A Guess-and-Assume Approach to Loop Fusion for Program Verification

Akifumi Imanishi

Kyoto University

Japan

imanishi@fos.kuis.kyoto-u.ac.jp

Kohei Suenaga

Kyoto University

Japan

ksuenaga@gmail.com

Atsushi Igarashi

Kyoto University

Japan

igarashi@kuis.kyoto-u.ac.jp

Abstract

Loop fusion—a program transformation to merge multiple consecutive loops into a single one—has been studied mainly for compiler optimization. In this paper, we propose a new loop fusion strategy, which can fuse *any* loops—even loops with data dependence—and show that it is useful for program verification because it can simplify loop invariants.

The crux of our loop fusion is the following observation: if the state after the first loop were known, the two loop bodies could be computed at the same time without suffering from data dependence by renaming program variables. Our loop fusion produces a program that *guesses* the unknown state after the first loop nondeterministically, executes the fused loop where variables are renamed, compares the guessed state and the state actually computed by the fused loop, and, if they do not match, diverges. The last two steps of comparison and divergence are crucial to preserve partial correctness. We call our approach “guess-and-assume” because, in addition to the first step to guess, the last two steps can be expressed by the pseudo-instruction *assume*, used in program verification.

We formalize our loop fusion for a simple imperative language and prove that it preserves partial correctness. We further extend the “guess-and-assume” technique to reversing loop execution, which is useful to verify a certain type of consecutive loops. Finally, we confirm by experiments that our transformation techniques are indeed effective for state-of-the-art model checkers to verify a few small programs that they could not.

CCS Concepts • Theory of computation → Logic and verification; Verification by model checking; Program verification; Hoare logic; Invariants; • Software and its engineering → Formal software verification;

Keywords guess-and-assume, loop fusion, program verification

ACM Reference Format:

Akifumi Imanishi, Kohei Suenaga, and Atsushi Igarashi. 2018. A Guess-and-Assume Approach to Loop Fusion for Program Verification. In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM’18)*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3162070>

1 Introduction

Loop fusion [2, 10] is a program transformation to merge multiple consecutive loops into a single one. For example, the following C code

```
for (i = 0; i < n; i++) { sum1 += i; }
for (i = 0; i < n; i++) { sum2 += i; }
```

is transformed to

```
for (i = 0; i < n; i++) {
    sum1 += i;
    sum2 += i;
}
```

by loop fusion. Although loop fusion has been studied mainly for compiler optimization, in this paper, we investigate an application of loop fusion to static program verification, in particular, *safety verification* (i.e., all the assertions in a verified program do not fail).

Loop fusion can be useful because a fused loop may have a simpler loop invariant condition. In order to motivate loop fusion for static verification, consider the following C function, which includes the two loops above:

```
void sample_01(int n) {
    int sum1 = 0, sum2 = 0;
    int i;
    for (i = 0; i < n; i++) { sum1 += i; }
    for (i = 0; i < n; i++) { sum2 += i; }
    assert (sum1 == sum2);
}
```

Although it is easy to see that the assertion at the end always holds for any argument n , it is actually hard for static program verifiers to prove it. In fact, the state-of-the-art software model checkers such as CPACHECKER [4] and SEAHORN [14] fail to verify this function in 900 seconds in our experimental environment (see Section 5 for details). It is perhaps because loop invariants to prove the assertion involve nonlinear arithmetic: $(\text{sum1} = \frac{i(i-1)}{2}) \wedge (i \leq n) \wedge (\text{sum2} = 0)$ for the first loop

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike International 4.0 License.

PEPM’18, January 8–9, 2018, Los Angeles, CA, USA

© 2018 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-5587-2/18/01.

<https://doi.org/10.1145/3162070>

and $(\text{sum1} = \frac{n(n-1)}{2}) \wedge (i \leq n) \wedge (\text{sum2} = \frac{i(i-1)}{2})$ for the second loop. On the other hand, if we replace the two loops with the fused single loop

```
void sample_01_fused(int n) {
  int sum1 = 0, sum2 = 0;
  int i;
  for (i = 0; i < n; i++) {
    sum1 += i;
    sum2 += i;
  }
  assert (sum1 == sum2);
}
```

there is actually a much simpler loop invariant $\text{sum1} = \text{sum2}$ and both CPACHECKER and SEAHORN successfully prove the absence of assertion failure within a few seconds!

However, applicability of standard techniques for loop fusion is rather limited. For example, loop fusion found in Bacon et al. [2] imposes the following two conditions to preserve the semantics of the program:

1. The two loops have the same loop bounds; and
2. The body of the first loop does not depend on that of the second.

So, the following two loops, in which the same variable sum is updated, do not satisfy the condition above and so cannot be fused.¹

```
for (i = 0; i < n; i++) { sum += i; }
for (i = 0; i < n; i++) { sum -= i; }
```

We propose a program transformation that addresses this problem. Our transformation is based on the following observation: One can easily fuse *any* two loops if we, at the beginning of the program execution, could predict the values kept by the variables when the program execution reaches the point between the loops. We explain this observation by step-by-step transformations of the following program P that includes the loops above:

```
void sample_02(int n) {
  int sum = 0;
  int i;
  for (i = 0; i < n; i++) { sum += i; }
  for (i = 0; i < n; i++) { sum -= i; }
  assert(sum == 0);
}
```

We first try to resolve the dependency between the bodies of the two loops. An easy way to achieve this is to introduce a new variable sum2 corresponding to sum and replace the occurrence of sum in the second loop with that of sum2 .

```
int sum = 0; int sum2 = 0;
int i;
for (i = 0; i < n; i++) { sum += i; }
/* (X) */
for (i = 0; i < n; i++) { sum2 -= i; }
assert(sum == 0);
```

¹Actually, naively applying loop fusion to these loops preserves the semantics and it would be possible for model checkers to verify the fused loop; of course, it is not the case in general.

Although we can fuse the two loops in this resulting program with a standard technique, this does not serve for our purpose. This program breaks the relation between the first loop and the second loop in the original program and thus is semantically different from P . Therefore, verifying the resulting program does not imply the safety of P .

In order to recover this relation between the two loops, we introduce variable sum_middle to represent the value of sum when the control reaches Line (X).

```
int sum = 0;
int sum2;
int i;
for (i = 0; i < n; i++) { sum += i; }
assume(sum_middle == sum);
sum2 = sum_middle;
/* (Y) */
for (i = 0; i < n; i++) { sum2 -= i; }
sum = sum2;
assert(sum == 0);
```

Let us call this program Q . Suppose that we can somehow predict the value of sum_middle *before starting the execution of* Q . Then, the safety of Q —the absence of the failure of assert —implies that of P because

- The values of sum and sum2 on Line (Y) are equal to sum_middle (thanks to the prediction) and
- The value of sum2 is copied to sum before $\text{assert}(\text{sum} == 0)$ is executed.

Here, the statement $\text{assume}(\text{sum} == \text{sum_middle})$ works as no-op if $\text{sum} = \text{sum_middle}$ holds; it diverges if $\text{sum} \neq \text{sum_middle}$ does not hold so that execution does not continue. This command is used for example in Boogie [3] to express assumptions used in verification as a program.

The last problem is how to predict the value of sum_middle at the beginning. Quite surprisingly, for our purpose of safety verification, it turns out that just *guessing* the value suffices:

```
int sum_middle = havoc();
int sum = 0, sum2 = sum_middle, i;
for (i = 0; i < n; i++) { sum += i; }
for (i = 0; i < n; i++) { sum2 -= i; }
assume(sum == sum_middle);
sum = sum2;
assert(sum == 0);
```

The command $\text{havoc}()$ nondeterministically chooses a value and returns it. We have pushed down the assume statement, which does not use sum2 , and pulled up $\text{sum2} = \text{sum_middle}$, which does not use sum . Since the two loops in the program now meet the conditions for loop fusion, we can fuse these loops to obtain the following program:

```
void sample_02_fused(int n) {
  int sum_middle = havoc();
  int sum = 0, sum2 = sum_middle, i;
  for (i = 0; i < n; i++) {
    sum += i; sum2 -= i;
  }
  assume(sum == sum_middle);
  sum = sum2;
  assert(sum == 0);
}
```

In retrospect, the resulting program first *guesses* the value of `sum` at Line (X) in P , executes the program, and then *assumes* that the guessed value is indeed a correct guess using `assume` to exclude an execution that is not possible in P . The main contribution of this paper is to formalize loop fusion using this *guess-and-assume* approach, to prove its soundness in safety verification, and to demonstrate its usefulness empirically.

Notice that the resulting program `sample_02_fused` is *not quite semantically equivalent* to the original program `sample_02`: If the guessed value of `sum_middle` is different from the actual value of `sum` in the middle of two loops of the original program, then the function `sample_02_fused` diverges whereas `sample_02` always terminates. This difference, however, is not problematic in this case because our focus is safety verification which is partial correctness and therefore only equivalence up to termination is required for a transformation.

In addition to the loop fusion, we show another application of the guess-and-assume approach: Loop reversal. We propose a program transformation that translates a loop to another loop whose iteration pattern is the reversal of the original. This transformation is useful in verifying a program like the following one.

```
int sum1 = 0, sum2 = 0;
for (i = 0; i < n; i++) { sum1 += i; }
for (i = 0; i < n; i++) { sum2 += n - i - 1; }
assert (sum1 == sum2);
```

To this program, normal loop fusion is less effective because an invariant for the fused loop is not very simple; the fused program cannot be verified within 900 seconds with CPACHECKER or SEAHORN. However, if we reverse the direction of the iterations of the second loop before fusion and applies simple postprocessing transformation, then the fused program can be verified in approximately 20 secs (by CPACHECKER) or less than a second (by SEAHORN).

We summarize our contribution as follows.

- We formalize loop fusion whose applicability is less limited than the methods proposed so far. Our transformation is based on a novel guess-and-assume approach. Although a program generated by this transformation is equivalent to the original one only up to termination, this transformation is often useful to make static verification of partial correctness faster. We show the correctness of the transformation.
- We propose a loop reversal transformation based on the guess-and-assume approach. It transforms a loop into another loop whose iteration behavior is the reversal of the original one. We show correctness of this transformations, as well.
- We implement the proposed two algorithms and conducted experiments using state-of-the-art static program verifiers: CPACHECKER and SEAHORN. We compared their performance on programs that contain consecutive loops that these verifiers fail to prove their

correctness with ones that are preprocessed by our transformation. We observe improvement of the performance in many cases.

The rest of the paper is structured as follows. Section 2 defines our target language, which is a simple imperative language extended with commands to guess and assume, its operational semantics, and a correctness criterion of our program transformation. Section 3 defines our loop fusion and proves its correctness. Section 4 formalizes the loop reversal and proves its correctness. Section 5 describes our implementation and shows results of our experiments. Section 6 discusses related work and Section 7 gives concluding remarks with discussion on future work. We often omit straightforward proofs.

2 Target Language

In this section, we formalize the syntax and the operational semantics of the target language, which is a variant of the IMP language [28]. The formalization is mostly standard except that we will make program variables used in programs more explicit throughout the formalization, in order to make it easier to state correctness of our loop fusion.

2.1 Syntax

We fix the set of n -ary operations Op_n for each nonnegative integer n (if $n = 0$ they are considered as constant values). For each finite set of variables V , the set Exp_V of expressions e_V and the set Com_V of commands P_V are defined as follows:

Definition 2.1 (Expressions and Commands).

$$\begin{aligned}
 e_V &\in Exp_V \stackrel{\text{def}}{=} x \mid f_n(\underbrace{e_V, \dots, e_V}_n) \\
 P_V &\in Com_V \stackrel{\text{def}}{=} \text{skip} \mid x := e_V \mid P_V; P_V \\
 &\quad \mid \text{if } e_V \text{ then } P_V \text{ else } P_V \\
 &\quad \mid \text{while } e_V \text{ do } P_V \text{ done} \\
 &\quad \mid \text{havoc } x \mid \text{assume } e
 \end{aligned}$$

(x and f_n range over V and Op_n , respectively.)

We do not have Boolean expressions; we use integer expressions for conditions of `if` and `while` statements and regard any nonzero value as true. `havoc  $x$` is the command that assigns a random integer to the variable x nondeterministically. `assume  $e$` is to assume an additional condition that e evaluates to a nonzero value. Operationally, it is a no-op if e evaluates to true; otherwise it diverges. The remaining constructs are the same as the usual while-language. We assume usual operations such as $\neg \in Op_1$, $= \in Op_2$, and $\vee \in Op_2$ (with standard semantics) and use $=$ and $\vee$ as infix operators.

We could dispense with `assume  $e$` because it can be expressed by a combination of other commands, but we include it as a primitive for simplicity. We do not include the

command assert, used in examples, because it is not needed for technical development.

2.2 Operational Semantics

We define a big-step operational semantics of the target language. The semantics is given by using the relation $P, \sigma_1 \Downarrow \sigma_2$, read “execution of command P in state σ_1 terminates in final state σ_2 ,” where P is a command and σ_1 and σ_2 are states. Below we assume for each n -ary operation f_n , a map from $\mathbb{Z}^n$ to $\mathbb{Z}$ (also denoted by f_n) is assigned.

Definition 2.2 (States). A state σ in State_V is a map from V to $\mathbb{Z}$. We write $\sigma|_W$ for the restriction of σ to the domain W .

Definition 2.3 (Semantics of Expressions). The semantics $\sigma \llbracket e \rrbracket$ of expression e under state σ is defined by:

$$\begin{aligned} \sigma \llbracket x \rrbracket & \stackrel{\text{def}}{=} \sigma(x) \\ \sigma \llbracket f_n(e_1, \dots, e_n) \rrbracket & \stackrel{\text{def}}{=} f_n(\sigma \llbracket e_1 \rrbracket, \dots, \sigma \llbracket e_n \rrbracket) \end{aligned}$$

Definition 2.4 (State Update). For a state $\sigma \in \text{State}_V$, a $x \in V$, and an $a \in \mathbb{Z}$, the state $\sigma[x \mapsto a] \in \text{State}_V$ is defined by:

$$\sigma[x \mapsto a](y) \stackrel{\text{def}}{=} \begin{cases} a & \text{if } x = y \\ \sigma(y) & \text{if } x \neq y \end{cases}$$

Definition 2.5 (Operational Semantics). For states $\sigma_1, \sigma_2 \in \text{State}_V$ and a command $P \in \text{Com}_V$, the relation $P, \sigma_1 \Downarrow \sigma_2$ is defined by the rules in Figure 1.

The rules are standard. In E_Havoc , an integer n is non-deterministically picked up. The rule E_Assume means that assume e is no-op if e is true (i.e., not zero). It diverges otherwise; so there is no rule for the case $\sigma \llbracket e \rrbracket = 0$.

2.3 Assertions and Hoare Triples

We define the assertions and the Hoare triples of the target language. For simplicity, assertions are semantically defined: we do not fix syntax for them and an assertion is given as a set of states.

Definition 2.6 (Assertions). An *assertion*, ranged over by φ and ψ , over V is a subset of State_V . We write Assertion_V for the set of assertions over V . In other words, the set Assertion_V is the power set of State_V .

We extend restrictions to assertions as follows: For sets of variables V and W such that $V \subseteq W$ and an assertion $\varphi \in \text{Assertion}_W$, we define an assertion $\varphi|_V \in \text{Assertion}_V$ as $\{\sigma_2 \mid \sigma_1 \in \varphi \wedge \sigma_1|_V = \sigma_2\}$.

Definition 2.7 (Hoare Triples). For a command $P \in \text{Com}_V$ and two assertions φ and ψ in Assertion_V , we write $\{\varphi\}P\{\psi\}$ if and only if the following condition is satisfied:

$$\forall \sigma_1, \sigma_2 \in \text{State}_V. (\sigma_1 \in \varphi) \wedge (P, \sigma_1 \Downarrow \sigma_2) \implies \sigma_2 \in \psi.$$

We introduce separating conjunction [24] for notational convenience. (Note that this is not a syntactic constructor but an operator on two assertions.)

Definition 2.8 (Separating conjunction). Let V and W be disjoint sets of variables. Let σ_1 and σ_2 be states in State_V , State_W respectively. We define $\sigma_1 \uplus \sigma_2 \in \text{State}_{V \cup W}$ as follows:

$$(\sigma_1 \uplus \sigma_2)(x) = \begin{cases} \sigma_1(x) & x \in V \\ \sigma_2(x) & x \in W. \end{cases}$$

We define the *separating conjunction* $\varphi * \psi$ of two assertions $\varphi \in \text{Assertion}_V$ and $\psi \in \text{Assertion}_W$ as follows:

$$\varphi * \psi = \{\sigma_1 \uplus \sigma_2 \mid \sigma_1 \in \varphi, \sigma_2 \in \psi\}$$

2.4 V-equivalence

In this section, we introduce the notion of V -equivalence to state correctness of our loop fusion. Usually, equivalence between P_1 and P_2 are defined by: for any pair of states σ_1 and σ_2 , $P_1, \sigma_1 \Downarrow \sigma_2$ if and only if $P_2, \sigma_1 \Downarrow \sigma_2$. We cannot, however, expect that our loop fusion yields an equivalent command in this sense because it introduces nondeterministically diverging behavior and extra variables. What we can expect is that the command after loop fusion will show the same input–output behavior as before—*only when it ever terminates and only for variables in the original command*. So, we index our equivalence by a set V of variables, corresponding to variables in the original command, and consider commands without strong dependency [7] of V on other variables.

Definition 2.9 (P depends only on V). For sets V and W of variables such that $V \subseteq W$, and $P \in \text{Com}_W$, we say P *depends only on V* if and only if: For any $\sigma_1, \sigma_2 \in \text{State}_W$ such that $\sigma_1|_V = \sigma_2|_V$, if $P, \sigma_1 \Downarrow \sigma'_1$ and $P, \sigma_2 \Downarrow \sigma'_2$, then $\sigma'_1|_V = \sigma'_2|_V$.

As the following lemma states, for a command depending only on V , one can weaken the precondition on variables other than V without affecting the postcondition on V .

Lemma 2.10. *Let $P \in \text{Com}_W$ be a command depending only on V . $\{\varphi\}P\{\psi * \text{State}_{W \setminus V}\}$ if and only if $\{\varphi|_V * \text{State}_{W \setminus V}\}P\{\psi * \text{State}_{W \setminus V}\}$.*

Proof. Easy. $\square$

Definition 2.11 (V -equivalence). Let two commands $P_1 \in \text{Com}_{V_1}$ and $P_2 \in \text{Com}_{V_2}$ depend only on V . P_1 and P_2 are V -equivalent, written $P_1 \sim_V P_2$, if and only if: For any assertions $\varphi, \psi \in \text{Assertion}_V$, $\{\varphi * \text{State}_{V_1 \setminus V}\}P_1\{\psi * \text{State}_{V_1 \setminus V}\}$ if and only if $\{\varphi * \text{State}_{V_2 \setminus V}\}P_2\{\psi * \text{State}_{V_2 \setminus V}\}$.

As Theorem 2.14 below shows, V -equivalence is closed under contexts that refer to only V . In the next section, we will formally define our loop fusion and prove that it yields a V -equivalent command (where V is the set of variables in the original command). The theorem ensures that applying loop fusion (to a subcommand) preserves partial correctness.

In what follows, a context C in Context_V is defined as a command ($\in \text{Com}_V$) which contains a hole $\square$ and $C[P]$ stands for the command obtained by replacing $\square$ in C with P .

$$\boxed{P, \sigma_1 \Downarrow \sigma_2}$$

$$\begin{array}{c}
\frac{}{\text{skip}, \sigma \Downarrow \sigma} \text{E_SKIP} \quad \frac{}{x:=e, \sigma \Downarrow \sigma[x \mapsto \sigma[e]]} \text{E_ASSIGN} \quad \frac{P_1, \sigma_1 \Downarrow \sigma_2 \quad P_2, \sigma_2 \Downarrow \sigma_3}{P_1; P_2, \sigma_1 \Downarrow \sigma_3} \text{E_SEQ} \\
\\
\frac{\sigma[e] \neq 0 \quad P_1, \sigma_1 \Downarrow \sigma_2}{\text{if } e \text{ then } P_1 \text{ else } P_2, \sigma_1 \Downarrow \sigma_2} \text{E_IFT} \quad \frac{\sigma[e] = 0 \quad P_2, \sigma_1 \Downarrow \sigma_2}{\text{if } e \text{ then } P_1 \text{ else } P_2, \sigma_1 \Downarrow \sigma_2} \text{E_IFF} \\
\\
\frac{\sigma[e] = 0}{\text{while } e \text{ do } P \text{ done}, \sigma \Downarrow \sigma} \text{E_WHILEEND} \quad \frac{\sigma[e] \neq 0 \quad P, \sigma_1 \Downarrow \sigma_2 \quad \text{while } e \text{ do } P \text{ done}, \sigma_2 \Downarrow \sigma_3}{\text{while } e \text{ do } P \text{ done}, \sigma_1 \Downarrow \sigma_3} \text{E_WHILELOOP} \\
\\
\frac{n \in \mathbb{Z}}{\text{havoc } x, \sigma \Downarrow \sigma[x \mapsto n]} \text{E_HAVOC} \quad \frac{\sigma[e] \neq 0}{\text{assume } e, \sigma \Downarrow \sigma} \text{E_ASSUME}
\end{array}$$

Figure 1. Operational Semantics.

Lemma 2.12. Let V, V_1 and V_2 be sets of variables such that $V \subseteq V_1$ and $V \subseteq V_2$ and $P_1, P_2 \in \text{Com}_{V_1}$ and $P_3, P_4 \in \text{Com}_{V_2}$. If P_i depends only on V for $i \in \{1, \dots, 4\}$ and $P_1 \sim_V P_3$ and $P_2 \sim_V P_4$, then $P_1; P_2 \sim_V P_3; P_4$.

Proof. We show that $\{\varphi * \text{State}_{V_1 \setminus V}\} P_1; P_2 \{\psi * \text{State}_{V_1 \setminus V}\}$ implies $\{\varphi * \text{State}_{V_2 \setminus V}\} P_3; P_4 \{\psi * \text{State}_{V_2 \setminus V}\}$ and omit the other direction. Let $\chi = \{\sigma' \mid \sigma \in \varphi * \text{State}_{V_1 \setminus V} \wedge P_1, \sigma \Downarrow \sigma'\}$. By definition, $\{\varphi * \text{State}_{V_1 \setminus V}\} P_1 \{\chi\}$ and it is easy to show that $\{\varphi * \text{State}_{V_1 \setminus V}\} P_1 \{\chi|_V * \text{State}_{V_1 \setminus V}\}$. Similarly, $\{\chi\} P_2 \{\psi * \text{State}_{V_1 \setminus V}\}$. Since P_1 depends only on V , we have $\{\chi|_V * \text{State}_{V_1 \setminus V}\} P_2 \{\psi * \text{State}_{V_1 \setminus V}\}$ by Lemma 2.10. We have $\{\varphi * \text{State}_{V_2 \setminus V}\} P_3 \{\chi|_V * \text{State}_{V_2 \setminus V}\}$ and $\{\chi|_V * \text{State}_{V_2 \setminus V}\} P_4 \{\psi * \text{State}_{V_2 \setminus V}\}$ by $P_1 \sim_V P_3$ and $P_2 \sim_V P_4$, respectively. $\square$

Lemma 2.13. For commands $P_1 \in \text{Com}_{V_1}$ and $P_2 \in \text{Com}_{V_2}$ such that $P_1 \sim_V P_2$ and $e \in \text{Exp}_V$, $\text{while } e \text{ do } P_1 \text{ done} \sim_V \text{while } e \text{ do } P_2 \text{ done}$.

Proof. For one direction, we show that, if $\{\varphi * \text{State}_{V_1 \setminus V}\} \text{while } e \text{ do } P_1 \text{ done} \{\psi * \text{State}_{V_1 \setminus V}\}$ and, for any $\sigma \in \varphi * \text{State}_{V_2 \setminus V}$, $\text{while } e \text{ do } P_2 \text{ done}, \sigma \Downarrow \sigma'$, then $\sigma' \in \psi * \text{State}_{V_2 \setminus V}$. This is shown by induction on the derivation of $\text{while } e \text{ do } P_2 \text{ done}, \sigma \Downarrow \sigma'$. The other direction is similar. $\square$

Theorem 2.14. For any C in Context_V and commands $P_1 \in \text{Com}_{V_1}$ and $P_2 \in \text{Com}_{V_2}$ such that $P_1 \sim_V P_2$, $C[P_1] \sim_V C[P_2]$.

Proof. By induction on structure of C . For sequential composition $P_1; P_2$, use Lemma 2.12, and for while, use Lemma 2.13. $\square$

3 Loop Fusion

In this section, we define our loop fusion transformation and show that it produces a V -equivalent command.

We begin with a few preliminary definitions.

Definition 3.1 (Prime notation). For a set of variables $V = \{x_1, \dots, x_n\}$, we write $V' = \{x'_1, \dots, x'_n\}$ as a set of variables

which is disjoint from V and has the same size as V . Given a command $P \in \text{Com}_V$, we also write P' for a command (in $\text{Com}_{V'}$) which is given by replacing each variable x_i in P with x'_i in V' .

Definition 3.2. For a set of variables $V = \{x_1, \dots, x_n\}$, we write $\vec{V} := \vec{V}'$ as an abbreviation of the command $x_1 := x'_1; \dots; x_n := x'_n$ and we write $\text{assume } \vec{V} = \vec{V}'$ as an abbreviation of the command $\text{assume } x_1 = x'_1; \dots; \text{assume } x_n = x'_n$.

3.1 Loop Fusion for Consecutive Two Loops

Definition 3.3 (Loop fusion for consecutive two loops). Let the following command be $P \in \text{Com}_V$:

$$P = \text{while } e_1 \text{ do } P_1 \text{ done}; \text{while } e_2 \text{ do } P_2 \text{ done}.$$

For such a command P , we define a command $\text{FUSION}[P] \in \text{Com}_{V \cup V' \cup V''}$, which consists of a single while loop, as follows:

$$\begin{aligned}
\text{FUSION}[P] = & \text{havoc } \vec{V}'; \\
& \vec{V}'' := \vec{V}'; \\
& \text{while } e_1 \vee e_2' \text{ do} \\
& \quad \text{if } e_1 \text{ then } P_1 \text{ else skip;} \\
& \quad \text{if } e_2' \text{ then } P_2' \text{ else skip} \\
& \text{done;} \\
& \text{assume } \vec{V} = \vec{V}''; \\
& \vec{V} := \vec{V}'
\end{aligned}$$

Theorem 3.4. For any $P \in \text{Com}_V$, $\text{FUSION}[P]$ does not depend on V .

Theorem 3.5. For any $P \in \text{Com}_V$, P and $\text{FUSION}[P]$ are V -equivalent.

Proof. By Figure 2. We define $P_1 \sim P_2$ by: $P_1 \sim P_2$ if and only if for any assertions φ, ψ , it holds that $\{\varphi\} P_1 \{\psi\}$ iff $\{\varphi\} P_2 \{\psi\}$. It is easy to show that $\sim$ is closed under contexts (regardless

of variables) and that if $P_1 \sim P_2$ and both P_1 and P_2 depend only on V , then $P_1 \sim_V P_2$.

In each step, we use lemmas to show V -equivalence or $\sim$ between certain subparts and then apply Theorem 2.14 or the properties of $\sim$ above. (It is easy to show that all intermediate commands depend only on V .) The first step, which is shown by Lemma 3.6, renames variables in the second loop. The second step, which is shown by Lemma 3.7, replaces assignments with havoc and assume. The third step, which is shown by Lemma 3.8, introduces $\vec{V}''$ to avoid the use of $\vec{V}'$, enabling the fourth step, which exchanges statements that use disjoint sets of variables and is shown by Lemma 3.9. Finally, we fuse the two consecutive loops, which refer to disjoint sets of variables; it is shown by Lemma 3.12. $\square$

Lemma 3.6. For any $e \in \text{Exp}_V$ and $P \in \text{Com}_V$,

$$\begin{array}{l} \vec{V}' := \vec{V}; \\ \text{while } e \text{ do } P \text{ done} \sim_V \text{while } e' \text{ do } P' \text{ done}; \\ \vec{V} := \vec{V}' \end{array}$$

Lemma 3.7. $\vec{V}' := \vec{V} \sim \text{havoc } \vec{V}'; \text{ assume } \vec{V} = \vec{V}'$.

Lemma 3.8.

$$\begin{array}{l} \text{assume } \vec{V} = \vec{V}' \sim_V \vec{V}'' := \vec{V}'; \\ \text{assume } \vec{V} = \vec{V}'' \end{array}$$

Lemma 3.9. If $P_i \in \text{Com}_{V_i}$ for $i \in \{1, 2\}$, and $V_1 \cap V_2 = \emptyset$, then $P_1; P_2 \sim P_2; P_1$.

Lemma 3.10. For $P \in \text{Com}_V$, $\sigma_1, \sigma_3 \in \text{State}_V$, $\sigma_2, \sigma_4 \in \text{State}_W$ such that $V \cap W = \emptyset$, if $P, \sigma_1 \uplus \sigma_2 \Downarrow \sigma_3 \uplus \sigma_4$, then $\sigma_2 = \sigma_4$ and $P, \sigma_1 \Downarrow \sigma_3$.

Lemma 3.11. Suppose $\text{while } e_i \text{ do } P_i \text{ done} \in \text{Com}_{V_i}$ and $V_1 \cap V_2 = \emptyset$ and let P be

$$\begin{array}{l} \text{while } e_1 \vee e_2 \text{ do} \\ \quad \text{if } e_1 \text{ then } P_1 \text{ else skip;} \\ \quad \text{if } e_2 \text{ then } P_2 \text{ else skip;} \\ \text{done} \end{array}$$

Then, $\text{while } e_i \text{ do } P_i \text{ done}, \sigma_i \Downarrow \sigma'_i$ for $i \in \{1, 2\}$ if and only if $P, \sigma_1 \uplus \sigma_2 \Downarrow \sigma'_1 \uplus \sigma'_2$.

Proof. ($\Leftarrow$): By induction on the sum of the depths of the derivations of $\text{while } e_i \text{ do } P_i \text{ done}, \sigma_i \Downarrow \sigma'_i$ for $i \in \{1, 2\}$.

($\Rightarrow$): By induction on the size of the depth of the derivation of $P, \sigma_1 \uplus \sigma_2 \Downarrow \sigma'_1 \uplus \sigma'_2$. $\square$

The following lemma shows correctness of classical loop fusion for loops without dependency.

Lemma 3.12. If $\text{while } e_i \text{ do } P_i \text{ done} \in \text{Com}_{V_i}$ for $i \in \{1, 2\}$ and $V_1 \cap V_2 = \emptyset$, then

$$\begin{array}{l} \text{while } e_1 \text{ do } P_1 \text{ done;} \\ \text{while } e_2 \text{ do } P_2 \text{ done} \end{array} \sim \begin{array}{l} \text{while } e_1 \vee e_2 \text{ do} \\ \quad \text{if } e_1 \text{ then } P_1 \text{ else skip;} \\ \quad \text{if } e_2 \text{ then } P_2 \text{ else skip;} \\ \text{done} \end{array}$$

Proof. Let P be the left-hand side command and P' the right-hand side.

($\Rightarrow$): Suppose $\{\varphi\}P\{\psi\}$ and $\sigma \in \varphi$ and $P', \sigma \Downarrow \sigma'$. We will show that $\sigma' \in \psi$. It is easy to show that there exist $\sigma_1, \sigma'_1 \in \text{State}_{V_1}$ and $\sigma_2, \sigma'_2 \in \text{State}_{V_2}$ such that $\sigma = \sigma_1 \uplus \sigma_2$ and $\sigma' = \sigma'_1 \uplus \sigma'_2$. By Lemma 3.11, while e_1 do P_1 done, $\sigma_1 \Downarrow \sigma'_1$ and while e_2 do P_2 done, $\sigma_2 \Downarrow \sigma'_2$. Then, it is easy to show that while e_1 do P_1 done, $\sigma \Downarrow \sigma'_1 \uplus \sigma_2$ and while e_2 do P_2 done, $\sigma'_1 \uplus \sigma_2 \Downarrow \sigma'$. By E_SEQ, we have $P, \sigma \Downarrow \sigma'$. By assumption $\sigma' \in \psi$.

($\Leftarrow$): Suppose $\{\varphi\}P'\{\psi\}$ and $\sigma \in \varphi$ and $P, \sigma \Downarrow \sigma'$. We will show that $\sigma' \in \psi$. By E_SEQ, we have while e_1 do P_1 done, $\sigma \Downarrow \sigma''$ and while e_2 do P_2 done, $\sigma'' \Downarrow \sigma'$ for some σ'' . It is easy to show that there exist $\sigma_1, \sigma'_1, \sigma''_1 \in \text{State}_{V_1}$ and $\sigma_2, \sigma'_2, \sigma''_2 \in \text{State}_{V_2}$ such that $\sigma = \sigma_1 \uplus \sigma_2$ and $\sigma' = \sigma'_1 \uplus \sigma'_2$ and $\sigma'' = \sigma''_1 \uplus \sigma''_2$. By Lemma 3.10, we have while e_1 do P_1 done, $\sigma_1 \Downarrow \sigma''_1$ and while e_2 do P_2 done, $\sigma''_2 \Downarrow \sigma'_2$ and $\sigma_2 = \sigma'_2$ and $\sigma''_1 = \sigma'_1$. By Lemma 3.11, $P', \sigma_1 \uplus \sigma''_2 \Downarrow \sigma'_1 \uplus \sigma'_2$. By assumption, $\sigma'_1 \uplus \sigma'_2 \in \psi$. Then, $\sigma' = \sigma'_1 \uplus \sigma'_2 = \sigma''_1 \uplus \sigma'_2$ finishes the case. $\square$

3.2 Loop Fusion for Remote Two Loops

Unfortunately, the transformation $\text{FUSION}[P]$ cannot be applied directly to examples in Section 1 because a for-loop has to execute initialization commands before entering a loop. So, two consecutive for-loops are expressed by the following form

$$\begin{array}{l} Q_1; \text{while } e_1 \text{ do } P_1 \text{ done;} \\ Q_2; \text{while } e_2 \text{ do } P_2 \text{ done} \end{array}$$

and it may be the case that Q_2 cannot be swapped with the first loop—and it is indeed the case for examples in Section 1 sharing the same index variable between the two loops.

In fact, our transformation can be easily adapted to two remote loops:

Definition 3.13 (Loop fusion for remote two loops). Let the following command be $P \in \text{Com}_V$. P consists of two remote while loops with another command sequence between them.

$$P = \text{while } e_1 \text{ do } P_1 \text{ done}; P_3; \text{while } e_2 \text{ do } P_2 \text{ done}$$

For such a command P , we define a command $\text{FUSIONR}[P] \in \text{Com}_{V \cup V' \cup V''}$ as follows:

$$\begin{array}{l} \text{FUSIONR}[P] = \text{havoc } \vec{V}'; \vec{V}'' := \vec{V}'; \\ P'_3; \\ \text{while } e_1 \vee e'_2 \text{ do} \\ \quad \text{if } e_1 \text{ then } P_1 \text{ else skip;} \\ \quad \text{if } e'_2 \text{ then } P'_2 \text{ else skip;} \\ \text{done;} \\ \text{assume } \vec{V} = \vec{V}''; \\ \vec{V} := \vec{V}'. \end{array}$$

Figure 2. Proof of Theorem 3.5.

It is also easy to adapt the correctness proof in Figure 2: In the first transformation, P_3 can be renamed together with the second loop; and, in the fourth step, P'_3 can be lifted above the first loop.

3.3 Optimization to Eliminate Redundant Variables

According to FUSIONR[P] described above, the number of variables triples in one application of loop fusion. Since the performance of a verifier tends to decrease as the number of variables increases, we introduce optimizations to save the introduction of fresh variables. Later, we will experiment with three variable introduction schemes and compare them.

The first, most straightforward baseline scheme is to take *all* the variables in the scope as the variable set V . It introduces two primed versions of each variable even when the variable does not appear at all in the two loops or the code between them.

The second scheme avoids introducing primed versions and accompanying code $\text{havoc } x', x'' := x', \text{ assume } x = x'', \text{ and } x := x'$ if there is no assignment to x . It is easy to see that such optimization does not change the semantics because $\text{assume } x = x''$ in an unoptimized version succeeds only when $\text{havoc } x'$ assigns the value of x at the beginning of FUSIONR[P]. None of x, x' or x'' is changed throughout and the final assignment $x := x'$ has no effect.

The third scheme, in addition to reduction by the second scheme, avoids primed versions and accompanying code for a variable that does not appear in both the first loop and the following part, that is, P_3 ; while e_2 do P_2 done. Correctness of this scheme can be shown similarly but may deserve some more discussion. Intuitively, if x does not appear in while e_1 do P_1 done (in FUSIONR), then we do not need to guess the value of x after the first loop because it is the very value of x before the first loop and so we can omit $\text{havoc } x'$ and so on.² In other words, an unoptimized version succeeds in $\text{assume } x = x''$ only if $\text{havoc } x'$ assigns the value of x at the

beginning of FUSIONR[P]: the value of x remains the same until assume because x appears in neither e_1 nor P_1 . At the final command $x := x'$, the value of x' is that of x after the second loop in P . The other case where x does not appear in the second part is similar to the second scheme above because the primed version x' appears in neither e'_2 nor P'_2 .

4 Guess-and-Assume for Loop Reversal

So far, we have presented the guess-and-assume transformation as a strategy for generalizing the loop fusion. This section shows another application of the guess-and-assume approach: loop reversal transformation, which is a useful preprocessing step before the loop fusion for certain cases. In order to motivate the new transformation, consider the following example.

```
void sample_03() {
  int n = random_int();
  int sum1 = 0, sum2 = 0;
  int i;
  for (i = 0; i < n; i++) { sum1 += i; /*(A)*/ }
  for (i = 0; i < n; i++) { sum2 += n - i - 1; /*(B)*/ }
  assert (sum1 == sum2);
}
```

The above program contains two loops: the loop marked (A) increases the value of i from 0 up to $n - 1$ and accumulates them to variable sum1 , whereas the loop marked (B) of the same loop pattern accumulates $n - 1$ down to 0 to variable sum2 . The program asserts $\text{sum1} == \text{sum2}$ at the end. In our experiment, neither CPACHECKER nor SEAHORN fed with the above program can prove its safety within 900 seconds; the reason seems that a verifier has to discover non-linear equations such as $\text{sum1} = \frac{i(i-1)}{2}$ and $\text{sum2} = \frac{(2n-i-1)i}{2}$ as loop invariants. In this case, our loop fusion does not help, either.

One can verify this program with simpler invariant conditions by observing the correspondences between the first iteration of the loop (A) and the $(n - 1)$ -th iteration of the loop (B), the second iteration of the loop (A) and the $(n - 2)$ -th iteration of the loop (B), and so forth. More precisely, the sum of sum1 after the i -th iteration of (A) and sum2 after the

²In this case, x is not renamed in P'_2 .

$(n - i)$ -th iteration of (B) is a constant. Moreover, it is not difficult to see that `sum1` after the loop (A) stores this constant because `sum2` stores zero before the loop (B) (i.e., after the 0th iteration) and similarly for `sum2` after the loop (B), ensuring that the assertion is valid. Note that this reasoning is possible without knowing the constant (which is actually $\frac{n(n-1)}{2}$) and the relation among program variables can be described by linear equations—although we have to relate variables at different program points.

One way to give a hint to a verifier to discover such correspondence is to *reverse* the execution of one of the loops. For example, the following program is a “reverse” of the loop (B):

```
for (i = n - 1; i >= 0; i--) { /* reversed loop */
  sum2 -= n - i - 1;
}
```

Increments `++` and `+=` are replaced with `--` and `-=`, respectively, to reverse the execution of the body. If the loop (A) and this reversed loop are fused according to the technique described in the last section, we can obtain a loop, which roughly looks like:

```
...
while (...) {
  if (i < n) { sum1 += i; i++; }
  if (j >= 0) { sum2 -= n - j - 1; j--; }
}
```

(`j` is introduced by renaming `i`) and a verifier will find a simple loop invariant that the sum of `i` and `j` is $n - 1$ and that of `sum1` and `sum2` is a constant because at each iteration `sum1` and `sum2` increase by `i` and $-(n - j - 1)$, respectively. This loop invariant corresponds to the fact that the sum of `sum1` after the i -th iteration of (A) and `sum2` after the $(n - i)$ -th iteration of (B) is a constant.

To verify the assertion at the end, we need one more twist: we have to make sure that the reversed loop starts with the state in which `sum2` stores the value of `sum2` after the loop (B). Here's where our guess-and-assume comes in!

In fact, our algorithm transforms a given loop to a program that guesses the final values of `sum2` and `i` and repeats the reversed loop body until the state matches the state before the loop. For example, the loop (B) is transformed to something like

```
havoc sum2; havoc i; havoc x;
assume (i >= n);
/* backup of guessed values */
int tmpsum2 = sum2, tmpi = i;
for (; x > 0; x--) {
  /* repeat the reversed loop body until ... */
  sum2 -= n - i - 1;
  i--;
  assume(i < n);
}
/* ... the state matches the initial state */
assume (i == 0 && sum2 == 0);
sum2 = tmpsum2; i = tmpi;
```

This loop, which runs the body an arbitrary number of times, terminates only if the guessed values for `sum2` and `i` are the

values after the loop (B) and the one for `x` is the number of iterations. (The guess for `i` must *not* satisfy the loop condition, hence assume $(i \geq n)$;) Since `sum2` and `i` are recovered at the end, this program will actually end up with the state after the loop (B) (if terminates) and the assertion `sum1 == sum2` will hold, too. Finally, fusing this loop with (A) also results in a loop with a simple, linear loop invariant, making verification easier.

For this preprocessing before loop fusion to work, we have to solve one last question: how to reverse the body of a given loop. Namely, given P , we would like a command $\text{REVERSE}[P]$ such that, for a given state, $\text{REVERSE}[P]$ computes the state from which the execution of P ends up with the given state. Again, we resort to our guess-and-assume scheme.

Definition 4.1 (Command reversal). For $P \in \text{Com}_V$, $\text{REVERSE}[P]$ is defined by:

$$\begin{aligned} \text{REVERSE}[P] = & \text{havoc } \vec{V}'; \vec{V}'' := \vec{V}'; \\ & P'; \\ & \text{assume } \vec{V} = \vec{V}'; \\ & \vec{V} := \vec{V}'' . \end{aligned}$$

$\text{REVERSE}[P]$ first guesses a state before P , stores it into $\vec{V}'$ (and $\vec{V}''$ as backup), runs P' , and checks if the resulting state is the same as the initial state (stored in $\vec{V}$). If the check succeeds, the initial guess is copied to $\vec{V}$.

Then, the entire preprocessing process can be formalized as $\text{REVERSE}[\text{REVERSELOOP}[\text{while } e \text{ do } P \text{ done}]]$, where $\text{while } e \text{ do } P \text{ done} \in \text{Com}_V$ and $\text{REVERSELOOP}[\text{while } e \text{ do } P \text{ done}]$ is given by:

```
assume ¬e;
havoc x;
while x do
  REVERSE[P];
  assume e;
  x := x - 1
done
```

(where x is a fresh variable). The resulting command $\text{REVERSE}[\text{REVERSELOOP}[\text{while } e \text{ do } P \text{ done}]]$ executes the loop body in a “reversed” manner but, as we will show below, this command is actually V -equivalent to $\text{while } e \text{ do } P \text{ done}$.

To state correctness of the transformation, we first formally define what it means for one command to be a *reversal* of another.

Definition 4.2 (V -reversal). For disjoint sets of variables V , W_1 and W_2 and a command P_1 in $\text{Com}_{V \cup W_1}$, we say P_2 in $\text{Com}_{V \cup W_2}$ is V -reversal of P_1 if the following condition holds: For any states $\sigma_1, \sigma_2 \in \text{State}_V$, $\sigma_3 \in \text{State}_{W_1}$, $\sigma_5 \in \text{State}_{W_2}$,

1. for any $\sigma_4 \in \text{State}_{W_1}$ such that $P_1, \sigma_1 \uplus \sigma_3 \Downarrow \sigma_2 \uplus \sigma_4$, there exists $\sigma_6 \in \text{State}_{W_2}$ such that $P_2, \sigma_2 \uplus \sigma_5 \Downarrow \sigma_1 \uplus \sigma_6$.

2. for any $\sigma_6 \in \text{State}_{W_2}$ such that $P_2, \sigma_2 \uplus \sigma_5 \Downarrow \sigma_1 \uplus \sigma_6$, there exists $\sigma_4 \in \text{State}_{W_1}$ such that $P_1, \sigma_1 \uplus \sigma_3 \Downarrow \sigma_2 \uplus \sigma_4$.

Lemma 4.3 (Correctness of REVERSE). *For any $P \in \text{Com}_V$, $\text{REVERSE}[P]$ is V -reversal of P .*

Proof. In what follows, for $\sigma \in \text{State}_V$, we write $\sigma' \in \text{State}_{V'}$ for the state defined by: $\sigma'(x') = \sigma(x)$.

(1): Suppose $P, \sigma_1 \Downarrow \sigma_3$. Then, it is easy to show that, for any $\sigma_{12} \in \text{State}_{V'}$ and $\sigma_{13} \in \text{State}_{V''}$, it holds that $\text{REVERSE}[P], \sigma_2 \uplus \sigma_{12} \uplus \sigma_{13} \Downarrow \sigma_1 \uplus \sigma'_{12} \uplus \sigma'_{13}$.

(2): Suppose $\text{REVERSE}[P], \sigma_1 \Downarrow \sigma_3$. Then, we have

$$\begin{aligned} & \text{havoc } \vec{V}'; \vec{V}'' := \vec{V}', \sigma_{11} \uplus \sigma'_{12} \uplus \sigma'_{13} \Downarrow \sigma_{11} \uplus \sigma'_{22} \uplus \sigma'_{22} \\ & P', \sigma_{11} \uplus \sigma'_{22} \uplus \sigma'_{22} \Downarrow \sigma_{11} \uplus \sigma'_{11} \uplus \sigma'_{22} \\ & P', \sigma'_{22} \Downarrow \sigma'_{11} \\ & \text{assume } \vec{V} = \vec{V}'; \vec{V} := \vec{V}'', \sigma_{11} \uplus \sigma'_{11} \uplus \sigma'_{22} \Downarrow \sigma_{22} \uplus \sigma'_{11} \uplus \sigma'_{22} \\ & \sigma_1 = \sigma_{11} \uplus \sigma'_{12} \uplus \sigma'_{13} \\ & \sigma_3 = \sigma_{22} \uplus \sigma'_{11} \uplus \sigma'_{22} \end{aligned}$$

for some $\sigma_{11}, \sigma_{12}, \sigma_{13}, \sigma_{22} \in \text{State}_V$. We have $P, \sigma_{22} \Downarrow \sigma_{11}$. $\square$

Lemma 4.4 (Correctness of REVERSELOOP). *For any $e \in \text{Exp}_V$, $P \in \text{Com}_V$, $\text{REVERSELOOP}[\text{while } e \text{ do } P \text{ done}]$ is V -reversal of $\text{while } e \text{ do } P \text{ done}$.*

Proof. (1): Suppose $\text{while } e \text{ do } P \text{ done}, \sigma_1 \Downarrow \sigma_2$. Then, by induction on the number of iterations, there exist states $\sigma_0 \dots \sigma_n$ such that

$$\begin{aligned} & P, \sigma_i \Downarrow \sigma_{i+1} \quad (i \in \{0, \dots, n-1\}) \\ & \sigma_i \llbracket e \rrbracket \neq 0 \quad (i \in \{0, \dots, n-1\}) \\ & \sigma_n \llbracket e \rrbracket = 0 \end{aligned}$$

For any $m \in \mathbb{Z}$, it holds that $\text{assume } \neg e; \text{havoc } x, \sigma_n \uplus \{x \mapsto m\} \Downarrow \sigma_n \uplus \{x \mapsto n\}$. By Lemma 4.3, for any $i \in 0, \dots, n-1$, $\text{REVERSE}[P], \sigma_{i+1} \uplus \{x \mapsto i+1\} \Downarrow \sigma_i \uplus \{x \mapsto i+1\}$. Moreover, $\text{assume } e; x := x - 1, \sigma_i \uplus \{x \mapsto i+1\} \Downarrow \sigma_i \uplus \{x \mapsto i\}$. By induction on n , we have $\text{while } e \text{ do } \text{REVERSE}[P]; \text{assume } e; x := x - 1 \text{ done}, \sigma_n \uplus \{x \mapsto n\} \Downarrow \sigma_0 \uplus \{x \mapsto 0\}$ and $\text{REVERSELOOP}[P], \sigma_n \uplus \{x \mapsto m\} \Downarrow \sigma_0 \uplus \{x \mapsto 0\}$.

(2) is similar. $\square$

Theorem 4.5. *For $e \in \text{Exp}_V$ and $P \in \text{Com}_V$, $\text{REVERSE}[\text{REVERSELOOP}[\text{while } e \text{ do } P \text{ done}]]$ depends only on V and is V -equivalent to $\text{while } e \text{ do } P \text{ done}$.*

Proof. Let P' be $\text{REVERSE}[\text{REVERSELOOP}[\text{while } e \text{ do } P \text{ done}]]$. It is easy to show P' depends only on V because other variables are assigned random integers before being used.

($\Rightarrow$): Suppose $\{\varphi\}P\{\psi\}$ and $\sigma_1 \in \varphi$ and $P', \sigma_1 \uplus \sigma_3 \Downarrow \sigma_2 \uplus \sigma_4$. We will show that $\sigma_2 \in \psi$. By Lemmas 4.3 and 4.4, we have $P, \sigma_1 \Downarrow \sigma_2$. By assumption, $\sigma_2 \in \psi$.

($\Leftarrow$): Suppose $\{\varphi * \text{State}_W\}P'\{\psi * \text{State}_W\}$ (where W is the set of variables introduced by transformation) and $\sigma_1 \in \varphi$ and $P, \sigma_1 \Downarrow \sigma_2$. We will show that $\sigma_2 \in \psi$. By Lemmas 4.3 and

4.4, for any $\sigma_3 \in \text{State}_W$, there exists σ_4 such that $P', \sigma_1 \uplus \sigma_3 \Downarrow \sigma_2 \uplus \sigma_4$. By assumption, $\sigma_2 \in \psi$. $\square$

5 Experiments

5.1 Implementation

We implemented a source-to-source translator that applies the transformations described in Sections 3 and 4 to a C program. It is written in OCaml and compiled with ocamlc 4.04.1; we used CIL 1.7.3 [23] as a C front-end.

The translator can be invoked in two modes: **fusion** mode and **rev_fusion** mode. If the translator is invoked in the **fusion** mode, then it scans the abstract syntax tree of the input code to detect a part where the transformation FUSIONR is applicable and applies the transformation. If it is invoked in the **rev_fusion** mode, then the translator applies REVERSELOOP and REVERSE to the second loop before applying FUSIONR. If there are several candidates for transformation in the input code, then the translator chooses the pair of loops that appears first in the innermost block. The transformation is repeatedly applied until the program contains no candidate.

The translator also applies the optimizations to eliminate redundant variables presented in Section 3.3. The translator provides three levels of optimizations corresponding to the three schemes in Section 3.3: Level 0 that takes all the variables appearing in the program as V ; Level 1 that excludes from V the primed version of the variables that are not changed throughout the program; and Level 2 that further excludes the variables that are not used in both of the first loop and the rest.

In addition to these optimizations, the translator tries to move down assignments so that they are postponed as much as possible. We found this postprocessing effective in a few benchmarks.

5.2 Experiment Environment

In order to measure the effectiveness of our transformation, we integrated our translator with two static verifiers: CPACHECKER [4] (version 1.6.12) and SEAHORN [14] (version 0.1.0). For each verifier, we measured the time to complete the following computation:

- **Vanilla:** Time to verify the given program using the verifier (without translation) and
- **Translated:** Time to translate the given program and to verify the translated program.

For the experiment **Translated**, we conducted the experiments under all the combinations of the following parameters:

- Mode of the fusion: **fusion** or **rev_fusion**;
- Redundant variable elimination: Level 0, 1, or 2; and
- Postponing assignments described above: **enabled** or **disabled**.

All the experiments are conducted on a server with two Intel Xeon processors E5-2670 (each of which is equipped with eight cores, 2.6 GHz) and 128 GB memory.

Benchmarks. We used the following programs as input to each experiment.

loop01 is the following program.

```
int main() {
  int n = random_int();
  int sum = 0;
  for (int i = 0; i < n; ++i) { sum += i; }
  for (int i = 0; i < n; ++i) { sum -= i; }
  assert (sum == 0);
  return 0;
}
```

This program contains two consecutive loops: The former accumulates $0, 1, \dots, n-1$ to the variable `sum`; the latter subtracts $0, 1, \dots, n-1$ from `sum`. The verifier needs to prove that `sum` is equal to 0 at the end.

loop02 is the following program.

```
int main() {
  int max_val = random_int();
  int val = random_int();
  int val_ = val;
  int n = 0;
  while (val < max_val) { val += 1; n += 1; }
  for (int i = 0; i < n; ++i) { val -= 1; }
  assert (val == val_);
  return 0;
}
```

This program contains two consecutive loops. The first loop increments `val` up to `max_val` and counts how many times the loop body has been executed with `n`. The second loop decrements `val` the same number of times as the first loop. The verifier has to prove that `val` is equal to `val_`, which remembers the value of `val` before the loops.

loop03 is the following program

```
int main() {
  int n = random_int();
  int sum = 0;
  assume (n >= 0);
  for (int i = 0; i < n; i += 2) { sum += 1; }
  for (int i = 1; i < n; i += 2) { sum -= 1; }
  assert (sum == n % 2);
  return 0;
}
```

This program contains two consecutive loops. The first loop increments `sum` in each iteration whereas the second decrements. The value of `sum` at the end is equal to 0 if the value of `n` is even because both loops iterate the same number of times in this case. If `n` is odd, the value of `sum` is 1 because the first loop iterates one more time than the second loop. The assertion at the end expresses this property using the remainder operator.

loop03_false is a program where an assertion failure is reachable. This program is the same as **loop03** except

that it does not have the statement `assume (n >= 0);` before the loop. So, the value of `n` may be negative in this program. If this is the case, the value of `sum` at the end is 0 because neither of the loop is executed. Therefore, if `n` is odd and negative, the assertion `assert(sum == n % 2)` fails.

loop04 is the following program.

```
int main() {
  int n = random_int();
  int sum = 0;
  assume (n >= 0);
  for (int i = 0; i < n; i += 2) { sum += i * 2; }
  for (int i = 1; i < n; i += 2) { sum -= i * 2; }
  assert (!(n % 2 == 0) || sum == -n);
  return 0;
}
```

This program exhibits the same iteration pattern as that of **loop03**. In this program, however, `sum` is incremented/decremented by the value of $2 \cdot i$, instead of 1, in each iteration. The value of `sum` is equal to $-n$ if `n` is even. This condition is expressed in the assertion at the end of the program.

loop05 is the following program.

```
int main() {
  int n = random_int();
  int sum = 0;
  for (int i = 0; i < n; ++i) { sum += i; }
  for (int i = 0; i < n; ++i) { sum -= (n - i - 1); }
  assert (sum == 0);
  return 0;
}
```

The first loop of this program adds $0, 1, \dots, n-1$ to `sum` in this order; the second subtracts $n-1, n-2, \dots, 0$ in this order. The value of `sum` is equal to 0 at the end for any `n`. The **rev_fusion** strategy should be effective for this program since the iteration patterns of the two loops are opposite.

5.3 Results

Tables 1 and 2 show the result in which we used CPACHECKER and SEAHORN, respectively. Each column shows the following information:

- The column “Program” shows the name of the input program;
- The column “Result” shows whether an assertion failure is unreachable (denoted by True in the tables) or reachable (denoted by False in the tables);
- The column CPACHECKER in Table 1 and SEAHORN in Table 2 show the time in seconds to complete the **Vanilla** experiment described above;
- The column **fusion** shows the result of the **Translated** experiment in which the translator is invoked in the **fusion** mode; and
- The column **rev_fusion** shows the result of the **Translated** experiment in which the translator is invoked in the **rev_fusion** mode.

Table 1. Performance Results (CPACHECKER).

Program	Result (expected)	CPACHECKER	fusion			rev_fusion		
			Level 0	Level 1	Level 2	Level 0	Level 1	Level 2
loop01	True	—	13.15	7.91	12.05	—	—	—
			48.16	12.33	11.31	—	—	—
loop02	True	—	49.93	40.77	32.64	25.65	22.72	18.60
			42.08	32.85	41.50	30.06	23.93	23.69
loop03	True	—	47.45	11.86	13.68	63.47	439.13	43.40
			59.44	20.95	13.32	86.37	285.26	307.00
loop03_false	False	5.33	6.49	6.49	6.03	10.36	9.92	6.68
			6.42	6.05	5.79	10.67	9.85	6.24
loop04	True	—	268.77	21.37	16.68	—	—	—
			46.45	21.62	17.15	—	—	—
loop05	True	—	—	—	—	—	—	—
			—	—	—	373.38	21.56	21.80

Table 2. Performance Results (SEAHORN).

Program	Result (expected)	SEAHORN	fusion			rev_fusion		
			Level 0	Level 1	Level 2	Level 0	Level 1	Level 2
loop01	True	—	0.39	0.22	2.59	—	—	—
			0.40	0.17	2.60	—	—	—
loop02	True	—	0.22	0.20	0.19	0.36	0.33	0.28
			0.22	0.21	0.20	0.32	0.34	0.27
loop03	True	—	—	—	—	—	—	—
			—	—	—	—	—	—
loop03_false	False	0.11	0.10	0.11	0.09	0.12	0.12	0.11
			0.11	0.10	0.10	0.10	0.14	0.13
loop04	True	—	—	—	—	—	—	—
			—	—	—	—	—	—
loop05	True	—	—	—	—	—	—	0.82
			—	—	—	34.99	—	—

Each of the columns **fusion** and **rev_fusion** is further divided into three columns corresponding the optimization level of the redundant variable elimination 0, 1, and 2. Each cell in these columns contains two vertically aligned numbers. The upper is the time in which the assignment-postponing postprocessing explained above is **enabled**; the lower is the time with this postprocessing **disabled**. “—” in the table denotes a computation timeout, which is set to 900 seconds.

5.4 Discussion

CPACHECKER and SEAHORN both failed to prove safety of the programs from **loop01** to **loop05**. This suggests that a program that contains consecutive loops is one of the main difficulties even for state-of-the-art verifiers considering that these programs are small.

If we chose the mode of the fusion (i.e., **fusion** or **rev_fusion**) and the optimization strategy appropriately, then most of the translated programs (8 out of 10) were successfully verified before the timeout; the time spent for the translation was negligible in all the cases. There was no program that was successfully verified but its translated version could

not be verified. We believe that this result demonstrates the effectiveness of our method, although we need more study on how to tune these parameters. In SEAHORN, the translated code of **loop03** and **loop04** could not be verified with any combination of the parameters. At the time of writing, the reason has not been investigated.

Choosing the mode **rev_fusion** tended to be “risky” in that it translated the program to one that could not be successfully verified even if its **fusion** counterpart could be; 3 programs translated by **rev_fusion** failed among 7 cases where the program translated by **fusion** successfully verified. However, for the program **loop05**, whose second loop exhibited the opposite iteration pattern to the second loop, the strategy **rev_fusion** was effective.

Our method incurred overhead to the benchmark **loop03_false** (doubled the verification time at worst). If the program contains a real bug, our translation seems to worsen the performance. One possible workaround to this problem is to run the verification of the original program and the translated program in parallel and to adopt the result obtained earlier.

The performance of the Level-1 optimization is better than that of Level 0 (without optimization) in 20 out of 48; Level 2 is better than Level 0 in 23 out of 48. Although it is difficult to conclude that the redundant variable elimination is always effective, it seems to be safe to argue that it does not degrade the performance of Level 0 in general: there are only 3 out of 48 cases where Level 0 is better than both Level 1 and Level 2.

The assignment-postponing optimization was also risky. It sometimes improved the performance much (e.g., a few cases of **loop05** with **rev_fusion**) but performance degradation was also observed in many cases.

6 Related Work

Program transformation is often used for enhancing the precision and the efficiency of static verification. Various methods for static verification use program transformation as a core component. The realm of such analyses spans widely, including safety verification of software [1, 6, 8, 9, 17, 19, 25, 26] and hybrid systems [15]; resource-bound analysis [12, 13, 21, 29]; termination analysis [5]; and information-flow analysis [16]. Despite this diversity, we have not noticed any methods that are based on loop fusion or the guess-and-assume approach.

Among these methods proposed so far, Sharma et al. [26] is the most relevant one to ours. Their program transformation is for efficiently verifying a program with a *multi-phase loop*: a loop that contains a conditional branch in the body (i.e., a loop of the shape `while (...) if (E) s1 else s2`) and the value of E turns from false to true at a certain point of the execution of the program (i.e., the sequence of the values obtained by evaluating E in each iteration is described by regular expression `false* true*`). Verification of such a program often requires a disjunctive invariant of the form $(E \wedge \phi_1) \vee (\neg E \wedge \phi_2)$ using formulae ϕ_1 and ϕ_2 , which are known to be difficult for many automated static verification algorithms to synthesize. To address this problem, they proposed a transformation that splits the multi-phase loop to consecutive two loops with simpler invariants.

Their method advocates *splitting* a loop into consecutive ones for efficient verification whereas ours *fuses* consecutive loops. This difference in the transformation strategies originates from the characteristic of the program statements on which each method works effectively. Their method tends to be effective for multi-phase loops, whereas ours for program statements that contain two loops whose iteration patterns are similar (in the normal fusion in Section 3) or opposite (in the reversed fusion in Section 4) to each other.

Loop fusion and loop reversal have been studied in the context of code optimization, specifically as a technique for loop reordering; we refer one to the survey by Bacon et al. [2] for the detailed exposition. The purpose of the transformations studied in this field (i.e., making the generated code

more efficient) is different from ours (i.e., making the verification faster). As we discussed in Section 1, our transformation is also technically different in that the guess-and-assume approach allows our transformation to be applied to the consecutive loops even if the body of the latter loop has data dependency to that of the former and it preserves semantics only up to termination.

In Section 4, we introduced the procedures `REVERSE[P]` and `REVERSELOOP[P]`, which return programs that are V -reversals of P . In this regard, *reversible languages* are relevant; using these languages, one can write only a program P such that the function implemented by P has a reverse function. Mu et al. [22] proposed a point-free reversible functional programming language with which one can implement only injective functions and therefore reversible. The procedures `REVERSE` and `REVERSELOOP` are also applicable to the programs that are not injective. However, the code generated by these procedures is not intended for execution.

7 Conclusions and Future Work

We have presented a new loop fusion technique to enhance the efficiency of static program verification. The technique is based on what we call the “guess-and-assume” approach, which allows two remote loops with data dependence to be fused. We have formalized the algorithm and proved its correctness for a simple imperative language. We have also developed a preprocessing technique for loop fusion based on reversal of loop execution. The effectiveness of these techniques is demonstrated by verifying a few small benchmark programs, which the state-of-the-art program verifiers fail to verify.

Future Work. We currently support a simple imperative programming language that can manipulate only integers. Extending the language is an important future direction of this work. We are especially interested in extension with arrays; we believe our method is effective for array-manipulating programs since an array is often processed using consecutive loop statements.

We are also planning experiments using larger programs than the programs used in the current experiments in Section 5. (However, we emphasize that these experiments with small programs already revealed a problem in state-of-the-art program verifiers and the effectiveness of our approach in addressing the problem.) An application we are interested in is *non-interference verification using self-composition* [27]. They reduce the problem of verifying non-interference [11, 18, 20] of a program P to a certain safety-verification problem of the *self-composed program* $P; P'$ where P' is the program obtained by priming all the variables in P . If P contains a loop, then the self-composed program contains loops with similar looping patterns, for which we believe our method is effective.

Acknowledgments

This work was supported in part by the JSPS KAKENHI Grant Numbers JP15H05706 and JP15KT0012, and JST PRESTO No. JPMJPR15E5, Japan.

References

- [1] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. 2014. An extension of lazy abstraction with interpolation for programs with arrays. *Formal Methods in System Design* 45, 1 (2014), 63–109. <https://doi.org/10.1007/s10703-014-0209-9>
- [2] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler Transformations for High-performance Computing. *ACM Comput. Surv.* 26, 4 (Dec. 1994), 345–420. <https://doi.org/10.1145/197405.197406>
- [3] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMC0 2005 (LNCS)*, Vol. 4111. 364–387. https://doi.org/10.1007/11804192_17
- [4] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *CAV 2011 (LNCS)*, Vol. 6806. Springer, 184–190. https://doi.org/10.1007/978-3-642-22110-1_16
- [5] Cristina Borralleras, Marc Brockschmidt, Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2017. Proving Termination Through Conditional Termination. In *TACAS 2017 (LNCS)*, Vol. 10205. 99–117. https://doi.org/10.1007/978-3-662-54577-5_6
- [6] Marc Brockschmidt, Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2015. Compositional Safety Verification with Max-SMT. In *Proc. of ACM FMCAD'15*. 33–40.
- [7] Ellis Cohen. 1977. Information transmission in computational systems. In *ACM SIGOPS Operating Systems Review*, Vol. 11. ACM, 133–139.
- [8] Aditya Desai, Era Jain, and Subhajit Roy. 2013. Facilitating Verification in Program Loops by Identification of Static Iteration Patterns. In *APSEC 2013*. 83–90. <https://doi.org/10.1109/APSEC.2013.22>
- [9] Kostas Ferles, Valentin Wüstholtz, Maria Christakis, and Isil Dillig. 2017. Failure-directed program trimming. In *ESEC/FSE 2017*. 174–185. <https://doi.org/10.1145/3106237.3106249>
- [10] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. <https://doi.org/10.1145/24039.24041>
- [11] Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. 11–20. <https://doi.org/10.1109/SP.1982.10014>
- [12] Sumit Gulwani, Sagar Jain, and Eric Koskinen. 2009. Control-flow refinement and progress invariants for bound analysis. In *PLDI 2009*. 375–385. <https://doi.org/10.1145/1542476.1542518>
- [13] Sumit Gulwani and Florian Zuleger. 2010. The reachability-bound problem. In *PLDI 2010*. 292–304. <https://doi.org/10.1145/1806596.1806630>
- [14] Arie Gurfinkel, Temesghen Kahsay, Anvesh Komuravelli, and Jorge A Navas. 2015. The SeaHorn verification framework. In *CAV 2015 (LNCS)*, Vol. 9206. Springer, 343–361. https://doi.org/10.1007/978-3-319-21690-4_20
- [15] Ichiro Hasuo and Kohei Suenaga. 2012. Exercises in Nonstandard Static Analysis of Hybrid Systems. In *CAV 2012 (LNCS)*, Vol. 7358. 462–478. https://doi.org/10.1007/978-3-642-31424-7_34
- [16] Boris Köpf and Andrey Rybalchenko. 2010. Approximation and Randomization for Quantitative Information-Flow Analysis. In *CSF 2010*. 3–14. <https://doi.org/10.1109/CSF.2010.8>
- [17] Matt Lewis. 2014. *Precise Verification of C Programs*. Ph.D. Dissertation. University of Oxford.
- [18] Peng Li and Steve Zdancewic. 2005. Downgrading policies and relaxed noninterference. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. 158–170. <https://doi.org/10.1145/1040305.1040319>
- [19] Vivien Maisonnewe. 2012. Convex Invariant Refinement by Control Node Splitting: a Heuristic Approach. *Electr. Notes Theor. Comput. Sci.* 288 (2012), 49–59. <https://doi.org/10.1016/j.entcs.2012.10.007>
- [20] John McLean. 1990. Security Models and Information Flow. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 7-9, 1990*. 180–189. <https://doi.org/10.1109/RISP.1990.63849>
- [21] Daniel Le Métayer. 1988. ACE: An Automatic Complexity Evaluator. *ACM Trans. Program. Lang. Syst.* 10, 2 (1988), 248–266. <https://doi.org/10.1145/42190.42347>
- [22] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. 2004. An Injective Language for Reversible Computation. In *MPC 2004 (LNCS)*, Vol. 3125. 289–313. https://doi.org/10.1007/978-3-540-27764-4_16
- [23] George Necula, Scott McPeak, Shree Rahul, and Westley Weimer. 2002. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction (CC 2002) (LNCS)*, Vol. 2304. Springer, 209–265. https://doi.org/10.1007/3-540-45937-5_16
- [24] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of IEEE Symposium on Logic In Computer Science (LICS'02)*.
- [25] Herbert Rocha, Hussama Ismail, Lucas C. Cordeiro, and Raimundo S. Barreto. 2015. Model Checking Embedded C Software Using k-Induction and Invariants. In *SBESC 2015*. 90–95. <https://doi.org/10.1109/SBESC.2015.24>
- [26] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Simplifying Loop Invariant Generation Using Splitter Predicates. In *CAV 2011 (LNCS)*, Vol. 6806. 703–719. https://doi.org/10.1007/978-3-642-22110-1_57
- [27] Tachio Terauchi and Alexander Aiken. 2005. Secure Information Flow as a Safety Problem. In *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*. 352–367. https://doi.org/10.1007/11547662_24
- [28] Glynn Winskel. 1993. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press.
- [29] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. 2011. Bound Analysis of Imperative Programs with the Size-Change Abstraction. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011, Proceedings*. 280–297. https://doi.org/10.1007/978-3-642-23702-7_22