Poster: Automatic Detection of Inverse Operations while Avoiding Loop Unrolling

Kunal Banerjee
Intel Labs
Bangalore, India
kunal.banerjee@intel.com

R. Chouksey, Chandan Karfa, Pankaj Kalita
IIT Guwahati
Guwahati, India
{r.chouksey,ckarfa,pankajkalita}@iitg.ernet.in

ABSTRACT

A pair of *inverse operations* is defined as two operations that when performed on a number or variable always results in the original number or variable. Novice programmers may introduce such inverse operations; automated parallelizing tools also employ such operations to undo the effects of some speculatively executed operation. Therefore, detection of inverse operations is helpful in both compiler optimization (redundant code elimination) as well as verification of parallelizing frameworks. In this work, we extend the definition of inverse operations to include a set of operations instead of only two and present a method for detecting inverse operations symbolically which would otherwise need complete unrolling of loops. Some interesting intricacies of detecting inverse operations are also discussed.

CCS CONCEPTS

• **Software and its engineering → Consistency**; **Automated static analysis**;

KEYWORDS

inverse operation, compiler optimization, Integer Set Library

ACM Reference Format:
Kunal Banerjee and R. Chouksey, Chandan Karfa, Pankaj Kalita. 2018. Poster: Automatic Detection of Inverse Operations while Avoiding Loop Unrolling. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, Article 4, 2 pages. https://doi.org/10.1145/3183440.3195083

1 INTRODUCTION

A pair of *inverse operations* may be thought of as two operations which "undo" the effect of each other. Such operations may be erroneously inserted by some novice programmer during coding or some automated tool, typically parallelizing frameworks [4], dynamically to reverse the effect of some speculatively executed operation because employing inverse operations often incur less overhead than restoring a state by rolling back to it [3]. Thus, detection of inverse operations may be beneficial for both compiler optimization and verification. Inverse operations have also been used for testing C++ libraries [6].

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05…$15.00
https://doi.org/10.1145/3183440.3195083

```
S1: x = t1;
S2: y = t2;
for (i = 1; i <= 99; i++) {
  if (i%3 == 0) { //C1
    S3: x = x + 1;
    S4: y = y + 1;
  } else if (i%3 == 1) { //C2
    S5: x = x - 2;
    S6: y = y * 2;
  } else //C3
    S7: x = x + 1;
}
```

Figure 1: An example of inverse operations.

Literature on inverse operations [3, 5] typically consider *pairs* of inverse operations, e.g., addition and subtraction by the same number or variable in a basic block, although, in theory, there may be multiple operations whose cumulative effect on a variable may be null. Moreover, detection of inverse operations basically involves symbolic or concolic execution of a program trace and therefore, detecting such inverse operations in the presence of loops is a challenging task without employing loop unrolling.

Let us consider the program given in Figure 1. It is easy to check that each of the three conditions `C1`, `C2` and `C3` in this program is satisfied exactly 33 (one-third of 99) times and thus the value of `x` (which is increased by 1 when conditions `C1` and `C3` are satisfied and decreased by 2 when condition `C2` is satisfied) remains the same before and after the `for` loop. Therefore, an optimized version of this program may eliminate the last `else` branch along with statements `S3`, `S5` and `S7` (redundant code elimination) and move the definition of variable `x` (statement `S1`) after the loop (loop invariant code motion) to reduce register pressure.

A pertinent point to note is that current loop invariant code detection algorithm cannot detect the fact that the variable `x` is invariant in the `for` loop of Figure 1 because the reaching definitions of `x` in each conditional branch have a cyclic dependency on one another [1]. We tried to check the equivalence between the program in Figure 1 and its optimized version as mentioned above using the equivalence checker provided in [7] which can run with different backend solvers; the verifier could report "equivalence" only when the the upper bound for the variable `i` is set to a low value, say 9, for higher values the tool failed to decide whether the programs are equivalent or non-equivalent or terminated abruptly with an error. A manual inspection of the proof generated when the tool succeeded in proving equivalence showed that the loop was completely unrolled to reach the decision. In this work, we address the problem of detecting inverse operations which may span across

multiple control branches inside a loop without unrolling the loop (in most cases). Furthermore, our method can be applied to loops where the lower and/or upper bounds are parametric constants.

2 INVERSE OPERATION DETECTION MECHANISM

Keeping the example shown in Figure 1 in mind, let us first define the notion of inverse operations.

Definition 2.1 (Set of inverse operations:). Given a set of k operations $o_1, o_2, \ldots, o_k$ and a corresponding ratio (in positive integers) $r_1 : r_2 : \ldots : r_k$, we say that the set represents a *set of inverse operations* iff when the variable(s) that these operations modify (lhs variable) remain unmodified when operation o_1 is executed r_1 times, o_2 is executed r_2 times, and so on, in any order.

It is trivial to see that typical pairs of inverse operations basically have two operations in the set of inverse operations and the ratio is 1:1. For the example in Figure 1, the set of inverse operations consists of the statements S3, S5, S7 and the ratio is 1:1:1. While the candidate operations for a set of inverse operations can be determined using the cyclic dependency of reaching definitions, the corresponding ratio can be obtained from the cardinality of the iteration domain of the iterator (variable `i` in Figure 1) as defined by the conditions of the (nested) loop(s) and the control branch(es) enclosing each statement inside a loop (e.g, $S1 := \{[i] : 1 <= i <= 99$ and $i\%3 = 0\}$ for statement S1). We use the Integer Set Library (ISL) [8] to find the iteration domains and their cardinalities; this tool is effective in deducing the domains even in case of nested loops and branches along with parametric constants [2]. The user of our tool may also supply a lookup table of uninterpreted functions and their inverses (or a set of such functions with accompanying ratios in accordance with Definition 2.1), such as those provided for linked data structures in [3]. It may be worth noting that a complete loop unrolling may be required if the ratios of the inverse operations are co-prime to one another and their sum is equal to the cardinality of the iteration domain.

Next we provide some intricacies of inverse operation detection method.

Overlapping iteration domains: Let us consider the case where the `for` loop iterates over $i = 1, 2, \ldots, 100$ and the condition of a former `if` branch is `(i % 4 == 0)` while that of a latter `else if` branch is `(i % 2 == 0)`. In this case, it may seem that the cardinality of the statement under the former control branch is 25 while that of the latter is 50. However, the control goes to the following `else if` branch only when it fails for the earlier one; therefore, we need to take the set difference of the iteration domains of the following branches with respect to the earlier ones. If the conditions were to be reversed, i.e., `(i % 2 == 0)` appears earlier than `(i % 4 == 0)`, then the control will never go into the latter branch which would have been revealed by the set difference returning *null* – thus, this step may also help in removing dead code.

Datatype dependent operations: Let us consider the case of integer multiplication and division. Suppose we start with `x = 1` and then `x = x*2` and `x = x/2` is performed an equal number of times alternately in different control branches; if we perform multiplication first, then the result is 1, whereas if division is performed first, then the result is `0`. We have incorporated special rules (e.g., whether the denominator perfectly divides the numerator for `int` datatypes) to handle such cases.

Overflow/underflow: Again consider the above example with the exception that `x` is assigned some value greater than `INT_MAX/2`, where `INT_MAX` represents the maximum representable integer value. In this case, division followed by multiplication never results in overflow, whereas performing multiplication first does. It is to be noted that detecting and getting rid of the inverse operations results in a code where overflow/underflow never occurs (although in the original code it may). Therefore, we do not have any special rules to detect overflow/underflow scenarios. Our method is described succinctly in Algorithm 1.

Algorithm 1 detectInverseOperations(CDFG of loop L)

1: Find a set of operations, S say, which have a cyclic dependency of reaching definitions.
2: Find the ratio in which the operations are executed.
3: If the combined effect of executing the operations as per Definition 2.1 is $null$ (while taking care of the intricacies mentioned in Section 2), then output S.

3 EXPERIMENTAL RESULTS

Table 1: Results of Applying Algorithm 1 to Some Benchmarks

Benchmarks	#nest	#branch	#const	#op	Time
Figure 1	1	3	99	3	4
Test0	1	3	100	3	4
Test1	2	2	1024	4	8
Test2	3	2	3000	6	10

The results on some benchmarks are given in Table 1. Specifically, the table reports what is the maximum depth of nesting of the loops involved, the number of control branches, the number of parametric constants involved, the number of operations in the set of inverse operations and the time in milli seconds. We intend to extend our framework to deploy bigger benchmarks in the near future.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Princiles, Techniques, and Tools.* Pearson Education.
[2] Chandan Karfa, Kunal Banerjee, Dipankar Sarkar, and Chittaranjan Mandal. 2013. Verification of Loop and Arithmetic Transformations of Array-Intensive Behaviours. *IEEE Trans. on CAD of ICS* 32, 11 (2013), 1787–1800.
[3] Deokhwan Kim and Martin C. Rinard. 2011. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *PLDI*. 528–541.
[4] Eric Koskinen and Matthew J. Parkinson. 2015. The Push/Pull model of transactions. In *PLDI*. 186–195.
[5] Trek S. Palmer. 2014. *Using Formal Methods to Verify Transactional Abstract Concurrency Control.* Ph.D. Dissertation. University of Massachusetts - Amherst.
[6] R. N. Shakirov. 2010. The use of error detection and compensation techniques for testing the integer class cBigNumber. *Programming and Computer Software* 36, 1 (2010), 36–47.
[7] Mattias Ulbrich, Vladimir Klebanov, and Moritz Kiefer. [n. d.]. Automatically check programs for equivalence - LLREVE. https://formal.iti.kit.edu/projects/improve/reve/. ([n. d.]).
[8] Sven Verdoolaege. 2010. ISL: An Integer Set Library for the Polyhedral Model. In *ICMS*. 299–302.