Automatic Generation of Debug Headers through BlackBox Equivalence Checking

Vaibhav Kiran Kurhe, Pratik Karia, Shubhani Gupta, Abhishek Rose, Sorav Bansal
Indian Institute of Technology Delhi, India

***Abstract*—Modern compiler optimization pipelines are large and complex, and it is rather cumbersome and error-prone for compiler developers to preserve debugging information across optimization passes. An optimization can add, remove, or reorder code and variables, which makes it difficult to associate the generated code statements and values with the source code statements and values. Moreover recent proposals for automatic generation of optimizations (e.g., through search algorithms) have not previously considered the preservation of debugging information.**

We demonstrate the application of a *blackbox equivalence checker* to automatically populate the debugging information in the debug headers of the optimized executables compiled from C programs. A blackbox equivalence checker can automatically compute equivalence proofs between the original source code and the optimized executable code without the knowledge of the exact transformations performed by the compiler/optimizer. We present an algorithm that uses these formal equivalence proofs to improve the executable's debugging headers. We evaluate this approach on benchmarks derived from the Testsuite of Vectorizing Compilers (TSVC) compiled through three different compilers: GCC, Clang/LLVM, and ICC. We demonstrate significant improvements in the debuggability of the optimized executable code in these experiments. The benefits of these improvements can be transparently realized through any standard debugger, such as GDB, to debug the updated executable.

I. Introduction

Debugging is a crucial part of the software development lifecycle. There are many ways to debug a program, e.g., using print statements within the code, log analysis, rubber duck debugging, etc. One such important technique involves the use of a debugging tool, that allows a programmer to pause the program at any execution point and observe its state. Examples of debugging tools, also called debuggers, are GDB, LLDB, Visual Studio Debugger, etc.

While using a debugger, a programmer would typically like to query the program state using "source code names", i.e., source line numbers, function names, variable names, file names, etc. However, the state of the executable program (that is generated by the compiler) involves "machine names", i.e., registers, memory addresses, program counter value (which is also a memory address), etc. Thus, for a smooth debugging experience, it is important for the debugger to have access to the mapping from source code names to their corresponding machine names. For example, consider the C program shown in fig. 1a and its unoptimized LLVM IR and optimized assembly shown in figs. 1b and 1d. The relations between the unoptimized IR program (which is trivially mapped to the source program) and the assembly program variables at location `(I2,A4)`

```
#define N 32000
int X[N], Y[N], val;
C0: void s000( ) {
C1:   int i = 0;
C2:   for( ; i < N; i++)
C3:     X[i] = Y[i] + val;
EC: }
```

(a) C program

```
I0: s000:
I1:   i.0 = 0; br I2
I2:    i = phi[i.0,I1], [i.1,I9]
I3:    cmp = (i < N)
I4:    br cmp, I5, EI
I5:    t0 = Y + 4 * i
I6:    t1 = mem[t0] + val
I7:    t2 = X + 4 * i
I8:   mem[t2] = t1
I9:    i.1 = i + 1; br I2
EI:   ret
```

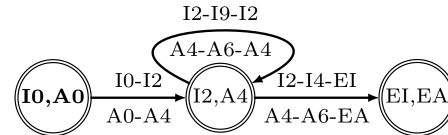
(b) (Abstracted) LLVM IR

Affine Invariants at (I2, A4)
$(X + 4 * i) = r1$
$(Y + 4 * i) = r2$
$val = r3$
$(X + 4 * N) = r4$

(c) Affine inductive invariants at node (I2, A4) for product program in fig. 1e

```
A0: s000:
A1:   r1 = &X[0]; r2 = &Y[0]
A2:   r3 = val
A3:   r4 = r1 + 4*N
A4:   mem[r1] = mem[r2] + r3
A5:    r1 += 4; r2 += 4
A6:    if(r1 != r4) goto A4
EA:   ret
```

(d) (Abstracted) Assembly program generated at -O2 optimization level



(e) Product program for IR and Assembly program shown in fig. 1b and fig. 1d respectively.

Fig. 1: "s000" example taken from TSVC suite

(shown in fig. 1c) map the source code variable '`i`' to the expression `(r1 - X`[1]`)/4`. Thus, if the programmer pauses the executable program at PC `A4`, and queries the value of the variable '`i`', the debugger should calculate the value of expression `(r1 - X)/4` using the value of register '`r1`' and display it as the value of '`i`' to the user.

Traditionally, the onus of generating these mappings has rested with the compiler. The compiler, during its operations of translating source code to machine code and optimizing it, populates the debugging headers (e.g., DWARF format) in the executable code file (e.g., ELF format). Roughly speaking, the debugging headers should contain (a) mappings from PC values in the executable code to line numbers in the source code; and (b) for each PC value, the mapping from the source code variable name to an expression (in terms of the machine state elements) that represents its value at that program point.

[1]Here, name of the array denotes its base address

```
(gdb) break s000:C3
Breakpoint 1 at 0x80483f6: s000:C3. (2 locations)
(gdb) run
Starting program: s000
Breakpoint 1, s000 () at s000:C3  X[i] = Y[i] + val;
(gdb) print i
$1 = 0
(gdb) continue
Continuing.
Breakpoint 1, s000 () at s000:C3  X[i] = Y[i] + val;
(gdb) print i
$2 = 0
```

Fig. 2: GDB based debugging session for executable in fig. 1d generated at optization level -O2

Long and complex optimization pipelines make it cumbersome and error-prone for the compiler developers to identify and update these mappings. The problem is exacerbated in the face of constant churn and updates to the compiler code — a typical production compiler has millions of lines of code being maintained by hundreds of developers, with a churn of hundreds of thousands of lines of code per annum. As a result, it is common for production compilers to generate incomplete or erroneous debugging headers. For example, fig. 2 shows a debugging session for the executable shown in fig. 1d generated at optimization level -O2. The execution is paused at the '`C3`' line number in the source file (fig. 1a) using a breakpoint and the value of variable '`i`' is printed. GDB reads the debugging headers present in the executable and prints `i` = 0 throughout the loop. Thus, maintaining the source names to machine names mappings by compilers during the long and complex optimization pipeline may seem achievable in theory, but we find that compilers typically generate buggy information in the debugging headers even at optimization level -O2 for small C programs.

Since compilers could produce relatively more accurate debugging information when the optimization levels are lower, optimization flags like `-Og` have been introduced that limit the optimization support so that the generated executable remains debuggable. While the `-Og` option is a reasonable compromise for the typical edit-test-debug development cycle, production code is usually compiled with aggressive optimizations. Thus in the absence of the required debugging information, debugging on-field failures of production code becomes challenging. Moreover, some bugs, such as those related to the undefined behavior semantics in the programming language, manifest only at higher optimization levels. Unavailability of the required debugging information at these higher optimization levels poses a major challenge for the developers while debugging.

In addition to the current compiler development models, researchers have also proposed automatic generation of optimizations, e.g., through superoptimization techniques [1]–[7]. In these settings, an algorithm is also required for the automatic generation (or updation) of debugging headers; we are not aware of any previous work that tackles this problem.

We present an algorithm and a tool that takes the original source code program and an existing equivalent executable program (generated by a compiler), and automatically generates and updates its debugging headers. Our updates ensure that the new executable is more debuggable (because the programmer would be able to observe more values during its runtime) than the original executable. Our updates do not change the execution behavior of the executable, e.g., the performance of the executable remains identical — we only update the mappings present in the debugging headers that are read by a debugger during a debugging session. Thus, our tool brings the debugging experience for an optimized executable closer to the debugging experience for an unoptimized executable.

Our tool is based on a *blackbox equivalence checker* which is a standalone tool that takes two input programs and determines whether they produce equivalent observable behavior. Because program equivalence is an undecidable problem, the blackbox equivalence checker is best-effort: if it is able to determine equivalence, we can be sure that the programs are equivalent; however, if it is unable to determine equivalence, that does not necessarily mean that the programs are inequivalent.

We apply the blackbox equivalence checker to identify whether the optimized executable code is equivalent to the original source code. If the tool is able to determine that the two programs are equivalent, it generates a formal mathematical proof of equivalence. The proof is represented as a bisimulation relation that involves (a) correlations between the program text segments across the two programs, and (b) relational invariants across the variables of the two programs at the intermediate correlated program points. We provide more background on blackbox equivalence checking and the resulting proof format in section II-A.

The primary contribution of this paper is an algorithm based on linear algebra techniques that converts the relational predicates in a bisimulation proof into a set of mappings between the source code names and the machine names. These mappings are added to the debugging headers of the executable file to improve its debuggability. The proposed algorithm also involves dataflow analyses on the executable code to improve the debugging information further. We show significant improvements in the debugging headers of executables generated by three different compilers: GCC, Clang/LLVM, and ICC. We demonstrate these improvements on programs taken from the Testsuite for Vectorizing Compilers (TSVC) [8].

II. Background

A. Blackbox Equivalence Checking

Determining if two Turing machines are equivalent is a classical problem in theoretical computer science, which is undecidable in general. However, best-effort solutions to this problem have been proposed for applications such as translation validation, regression verification, superoptimization, etc. In our work, we use a blackbox equivalence checker to generate an equivalence proof between the source code and the optimized executable code. Here we provide a brief overview of the blackbox equivalence checker's input/output behavior – we refer the reader to recent work in this area [9]–[11] for the internal details.

Our equivalence checker takes a C program and an assembly program as input. It converts the C program to an unoptimized intermediate representation (IR) program and tries to identify a proof of equivalence between the IR (referred to as I) and the assembly program (referred to as A). The equivalence proof is computed at function granularity, and so any inter-procedural transformations, such as function inlining, are not supported by the checker. If a proof is identified, it is output as a *product program*. A product program encodes the lockstep execution of I and A: at every step (of this lockstep execution), a segment of program I is executed concurrently with a *correlated* segment of program A. For example, Figure 1e shows the product program for the IR shown in Figure 1b and the assembly program shown in Figure 1d. One of the correlations shown in this product program is across the program segment `(I0-I2)` in I with segment `(A0-A4)` in A.

These correlations between the program segments are chosen carefully so that the relationships between the program variables can be easily expressed at the endpoints of each step (of the lockstep execution). For example, prior work has shown that affine, equality, and inequality relations between variables of programs I and A are usually sufficient for determining equivalence across an overwhelmingly large set of typical compiler transformations. Thus, the equivalence checker employs a data-driven invariant inference procedure for inferring affine, equality and inequality invariants at the endpoints of the correlated program segments (a step in the product program). For example, Figure 1c shows the affine invariants inferred at the correlated endpoint `(I2,A4)` for the product program in Figure 1e.

Finally, the proof of observational equivalence is attempted based on the inferred invariants. To summarize: the goal of the equivalence checker is to identify a product program that allows the inference of observational equivalence through invariant inference.

B. Debugging Headers

The debugging headers are produced by the compiler and consumed by the debugger tool (e.g., GDB). To make these independent software efforts interoperable, standard formats are used for encoding the debugging information in the headers. For the ELF executable file format, a popular debugging header format is DWARF. Because our tool is based on ELF/DWARF, we provide a brief introduction to the structure of DWARF headers.

A DWARF debugging header consists of multiple sections:

- `.debug_info`: This section lists the program's "nouns" as *DWARF Debugging Information Entities* (DIEs) where each DIE denotes an individual unit of a program, such as a variable, function, data type, etc. These DIEs represent the source names and are interlinked with each other using parent-child (e.g., for nested scopes) and sibling relationships (e.g., for variables listed in the same scope) to form a graph.
- `.debug_loc`: This section presents the value and address information about some of the DIEs listed in `.debug_info` through `location lists`. Each location in the location list represents a program PC (or a range of PCs) and a location list can thus represent different values/addresses for a DIE for each different location in a location list.
- `.debug_ranges`: This section lists the address ranges for some of the DIEs listed in `.debug_info`.

```
// .debug_info section ahead
<1><3f> (DW_TAG_base_type)
   <40>   DW_AT_byte_size     : 4
   <41>   DW_AT_encoding      : 5          (signed)
   <42>   DW_AT_name          : int
              .
              .
<1><6e4> (DW_TAG_subprogram)
   <6e5>   DW_AT_name         : s000
   <6e9>   DW_AT_decl_file    : 1
   <6ea>   DW_AT_decl_line    : 33
   <6eb>   DW_AT_decl_column  : 5
   <6ec>   DW_AT_type         : <0x3f>
   <6f0>   DW_AT_low_pc       : 0x0
   <6f4>   DW_AT_high_pc      : 0x3d
<2><6fa> (DW_TAG_lexical_block)
   <6fb>   DW_AT_low_pc       : 0xd
   <6ff>   DW_AT_high_pc      : 0x28
<3><703> (DW_TAG_variable)
   <704>   DW_AT_name         : i
   <706>   DW_AT_decl_file    : 1
   <707>   DW_AT_decl_line    : 39
   <708>   DW_AT_decl_column  : 12
   <709>   DW_AT_type         : <0x3f>
   <70d>   DW_AT_location     : 0x0 (location list)

// .debug_loc section ahead
Offset    Begin    End      Expression
00000000 0000000d 00000025 (DW_OP_lit0; DW_OP_stack_value)
0000000c <End of list>
```

Fig. 3: Example of the `.debug_info` and `.debug_loc` sections of the DWARF format

A value in the debugging header relates a DIE to an expression involving machine names, i.e., machine registers, memory addresses, memory values, etc. The DWARF standard specifies an expression language that includes standard arithmetic, logical, and memory operators. DWARF uses a postfix representation for these expressions.

We show an example of the `.debug_info` and `.debug_loc` sections in fig. 3. In this example, there are four DIEs in the `.debug_info` section: base_type, subprogram, lexical_block, and variable.

- A base_type DIE specifies the details about a type in a programming language e.g. a signed integer of size 4 bytes in C.
- The subprogram DIE is denoting a function named 's000', with its line number and column of its definition in the source code file. It also contains a reference to the type of the value it would return and a range of addresses for the function.
- The lexical block DIE defines a static scope delineated by the given range of addresses. Instead of a single contiguous range of addresses, a lexical block can also have a reference to a set of address ranges located in the `.debug_ranges` section.

- Each variable DIE denotes a variable, with its exact position of declaration, a reference to its type, and a reference to the location list in the `.debug_loc` section.

The starting number e.g. `<1>` in the `.debug_info` section denotes the level of the DIE – these levels are used to create a parent-child relationship among DIEs, e.g., the variable DIE is a child of the lexical_block DIE, which in turn is a child of the subprogram DIE, and so on.

A location list inside the `.debug_loc` section provides a DWARF expression denoting the value and/or location of a variable. For instance, the location list for variable '`i`' shown in fig. 3 provides a value denoted by postfix DWARF expression (DW_OP_lit0; DW_OP_stack_value) from location 0xd to 0x25. The postfix DWARF expression can be read as a sequence of DWARF operations operating on an evaluation stack: DW_OP_lit0 denotes a literal '0' (pushed on the top of the evaluation stack); DW_OP_stack_value terminates the expression by denoting that the current value at the top of stack is also the value of the full current DWARF expression. In this example, the value of '`i`' is `0` for the PC locations `0xd-0x25` in the program.

III. Problem Statement

Given the product program and the inferred affine invariants (as shown in fig. 1), we are interested in automatically populating the incomplete DWARF headers, which take the form shown in fig. 3. We need to solve multiple subproblems to meet this objective, and we discuss each of them in this section

A. Identifying Maps from Unoptimized IR PCs to Source Code Line/Column Numbers

As noted in section II-A, the equivalence checker identifies a bisimulation proof between the unoptimized IR program and the optimized assembly code. Thus, the generated proof relates IR PCs to assembly code PCs. Similarly, the inductive invariants relate IR registers to assembly registers and memory locations. In contrast, debugging headers must refer to source code line and column numbers as the programmer cannot understand IR PCs and registers.

The equivalence checker's IR is almost identical to the LLVM IR except that it does not include non-deterministic LLVM values like `undef` and `poison`. Thus we leverage LLVM's existing infrastructure to generate these correspondences between the IR PCs and registers and the source code line/column numbers and variable names. We represent this information as two maps: (1) a map P_{is} from IR PCs to the corresponding source line/column numbers and (2) a map R_{is} from IR registers to their source variable names or expressions. Figure 1b shows the unoptimized LLVM IR representation of the C program shown in fig. 1a and the corresponding P_{is} and R_{is} maps are shown in fig. 4.

To simplify the discussion, we will henceforth omit the discussion on IR PCs/registers because it is straightforward to translate IR PCs and registers to source code line/column numbers and variables. Instead, we will directly relate assembly PCs with source "PCs" (internally characterized by source code line and column numbers) and assembly registers/memory locations with source variable values.

IR PC	Source L/C
I0	C0/8
I1	C1/9
I3	C2/12
I4	C2/12
I6-I8	C3/7
I9	C2/19
EI	EC/2

(a) P_{is} map (L/C denotes the source line and column number)

IR register	Source variable
i.0	i
i.1	i
t0	Y + 4*i
t2	X + 4*i

(b) R_{is} map

Fig. 4: P_{is} and R_{is} maps for C progarm and IR shown in figs. 1a and 1b respectively

B. Identifying Maps from Assembly PCs to Source Code Line Numbers

The `.debug_line` section in DWARF headers encodes the mapping to the Source code line numbers from Assembly Instructions (PCs). From the perspective of a compiler developer, it is easier to maintain these mappings between the source code line numbers and assembly PCs even across complex transformations. Thus we have found these mappings to be near-precise in all our experiments with all the three compilers we tested, namely GCC, Clang/LLVM, and ICC. In this paper, we do not further improve upon these mappings.

C. Identifying Maps from Source Variables to Assembly Registers and Stack Slots at Every PC

The mappings from source variables to assembly locations form the bulk of the debugging header data. As discussed in section II-B, this information is specified through the variable DIEs (`DW_TAG_variable`) in the `.debug_info` section and they may further be associated with locations lists (specified in `.debug_loc`). Each entry in the location list contains a begin PC, an end PC, and a postfix expression which refers to the assembly registers and stack slots. The begin/end PCs are assembly PCs, and their corresponding source PCs can be derived from the other sections of the debug headers.

We find that *this part of the debugging information is often missing or inaccurate in the executables generated by modern optimizing compilers*. Thus, **we focus our attention on repairing this correlation of source variables with assembly locations at all the assembly PCs**.

IV. Algorithm

The problem of identifying a map from source variables to assembly locations for each assembly PC can be further broken down into two high-level subproblems, described next in sections IV-A and IV-B.

A. Identifying the Assembly Expression for Each Source Variable

At any pair of source and assembly PCs, the relational invariants that relate the variables across the source and

```
#define N 4096
int a[N], b[N];
int aa[N][N];
C0: void foo( ) {
C1:  int *idx = a;
C2:  for(int i=0; i<N; i++){
C3:   int sum = *idx;
C4:   for(int j=0; j<N; j++)
C5:    sum += aa[i][j]*b[i];
C6:   *idx++ = sum;
C7:  }
C8: }
```

(a) C Program

Freestyle Affine Expressions
$\mathtt{r1} - 4096 * \mathtt{i} - 4 * \mathtt{j} - \mathtt{aa} = 0$
$\mathtt{r2} - \mathtt{idx} - \mathtt{b} + \mathtt{a} = 0$
$4 * \mathtt{i} - \mathtt{idx} + \mathtt{a} = 0$

(b) Affine inductive invariants at one of the correlated nodes in the product program across unoptimized IR and optimized assembly for C program in fig. 5a

Fig. 5: Motivating Example for section IV-A

assembly programs can be arbitrary affine predicates of the form $\sum_i c_i * x_i + c_0 = 0$. Here x_is represent variables in both source and assembly programs and c_is represent constant coefficients. We restrict our attention to affine invariants because an overwhelmingly large category of compiler transformations can be modeled through affine relational invariants [9], [11], [12]. We call affine expressions of this form, *freestyle affine expressions*. Some examples of freestyle affine expressions (appearing as inductive invariants) can be seen in fig. 5b.

However, we are interested in representing source variable values as expressions over assembly variables, e.g., $s_i = \sum_j c_j * a_j + c_0$ where s_i represents a source variable, a_j represents an assembly register or memory location, and c_js represent constant coefficients. This form of an affine expression where a source variable is represented as a linear combination of assembly values is called a *source-to-assembly affine expression*. For example, the source-to-assembly affine expressions for the freestyle expressions in fig. 5b would include $(\mathtt{i} = (\mathtt{r2} - \mathtt{b})/4)$ and $(\mathtt{j} = (\mathtt{r1} - 1024 * (\mathtt{r2} - \mathtt{b}) - \mathtt{aa})/4)$.

Thus, we need to convert freestyle affine expressions to source-to-assembly affine expressions. The problem becomes slightly more complex because of the fact that these affine expressions are over bitvectors of a finite bitwidth (e.g., 32 or 64). Analysis of a system of affine equations over bitvectors have been studied in mathematical depth in prior work [13]. In our work, we model these affine expressions using integer arithmetic: even though models based on bitvector arithmetic would be theoretically more precise, in practice, compiler transformations that would expose this relative imprecision of integer arithmetic are rare.

Obtaining source-to-assembly affine expressions from freestyle affine expressions involves the application of standard linear algebra techniques on a carefully-chosen order of variables of the two programs, as described through the following algorithm steps.

- First, the freestyle affine expressions (at a given correlated PC pair) are represented using a matrix of the form: $AX = 0$, where X is a vector of $m+n+1$ variables (for m source variables, n assembly registers/memory locations and a constant 1), and A represents a coefficient matrix with $m+n+1$ columns formed by the c_is in the freestyle affine expressions. The last row in the vector X (constant 1) is meant to represent the constant coefficient c_0 in the freestyle affine expression. The number of rows in A is equal to the number of freestyle affine expressions (at the given pair of correlated PCs). The matrix representation of the freestyle affine equations in fig. 5b is:

$$\begin{bmatrix} 1 & -4096 & -4 & 0 & 0 & -aa \\ 0 & 0 & 0 & 1 & -1 & -b+a \\ 0 & 4 & 0 & 0 & -1 & a \end{bmatrix} \begin{bmatrix} \mathtt{r1} \\ \mathtt{i} \\ \mathtt{j} \\ \mathtt{r2} \\ \mathtt{idx} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

- The next step is to rearrange the rows of variable matrix X such that the first m rows in X represent source variables and the next n rows in X represent assembly registers and memory locations (and the last row in X represents the constant 1). The columns of the coefficient matrix A are accordingly ordered. This ordering of variables is an important component of our algorithm. In our running example of fig. 5b, the resulting matrix equations after variable reordering are:

$$\begin{bmatrix} -4096 & -4 & 0 & 1 & 0 & -aa \\ 0 & 0 & -1 & 0 & 1 & -b+a \\ 4 & 0 & -1 & 0 & 0 & a \end{bmatrix} \begin{bmatrix} \mathtt{i} \\ \mathtt{j} \\ \mathtt{idx} \\ \mathtt{r1} \\ \mathtt{r2} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

- Finally, standard linear algebra techniques are then used to convert the ordered matrix A into the canonical *row-reduced echelon form* (RREF) [14], A_{rref}. The RREF of a matrix is a unique representation with the following properties:
  a) the leading coefficient (pivot) of a non-zero row is 1,
  b) the column containing a leading 1 (pivot) has zeros in all its other entries,
  c) the pivot of a nonzero row is always strictly to the right of the pivot of the row above it,
  d) all rows consisting of only zeroes are at the bottom.
  The A_{rref} matrix for the example in fig. 5b is:

$$\begin{bmatrix} 1 & 0 & 0 & 1 & -1/4 & \mathtt{b}/4 \\ 0 & 1 & 0 & -1/4 & 256 & -256 * \mathtt{b} + \mathtt{aa}/4 \\ 0 & 0 & 1 & 0 & -1 & \mathtt{b} - \mathtt{a} \end{bmatrix}$$

Among all the non-zero rows in the RREF matrix A_{rref}, we pick those rows that have one of the source variables as the pivot element. Based on this, the resulting Source-to-assembly affine expressions for the example in fig. 5b are:

$$\mathtt{i} = (\mathtt{r2} - \mathtt{b})/4$$
$$\mathtt{j} = (\mathtt{r1} - \mathtt{aa} - 1024 * (\mathtt{r2} - \mathtt{b}))/4$$
$$idx = r2 + a - b$$

We claim that this set of equations represents a maximal set of linearly-independent source-to-assembly affine expressions — this property follows directly from the properties of the row-reduced echelon form. The proof for this claim involves showing that if there exists a source-to-assembly affine expression that is derivable from the original matrix A, then that affine expression would be always available as a row in the A_{rref} (when the columns of A have been carefully ordered such that the source variables appear before the assembly variables). We

TABLE I
FORWARD AND BACKWARD DFAS FOR SPREADING THE PREDICATES. WHEREVER THE TWO DFAS DIFFER, THE VALUES ABOVE THE HORIZONTAL LINE SHOW THE FORWARD DFA VALUES AND THE VALUES BELOW SHOW THE BACKWARD DFA VALUES.

Domain	Sets of Affine Predicates
Direction	Forward / Backward
Transfer function	$f_a(x) = (x - \texttt{KILL}_a) \cup \texttt{GEN}_a$
Meet operator $\wedge$	Set Intersection (also, $x \wedge \top = x$)
Boundary condition	$\texttt{out}[n_{entry}] = \{\}$ (empty) / $\texttt{in}[n_{exit}] = \{\}$ (empty)
Initialization for non-entry nodes n	$\texttt{in}[n] = \top$ / $\texttt{out}[n] = \top$

refer the reader to existing literature on row-reduced echelon form [14] for a proof.

B. Identifying the Maximal Set of Assembly PC Locations where the Variable Correlations Hold

We have now obtained source-to-assembly affine expressions at those PC locations in the assembly program that have been correlated with PC locations in the source program. The equivalence checker typically minimizes the number of PC locations that it needs to correlate as it makes the proof effort simpler and smaller – for example, our equivalence checker only correlates the program entry, program exits, loop heads and function calls. To improve the coverage of the inferred debugging information, we next *spread* the information in both forward and backward directions of the assembly program. In other words, for any given source-to-assembly affine relation at a given pair of PCs, we are interested in finding the maximal range of PCs in both programs (in both forward and backward directions) where this affine relation (or its modified form) would continue to hold.

We use three subprocedures to implement this spreading of an affine relation:

- We use a forward DataFlow Analysis (DFA) to transfer the information encoded in the affine invariants in the forward direction (section IV-B1).
- We use the reversibility of computation to extend the coverage of the forward DFA (section IV-B2).
- We use a backward DFA to transfer the information encoded in the affine invariants in the backward direction (section IV-B3).

1) Forward DFA: The forward DFA is specified in table I. The domain of DFA values is a set of predicates that relate source variables to assembly expressions, i.e., source-to-assembly affine predicates.

At an assembly PC that is already correlated with a source PC, this set of predicates is exactly equal to the relational invariants present in the bisimulation proof at the corresponding PC pair. For other assembly PCs (that are not correlated through the bisimulation proof with a source PC), we try

```
s000:
// move 32-bit val to xmm2[0-31]
0: movd    val, %xmm2
// assign 0 to %eax
8: xorl    %eax, %eax
//copy xmm2[0-31] bits to four positions in xmml
//-- [0-31], [32-63], [64-95] and [96-127]
a: pshufd $0x0, %xmm2, %xmm1
f: nop
//copy 4 elements from Y to xmm0
10: movdqa Y(%eax), %xmm0
18: add    $0x10, %eax
//add four elements from Y in xmm0 to val in xmml
1b: paddd  %xmm1, %xmm0
// store the four elements in xmm0 to X
1f: movaps %xmm0, X(%eax-0x10)
//compare eax with N*4
26: cmp    $0x1f400, %eax
// loop back
2b: jne    10 <s000+0x10>
2d: xorl   %eax,%eax
2f: ret
```

Fig. 6: Vectorized assembly code generated at -O3 optimization level for C program shown in fig. 1a

and infer predicates by transferring the predicates from a previous correlated assembly PC. The DFA's meet operator is set intersection, indicating that a predicate should be true on all incoming paths for it to be added to the debugging headers. All nodes except the start node are initialized with a special value denoted by $\top$ such that $x \wedge \top$ evaluates to x.

The transfer function is expressed using the $\texttt{GEN}_a$ and $\texttt{KILL}_a$ sets for an assembly instruction $\texttt{a}$. We define $\texttt{GEN}_a$ as:

$$\texttt{GEN}_a = \left\{ \begin{array}{ll} Inv_{s,a} & \text{for } a \text{ correlated with } s \text{ in product program,} \\ & \text{and } a \text{ is mapped to } s \text{ in debug headers} \\ empty & \text{otherwise} \end{array} \right\}$$

$Inv_{s,a}$ represents the source-to-assembly affine predicates inferred at the correlated PC pair (s, a). $\texttt{GEN}_a$ represents the set of predicates that are generated at an assembly instruction a: if a is correlated with *some* source instruction s in the proof, and a also maps to s in the debugging headers, then we use $Inv_{s,a}$ to populate the predicates at a. Otherwise, $\texttt{GEN}_a$ evaluates to the empty set. $\texttt{KILL}_a$ represents those predicates that become invalid because of the modification of an assembly register/memory-location by the instruction a.

$$\texttt{KILL}_a = \{p \mid p \text{ refers to a reg/memloc modified by } a\}$$

We point out here that this definition of $\texttt{KILL}_a$ is imprecise because we do not kill those predicates whose source variables have been modified. In our experience, designing a DFA that takes into account the modifications on both source and assembly programs is challenging, and so our DFA only takes into account the assembly instruction's modifications. This imprecision in our analysis could have the following ramification: during debugging, the programmer may observe slightly stale values in the short interval of assembly instructions that are uncorrelated (but whose endpoints are correlated) in the bisimulation proof. This will become clearer in our discussion through an example below.

We use the vectorized assembly code as shown in fig. 6 as the running example for our subprocedures for *Spreading* the predicates. One of the affine invariants generated by the equivalence checker in the bisimulation proof across the IR shown in fig. 1b and this vectorized assembly at the correlated end point (I2, 0x10) is : '`i=%eax/4`'. When we perform the forward DFA, the predicate gets spread across the other PCs inside the loop – from 0x10 to 0x18. Since the instruction at PC 0x18 is modifying the register `%eax`, the predicate is killed and hence not spread further to the succeeding PCs. While debugging the executable updated with this resulting predicate after the forward DFA, the debugger will print a constant value of `i = 0` in the first loop iteration of the assembly code, `i = 4` in the second assembly loop iteration, `i = 8` in the third, and so on. Notice that one assembly loop iteration is equivalent to four source loop iterations (as correctly captured in `'.debug_line'`). Thus, our updated debugging information is slightly imprecise — a more precise debugging header would allow the debugger to print incremental values of `i = {0,1,2,3}` in the first assembly loop iteration (depending on the assembly PC value), `i = {4,5,6,7}` in the second assembly loop iteration, `i = {8,9,10,11}` in the third assembly loop iteration, and so on. This imprecision occurs because when we spread the predicates across assembly PCs, we do not account for changes in the source variables on the correlated paths (as discussed above). These imprecisions are local and confined to small code regions limited by the length of the correlated paths in the product-CFG. Even with these imprecisions, our tool provides significant improvements over the existing debugging headers, as we show in our experiments: we find that the existing debugging headers either have missing information or grossly incorrect information (e.g., `i = 0` for the entire duration of the loop). It is worth noting here that fully precise debugging information is sometimes impossible to realize, e.g., if multiple source statements are collapsed into a single assembly instruction. Even so, we think it is possible to improve the precision of our algorithm and we leave this for future work.

2) Exploiting Reversibility of Computation: In the forward DFA discussed in section IV-B1, a predicate of the form `x=expression(r1)` gets killed across an assembly instruction a: `r1 = r1+c`, where `c` is a constant, because a modifies `r1`. However, it is possible to exploit the reversibility of the addition operator to modify the predicate to `x=expression(r1-c)` (instead of killing it).

Thus we improve our forward DFA's transfer function to take into consideration the reversibility of the bitvector addition and subtraction operations: we do not kill the predicates across these reversible operations but instead modify them appropriately. Because addition and subtraction are common program operations, we find that this improvement to the transfer function often results in noticeable improvements (detailed in section V-B).

For the running example, the instruction at PC 0x18 in fig. 6, adds 16 to register `%eax`. The improved transfer function results in an expanded range of PCs for the modified predicate '`i=(%eax−16)/4`', from 0x1b to 0x2d.

3) Backward DFA: We also propagate the source-to-assembly affine relations (obtained from the bisimulation proof) in the backward direction, just like we propagate them in the forward direction. The backward DFA specified in table I is almost identical to the forward DFA: the transfer function is based on identical $\texttt{GEN}_a$ and $\texttt{KILL}_a$ set values, and the meet operator is set intersection. The primary difference is in the DFA's boundary condition: for the backward DFA, the boundary conditions initialize the `in` values for exit nodes: $\texttt{in}[n_{exit}] = \{\ \}$ (empty). Further, instead of initializing the `in` values for non-entry nodes, backward DFA involves the initialization of the `out` values for the non-exit nodes(n): $\texttt{out}[n] = \top$.

For the running example (fig. 6), backward DFA spreads the predicate '`i=%eax/4`' (which was valid over 0x10 to 0x18 earlier) over an expanded PC range of 0xa to 0x18. It doesn't spread to PC 0x8 as the instruction at PC 0x8 modifies the register `%eax` (with a non-reversible computation).

Notice that we do not allow a predicate to spread backwards across a non-reversible instruction, even though this is possible to do through the weakest-precondition predicate transformer. For example, the predicate '`i=%eax/4`' would change to '`i=` 0' (weakest-precondition) while passing backwards through the non-reversible instruction at 0x8 (to spread to PCs 0x8 and 0x0). However notice that `i` is not even defined at PC 0x0, and so the addition of the predicate '`i=` 0' at PC 0x0 would be superfluous. To avoid this, we restrict the backward DFA to spread predicates only across reversible instructions.

V. Evaluation

For the experimental evaluation, we use optimized ELF object files generated using recent versions of production compilers, namely, GCC-8.4, Clang/LLVM-12, and ICC-2021.2.0 with the optimization flags `-O3 -msse4.2`. The set of functions for evaluation are taken from the TSVC benchmark, as these functions represent programs which involve a large set of compiler transformations, including Loop invariant code motion, loop peeling, instruction selection, loop inversion, loop unswitching, strength reduction, and many more. Moreover, these programs also involve the most complex vectorizing transformations — thus it is harder for compiler developers to preserve debugging information for these programs. The equivalence proofs between the C program and the optimized x86 assembly are generated using the Counter [11] equivalence checker.

The original source file, the corresponding optimized ELF object file and the bisimulation proof across the original and optimized programs are given as inputs to the proposed tool. The output of the tool is an updated object file where the debugging headers have been improved using the algorithm discussed in section IV. We use the `gimli` library in Rust to update the debugging headers in the object files.

We find that the existing debugging information is often buggy across the types of transformations produced on the TSVC benchmarks. In some cases, the debugging information

for a variable is missing. In other cases, the debugging information for a variable is incorrect – i.e., GDB displays the wrong value when that variable is queried at a program point. Whenever our tool is able to find a new expression for a variable at a program point, we simply remove the old expression (for that variable) from the debugging header and replace it with the new expression.

For evaluation, we count the number of instances where:

- The debugging information was not present in the original object file and was present in the updated object file. We call this the case of *Missing* debugging information.
- In the original object file, the debugging information for a variable at a program point indicated that it has a constant value (i.e., a value that is independent of the assembly registers/memory locations) *and* in the updated object file, that the same variable is associated with a non-constant expression (i.e., an expression that depends on the current state of the assembly registers/memory locations). We call this the case of *IncorrectlyConstant* debugging information.

In both cases, we can be certain that there is an *improvement* in the debugging headers. We find that it is a common occurrence that a variable is incorrectly associated with a constant value in the debugging headers (where an expression was required) and so we count such improvements in our evaluator. However, this method of evaluation is conservative, i.e., there may have been more improvements that we are not counting, e.g., if both the original and the updated object files associated an expression with a variable, but the expression in the original object file was incorrect. Such improvements can only be counted through manual examination — instead, we chose to use an automatic evaluator.

A. Results

We show the improvement metrics in the debugging headers for object files generated by GCC, Clang/LLVM (table II) and by ICC (table III). The total number of TSVC functions we tested are 75. Some function-compiler pairs are excluded in the results because our best-effort equivalence checker could not generate an equivalence proof for them. For each function-compiler pair, we show

- the total number of PCs in the optimized assembly program (T),
- the number of PCs for which at least one variable's debugging information was updated (U).
- the number of variables for which the debugging information was improved for at least one PC (V).
- the cumulative count of the number of PC-variable pairs for which the debugging information was already available in the original object file (but some of these mappings could be potentially incorrect before updation) (O)
- the cumulative number of PC-variable pairs for which the debugging information was not present (*Missing cases*) in the original object file and is present in the updated object file (M)

TABLE II
TSVC BENCHMARK RESULTS FOR (A) CLANG AND (B) GCC.

Table (2a)				**Table (2b)**			
F_n	*T/U*	V	*O/M/ I*	F_n	*T/U*	V	*O/M/I*
s000	19/14	1	17/0/14	s000	11/ 7	1	3/ 6/ 1
				s1111	20 / 19	1	0 / 19 / 0
s1112	27/22	1	25/0/22	s1112	12/ 8	1	3/ 7/ 1
s1119	17/8	1	20/0/ 8				
				s111	28 / 28	1	0 / 28 / 0
s112	13/10	1	0/10/ 0	s112	22/ 22	1	0/ 22/ 0
				s113	20 / 7	1	9 / 7 / 0
s116	17/14	1	1/13/ 1				
s119	37/12	1	52/0/12	s119	27/ 26	2	12/ 32/ 1
s121	40/17	1	60/0/17	s121	18/ 18	1	11/ 18/ 0
s1221	16/12	1	12/0/12	s1221	9/ 5	1	1/ 5/ 0
s122	16/14	2	63/12/ 3	s122	16/ 16	2	55/ 21/ 6
s1251	20/17	1	17/0/17	s1251	13/ 12	1	0/ 12/ 0
				s125	24/ 20	3	6/ 43/ 0
				s127	17/ 16	2	1/ 31/ 0
				s1281	17/ 16	1	0/ 16/ 0
				s128	20/ 19	2	2/ 36/ 1
s131	37/17	1	70/0/17	s131	15/ 15	1	15/ 15/ 0
s132	53/21	1	210/0/21	s132	28/ 28	1	84/ 28/ 0
s1351	17/15	4	14/45/14	s1351	9/ 8	4	0/ 29/ 0
s162	52/41	1	54/41/ 0	s162	43/ 9	1	62/ 8/ 1
s171	40/25	1	66/7/18				
s173	17/14	1	30/0/14	s173	9/ 8	1	9/ 8/ 0
				s174	64/ 48	1	64/ 48/ 0
				s2233	39/ 37	1	13/ 35/ 2
s2244	47/15	1	44/0/15	s2244	24/ 24	1	0/ 24/ 0
s243	51/16	1	48/0/16	s243	21/ 19	1	0/ 19/ 0
s251	15/12	1	12/0/12				
s252	18/14	1	30/0/14				
				s311	15/ 5	1	3/ 5/ 0
s319	32/26	1	53/0/26	s319	22/ 15	1	2/ 15/ 0
				s3251	44/ 44	1	0/ 44/ 0
s351	25/19	1	26/19/ 0				
s352	34/28	2	31/28/ 0				
s452	31/26	1	29/0/26	s452	14/ 10	1	3/ 9/ 1
s453	19/14	1	17/0/14	s453	11/ 7	1	3/ 6/ 1
				sum1d	15/ 10	1	3/ 10/ 0
				va	8/ 7	1	0/ 7/ 0
vdotr	25/19	1	39/0/19	vdotr	17/ 10	1	2/ 10/ 0
				vpv	9/ 8	1	0/ 8/ 0
vpvpv	21/18	1	18/0/18	vpvpv	10/ 9	1	0/ 9/ 0
vpvts	23/18	1	21/0/18	vpvts	12/ 8	1	3/ 7/ 1
vpvtv	21/18	1	18/0/18	vpvtv	10/ 9	1	0/ 9/ 0
vtv	17/14	1	14/0/14	vtv	9/ 8	1	0/ 8/ 0
vtvtv	21/18	1	18/0/18	vtvtv	10/ 9	1	0/ 9/ 0
Avg.	**27/18**	**1.2**	**36/6/13.5**		**19/16**	**1.2**	**10/18/0.4**

- the cumulative number of PC-variable pairs which had an *IncorrectlyConstant* debugging information was present in the original object file and are replaced with a non-constant expression in the updated object file (I)

On average, the percentage of PCs where at least one variable's *Missing* information is added is: 18% for Clang/LLVM, 73% for GCC, and 12% for ICC. Similarly, the percentage of PCs where at least one variable's *IncorrectlyConstant*

TABLE III
(A) TSVC BENCHMARK RESULTS FOR ICC. (B) TSVC BENCHMARKS WITH NO IMPROVEMENT IN ICC.

Table (3a)				Table (3b)			
F_n	*T/U*	*V*	*O/M/I*	Functions with no improvement for icc			
s114	46/46	2	0 /79/0	s000	s132	s272	s441
s124	50/44	1	50/44/0	s111	s1421	s274	s452
s125	37/36	1	74/36/0	s112	s173	s293	s453
s127	63/57	1	63/57/0	s119	s2244	s311	sum1d
s252	19/9	1	19/9 /0	s122	s251	s317	va
				s1279	s254	s319	vdotr
				s1281	s2711	s4115	vif
				vpv	vpvpv	vpvts	vpvtv
Avg.	**43/38**	**1.2**	**41/45/0**				

information is corrected is: 55% for Clang/LLVM, 2% for GCC, and 0% for ICC. It is interesting to see how Clang/LLVM has more number of *IncorrectlyConstant* debug information, while GCC and ICC have a larger fraction of *Missing* debug information. Moreover, in our experiments, we find that ICC preserves the most amount of correct debugging information, because our modifications produced only a 12% overall improvement for ICC-compiled object files.

Recall that because our forward DFA does not take into account the source program's updates, the results of our analysis could potentially be slightly imprecise (section IV-B1). We have manually checked several executables to confirm that this imprecision is bounded by the amount of unrolling performed by the compiler, e.g., if the compiler unrolls the loop by a factor of eight, then the maximum deviation between GDB's reported value and the actual value of a variable is upper-bounded by seven. Even with this imprecision, our tool makes a significant improvement over the existing debugging headers, where the values were either missing or incorrectly constant (e.g., zero) for the entire loop duration.

All the TSVC programs involve at most two (potentially nested) loops. We have also tested our tool on larger programs involving three or more loops, and potentially function calls and other complex control flow. Although the larger programs trigger complex compiler transformations where debugging information is typically lost, the improvement results remain similar. We are sometimes limited by the scale of the programs that the equivalence checker can handle in a reasonable amount of time — our goal in this work is not to evaluate the capabilities of the equivalence checker, but to evaluate our algorithms to update the debugging headers based on an obtained equivalence proof. As such, we find that our updation algorithms are scalable and we don't see any difficulties posed by the size of the programs on which the tool is applied.

The average change in the size of the updated executable ranges from -2% to 21%, with a mean increase in the executable size of 3.4%. This change in executable size has no effect on the size of the memory consumption and runtime of the executable code; the updated headers only take more disk space and are accessed only during a debugging session.

B. Ablation Studies

Our proposed technique for improving the debugging headers involves three algorithms (1) a forward DFA (section IV-B1), (2) a modification to the forward DFA to exploit reversibility of computation (section IV-B2), and (3) a backward DFA (section IV-B3). To study the individual contributions of each of these three algorithms, we list the improvement results for four different variants of our tool: (a) where all three algorithms (forward DFA with reversibility and backward DFA) are enabled, (b) where only the first two algorithms (forward DFA with reversibility), (c) where only the first algorithm is enabled (forward DFA without reversibility), and (d) where none of these algorithms are used (only the proof file is used to populate the debugging headers).

Based on our evaluation of all these four variants, we make the following observations:

1) Just using the proof file without any spreading through DFAs results in only 1-2 PCs to be updated in the whole program. This is unsurprising because the bisimulation proof generated by the equivalence checker only correlates one PC in each loop, and hence that is the only PC for which we are able to update the debugging headers.
2) The percentage improvements shown by
   - variant (c) over (d) are 55% for Clang/LLVM, 17% for GCC and 8% for ICC. This measures the improvements due to the forward-DFA to spread the debug information.
   - variant (b) over (c) are 7% for Clang/LLVM, 33% for GCC and 1% for ICC. This measures the improvements due to the use of reversibility of computation in the forward DFA.
   - variant (a) over (b) are 3% for Clang/LLVM, 1.5% for GCC and 2% for ICC. This measures the improvements due to the backward DFA.

   We observe that Clang/LLVM has most improvements for variant (c) (forward DFA without reversibility) while GCC shows most improvements for variant (b) (reversibility). This can be attributed to the position of the iterator increment/decrement instruction in Clang/LLVM and GCC – it's typically present near the tail of a loop for Clang/LLVM while it's near the head of a loop for GCC.

C. Limitations

Our tool is not without limitations, the primary limitation being its dependence on the equivalence checker for the generation of a proof of equivalence across the source and the lowered and optimized assembly version. For our experiments, we consciously chose benchmarks for which the equivalence checker can automatically compute the equivalence proof *and* which exhibit aggressive optimizations that typically cause a loss of debugging information. Our results show the efficacy of our algorithms to spread the debugging information for these challenging benchmarks. We expect future improvements in equivalence checking to seamlessly integrate with our

algorithms to make this method of improving debugging information widely applicable.

Another limitation is that our tool does not support product programs that require correlations of a single assembly point with several source points. These product programs are required to prove equivalences in the presence of transformations like loop fusion, loop re-rolling, etc., and can have multiple potentially different set of invariants at the same assembly location. The support for such product programs would require techniques to choose the right set of invariants from these multiple possible relations while keeping the resulting debug information sound.

VI. Related Work and Conclusions

Debuggability and optimization are often viewed as conflicting goals: compiler optimizations are often disabled (e.g., `-Og`) to keep the program debuggable. Keeping the debugging headers consistent adds to the developer burden during compiler development which is already highly complex. Our proposal leverages the capabilities of a blackbox equivalence checker to somewhat relieve the developer through an automatic tool to improve the debugging headers. Our limited experimental studies demonstrate promise, and we expect future improvements in equivalence checking to make this approach even more practical. We contribute algorithms to maximize the debugging information obtained from an equivalence proof.

This problem of improving debuggability has been previously tackled through various diverse approaches, e.g., by using recompilation or dynamic de-optimization to undo the optimizations [15], by changing the source program to reflect the effects of compiler optimizations [16], and by providing new compiler-debugger interfaces [17].

Custom compilers have been developed by modifying an existing compiler to gather extra debugging information [15]–[21]. Algorithms have been proposed to determine the *currentness* of a variable [21]–[23], where a variable is considered non-current if it is inconsistent with the source-level value expected at a breakpoint. Another interesting approach involves the concurrent execution of unoptimized and optimized programs on identical inputs and comparing their behavior; in case of differences, the debugging tool suggests the disabling of specific optimizations on programmer queries [24]. Because errors in debugging headers are common, prior work has also studied the validation of existing debugging information [25], [26] — in contrast, we try and fix/improve it. Our approach resembles the symbolic debugging proposal [27] where the aim is to recover debugging information through static analysis approaches.

In contrast to prior approaches, we support existing unmodified compilers and our approach is agnostic of the compiler and the debugger being used. The debuggability improvements produced by our tool are within the constraints of the current compilation and debugging toolchains, and allow programmers to transparently enjoy benefits of these improvements.

References

[1] H. Massalin, "Superoptimizer: A look at the smallest program," in *ASPLOS '87: Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 1987, pp. 122–126.

[2] R. Joshi, G. Nelson, and K. H. Randall, "Denali: A goal-directed superoptimizer," in *PLDI '02: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002, pp. 304–314.

[3] S. Bansal and A. Aiken, "Automatic generation of peephole superoptimizers," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: ACM, 2006, pp. 394–403. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168906

[4] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: a new approach to optimization," in *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 2009, pp. 264–276. [Online]. Available: http://www.cs.cornell.edu/~ross/publications/eqsat/

[5] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 305–316. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451150

[6] B. Churchill, R. Sharma, J. Bastien, and A. Aiken, "Sound loop superoptimization for Google native client," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. ACM, 2017, pp. 313–326.

[7] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, J. Taneja, and J. Regehr, "Souper: A synthesizing superoptimizer," *CoRR*, vol. abs/1711.04422, 2017. [Online]. Available: http://arxiv.org/abs/1711.04422

[8] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua, "An evaluation of vectorizing compilers," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 372–382. [Online]. Available: https://doi.org/10.1109/PACT.2011.68

[9] M. Dahiya and S. Bansal, "Black-box equivalence checking across compiler optimizations," in *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings*, 2017, pp. 127–147. [Online]. Available: https://doi.org/10.1007/978-3-319-71237-6_7

[10] B. Churchill, "Blackbox equivalence checking of program optimizations," Ph.D. dissertation, Stanford University, 2019.

[11] S. Gupta, A. Rose, and S. Bansal, "Counterexample-guided correlation algorithm for translation validation," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428289

[12] B. Churchill, O. Padon, R. Sharma, and A. Aiken, "Semantic program alignment for equivalence checking," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: ACM, 2019, pp. 1027–1040. [Online]. Available: http://doi.acm.org/10.1145/3314221.3314596

[13] M. Müller-Olm and H. Seidl, "Analysis of modular arithmetic," in *Programming Languages and Systems*, M. Sagiv, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 46–60.

[14] H. Anton and C. Rorres, *Elementary Linear Algebra: Applications Version*. Wiley.

[15] U. Hölzle, C. Chambers, and D. Ungar, "Debugging optimized code with dynamic deoptimization," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, ser. PLDI '92. New York, NY, USA: Association for Computing Machinery, 1992, p. 32–43. [Online]. Available: https://doi.org/10.1145/143095.143114

[16] C. M. Tice, "Non-transparent debugging of optimized code," 2 2000. [Online]. Available: https://dl.acm.org/doi/book/10.5555/894940

[17] L. Berger and R. Wismüller, "Source-level debugging of optimized programs using data flow analysis," 1992.

[18] C. Jaramillo, R. Gupta, and M. L. Soffa, "Fulldoc: A full reporting debugger for optimized code," in *Static Analysis*, J. Palsberg, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 240–259.

[19] P. T. Zellweger, "An interactive high-level debugger for control-flow optimized programs," in *Proceedings of the Symposium on High-Level Debugging*, ser. SIGSOFT '83. New York, NY, USA: Association

for Computing Machinery, 1983, p. 159–172. [Online]. Available: https://doi.org/10.1145/1006147.1006183

[20] G. Brooks, G. J. Hansen, and S. Simmons, “A new approach to debugging optimized code,” in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, ser. PLDI ’92. New York, NY, USA: Association for Computing Machinery, 1992, p. 1–11. [Online]. Available: https://doi.org/10.1145/143095.143108

[21] A.-R. Adl-Tabatabai and T. Gross, “Source-level debugging of scalar optimized code,” in *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, ser. PLDI ’96. New York, NY, USA: Association for Computing Machinery, 1996, p. 33–43. [Online]. Available: https://doi.org/10.1145/231379.231388

[22] R. Wismüller, “Debugging of globally optimized programs using data flow analysis,” in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI ’94. New York, NY, USA: Association for Computing Machinery, 1994, p. 278–289. [Online]. Available: https://doi.org/10.1145/178243.178430

[23] M. Copperman, “Debugging optimized code without being misled,” *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, p. 387–427, may 1994. [Online]. Available: https://doi.org/10.1145/177492.177517

[24] C. Jaramillo, R. Gupta, and M. L. Soffa, “Comparison checking: An approach to avoid debugging of optimized code,” in *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-7. Berlin, Heidelberg: Springer-Verlag, 1999, p. 268–284.

[25] G. A. Di Luna, D. Italiano, L. Massarelli, S. Österlund, C. Giuffrida, and L. Querzoni, “Who’s debugging the debuggers? exposing debug information bugs in optimized binaries,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1034–1045. [Online]. Available: https://doi.org/10.1145/3445814.3446695

[26] Y. Li, S. Ding, Q. Zhang, and D. Italiano, “Debug information validation for optimized code,” 2020.

[27] J. Hennessy, “Symbolic debugging of optimized code,” *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, p. 323–344, jul 1982. [Online]. Available: https://doi.org/10.1145/357172.357173