ISSN 1342-2812

Research Reports on Mathematical and Computing Sciences

Comparison and evaluation of back-translation algorithms for static single assignment forms

Masataka Sassa
Yo Ito
Masaki Kohama

October 2005, C–214

Department of Mathematical and Computing Sciences
Tokyo Institute of Technology

SERIES C: Computer Science

Comparison and Evaluation of Back-Translation Algorithms for Static Single Assignment Forms

Masataka Sassa†, Yo Ito† and Masaki Kohama‡

†Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2-12-1, O-okayama, Meguro-ku, Tokyo 152-8552, Japan
‡Fuji Photo Film Co., Ltd.

Correspondence:
Masataka Sassa
Dept. of Mathematical and Computing Sciences,
Tokyo Institute of Technology
W8-27, 2-12-1, O-okayama, Meguro-ku, Tokyo 152-8552, Japan
E-mail: sassa@is.titech.ac.jp
Tel: +81-3-5734-3228 (direct), -3210
FAX: +81-3-5734-3210

Earlier versions of parts of this article appeared at IPSI-2004 Prague meeting and at Transactions of Information Processing Society of Japan - Programming in Japanese.

SUMMARY

The static single assignment form (SSA form) is becoming popular as an intermediate representation in compilers. In the SSA form, the definition of each variable textually appears only once in the program by using a hypothetical function called a ϕ-function. Because these functions are nonexecutable, it is necessary to delete the ϕ-function and return the SSA form to the normal form before code generation. This conversion is called *SSA back-translation*. Two major algorithms exist for SSA back-translation. One is the method by Briggs et al., the other is the method by Sreedhar et al. To date, there has been almost no research that compares these SSA back-translation algorithms.

In this paper, we clarify the merits and demerits of these algorithms. We also propose an improvement of Briggs' algorithm. We then compare the three methods through experiments using the SPEC benchmarks.

Our experiments have shown that Sreedhar's method is the most favorable. The efficiency of its object code is better than that from Briggs's method, by a few percent in general, up to a maximum of 28%. The experiments have also clarified several characteristic features of these methods.

KEY WORDS: Static single assignment form; back-translation; machine-independent program transformation; compiler;

1 Introduction

Optimizations in compilers are becoming crucial for efficient execution of large software. The static single assignment form (SSA form) [7] is becoming popular as an intermediate representation of code in compilers because of its simplicity for dataflow analysis and the efficiency of optimization algorithms [8, 9, 11, 15, 18].

In the SSA form, the definition of each variable textually appears only once in the program by using a hypothetical function called a ϕ (phi)-function. Because the ϕ-functions appearing in the SSA form are nonexecutable, it is necessary to delete the ϕ-function and return the SSA form to the normal form before code generation. This translation is called *SSA back-translation*.

The naive SSA back-translation algorithm by Cytron et al. [7] has several critical problems, and it behaves incorrectly when it is applied to SSA forms that have been transformed by some optimizations [3]. To remedy this, Briggs et al. proposed a new back-translation algorithm [2, 3]. Later, Sreedhar et al. proposed another back-translation algorithm based on a completely different approach [20]. (Hereafter those methods are simply called Briggs' method and Sreedhar's method.) The former is adopted in many compilers that use the SSA form, such as Marmot [8], gcc [9], Machine SUIF [15] and Scale [18]. However, the latter is not widely adopted, except in [13], probably because it was proposed a few years after Briggs' method.

Briggs's method 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. Sreedhar's method accomplishes a kind of coalescing, which we call "uniting", to the target (left-hand side) and parameters of ϕ-functions.

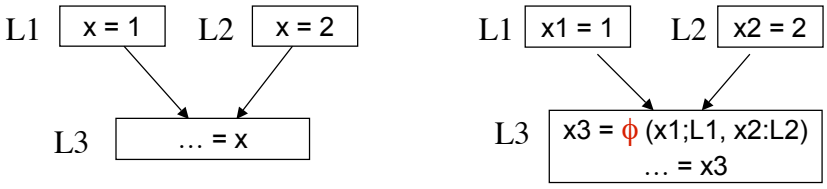
In Sreedhar's method, uniting may augment the length of live ranges of variables associated with ϕ-functions. This may increase the register pressure and thus may cause spilling of registers if the number of allocatable registers is relatively small. This may harm execution efficiency.

On the other hand, in Briggs' method, many copy statements are inserted, which may increase the number of executed instructions. However, it was thought that an undesirable situation as in Sreedhar's method would not occur if coalescing was properly performed in a later phase.

Thus, the two SSA back-translation algorithms have different characteristics, and the choice of algorithm may affect the runtime efficiency of the program after register allocation and code generation. However, there have been no careful comparisons of these algorithms to date. Sreedhar et al. gave some theoretical discussion of whether the result of their translation has minimum code length, but the minimality cannot be shown, and they offered no comparisons with other SSA back-translation algorithms [20]. In addition, there has been no empirical comparison of back-translation algorithms, especially considering register allocation.

In this paper, we clarify the merits and demerits of the above two major algorithms for SSA back-translation. We also propose an improvement of Briggs' algorithm. We implemented Briggs' algorithm, its improvement, and Sreedhar's algorithm on the same compiler, and experimented using the SPEC benchmarks by changing the number of allocatable registers. We evaluated the influence of coalescing, the relationship with the number of allocatable registers and the relationship with the set of optimizations, which gave insights into the results.

The results show that in Briggs' method, many copy statements that cannot be coalesced remain, which falls short of the original expectations.



(a) Normal form (b) SSA form

Figure 1: The static single assignment form

These copy statements not only cause extra execution time but may also consume registers wastefully. The combination of optimizations does not much affect the differences between the back-translation algorithms. The main result of the experiment is that in most cases Sreedhar's method is the most favorable. The efficiency of its object code is better than that of Briggs' method by a few percent in general and by up to 28% at maximum.

2 Static single assignment form

The static single assignment form (SSA form) [7, 1] is an internal representation of programs where indices are attached to variables so that the definition of each variable in a program becomes textually unique. At a joining point of the control flow graph (CFG) reached by two or more different definitions of a variable, a hypothetical function called a *ϕ-function* is inserted so that these multiple definitions are merged into one (Fig. 1).

The statement "*x3 = ϕ(x1:L1, x2:L2);*" in Fig. 1(b) means that *x3* is assigned the value of *x1* when the control comes from *L1* and the value of *x2* when the control comes from *L2*.

Dataflow analysis and optimization for sequential execution can be compacted using the SSA form, because it clarifies the relations between defini-

tions and uses of variables.

2.1 SSA translation

Let us call the conventional representation form before translating into SSA form the *normal form*. *SSA translation* translates the normal form into the SSA form. The SSA translation algorithm proposed by Cytron et al. [7] and that by Sreedhar et al. [19] are well known.

3 SSA back-translation

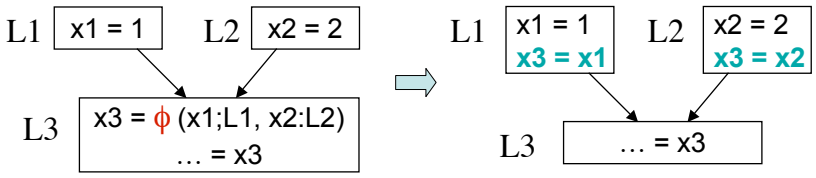
The SSA form includes the hypothetical ϕ-functions and cannot be directly translated into assembly code or machine code. Therefore, translation back to normal form, which deletes the ϕ-functions, is necessary before code generation.

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

3.1 Naive algorithm for SSA back-translation

Cytron et al. [7] presented a naive algorithm for SSA back-translation. The process formed by a ϕ-function is moved 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. Figure 2 shows an example of the naive SSA back-translation.

In Fig. 2(a), variables *x1* and *x2* used in the parameters of the ϕ-function in block L3 are defined in blocks L1 and L2, respectively. For example, if control goes from L1 to L3, the value of *x3* becomes the value of *x1*. The naive method for SSA back-translation puts the definitions of variable *x3*,



(a) SSA form (b) Normal form

Figure 2: Naive SSA back-translation

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 will show, it introduces problems. We therefore call it the *naive back-translation algorithm*.

3.2 Problems of the naive back-translation algorithm

It has been pointed out that the naive back-translation algorithm does not work correctly in some cases [3, 20]. There are two main factors that make the naive back-translation algorithm cause such problems.

The first problem is related to the destruction of the live range when inserting copy statements. An example of such an incorrect behavior is the so-called *lost copy problem* and is shown in Fig. 3.

Figure 3(a) shows the usual SSA form. After applying copy propagation optimization to this SSA form, "$y = x1$" in block 2 is deleted, and y in block 3 is replaced by *x1*, as in Fig. 3(b). This SSA form is correct. Then, if we apply the naive algorithm to this optimized SSA form, it inserts the copy statements at the end of each predecessor block. That is, it inserts a copy

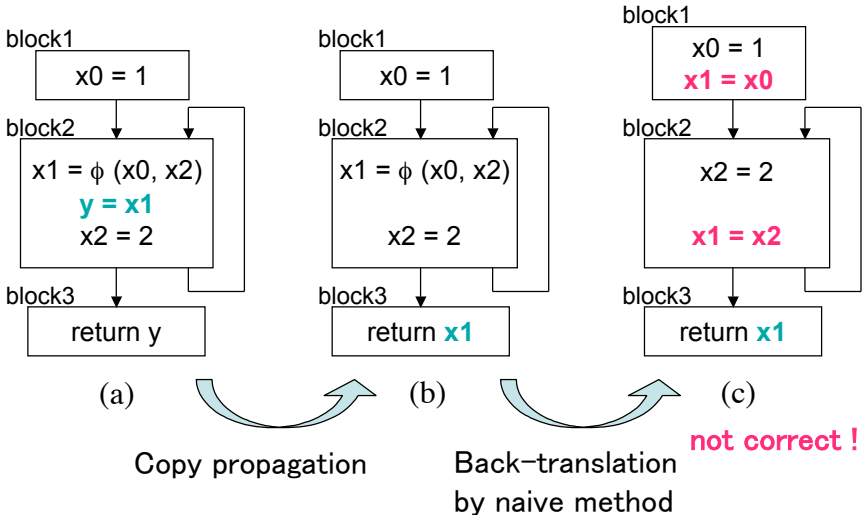


Figure 3: The lost copy problem

statement "*x1 = x0*" at the end of block 1, and "*x1 = x2*" at the end of block 2, and deletes the ϕ-function (Fig. 3(c)). However, this is incorrect. The value returned by "*return x1*" is now always 2, which is different from the original SSA form program. The reason for this error is that the method inserts the copy statement "*x1 = x2*" at a point where *x1* is live, destroying the current value of *x1*.

A second problem with the naive SSA back-translation algorithm occurs when there are multiple ϕ-functions in the same block. Examples can be seen in the so-called *simple ordering problem* and the *swap problem* [3, 20].

The simple ordering problem is shown in Fig. 4. Figure 4(a) is the usual SSA form. After applying copy propagation optimization to this SSA form, "*y2 = x1*" of block 2 is deleted, and "*y1 = ϕ(y0, y2)*" is replaced by "*y1 = ϕ(y0, x1)*" (Fig. 4(b)). This SSA form is correct. Then, if we apply the naive algorithm to this optimized SSA form, it inserts the copy statements "*x1 = x0*" and "*y1 = y0*" at the end of block 1, and "*x1 = x2*" and "*y1 = x1*" at

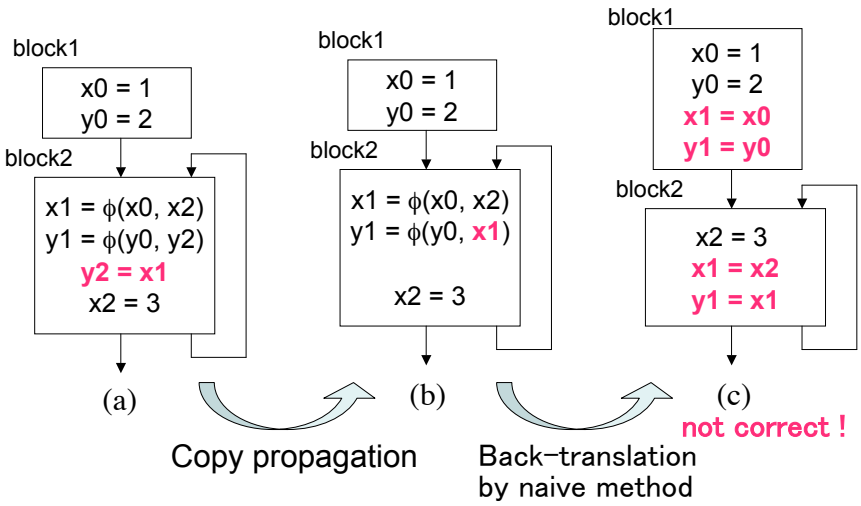


Figure 4: The simple ordering problem

the end of block 2, and we obtain the SSA form in Fig. 4(c). However, this is incorrect. The value of "*y1*" at the exit of block 2 is now always 3, which is different from the original SSA form program.

The semantics of the 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 Fig. 4(b), assignments to *x1* and *y1* must be considered as occurring simultaneously. The reason for the error in the back-translation here is that the naive method inserts copy statements without considering the simultaneous assignment property of the ϕ-function. As a result, a dependency for *x1* is introduced among the copy statements inserted in block 2. By executing these statements sequentially, program behavior differs from the original.

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 [16], but they can be thought of as a subset of the major two algorithms (cf. Section 9.1).

One is by Briggs et al. [3, 2] and the other is by Sreedhar et al. [20]. Hereafter, we often call them simply Briggs' and Sreedhar's algorithms.

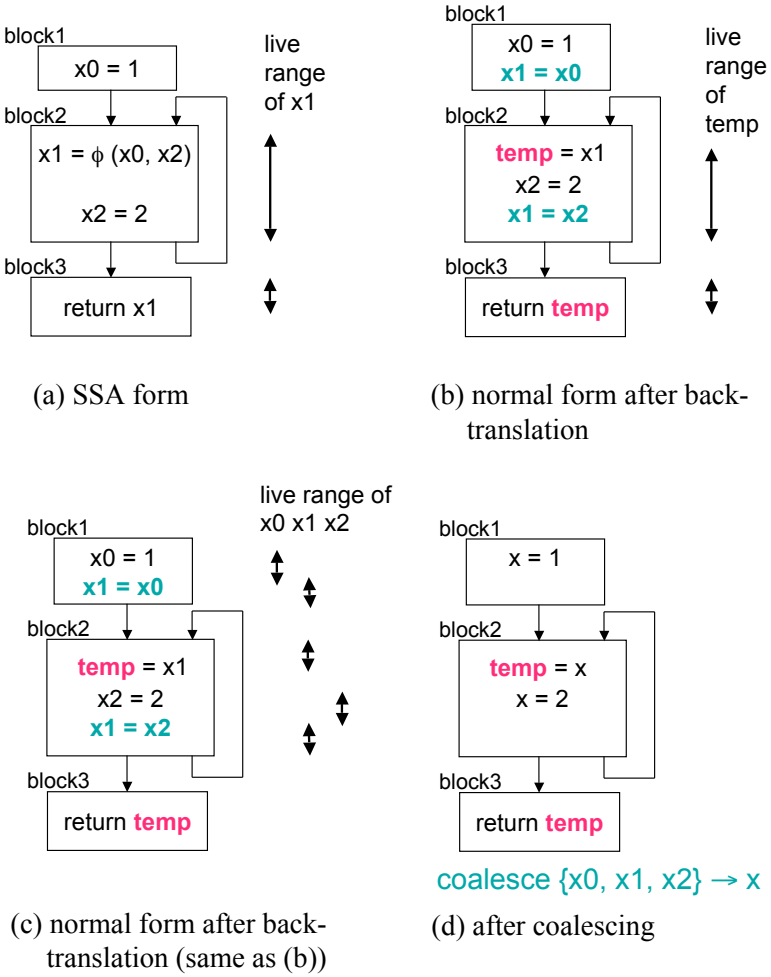
4.1 The algorithm of Briggs et al.

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

Among the problems with the naive back-translation algorithm, here we concentrate on the "lost copy problem" described in Section 3.2.

The problem with the naive 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 naive method.

We see from this example that the SSA back-translation algorithm by Briggs et al. may insert many copies, that is, both the copies inserted by the naive method and copies inserted to avoid these critical problems. However, Briggs et al. claimed that *coalescing* the live ranges (coalescing [5] is usually performed in the register allocation phase) after the back-translation can eliminate most of these copies. This is illustrated in Fig. 5(c) and (d). Because the live ranges of *x0*, *x1* and *x2* that are connected by copy statements do not interfere in Fig. 5(c), they can be coalesced into a single variable x as



(a) SSA form

(b) normal form after back-translation

(c) normal form after back-translation (same as (b))

(d) after coalescing

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

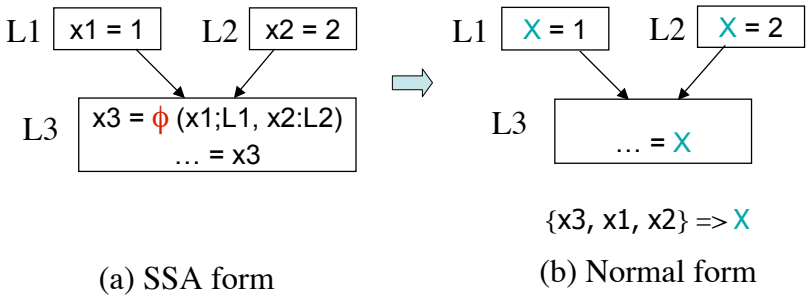


Figure 6: SSA back-translation by Sreedhar et al. – principle

shown in Fig. 5(d).

We will also perform coalescing in the experiments in Sections 6 and 7.

4.2 Algorithm of Sreedhar et al.

The SSA back-translation algorithm by Sreedhar et al. [20] uses a completely different approach from the naive algorithm or Briggs' algorithm.

Rather than insert copy statements to delete ϕ-functions, Sreedhar's algorithm checks whether 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 another parameter. If there is such interference, the algorithm renames some of the parameters by inserting copy statement(s) so that finally there is no more interference of live ranges between parameters. Then, it replaces all the parameters of the ϕ-function by a single variable and deletes the ϕ-function.

A very simple example of Sreedhar's algorithm is shown 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*) 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). In this example, no copy statements are inserted.

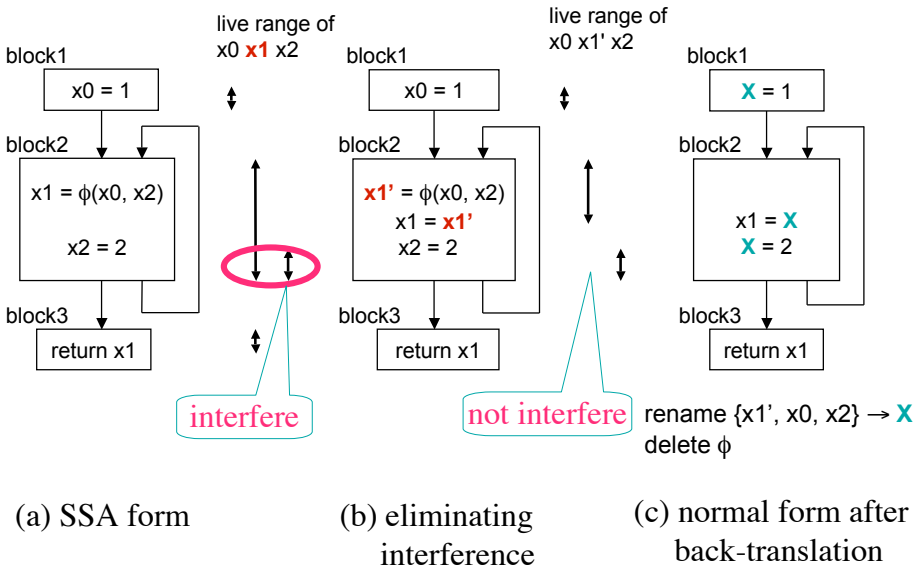


Figure 7: SSA back-translation by Sreedhar et al. (lost copy problem)

(We can regard this replacement to the same variable as a kind of coalescing. We call it *uniting*). There is no need to insert copy statements in this case as opposed to the naive or Briggs' algorithm.

A more complex example can be found in the treatment of the lost copy problem. Figure 7(a) is the same SSA form as in Fig. 3(b). 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. We therefore 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

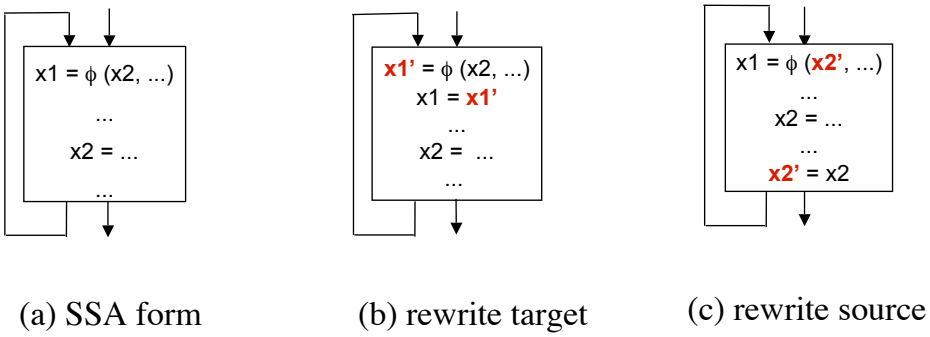


Figure 8: Sreedhar's rewriting of the ϕ-function

interference between the live ranges of the parameters of a ϕ-function. In general, if there is such interference, uniting these parameters is not possible. In that case, we must remove the interference caused by 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 the effect of shortening the live ranges of the parameters of the ϕ-function, and they finally remove the interference.

- The left-hand side variable of a ϕ-function is called a *target*. The target of a ϕ-function is considered live at the entry of the block where the ϕ-function is placed. Therefore, to minimize the live range of the target, perform rewriting as in Fig. 8(b).

- A parameter in the right-hand side of a ϕ-function is called a *source*. The source of a ϕ-function is considered live at the exit of the corresponding block that is predecessor to the block where the ϕ-function is placed. (It is not live at the entry of the block where the ϕ-function is placed.) Therefore, to minimize the live range of the source, perform rewriting as in Fig. 8(c).

Note again that in Sreedhar's algorithm, no insertion of copy statements for *all* parameters of the ϕ-functions is generally required, in contrast to Briggs' algorithm. Therefore, there are generally less inserted copy statements.

Note also that Sreedhar's algorithm can perform coalescing based on the SSA form at back-translation time [20].

5 Problems and proposal for improvement

The two algorithms for SSA back-translation presented so far have some weaknesses. 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 ranges

In Briggs' algorithm, copy statements like "*temp* = *target*" are sometimes inserted to save the value of the target of a ϕ-function. This often causes related variables to have unnecessarily long live ranges.

For example, in Fig. 9(b), although the value of *x3* is stored in *temp*, the live range of *x3* continues up to "...=*x3*+1", because *x3* is used there. This unnecessarily long live range causes *x3* to interfere with *x2*.

In this case, if we make an improvement that first rewrites the target *x3* of the ϕ-function of Fig. 9(a) as in Fig. 9(c), and then perform Briggs' SSA back-translation, we obtain Fig. 9(d). (Rewriting the target of a ϕ-function is made similarly to that of Sreedhar's.) In Fig. 9(d), we can coalesce *x2* and *x3'* and delete "*x3'* = *x2*".

This improvement may be very effective because such a situation of saving the target always occurs in loops.

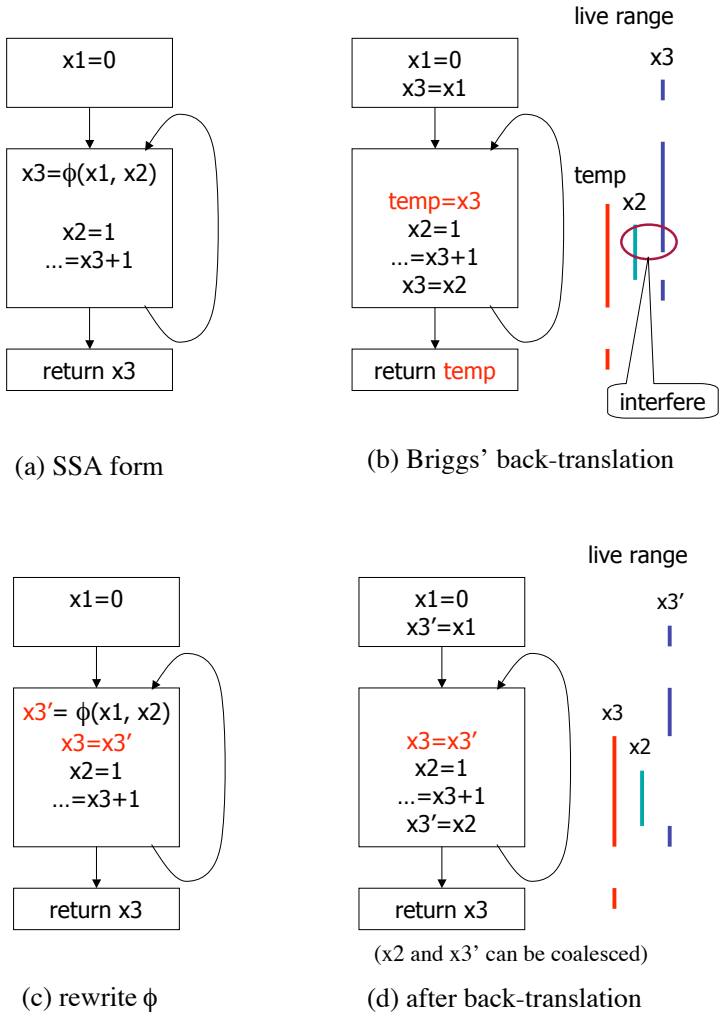
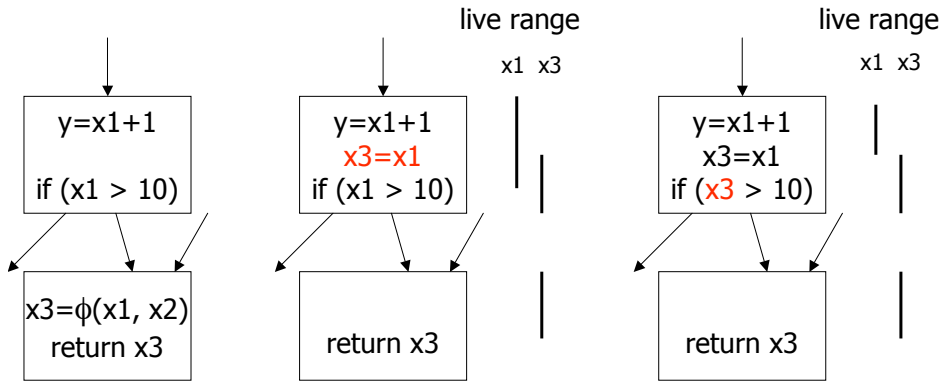


Figure 9: Example with unnecessarily long live range



(a) SSA form (b) back-translation by Briggs (c) improvement

Figure 10: Interference of live ranges because of conditional branch

5.1.2 Conditional branch

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

For example, 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.

5.2 Weakness in Sreedhar's algorithm

Sreedhar's algorithm deletes ϕ-functions by uniting all parameters of a ϕ-function into the same variable. Therefore, by regarding the ϕ-function as a group of copy statements, we can analyze it in the same way as the coalescing in the register allocation phase. That is, the live ranges of the united variables become long, which may be a disadvantage when the number of allocatable registers is small.

We will investigate this problem in Sections 6 and 7 through experiments.

6 Experiments 1

6.1 Purpose and criteria of evaluation

The purpose of this set of experiments was to investigate the general tendencies of the three back-translation algorithms, Briggs', Sreedhar's and our proposed improvement. To achieve a generally applicable result, we experimented by changing the number of registers and the combination of optimizations.

We used the C-to-SPARC compiler of the COINS compiler infrastructure [6]. The comparisons were made on the same framework of the SSA module of COINS [17].

The benchmarks were C programs from SPECint2000.

The processing by this compiler is as follows:

1. The source program is first converted into a pruned SSA form [3] intermediate representation.

2. Several kinds of optimizations are applied.

Table 1: Main specifications of Sun-Blade-1000

architecture	Superscalar SPARC V9
processor	UltraSPARC-III 750 MHz × 2
L1 cache	64 KB (data) 32 KB (instruction)
L2 cache	8 MB external cache
memory	1 GB
OS	SunOS 5.8

3. The SSA form is back-translated into normal form using one of Briggs', Sreedhar's or our proposed algorithm.

4. Register allocation with iterated register coalescing [10] is applied.

5. The object code is generated.

Instruction scheduling in COINS is at the test stage and does not run for some benchmarks used in this experiment, so we do not use instruction scheduling.

The experiments were run on a Sun-Blade-1000. The main specifications are shown in Table 1.

Our experiments used 20 and eight registers. The case of 20 registers is the model for architectures such as SPARC and MIPS, while the use of eight registers investigates the tendency for CPUs in embedded systems to have few registers.

The combinations of optimizations used are shown in Table 2. "-opt1" in the graphs of this section corresponds to "opt1" in Table 2, and so on.

For benchmarks, we used `gzip` and `mcf` from SPECint2000.

Table 2: Combination of optimizations

opt1	copy propagation
opt2	copy propagation, dead code elimination, common subexpression elimination
opt3	copy propagation, loop invariant code motion

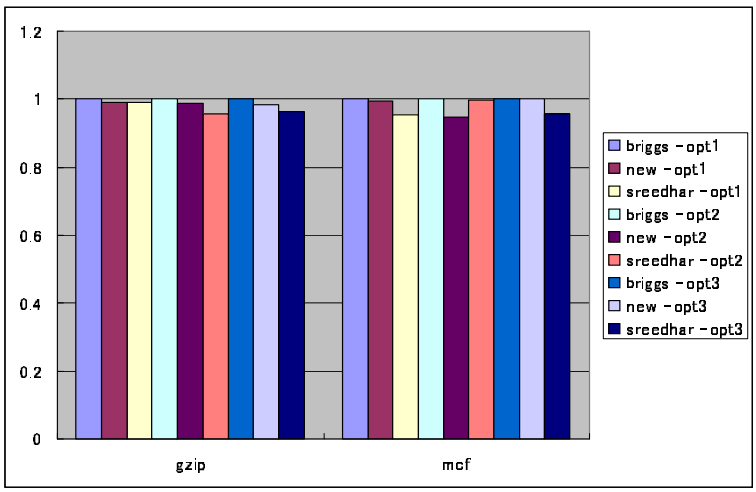


Figure 11: Ratios of execution times in eight registers

6.2 Comparison of execution times

We measured several aspects, but because this is a preliminary experiment, we show only the comparison of execution times here.

6.2.1 With eight registers

Figure 11 shows the relative ratios of the execution times of the object code when the number of registers is limited to eight (small values are better, the reference value is the time with Briggs's method, which is normalized to one.) 'new' means our proposed algorithm for improvement of Briggs'

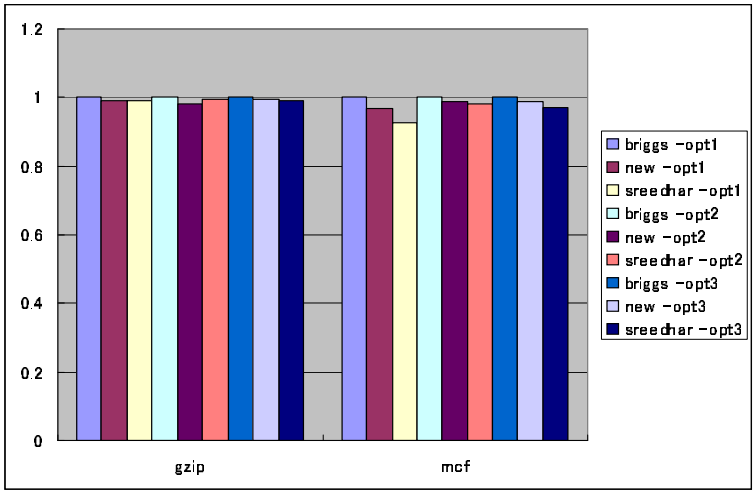


Figure 12: Ratios of execution times with 20 registers

method (Section 5.1).

We can see that the execution time with Sreedhar's method is generally shorter than that with Briggs' method, by 3˜4% in `gzip` except with optimization 1, and also faster by 4˜5% in `mcf` except with optimization 2.

Our proposed improvement also raised the performance a little compared to Briggs' method, by 1% in `gzip` and is also in `mcf` except with optimization 3.

6.2.2 With 20 registers

Next, Fig. 12 shows the relative ratios of execution times when the number of registers is 20. We can see that Sreedhar's method generally produces faster code than Briggs' method, by 1% in `gzip`, and by 2˜7% in `mcf`.

Our proposed improvement also raised the performance compared to Briggs' method, by 1% in `gzip` and by 1˜3% in `mcf`.

From these experiments we realize that Sreedhar's method is superior to Briggs' method in terms of the execution time of the object code. Detailed analysis can be found in [14]. Our proposal for improvement of Briggs' method was effective to a certain degree, but it could not overcome the predominance of Sreedhar's method. Therefore, in the following section we focus on Briggs' method and Sreedhar's method and make more detailed evaluations using further benchmarks.

7 Experiments 2

7.1 Purpose and criteria of evaluation

The purpose of this set of experiments was to focus on Briggs' method and Sreedhar's method, and make more detailed evaluation than that of the previous section, using more benchmarks

We used eight and 20 registers, as in the previous section. The combinations of optimizations are shown in Table 3. In the graphs of figures in this section, we denote "no optimization" as "-opt0", "optimization 1" as "-opt1", and "optimization 2" as "-opt2". (This is different from the previous section.)

From our previous discussion, we can anticipate that in Briggs' method, many copy statements that cannot be coalesced (in register allocation) are inserted (Section 5.1). Copy statements that cannot be coalesced may possibly waste registers unnecessarily.

On the other hand, in Sreedhar's method, we can anticipate problems similar to the aggressive coalescing in register allocation (Section 5.2).

Therefore, we first made a *static* evaluation by measuring the number of move instructions (the number of copy statements that cannot be coalesced)

Table 3: Combination of optimizations

optimization 1	copy propagation, constant propagation, dead code elimination, empty block elimination
optimization 2	remove critical edge, loop invariant code motion, constant propagation, common subexpression elimination, copy propagation, constant propagation, dead code elimination, empty block elimination

in the object code, and the number of variables that are spilled in the register allocation phase. This gives insight into how copy statements that cannot be coalesced affect the object code and register allocation. (The *static* number means the number at compile time or the number in the text of object code. It contrasts with the *dynamic* number that represents the number at execution time.)

Next, we made a *dynamic* evaluation by measuring the number of executed move instructions, and the number of executed load and store instructions. This is to check the actual influence of static measurements on runtime measurements.

We finally measured the execution time as the final criterion.

As for benchmarks, we used `gzip`, `vpr`, `mcf`, `parser`, `bzip2`, `twolf` from SPEC CINT2000. The environment of the experiments is the same as for the first experiments.

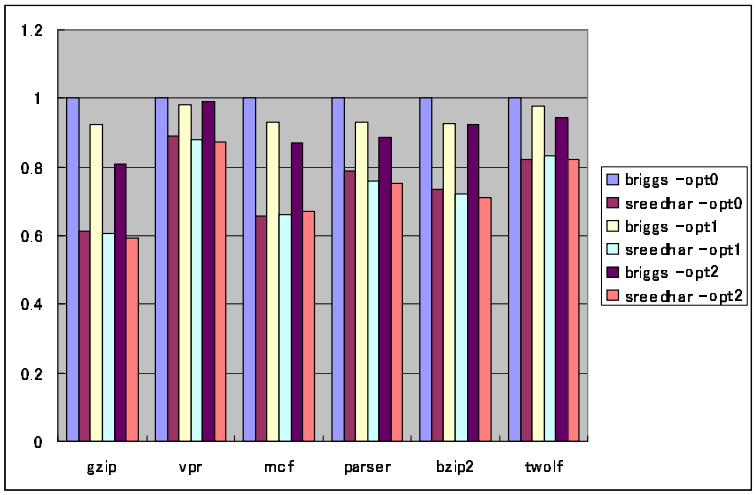


Figure 13: Ratios of the numbers of moves with eight registers

7.2 Comparisons of the static numbers of move instructions in the object code

To examine the differences in back-translation algorithms, we measured the static number of move instructions in the object code. This measurement helps us understand how the differences in the number of copy statements that cannot be coalesced and the process of uniting parameters in a ϕ-function (making them a single variable in Sreedhar's method) affect the object code.

Note that the number of move instructions in this section is static. We therefore cannot judge the relative superiority of algorithms at this stage.

7.2.1 With eight registers

Figure 13 shows the relative ratios of the number of move instructions in the object code when the number of registers is limited to eight (small values are better, the reference value is the case of no optimization in Briggs' method,

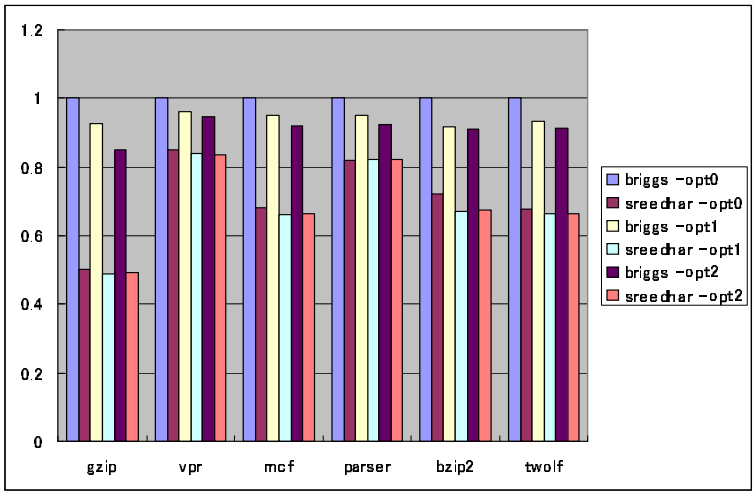


Figure 14: Ratios of the numbers of moves with 20 registers

which is normalized to one.)

We can see that the number of move instructions in Sreedhar's method is 60%˜90% of that of Briggs' method. From this, we confirm that in Briggs' method many copy statements that cannot be coalesced are inserted. We can anticipate that the dynamic number of move instructions at execution time will be larger with Briggs' method. This will be the subject of a later experiment.

Comparing the combination of optimizations, we see that in Briggs' method the number of move instructions becomes less as we perform more optimizations. This is in contrast to Sreedhar's method where the number does not generally change. We think this is because the process of uniting in Sreedhar's method has an effect similar to a kind of optimization.

7.2.2 With 20 registers

The result of the same experiment for 20 registers is shown in Fig. 14.

We can see that the number of move instructions in Sreedhar's method is about 50%˜80% of that of Briggs' method. The difference is much greater than with eight registers.

When more optimizations are applied, the number of move instructions decreases in Briggs' method, which is similar to the case of eight registers. This is in contrast to Sreedhar's method where the number does not generally change. This behavior can be explained similarly to the case of eight registers.

7.3 Comparison of the number of spills

We counted the (static) number of variables that are spilled out at the register allocation phase, to investigate how the differences in SSA back-translation algorithms affect register allocations, through the differences in the number of copy statements that cannot be coalesced, and the uniting process of the parameters in ϕ-functions. Note however that we cannot judge the superiority of algorithms at this stage, because this number is static and does not represent the dynamic cost of spills, similarly to the comparison of the number of move instructions.

7.3.1 With eight registers

Figure 15 shows the number of variables that are spilled at the register allocation phase when the number of registers is limited to eight.

We can see that fewer variables are spilled out with Sreedhar's method than with Briggs' method. We assume this is because registers are uselessly consumed in Briggs' method because of copy statements that cannot be coalesced. From this, we can anticipate that the dynamic number of load and store instructions at execution time will be bigger in Briggs' method. This will be the subject of a later experiment.

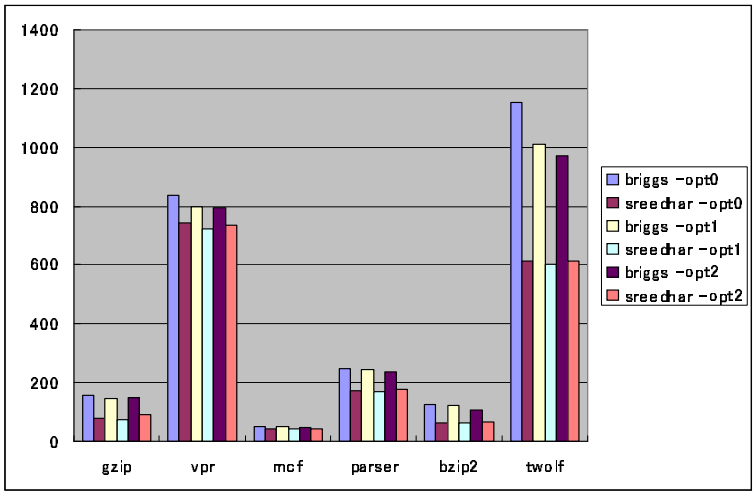


Figure 15: Number of spills with eight registers

7.3.2 With 20 registers

Figure 16 shows the number of variables that are spilled at the register allocation phase when the number of registers is 20.

We can see that fewer variables are spilled out with Sreedhar's method than with Briggs' method, similarly to the case of eight registers. As well, there are fewer spills than with eight registers. From this, we can anticipate that the influence of spills on the dynamic number of load and store instructions at execution time with 20 registers will be less than with eight registers. We will also confirm this in a later experiment.

7.4 Comparison of the number of executed move instructions

In the static measurements, the number of move instructions in Sreedhar's method was less than that in Briggs' method. Here, we measure the dynamic count of move instructions of the object code at execution time.

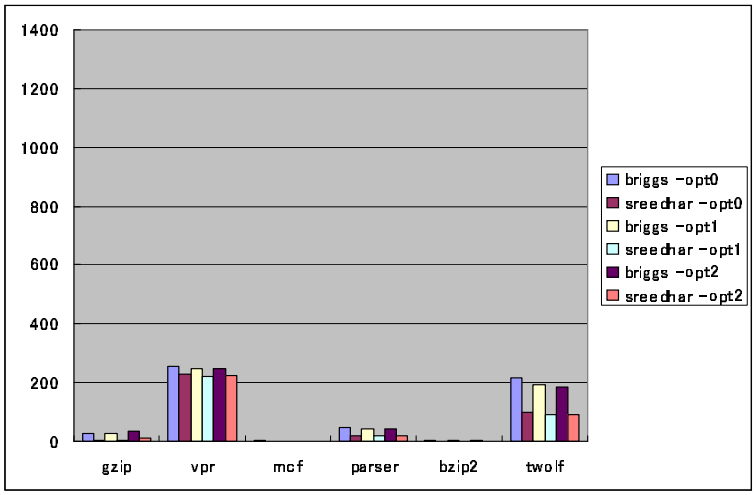


Figure 16: Number of spills with 20 registers

7.4.1 With eight registers

Figure 17 shows the relative ratio of the dynamic count of executed move instructions when the number of registers is limited to eight (small values are better, the reference value is the case of no optimization in Briggs' method, which is normalized to one.)

We can see that the dynamic count of executed move instructions in Sreedhar's method is 10% of that of Briggs' method in `gzip` and about 50% 80% of that of Briggs in others.

Changing the combination of optimizations shows the same tendencies as in the static measurements. The number of executed move instructions does not generally change by optimizations in Sreedhar's method.

7.4.2 With 20 registers

Figure 18 shows the relative ratios of the dynamic count of executed move instructions when the number of registers is 20.

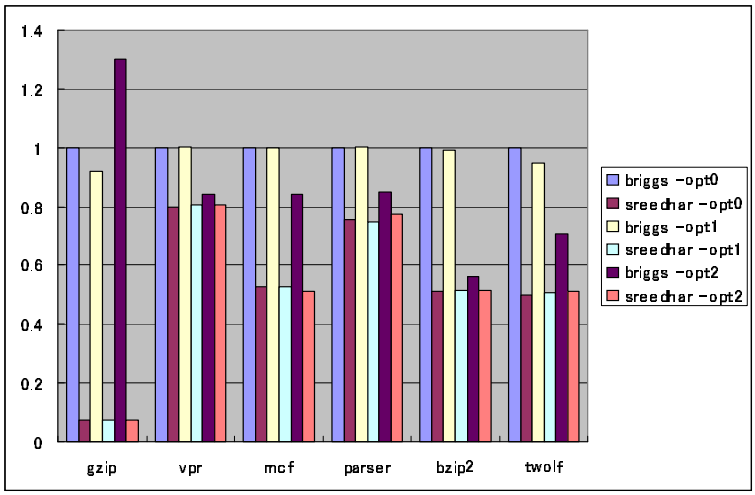


Figure 17: Ratios of the numbers of executed moves with eight registers

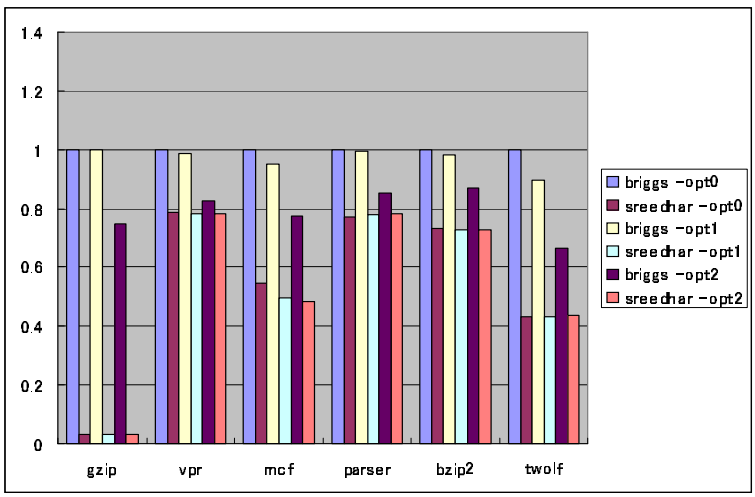


Figure 18: Ratios of the numbers of executed moves with 20 registers

We can see that the dynamic count of executed move instructions in Sreedhar's method is 3% in gzip and about 40% 80% in others, compared to Briggs' method. This shows greater difference than with eight registers, similarly to the static measurement.

Comparison by changing the combination of optimizations shows that the number of executed move instructions in Sreedhar's method does not generally change.

7.5 Comparison of the number of executed load and store instructions

In static measurements, Sreedhar's method has less spills. However, in Sreedhar's method, the parameters of the ϕ-functions are united. This may lengthen the live range of variables involved in ϕ-functions, and may result in increasing the cost of spills per variable.

Therefore, we measured the dynamic count of executed load and store instructions of the object code, as an approximation of the cost of spills at execution time.

7.5.1 With eight registers

Figure 19 shows the relative ratio of the dynamic count of executed load and store instructions of the object code when the number of registers is limited to eight (small values are better, the reference value is the case of no optimization in Briggs' method, which is normalized to one.)

We can see that Sreedhar's method is favorable in four out of six benchmarks while Briggs' method is favorable in two of the benchmarks. In particular, Sreedhar's method is quite advantageous in `gzip`.

Moreover, although the number of spilled variables was absolutely less

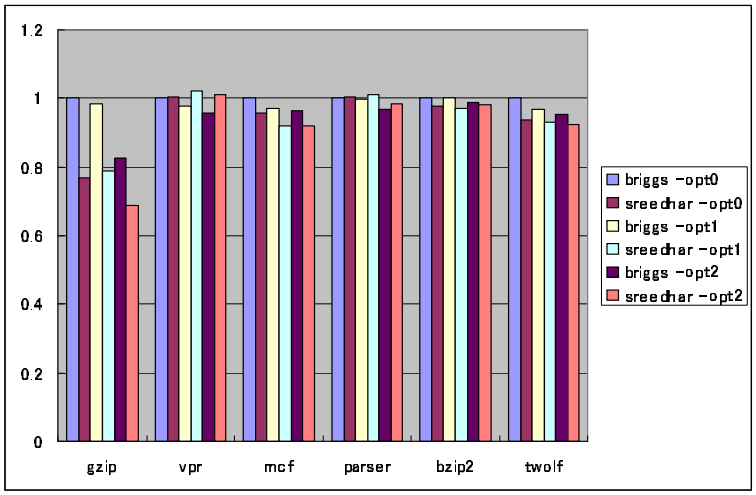


Figure 19: Ratios of the numbers of executed loads and stores with eight registers

with Sreedhar's method than with Briggs' method, the difference in the numbers of executed load and store instructions in both methods is reduced or partly reversed. Therefore, we can assume that the cost of spills per variable is higher with Sreedhar's method.

Even if the combination of optimizations is changed, the difference in the numbers of executed load and store instructions in both methods is preserved.

7.5.2 With 20 registers

Figure 20 shows the relative ratio of the dynamic count of executed load and store instructions with 20 registers. Little difference is found, as opposed to the case of eight registers. This is because the difference in the numbers of spilled variables in Fig. 16 is less than in Fig. 15.

Thus, because there is almost no difference in the number of executed load and store instructions in both methods, we expect that the execution time with 20 registers depends mainly on the difference of the numbers of

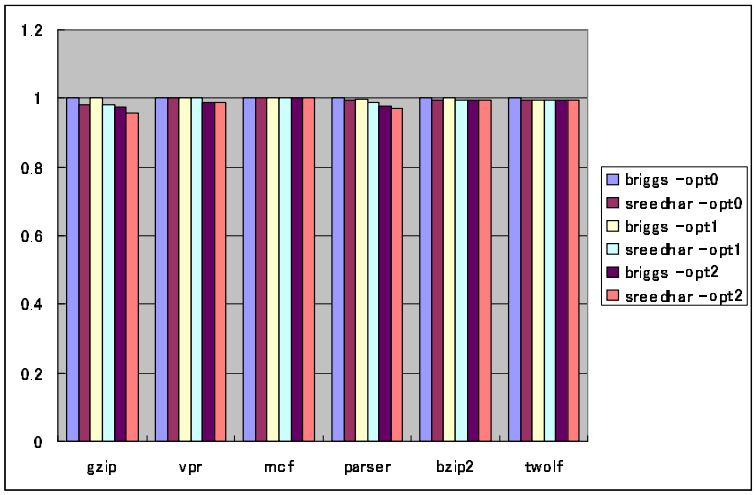


Figure 20: Ratios of the numbers of executed loads and stores with 20 registers

executed move instructions.

7.6 Comparison of execution times

Lastly, we present the comparison of execution time of the object code.

7.6.1 With eight registers

Figure 21 shows the relative ratios of the execution times with eight registers (small values are better, the reference value is the case of no optimization in Briggs' method, which is normalized to one.)

The execution speed of the object code created with Sreedhar's method is generally faster than that with Briggs' method, by about 28% in `gzip`, and a little faster in all the others. We discuss scattering in execution time and the relative ratios in some detail later.

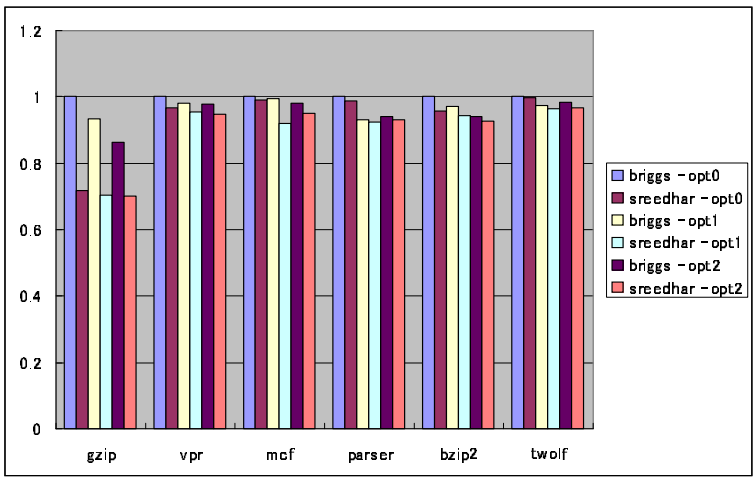


Figure 21: Ratios of execution times with eight registers

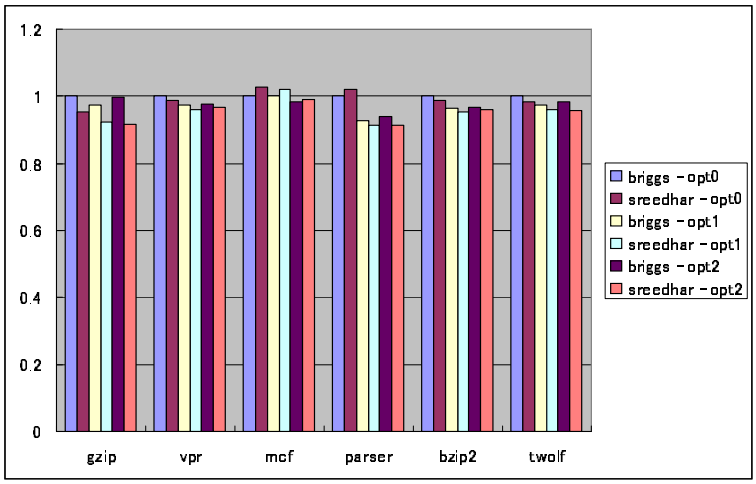


Figure 22: Ratios of execution times with 20 registers

7.6.2 With 20 registers

Next, Fig. 22 shows the relative ratios of the execution time with 20 registers. The execution speed of the object code from Sreedhar's method is faster in most benchmarks (by as much as 8%) compared to that from Briggs' method, except with all combinations of optimizations in `mcf` and the no-optimization case in `parser`, where the object code from Sreedhar's method is slower by up to about 3%.

In the comparison of execution times, the results mostly coincide with the previous analysis. However, the results do differ partly in showing some fluctuation or scattering.

We think that these fluctuations of the results are mainly because of the characteristics of the target architecture such as superscalar and cache lines, rather than the effect of SSA back-translation algorithms.

One possible reason relates to instruction fetch cycles. For example, if there is a branch instruction in a loop, the processor tries to fetch a sequence of next instructions using branch prediction, while processing this branch instruction. If instructions to be fetched reside in one cache line, they can be fetched together at once. If they straddle the boundaries of cache lines, however, fetching takes extra time.

In fact, when we have re-aligned instructions in a loop that takes most of the execution time according to the cache lines, the execution time sometimes decreases by 4˜5% [14]. This kind of internal processing mechanism has particular influence in superscalar machines, such as the one used in this experiment.

Therefore, for the execution time of the object code, Sreedhar's method is favorable on average, although in some cases the effect of the difference of back-translation algorithms is hidden by other factors.

In summary, with eight registers, Sreedhar's method achieves shorter execution time of the object code than Briggs' method. This is considered to be because of the smaller dynamic cost of spills and the smaller number of executed move instructions.

On the other hand, with 20 registers, Sreedhar's method achieves shorter execution time of the object code in most cases, although there are a few exceptions. This is considered to be because of the smaller number of executed move instructions required.

Moreover, the differences in SSA back-translation algorithms are not generally influenced by changing the combination of optimizations.

8 Discussion on instruction scheduling

In our experiments, no instruction scheduling was performed, as mentioned in Section 6.1. However, as most modern processors provide instruction-level parallel processing, its influence on performance is not negligible. In addition, it is known that instruction scheduling and register allocation interfere with each other. That is, if we decrease the number of actual registers in the register allocation, the parallelism among instructions is hampered. On the other hand, if we want to enhance parallelism by instruction scheduling, the register pressure will increase.

The consequence for our experiment is that if we increase the reusability of registers by coalescing, the dependencies among instructions will increase, and this may decrease the instruction-level parallelism. In processors performing out-of-order execution, this may hinder the exchange of instructions.

Because register allocation and instruction scheduling depend on each other, many studies have been carried out concerning the order of their ap-

plication and their cooperative processing [12].

In the COINS compiler, an experimental implementation of instruction scheduling was made, and application in the order of "instruction scheduling → register allocation → instruction scheduling" was tested, and gave good runtime performance in small examples.

In the experiments described in this paper, we have evaluated the effect of two major SSA back-translation algorithms, Briggs' method and Sreedhar's method, with register allocation. Even if one back-translation algorithm with register allocation gives a better result than the other, it is possible that the advantage will not be preserved if the instruction scheduling phase is performed. Therefore, evaluation using instruction scheduling will be necessary in the future. Besides, in processors that perform register renaming by hardware, evaluation from a different point of view will be required.

9 Related work

Our work aims at empirically comparing SSA back-translation algorithms. As far as we know, there have been no similar studies. Therefore, we present other SSA back-translation algorithms in this section.

9.1 Morgan's SSA back-translation

First, we briefly explain the back-translation algorithm by Morgan[16]. This algorithm replaces ϕ-functions by copy statements, similarly to Briggs' method.

In Morgan's method, a graph like Fig. 23 is used. In this graph, nodes represent variables, and directed edges are drawn from the targets (left-hand side variables) to the parameters of ϕ-functions.

In this graph, variables (nodes) having no predecessor nodes do not appear in the parameters of ϕ-functions. This means that such a variable can

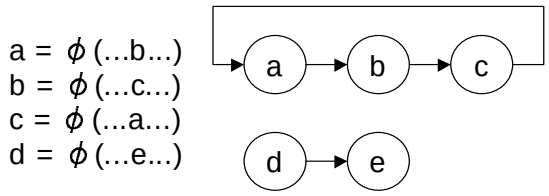


Figure 23: Morgan's method

be overwritten by back-translation, and a copy statement "*that variable = variable of its successor node*" can be safely inserted.

In addition, if this graph has a cycle, it will lead to dependency among copy statements to be inserted, which must be resolved. The cyclic dependency is cut by using a temporary variable.

For example in Fig. 23, because d has no predecessor node, "$d = e$" can be inserted. Next, because $a \rightarrow b \rightarrow c \rightarrow a$ has a cycle, we first insert "$temp = a$" to save the value of a, and then insert copy statements along the directed edges. The resulting copy statements are as follows.

$$d = e$$
$$temp = a$$
$$a = b$$
$$b = c$$
$$c = temp$$

Briggs' method can also deal with such a case, and in certain forms of cycles, Briggs' method inserts fewer copy statements. Therefore, Morgan's method is actually a subset of Briggs' method.

9.2 SSA back-translation by Budimlic et al.

Next, we briefly present the SSA back-translation by Budimlic et al. [4] This back-translation algorithm was developed mainly for just-in-time compilers. Unlike the algorithms that are the subjects of our experiments, it aims at reducing compilation time.

The basic idea is to perform the back-translation by uniting parameters of ϕ-functions, similarly to Sreedhar's method. This algorithm first approximates live ranges using the properties of the program represented in SSA form. The number of copy statements inserted in their method is expected to be greater than in Sreedhar's method, because removing the interference of the ϕ-functions is made using these approximate live ranges, and the classification of cases of interference is simpler than in Sreedhar's method.

10 Conclusions

Two major algorithms exist for the back-translation from SSA form to normal form. However, there have been no previous studies that empirically compare them and investigate their characteristics in depth. As a result, many compilers using the SSA form simply adopted the method by Briggs et al. without much consideration, because it was the first solution of the critical problems of the early naive algorithm.

In this paper, we clarified the merits and demerits of different SSA back-translation algorithms, and validated them through experiments. We showed that the selection of the back-translation algorithm affects the runtime efficiency of the object code in no small way, which has not previously received attention. The effect is comparable to applying a middle-level global optimization. A major contribution of our work was to give criteria for selecting

the SSA back-translation algorithm in future compilers.

The main results are the following:

- In Briggs' method, a large number of copy statements that cannot be coalesced are inserted. They affect the performance of the object code more than the increase of register pressure that is a concern in Sreedhar's method.

- When there are relatively few allocatable registers, Sreedhar's method, which performs uniting to the ϕ-functions, is superior because the dynamic cost of spills and the number of executed copy statements are less. Its execution time is better than Briggs' method by a few percent in general, and by 28% at maximum.

- When there are relatively many allocatable registers, Sreedhar's method is favorable in most cases, because the number of executed copy statements is low. Its execution time is better than Briggs' method by a few percent in most cases.

Further experiments using a variety of compiler phases such as instruction scheduling and its analysis are left for future research.

Acknowledgments

This research was partially supported by the Japanese Ministry of Education, Culture, Sports, Science and Technology under the Grant "Special Coordination Fund for Promoting Science and Technology" and by the Japan Society for the Promotion of Science under the Grant-in-Aid for Scientific Research, and by the Kayamori Foundation for Informational Science Advancement.

References

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

[2] Briggs, P., Harvey, T., Simpson, T. Static single assignment construction, version 1.0, `ftp://ftp.cs.rice.edu/public/compilers/ai/SSA.ps`, January 1996.

[3] Briggs, P., Cooper, K. D., Harvey, T. J., Simpson, L. T. 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] Budimlic, Z., Cooper, K. D., Harvey, T. J., Kennedy, K., Oberg, T. S., Reeves, S. W. Fast Copy Coalescing and Live-Range Identification, *Proceeding of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pp. 25–32, 2002.

[5] Chaitin, G. J. Register Allocation & Spilling via Graph Coloring, *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pp. 98–105, 1982.

[6] The Coins Project. Research of a Common Infrastructure for Parallelizing Compilers. `http://www.coins-project.org/`.

[7] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., Zadeck, F. K. 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, 1991.

[8] Fitzgerald, R., Knoblock, T. B., Ruf, E., Steensgaard, B., Tarditi, D. Marmot: An Optimizing Compiler for Java. *Software – Practice and Experience*, Vol. 30, No. 3, pp. 199–232, 2000.

[9] GCC: GCC Homepage. `http://gcc.gnu.org/`.

[10] George, L., Appel, A. W. Iterated Register Coalescing. *ACM Transactions on Programming Languages and Systems*, Vol. 18, No. 3, pp. 300–324, 1996.

[11] IBM: Jikes Research Virtual Machine. `http://jikesrvm.sourceforge.net/`.

[12] Inagaki, T., Komatsu, H., Nakatani, T. Integrated Prepass Scheduling for a Java Just-in-time Compiler on the IA-64 Architecture. In *Proc. International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, pp. 159–168, 2003.

[13] Ishizaki, K., Takeuchi, M. et al. Effectiveness of Cross-platform Optimizations for a Java Just-in-time Compiler. *OOPSLA '03: Proceedings of the 18th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pp. 187–204, 2003.

[14] Kohama, M. Comparison and Evaluation of SSA Normalization Algorithms. *Master's Thesis, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology*, (in Japanese), 2004.

[15] MachSUIF Homepage. `http://www.eecs.harvard.edu/machsuif/`.

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

[17] Sassa, M., Nakaya, T., Kohama, M., Fukuoka, T., Takahashi, M. Static Single Assignment Form in the COINS Compiler Infrastructure. *Proc. SSGRR 2003w*, 2003.

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

[19] Sreedhar, V. C., Gao, G. R. A Linear Time Algorithm for Placing ϕ-nodes. *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 62–73, 1995.

[20] Sreedhar, V. C., Ju, R. D.-C., Gillies, D. M., Santhanam, V. Translating Out of Static Single Assignment Form. *Proceedings of the 6th International Symposium on Static Analysis*, *Lecture Notes in Computer Science*, Vol. 1694, pp. 194–210, Springer-Verlag, 1999.