Debugging Optimized Programs using Black-Box Equivalence Checker

Advisor: Prof. Sorav Bansal
Vaibhav Kurhe

June 2021

Acknowledgements

I would like to wholeheartedly thank my advisor Prof. Sorav Bansal for his constant guidance and support throughout the project and for floating courses on Compiler Design and Optimization, which also helped me in my project. I am grateful to Shubhani and Abhishek for helping me and solving my doubts whenever I was stuck. I am thankful to Pratik for collaborating with me on this project. Pratik worked on and took the responsibility for the modules 3.4(Expression Generator) and 3.5.1(TFG Traversal). I would also like to thank Philip Craig - Lead Gimli developer - for providing his support and reference code while using the libraries gimli and object.

Abstract

Debugging a program usually necessitates disabling the compiler optimizations. This is due to the loss of much of the debugging information in presence of multiple compiler optimizations. Yet, there are some scenarios when a developer would need the ability to debug optimized programs. This thesis presents a technique that can improve the debugging information in an optimized program, to allow debugging the optimized program in debuggers such as gdb.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Debugging is a crucial part of the software development lifecycle. There are many ways to debug a program such as rubber duck debugging, using print statements within the code, analyzing the logs, etc. One such important technique involves the use of a debugger tool, which helps a programmer trace the program step-by-step as it runs.

A debugger tool such as gdb, uses the debugging information present inside an executable program to allow stepping through the program when it runs. There are multiple standards defining the structure of an executable program file and ELF is one such standard widely used across Linux and UNIX based systems. Similarly, there are many standards defining the debugging information format in an executable program file; DWARF is one such widely used standard.

Although the production code is almost always compiled with the highest level of optimization, when debugging, programmers typically have to use flags that instruct the compiler to disable all optimizations and insert the debugging information into the executable. In scenarios where it is necessary to debug the optimized executable, such as when a programmer is developing a tool that applies code transformations/optimizations or when a certain bug is present only in an optimized version of a program, it becomes difficult and error-prone due to the rigorous transformations/optimizations performed by the compiler or the tool itself.

We introduce a tool to improve the debugging information in DWARF format present inside an optimized ELF executable program, so that we could step through the updated executable program in a debugger such as gdb & inspect the variables, in similar way as much as possible to how it is done for an unoptimized program.

We do this by using an equivalence checker – it's a tool which when given two input programs, checks whether the two programs are equivalent in terms of their

functionality and if so, generates a mathematical proof. We first generate optimized and unoptimized versions of the same source program written in C. We pass them to an equivalence checker that produces an equivalence proof, which has correlations between the two programs. We further process the proof, to get predicates, i.e. the value/address of source variables in terms of registers used in the optimized executable program. We finally convert these predicates into DWARF expressions and add them into the debugging information present in the optimized program.

The thesis is organized as follows:-

We start with a concrete example denoting the issues faced while debugging an optimized program inside a debugger(gdb). We then sketch out a high-level architecture of the tool pipeline and discuss each component in a separate section. This is followed by the evaluation section at the end.

Chapter 2

Issues in debugging optimized programs

Unlike for unoptimized code, while trying to debug an optimized code, we may come across a variable being optimized out, a variable being re-allocated to a register, etc. So, it becomes difficult to track a variable and debug the program. To demonstrate this, we show the difference in debugging an unoptimized vs an optimized program.

In figures 2.1 and 2.2, we show part of the debugging sessions for a function s000() inside the TSVC benchmark.

Following is the relevant source code for s000() function:-

Source Code 2.1: s000 function

```
#define LEN 32000
#define TYPE int
#define lll LEN
TYPE val = 1;
__attribute__((aligned(16))) TYPE X[lll], Y[lll];
int s000()
{
    for (int i = 0; i < lll; i++) {
        X[i] = Y[i] + val;
    }
    return 0;
}
```

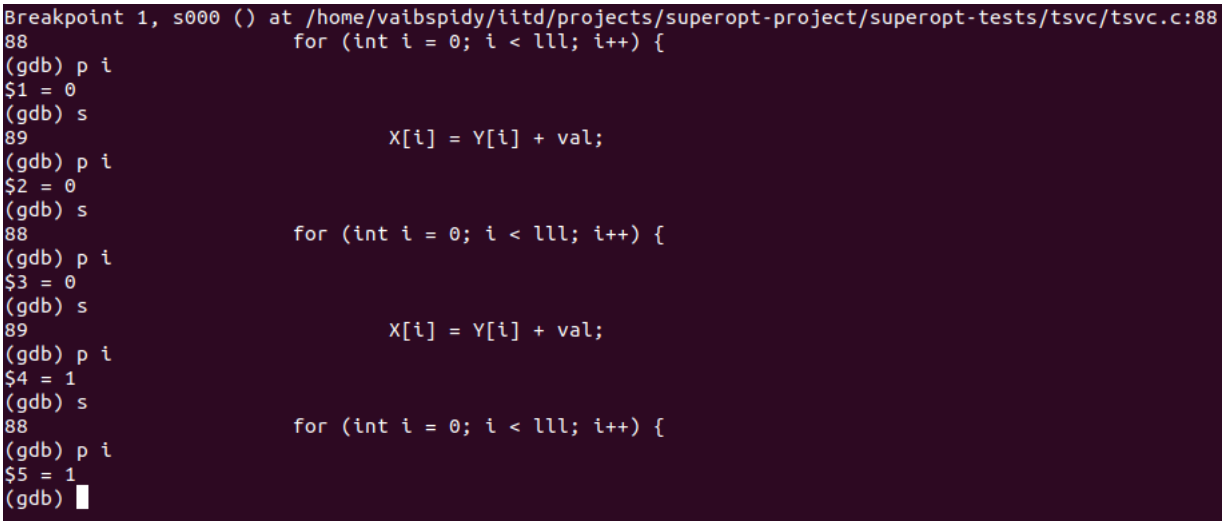

Figure 2.1: GDB debugging session - Unoptimized code

```
int main() {
    s000();
    return 0;
}
```

In the unoptimized code(2.1), we can see that the iterator variable - *'i'* is updated in the for loop. Initially it was zero and as we stepped through the unoptimized program, it got updated to 1 when we reached the "i++" instruction. But, in the optimized code(2.2), the value for variable *'i'* didn't change – it remained as zero throughout the loop.
This happened because at optimization level O3, the compiler has applied rigorous transformations to the original program and it resulted in storing the value of variable *'i'* into a register, as it's being frequently accessed.

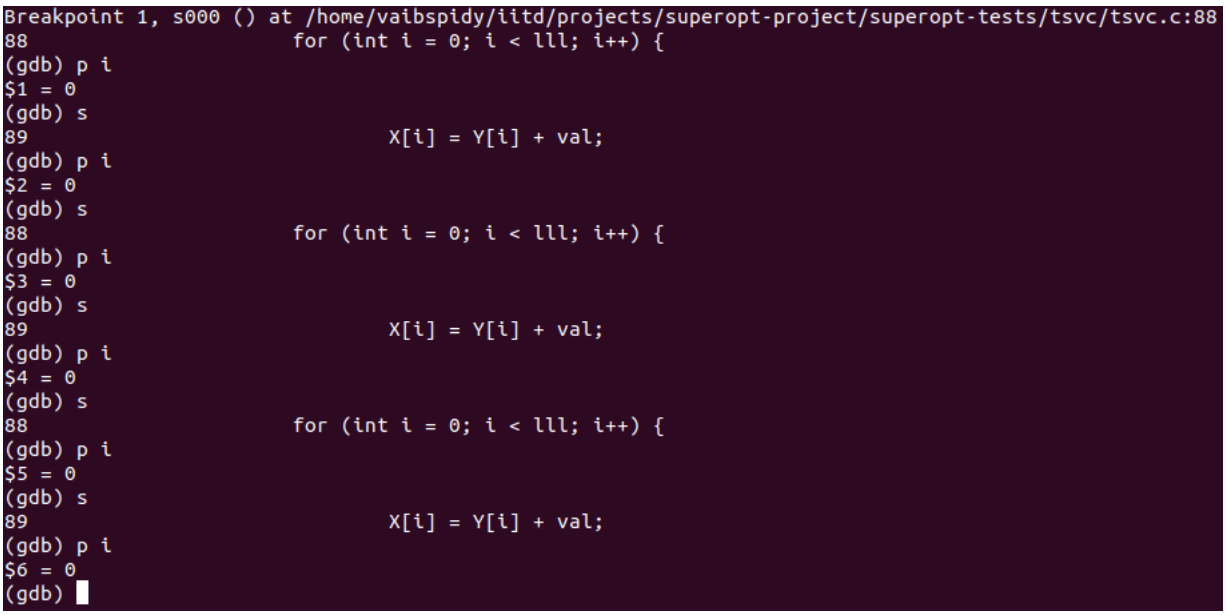

Figure 2.2: GDB debugging session - Optimized code

Chapter 3

Improving Debugging Information

We will first describe the terminologies and tools used to achieve our desired goal:

3.1 Equivalence Checker

Determining if two Turing machines are equivalent is a classical problem in theoretical computer science, which is undecidable in its generality. Equivalence Checker [5] [7] is realization of such a tool which when given two input programs, checks if they are equivalent and produces a proof of equivalence.

3.2 DWARF Debugging Standard

For a debugger such as GDB to work on an ELF executable, it needs debugging information in a particular format available inside the ELF. DWARF is one such standard/format for defining the debugging information in an ELF file.

DWARF debugging information consists of multiple sections such as **.debug_info**, **.debug_loc**, etc. The **.debug_info** section consists of **DIEs (DWARF Debugging Information Entities)** which can denote individual units of a program such as variables, functions, different types, etc.

These DIEs are interlinked with each other using parent-child and sibling relationships to form a graph. And the **.debug_loc** section consists of the information about the locations and/or values of variables in the form of location lists. Each location list essentially represents values of the corresponding variable at different program points using DWARF expressions.

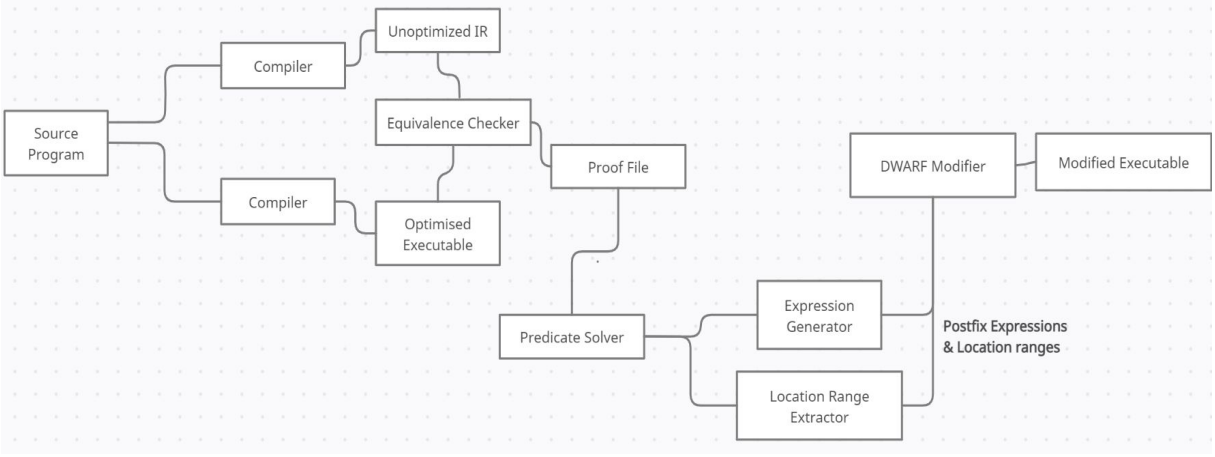

Figure 3.1: Tool Pipeline

A DWARF expression is a postfix expression, that describes how to compute a value or name a location during debugging of a program. It is expressed in terms of a set of DWARF operations that operate on a stack of values.

3.3 Tool Pipeline

The overall pipeline of our tool is shown in the Figure [3.1]. To improve debugging information of a particular program e.g. s000.c, we first compile it with -O0 and -O3 flags separately. Then, we pass these unoptimized and optimized versions to an equivalence checker, which outputs a mathematical proof of how those two programs are equivalent in a .proof file.

We take this proof file and process it in different stages as follows:-

3.4 Expression Generator

The high-level logic for main() function in Expression Generator module is shown in algorithm 1.

A proof file from equivalence checker consists of a predicates section, where program points from unoptimized and optimized versions of a source program are correlated and predicates are established between the source and destination variables. An example of a predicate in proof file is shown as below:-

```
=pc Lfor.body%1%1_L3%1%0 type bv pred 6
```

input : A predicates file, LLVM-to-src-var-mapping
result: Generates the solutions for given set of equations between the unoptimized and optimized variables

```
1  while there is some predicate to read do
2  |  pc ← read the PC;
3  |  pred ← read and parse the predicate;
4  |  pred ← PredicateRearrange(pred);
5  |  atoms_lhs ← GetArithmeticAddSubAtoms(pred.LhsExpr);
6  |  lhsMap ← Maintain the set of atoms_lhs for pc;
7  |  rhsMap ← Maintain the set of pred.RhsExpr for pc;
8  |  predicateMap ← Maintain the set of pred for pc;
9  end
10 for each pc in predicateMap do
11 |  solutions ← Solve(pc, set of pred, lhsMap, rhsMap);
12 end
```

Algorithm 1: Expression Generator: Main() function

input : set of predicates for given PC, lhsMap, rhsMap
output: solutions to the set of predicates

```
1 matrixX ← CreateVariableMatrixX();
2 matrixA ← CreateCoeffMatrixA();
3 matrixB ← CreateRhsMatrixB();
4 RearrangeMatrices(matrixX, matrixA);
5 augmentedMatrix ← CreateAugmentedMatrix(matrixA, matrixB);
6 augmentedMatrix ← ConvertToRowEchelonForm(augmentedMatrix);
7 solutions ← GetSolutionsBottomUp(augmentedMatrix, matrixX);
8 return solutions;
```

Algorithm 2: Expression Generator: Solve() function

```
=LhsExpr
1 : input.src.llvm-%i.0 : BV:32
2 : 4 : BV:32
3 : bvmul(1, 2) : BV:32
4 : 4294839296 { -128000 } : BV:32
5 : bvadd(3, 4) : BV:32
=RhsExpr
1 : input.dst.exreg.0.0 : BV:32
```

The above predicate specifies the relation between LLVM source variable i.0 and a destination program register exreg.0.0 (which is %eax) at correlated PCs defined by the labels Lfor.body%1%1 and L3%1%0 from source and destination programs respectively. The above predicate is built from bitvector operations of specified size (32-bits). The simplified infix form of the above equation/predicate is as follows:-

```
4 * i.0 - 128000 = %eax
```

We read and parse the predicates from the proof file one-by-one and we rearrange them by calling PredicateRearrange(). The function pushes all the constants to RHS, combines all other terms in the predicate and puts them into LHS. The above example after rearrangement looks like this:-

```
4 * i.0 - %eax = 128000
```

We also extract individual terms from LHS using GetArithmeticAddSubAtoms() function as shown in Algorithm 1 at line no. 5. We then store and maintain these terms from LHS and the constant expression from RHS into helper maps lhsMap and rhsMap. This is done so that we could readily create matrices needed in the next steps (Algorithm 2). We also maintain a predicateMap that stores the set of predicates valid for a PC.

Once we parse and process the predicates from the proof file, we try to solve them for each PC by calling Solve() function, as shown in Algorithm 1, line no. 11.

3.4.1 Solving predicates

We describe the individual steps specified in algorithm 2 - Solve() function.

- **Creating Variable Matrix**
  We first scan the individual terms from LHS of each predicate. We remove the coefficients from the expression variables and we insert the variables into a helper map. As the same variable may be involved into multiple predicates, we add the

variable to the map only if it is unique. For each expression variable, we add its corresponding index i.e. we number the variables.
We add all the unique variables from map into a variables_vector and this way, we get our variable column matrix.

- **Creating Coefficient Matrix and RHS Matrix**
  Once we get the variables_vector (variable column matrix), we know the no. of entries in a row from Coefficient Matrix will be same as the size of the variables_vector. Then, for each predicate, using lhsMap that we maintained, we go through the individual terms of its LHS and extract the coefficients to create a Coefficient Matrix.
  Similarly, using the rhsMap maintained, we go through constant expressions in RHS for each predicate and create an RHS matrix of constants.

- **Rearranging Matrices**
  The Coefficient Matrix A and the Variable Matrix X created so far have no ordering in particular among the source and destination predicates. Consider below examples of the matrices - coefficient matrix A of order 3×4 and the variable matrix X of order 4×1.

$$\begin{bmatrix} 0 & -16 & 1 & 0 \\ 6 & 0 & 0 & -10 \\ 0 & -4 & 0 & 8 \end{bmatrix} \begin{bmatrix} input.src.llvm - \%j.0 \\ input.dst.exreg.0.2 \\ input.src.llvm - \%i.0 \\ input.dst.exreg.0.3 \end{bmatrix}$$

  We rearrange them so that the source variables(starting with "input.src") come before the destination variables(starting with "input.dst") inside the Variable Matrix X. The matrix X after the rearranging is shown below.

$$\begin{bmatrix} input.src.llvm - \%j.0 \\ input.src.llvm - \%i.0 \\ input.dst.exreg.0.2 \\ input.dst.exreg.0.3 \end{bmatrix}$$

  We moved all the source variables to the top of the column matrix X and the destination variables to the bottom. To maintain our original predicates, we correspondingly rearrange the columns of the Coefficient Matrix A i.e. if we exchange a source variable at $n1^{th}$ row in variable matrix with a destination variable at $n2^{th}$ row, we correspondingly exchange the whole $n1^{th}$ column with $n2^{th}$ column in the Coefficient Matrix.

  For our example, as we exchanged the 2nd and 3rd rows of the variable matrix X, we exchange the 2nd column in coefficient matrix A (corresponding to 2nd row

in variable matrix X) with the 3rd column (corresponding to 3rd row in variable matrix) as follows:

$$\begin{bmatrix} 0 & 1 & -16 & 0 \\ 6 & 0 & 0 & -10 \\ 0 & 0 & -4 & 8 \end{bmatrix}$$

- **Creating Augmented Matrix**
  We combine the Coefficient Matrix A with the RHS Matrix B to create an Augmented Matrix $A|B$.

- **Converting into Row Echelon Form**
  Now we convert our Augmented Matrix into the Row Echelon Form by performing the standard Matrix Row transformations.

- **Bottom-Up solutions**
  Once we get the Row Echelon Form for our Augmented Matrix, we traverse the matrix from bottom-to-top. We would first get a series of equations involving only the destination variables, followed by a series of equations with source variables. Even after performing a set of row transformations during the Row Echelon Form conversion, this will hold true, as we had arranged our variable vector and coefficient matrix such that the source variable rows and columns would come before destination variables in the variable vector and coefficient matrix respectively. We have also not interchanged any columns in the coefficient matrix. And even so, if an only-destination variable row comes before a source variable row, that would mean the leading non-zero coefficient of a row isn't to the right of the leading non-zero coefficient of the row above it. This would violate the Row Echelon Form property and hence it wouldn't be possible.
  So we go from bottom to top, storing the values of each destination variable in terms of the others and when we come across source variable predicates, we use the already calculated expression values to get the expressions for the source variables.

Next, we use the solutions thus generated by the Solve() function as shown in algorithm 2, and try to expand the range of PCs over which a predicate is valid. This is described in the next section.

3.5 Location Range Extractor

Before equivalence checking, the unoptimized and optimized versions of the source program are transformed into corresponding Transfer Function Graph (TFG) files.

Domain	{ P	where P is a predicate with an LHS and RHS expression and LHS = RHS}
Direction	Forward	
Transfer function	f_{xfer}, as specified in algo.3	
Meet operator	$\otimes$ as specified in algo.4	
Boundary condition	out$[n^{start}] = \{\}$	
Initialization to $\top$	in$[n] = \{\}$ for all non-start nodes	

Table 3.1: Valid Predicates Data Flow Analysis Formulation

Along with encoding the control flow, a TFG also has a set of transfer functions for each edge in the graph.

The equivalence checker typically outputs the predicates over loop heads in a program i.e. for each predicate, only a single PC is provided by the equivalence checker. But, to we would like to improve/add debugging information to as many PCs as possible. Thus, after getting the set of predicates for each PC, we traverse the TFG for the optimized program to try to cover as much other PCs in the optimized program as possible.

3.5.1 TFG Traversal

Initially, we started with a simple TFG traversal from a loop head in the optimized program. We start with the loop head and traverse next edges until we either find a branch or we find that next instruction is modifying any of the destination variables(registers) present in a predicate.
This way, we could expand the predicates within a simple loop body, although we could not handle multiple branches and merging of predicates when two branches meet.
A Data Flow Analysis (DFA) technique is suitable for flowing information throughout the program, involving any number of loops or branches and so we designed a DFA that is described in the next section.

3.5.2 Valid Predicates Data Flow Analysis

As the same common framework can be used for implementing multiple, different Data Flow Analyses, we use a base DFA framework from superopt library, which is part of the equivalence checker infrastructure.
The formulation of the DFA is as specified in the table 3.1. The domain of values is the set of predicates and the direction of the valid predicates DFA is forward.

We pass the solutions, i.e. the set of predicates for each PC, generated by the Solve() function to our DFA. This will be used later during the DFA, to simulate the GEN set (generating a new set of predicates at a PC).
We initialize the set of predicates at each PC to be an empty set.

We motivate the use of Transfer Function and Meet Operator as specified in the table 3.1 by providing initial version of Transfer Function and Meet Operator first, followed by the set of modifications performed in a few of the next subsections.

3.5.2.1 Transfer Function

The transfer function takes as input, the instruction through which we want to pass our DFA value (or an edge in a TFG), the current DFA IN value or the set of predicates for the program point just before the current instruction and the current DFA OUT value (set of predicates) for the program point just after the current instruction.

We extract the set of destination variables (registers) being modified at the current instruction (or edge in the TFG) and check if any of the predicates present in the DFA IN value have those registers present in their RHS expression. As specified earlier, the LHS expression will have just the source variable and the RHS will have the value of the source variable in terms of destination variables (registers). So, if any of the modified registers are present in a predicate's RHS expression, we kill that predicate, as that predicate will no longer be held true at the program point after the current instruction due to change in the value of the register.

After killing the appropriate predicates, we use the solutions (the set of predicates valid over a PC) generated earlier from the Solve() function to simulate the GEN set (generation of new predicates while we pass through the current instruction). We merge these new predicates with the current set of predicates remained after applying KILL operation above.

Once we finish transferring the DFA values (set of predicates) through an instruction (an edge in a TFG), we perform the Meet operation as described in the next subsection.

3.5.2.2 Meet operator

The meet operation takes two DFA values (set of predicates) as input, one which is a DFA value got after performing transfer function on the DFA IN value of an edge/instruction and another which is the current DFA OUT value of the edge/instruction. We compare the both the set of predicates with one another and check if any two

of the predicates have the same source variable. If so, we randomly pick one of the predicates, otherwise we pick both of them.
This way, we get a new DFA OUT value for the current instruction/edge in TFG. As the worklist algorithm is being used in the DFA implementation, we also need to check and return if the original DFA OUT value was changed or not. Based on this, the worklist algorithm will continue the DFA further or stop it.

3.5.3 Modifications to DFA

Consider the head of a simple loop in the TFG (as shown in figure 3.2) of a program (as shown in source code listing 2.1). The loop head(labeled as L3%1%0) will have an incoming forward edge from the top and a backward edge that will be coming from the loop tail(labeled as L16%1%1). Let's say the set of predicates coming from the forward edge and the backward edge are preds_forward and preds_backward. In our earlier implementation of the DFA, while merging the two set of predicates, we would pick only one if the two predicates have same source variable. But, the choice of one predicate over another was random.

For a typical loop, an iterator variable will be initialized to some constant and then the loop body starts. So, we would get a constant value predicate (e.g. src_variable == constant) from the forward edge, while the back edge would give us the predicate that is valid inside the loop body, which would typically be a non-constant predicate (e.g. src_variable == non-constant expression in terms of registers). So we should pick the predicate having non-constant expression in RHS.

Thus we assigned TAGs to every predicate. A predicate coming from a forward edge would have a FORWARD TAG, while the one coming from a back edge will have a BACKWARD TAG. We modified our Transfer Function to change the TAGs for incoming predicates, based on whether the current edge is back-edge or not. These TAGs will be used in the meet operation - when we come across two predicates having same LHS expression i.e. source variable, we would pick one based on their TAGs. So if their TAGs are different, we will pick one with the BACKWARD TAG. On the other hand, if their TAGs are same, then we pick any of them randomly. This way we could differentiate between predicates coming from a tail of the loop and those coming from the program point before the loop head.

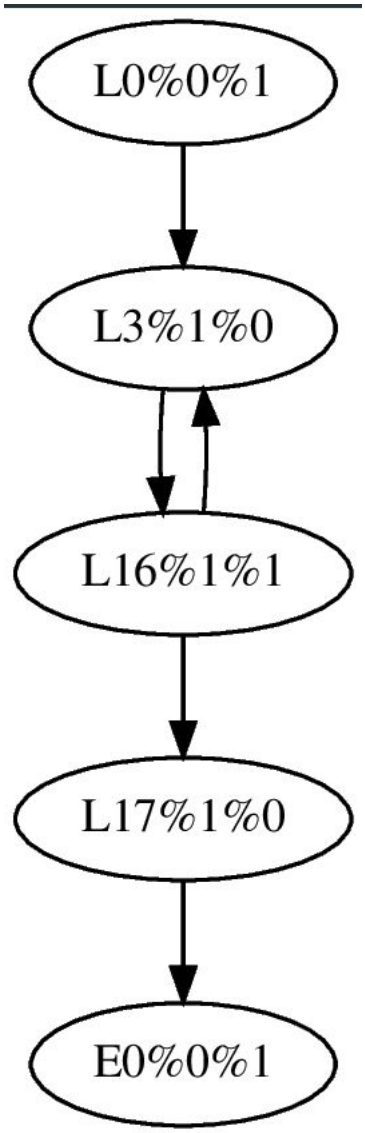

Figure 3.2: Example of a TFG involving a simple loop

3.5.4 Making the DFA more precise

As we saw in the earlier section, in the valid predicates DFA, when we transfer a DFA IN value (set of predicates) through an instruction/edge in TFG, we check whether the current instruction modifies any of the registers present in any of the input predicates. If so, we kill the predicates i.e. remove them.

But this limits the range of PCs over which we can generate predicates. As we show later, based on the compiler used to generate the code, an instruction modifying the register might come immediately after the loop head or near the loop tail. So for the programs of former kind (modifying instruction coming immediately after loop head), the predicates will get killed immediately and they will not hold over any of the later PCs in the loop body. Also, for the programs of latter kind (modifying instruction near the loop tail), we will not be able to flow the predicates outside the loop.

So we modified our DFA implementation to be more precise about the current instruction. We check if the current instruction that modifies a register present in some predicate has a reversible operation e.g. addition, subtraction. If so, we do not kill the original predicate. Instead, we modify the predicate itself by adding a reverse of the computation done by the current instruction. e.g. if the original predicate was "src_var == %ecx", and the current instruction was "%ecx = %ecx + 8", we would change our predicate to account for the change in register value i.e. we would make it as "src_var == %ecx - 8".

But, due to this change, combined with the priority for backedge predicates would imply that at the loop head in a program, we would pick the predicate "src_var == %ecx - 8" over "src_var == %ecx" in the next DFA iteration. Also, this would go out indefinitely. So, we added another TAG for predicates – ORIGINAL, which will have higher priority than the BACKEDGE TAGs. We set this ORIGINAL TAG for all those predicates that are part of the GEN set, when we simulate generating new predicates in the Transfer Function. If we consider the whole loop body now, this has the effect of doing the iterator variable updation at the loop head instead of the actual place in the optimized program, as only then the new updated value of the register will be used in the predicate.

The valid predicates DFA with changes described so far is shown in algorithm 3 and algorithm 4.

3.5.5 Adding a Backward DFA pass

The valid predicates DFA pass that we describe in the previous sections, would expand the range of PCs for which a predicate is valid in the forward direction.

```
input : TFG edge e, DFA IN value inPreds, DFA OUT value outPreds
result: Updated DFA OUT value outPreds
1  regNames ← GetModifiedRegNames(e);
2  genPreds ← GetPredsAtPc(e.to_pc);
3  thisPreds ← inPreds;
4  for pred in thisPreds do
5      if e is backEdge then
6          tag pred as backEdge;
7      end
8      else
9          tag pred as forwardEdge;
10     end
11 end
12 for pred in thisPreds do
13     for regName in regNames do
14         if regName present in pred then
15             if reversible computation at e then
16                 modify pred to add complement operation;
17             end
18             else
19                 remove pred;
20             end
21         end
22     end
23 end
24 for genPred in genPreds do
25     for pred in thisPreds do
26         if genPred.LhsExpr = pred.LhsExpr then
27             remove pred;
28         end
29     end
30 end
31 insert all genPreds into thisPreds with original tag;
32 Meet(thisPreds, outPreds);
```

Algorithm 3: Data Flow Analysis: Transfer Function

```
input  : DFA IN value thisPreds, DFA OUT value outPreds
result : Updated DFA OUT value outPreds
returns: boolean value representing if outPreds was modified
1  for pred in thisPreds do
2      for outPred in outPreds do
3          if pred.LhsExpr = outPred.LhsExpr then
4              pick one based on tag where original > backEdge > forwardEdge;
5          end
6          else
7              pick both;
8          end
9      end
10 end
11 if outPreds is modified in the process then
12     return true;
13 end
14 else
15     return false;
16 end
```

Algorithm 4: Data Flow Analysis: Meet Operation

To extend them even further where possible, we implemented a Backward valid predicates DFA pass. When we finish our Forward valid predicates DFA, we get the set of predicates for each PC. We run our Backward DFA pass in sequence after the Forward DFA and use the DFA values generated from Forward DFA as an input to the Backward DFA.

We incorporate all the modifications specified for the Forward DFA in earlier sections into the Backward DFA e.g. the prioritizing predicates based on their TAGs - the ORIGINAL TAG having the highest priority, followed by the BACKWARD and the FORWARD TAG respectively. There are some changes due to reversal of the direction of information flow, that are specified below.

In the Transfer Function of the Backward DFA, we do the complement of the operations as done inside a Forward DFA. e.g. we would create GEN set (set of predicates) at the IN program point of the current instruction as opposed to OUT that is done in the Forward DFA.

Also, while handling a register-modifying instruction and doing the reversible computation in the predicate, we would do addition or subtraction if we pass through an instruction doing addition or subtraction respectively. This is because the DFA values are moving backwards through the edge. So, passing through an instruction doing addition in backwards direction has an effect of doing subtraction over the involved register.

Once we get the results(set of predicates for each PC) from Backward DFA, we calculate the PC range for each predicate, convert the predicate into Postfix form and then pass them along with the corresponding PC range to the DWARF Modifier.

3.6 DWARF Modifier

DWARF Modifer reads the predicates generated after the Data Flow Analysis from STDIN. It takes two arguments :- the input object file(the object file to be modified) and the output object filename(to be generated).

The high-level logic for DWARF Modifier is specified in algorithm 5 and algorithm 6.

We first open the input object file and create a memory-map of its file contents. We then parse the memory-map contents and create an object::Object struct out of it.

Next, we create a writable Object using write::Object::new(), which will have the same format and architecture as that of the input object. We set its mangling to be none and its flags to the input object flags.

input : Input object file inputFile, Output object filename outputFile, STDIN
result: Creates Output object file with updated debug information

```
1  inputObject ← Open(inputFile);
2  create new outputObject;
3  for each section in inputObject do
4  |  if not a metadata or a debugSection then
5  |  |  create corresponding outputSection;
6  |  |  add outputSection to outputObject;
7  |  end
8  end
9  for each symbol in inputObject do
10 |  if symbol is not null then
11 |  |  create corresponding outputSymbol;
12 |  |  add outputSymbol to outputObject;
13 |  end
14 end
15 for each section in inputObject do
16 |  for each relocation in section.Relocations do
17 |  |  create corresponding outputRelocation;
18 |  |  get corresponding outputSection from outputObject;
19 |  |  add outputRelocation to outputSection;
20 |  end
21 end
22 outputObject ← RewriteDwarf(inputObject, outputObject);
23 write out the outputObject contents to outputFile;
```

Algorithm 5: DWARF Modifier: Main() function

```
input : Input Object inputObject, Output Object outputObject
result: Stores the updated debug information into Output Object
        outputObject
1  for each debugSection in inputObject do
2  |  (sectionData, relocations) ← GetSection(debugSection);
3  end
4  readDwarf ← add sectionData and relocations for each debugSection;
5  buffer ← read from stdin;
6  functionName ← read from buffer;
7  varMap ← ReadExistingLocationLists(readDwarf);
8  writeDwarf ← CreateWritableDwarf(readDwarf, convertAddress);
9  for each predicateLine in buffer.lines do
10 |  funcId ← GetFuncId(functionName);
11 |  varId ← GetVarId(predicateLine.srcVar);
12 |  varInfo ← GetVarInfo(varMap);
13 |  dwarfExpr ← CreateDwarfExpr(predicateLine.RhsExpr);
14 |  for each existing locRange in varInfo do
15 |  |  ProcessLocation(locRange, predicateLine.NewLocRange);
16 |  |  newLocList ← add processed locRange;
17 |  end
18 |  newLocList ← add predicateLine.NewLocRange;
19 |  insert the newLocList into debugLoc section;
20 |  modify locAttr in variable DIE to refer to newLocList;
21 end
22 write each debugSection from writeDwarf to outputObject;
```

Algorithm 6: DWARF Modifier: RewriteDwarf() function

Now, we create a HashMap – outSections, which will basically map an input object index to the output object id. We traverse all the sections of the input object (while skipping the Metadata and the ".debug_*" sections) and create corresponding section in the output object. We then add a mapping between the input section index and the output section id, which will be used later.

Once the sections in the input object file are processed, we read the symbols in the input object file. We create a HashMap – outSymbols, which has a mapping from the input object symbol index to the output object symbol id. We traverse the symbols from input object (skipping ones of type Null), convert the input section (from object::SymbolSection to object::write::SymbolSection), flags and value for the symbols accordingly, so as to create the corresponding object::write::Symbol. We add the mapping from input object symbol index to the output object symbol id into the HashMap - outSymbols.

Finally, we read the input object sections, to get the relocations for each section and covert them into write::Relocation structs. For each output object section, we add the corresponding relocation thus generated into the object file.

We then call a RewriteDwarf() function with the arguments inObject, outObject and outSymbols, where only the outObject is mutable.

Once we get the modified out_object from the RewriteDwarf() function, we call its write() function to write out the data to a string and finally call fs::write() to write this string to the output object file path.

3.6.1 Rewriting DWARF debug information

The function RewriteDwarf() accepts three arguments :- the input object, output object (mutable) and the symbols map (from input symbol index to output symbol id).

We first call GetSection() on the input object, for every debug section – e.g. .debug_info, .debug_line, .debug_str, .debug_loc, etc. to get the corresponding section's data and relocations. Then, we combine this information to create instances for the structs such as gimli::read::DebugInfo, DebugAddr, DebugLine, etc. And we encapsulate these instances to create a gimli::read::Dwarf instance.

Now, we read from STDIN, to get the intermediate postfix expressions, generated at previous step in out tool pipeline. We store it in a buffer as a string. Then, we read each line one by one. There are fields marked with "=Function" and "=ZeroAddress", which denote the function name and the starting/low address for that function.

Next, we define a HashMap named VarMap, where we will store a mapping from

a variable name to the corresponding DWARF expression and the address range over which that expression is present. Then we call a function - ReadExistingLocationLists() - which traverses through the DIEs starting from the function and stores all the existing variable's debug information into the VarMap defined earlier.

Now, we call gimli::write::Dwarf::from() function to convert the readable Dwarf instance to a writable one. For this, we use a ConvertAddress() closure - which converts the gimli::read::Addresses to gimli::write::Addresses.

And then we parse the "=Expressions" line from STDIN/buffer and start reading the predicates of the form LHS=RHS one by one, where RHS is an expression written in postfix form and LHS is a source variable.

We use a loop to iterate over each such predicate. For each predicate, STDIN will have a corresponding address range over which the predicate is valid.

For each loop iteration, we perform following steps:-

- We first get the first compilation unit from the Dwarf struct instance. Then, we call GetFuncId() to get an id for the function. The id is of type gimli::write::UnitEntryId.
- We calculate the start and end address for the predicate, using given zeroaddress and the address range.
- Then, we verify that our VarMap indeed has an entry for the variable name mentioned in LHS of the predicate.
- Next, we call a function - GetVar() - to get the UnitEntryId for the variable inside the function denoted by UnitEntryId found earlier.
- Then, we query the VarMap, to get the location info and the DWARF expression for the variable.
- Now, we call CreateDwarfExpr() function, with the postfix expression from STDIN as an argument. It creates and returns an instance of gimli::write::Expression, which denotes the DWARF expression corresponding the postfix expression string given as an input.
- Now, we try to insert the DWARF expression into the existing set of expressions for the variable. For this, based on the presence of earlier expressions, we split into two or three.
- Finally, we create a new location list with all these expressions and insert it into the output object - in the .debug_loc section.

- We also change the variable DIE attribute - DW_AT_location - to point to the new location list thus created. If the DIE attribute DW_AT_location was not present earlier and the DIE had DW_AT_const attribute earlier, we remove the DW_AT_const before adding the DW_AT_location attribute.

This is the end of our for loop iteration/body.

After this, we write all the sections data from the gimli::write::Dwarf struct to a new sections struct - gimli::write::Sections and add these sections into the output object.

3.7 Evaluation

We run our tool on a set of functions in TSVC benchmark and get the updated object files for each of these functions.

The evaluator uses the original and updated object files to generate the result tables for compilers Clang, GCC and ICC.

We describe the results and the evaluator used to generate these results in the following sections.

3.7.1 Evaluator

We have created an evaluator using a library - 'gimli' in Rust.
The evaluator takes the original and updated object files as input for each TSVC function, analyzes the two object files, compares them and then generates the results table (shown in the following section).

The evaluator starts by first traversing through the DIEs(DWARF Debugging Information Entries) to find the corresponding function's DIE and its starting offset, for each object file. Next, we start a Depth First Search on the DIE graph starting from the function's DIE. While scanning each DIE, we note its static scope using DIE attributes 'low_pc' and 'high_pc'. Whenever we encounter a DIE of type 'variable' or 'formal parameter', we look at its 'location' attribute, and check its DWARF expression. If the DWARF expression has a 'constant signed / unsigned' value, we classify it as a constant expression and all other expressions as non-constant.

For each variable, we maintain a map from its name to a set of tuples of the form "(range_begin, range_end, is_const)", which denotes the range of PCs where debug info for the variable is present in the object file and whether the DWARF expression over the range is constant or non-constant.

We also maintain an instructions map, which represents the count of variables

accessible at a particular PC. For this, we read the '.text' section in each object file and perform instructions disassembly using the capstone library in Rust.

Upon getting the maps for both the object files (original and updated) and the instructions map, we analyze and compare them to calculate the number of PCs where debugging information is improved (was a constant earlier and became non-constant later, after updation), number of PCs where the debugging information was absent earlier and which was added during the debug headers updation and a cumulative count of PCs before and after updation.

In the next section, we describe the results generated by the evaluator.

3.7.2 Results

3.7.2.1 Metrics Description

- **Improved PCs** : This metric denotes the number of PCs where debug information has been improved, where "Improved" is defined as – it was a constant before and we've updated it to point to some register.
  Note that each PC is counted only once here.

- **Improved Variable Count** : It denotes the count of variables, whose debug information has been improved (according to the definition of improvement specified above).

- **Missing PCs** : It denotes the number of PCs where no debug information was present earlier and we've added new debug information there. (Note: Each PC is counted only once.)

- **Missing Variable Count** : It denotes the count of variables, whose debug information was absent earlier and new debug information for them has been added.

- **Total PCs** : This denotes the total number of PCs present inside the given function.

- **Before Cumulative Count** : This represents the cumulative count of PCs where debug information was present earlier, i.e. before debug headers updation.

- **After Cumulative Count** : This represents the cumulative count of PCs where debug information is present, after the debug headers updation.

The tables table 3.2, table 3.3 and table 3.4 display the results for TSVC benchmarks. We also tested our tool on larger examples as given in fig. 3.3(example1) and

Table 3.2: TSVC benchmarks results for Clang.

Function Name	Improved PCs	Improved Variable Count	Missing PCs	Missing Variable Count	Total PCs	Before Cumulative Count	After Cumulative Count
s000	14	1	-	-	21	19	19
s1112	22	1	-	-	29	27	27
s1119	8	1	-	-	20	20	20
s112	-	-	10	1	15	0	10
s116	1	1	15	1	19	1	16
s119	12	1	-	-	40	60	60
s121	17	1	-	-	42	62	62
s1221	12	1	-	-	17	12	12
s1251	17	1	-	-	22	17	17
s131	17	1	-	-	39	76	76
s132	21	1	-	-	55	220	220
s1351	14	1	15	3	19	14	59
s162	-	-	35	1	55	57	92
s173	14	1	-	-	19	34	34
s2244	15	1	-	-	49	46	46
s243	16	1	-	-	53	50	50
s251	12	1	-	-	17	12	12
s252	14	1	-	-	19	32	32
s319	27	1	1	1	33	55	56
s351	-	-	19	1	27	28	47
s352	-	-	28	2	36	33	61
s452	26	1	-	-	33	31	31
s453	14	1	-	-	21	19	19
vdotr	20	1	1	1	26	41	42
vpvpv	18	1	-	-	23	18	18
vpvts	18	1	-	-	25	23	23
vpvtv	18	1	-	-	23	18	18
vtv	14	1	-	-	19	14	14
vtvtv	18	1	-	-	23	18	18

Table 3.3: TSVC benchmark results for GCC.

Function Name	Improved PCs	Improved Variable Count	Missing PCs	Missing Variable Count	Total PCs	Before Cumulative Count	After Cumulative Count
s000	1	1	7	1	12	3	10
s111	1	1	13	1	29	15	28
s1111	-	-	18	1	21	1	19
s1112	1	1	8	1	13	3	11
s112	1	1	10	1	23	12	22
s113	1	1	8	1	21	13	21
s119	1	1	30	2	31	23	61
s121	1	1	6	1	19	23	29
s1221	-	-	6	1	10	2	8
s122	6	1	11	2	18	73	84
s1251	-	-	11	1	14	1	12
s127	-	-	15	2	18	2	32
s1281	1	1	16	1	19	2	18
s128	1	2	18	2	21	4	40
s131	1	1	6	1	16	25	31
s132	4	1	8	1	29	108	116
s1351	-	-	7	4	10	1	29
s162	1	1	8	1	46	66	74
s173	-	-	7	1	10	11	18
s174	2	1	8	1	68	112	120
s2233	2	1	34	1	41	24	75
s2244	1	1	9	1	25	15	24
s243	-	-	-	-	23	21	21
s311	1	1	7	1	16	4	11
s319	1	1	15	1	24	4	19
s3251	3	1	12	1	46	32	44
s423	5	1	6	1	35	62	68
s452	1	1	10	1	15	3	13
s453	1	1	7	1	12	3	10
sum1d	1	1	13	1	16	4	17
va	-	-	6	1	9	1	7
vdotr	1	1	10	1	19	4	14
vpv	-	-	7	1	10	1	8
vpvpv	-	-	8	1	11	1	9
vpvts	1	1	8	1	13	3	11
vpvtv	-	-	8	1	11	1	9
vtv	-	-	7	1	10	1	8
vtvtv	-	-	8	1	11	1	9

Table 3.4: TSVC benchmark results for ICC.

Function Name	Improved PCs	Improved Variable Count	Missing PCs	Missing Variable Count	Total PCs	Before Cumulative Count	After Cumulative Count
s000	-	-	-	-	25	22	22
s111	-	-	-	-	19	15	15
s112	-	-	-	-	22	18	18
s114	-	-	53	2	60	0	99
s119	-	-	-	-	29	50	50
s122	-	-	-	-	25	96	96
s124	-	-	44	1	52	49	93
s125	-	-	36	1	38	64	100
s127	-	-	57	1	65	62	119
s1279	-	-	-	-	51	48	48
s1281	-	-	-	-	67	64	64
s132	-	-	-	-	41	39	39
s1421	-	-	-	-	42	62	82
s173	-	-	-	-	28	25	25
s2244	-	-	-	-	67	63	63
s252	-	-	9	1	21	17	26
s254	-	-	-	-	16	13	13
s2711	-	-	-	-	47	44	44
s274	-	-	-	-	46	43	43
s293	-	-	-	-	18	15	15
s311	-	-	-	-	23	42	42
s317	-	-	-	-	15	25	25
s319	-	-	-	-	38	73	73
s4115	-	-	-	-	31	86	86
s441	-	-	-	-	54	51	51
s452	-	-	-	-	69	66	66
s453	-	-	-	-	53	50	50
sum1d	-	-	-	-	26	22	22
va	-	-	-	-	14	11	11
vdotr	-	-	-	-	65	127	127
vif	-	-	-	-	29	26	26
vpv	-	-	-	-	28	25	25
vpvpv	-	-	-	-	35	32	32
vpvts	-	-	-	-	61	58	58
vpvtv	-	-	-	-	65	62	62

```c
#include "globals.h"

extern void dummy_function();

int example1(int arg1)
{
  int l = 10 * arg1;
  for (int i = 0; i < lll; i++) {
    X[i] = Y[i] + val;
  }
  dummy_function();
  for (int j = 0; j < lll; j++) {
    Y[j] = X[j] - val + l;
  }
  dummy_function();
  for (int k = 0; k < lll; k++) {
    X[k] = X[k] + val - l;
    ++l;
  }
  return 0;
}
```

Figure 3.3: Example 1 source code

Table 3.5: Results for Example 1.

Function Name	Improved PCs	Improved Variable Count	Missing PCs	Missing Variable Count	Total PCs	Before Cumulative Count	After Cumulative Count
Example1	7	2	27	3	52	94	121

fig. 3.5(example2). The intermediate postfix expressions and metrics for example 1 are given in fig. 3.4 and table 3.5 respectively. Similarly, the intermediate postfix expressions and metrics for example 2 are given in fig. 3.6 and table 3.6 respectively.

3.7.2.2 Comparison between compilers

From the tables table 3.2, table 3.3 and table 3.4, we can see that most debug information improvements/additions are done for Clang, GCC mostly has only debug information additions, while only a few functions in ICC have any improvements/additions.

Our tool is observed to perform best for the Clang compiler, whereas we didn't see much improvements/additions for ICC compiler, as even with optimizations, ICC was able to retain a considerable amount of debug information.

Apart from these, we see that our tool performs considerably well for GCC compiler, where most of the updates in debug information were in the form of new additions rather than improving on the existing debug information.

```
=ZeroAddress
0x0
=TotalPCs
49
=Function
example1
=Expressions
arg1=1 -1 %ebx * 0 + -10 / * 0 +  0x20->0x58
arg1=1 -1 %ebx * 0 + -10 / * 0 +  0x6e->0xb0
arg1=1 bvextract(%xmm6, 31, 0) * 0 + 10 /   0xb0->0xda
i=4 -1 %eax * 0 + -16 / * 0 +   0x1a->0x2b
i=4 -1 %eax 16 - * 0 + -16 / * 0 +  0x2b->0x42
j=4 -1 %eax * 0 + -16 / * 0 +   0x50->0x63
j=4 -1 %eax 16 - * 0 + -16 / * 0 +  0x63->0x7a
k=4 -1 %eax * -905315344 + -16 / * 0 +  0x83->0xbf
k=4 -1 %eax 16 - * -905315344 + -16 / * 0 +   0xbf->0xd9
l=1 bvextract(%xmm6, 31, 0) * 4 -1 %eax * -905315344 + -16 / * + 0 +  0x83->0xbf
l=1 bvextract(%xmm6, 31, 0) * 4 -1 %eax 16 - * -905315344 + -16 / * + 0 +   0xbf->0xd9
```

Figure 3.4: Expressions generated for Example 1

```c
#include "globals.h"

extern void dummy_function();

int example2(int arg1, int arg2)
{
  int l = 10 * arg1;
  int m = 20 * arg2;
  for (int i = 0; i < LEN; i++) {
    a[i] = a[i] * b[i] * c[i] + val;
  }
  dummy_function();
  for (int j = 0; j < lll; j++) {
    c[j] = b[j] - m;
    m += 2;
  }
  dummy_function();
  for (int k = 0; k < lll; k++) {
    b[k] = b[k] + val - l;
    --l;
  }
  return 0;
}
```

Figure 3.5: Example 2 source code

Table 3.6: Results for Example 2.

Function Name	Improved PCs	Improved Variable Count	Missing PCs	Missing Variable Count	Total PCs	Before Cumulative Count	After Cumulative Count
Example2	-	-	24	2	62	188	212

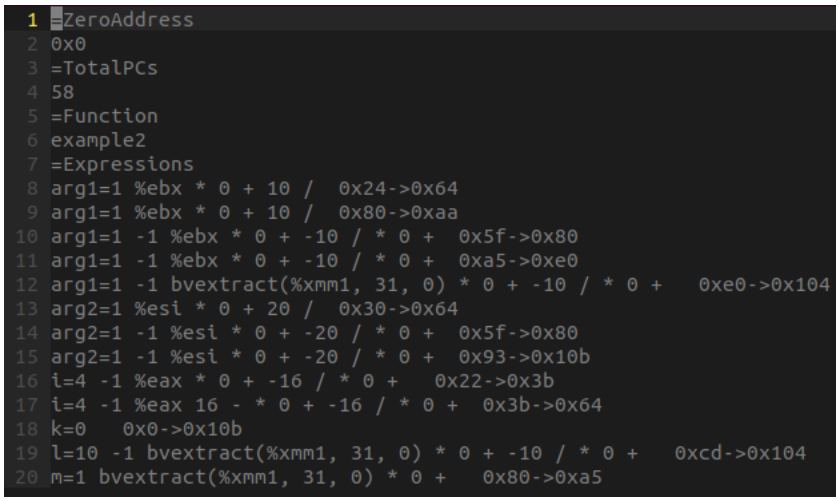

Figure 3.6: Expressions generated for Example 2

3.7.3 Ablation Studies

To show the contribution of individual components of our tool, we show and then discuss about the TSVC benchmark results for each of them.

The TSVC benchmark results for

- Clang, GCC and ICC compiler **without the Forward and Backward DFA** (*clang_no_dfa, gcc_no_dfa, icc_no_dfa*) are shown in table 3.7, table 3.10 and table 3.13 respectively.
- Clang, GCC and ICC compiler **with Forward DFA but without the reversible computation handling**(*clang_basic_dfa, gcc_basic_dfa, icc_basic_dfa*) are shown in table 3.8, table 3.11 and table 3.14 respectively.
- Clang, GCC and ICC compiler **with Forward DFA and the reversible computation handling**(*clang_reversible, gcc_reversible, icc_reversible*) are shown in table 3.9, table 3.12 and table 3.15 respectively.
- Clang, GCC and ICC compiler **with Forward DFA, the reversible computation handling as well as Backward DFA** (*clang_upto_backward, gcc_upto_backward, icc_upto_backward*) are shown in table 3.2, table 3.3 and table 3.4 respectively.

For *clang_no_dfa, gcc_no_dfa and icc_no_dfa*, we can see that the count of PCs inside the Improved or Missing column is only 1 or 2. This is because the equivalence checker typically gives predicates only at certain points in a program e.g. at the loop heads. We can observe that the Improved PCs are more dominant for clang, whereas gcc and icc have only Missing PCs.

Once we add the Forward DFA, the count of PCs inside Improved and Missing columns for is increased significantly, as the DFA is able to flow the information(set of predicates) through the program over multiple PCs. But for gcc, in many functions, the improvement in Missing PCs column seems to be smaller. This can be attributed to the fact that optimized programs produced by gcc are observed to be having the iterator variable increment/decrement instruction much earlier in a loop body. And as we are killing predicates when it contains any modified register at a program point, the range of PCs will be smaller.

After we handle the reversible computation inside the Forward DFA, we can see that the results for clang improved by a small margin, while there is improvement by larger margin for gcc. This would be due to the same reason described earlier – gcc producing optimized programs with loops having their iterator variable updation

instruction much closer to the loop head. But, because we can now flow our information(set of predicates) through such instructions, it increased the PC range. Still, for some functions, it is not increased significantly. This can be attributed to the presence of some instructions such as "mov" in the loop body, whose computations are not reversible.

After adding the Backward DFA with same features as the Forward DFA, we can see improvement in the metrics for some functions in clang, gcc and icc. The improvements are smaller as the PCs after the loop head would already be covered by the Forward DFA earlier and the scope of improvement would be at the program points before the loop head which is usually smaller.

3.7.4 Limitations

Currently, our tool supports modifications only in the '.debug_info' and '.debug_loc' section of an object file. We are not able to update any other sections such as '.debug_line'.

Our tool strongly relies on the Equivalence Checker which generates a proof of equivalence between an un-optimized and optimized versions of a program. Thus the debugging information modification is possible only for those programs for whom a proof of equivalence can be established.

Table 3.7: Results for clang without Forward and Backward DFA.

Function Name	Improved PCs	Improved Variable Count	Missing PCs	Missing Variable Count	Total PCs	Before Cumulative Count	After Cumulative Count
s000	1	1	-	-	21	19	19
s1112	1	1	-	-	29	27	27
s1119	1	1	-	-	20	20	20
s112	-	-	1	1	15	0	1
s116	-	-	1	1	19	1	2
s119	1	1	-	-	40	60	60
s121	1	1	-	-	42	62	62
s1221	1	1	-	-	17	12	12
s1251	1	1	-	-	22	17	17
s131	1	1	-	-	39	76	76
s132	1	1	-	-	55	220	220
s1351	1	1	1	3	19	14	17
s162	-	-	2	1	55	57	59
s173	1	1	-	-	19	34	34
s2244	1	1	-	-	49	46	46
s243	1	1	-	-	53	50	50
s251	1	1	-	-	17	12	12
s252	1	1	-	-	19	32	32
s319	1	1	1	1	33	55	56
s351	-	-	1	1	27	28	29
s352	-	-	1	1	36	33	34
s452	1	1	-	-	33	31	31
s453	1	1	-	-	21	19	19
vdotr	1	1	1	1	26	41	42
vpvpv	1	1	-	-	23	18	18
vpvts	1	1	-	-	25	23	23
vpvtv	1	1	-	-	23	18	18
vtv	1	1	-	-	19	14	14
vtvtv	1	1	-	-	23	18	18

Table 3.8: Results for clang with Forward DFA without handling reversible computation.

Function Name	Improved PCs	Improved Variable Count	Missing PCs	Missing Variable Count	Total PCs	Before Cumulative Count	After Cumulative Count
s000	13	1	-	-	21	19	19
s1112	21	1	-	-	29	27	27
s1119	7	1	-	-	20	20	20
s112	-	-	9	1	15	0	9
s116	-	-	11	1	19	1	12
s119	11	1	-	-	40	60	60
s121	16	1	-	-	42	62	62
s1221	11	1	-	-	17	12	12
s1251	15	1	-	-	22	17	17
s131	16	1	-	-	39	76	76
s132	20	1	-	-	55	220	220
s1351	13	1	13	3	19	14	53
s162	-	-	23	1	55	57	80
s173	13	1	-	-	19	34	34
s2244	13	1	-	-	49	46	46
s243	13	1	-	-	53	50	50
s251	11	1	-	-	17	12	12
s252	13	1	-	-	19	32	32
s319	19	1	1	1	33	55	56
s351	-	-	17	1	27	28	45
s352	-	-	25	1	36	33	58
s452	25	1	-	-	33	31	31
s453	13	1	-	-	21	19	19
vdotr	13	1	1	1	26	41	42
vpvpv	17	1	-	-	23	18	18
vpvts	17	1	-	-	25	23	23
vpvtv	17	1	-	-	23	18	18
vtv	13	1	-	-	19	14	14
vtvtv	17	1	-	-	23	18	18

Table 3.9: Results for clang with Forward DFA and reversible computation.

Function Name	Improved PCs	Improved Variable Count	Missing PCs	Missing Variable Count	Total PCs	Before Cumulative Count	After Cumulative Count
s000	14	1	-	-	21	19	19
s1112	22	1	-	-	29	27	27
s1119	8	1	-	-	20	20	20
s112	-	-	10	1	15	0	10
s116	-	-	13	1	19	1	14
s119	12	1	-	-	40	60	60
s121	17	1	-	-	42	62	62
s1221	12	1	-	-	17	12	12
s1251	17	1	-	-	22	17	17
s131	17	1	-	-	39	76	76
s132	21	1	-	-	55	220	220
s1351	14	1	15	3	19	14	59
s162	-	-	25	1	55	57	82
s173	14	1	-	-	19	34	34
s2244	15	1	-	-	49	46	46
s243	16	1	-	-	53	50	50
s251	12	1	-	-	17	12	12
s252	14	1	-	-	19	32	32
s319	26	1	1	1	33	55	56
s351	-	-	19	1	27	28	47
s352	-	-	27	1	36	33	60
s452	26	1	-	-	33	31	31
s453	14	1	-	-	21	19	19
vdotr	19	1	1	1	26	41	42
vpvpv	18	1	-	-	23	18	18
vpvts	18	1	-	-	25	23	23
vpvtv	18	1	-	-	23	18	18
vtv	14	1	-	-	19	14	14
vtvtv	18	1	-	-	23	18	18

Table 3.10: Results for gcc without Forward and Backward DFA.

Function Name	Improved PCs	Improved Variable Count	Missing PCs	Missing Variable Count	Total PCs	Before Cumulative Count	After Cumulative Count
s000	-	-	1	1	12	3	4
s1111	-	-	1	1	21	1	2
s111	-	-	1	1	29	15	16
s1112	-	-	1	1	13	3	4
s112	-	-	1	1	23	12	13
s113	-	-	1	1	21	13	14
s119	-	-	2	2	31	23	26
s121	-	-	1	1	19	23	24
s1221	-	-	1	1	10	2	3
s122	-	-	1	1	18	73	74
s1251	-	-	1	1	14	1	2
s127	-	-	1	2	18	2	4
s1281	-	-	1	1	19	2	3
s128	-	-	1	2	21	4	6
s131	-	-	1	1	16	25	26
s132	-	-	1	1	29	108	109
s1351	-	-	1	4	10	1	5
s162	-	-	1	1	46	66	67
s173	-	-	1	1	10	11	12
s174	-	-	1	1	68	112	113
s2233	-	-	3	1	41	24	28
s2244	-	-	1	1	25	15	16
s243	-	-	-	-	23	21	21
s311	-	-	1	1	16	4	5
s319	-	-	1	1	24	4	5
s3251	-	-	1	1	46	32	33
s423	-	-	1	1	35	62	63
s452	-	-	1	1	15	3	4
s453	-	-	1	1	12	3	4
sum1d	-	-	1	1	16	4	5
va	-	-	1	1	9	1	2
vdotr	-	-	1	1	19	4	5
vpv	-	-	1	1	10	1	2
vpvpv	-	-	1	1	11	1	2
vpvts	-	-	1	1	13	3	4
vpvtv	-	-	1	1	11	1	2
vtv	-	-	1	1	10	1	2
vtvtv	-	-	1	1	11	1	2

Table 3.11: Results for gcc with Forward DFA without handling reversible computation.

Function Name	Improved PCs	Improved Variable Count	Missing PCs	Missing Variable Count	Total PCs	Before Cumulative Count	After Cumulative Count
s000	-	-	2	1	12	3	5
s1111	-	-	15	1	21	1	16
s1112	-	-	3	1	13	3	6
s111	-	-	3	1	29	15	18
s112	-	-	3	1	23	12	15
s113	-	-	2	1	21	13	15
s119	-	-	25	2	31	23	54
s121	-	-	3	1	19	23	26
s1221	-	-	2	1	10	2	4
s122	-	-	4	1	18	73	77
s1251	-	-	3	1	14	1	4
s127	-	-	12	2	18	2	26
s1281	-	-	3	1	19	2	5
s128	-	-	3	2	21	4	10
s131	-	-	3	1	16	25	28
s132	-	-	2	1	29	108	110
s1351	-	-	3	4	10	1	13
s162	-	-	3	1	46	66	69
s173	-	-	3	1	10	11	14
s174	-	-	4	1	68	112	116
s2233	-	-	23	1	41	24	67
s2244	-	-	3	1	25	15	18
s243	-	-	-	-	23	21	21
s311	-	-	2	1	16	4	6
s319	-	-	3	1	24	4	7
s3251	-	-	3	1	46	32	35
s423	2	1	-	-	35	62	37
s452	-	-	3	1	15	3	6
s453	-	-	2	1	12	3	5
sum1d	-	-	2	1	16	4	6
va	-	-	2	1	9	1	3
vdotr	-	-	2	1	19	4	6
vpv	-	-	3	1	10	1	4
vpvpv	-	-	3	1	11	1	4
vpvts	-	-	2	1	13	3	5
vpvtv	-	-	2	1	11	1	3
vtv	-	-	2	1	10	1	3
vtvtv	-	-	2	1	11	1	3

Table 3.12: Results for gcc with Forward DFA and reversible computation.

Function Name	Improved PCs	Improved Variable Count	Missing PCs	Missing Variable Count	Total PCs	Before Cumulative Count	After Cumulative Count
s000	-	-	6	1	12	3	9
s111	1	1	13	1	29	15	28
s1111	-	-	18	1	21	1	19
s1112	-	-	7	1	13	3	10
s112	1	1	10	1	23	12	22
s113	-	-	7	1	21	13	20
s119	1	1	25	2	31	23	56
s121	1	1	6	1	19	23	29
s122	1	1	9	1	18	73	82
s1221	-	-	6	1	10	2	8
s1251	-	-	11	1	14	1	12
s127	-	-	15	2	18	2	32
s1281	-	-	14	1	19	2	16
s128	1	2	17	2	21	4	38
s131	1	1	6	1	16	25	31
s132	1	1	7	1	29	108	115
s1351	-	-	7	4	10	1	29
s162	1	1	8	1	46	66	74
s173	-	-	7	1	10	11	18
s174	-	-	7	1	68	112	119
s2233	-	-	34	1	41	24	80
s2244	1	1	9	1	25	15	24
s243	-	-	-	-	23	21	21
s311	-	-	6	1	16	4	10
s319	-	-	13	1	24	4	17
s3251	3	1	12	1	46	32	44
s423	5	1	6	1	35	62	68
s452	-	-	9	1	15	3	12
s453	-	-	6	1	12	3	9
sum1d	-	-	12	1	16	4	16
va	-	-	6	1	9	1	7
vdotr	-	-	8	1	19	4	12
vpv	-	-	7	1	10	1	8
vpvpv	-	-	8	1	11	1	9
vpvts	-	-	7	1	13	3	10
vpvtv	-	-	8	1	11	1	9
vtv	-	-	7	1	10	1	8
vtvtv	-	-	8	1	11	1	9

Table 3.13: Results for icc without Forward and Backward DFA.

Function Name	Improved PCs	Improved Variable Count	Missing PCs	Missing Variable Count	Total PCs	Before Cumulative Count	After Cumulative Count
s000	-	-	-	-	25	22	22
s111	-	-	-	-	19	15	15
s112	-	-	-	-	22	18	18
s114	-	-	2	2	60	0	4
s119	-	-	-	-	29	50	50
s122	-	-	-	-	25	96	96
s124	-	-	1	1	52	49	50
s125	-	-	2	1	38	64	66
s127	-	-	1	1	65	62	63
s1279	-	-	-	-	51	48	48
s1281	-	-	-	-	67	64	64
s132	-	-	-	-	41	39	39
s1421	-	-	-	-	42	62	63
s173	-	-	-	-	28	25	25
s2244	-	-	-	-	67	63	63
s252	-	-	1	1	21	17	18
s254	-	-	-	-	16	13	13
s2711	-	-	-	-	47	44	44
s274	-	-	-	-	46	43	43
s293	-	-	-	-	18	15	15
s311	-	-	-	-	23	42	42
s317	-	-	-	-	15	25	25
s319	-	-	-	-	38	73	73
s4115	-	-	-	-	31	86	86
s441	-	-	-	-	54	51	51
s452	-	-	-	-	69	66	66
s453	-	-	-	-	53	50	50
sum1d	-	-	-	-	26	22	22
va	-	-	-	-	14	11	11
vdotr	-	-	-	-	65	127	127
vif	-	-	-	-	29	26	26
vpv	-	-	-	-	28	25	25
vpvpv	-	-	-	-	35	32	32
vpvts	-	-	-	-	61	58	58
vpvtv	-	-	-	-	65	62	62

Table 3.14: Results for icc with Forward DFA without handling reversible computation.

Function Name	Improved PCs	Improved Variable Count	Missing PCs	Missing Variable Count	Total PCs	Before Cumulative Count	After Cumulative Count
s000	-	-	-	-	25	22	22
s111	-	-	-	-	19	15	15
s112	-	-	-	-	22	18	18
s114	-	-	53	2	60	0	90
s119	-	-	-	-	29	50	50
s122	-	-	-	-	25	96	96
s124	-	-	39	1	52	49	88
s125	-	-	20	1	38	64	84
s127	-	-	53	1	65	62	115
s1279	-	-	-	-	51	48	48
s1281	-	-	-	-	67	64	64
s132	-	-	-	-	41	39	39
s1421	-	-	-	-	42	62	82
s173	-	-	-	-	28	25	25
s2244	-	-	-	-	67	63	63
s252	-	-	3	1	21	17	20
s254	-	-	-	-	16	13	13
s2711	-	-	-	-	47	44	44
s274	-	-	-	-	46	43	43
s293	-	-	-	-	18	15	15
s311	-	-	-	-	23	42	42
s317	-	-	-	-	15	25	25
s319	-	-	-	-	38	73	73
s4115	-	-	-	-	31	86	86
s441	-	-	-	-	54	51	51
s452	-	-	-	-	69	66	66
s453	-	-	-	-	53	50	50
sum1d	-	-	-	-	26	22	22
va	-	-	-	-	14	11	11
vdotr	-	-	-	-	65	127	127
vif	-	-	-	-	29	26	26
vpv	-	-	-	-	28	25	25
vpvpv	-	-	-	-	35	32	32
vpvts	-	-	-	-	61	58	58
vpvtv	-	-	-	-	65	62	62

Table 3.15: Results for icc with Forward DFA and reversible computation.

Function Name	Improved PCs	Improved Variable Count	Missing PCs	Missing Variable Count	Total PCs	Before Cumulative Count	After Cumulative Count
s000	-	-	-	-	25	22	22
s111	-	-	-	-	19	15	15
s112	-	-	-	-	22	18	18
s114	-	-	53	2	60	0	89
s119	-	-	-	-	29	50	50
s122	-	-	-	-	25	96	96
s124	-	-	42	1	52	49	91
s125	-	-	29	1	38	64	93
s127	-	-	56	1	65	62	118
s1279	-	-	-	-	51	48	48
s1281	-	-	-	-	67	64	64
s132	-	-	-	-	41	39	39
s1421	-	-	-	-	42	62	95
s173	-	-	-	-	28	25	25
s2244	-	-	-	-	67	63	63
s252	-	-	3	1	21	17	20
s254	-	-	-	-	16	13	13
s2711	-	-	-	-	47	44	44
s274	-	-	-	-	46	43	43
s293	-	-	-	-	18	15	15
s311	-	-	-	-	23	42	42
s317	-	-	-	-	15	25	25
s319	-	-	-	-	38	73	73
s4115	-	-	-	-	31	86	86
s441	-	-	-	-	54	51	51
s452	-	-	-	-	69	66	66
s453	-	-	-	-	53	50	50
sum1d	-	-	-	-	26	22	22
va	-	-	-	-	14	11	11
vdotr	-	-	-	-	65	127	127
vif	-	-	-	-	29	26	26
vpv	-	-	-	-	28	25	25
vpvpv	-	-	-	-	35	32	32
vpvts	-	-	-	-	61	58	58
vpvtv	-	-	-	-	65	62	62

3.8 Related work

Debugging optimized code is a problem for which many approaches have been suggested over the years, including limiting compiler optimizations, restricting the debugger functionality, using recompilation or dynamic de-optimization to undo the optimizations, having the debugger determine the effect of optimizations and mask them from the user, changing the source program to reflect the effects of compiler optimizations and providing new compiler-debugger interfaces.

Some of the approaches make it transparent to the user while others prefer non-transparent method, where being non-transparent to the user means exposing the optimization effects. Some of them use their custom language/compiler and/or debugger. We now summarize different approaches and highlight the difference with our approach.

[11], [13], [15], [2], [3], [1], and [9] all use a specific custom compiler and modify it to gather extra information for debugging. [14], [1] and [4] provide techniques to *determine* the currentness of variables (a variable value is considered as non-current, if it is inconsistent with the source-level value expected at a breakpoint). [10] takes a very different approach, where it runs both unoptimized and optimized programs together, compares their behaviour and reports if any difference is found, while suggesting disabling the concerned optimization in a traditional debugger.

[8] gives algorithms to recover debugging information by modeling unoptimized and optimized programs as graphs and performing static analysis on them, although no implementation is provided that use these techniques. [9] provides a complete debugging system SELF and suggests that this could be used for other languages such as C, C++. [2] uses Data Flow Analysis and provides a new compiler-debugger interface in the form of flow graphs to support source-level debugging of optimized programs. [12] and [6] validate the already existing debugging information in optimized programs and demonstrate the bugs found in optimization passes of respective compilers.

Our approach supports the use of existing modern compilers GCC, LLVM/Clang and ICC and the debuggers such as GDB for debugging optimized versions of programs written in C. We do not assume any knowledge about the optimization passes or their order in these compilers. We present a novel approach of using a black-box equivalence checker [5] [7] to correlate the unoptimized and optimized versions of a source program and provide predicates for source variables in terms of registers in the optimized program, which are then converted into DWARF expressions and inserted into the optimized program to improve its debugging information. The updated optimized program can be directly run inside standard debuggers such as GDB.

Chapter 4

Conclusion

This thesis presents a technique of using an equivalence checker to improve debugging information present in optimized executable programs, which are difficult to debug due to the loss of debugging information while doing rigorous transformations by a compiler. We evaluated our tool across a set of benchmarks for three compilers - Clang, GCC and ICC and we could add/improve the debugging information in the corresponding optimized programs. The improvement varies across the three compilers, as it depends upon the kind of transformations done and the amount of debugging information the compiler was able to persist during compilation.

Bibliography

[1] Ali-Reza Adl-Tabatabai and Thomas Gross. "Source-level debugging of scalar optimized code". In: May 1996, pp. 33–43. ISBN: 0897917952. DOI: 10.1145/231379.231388.

[2] Lutz Berger and Roland Wismüller. "Source-Level Debugging of Optimized Programs Using Data Flow Analysis". In: Dec. 1992. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.2997.

[3] Gary Brooks, Gilbert J. Hansen, and Steve Simmons. "A new approach to debugging optimized code". In: July 1992, pp. 1–11. ISBN: 0897914759. DOI: 10.1145/143103.143108.

[4] Max Copperman. "Debugging optimized code without being misled". In: May 1994, pp. 387–427. DOI: 10.1145/177492.177517.

[5] Manjeet Dahiya and Sorav Bansal. "Black-Box Equivalence Checking Across Compiler Optimizations". In: Nov. 2017, pp. 127–147. ISBN: 978-3-319-71236-9. DOI: 10.1007/978-3-319-71237-6_7.

[6] Giuseppe Antonio Di Luna et al. "Who's debugging the debuggers? exposing debug information bugs in optimized binaries". In: Apr. 2021, pp. 1034–1045. ISBN: 9781450383172. DOI: 10.1145/3445814.3446695.

[7] Shubhani Gupta, Abhishek Rose, and Sorav Bansal. "Counterexample-Guided Correlation Algorithm for Translation Validation". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428289.

[8] John Hennessy. "Symbolic Debugging of Optimized Code". In: July 1982, pp. 323–344. DOI: 10.1145/357172.357173.

[9] Urs Hölzle, Craig Chambers, and David Ungar. "Debugging optimized code with dynamic deoptimization". In: July 1992, pp. 32–43. ISBN: 0897914759. DOI: 10.1145/143095.143114.

[10] Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. “Comparison Checking: An Approach to Avoid Debugging of Optimized Code”. In: Aug. 1999, pp. 268–284. ISBN: 978-3-540-48166-9. DOI: 0.1007/3-540-48166-4_17.

[11] Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. “FULLDOC: A Full Reporting Debugger for Optimized Code”. In: June 2000, pp. 240–259. ISBN: 3540676686. DOI: 10.5555/647169.718156.

[12] Yuanbo Li et al. “Debug information validation for optimized code”. In: June 2020, pp. 1052–1065. ISBN: 9781450376136. DOI: 10.1145/3385412.3386020.

[13] Caroline M. Tice. “Non-Transparent Debugging of Optimized Code”. In: Feb. 2000. URL: https://dl.acm.org/doi/book/10.5555/894940.

[14] Roland Wismüller. “Debugging of globally optimized programs using data flow analysis”. In: Aug. 1994, pp. 278–289. DOI: 10.1145/178243.178430.

[15] Polle T. Zellweger. “An interactive high-level debugger for control-flow optimized programs”. In: Aug. 1983, pp. 159–172. DOI: 10.1145/1006142.1006183.