Static Single Assignment Form in the COINS Compiler Infrastructure

Masataka Sassa, Toshiharu Nakaya, Masaki Kohama, Takeaki Fukuoka and Masahito Takahashi

***Abstract*—Static single assignment (SSA) form is a program representation that is becoming increasingly popular in language processors. In SSA form, each use of a variable has a single definition point. This property facilitates program analysis and optimization in compilers. We give some preliminary results concerning the SSA form in COINS, which is a compiler infrastructure recently developed by Japanese institutions. In this paper we present (i) the current status of optimization using SSA form in COINS infrastructure, (ii) a comparison of two major algorithms for translating from normal intermediate form into SSA form, and (iii) a comparison of two major algorithms for translating back from SSA form into normal intermediate form.**

***Index Terms*—Compiler infrastructure, Optimization, Static single assignment form (SSA form)**

I. Introduction

DEVELOPING a good compiler is indispensable for producing high-performance software. However, it is not easy to make a compiler that generates efficient object codes. To reduce the efforts needed to develop high quality compilers, two aspects have been studied.

One attempt involves developing compiler infrastructures. COINS (COmpiler INfraStructure) [7] is one such infrastructure being developed in a research project entitled "Research on common infrastructure for parallelizing compilers". Its development, by Japanese institutions, began in 2000.

Another attempt involves producing a representation designed to be suited for optimizations. *Static single assignment* (*SSA*) *form* is a program representation that is becoming increasingly popular in language processors. In SSA form, each use of a variable has a single definition point. This property facilitates program analysis and optimization in the compiler.

Manuscript received November 15, 2002. This work was supported in part by the Japanese Ministry of Education, Culture, Sports, Science and Technology under Grant "Special Coordination Fund for Promoting Science and Technology" and Grant-in-Aid for Scientific Research (C)(2).

M. Sassa, T. Nakaya and M. Kohama are with Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, Tokyo, 152-8552, Japan (e-mail: {sassa, nakaya8, kohama9}@is.titech.ac.jp).

T. Fukuoka and M. Takahashi are with Kanrikogaku Kenkyusho, Ltd., Tokyo, Japan.

In this paper, we give some preliminary results concerning the SSA form in COINS. We present (i) the current status of optimization using SSA form in the COINS infrastructure, (ii) a comparison of two major algorithms for translating from normal (conventional) intermediate form into SSA form, and (iii) a comparison of two major algorithms for translating back from SSA form into normal intermediate form.

A characteristic feature of the SSA module in COINS is that it provides optimization in SSA form and related utility modules as an infrastructure. This makes it easier for the compiler writer to compare and evaluate various optimization methods and to add new optimization methods and translation utilities in SSA form.

II. The COINS Compiler Infrastructure

COINS is a retargetable compiler infrastructure, aiming at handling multiple source languages and multiple object architectures. It is written in Java. Fig. 1 shows the outline of the COINS compiler infrastructure.

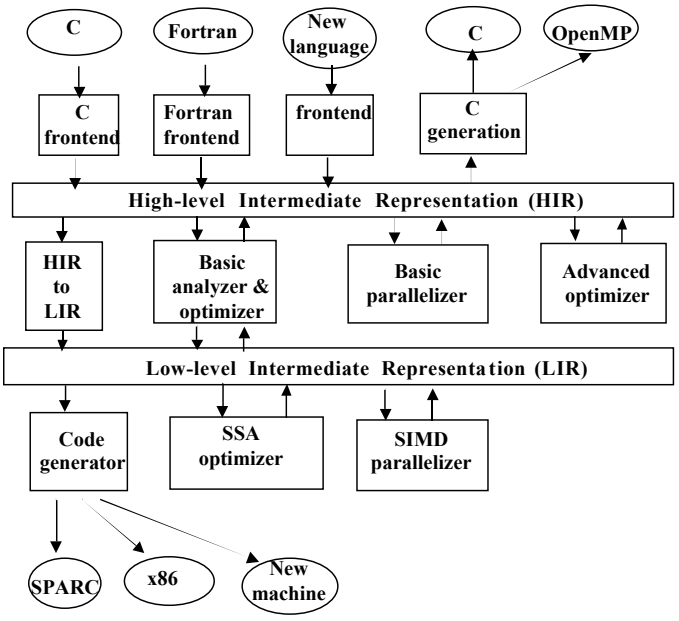


Fig. 1. Outline of the COINS compiler infrastructure

The COINS infrastructure uses High-level Intermediate Representation (HIR) and Low-level Intermediate Representation (LIR). HIR is a data structure holding the source language information. It can be used for optimization based on the source language information and for back generation of a source program. LIR is a low-level intermediate representation in which only the necessary memory references are explicitly described and the remaining variables are in virtual registers. LIR can be used for optimization to the machine architecture.

In the COINS infrastructure, optimization and analysis are called a "pass". The SSA optimization module is one such pass. The compiler writer can easily embed optimization and analysis in COINS by adding passes in the compiler driver, i.e., the main program of the compiler.

At present, the COINS compiler inputs C language programs and outputs assembly language codes for the SPARC processor. In the future, we are planning to accept input in Fortran and a new language, and produce codes for x86 and a new machine as output. As for optimization, COINS currently supports control flow and dataflow analyses in HIR and LIR, and basic optimization, such as common subexpression elimination.

III. Static Single Assignment Form

The *static single assignment form* (*SSA form*) [1] [13] [8] [2] [11] [12] is a representation where indices are attached to variables so that every definition of each variable in a program becomes unique. At a joining point of the control flow graph (CFG) where two or more different definitions of a variable reach, a hypothetical function called a ϕ (phi)-function is inserted so that these multiple definitions are merged. Dataflow analysis and optimization for sequential execution can be compacted using the SSA form.

A. SSA Translation

Let us call the conventional representation form before translating into SSA form, *normal form*. In translating from normal form into SSA form (*SSA translation*) the algorithm proposed by Cytron et al. [8][2][11][12] and that by Sreedhar et al. [15] are well known. SSA translation generally consists of two phases, insertion of ϕ-functions and the renaming of variables. Three types of the translated SSA form are proposed, i.e., minimal SSA form [8][2][12], semi-pruned SSA form [3][12], and pruned SSA form [6][12]. The algorithms for translating into these forms are almost the same, only the live variable analysis being different. The difference between the three SSA forms is shown in Fig. 2.

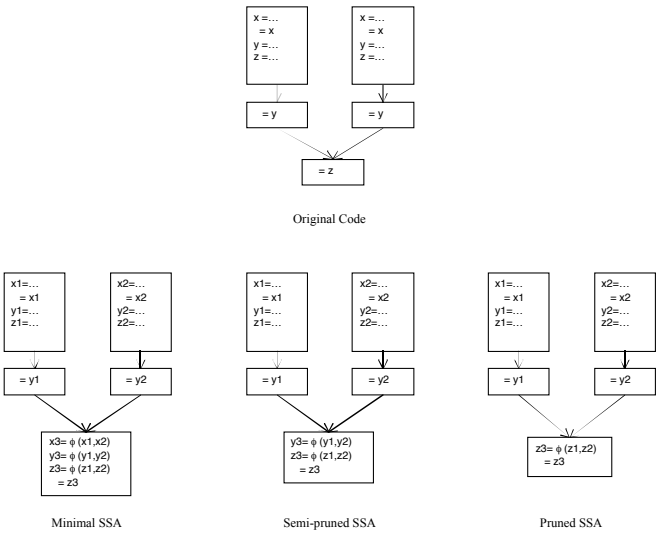
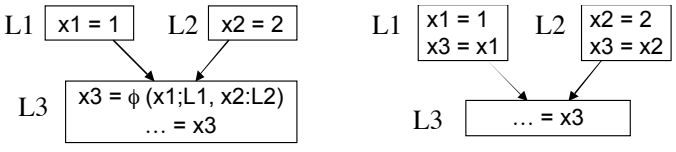


Fig. 2 Minimal, semi-pruned, and pruned SSA forms

B. SSA Back Translation

We call the translation from SSA form into normal form *SSA back translation*. For SSA back translation, the algorithms proposed by Briggs et al. [3], Morgan [11] and by Sreedhar et al. [16] are well known, although there are others.

In SSA back translation, the process formed by a ϕ-function is divided into the predecessor basic blocks. Therefore, in most cases, the back translation inserts copy statements for variables used in the ϕ-function into the predecessor block of the basic block where the function resides, and then deletes the ϕ-function. This gives the normal form. Fig. 3 shows an example of SSA back translation. In Fig. 3 (a), variables x1 and x2 used in the parameters of ϕ-function in block L3 are the use of definitions reached from block L1 and block L2, respectively. The SSA back translation puts the definitions of variable x3, originally defined in the ϕ-function of block L3, into L1 and L2, which are the predecessor blocks of L3, and then deletes the ϕ-function. This produces the result shown in Fig. 3 (b).



(a) SSA form (b) Normal form

Fig. 3 SSA back translation

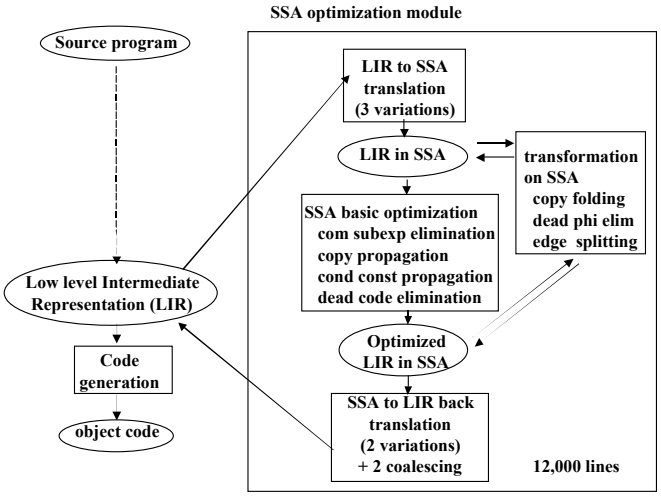


Fig. 4 Organization of the SSA form optimization module

IV. Organization of SSA form optimization module

The organization of the SSA form optimization module is shown in Fig. 4. This deals with the LIR. Because future use as an infrastructure was envisaged, the SSA module was designed so that the compiler writer can select from various representation forms and phases.

A. SSA Translation

We implemented SSA translation based on the algorithm by Cytron et al. [8]. In our system, the translation can produce three SSA forms, i.e., minimal, semi-pruned, and pruned, by performing live range analysis of varying precision at the ϕ-function insertion phase. Furthermore, copy folding and dead ϕ-function elimination can be performed simultaneously with the variable renaming phase of SSA translation [3][4].

The types of the translated SSA form can be selected, and whether or not simultaneous copy folding and dead ϕ-function elimination with variable renaming phase are required, can be specified by command options.

B. SSA Form Optimization

The following optimization in SSA form is currently implemented [2][11][12][18].

- dead code elimination
- conditional constant propagation
- common subexpression elimination
- copy propagation

As shown in Fig. 4, these optimizations are all performed in a submodule, and the compiler writer can specify the type and the order of optimization to be executed.

C. SSA Back Translation

We adopted the algorithm proposed by Sreedhar et al. [16] to implement the translation from SSA form into normal form. The following steps are executed in this algorithm.

Step 1: For each ϕ-function in the program, if there is interference between variables used in the ϕ-function, rename the variable and insert a copy statement.

Step 2: Eliminate dead copy.

Step 3: Delete ϕ-function and translate into normal form.

As regards Step 1, Sreedhar et al. proposed three algorithms, Method I, Method II, and Method III. Method I naively inserts copy statements for all variables used in the ϕ-functions. Method II is an extension of Method I, and inserts copy statements only for interfering variables. Compared to Method II, Method III further reduces the number of copy statements to be inserted. This is done by using not only interference information, but also live range information of the variables used in the ϕ-functions. We implemented Method I and Method III in our system. The Method can be selected by command option at compile time.

Step 2 eliminates unnecessary copy statements by executing SSA-based coalescing as proposed by Sreedhar et al. [16]. Our SSA module can also execute this step selectively.

In addition to Sreedhar et al.'s coalescing, we also implemented another coalescing algorithm based on the interference graph, which was originally proposed by Chaitin for register allocation [5]. This can be called selectively for the normal form after SSA back translation.

Kohama et al. implemented another SSA back translation algorithm on top of our system for comparison and further investigation. This is a good example of using our system as an infrastructure. We present this in Section VI.

V. Comparison of SSA translation methods

Several algorithms have been proposed for translation from normal form into SSA form. Two representative algorithms are that of Cytron et al. [8] and that of Sreedhar et al. [15]. Both algorithms use a data structure called the dominance frontier [2][11] and can efficiently translate even irreducible control flow graphs into SSA form.

Before deciding which algorithms should be included in the COINS infrastructure, we first made a prototype implementation of both these SSA translation algorithms and compared the two.

Generally, an SSA translation is divided into two phases, (i) insertion of ϕ-functions, and (ii) renaming of variables. The difference between the algorithms of Cytron et al. and Sreedhar et al. is in the way ϕ-functions are inserted. Cytron et al.'s method first computes the dominance frontier and then inserts ϕ-functions using that dominance frontier.

In contrast to this, Sreedhar's method can insert ϕ-functions at the same time as it computes the dominance frontier. This is done by using a special data structure called the DJ-graph. Sreedhar et al. claim that their method is faster than Cytron et al.'s method [15].

It is known that Cytron et al.'s method (i) can insert ϕ-functions in linear time with respect to the size of the control flow graph for most realistic programs, but that (ii) it requires quadratic time for programs with repeatedly nested do-while statements and for programs whose control flow graph becomes the so-called ladder graph (Fig. 5).

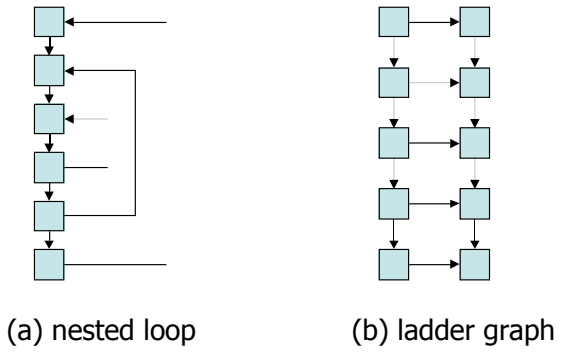


Fig. 5 Nested do-while and ladder graph

In contrast to Cytron et al.'s method, Sreedhar et al.'s method can insert ϕ-functions in linear time for any control flow graph.

We produced a prototype implementation of both algorithms and measured the time for SSA translation. The experiment was performed on CPU Athlon 650 MHz, memory 256 MB, OS Linux 2.4.0, and the algorithms were implemented in Java, Sun Java2 SDK 1.3.0.

The results are shown in Figs. 6, 7, and 8. The gap in the execution time is due to the garbage collection.

We can see from Fig. 6 that for normal programs Cytron et al.'s method is slightly faster than Sreedhar et al.'s method. This result is different from Sreedhar et al.'s claim.

We also found that in translating normal programs into minimal SSA form, the time for renaming the variables accounts for 60–70% of the total translation time. Therefore, the efficiencies of ϕ-function insertion methods are often not critical.

On the basis of this prototype implementation, we have decided to adopt Cytron et al.'s method in the SSA module of the COINS infrastructure.

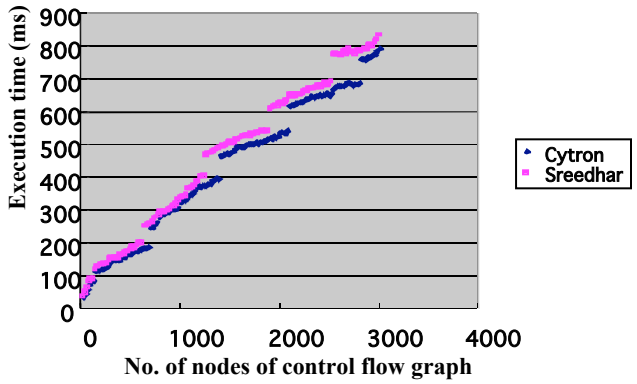


Fig. 6 Result for normal program

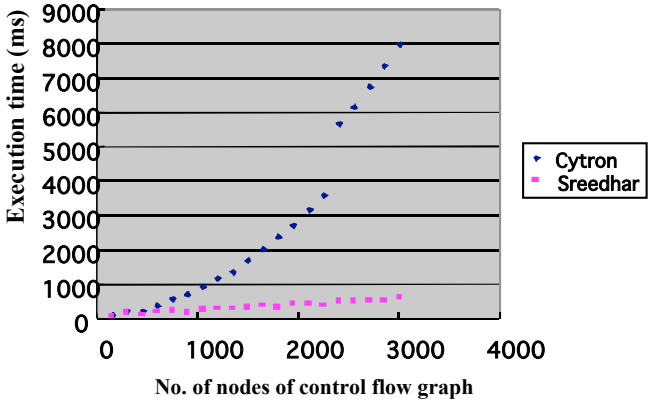


Fig. 7 Result for nested do-while statements

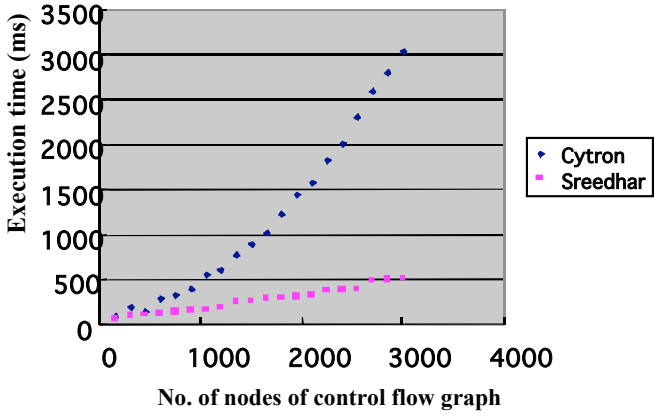


Fig. 8 Result for ladder graph

VI. Comparison of SSA Back Translation Methods

A. Problems of the Naive Algorithms

Several algorithms have been proposed for SSA back translation, namely the translation from SSA form to normal form.

One of the representative previous algorithms is that of Cytron et al. [8]. Their algorithm translates SSA form (Fig. 3 (a)) into normal form (Fig. 3 (b)) by inserting copy statements in the predecessor blocks of each ϕ-function and by deleting the ϕ-function. However, it has recently been pointed out that their algorithm does not work correctly in some cases [3][16][11]. One example of incorrect behavior is named the "lost copy problem" and is shown in Fig. 9.

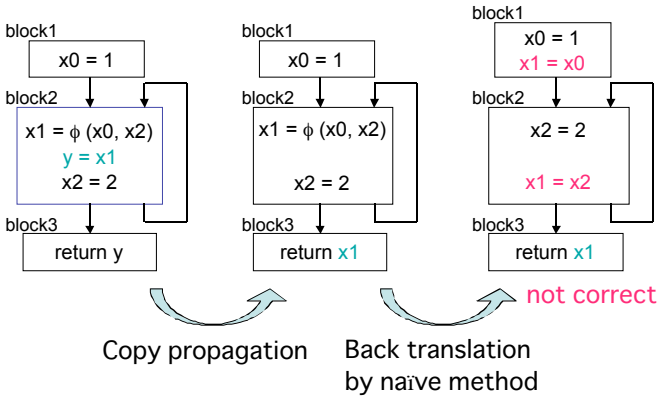


Fig. 9 The lost copy problem

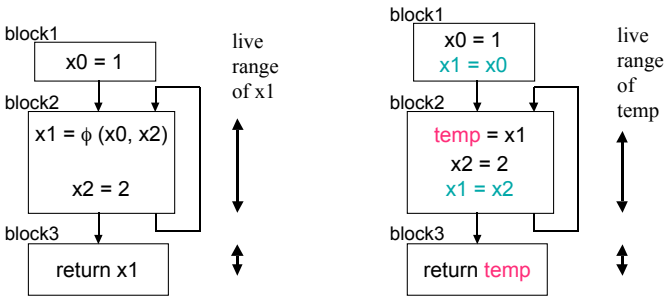
In Fig. 9, the figure on the left is a usual SSA form. After applying copy propagation optimization to the SSA form on the left, we obtain the SSA form in the middle. This SSA form is correct. Then, if we apply a naive algorithm such as Cytron et al.'s to this optimized SSA form, it inserts a copy statement "x1 = x2" at the end of block 2, i.e., the predecessor block of the block where the ϕ-function resides and we obtain the SSA form on the right hand side, which is incorrect. The value returned by "return x1" is now always 2, which is different from the original SSA form. The reason for this error is that the method inserts copy "x1 = x2" at the point where x1 is live.

There are also other problems in previous SSA back translation algorithms that give incorrect results, such as the so-called "simple ordering problem" and the "swap problem" [3][16].

To remedy these problems, two algorithms have been proposed. One is by Briggs et al. [3][4][11] and the other is by Sreedhar et al. [16]. Hereafter, we often call them simply Briggs' and Sreedhar's algorithm, respectively.

B. Solution by the Algorithm of Briggs et al.

The SSA back translation algorithm by Briggs et al. [3][4] extends Cytron et al.'s method to perform a safe translation (Fig. 10). As described above, the problem of the naive translation in Fig. 9 arose because the value of x1 in block 2 is destroyed by the insertion of "x1 = x2". In Briggs' method, it inserts an assignment to a temporary "temp = x1" at the entry of block 2 to save the value of x1 at this point to temp, and replaces the use of x1 in block 3 by temp, as in Fig. 10 (b).



(a) SSA form (b) after back translation

Fig. 10 SSA back translation by Briggs et al.

We see from this example that the SSA back translation algorithm by Briggs et al. inserts many copies, that is, copies inserted by naive methods and also copies inserted to avoid these critical problems. However, Briggs et al. claims that coalescing the live ranges afterwards can eliminate many of these copies.

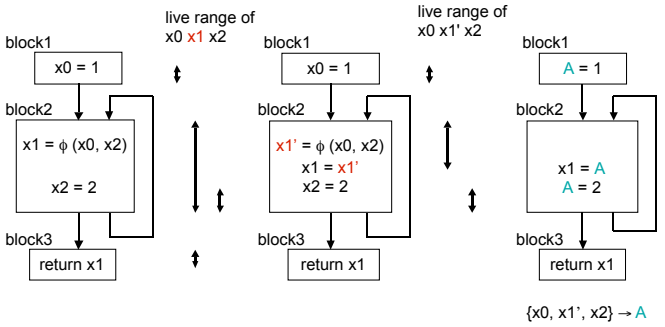
C. Solution by the Algorithm of Sreedhar et al.

The SSA back translation algorithm by Sreedhar et al. [16] uses a completely different approach from the naive methods or Briggs' method. Sreedhar's algorithm checks if there is interference between the live ranges of parameters of each ϕ-function. Here we assume that the parameters include the left hand side variable of the ϕ-function. If there is interference between the live ranges of parameters of a ϕ-function, the algorithm renames such a parameter and inserts a copy so that there is no more interference of live ranges between such parameters.

For example, Fig. 11 (a) is the same SSA form as in the center of Fig. 10. Consider the ϕ-function in block 2. It has three parameters x1, x0, and x2. Because there is interference of live ranges between x1 and x2, we replace x1 by x1' and insert a copy "x1 = x1' ", producing Fig. 11 (b). Now there is no interference between the live ranges of parameters x1', x0 and x2 of the ϕ-function. When there is no more interference between parameters of every ϕ-function, we replace all the parameters of a ϕ-function by a single variable and delete the

ϕ-function. In this example, we replace x0, x1' and x2 by A, and obtain the normal form shown in Fig. 11 (c).

Note that in general we do not need to insert copy statements for all parameters of ϕ-functions, in contrast with Briggs' algorithm.



(a) SSA form (b) eliminating interference (c) after back translation

Fig. 11 SSA back translation by Sreedhar et al.

D. *Preliminary Empirical Comparison of Briggs' and Sreedhar's Algorithms*

Selecting an appropriate SSA back translation algorithm is important, because it influences the run-time efficiency of the program after translation. To date there have been almost no studies comparing SSA back translation algorithms. As for theoretical optimality or minimality of the result of the translation, Sreedhar et al. presents some discussion, but he states his algorithm is not always optimal [16]. As far as we know, there is no discussion of optimality in the other papers.

Therefore, we made a preliminary empirical comparison of Briggs' and Sreedhar's algorithms on the same compiler, i.e., by utilizing the COINS compiler infrastructure. As stated in Section IV, the SSA back translation algorithm by Sreedhar et al. and Chaitin's coalescing algorithm [5] are already included in the SSA optimization module of COINS. Therefore, we implemented the SSA back translation by Briggs et al. using the same module in COINS, and made a preliminary empirical evaluation.

Part of the result is shown in Table 1. Columns in the table indicate, from left to right, the source program, the number of copies (copy statements) in SSA form before SSA back translation, the number of copies after translation by Briggs' algorithm, the number of copies when Chaitin's coalescing is applied after Briggs' algorithm, and the number of copies after translation by Sreedhar's algorithm. The number in parentheses is the number of copies within loops.

We see from Table 1 that in many source programs, the number of copies are the same in columns "Briggs + Coalescing" and "Sreedhar". The result is similar for the number of copies in loops. However, we found that in some programs, such as Swap, Swap-lost and "do", Sreedhar's translation algorithm gives better results, that is, the number of copies in Sreedhar's method is less than in the others. Furthermore, the difference in the number of copies occurs in loops. This means that in these programs Briggs' algorithm may introduce a run-time overhead due to the copy statements.

Conversely, in Hige Swap, which is a program that we have intentionally composed, "Briggs + Coalescing" gives a better result than Sreedhar's.

Overall, we found from this experiment that no single SSA back translation algorithm gives optimal results. We are planning to develop a new algorithm that is better than these two algorithms.

TABLE 1 Empirical comparison of SSA back translation

	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)

VII. Discussion

As far as we know there are few compiler infrastructures that include SSA form. SUIF [17] has very little support of the SSA form. Machine SUIF [10] has only dead code elimination as SSA form optimization. Scale [14] has several SSA form optimizations, but it generates only C programs and cannot generate machine code, as opposed to COINS. GCC [9] attempts SSA optimizations but they are written to be experimental.

VIII. Conclusion

In this paper, we presented the basic SSA form module of the COINS compiler infrastructure. We also showed an experimental comparison of SSA translation algorithms using our prototype implementation. We empirically compared SSA back translation algorithms and showed no single algorithm gives optimal translation.

Currently, the development stage of COINS is at the prototype phase. In the near future, we expect that further algorithms for the SSA form can be developed simply, using this compiler infrastructure.

REFERENCES

[1] B. Alpern, M. N. Wegman and F. K. Zadeck. Detecting Equality of Variables in Programs. In Proc. of the 15th ACM POPL, pp. 1-11, 1988.

[2] A. W. Appel. Modern Compiler Implementation in Java, Cambridge Univ. Press, 1998.

[3] P. Briggs, K. D. Cooper, T. J. Harvey and L. T. Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. Softw. Pract. Exper., Vol. 28, No. 8, pp. 859-881, 1998.

[4] P. Briggs, T. J. Harvey and L. T. Simpson. Static Single Assignment Construction, Version 1.0. January 1996. ftp://ftp.cs.rice.edu/public/compilers/ai/SSA.ps

[5] G. J. Chaitin. Register Allocation and Spilling via Graph Coloring. SIGPLAN Notices 17(6), Proc. of the ACM SIGPLAN '82 Symp. on Compiler Construction, pp. 98-105, 1982.

[6] J.-D. Choi, R. Cytron and J. Ferrante. Automatic Construction of Sparse Data Flow Evaluation Graphs. In Proc. of 18th ACM POPL, pp. 55-66, 1991.

[7] COINS. http://www.coins-project.org/.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM Trans. Prog. Lang. Syst., Vol. 13, No. 4, pp. 451-490,1991.

[9] GCC. http://www.gnu.org/software/gcc/.

[10] Machine SUIF. Harvard University. http://www.eecs.harvard.edu/machsuif/.

[11] R. Morgan. Building an Optimizing Compiler. Digital Press, 1998.

[12] Ikuo Nakata. Compiler organization and optimization. Asakura Shoten, 1999 (in Japanese).

[13] B. K. Rosen, M. N. Wegman and F. K. Zadeck. Global Value Numbers and Redundant Computations. Proc. of the 15th ACM POPL, pp. 12-27, 1988.

[14] Scale Compiler Group. University of Massachusetts. http://www-ali.cs.umass.edu/Scale/.

[15] V. C. Sreedhar and G. R. Gao. A Linear Time Algorithm for Placing ϕ-Nodes, Proc. of the 22nd ACM POPL, pp. 62-73, 1995.

[16] V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies and V. Santhanam. Translating Out of Static Single Assignment Form. SAS'99, Vol. 1694 of Lecture Notes in Computer Science, Springer-Verlag, pp. 194-210, 1999.

[17] SUIF. Stanford University. http://www-suif.stanford.edu/.

[18] M. N. Wegman and F. K. Zadeck. Constant Propagation with Conditional Branches. ACM Trans. Prog. Lang. Syst., Vol. 13, No. 4, pp. 181-210, 1991.