

# Static Single Assignment Form in the COINS Compiler Infrastructure - Current Status and Background -

Masataka Sassa, Toshiharu Nakaya, Masaki Kohama  
(Tokyo Institute of Technology)

Takeaki Fukuoka, Masahito Takahashi and Ikuo Nakata

# Background

**Static single assignment (SSA) form** facilitates compiler optimizations.

**Compiler infrastructure** facilitates compiler development.

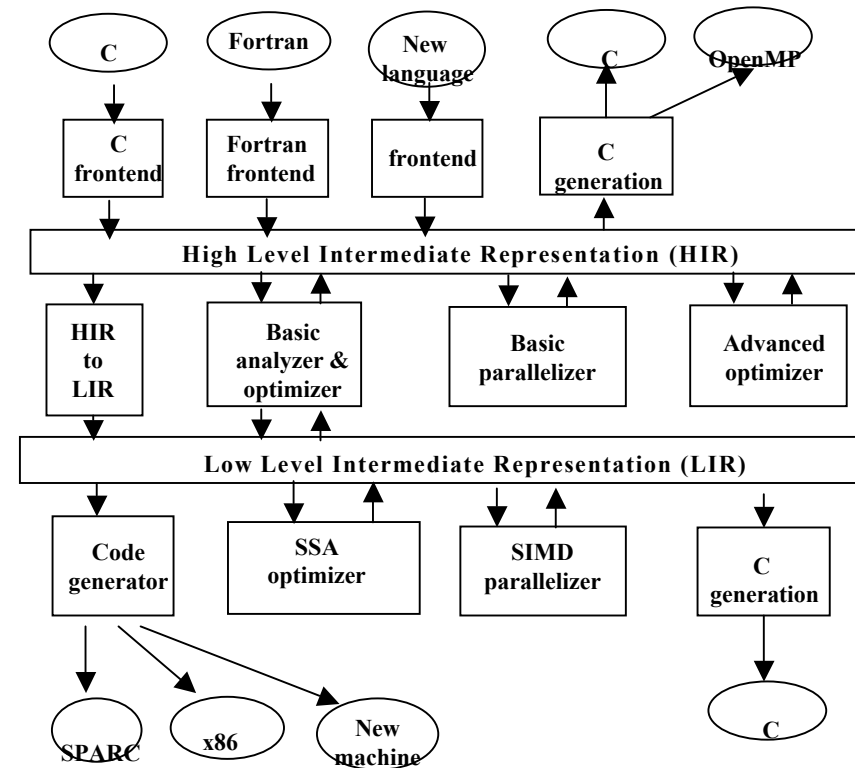
## Outline

0. COINS infrastructure and the SSA form
1. Current optimization using SSA form in COINS
2. A comparison of SSA translation algorithms
3. A comparison of SSA back translation algorithms
4. A survey of compiler infrastructures

# 0. COINS infrastructure and Static Single Assignment Form (SSA Form)

# COINS compiler infrastructure

- Multiple source languages
- Retargetable
- Two intermediate form, 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: a1 = x0 + y0
2: a2 = a1 + 3
3: b1 = x0 + y0
```

(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)

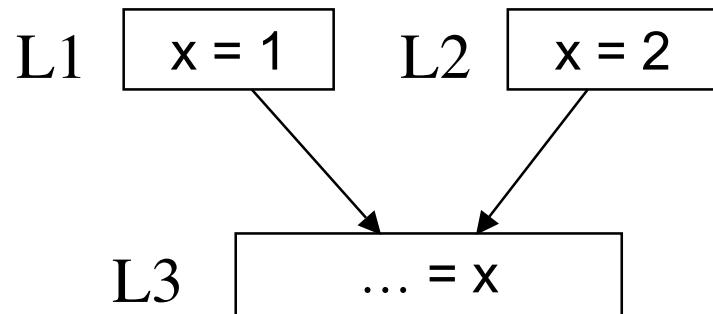


(c) After SSA form optimization

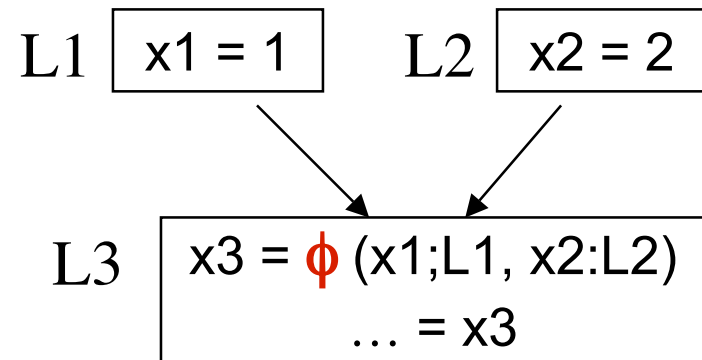
(d) Optimized normal form

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)

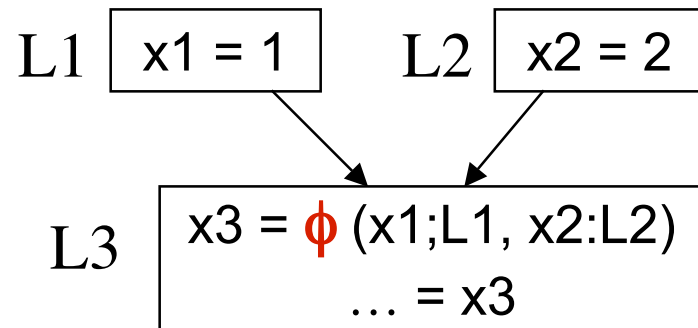


(a) Normal form

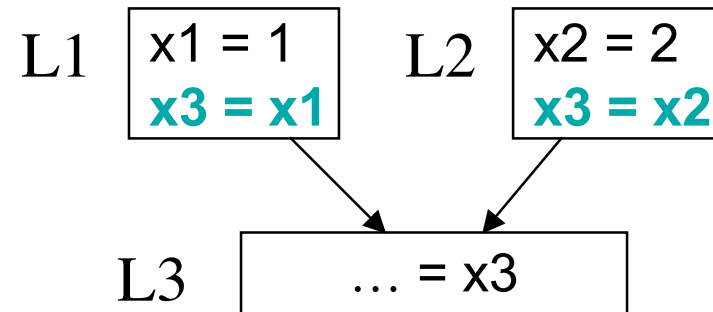


(b) SSA form

# Translating back from SSA form (SSA back translation)



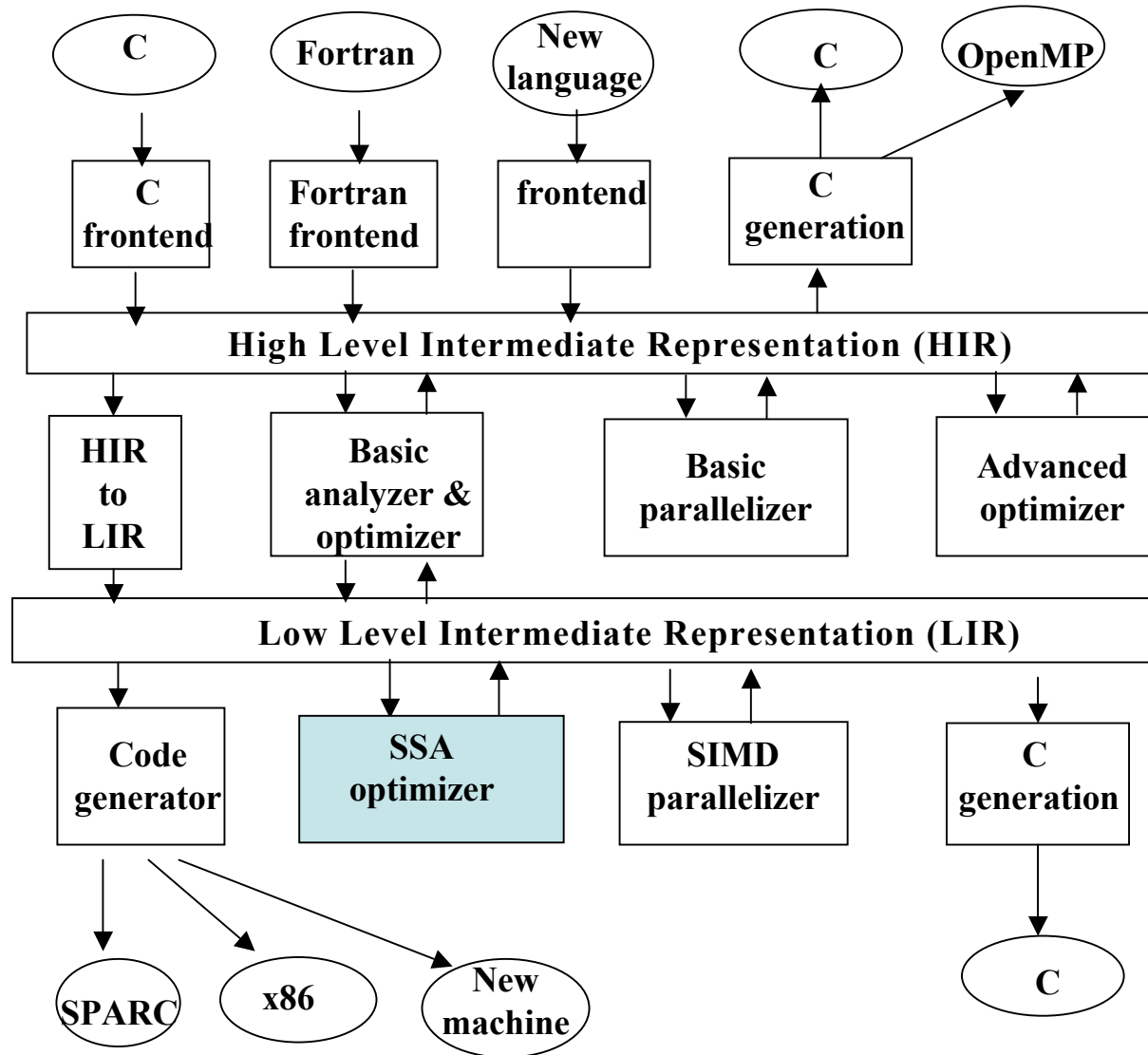
(a) SSA form



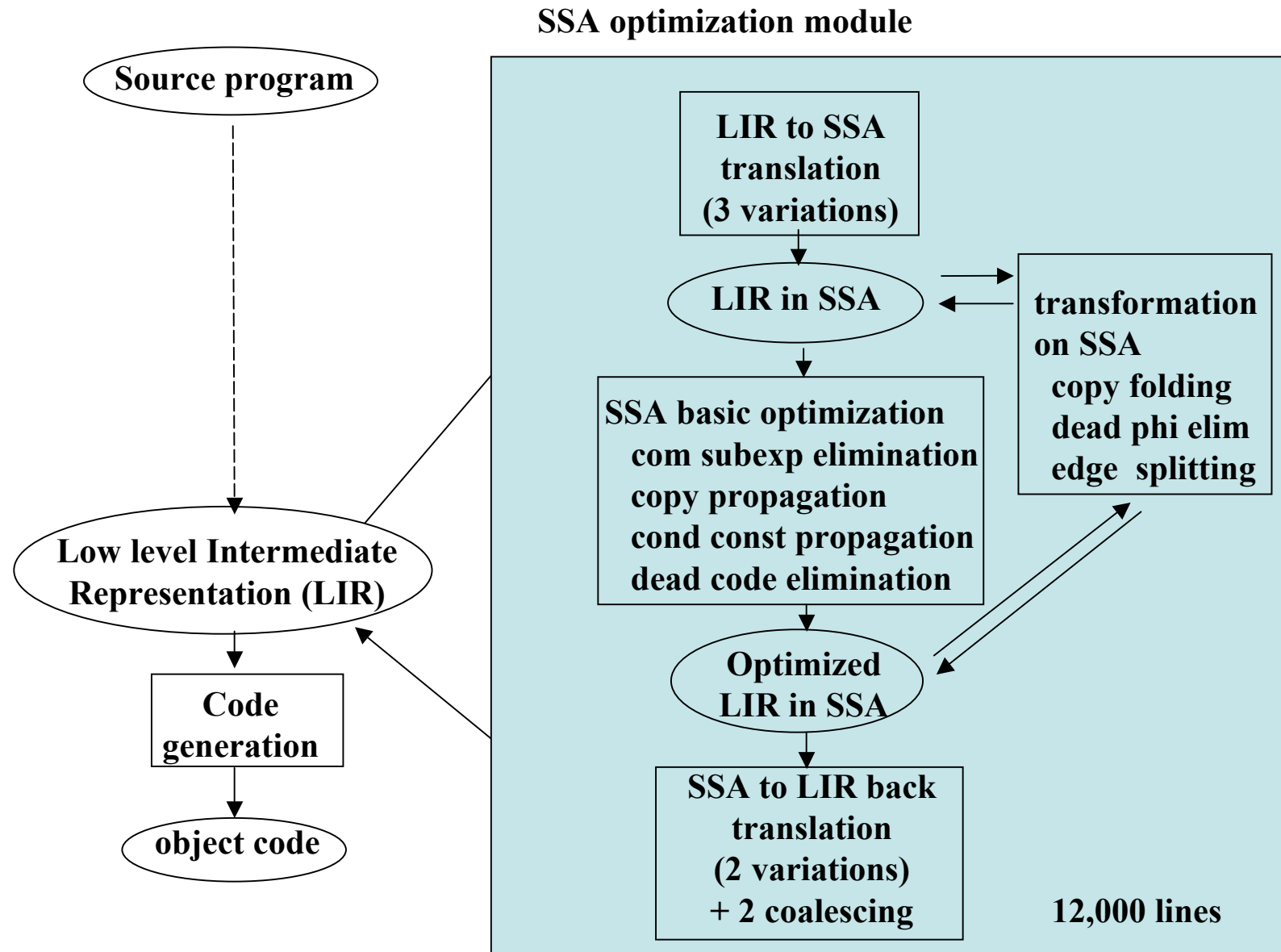
(b) Normal form

1. SSA form module in the  
COINS compiler infrastructure

# COINS compiler infrastructure



# SSA optimization module in COINS



# Outline of SSA module in COINS

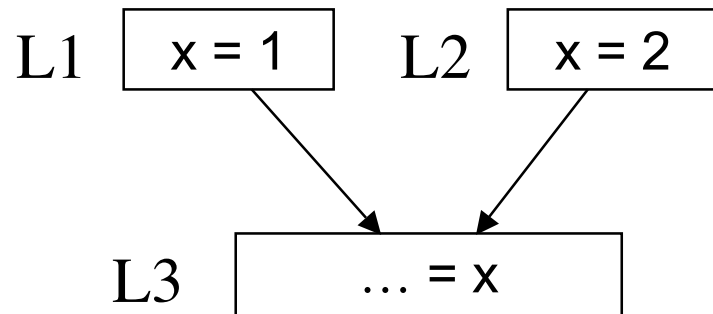
- Translation into and back from SSA form on Low Level Intermediate Representation (LIR)
  - SSA translation: Use dominance frontier [Cytron et al. 91]
  - SSA back translation: [Sreedhar et al. 99]
  - Basic optimization on SSA form: dead code elimination, copy propagation, common subexpression elimination, conditional constant propagation
- Useful transformation as an infrastructure for SSA form optimization
  - Copy folding at SSA translation time, critical edge removal on control flow graph ...
  - Each variation and transformation can be made selectively
- Preliminary result
  - 1.43 times faster than COINS w/o optimization
  - 1.25 times faster than gcc w/o optimization

## 2. A comparison of two major algorithms for SSA translation

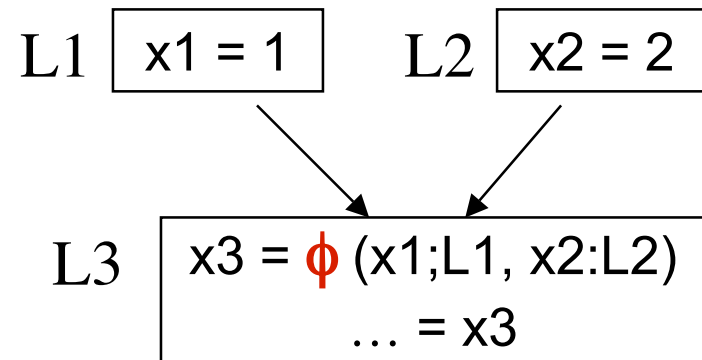
- Algorithm by Cytron [1991]  
Dominance frontier
- Algorithm by Sreedhar [1995]  
DJ-graph

Comparison made to decide the algorithm to be included in COINS

# Translating into SSA form (SSA translation)

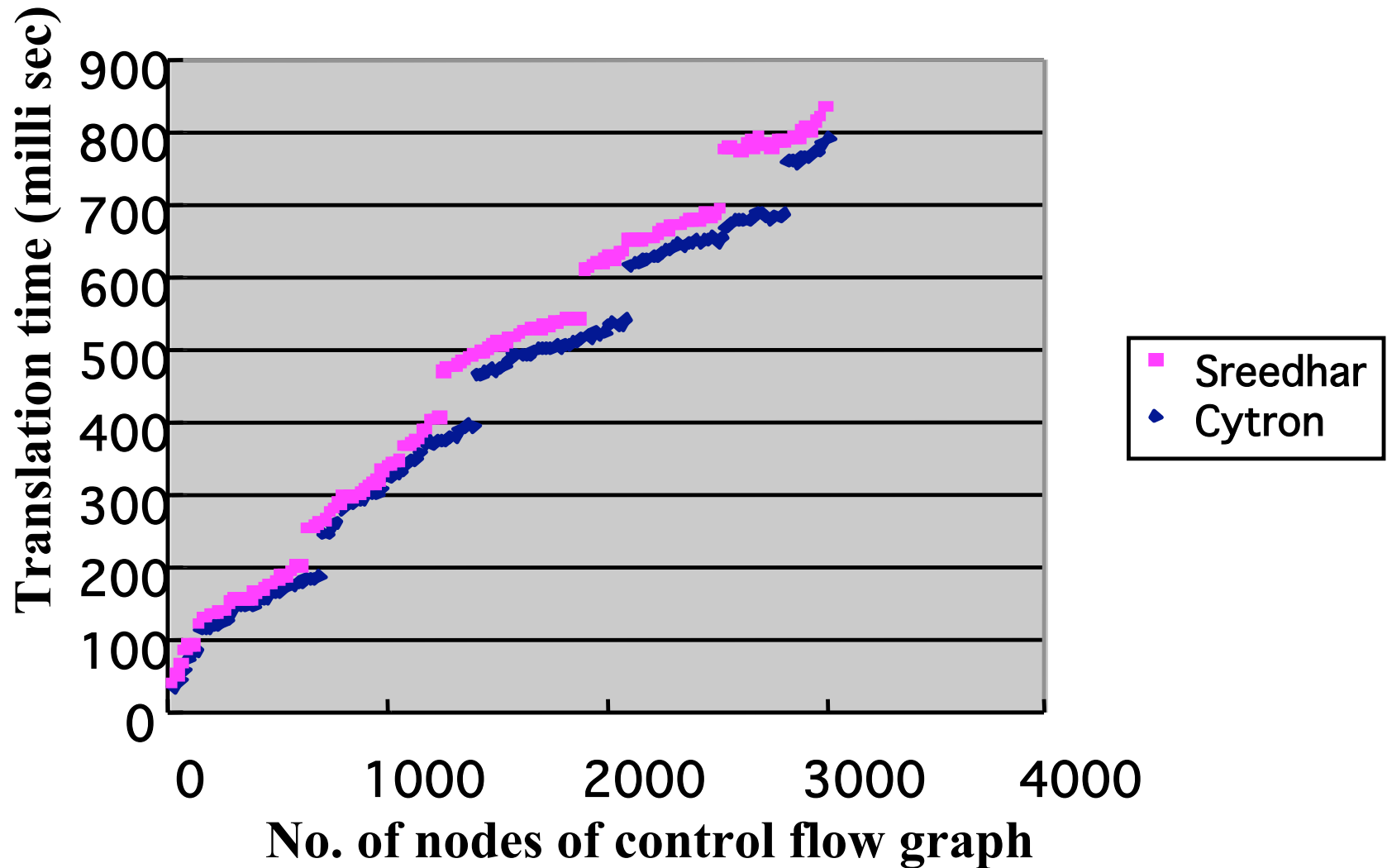


(a) Normal form



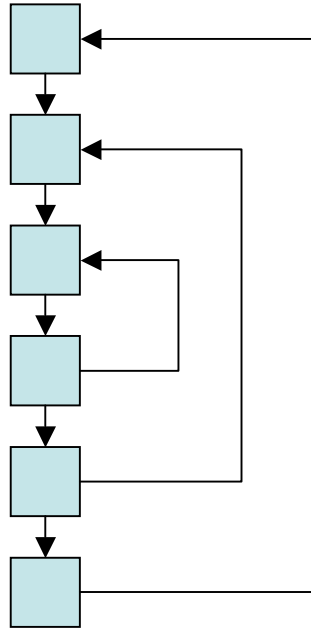
(b) SSA form

# SSA translation time (usual programs)

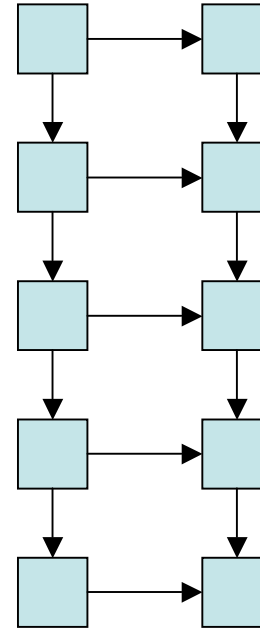


(The gap is due to the garbage collection)

# Peculiar programs

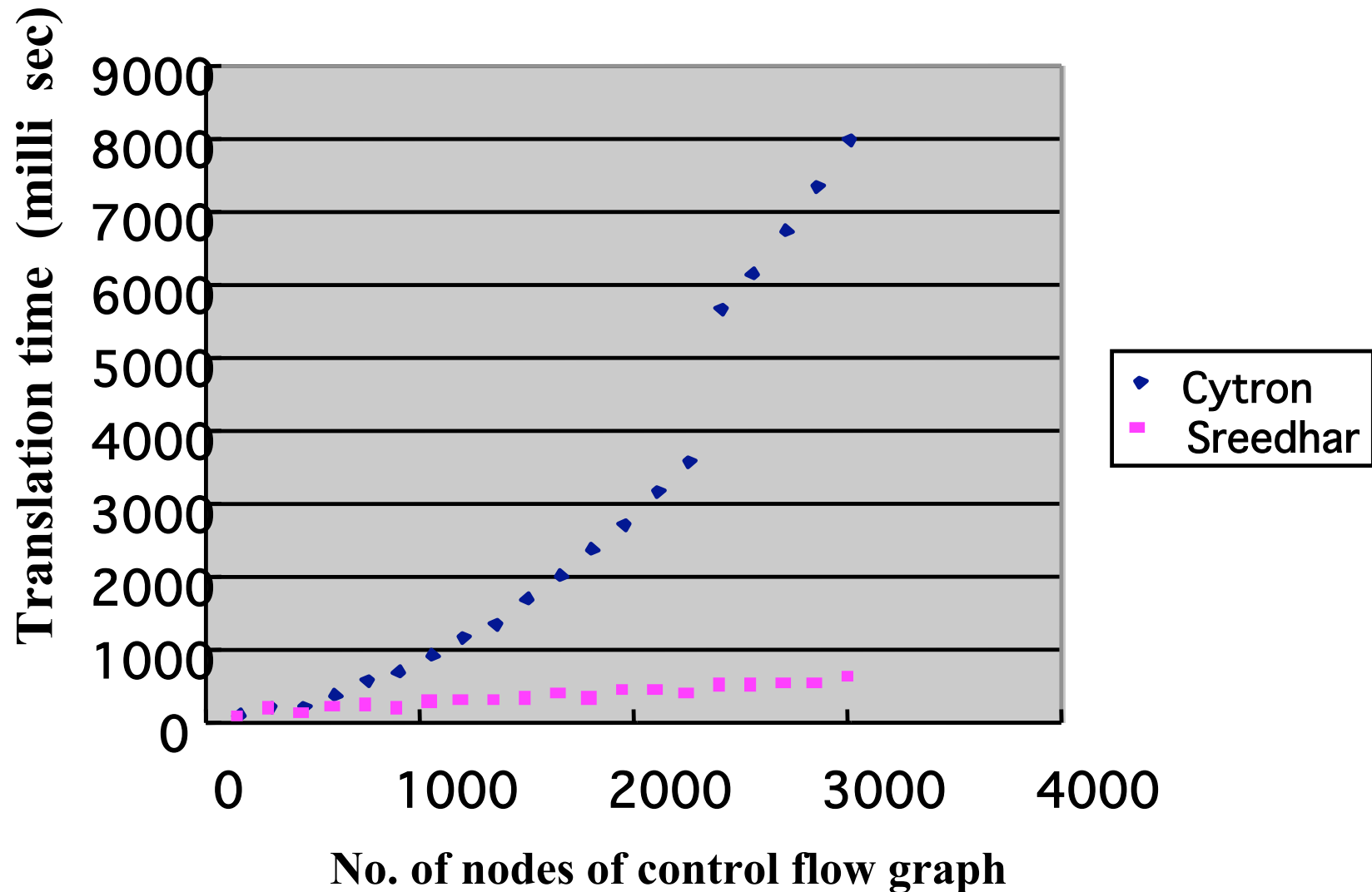


(a) nested loop

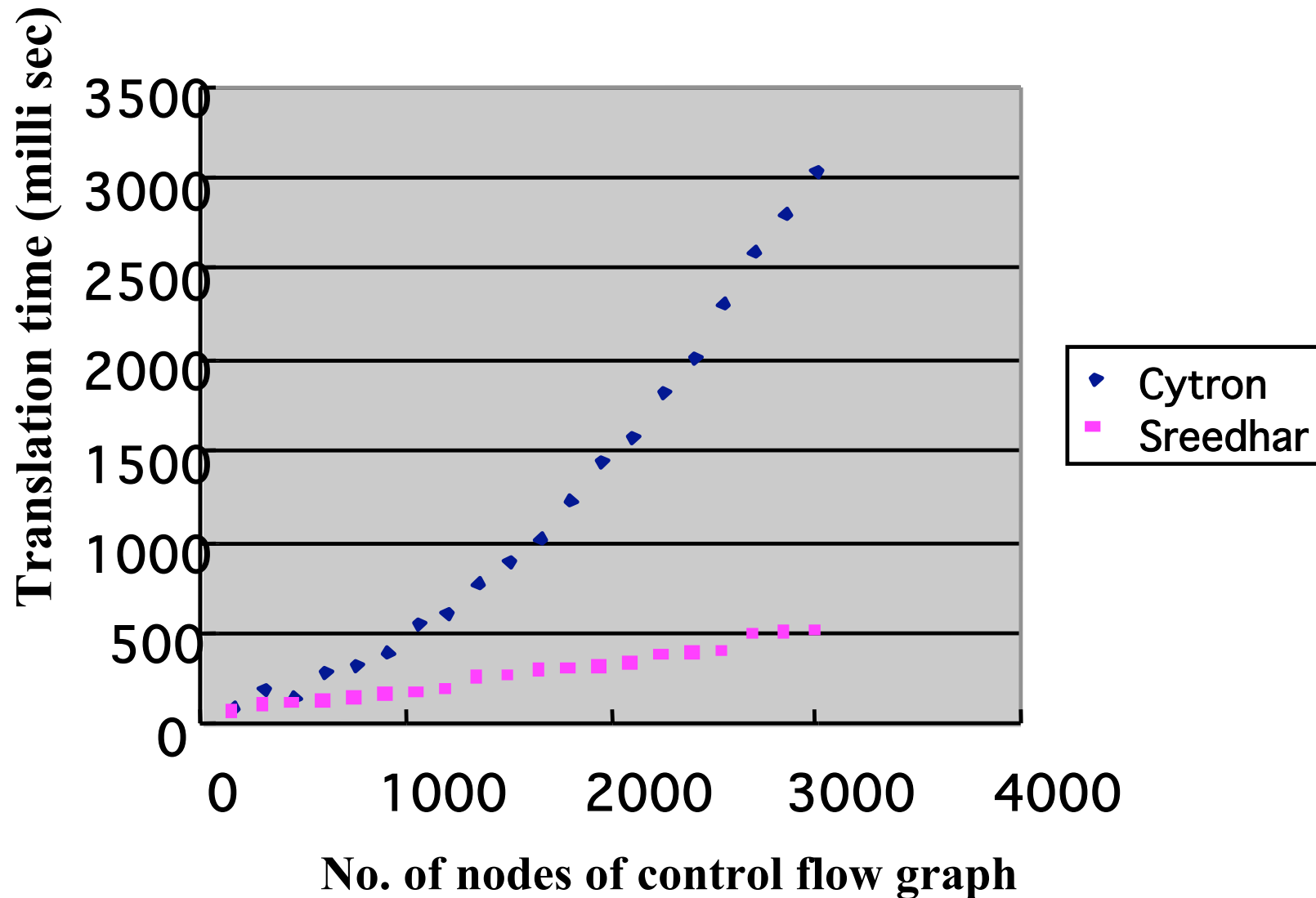


(b) ladder graph

# SSA translation time (nested loop programs)



# SSA translation time (ladder graph programs)

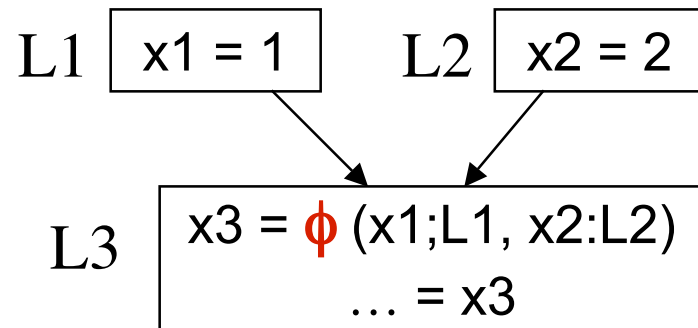


### 3. A comparison of two major algorithms for SSA back translation

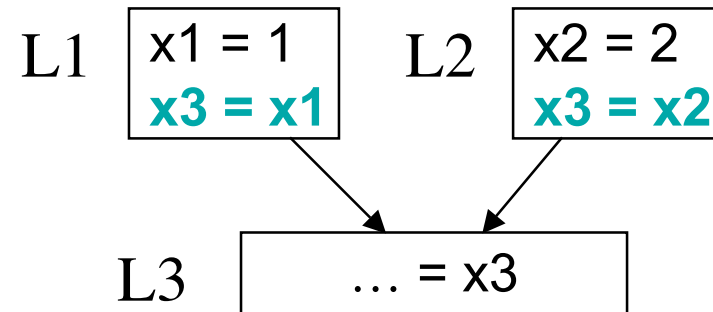
- Algorithm by Briggs [1998]  
Insert copy statements
- Algorithm by Sreedhar [1999]  
Eliminate interference

There have been no studies of comparison  
Comparison made on COINS

# Translating back from SSA form (SSA back translation)

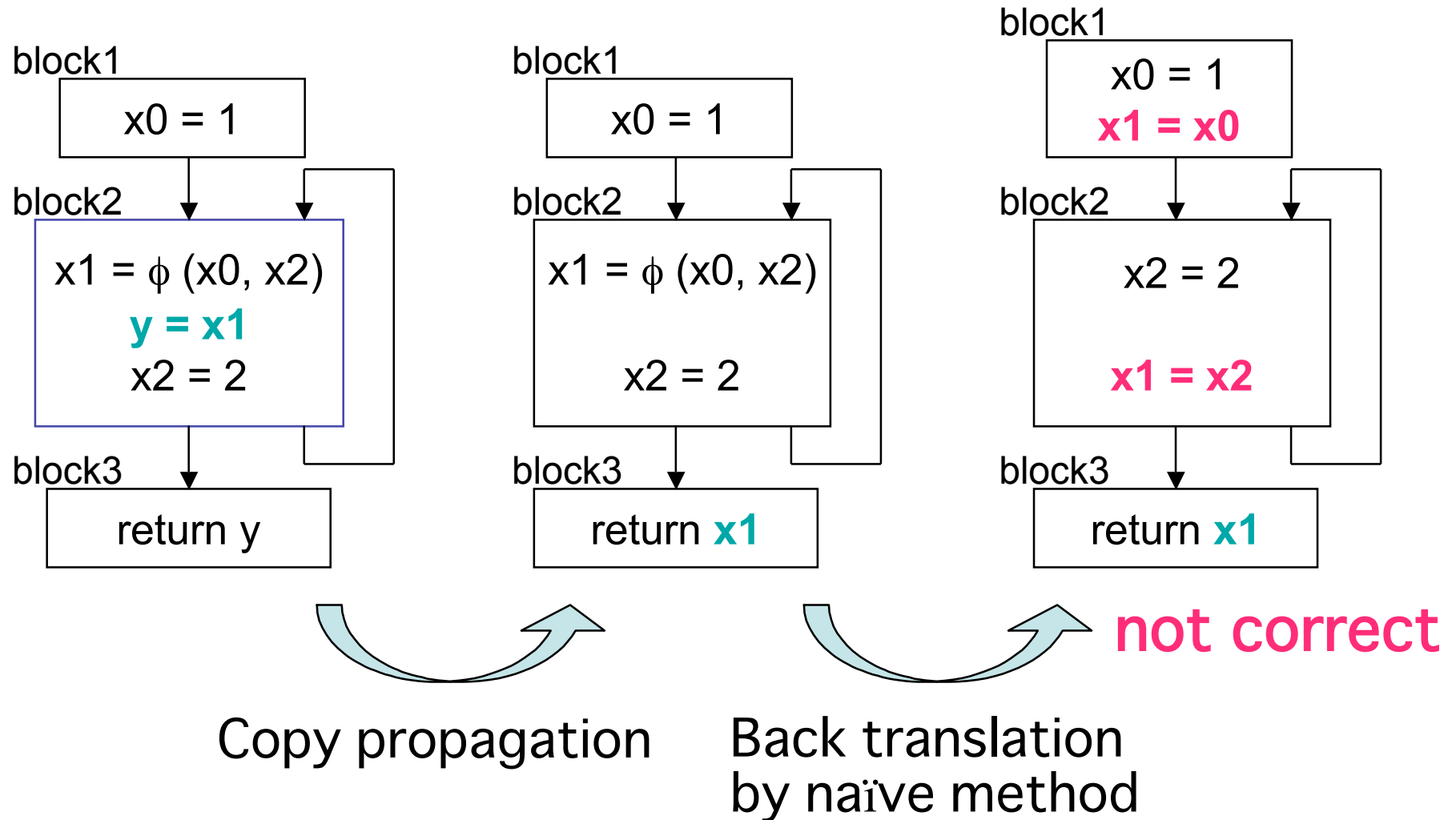


(a) SSA form



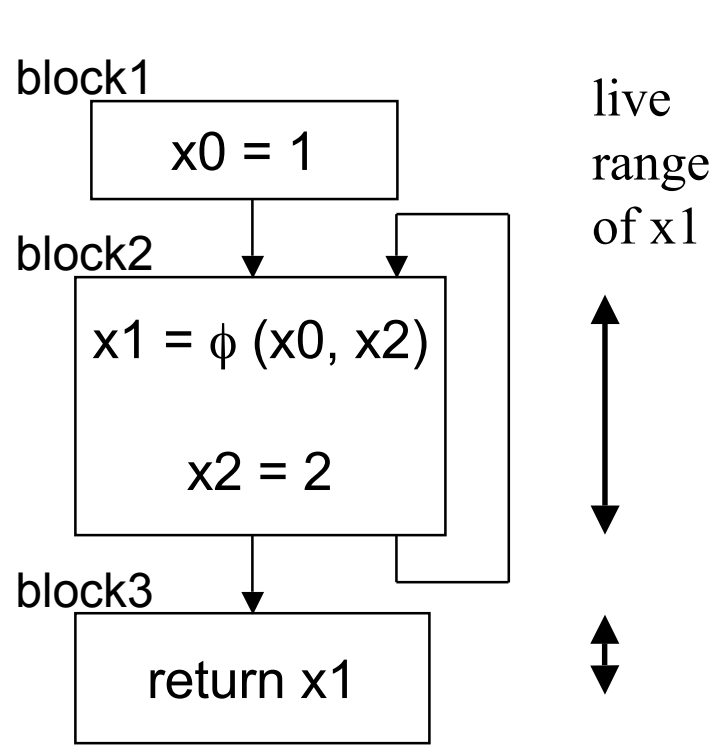
(b) Normal form

# Problems of naïve SSA back translation (lost copy problem)

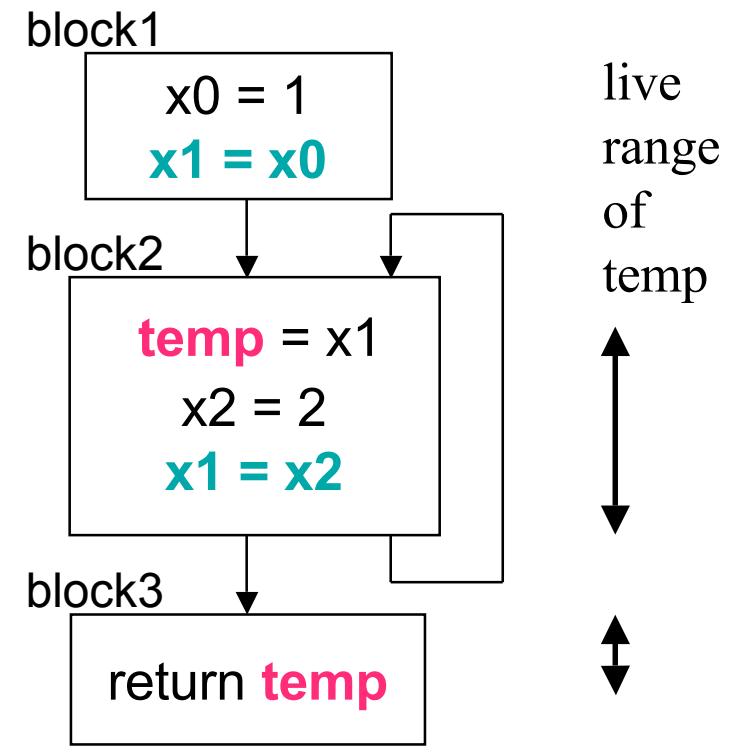


To remedy these problems...

(i) SSA back translation algorithm by Briggs

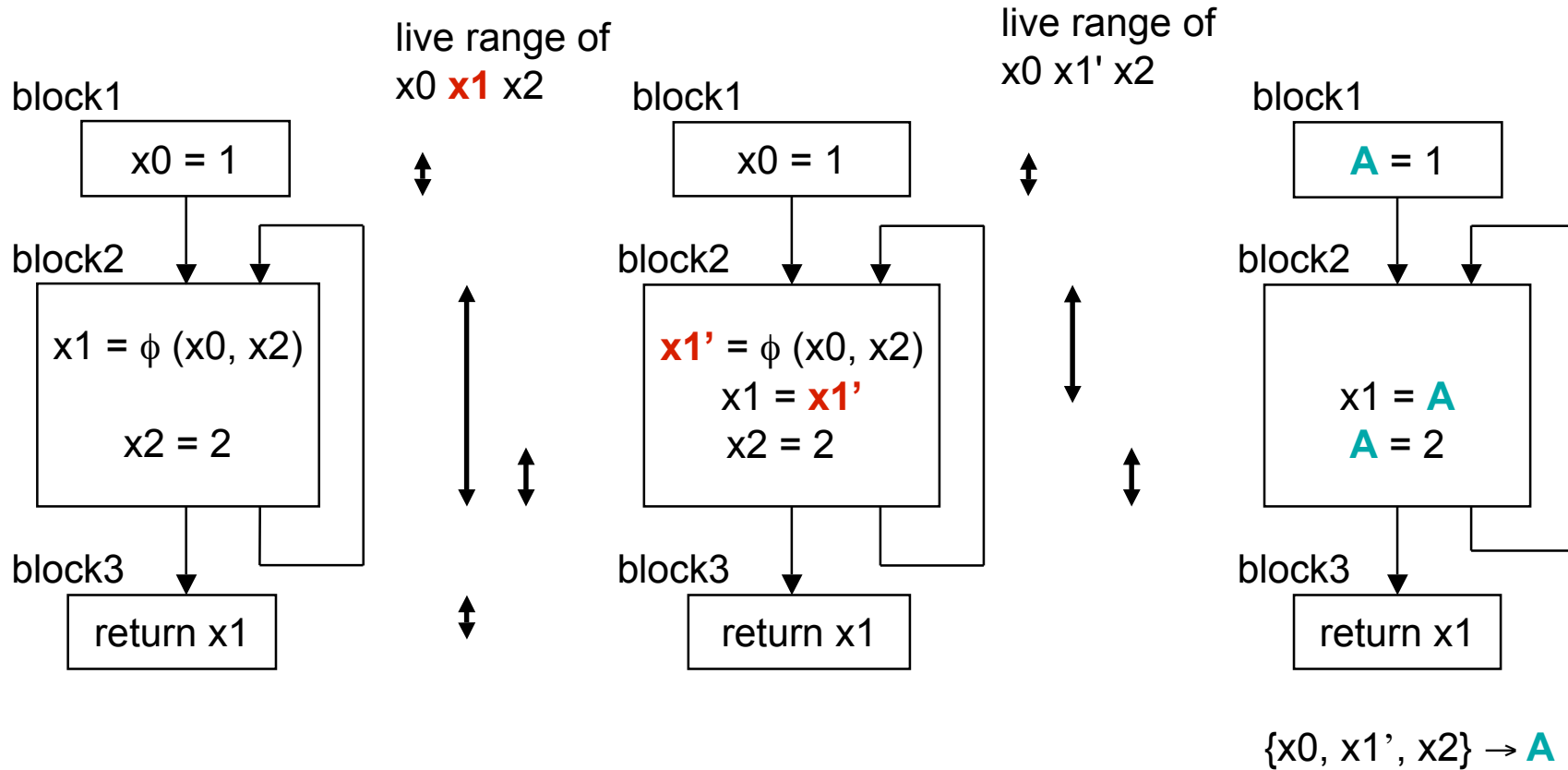


(a) SSA form



(b) normal form after back translation

# (ii) SSA back translation algorithm by Sreedhar



(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]

	<b>SSA form</b>	<b>Briggs</b>	<b>Briggs + Coalescing</b>	<b>Sreedhar</b>
<b>Lost copy</b>	0	3	1 [1]	1 [1]
<b>Simple ordering</b>	0	5	2 [2]	2 [2]
<b>Swap</b>	0	7	5 [5]	3 [3]
<b>Swap-lost</b>	0	10	7 [7]	4 [4]
<b>do</b>	0	9	6 [4]	4 [2]
<b>fib</b>	0	4	0 [0]	0 [0]
<b>GCD</b>	0	9	5 [2]	5 [2]
<b>Selection Sort</b>	0	9	0 [0]	0 [0]
<b>Hige Swap</b>	0	8	3 [3]	4 [4]

# 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 clarified there is no single algorithm which gives optimal result

# 4. A Survey of Compiler Infrastructures

- SUIF \*
- Machine SUIF \*
- Zephyr \*
- Scale
- gcc
- COINS
- Saiki & Gondow

\* National Compiler Infrastructure (NCI) project

# An Overview of the SUIF2 System

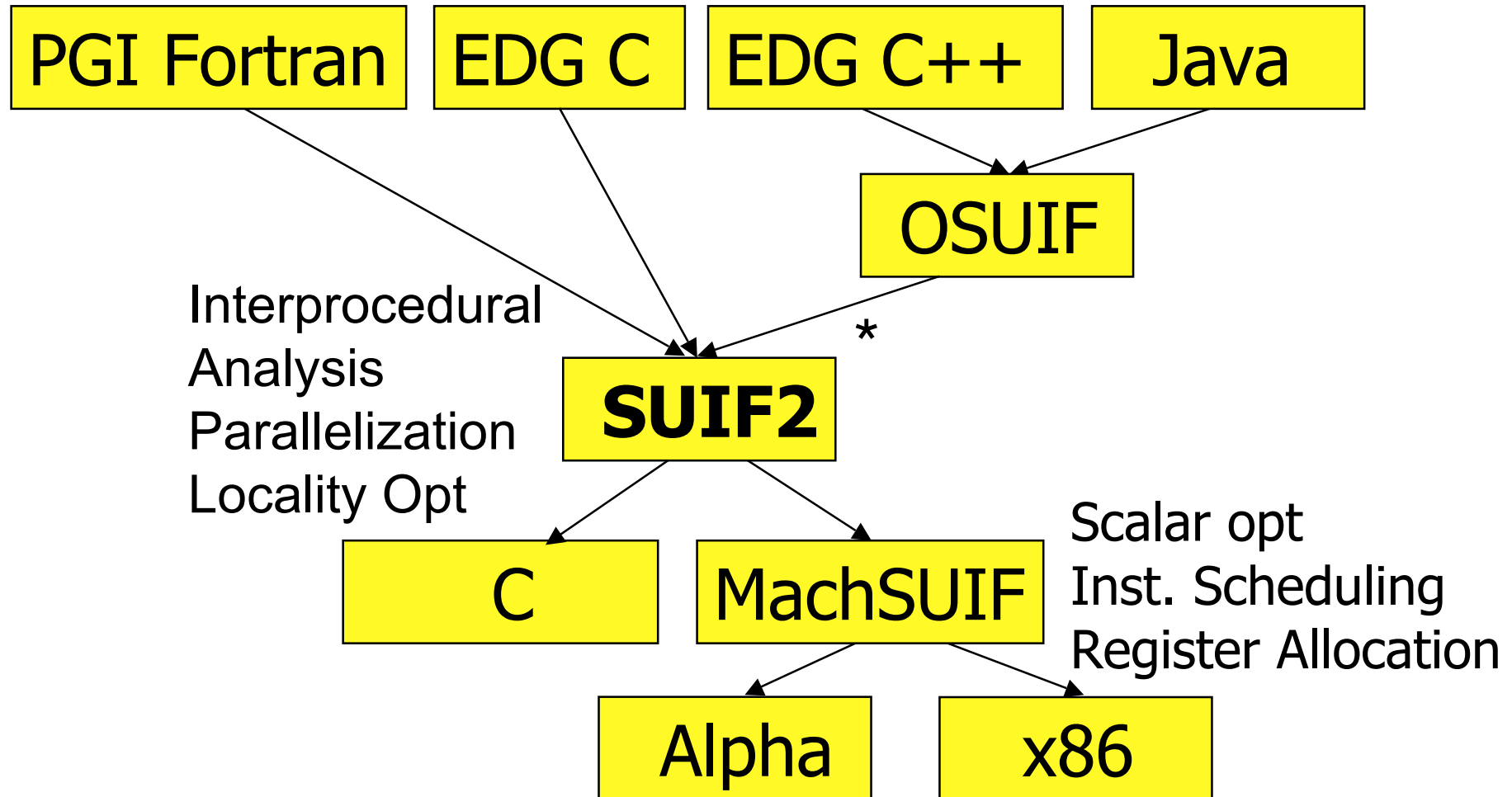
Monica Lam

Stanford University

<http://suif.stanford.edu/>

[PLDI 2000 tutorial]

# The SUIF System



\* C++ OSUIF to SUIF is incomplete

# Overview of SUIF Components (I)

<p>Basic Infrastructure</p> <ul style="list-style-type: none"><li><b>Extensible IR</b> and utilities</li><li><b>Hoof</b>: Suif object specification lang</li><li>Standard IR</li><li>Modular compiler system</li><li>Pass submodule</li><li>Data structures (e.g. hash tables)</li></ul>	<p>FE: PGI Fortran, EDG C/C++, Java SUIF1 / SUIF2 translators, S2c Interactive compilation: <b>suifdriver</b> Statement dismantlers SUIF IR consistency checkers Suifbrowser, TCL visual shell Linker</p>
<p>Object-oriented Infrastructure</p> <ul style="list-style-type: none"><li><b>OSUIF</b> representation</li></ul>	<p>Java OSUIF -&gt; SUIF lowering object layout and method dispatch</p>
<p>Backend Infrastructure</p> <ul style="list-style-type: none"><li><b>MachSUIF</b> program representation</li><li>Optimization framework</li></ul>	<p>Scalar optimizations common subexpression elimination deadcode elimination peephole optimizations Graph coloring register allocation Alpha and x86 backends</p>

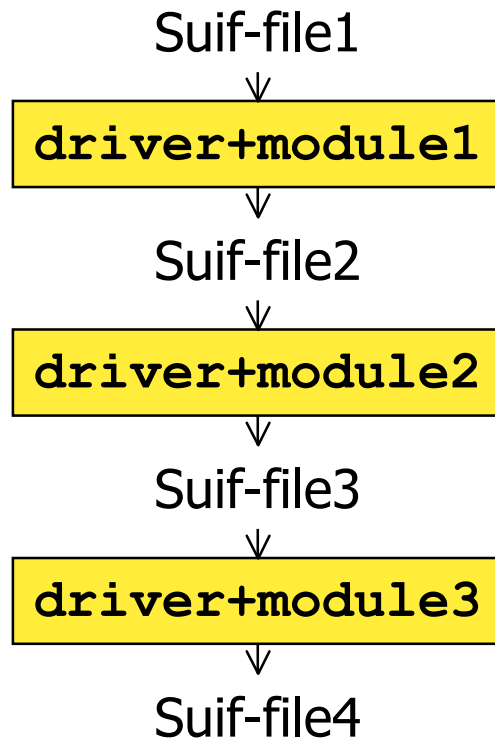
# Overview of SUIF Components (II)

<p>High-Level <b>Analysis Infrastructure</b> Graphs, sccs Iterated dominance frontier Dot graph output</p>	<p>Intraprocedural analyses copy propagation deadcode elimination <b>Steensgaard's alias analysis</b> Call graph</p>
<p>Region framework <b>Interprocedural analysis</b> framework</p>	<p>Control flow graphs Interprocedural region-based analyses: <b>array dependence &amp; privatization</b> <b>scalar reduction &amp; privatization</b> <b>Interprocedural parallelization</b></p>
<p>Presburger arithmetic (omega) Farkas lemma Gaussian elimination package</p>	<p><b>Affine partitioning for parallelism &amp; locality unifies:</b> unimodular transform (interchange, reversal, skewing) fusion, fission statement reindexing and scaling Blocking for nonperfectly nested loops</p>

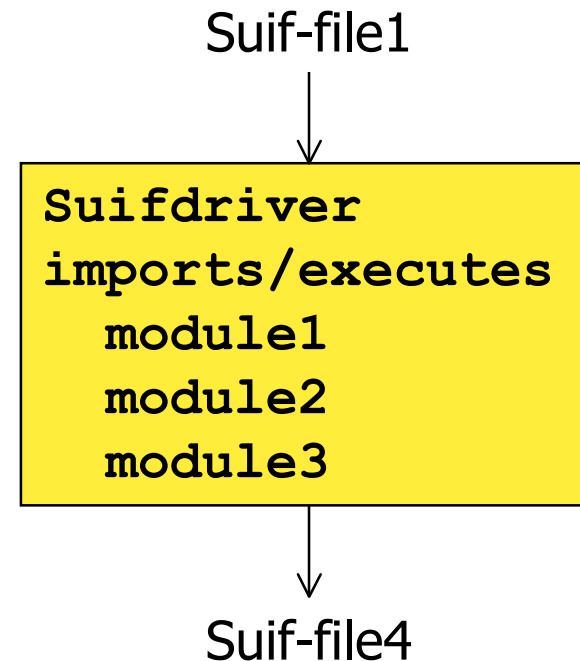
# Memory/Memory vs File/File Passes

## COMPILER

A series of stand-alone programs



A driver that imports & applies modules to program in memory



# Machine SUIF

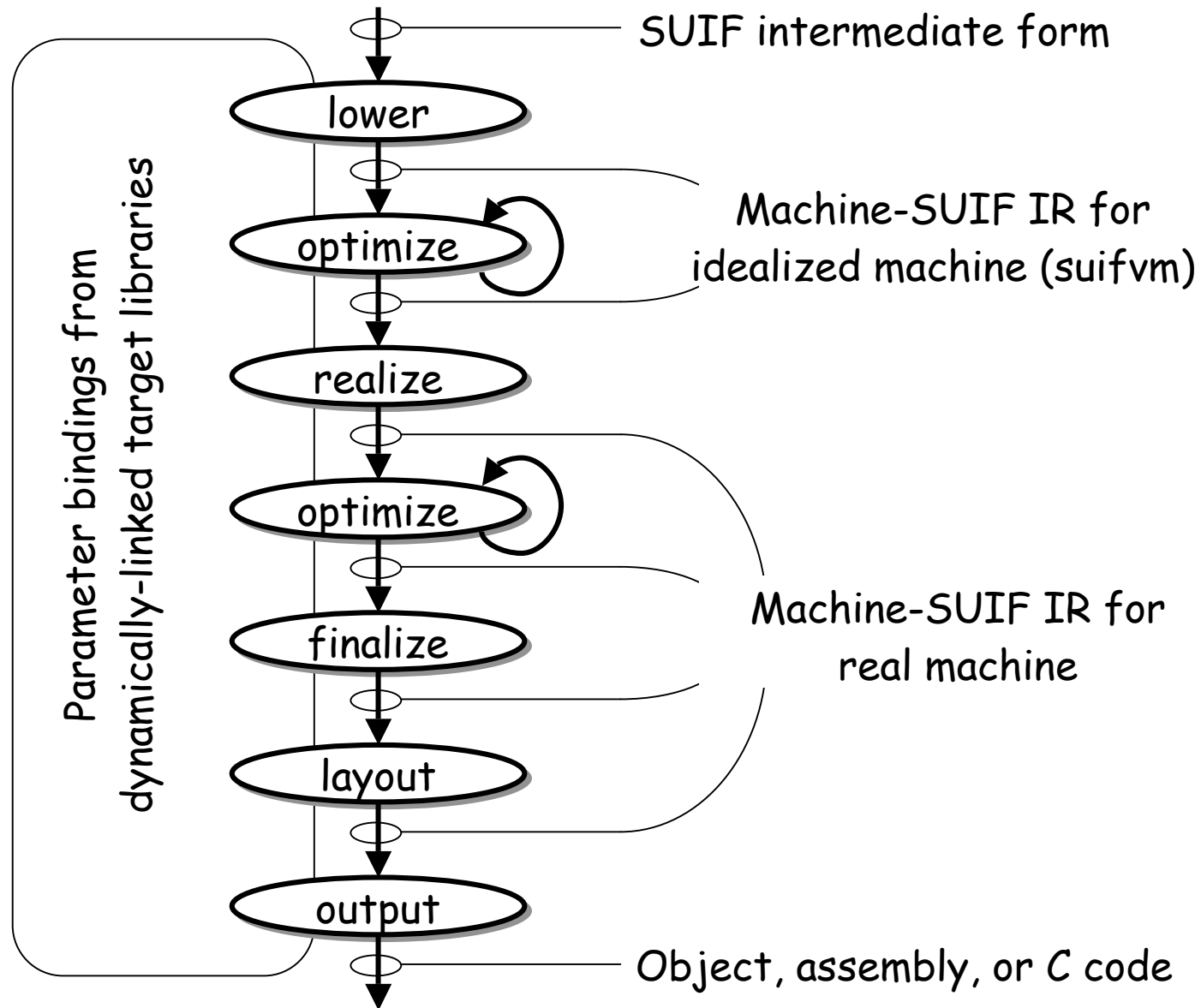


Michael D. Smith

Harvard University  
Division of Engineering and Applied  
Sciences

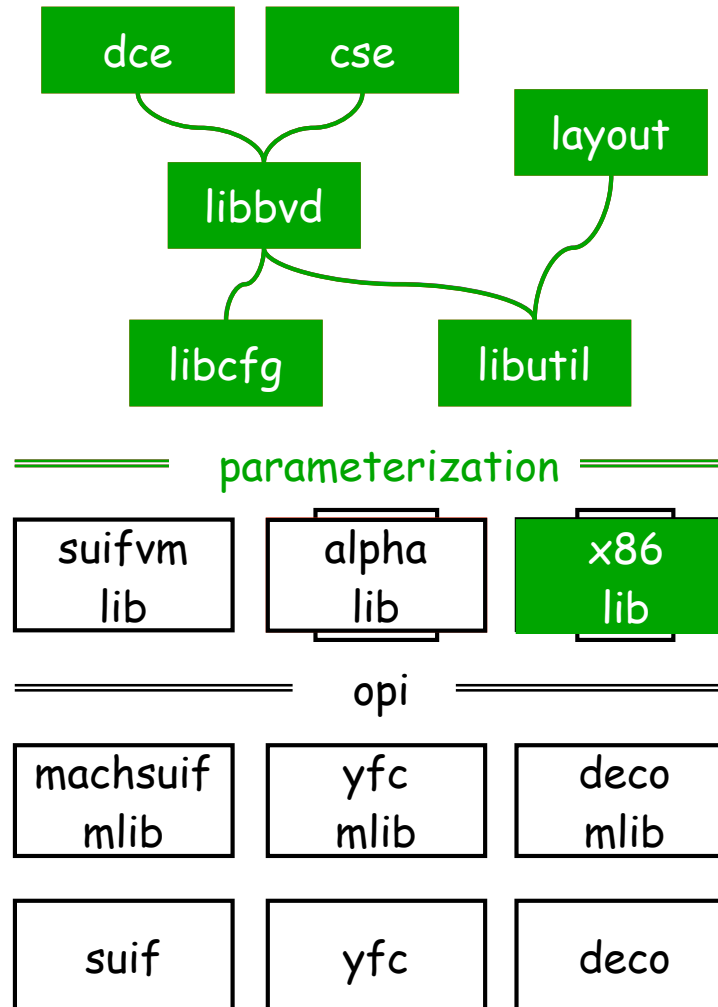
June 2000

# Typical Backend Flow



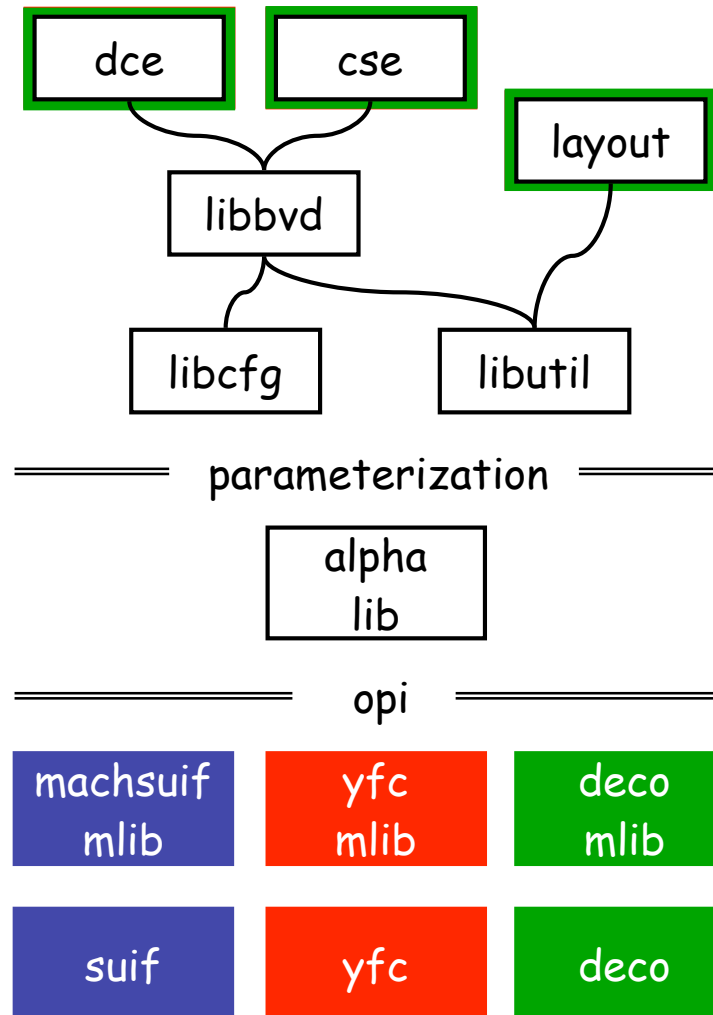
# Target Parameterization

- Analysis/optimization passes written without direct encoding of target details
- Target details encapsulated in OPI functions and data structures
- Machine-SUIF passes work without modification on disparate targets

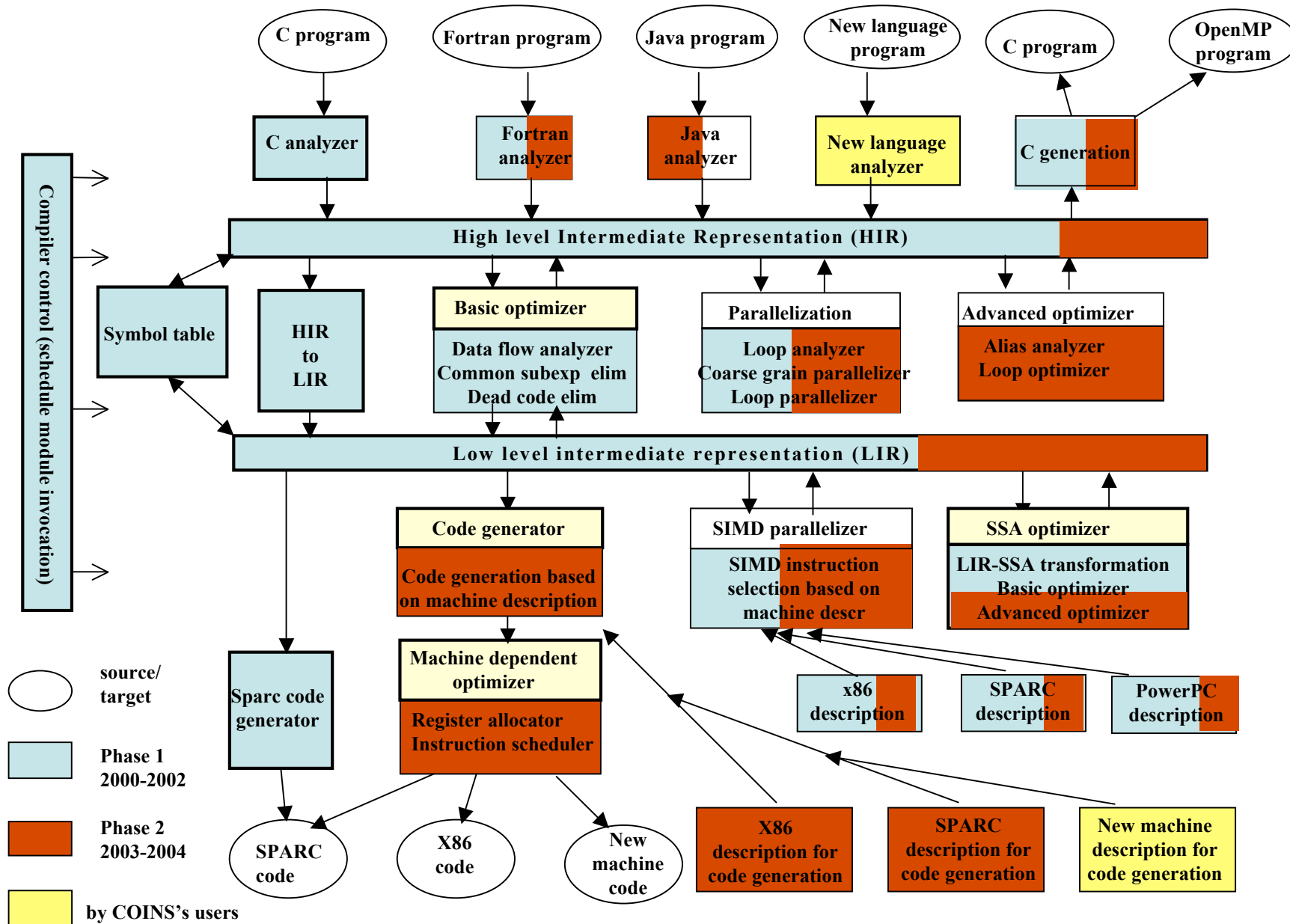


# Substrate Independence

- Optimizations, analyses, and target libraries are substrate-independent
- Machine SUIF is built on top of SUIF
- You could replace SUIF with Your Favorite Compiler
- Deco project at Harvard uses this approach



# Overall structure of COINS



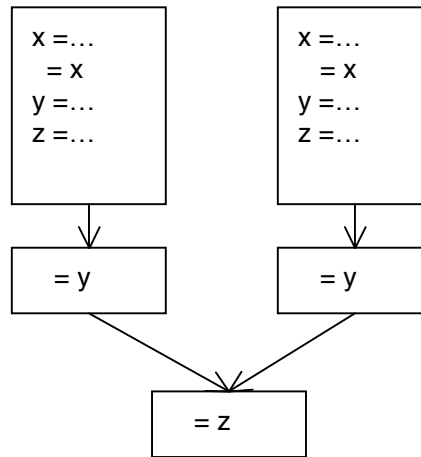
# Features of COINS

- Multiple source languages
- Multiple target architectures
- HIR: abstract syntax tree with attributes
- LIR: register transfer level with formal specification
- Enabling source-to-source translation and application to software engineering
- Scalar analysis & optimization (in usual form and in SSA form)
- Basic parallelization (e.g. OpenMP)
- SIMD parallelization
- Code generators generated from machine description
- Written in Java (early error detection), publicly available

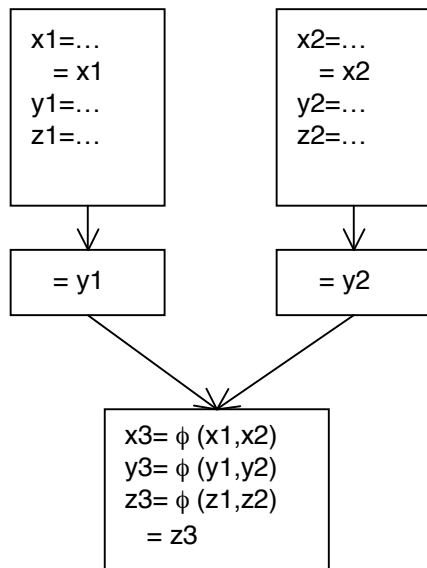
[<http://www.coins-project.org/>]



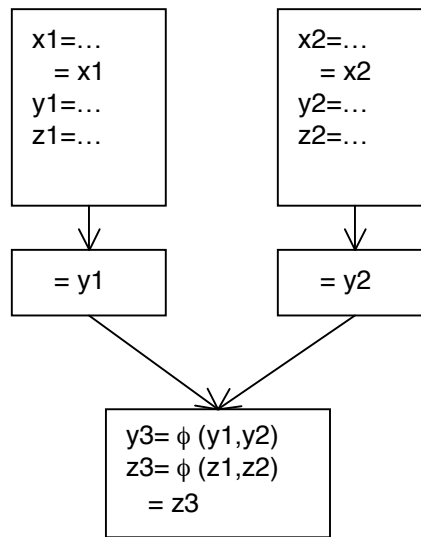
# Translating into SSA form (SSA translation)



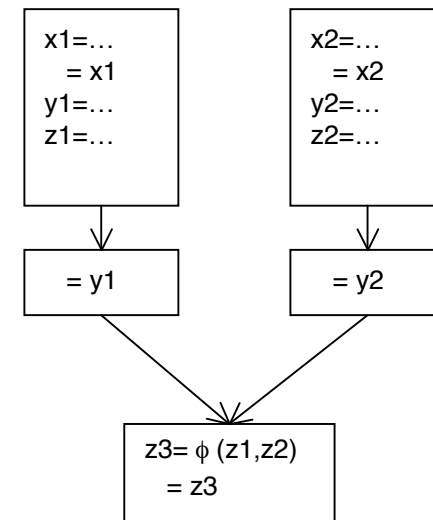
Normal form



Minimal SSA form



Semi-pruned SSA form



Pruned SSA form

# Previous work:

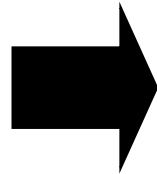
## SSA form in compiler infrastructure

- 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 code like in COINS.
- GCC: some attempts but experimental
- Only COINS will have full support of SSA form as a compiler infrastructure

# Example of a Hoof Definition

hoof

```
concrete New  
{ int x; }
```



C++

```
class New : public SuifObject  
{  
    public:  
    int get_x();  
    void set_x(int the_value);  
    ~New();  
    void print(...);  
    static const Lstring get_class_name();  
    ...  
}
```

- Uniform data access functions (get\_ & set\_)
- Automatic generation of meta class information etc.

# HIR (high-level intermediate representation)

HIR (abstract syntax tree with attributes)

Input program

```
for (i=0; i<10; i=i+1) {  
    a[i]=i;  
    ...  
}
```

```
(for  
  (assign <var i int> <const 0 int>)  
  (cmpLT <var i int> <const 10 int>)  
  (block  
    (assign  
      (subs <var a <VECT 10 int>>  
        <var i int>)  
      <var i int>)  
    ....  
  )  
  (assign  
    <var i int>  
    (add <var i int> <const i int>)  
  ) )
```

# LIR (low-level intermediate representation)

Source program

```
for (i=0; i<10; i=i+1) {  
    a[i]=i; ...  
}
```

LIR

```
(set (mem (static (var i))) (const 0))  
(labeldef _lab5)  
(jumpc (tstlt (mem (static (var i)))  
              (const 10))  
       (list (label _lab6) (label _lab4))))  
(labeldef _lab6)  
(set (mem (add (static (var a))  
              (mul (mem (static (var i)))  
                  (const 4))))  
     (mem (static (var i))))  
... ..
```