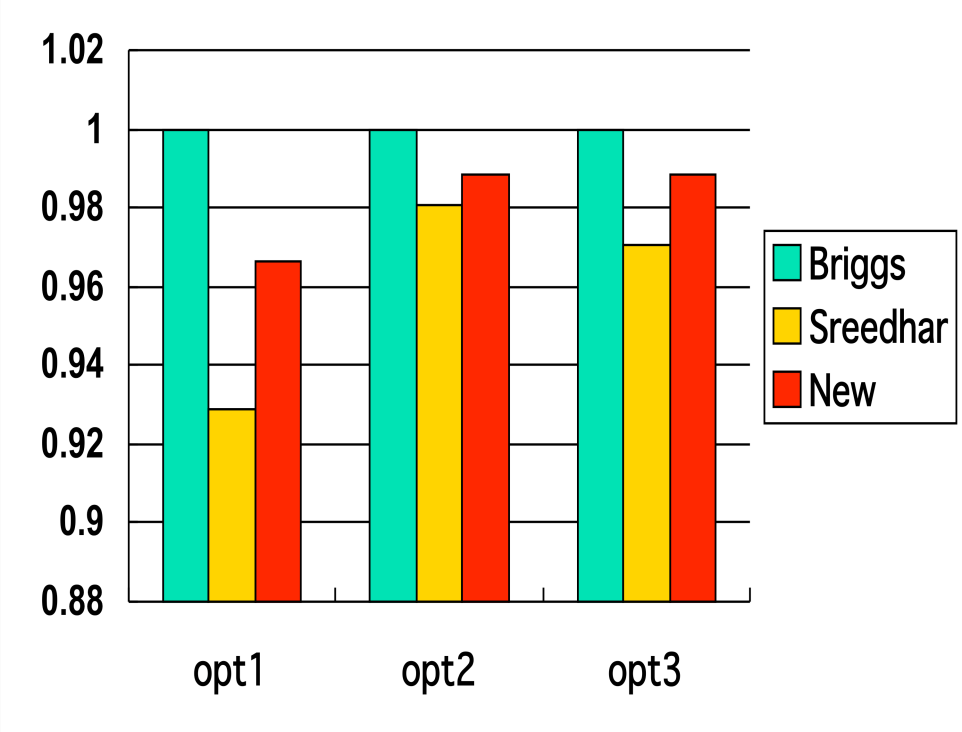


mcf

Relative execution time
Number of registers = 20



gzip



mcf