Partial Redundancy Elimination via Lazy Code Motion

Tarun Rajendran and Yiyi Wang

1 Problem Statement & Motivation

Partial Redundancy Elimination is a well used transformation to optimize code. A very powerful optimization technique, Partial Redundancy Elimination (PRE) seeks to identify partially redundant expressions and makes them fully redundant, creating more opportunities for optimizations such as Common Subexpression Elimination (CSE) and Loop Invariant Code Motion (LICM).

The requirements for identifying partially redundant expressions is very strict, and can limit the opportunities for PRE especially due to SSA form. Briggs and Cooper [2] explains that more opportunities can be created to identify partial redundancy by first transforming the code with Global Reassociation and Global Value Numbering (GVN).

Another traditional drawback to PRE is the increased amount of memory bandwidth, and register pressure. Adopting Lazy Code Motion techniques, from Knoop et al. [4], allows us to alleviate these issues and allows us to have a powerful algorithm while still maintaining performance. We believe that combining the ideas in these papers may lead to significant improvement to this fundamental optimization algorithm and may yield interesting results to guide future work in this area.

2 Existing Work

Effective Partial Redundancy Elimination [2]

This paper showcases the value of performing Global Reassociation and Global Value Numbering before performing Partial Redundancy Elimination. It shows off the increased performance effects of PRE and illustrates the new opportunities that performing these passes create for PRE. It also introduces an new implementation of Global Reassociation to create new opportunities for PRE.

Lazy Code Motion [4]

This paper contributes an unidirectional iterative data flow algorithm for performing PRE. We base our PRE implementation off of this algorithm. This algorithm allows us to efficiently and lazily perform PRE, which gives us the benefit of reduced code motion and register pressure.

A New Algorithm for Partial Redundancy Elimination based on SSA Form [1]

In contrast to the traditional bitvector algorithm for performing PRE, Chow et al. [1] introduces a new SSA PRE transformation pass which seeks to increase opportunities of PRE and while avoiding iterative data flow analysis. It also adopts similar and independently implemented lazy code motions techniques from Knoop et al [4]. Notably this implementation avoids a separate phase for data flow analysis to gather information about the expressions in the program, something that our implementation is forced to perform.

GVN in LLVM

The GVN algorithm in LLVM is closely coupled with its own PRE transformation and Dead Load Elimination passes, and performs PRE in its transformation pass. This algorithm also seems to be an iterative bit vector algorithm which transforms the code after performing GVN. The Dead Load Elimination pass helps clean up the increased code size that PRE creates.

3 Algorithm Overview

Before our algorithm runs, we perform Global Reassociation and Global Value Numbering (SSA) before Lazy Code Motion [4] to help increase the opportunities for eliminating partially redundant expressions in the code. Next we perform an algorithm very similar to lazy code motion to eliminate partially redundant expressions.

To begin our algorithm, we need to identify the set of terms to perform our transformation on. We look through all Binary Operations and create a term WorkList. We define a term to be of two operands and one binary operator. These operands in the term must be a local variable, global variables, constant, or function argument. Then we analyze all of the terms in the WorkList one by one.

For each term, we perform iterative dataflow analysis to calculate the `D-Safe`, `Earliest`, `Delay`, `Latest`, and `Isolated` sets to be able to determine the `Optimal Conditional Points (OCP)` and `Redundant Occurrences (RO)` sets. In order to be comprehensive, we must perform multiple iterations to guarantee correctness. This is to ensure the analysis is correct even when there is a backedge in the code. Therefore we iterate until the appropriate set no longer changes, which should be N-1 times, N being the number of back edges in the code.

Once we have calculated the `OCP` and `RO` sets, we can then perform the `OCP-RO Transformation` detailed by Knoop et al. [4]. We first introduce a new auxiliary variable to replace the computation of the term. We insert the assigned aux variable at the Optimal Conditional Point to store the result of computation of the term. Performing the computation and storing it into a auxiliary variable, is how we maintain laziness. Finally, we replace each Redundant Occurrence with the auxiliary variable.

We then perform this algorithm on the rest of the WorkList until it is empty.

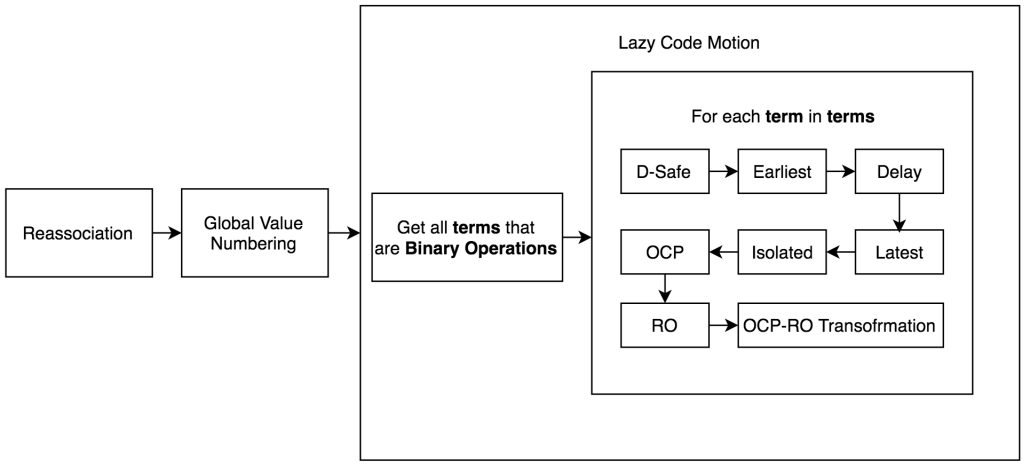


Fig. 1: The general flow of the algorithm

4 Implementation Details

The LLVM `reassociation` pass works properly, so we just used it directly.

We tried to use the existing `gvn` pass from LLVM, but that `gvn` pass already performs `pre` and a lot of other eliminations such as dead load elimination, therefore we decided not to use the `gvn` pass. We discovered a new GVN transformation in LLVM 4.0 called `new-gvn`. The `new-gvn` pass uses a sparse algorithm based on the ideas of Karthik Gargi [3]. Most importantly it does not have its own PRE algorithm.

After finish performing the `reassociation` pass and `new-gvn` pass, we continue to retrieve a set of `terms`. A `term` stores two operands and one operator. The `terms` can be easily obtained by checking if an instruction is of `BinaryOperator` type. The operand in a term must be a local variable, global variables, constant, or function argument. Since we only consider Scalar expressions, we do not consider `gep` instructions loaded from a struct or array.

Once we get the `terms` set. We start analyzing them one by one. We define four helper functions `Used`, `Transp`, `Succ` and `Pred`:

- `Used(inst, term)` is true if the `inst` and `term` have the same operands and operator.
- `Transparent(inst, term)` is false if that `inst` modifies the operand of `term`. Also if `inst` is a call instruction and one operand of the `term` is passed in as argument, then the transparency is also set to false. We did this because we are not able to judge whether the operand passed as parameter will be changed or not in the calling function. For simplicity, we just set the transparency of that case as false.
- `Succ(inst)` returns the successor of `inst`. If `inst` is not the last instruction in its basic block, then return the next instruction of `inst`. Otherwise, return the first instructions of basic blocks that are the successor of the basic block where `inst` is.
- `Pred(inst)` returns the predecessor of `inst`. If `inst` is not the first instruction in its basic block, then return the previous instruction of `inst`. Otherwise, return the last instructions of basic blocks that are the predecessor of the basic block where `inst` is.

.

Then for each term, we perform iterative dataflow analysis to calculate the `D-Safe`, `Earliest`, `Delay`, `Latest`, and `Isolated` sets.

At start, we create $mem_{dsafe}, mem_{earliest}, mem_{delay}, mem_{latest}$ and $mem_{isolated}$ for memorization. They are `instruction` and `boolean` maps that are used to save calculated results and for checking convergence.

To calculate $D\text{-}Safe$ for each instruction, we follow the definition from Knoop et al. [4]:

$$D\text{-}Safe(n, term) = \begin{cases} false & \text{if } n \text{ is the last instruction.} \\ Used(n, term) \bigvee & \\ Transp(n, term) \bigwedge \prod_{m \in Succ(n)} \bigwedge_{m \in mem_{dsafe}} mem_{dsafe}[m] & otherwise \end{cases}$$

Once we calculate `D-Safe` for an instruction n, we save the boolean result to $mem_{dsafe}[n]$. We stop calculating `D-Safe` until mem_{dsafe} doesn't change any more.

We then continue to calculate `Earliest`, `Delay`, `Latest`, and `Isolated` in the same way according to the equations defined by Knoop et al. [4] found in **Appendix (A)**.

Next, we compute the `Optimal Computation Points (OCP)` and `Redundant Occurrence (RO)` sets:

- $OCP(term) = \{\ n \mid mem_{latest}[n] \bigwedge \neg mem_{isolated}[n]\ \}$
- $RO(term) = \{\ n \mid Used(n, term) \bigvee \neg(mem_{latest}[n] \bigwedge mem_{isolated}[n])\ \}$

Once we have calculated the `OCP` and `RO` sets, we can then perform the `OCP-RO Transformation`. We first introduce a new auxiliary variable to store the computation of the term. The auxiliary variable in LLVM is a local variable created by an allocation instruction (AllocaInst). We insert the allocation instruction at the very start of function basic block. Then we insert the auxiliary variable before the instructions at the `Optimal Conditional Point` to store the result of computation of the term. Finally, we replace instructions at each `Redundant Occurrence` of the auxiliary variable. To do this in LLVM, we replace them with load instructions that load our auxiliary variable.

To test this transformation pass we created twelve unit tests designed to test basic functionality and corner cases involving conditional statements, loops, constants, globals, and function calls. Our final transformation pass correctly passes these test cases.

For our experiment we decided to test three different test cases

- Test 1 : `-reassociate -newgvn`
- Test 2 : `-reassociate -newgvn -pre`
- Test 3 : `-reassociate -gvn`

Test 1 serves as our control in this experiment, we use this test to base our effectiveness off of. Test 2 utilizes our new PRE transformation pass. Test 3 uses the LLVM GVN transformation pass which performs PRE. Since there is no existing standalone PRE transformation in LLVM, this was the closest to an existing solution to compare our results off of. Comparisons of this test to others may not be useful as performance discrepancies may occur due to the different the GVN algorithm used in these test, in addition to the different PRE algorithm. The main purpose of this test is to compare our transformation with current implementations. With our test script, we are able to run clang in `-O0` mode with only the specified passes being run. As a general note we decided to not perform our tests with a large amount of transformation passes, testing with more passes ends up drastically lowering the amount of opportunities for PRE. For example, running SROA drastically lowers the amount of PRE opportunities. We felt that these test cases allows us to maximize the opportunities for PRE in order to properly show the effectiveness of our pass.

To perform this experiment we used the LLVM Test Suite, testing on the SingleSource and MultiSource programs. To test our pass with this test suite, we designed a clang wrapper to wrap clang and opt together in order to control which passes are specifically run during compilation. Our transformation correctly passes all of these test cases, except for 0.01% of them. This remaining percentage is due to building issues due to the lack of robustness in our testing infrastructure. Nonetheless we believe that the results are comprehensive and still illustrate the capabilities of our transformation.

5 Experimental Results

In our data graphs, we aim to holistically show the results of our tests. The x-axis in the graphs represents the benchmark programs, and y axis the time difference in execution. To further elaborate, in Figure 2 and 3, for any single benchmark the time difference shows us:

Execution Time of the Benchmark in Test 1 - Execution Time of the Benchmark in Test 2

In figure 4 and 5, the time difference illustrates:

Execution Time of the Benchmark in Test 3 - Execution Time of the Benchmark in Test 2

In both of these tests, positive numbers mean that Test 2, which included our transformation pass, executed faster than the other tests and negative numbers mean that it executed slower.

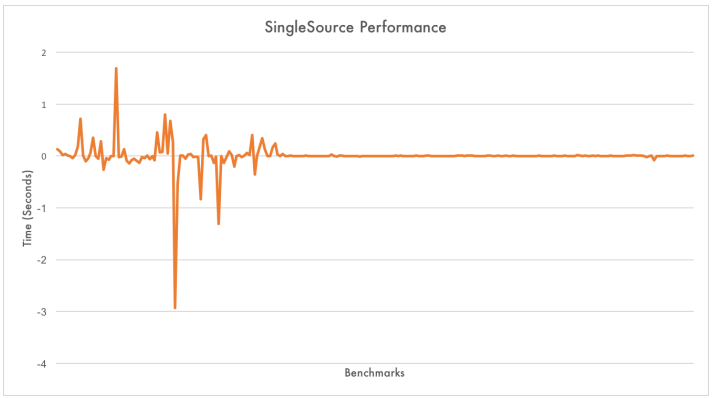


Fig. 2

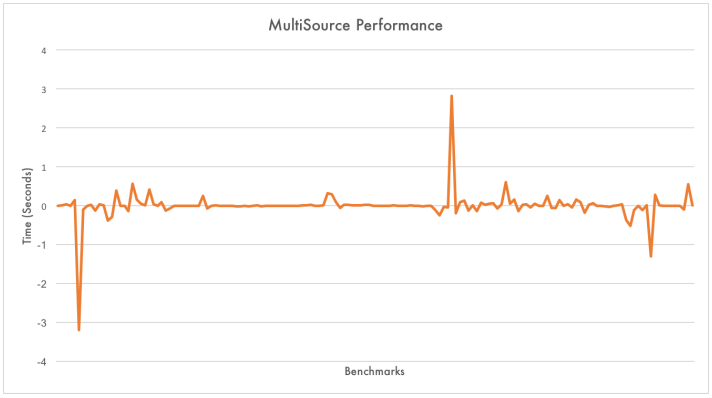


Fig. 3

Figure 4 and 5 can be found in Appendix B.

Result Analysis

We expected to see the biggest fluctuations in smaller SingleSource programs as these programs tend to have less `ROs`. Since we basically add multiple `load` instructions and a `store` instruction in place of a computation, we will see slower performance if the computation is used only once or twice in the program. But for the most part, we are glad that our transformation pass is able to meet similar performance for most of the SingleSource Benchmarks.

The performance of transformation pass is great on the MultiSource programs. A majority of the tests indicate that our transformation pass is able to be slightly faster than our control test. In one of our best benchmarks, we were able to see a 9% faster performance in the benchmark of a large program such as SQLite3 (from 4.29 seconds to 3.91 seconds).

Looking at our performance versus GVN's PRE algorithm is also encouraging (In Appendix B). While we cannot comment on the performance of the PRE algorithms, due to the different GVN algorithms, it is nice to see that our PRE algorithm does not seem to drastically hinder performance in the majority of the benchmarks.

In general we are satisfied with the performance of our PRE transformation. While we cannot claim that our algorithm will be able to increase the performance of all applications, we believe that we have demonstrated that there are situations where this transformation is relevant.

6 References

[1] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. 1997. A new algorithm for partial redundancy elimination based on SSA form. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation* (PLDI '97), A. Michael Berman (Ed.). ACM, New York, NY, USA, 273-286. DOI=http://dx.doi.org/10.1145/258915.258940

[2] Preston Briggs and Keith D. Cooper. 1994. Effective partial redundancy elimination. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation* (PLDI '94). ACM, New York, NY, USA, 159-170. DOI=http://dx.doi.org/10.1145/178243.178257

[3] Karthik Gargi. 2002. A sparse algorithm for predicated global value numbering. In *Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation* (PLDI '02). ACM, New York, NY, USA, 45-56. DOI=http://dx.doi.org/10.1145/512529.512536

[4] Jens Knoop, Oliver Rthing, and Bernhard Steffen. 1992. Lazy code motion. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation* (PLDI '92), Richard L. Wexelblat (Ed.). ACM, New York, NY, USA, 224-234. DOI=http://dx.doi.org/10.1145/143095.143136

7 Appendix

(A) D-Safe, Earliest, Delay, Latest, Isolated

- $D\text{-}Safe(n, term) = \begin{cases} false & \text{if } n \text{ is the last instruction.} \\ Used(n, term) \bigvee & \\ Transp(n, term) \bigwedge \prod\limits_{m \in Succ(n)} \bigwedge_{m \in mem_{dsafe}} mem_{dsafe}[m] & otherwise \end{cases}$
- $Earliest(n, term) = \begin{cases} true & \text{if } n \text{ is the first instruction} \\ \sum\limits_{m \in Pred(n)} \bigwedge_{m \in mem_{earliest}} & \\ (\neg Transp(m, term) \bigvee & \\ \neg mem_{dsafe}[m] \bigwedge mem_{earliest}[m] \ ) & otherwise \end{cases}$
- $Delay(n, term) = mem_{dsafe}[n] \bigwedge mem_{earliest}[n] \bigvee$

$$\begin{cases} false & \text{if } n = s \\ \prod\limits_{m \in pred(n)} \bigwedge_{m \in mem_{delay}} \neg Used(m, term) \bigwedge mem_{delay}[m] & otherwise \end{cases}$$

- $Latest(n, term) = mem_{delay}[n] \bigwedge (Used(n, term) \bigvee \neg \prod\limits_{m \in Succ(n)} mem_{delay}[m])$
- $Isolated(n, term) = \prod\limits_{m \in Succ(n)} \bigwedge_{m \in mem_{isolated}} (mem_{latest}[m] \bigvee (\neg Used(m, term) \bigwedge mem_{isolated}[m])$

(B) Test 2 vs Test 3 Graphical Results

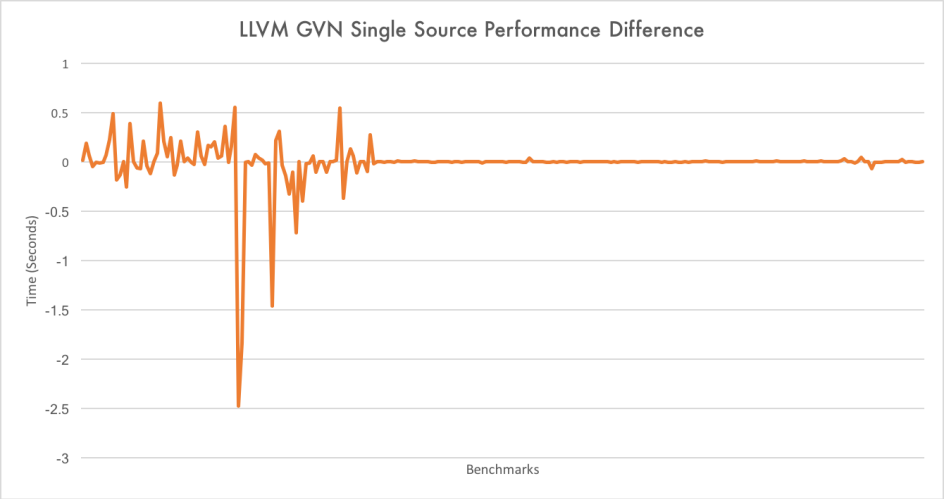


Fig. 4

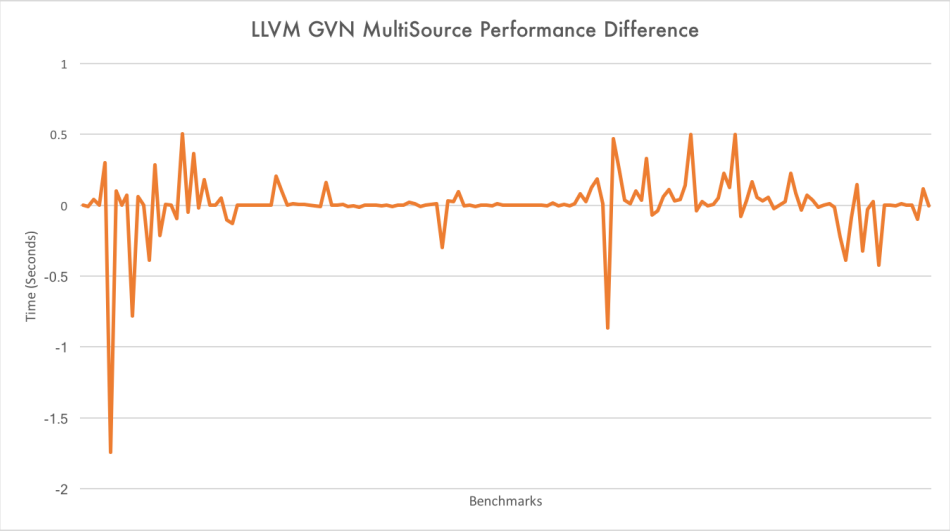


Fig. 5

(C) Instructions on how to run

1. GitHub Clone Link: `https://github.com/TarunRaj16/PREviaLCM.git`
2. Clone repo or uncompress zip into `lib/Transforms/` in llvm source
3. Add following into `lib/Transforms/CMakeLists.txt` : `add_subdirectory(PREviaLCM)`
4. Build llvm
5. `-pre` invokes optimization
6. Go to test directory. The tests directory in the project contains unit tests and some LLVM Single Source tests.
7. Replace `LLVM_DIR` and `PRE_LIB` to point to LLVM build directory and PRE shared library respectively. The following are some useful commands (replace foo with program you want to test):
   - `make foo.ll` : creates an *.ll without our PRE Transformation pass
   - `make foo-pre.ll` : creates an *.ll with our PRE Transformation pass
   - `make foo.exe` : creates an *.exe without our PRE Transformation pass
   - `make foo-pre.exe` : creates an *.exe with our PRE Transformation pass

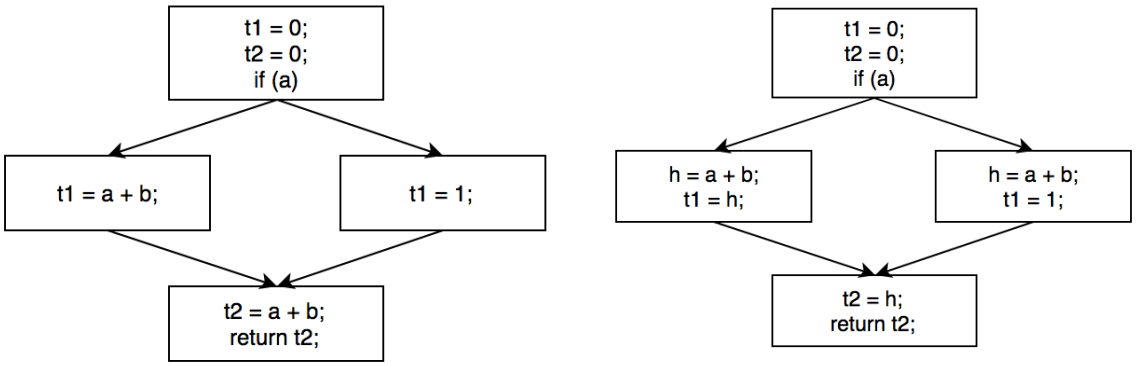
(D) Extended Example

- The code below shows that `a+b` is partially redundant. After calculation, `OCP` is {`5:t1=a+b`, `7:t1=1`} and `RO` is {`5:t1=a+b`, `9:int t2=a+b`}. We create an auxiliary variable `h=a+b`. We insert `h=a+b` before `5:t1=a+b` and `7:t1=1`. Then we replace `5:t1=a+b` and `9:int t2=a+b` with `t1=h` and `t2=h`. Figure 6 on the next page shows how the transformation would look like.

```
1: int test(int a, int b) {
2:   int t1 = 0;
3:   int t2 = 0;
4:   if (a) {
5:     t1 = a + b;
6:   } else {
7:     t1 = 1;
8:   }
9:   int t2 = a + b;
10:  return t3;
11: }
```

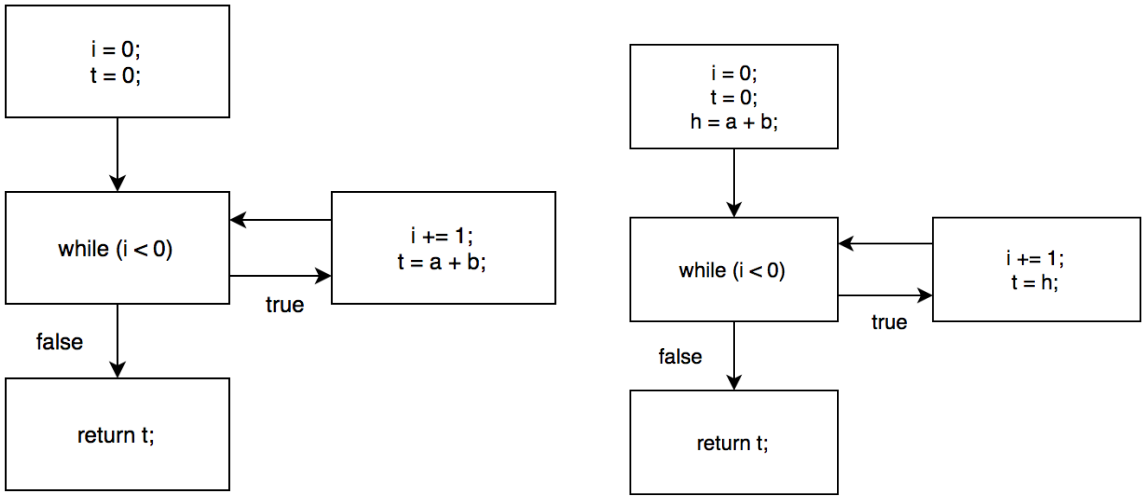
- The code below demonstrates that loop invariants are partially redundant. After calculation, `OCP` is {`4:while(i<10)`} and `RO` is {`6:t=a+b`}. We create an auxiliary variable `h=a+b`. We insert `h=a+b` before `4:while(i<10)`. Then we replace `6:t=a+b` with `t=h`. Figure 7 on the next page shows how the transformation would look like.

```
1: int test(int a, int b) {
2:   int i = 0;
3:   int t = 0;
4:   while (i < 10) {
5:     i += 1;
6:     t = a + b;
7:   }
8:   return t;
9: }
```



(a) without PRE (b) with PRE

Fig. 6: Comparison of CFG with PRE and without PRE



(a) without PRE (b) with PRE

Fig. 7: Comparison of CFG with PRE and without PRE