

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.

{sassa,ito1}@is.titech.ac.jp

Optimizations in compilers are becoming crucial for efficient execution of large software. The static single assignment form (SSA form) is becoming popular as the intermediate representation of compilers because of its simplicity in the dataflow analysis and optimization algorithms. However, the SSA form includes hypothetical functions, and translation into normal form is necessary before code generation. This paper compares and evaluates two major algorithms for back translating the SSA form into normal form. We made experiments by changing the number of allocatable registers. The 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 7%, which is equal to applying a middle-level global optimization.

1 Introduction

Optimizations in compilers are becoming crucial for efficient execution of large software. The static single assignment form (SSA form) [6] is becoming popular as the intermediate representation of compilers because of its simplicity in the dataflow analysis and optimization algorithms [7, 9, 14].

However, the SSA form includes hypothetical functions called ϕ -functions, and it cannot be directly translated into assembly code or machine code. Therefore, translation into normal form which deletes the ϕ -function is necessary before code generation. This translation is called *SSA back translation* or *normalization*.

The naïve SSA back translation algorithm by Cytron et al. [6] had several critical problems [3]. To remedy this, Briggs et al. proposed a new back translation algorithm [3]. After that Sreedhar et al. proposed another back translation algorithm based on a different approach [16]. The former is adopted in many compilers which use the SSA form, such as Marmot [7], Jikes [9], and Scale [14]. But the latter is not widely adopted except in [10], probably only because it is proposed later than the former.

The method by Briggs et al. replaces ϕ -functions by copy statements. This augments the number of copy statements, but they claim that those copy statements can be deleted by performing a coalescing pass. The method by Sreedhar et al. accomplishes a kind of coalescing to the destination and parameters of ϕ -

functions. Copy statements increase the number of executed instructions. On the other hand, coalescing may augment the length of live range, which affects the register pressure and then causes spilling of registers if the number of allocatable registers is small. This may harm execution efficiency.

In this way, the choice of SSA back translation algorithm is significant since it may affect the run-time efficiency of the program after register allocation and code generation. However, there have been no faithful comparison of these two algorithms. Sreedhar et al. make some theoretical discussion whether the result of their translation is minimum, but the minimality cannot be shown [16]. Also, there has been no empirical comparison, especially considering register allocation.

This paper clarifies advantages and disadvantages of the above two major algorithms for SSA back translation. We also propose an improvement of Briggs et al.'s algorithm. We implemented the above algorithms on the same compiler, and made experiments for several benchmarks by changing the number of allocatable registers and the set of optimizations.

The result shows that in the method of Briggs et al., many copy statements that can not be coalesced remain, which falls short of their expectations. The method of Sreedhar et al. was superior in this empirical study, and the efficiency of its object code is better than that of Briggs et al. by 1% to 7%. In summary, selecting a good back translation algorithm is quite important since it reduces the execution time of the

object code in no small way, which is equal to applying a middle-level global optimization.

2 Static Single Assignment Form

The static single assignment form (SSA form) [6, 1] is an internal representation of programs where indices are attached to variables so that the 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 into one definition point (Fig. 1).

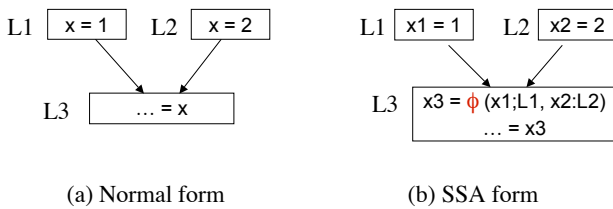


Fig. 1: Static single assignment form

Dataflow analysis and optimization for sequential execution can be compacted using the SSA form, since it makes the relation of definitions and uses of variables clear.

2.1 SSA Translation

Let us call the conventional representation form before translating into SSA form, *normal form*. *SSA translation* translates normal form into SSA form. The SSA translation algorithm proposed by Cytron et al. [6] and that by Sreedhar et al. [15] are well known.

3 SSA back translation

The SSA form includes hypothetical ϕ -functions and cannot be directly translated into assembly code or machine code. Therefore, translation into normal form which deletes the ϕ -function is necessary before code generation.

We call the translation from SSA form into normal form *SSA back translation* or *normalization*.

3.1 Naïve algorithm for SSA back translation

Cytron et al. [6] presented a basic algorithm for SSA back translation. In SSA back translation, the process formed by a ϕ -function is divided into the predecessor basic blocks. Therefore, the back translation inserts

copy statements for variables used in the ϕ -function into the predecessor blocks of the basic block where the ϕ -function resides, and then deletes the ϕ -function. This gives the normal form. Fig. 2 shows an example of SSA back translation.

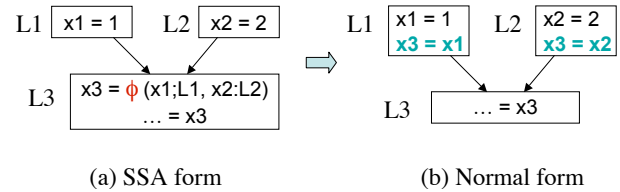


Fig. 2: SSA back translation

In Fig. 2(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. For example, if control goes from L1 to L3, the value of $x3$ becomes the value of $x1$. The SSA back translation puts the definitions of variable $x3$, originally defined in the ϕ -function of block L3, at the end of instructions of L1 and L2, which are the predecessor blocks of L3. For example, it puts the copy statement “ $x3 = x1$ ” at the end of block L1. Then it deletes the ϕ -function. This produces the result shown in Fig. 2(b). This algorithm is simple, but as we show shortly, there are problems in this algorithm. So we call it *naïve back translation algorithm*.

3.2 Problems of the naïve back translation algorithm

It has been pointed out that the naïve back translation algorithm does not work correctly in some cases [3, 16]. There are two main factors that makes the naïve back translation algorithm cause such problems.

One problem is related to the destruction of the live range when inserting copy statements. An example of such an incorrect behavior can be found in the so-called “lost copy problem” and is shown in Fig. 3.

In Fig. 3, 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 the naïve algorithm 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

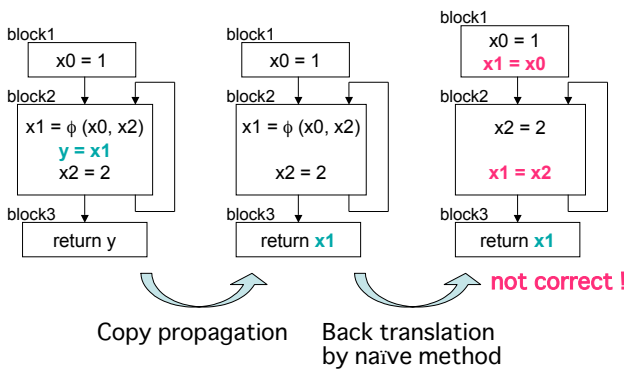


Fig. 3: The lost copy problem

the method inserts the copy statement “ $x1 = x2$ ” at the point where $x1$ is live, destroying the current value of $x1$.

The second problem of the naïve SSA back translation algorithm occurs when there are a plurality of ϕ -functions in the same block. Examples can be seen in the so-called “simple ordering problem” (Fig. 4) and the “swap problem” [3, 16].

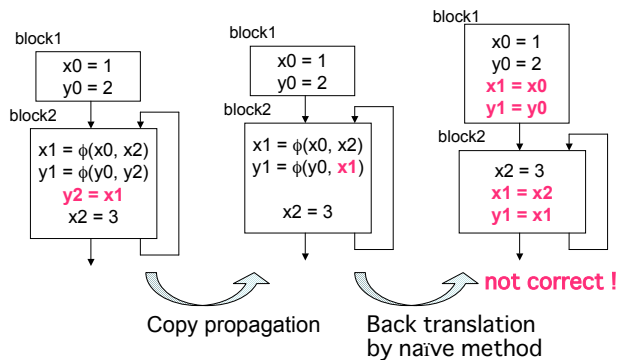


Fig. 4: The simple ordering problem

The simple ordering problem is shown in Fig. 4. In Fig. 4, 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 the naïve algorithm to this optimized SSA form, it inserts copy statements “ $x1 = x0$ ” and “ $y1 = y0$ ” at the end of block 1, and “ $x1 = x2$ ” and “ $y1 = x1$ ” at the end of block 2, and we obtain the SSA form on the right hand side. But this is incorrect. The value of “ $y1$ ” is now always 3, which is different from the original SSA form.

The semantics of SSA form requires that a plurality of ϕ -functions in the same block must be regarded as simultaneous assignments. For example, in the SSA

form in the middle of Fig. 4, assignments to $x1$ and $y1$ must be considered as occurring simultaneously. The reason of the error in the back translation here is that the naïve method inserts copy statements without considering the simultaneous assignment property of the ϕ -functions. As a result, dependency of $x1$ is introduced among the copy statements inserted in block 2. By sequentially executing these statements, it causes a behavior different from the original program.

4 Two major algorithms for SSA back translation

To remedy the problems presented in section 3.2, two major algorithms have been proposed. There are also other algorithms like Morgan’s [12], but they can be thought of as a subset of the following two algorithms.

One is by Briggs et al. [3, 2] and the other is by Sreedhar et al. [16]. Hereafter, we often call them simply Briggs’ and Sreedhar’s algorithm, respectively.

4.1 Algorithm of Briggs et al.

The SSA back translation algorithm by Briggs et al. [3, 2] extends the naïve back translation algorithm to perform a safe translation (Fig. 5).

As described in section 3.2, there were several problems in the naïve back translation algorithm. However, here we only take up the “lost copy problem”.

The problem of the naïve translation in Fig. 3 arose because the value of $x1$ in block 2 is destroyed by the insertion of “ $x1 = x2$ ”. Briggs’ method inserts an assignment to a temporary, “ $temp = x1$ ”, at the entry of block 2 to save the value of $x1$ at this point into $temp$, and replaces the use of $x1$ in later blocks by $temp$, as in Fig. 5(b). In this way, Briggs’ method realizes a correct SSA back translation by inserting copy statements such as “ $temp = x1$ ” to remedy the problems of the naïve method.

We see from this example that the SSA back translation algorithm by Briggs et al. inserts many copies, that is, copies inserted by the naïve method and also copies inserted to avoid these critical problems. However, Briggs et al. claim that *coalescing* the live ranges (coalescing [4] was originally proposed for register allocation) after the back translation can eliminate many of these copies. This is illustrated in Fig. 5(c) and (d). Since the live ranges of $x0$, $x1$ and $x2$ do not interfere in Fig. 5(c), they can be coalesced into a single variable x as in Fig. 5(d).

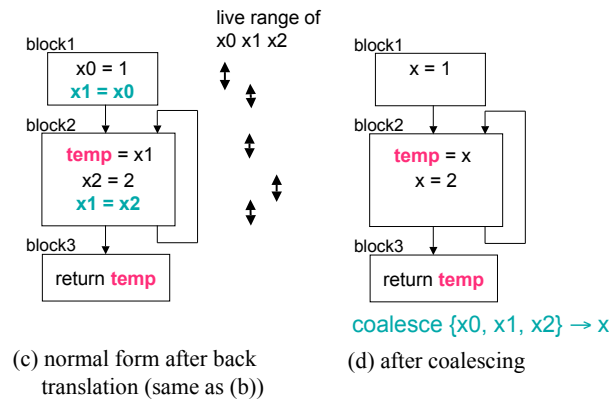
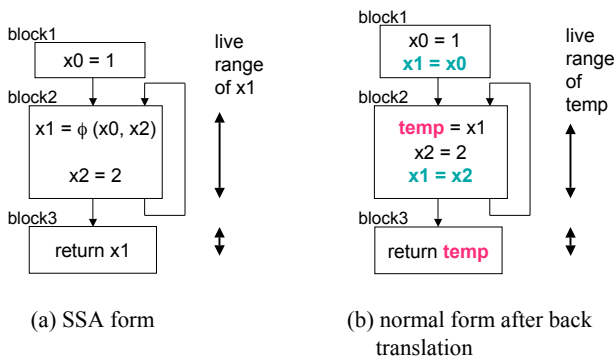


Fig. 5: SSA back translation by Briggs et al. (lost copy problem)

4.2 Algorithm of Sreedhar et al.

The SSA back translation algorithm by Sreedhar et al. [16] uses a completely different approach from the naïve algorithm or Briggs’ algorithm.

In contrast to the naïve or Briggs’ algorithm which insert copy statements to delete ϕ -functions, Sreedhar’s algorithm checks if there is interference between the live ranges of the parameters of each ϕ -function. Here we regard the left hand side variable of the ϕ -function as parameters. If there is interference between the live ranges of parameters of a ϕ -function, the algorithm renames such a parameter and inserts a copy statement so that finally there is no more interference of live ranges between parameters. Then, it replaces the parameters of the ϕ -function by the same variable and delete the ϕ -function.

A very simple example of Sreedhar’s algorithm can be found in Fig. 6. In Fig. 6(a) there is no interference between the parameters $x3, x1$ and $x2$ (including the left-hand side variable $x3$, called *target*) of the ϕ -function. So, all the parameters $x3, x1$ and $x2$ can be replaced by the same variable X and the ϕ -function can be deleted as in Fig. 6(b). (We can regard this replacement to the same variable as a kind of coalesc-

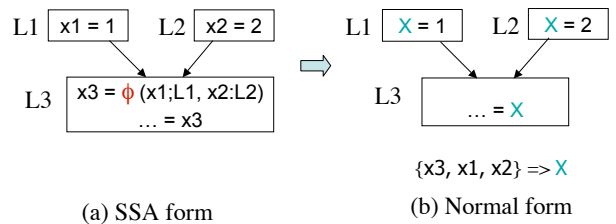


Fig. 6: SSA back translation by Sreedhar et al. - principle

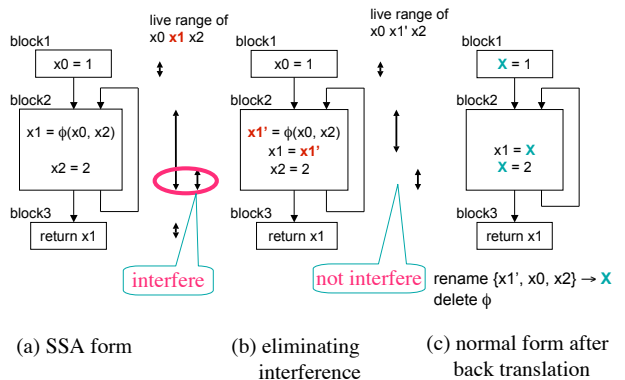


Fig. 7: SSA back translation by Sreedhar et al. (lost copy problem)

ing.) There is no need to insert copy statements in this case.

A more complex example can be found in Fig. 7. Fig. 7(a) is the same SSA form as in the center of Fig. 3. 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. 7(b). Now there is no interference between the live ranges of parameters $x1', x0$ and $x2$ of the ϕ -function. So, we replace all the parameters of the ϕ -function by a single variable and delete the ϕ -function. In this example, we replace $x1', x0$ and $x2$ by X , and obtain the normal form shown in Fig. 7(c).

The main part of Sreedhar’s algorithm resides in the treatment of the interference among live ranges of the parameters of a ϕ -function. In general, if there are such interference, coalescing these parameters is not possible. In that case, we have to remove the interference among these parameters. Sreedhar’s algorithm removes this interference by rewriting the ϕ -function (Fig. 8). This is done by combining the following processes. They have an effect of shortening the live ranges of parameters of the ϕ -function, and they finally

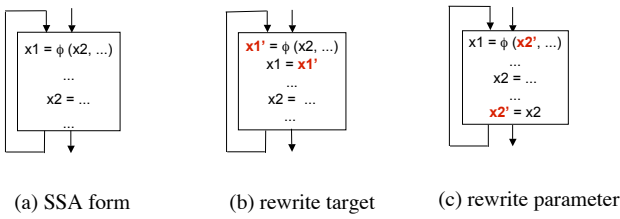


Fig. 8: Rewriting ϕ -function in Sreedhar et al.

remove the interference.

- The target of a ϕ -function is considered live at the entry of the block where the ϕ -function is placed. So, in order to make the live range of the target minimum, perform rewriting as in Fig. 8(b).
- The source (i.e. parameter in the right hand side) of a ϕ -function is considered live at the exit of the corresponding predecessor block of the block where the ϕ -function is placed. So, in order to make the live range of the source minimum, perform rewriting as in Fig. 8(c).

Note again that generally in Sreedhar’s algorithm, we do not need to insert copy statements for *all* parameters of ϕ -functions, in contrast to Briggs’ algorithm.

5 Problems and proposal for improvement

The two algorithms for SSA back translation presented so far have some drawback or weakness. We discuss them and propose an improvement in this section.

5.1 Drawback in Briggs’ algorithm and proposal for improvement

5.1.1 Unnecessarily long live range

In Briggs’ algorithm, copy statement like “*temp = target*” is sometimes inserted to save the target variable of a ϕ -function. This often causes related variables to have unnecessary long live ranges.

For example, consider Fig. 9. In Fig. 9(b), although the value of $x3$ is stored in *temp*, the live range of $x3$ continues up to “ $\dots = x3 + 1$ ” since $x3$ is used there. Due to this unnecessary live range, $x3$ interferes with $x2$.

In this case, if we make an improvement which first rewrite the target $x3$ of the ϕ -function as in Fig. 9(c), and then perform Briggs’ SSA back translation, we get Fig. 9(d). (Rewriting the target of a ϕ -function

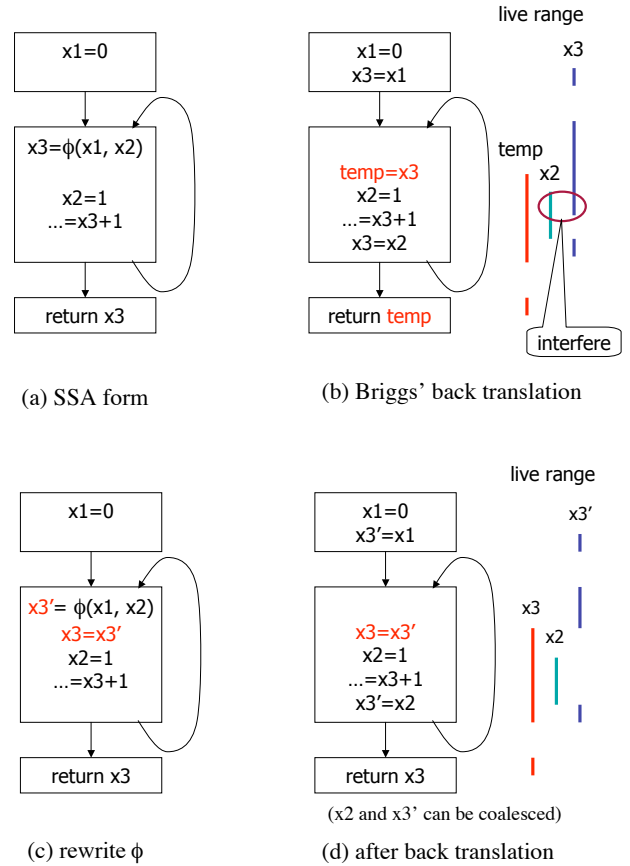


Fig. 9: Example with unnecessary live range

is similar to that of Sreedhar’s.) In Fig. 9(d), we can coalesce $x2$ and $x3'$ and delete “ $x3' = x2$ ”.

This improvement may be largely effective since saving the target always occurs in loops.

5.1.2 Conditional branch

The last statement of a block is often a conditional branch. In that case, copy statements to be inserted at the end of a block are put immediately before the conditional branch instruction which includes a conditional expression. In this case, if the *source* (the right-hand side) of the inserted copy statement is used in the conditional expression, the live range of the *target* and the *source* of the copy statement interfere.

For example, consider the example of Fig. 10. In Fig. 10(b), the live ranges of $x1$ and $x3$ interfere. This occurs because $x1$ is used in the conditional expression, although $x3$ keeps the same value as $x1$. Therefore, if we make an improvement that rewrites the conditional expression as in Fig. 10(c), $x3$ and $x1$ can be coalesced and “ $x3 = x1$ ” can be deleted.

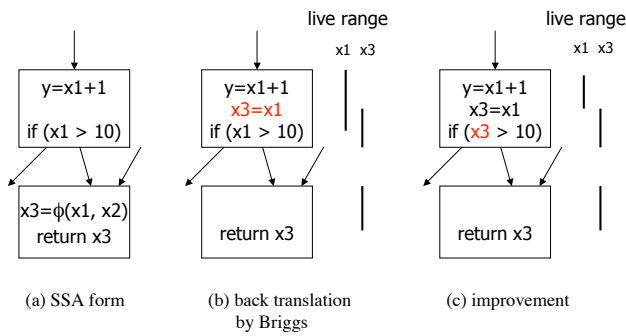


Fig. 10: Interference of live ranges due to conditional branch

5.2 Weakness in Sreedhar’s algorithm

Sreedhar’s algorithm deletes ϕ -functions by coalescing all parameters of a ϕ -function into the same variable. Therefore, by regarding ϕ -functions as copy statements, we can make a discussion similar to the register allocation that performs coalescing. That is, the live ranges of the coalesced variables become long, and it may be a disadvantage when the number of available registers is small.

6 Experiments

We made experiments using the C-to-SPARC compiler of the Coins compiler infrastructure [5]. The comparison is made on the same framework of the SSA module of Coins [13]. The experiments are made on UltraSPARC-III, 750MHz \times 2, 64kB (data) and 32kB (instruction) L1 cache, 1GB memory, SunOS 5.8.

The benchmarks are C programs from SPECint2000.

6.1 Flow of experiments

The source program is first converted into a pruned SSA form [3] intermediate representation, then several kinds of optimizations are applied. After that, the SSA form is back translated into normal form using one of Briggs’, Sreedhar’s or proposed algorithm, then register allocation with iterated register coalescing [8] is applied, and finally the object code is generated. The optimizations applied in this experiment are one of the following: opt1 (copy propagation), opt2 (copy propagation, dead code elimination, common subexpression elimination), or opt3 (copy propagation, loop invariant code motion).

We measured the number of copy statements that cannot be coalesced (static count), the number of variables that are spilled in register allocation phase (static count), the number of load and store instructions (dy-

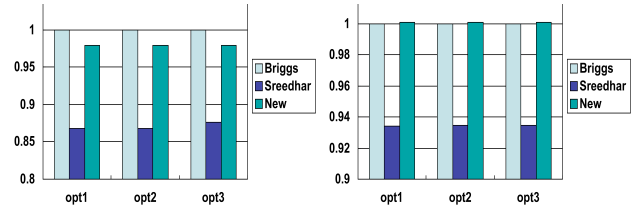


Fig. 11: Relative dynamic count of load and store (8 registers) (left: gzip, right: mcf)

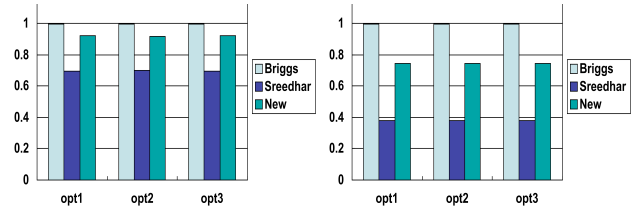


Fig. 12: Relative dynamic count of copies (8 registers) (left: gzip, right: mcf)

amic count), the number of copy statements (dynamic count), and execution time. We only include part of the results due to space limitation.

6.2 When the number of registers is 8

The relative ratio of the dynamic count of load and store instructions of the object code when the number of registers is 8 is shown in Fig. 11 (small is better). The reference values are those of Briggs’ algorithm, which are all normalized to 1. “New” means the algorithm proposed in this paper for improvement.

We can see from this figure that the dynamic count of load and store instructions in Sreedhar’s algorithm is about 87% to 93% of that of Briggs’. Our proposed algorithm is a little better than that of Briggs’ in gzip.

The relative ratio of the dynamic count of copy instructions of the object code is shown in Fig. 12 (small is better). We can see from this figure that the dynamic count of copy instructions in Sreedhar’s algorithm is about 70% to 38% of that of Briggs’. Our proposed algorithm is about 92% to 74% of that of Briggs’.

The relative ratio of the execution time of the object code is shown in Fig. 13 (small is better). As we can see from this figure, the object code made by Sreedhar’s algorithm is generally faster than that of Briggs’. That is, the former is faster by 3-4% in gzip except optimization 1, and faster by 4-5% in mcf except optimization 2.

The improvement proposed in this paper also raised the performance a little compared to Briggs’ algo-

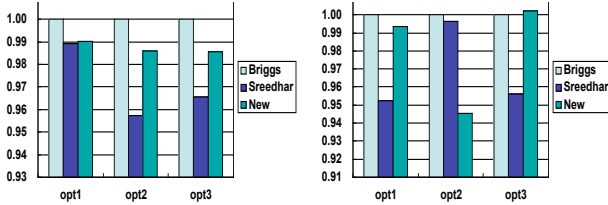


Fig. 13: Relative execution time (8 registers) (left: gzip, right: mcf)

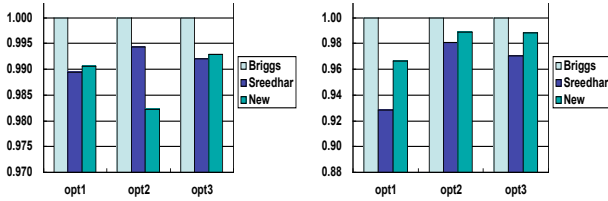


Fig. 14: Relative execution time (20 registers) (left: gzip, right: mcf)

rithm, i.e. faster by 1% in `gzip` and achieved moderate improvement in `mcf` except optimization 3.

6.3 When the number of registers is 20

When the number of registers is 20, the relative ratio of the dynamic count of load and store instructions and that of copy instructions had the same tendency as when the number of registers is 8, but they were sometimes less influential.

The relative ratio of the execution time when the number of registers is 20 is shown in Fig. 14. As we can see from this figure, Sreedhar’s algorithm generally produces faster code than Briggs’ algorithm. That is, the code by the former is faster by 2-7% in `gzip` and faster by 1% in `mcf`.

The improvement proposed in this paper also raised the performance compared to Briggs’ algorithm, i.e. faster by 1-3%.

There are also some fluctuation of results similarly to the case of 8 registers. We think that this fluctuation is mainly due to the characteristics of the target architecture, such as superscalar, pipelining, instruction level parallelism, and cache lines.

7 Concluding remarks

Two major algorithms are known for SSA back translation. However, there has been almost no previous research which compares them in depth. Therefore, many compilers using the SSA form simply adopted, without much consideration, the algorithm by Briggs et

al. which first solved the critical problems of the early naïve algorithm.

In this paper, we clarified the strength and weakness of different SSA back translation algorithms, proposed an improvement, and validated them through experiments. We have shown that the SSA back translation algorithm affects the efficiency of the object code in no small way, which has not drawn attention so far. A major contribution of this work was to give a criterion for choosing the SSA back translation algorithm in future compilers.

Our results to the point at issue are the following:

- In Briggs’ algorithm, a large amount of copy statements that cannot be coalesced are generated. It has more influence than the increase of register pressure that is worried in Sreedhar’s method.
- In the case when available registers are relatively few, Sreedhar’s algorithm, which make a kind of coalescing to the ϕ -functions, is superior from the viewpoint of the dynamic cost of spills and the number of executed copy statements
- In the case when available registers are relatively large, Sreedhar’s algorithm is favorable to some extent from the viewpoint of the number of executed copy statements
- We also gave a proposal for improving Briggs’ algorithm that is an approach based on replacing ϕ -functions by copy statements. This proposed method reduces the number of non-coalescible copy statements by about 40% and gives a few improvement on spill cost, but it could not surpass the superiority of Sreedhar’s algorithm.

Detailed discussion and a rich set of experimental results can be found in [11].

Further experiment on a variety of benchmarks and its analysis are left for future research.

Acknowledgments

This research has been partially supported 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), and the Kayamori Foundation for Informational Science Advancement.

References

- [1] Andrew W. Appel. *Modern Compiler Implementation in Java, 2nd ed.* Cambridge University Press, 2002.

- [2] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Value numbering. *Software – Practice and Experience*, Vol. 27, No. 6, pp. 701–724, June 1997.
- [3] 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*, Vol. 28, No. 8, pp. 859–881, July 1998.
- [4] G. J. Chaitin. Register allocation & spilling via graph coloring. *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pp. 98–105, 1982.
- [5] The Coins project. Research of a common infrastructure for parallelizing compilers. <http://www.coins-project.org/>.
- [6] 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*, Vol. 13, No. 4, pp. 451–490, October 1991.
- [7] Rober Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an optimizing compiler for Java. *Software – Practice and Experience*, Vol. 30, No. 3, pp. 199–232, March 2000.
- [8] Lal George and Andrew W. Appel. Iterated register coalescing. *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 208–218, January 1996.
- [9] IBM Jikes Research Virtual Machine. <http://oss.software.ibm.com/developerworks/oss/jikesrvm/>.
- [10] K. Ishizaki, M. Takeuchi et al. Effectiveness of cross-platform optimizations for a java just-in-time compiler. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003)*, pp. 187–204, 2003.
- [11] Masaki Kohama. Comparison and evaluation of SSA normalization algorithms. *Master's Thesis, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology*, (in Japanese), 2004.
- [12] Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.
- [13] Masataka Sassa, Toshiharu Nakaya, Masaki Kohama, Takeaki Fukuoka and Masahito Takahashi. Static Single Assignment Form in the COINS Compiler Infrastructure. *Proc. SSGRR 2003w*, 2003.
- [14] Scale Compiler Group. University of Massachusetts. <http://www-ali.cs.umass.edu/Scale/>.
- [15] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ -nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 62–73, January 1995.
- [16] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vasta Santhanam. Translating out of static single assignment form. In *Proceedings of the 6th International Symposium on Static Analysis*, Vol. 1694 of *Lecture Notes in Computer Science*, pp. 194–210. Springer-Verlag, 1999.