Efficiently Computing the Static Single Information Form

Jeremy Singer

September 27, 2002

1 Introduction

The static single assignment (SSA) form [CFR$^+$91] is now an established compiler intermediate representation. Sun's HotSpot [Sun99] and Microsoft's Marmot [FKR$^+$00] are two recent examples of SSA-based compilers. The beauty of SSA form is that each variable has only one definition site in the program. This eliminates unrelated uses of the same variable name from the original source code, which simplifies dataflow analyses and optimisations

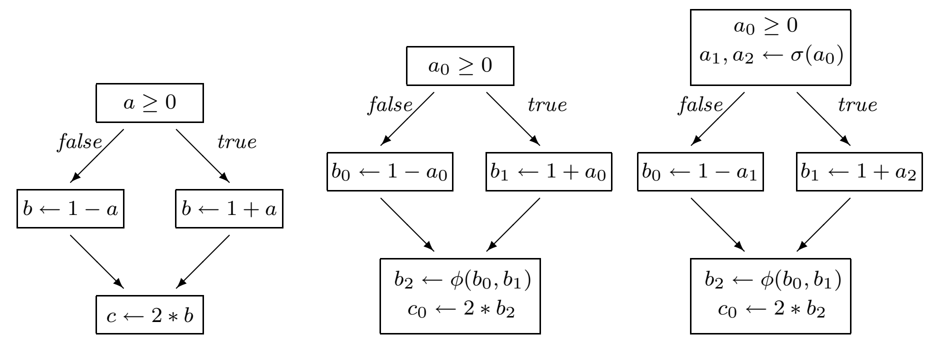
The most notable feature of SSA form is the ϕ-node, a pseudo-assignment function which is used to combine multiple incoming variable definitions at control flow merge points, thus (albeit rather artificially) preserving the property that each variable has a unique definition site.

Construction of SSA form typically proceeds in two phases:

1. identification of the points where ϕ-nodes are needed.
2. variable renaming to create a unique name for each definition point.

The static single information (SSI) form [Ana99] builds upon SSA form, with the addition of a single construct, the σ-node, which is used to separate variables at control flow split points. In this way we differentiate between uses of a variable in separate arms of a conditional branch. SSI form allows us to store data flow information in a sparse manner, *i.e.* once per variable, rather than once per point in program where a variable is used. This is justified since information can only be gleaned about a variable at certain places in the program:

1. from assignments (where we are creating new facts about the value held by a variable)
2. from control flow merges (where we are obtaining a disjunction of possible facts about the value held by a variable)
3. from conditional tests (where we are attempting to discover facts about the value held by a variable).



(a) Control flow graph *(b) SSA form* *(c) SSI form*

Figure 1: One program, three representations

ϕ-nodes deal with item 2. σ-nodes deal with item 3 .

SSI form was originally described by Ananian [Ana99]. He states that "the principal benefits of using SSI form are the ability to do predicated and backward dataflow analyses efficiently." He gives several examples including *very busy expressions analysis* and *sparse predicated typed constant propagation*. Indeed, SSI form can be applied to a wide range of problems, [RR00, SBA00].

By its very nature, SSI form is naturally suited to compilers for instruction set architectures that support predicated execution. Two of these architectures (IA-64 and StrongARM) are rapidly gaining popularity.

Minimal and pruned SSI forms parallel their SSA counterparts. They are clearly defined in [Ana99]. Minimal SSI form has the smallest number of ϕ- and σ-nodes such that the conditions for SSI form are satisfied. Pruned SSI form is the minimal form with any unused ϕ- and σ-nodes deleted.

Construction of SSI form takes place in two phases, in the same manner as for SSA form:

1. identification of the points where ϕ- and σ-nodes are needed.

2. variable renaming to create a valid SSI form program.

In this paper, we concentrate on the placement algorithm, rather than the renaming algorithm. Ananian presents a linear-time renaming algorithm in his thesis [Ana99].

2 Related Work

The MIT Flex compiler [Fle98] is based upon the SSI algorithms invented by Ananian [Ana99]. The Flex compiler is a compiler for Java, written in Java. As far as we are aware, this is the only compiler that uses SSI as its intermediate representation.

The Dependence Flow Graph (DFG) of Johnson and Pingali [JP93] is very similar to SSI form. DFG `merge` nodes correspond to the ϕ-nodes of SSA (as noted in [JP93]) and DFG `switch` nodes correspond to the σ-nodes of SSI. Ananian's SSI construction algorithm owes a great deal to the work of Johnson and Pingali.

The Program Dependence Web (PDW) of Ballance, Maccabe and Ottenstein [BMO90] is another intermediate representation which is similar to SSI form. PDW γ-, μ- and η-nodes correspond to various kinds of SSA ϕ-nodes. PDW `switch` nodes correspond to the σ-nodes of SSI.

The papers on the DFG [JP93] and the PDW [BMO90] both claim to have linear time construction algorithms. It should be possible to translate from either of the representations into SSI form, however we assume that Ananian's method (which is also linear according to [Ana99]) should be better than either of the other two, since it computes SSI directly rather than via another intermediate form.

3 Original Method

Ananian describes his algorithm for calculating SSI form in great detail, in [Ana99]. However, his description contains slightly bemusing statements such as: "Our algorithm for placing ϕ- and σ-functions in SSI form is *pessimistic*; that is, we at first assume every node in the control flow graph with input arity larger than one requires a ϕ-function for every variable and every node with out-arity larger than one requires a σ-function for every variable, and then use the program structure tree, liveness information, and unused code elimination to determine safe places to *omit* ϕ- or σ-functions."

3.1 Background

Ananian's construction algorithm begins with a program structure tree of single-entry single-exit (SESE) regions, as described by [JPP93, JPP94].

A SESE region in a graph G is an ordered pair (a, b) of distinct control flow edges a and b where:

1. a dominates b,
2. b postdominates a, and
3. every cycle containing a also contains b and vice-versa.

A program structure tree records the nesting structure of SESE regions in a control flow graph. Each node in this tree represents a SESE region. The parent of a node is the closest containing region and the children of a node are all the regions immediately contained within it.

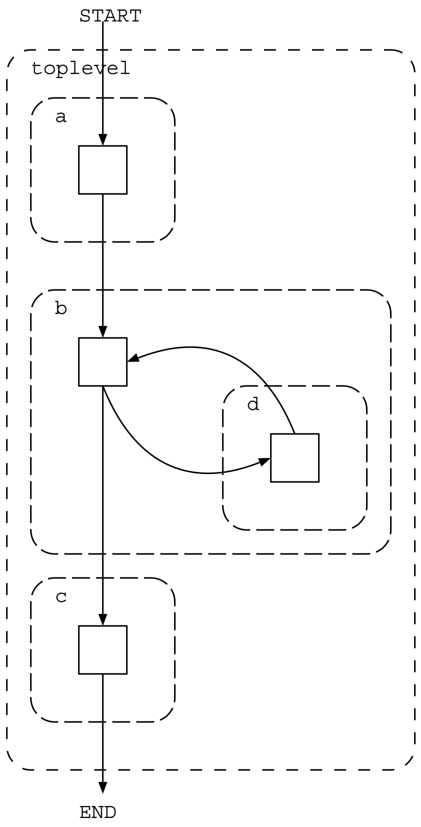


Figure 2: A control flow graph with marked SESE regions

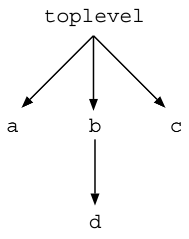


Figure 3: A program structure tree of SESE regions

Figure 2 shows a control flow graph with the SESE regions marked in dashed lines. Figure 3 shows the program structure tree for the same control flow graph.

3.2 Placement algorithm

Ananian's algorithm is presented in figure 4. It performs a post-order traversal of the program structure tree for each variable v. That is to say, it visits nested child SESE regions before visiting a parent region. It determines which regions require ϕ- or σ-nodes for variable v. A region requires a ϕ-node (or σ-node) if it contains a definition (or use) of variable v or if a ϕ- or σ-node has already been placed in a child of this region.

The MaybeLive function should give a conservative approximation to liveness. MaybeLive(v, n) should return `true` when v may possibly be live at node n. In the simplest case, MaybeLive can be programmed always to return `true`. This causes the placement algorithm to produce minimal SSI form. A more pruned SSI form may be obtained by a more sophisticated implementation of the MaybeLive function.

3.3 Complexity

Ananian [Ana99] claims that his SSI computation algorithm is linear. Constructing the program structure tree takes linear time in the size of the program [JPP93, JPP94]. The placement algorithm in figure 4 makes a single pass through the program structure tree, thus it too is linear.

4 Alternative Method

Our method is, in stark contrast to Ananian's method, an optimistic approach to the problem. We initially assume that no ϕ- or σ-nodes are needed, and then analyse the control flow graph to determine whether any nodes need to be inserted. Thus we apply a ϕ-node placement pass, followed by a σ-node placement pass, and then iterate to a fixed point. The algorithm is presented in figure 5.

The iteration is necessary because placing σ-nodes may then require extra ϕ-nodes to be inserted, and vice versa.

We follow the classical method of using dominance frontiers to discover where ϕ- and σ-nodes are needed. Thus our place-ϕ-functions algorithm is identical to the standard SSA ϕ-node placement algorithm [CFR$^+$91]. The place-σ-functions algorithm is the mirror image of place-ϕ-functions. It has the same shape but it is exactly the opposite. It tracks variable uses rather than definitions, it uses reverse dominance frontiers rather than standard dominance frontiers, it inserts σ-nodes at the end of basic blocks rather than ϕ-nodes at the beginning of basic blocks.

```
Place($G$: CFG) =
    let $r$ be the top-level region for $G$
    for each variable $v$ in $G$
        PlaceOne($r$, $v$, false) /* place $\phi$-functions */
        PlaceOne($r$, $v$, true) /* place $\sigma$-functions */

PlaceOne($r$: region, $v$: variable, $ps$: boolean): boolean =
    /* post-order traversal */
    $flag \longleftarrow$ false
    for each child region $r'$
        if PlaceOne($r'$, $v$, $ps$)
            $flag \longleftarrow$ true

    for each node $n$ in region $r$ not contained in a child region
        if $ps$ is false and $n$ contains a definition of $v$
            $flag \longleftarrow$ true
        if $ps$ is true and $n$ contains a use of $v$
            $flag \longleftarrow$ true

    /* add $\phi$-/$\sigma$-functions to merges/splits where $v$ may be live */
    if $flag$ = true
        for each node $n$ in region $r$ not contained in a child region
            if MaybeLive($v$, $n$) = true
                if $ps$ is false and the input arity of $n$ exceeds 1
                    place a $\phi$-function for $v$ at $n$
                if $ps$ is true and the output arity of $n$ exceeds 1
                    place a $\sigma$-function for $v$ at $n$

    return $flag$
```

Figure 4: Ananian's SSI node placement algorithm

```
while ($change$)
    $change \longleftarrow$ false
    place-$\phi$-functions
    if ($change$)
        place-$\sigma$-functions
```

Figure 5: Alternative SSI node placement algorithm

```
place-$\phi$-functions =
    for each node $n$
        for each variable $a \in A_{orig}[n]$
            $\mathit{defsites}[a] \longleftarrow \mathit{defsites}[a] \cup \{n\}$
    for each variable $a$
        $W \longleftarrow \mathit{defsites}[a]$
        while $W$ not empty
            remove some node $n$ from $W$
            for each $y \in DF[n]$
                if $y \notin A_{\phi}[a]$
                    insert statement $a \longleftarrow \phi(a, a, ..., a)$ at the
                        top of block $y$, where the $\phi$-function has
                        as many arguments as $y$ has predecessors
                    $A_{\phi}[a] \longleftarrow A_{\phi}[a] \cup \{y\}$
                    if $a \notin A_{orig}[y]$
                        $W \longleftarrow W \cup \{y\}$
```

Figure 6: ϕ-node placement algorithm

4.1 Explanations

We follow the notational conventions of Appel [App98] for presenting our ϕ- and σ-node placement algorithms, in figures 6 and 7 respectively.

$A_{orig}[n]$ contains the set of variables that are assigned a value at node n. $\mathit{defsites}[a]$ is initialised to contain the set of nodes that assign a value to variable a. W is a work-list of nodes that need to be processed. $DF[n]$ is the set of nodes in the dominance frontier of node n. See [CFR+91, App98] for more details. $A_{\phi}[a]$ is the set of nodes that contain a ϕ-node for variable a.

$U_{orig}[n]$ contains the set of variables that are used at node n. $\mathit{usesites}[a]$ is initialised to contain the set of nodes that use the value of variable a. $RDF[n]$ is the set of nodes in the reverse dominance frontier of node n. See [CFR+91, App98] for more details. $A_{\sigma}[a]$ is the set of nodes that contain a σ-node for variable a.

4.2 Complexity

Cytron et al state that their SSA ϕ-node placement algorithm is “linear in practice.” In fact, there are genuine linear time ϕ-node placement algorithms [BP99, SG95] which we might have used instead. Our fixed-point iteration is bounded by the size of the control flow graph. Thus, we say that our placement algorithm is potentially quadratic, however worst-case behaviour is unlikely.

```
place-$\sigma$-functions =
    **for each** node $n$
        **for each** variable $a \in U_{orig}[n]$
            $usesites[a] \longleftarrow usesites[a] \cup \{n\}$
    **for each** variable $a$
        $W \longleftarrow usesites[a]$
        **while** $W$ not empty
            remove some node $n$ from $W$
            **for each** $y$ in $RDF[n]$
                **if** $y \notin A_\sigma[a]$
                    insert the statement $a, a, ..., a \longleftarrow \sigma(a)$
                        at the bottom of block $y$, where the
                        $\sigma$-function has as many results
                        as $y$ has successors
                    $A_\sigma[a] \longleftarrow A_\sigma[a] \cup \{y\}$
                    **if** $a \notin U_{orig}[y]$
                        $W \longleftarrow W \cup \{y\}$
```

Figure 7: σ-node placement algorithm

5 Implementation

We have implemented both of the placement algorithms outlined above, in C++ using the Machine-SUIF compiler framework from Harvard [Smi96]. It is relatively easy to write Machine-SUIF passes which operate at the control flow graph level.

We implemented Ananian's algorithm from scratch. The pass was written in just over 3.5 kLOC. Our alternative placement algorithm is based on the existing Machine-SUIF SSA pass [Hol01b]. We only needed to make slight modifications and extensions to suit our purpose.

One novel feature of our Machine-SUIF passes is that they produce SSI code in semi-pruned form. The original Machine-SUIF SSA pass had an option to build semi-pruned form, and we retain this. Semi-pruned form was introduced by Briggs et al [BCHS98]. It is based on the observation that many of the variables in a program being converted to SSI form are local to one basic block. (This is especially true of compiler generated temporary variables.) Such variables will never require ϕ- or σ-nodes, so we do not take them into consideration when generating SSI form. In this way, semi-pruned form avoids many of the redundant nodes present in minimal form without attempting all the expensive dataflow analysis necessary for fully pruned form.

program	*size/KB*	*original*			*alternative*		
		time/s	ϕ	σ	*time/s*	ϕ	σ
164.gzip	1936	7.98	6145	10246	7.07	1526	2749
175.vpr	4964	18.01	11470	25240	15.43	5395	8553
176.gcc	72936	2336.63	2741414	5417750	650.15	258668	802049
181.mcf	1124	4.05	1108	2057	3.99	473	720
186.crafty	9072	117.10	85403	169570	55.33	12255	42283
197.parser	3876	16.42	18981	34679	13.19	4815	8638
253.perlbmk	15672	98.29	163992	310138	72.45	37343	84411
254.gap	24392	131.87	243069	382266	81.67	38098	77647
255.vortex	19568	80.34	47191	171993	60.98	16702	39791
256.bzip2	984	4.06	5342	7439	3.03	1128	1555
300.twolf	11264	81.60	105951	165943	43.14	15841	25465

Table 1: Summary of results obtained from the two SSI placement algorithms

6 Results

We tested our two SSI placement algorithms using the SPEC CINT 2000 benchmark [Spe00]. We compiled all the C language programs into Machine-SUIF control flow graphs. There is one control flow graph for each procedure. (We were unable to compile a small number of the files, so these are ignored in our analyses.) Then we ran our two placement algorithms on the control flow graphs. A breakdown of our results is presented in table 1. In this table, *program* refers to the CINT 2000 benchmark test under consideration. The *size* recorded is the size of the Machine-SUIF control flow graph files for that program. The *original* pass is Ananian's method, which was described in section 3. The *alternative* pass is our new method, which was described in section 4. We measured the time taken to complete each pass five times, and recorded the median *time*. ϕ is the number of ϕ-nodes placed by each pass, summed over all control flow graphs of the program. Similarly, σ is the number of σ-nodes placed by each pass.

There are a number of interesting observations in our results:

- Our alternative pass is no slower than the Ananian pass in any of the measured cases. In the best case (176.gcc), our method takes just a quarter of the time taken by the original method. The programs in the CINT 2000 benchmark are selected to be representative real-life programs that any respectable optimising compiler should be expected to handle easily.

- Our pass places far fewer ϕ- and σ-nodes than the Ananian pass. In the best case (176.gcc), our method places less than 10% of the ϕ-nodes placed by the original method.

- There are approximately half as many σ-nodes as ϕ-nodes in each case. This is independent of the placement algorithm used.

6.1 Analysis

We presume that Ananian's algorithm places more nodes because of its pessimistic approach to the problem. This is the main contributor to its observed sluggish performance. It is significant that the greatest relative time difference between the two algorithms is in 176.gcc, and this test also produced the greatest relative ϕ-node count difference.

In his thesis [Ana99], Ananian suggests that a dead code elimination pass should occur after the placement algorithm. This may produce more pruned SSI form, but it would take even longer to complete.

7 Future Work

At present, the Harvard Machine-SUIF control flow analysis library calculates dominators using a simple iterative algorithm [Hol01a]. The standard Lengaur-Tarjan dominators algorithm [LT79] would be a more efficient method. We need to plug this gap in the Machine-SUIF libraries.

Our fixed-point placement algorithm recalculates variable definition and use sites for each iteration. This could be improved by incrementally updating the existing information throughout each iteration.

8 Conclusions

We have described an optimistic approach to computing SSI form, which is quite the opposite to what Ananian initially created. When we compare the two placement algorithms as they are set loose on typical contemporary programs, our alternative algorithm is a clear winner in every case.

References

[Ana99] C. Scott Ananian. The static single information form. Master's thesis, Massachusetts Institute of Technology, Sep 1999.

[App98] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[BCHS98] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software—Practice and Experience*, 28(8):859–881, 1998.

[BMO90] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 257–271, 1990.

[BP99] Gianfranco Bilardi and Keshav Pingali. The static single assignment form and its computation. Technical report, Department of Computer Science, Cornell University, Jul 1999. `http://www.cs.cornell.edu/Info/Projects/Bernoulli/papers/ssa.ps`.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[FKR+00] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an optimizing compiler for Java. *Software—Practice and Experience*, 30(3):199–232, 2000.

[Fle98] The Flex compiler infrastructure, 1998. `http://www.flex-compiler.lcs.mit.edu/Harpoon/`.

[Hol01a] Glenn Holloway. The Machine-SUIF control flow analysis library, 2001. `http://www.eecs.harvard.edu/hube/software/nci/cfa.pdf`.

[Hol01b] Glenn Holloway. The Machine-SUIF static single assignment library, 2001. `http://www.eecs.harvard.edu/hube/software/nci/ssa.pdf`.

[JP93] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 78–89, 1993.

[JPP93] Richard Johnson, David Pearson, and Keshav Pingali. Finding regions fast: Single entry single exit and control regions in linear time. Technical Report CTC93TR141, Department of Computer Science, Cornell University, Jul 1993.

[JPP94] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1994.

[LT79] Thomas Lengauer and Robert E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, Jul 1979.

[RR00] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices and accessed memory regions. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 182–195, 2000.

[SBA00] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bidwidth analysis with application to silicon compilation. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 108–120, 2000.

[SG95] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 62–73, Jan 1995.

[Smi96] Michael D. Smith. Extending SUIF for machine-dependent optimizations. In *Proceedings of the First SUIF Compiler Workshop*, pages 14–25, Jan 1996. `http://www.eecs.harvard.edu/machsuif/`.

[Spe00] SPEC CPU2000 benchmark, 2000. `http://www.spec.org`.

[Sun99] Java hotspot, 1999. `http://java.sun.com/products/hotspot/`.