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

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

1: \( a = x + y \)
2: \( a = a + 3 \)
3: \( b = x + y \)

(b) SSA form

1: \( a_1 = x_0 + y_0 \)
2: \( a_2 = a_1 + 3 \)
3: \( b_1 = x_0 + y_0 \)

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

1: \(a = x + y\)
2: \(a = a + 3\)
3: \(b = x + y\)

(b) SSA form

1: \(a_1 = x_0 + y_0\)
2: \(a_2 = a_1 + 3\)
3: \(b_1 = x_0 + y_0\)

(c) After SSA form optimization

1: \(a_1 = x_0 + y_0\)
2: \(a_2 = a_1 + 3\)
3: \(b_1 = a_1\)

(d) Optimized normal form

1: \(a_1 = x_0 + y_0\)
2: \(a_2 = a_1 + 3\)
3: \(b_1 = a_1\)

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

L1 \( x = 1 \)  
L2 \( x = 2 \)  
L3 \( \ldots = x \)

(b) SSA form

L1 \( x1 = 1 \)  
L2 \( x2 = 2 \)  
L3 \( x3 = \phi (x1:L1, x2:L2) \)  
\( \ldots = x3 \)
Translating back from SSA form
(SSA back translation)

(a) SSA form

L1

x1 = 1

L3

x3 = φ (x1:L1, x2:L2)
... = x3

(b) Normal form

L1

x1 = 1
x3 = x1

L2

x2 = 2
x3 = x2

L3

... = x3
1. SSA form module in the COINS compiler infrastructure
COINS compiler infrastructure

- C frontend
- Fortran frontend
- New language frontend
- C generation

High Level Intermediate Representation (HIR)

- HIR to LIR
- Basic analyzer & optimizer
- Basic parallelizer
- Advanced optimizer

Low Level Intermediate Representation (LIR)

- Code generator
- SSA optimizer
- SIMD parallelizer

Platforms:
- SPARC
- x86
- New machine
- C
SSA optimization module in COINS

Low level Intermediate Representation (LIR)

Source program

Code generation

Object code

LIR to SSA translation (3 variations)

LIR in SSA

SSA basic optimization:
- com subexp elimination
- copy propagation
- cond const propagation
- dead code elimination

Optimized LIR in SSA

SSA to LIR back translation (2 variations) + 2 coalescing

transformation on SSA:
- copy folding
- dead phi elimination
- edge splitting

12,000 lines

Transformations and variations are applied throughout the SSA optimization process.
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)

- No. of nodes of control flow graph
- Translation time (milli sec)

Graph showing the translation time (in milliseconds) for Cytron and Sreedhar's methods as a function of the number of nodes in the control flow graph.
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

\[ x_0 = 1 \]

\[ x_1 = \phi(x_0, x_2) \]

\[ x_2 = 2 \]

\[ \text{return } x_1 \]

\[ x_0 = 1 \]

\[ x_1' = \phi(x_0, x_2) \]

\[ x_1 = x_1' \]

\[ x_2 = 2 \]

\[ \text{return } x_1 \]

\[ A = 1 \]

\[ x_1 = A \]

\[ A = 2 \]

\[ \{x_0, x_1', x_2\} \rightarrow A \]
Empirical comparison of SSA back translation

<table>
<thead>
<tr>
<th>No. of copies [no. of copies in loops]</th>
<th>SSA form</th>
<th>Briggs</th>
<th>Briggs + Coalescing</th>
<th>Sreedhar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lost copy</td>
<td>0</td>
<td>3</td>
<td>1 [1]</td>
<td>1 [1]</td>
</tr>
<tr>
<td>Swap</td>
<td>0</td>
<td>7</td>
<td>5 [5]</td>
<td>3 [3]</td>
</tr>
<tr>
<td>fib</td>
<td>0</td>
<td>4</td>
<td>0 [0]</td>
<td>0 [0]</td>
</tr>
<tr>
<td>GCD</td>
<td>0</td>
<td>9</td>
<td>5 [2]</td>
<td>5 [2]</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>0</td>
<td>9</td>
<td>0 [0]</td>
<td>0 [0]</td>
</tr>
</tbody>
</table>
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

- PGI Fortran
- EDG C
- EDG C++
- Java

- Interprocedural Analysis
- Parallelization
- Locality Opt

- OSUIF
- SUIF2

- C
- MachSUIF

- Alpha
- x86

* C++ OSUIF to SUIF is incomplete
## Overview of SUIF Components (I)

<table>
<thead>
<tr>
<th>Basic Infrastructure</th>
<th>Backend Infrastructure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Extensible IR and utilities</strong></td>
<td><strong>MachSUIF program representation</strong></td>
</tr>
<tr>
<td><strong>Hoof</strong>: Suiif object specification lang</td>
<td><strong>Optimization framework</strong></td>
</tr>
<tr>
<td>Standard IR</td>
<td><strong>Scalar optimizations</strong></td>
</tr>
<tr>
<td>Modular compiler system</td>
<td>common subexpression elimination</td>
</tr>
<tr>
<td>Pass submodule</td>
<td>deadcode elimination</td>
</tr>
<tr>
<td>Data structures (e.g. hash tables)</td>
<td>peephole optimizations</td>
</tr>
<tr>
<td></td>
<td>Graph coloring register allocation</td>
</tr>
<tr>
<td></td>
<td>Alpha and x86 backends</td>
</tr>
</tbody>
</table>

### Functional Elements

**FE**: PGI Fortran, EDG C/C++, Java
SUIF1 / SUIF2 translators, S2c
Interactive compilation: suifdriver
Statement dismantlers
SUIF IR consistency checkers
Suifbrowser, TCL visual shell
Linker

**Object-oriented Infrastructure**

**OSUIF** representation

Java OSUIF -> SUIF lowering
object layout and method dispatch

**Backend Infrastructure**

**MachSUIF** program representation

Optimization framework
## Overview of SUIF Components (II)

<table>
<thead>
<tr>
<th>High-Level Analysis Infrastructure</th>
<th>Intraprocedural analyses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphs, sccs</td>
<td>copy propagation</td>
</tr>
<tr>
<td>Iterated dominance frontier</td>
<td>deadcode elimination</td>
</tr>
<tr>
<td>Dot graph output</td>
<td>Steensgaard’s alias analysis</td>
</tr>
<tr>
<td></td>
<td>Call graph</td>
</tr>
</tbody>
</table>

| Region framework                                     | Control flow graphs                                                                      |
| Interprocedural analysis framework                   | Interprocedural region-based analyses:                                                  |
|                                                      |   - array dependence & privatization                                                     |
|                                                      |   - scalar reduction & privatization                                                     |
|                                                      |   - Interprocedural parallelization                                                     |

| Presburger arithmetic (omega)                         | Affine partitioning for parallelism & locality                                           |
| Farkas lemma                                          | unifies:                                                                                  |
| Gaussian elimination package                          |   - unimodular transform                                                                 |
|                                                      |   (interchange, reversal, skewing)                                                       |
|                                                      |   - fusion, fission                                                                       |
|                                                      |   - statement reindexing and scaling                                                     |
|                                                      |   - Blocking for nonperfectly nested loops                                               |
Memory/Memory vs File/File Passes

COMPILER

A series of stand-alone programs

- Suif-file1
  - driver+module1
    - Suif-file2
    - driver+module2
      - Suif-file3
      - driver+module3
        - Suif-file4

A driver that imports & applies modules to program in memory

- Suif-file1
  - Suifdriver imports/executes
    - module1
    - module2
    - module3
  - Suif-file4
Typical Backend Flow

SUIF intermediate form

Machine-SUIF IR for idealized machine (suifvm)

Machine-SUIF IR for real machine

Object, assembly, or C code

Parameter bindings from dynamically-linked target libraries
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.

Diagram:
- dce
- cse
- libbvd
- libcfg
- libutil
- layout
- parameterization
- alpha
- lib
- opi
- machsuir
- mlib
- yfc
- mlib
- deco
- mlib
- suif
- yfc
- deco
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)

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

- Uniform data access functions (get_ & set_)
- Automatic generation of meta class information etc.

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

concrete New
{ int x; }
HIR (high-level intermediate representation)

Input program

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

HIR (abstract syntax tree with attributes)

```c
(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

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

LIR

```lisp
(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)))
  ...
```