PLINER: Isolating Lines of Floating-Point Code for Compiler-Induced Variability

Hui Guo
Department of Computer Science
University of California, Davis
Davis, CA, USA
higuo@ucdavis.edu

Ignacio Laguna
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA, USA
ilaguna@llnl.gov

Cindy Rubio-González
Department of Computer Science
University of California, Davis
Davis, CA, USA
crubio@ucdavis.edu

Abstract—Scientific applications are often impacted by numerical inconsistencies when using different compilers or when a compiler is used with different optimization levels; such inconsistencies hinder reproducibility and can be hard to diagnose. We present PLINER, a tool to automatically pinpoint code lines that trigger compiler-induced variability. PLINER uses a novel approach to enhance floating-point precision at different levels of code granularity, and performs a guided search to identify locations affected by numerical inconsistencies. We demonstrate PLINER on a real-world numerical inconsistency that required weeks to diagnose, which PLINER isolates in minutes. We also evaluate PLINER on 100 synthetic programs, and the NAS Parallel Benchmarks (NPB). On the synthetic programs, PLINER detects the affected lines of code 87% of the time while the state-of-the-art approach only detects the affected lines 6% of the time. Furthermore, PLINER successfully isolates all numerical inconsistencies found in the NPB.

Index Terms—reproducibility, numerical reliability, floating-point arithmetic, compiler optimizations, scientific computing

I. INTRODUCTION

Floating-point arithmetic is used in virtually all classes of scientific and engineering applications and it is the bedrock for the numerical computations that these applications execute. Since important decisions are made based on the numerical results of these applications, the degree of reproducibility and numerical consistency of these applications is crucial. With advanced HPC systems moving to an era of heterogeneous computing where code can be run on different architectures in the same system (e.g., CPUs and accelerators), and compilers for different architectures can yield different floating-point programs, reproducibility and numerical consistency in such scenarios becomes a challenging problem.

While many programming languages provide support for floating-point arithmetic, usually floating-point is underdefined in the language specification, and as a result compilers have a great degree of freedom in generating code around floating-point programs. This freedom comes at the price of having possibly different numerical results when two different compilers optimize the same code—since floating-point operations are non-associative—or even for the same compiler when different optimization levels are used, e.g., -O2 versus -O3. We refer to such inconsistencies as compiler-induced numerical inconsistencies. These inconsistencies can sometimes significantly impact the productivity of programmers when they affect numerical results in unpredictable ways.

When compiler-induced inconsistencies arise in floating-point programs, programmers are interested in understanding what parts of their application are causing these inconsistencies. However, currently tools support to help programmers identify the source of such inconsistencies is scarce, and as a result programmers spend a significant amount of time isolating such issues. This paper presents a novel approach to mitigate such numerical inconsistencies more effectively and help programmers identify their origin faster.

A Real-World Motivating Example. Consider a real-world numerical inconsistency encountered recently at the Lawrence Livermore National Laboratory (LLNL) in the development of a new hydrodynamics application [1, 11], Laghos. The application was being adapted to study the impact of using performance portability layers [16] in the new Sierra system at LLNL, which uses NVIDIA V100 GPUs. During the porting process, programmers observed that the energy computed in the application had a significant numerical inconsistency when the code was compiled using the IBM XL C/C++ compiler (xlc) with -O3 in contrast to other compilers available in the system, such as gcc and clang. Table I shows the computed energy [\text{\texttt{e}}] for several compilers and optimization levels (see the last row). As we can see, the computed energy was very different for xlc -O3; note that using lower optimizations is usually not appealing as programmers want to extract most of the performance out of compiler optimizations.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Optimization</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>clang</td>
<td>-O1</td>
<td>129664.9230611104</td>
</tr>
<tr>
<td>clang</td>
<td>-O2</td>
<td>129664.9230611104</td>
</tr>
<tr>
<td>clang</td>
<td>-O3</td>
<td>129664.9230611104</td>
</tr>
<tr>
<td>gcc</td>
<td>-O1</td>
<td>129664.9230611104</td>
</tr>
<tr>
<td>gcc</td>
<td>-O2</td>
<td>129664.9230611104</td>
</tr>
<tr>
<td>gcc</td>
<td>-O3</td>
<td>129664.9230611104</td>
</tr>
<tr>
<td>xlc</td>
<td>-O1</td>
<td>129664.9230611104</td>
</tr>
<tr>
<td>xlc</td>
<td>-O2</td>
<td>129664.9230611104</td>
</tr>
<tr>
<td>xlc</td>
<td>-O3</td>
<td>144174.9336610391</td>
</tr>
</tbody>
</table>

It took several weeks of effort to diagnose the issue and isolate the code in the application that, combined with this
compiler and optimization level, caused the numerical inconsistency. After a large effort using several ad-hoc methods to compile files and functions with different levels of optimization, the issue was isolated to a single function. It was observed that when this function was compiled with -O2 and the rest of the program was compiled with -O3 the numerical inconsistency disappeared.

Although the programmers of the application wanted to know the exact line of code in the isolated function that caused the issue, this was an even more challenging task: this function was significantly large to analyze manually and no tools were available to automatically identify the lines of code in the function that caused the inconsistency. Examples of hard-to-diagnose inconsistencies such as this are not uncommon in the scientific computing community. With trends on floating-point precision reduction in accelerators (e.g., half precision) and more aggressive compiler optimizations, programmers are likely to increasingly experience such issues.

**Our Contributions.** In this paper, we present PLINE, a new approach to automatically isolate the lines of code that cause compiler-induced numerical inconsistencies in floating-point programs. PLINE consists of a search engine and a precision transformer. The search engine iteratively divides the code and invokes the transformer to enhance precision for a sequence of code areas. A key concept in PLINE is that we recompile the transformed program to test result consistency—if it yields consistent results, we believe that the origin of the compiler-induced numerical inconsistency is included in the transformed code sequence. We iteratively zoom into the suspicious code sequence until the lines of code responsible for the compiler-induced inconsistency are isolated. To scale the code search, we identify four levels of granularity: function, loop, basic block, and line. We first break and search the code with the top level of granularity (function), and gradually refine our search until the isolated code is not divisible.

Another crucial insight of PLINE is that we use high precision to simulate arithmetic over reals. It is known that floating-point compiler optimizations that induce numerical inconsistencies perform well on real arithmetic, but not necessarily well when precision is lost. As a result, the transformed program with high precision should yield consistent results. Lastly, we design the precision transformer to manipulate the abstract syntax tree (AST) of the program and rewrite the source code. This makes our algorithm applicable to different compilers and architectures.

Using PLINE we were able to automatically identify the line of code that cause the inconsistency in the hydrodynamic application described above in only a few minutes, while it took weeks of effort for the programmers to find the root cause. Moreover, we evaluate PLINE on 100 floating-point synthetic programs produced by Varity [4], a framework to generate random floating-point programs that produce numerical variations when compiled with different compiler optimization levels.

In our evaluation, 50 Varity programs produce inconsistent results when compiled with IBM XL C/C++ compiler with -O3 optimization level on the Power8 platform and 50 Varity programs produce inconsistent results when compiled with GCC compiler with -O3 -ffast-math compiler option on Intel platform in contrast to unoptimized code. PLINE isolated the origin of inconsistency in 87 out of 100 programs.

Moreover, we compare PLINE to the state-of-the-art tool that identifies root causes of floating-point errors, Herbgrind [23], on the 50 synthetic programs on an Intel platform. Herbgrind only reported root causes of floating-point errors in 5 out of 50 benchmarks, where only 3 of the reported cases correctly isolated the affected lines of code. We find that PLINE significantly outperforms Herbgrind in isolating compiler-induced numerical inconsistencies.

Lastly, we triggered three compiler-induced inconsistencies in the SNU NPB Suite [26], and show that PLINE successfully isolates the origin of the three inconsistencies in minutes.

To summarize, the contributions of this paper are:

- We design an algorithm that applies precision enhancement and hierarchical search to isolate the origin of compiler-induced numerical inconsistencies in floating-point programs (Section III).
- We present the implementation of our algorithm in the PLINE tool as an extension of the clang/LLVM compiler. To the best of our knowledge, PLINE is the first tool to combat compiler-induced numerical inconsistencies at a fine-grained level (lines of code) (Section IV).
- We show that PLINE is effective at isolating compiler-induced inconsistencies by evaluating PLINE on 100 floating-point programs across different architectures and compilers, the SNU NPB Suite, and a real-world numerical inconsistency in a hydrodynamics application (Section V).
- We compare PLINE to the state of the art that identifies root causes of floating-point errors, Herbgrind, and show that PLINE outperforms the state of the art in isolating compiler-induced numerical inconsistencies (Section V).

## II. Background

In this section, we discuss background information that will be useful later and a high-level overview of our approach.

### A. Compiler-Induced Numerical Inconsistencies

The real-world case of numerical inconsistency presented in Section I from the hydrodynamics application (Laghos) is an example of a compiler-induced numerical inconsistency. Generally speaking, there are two classes of compiler-induced numerical inconsistencies. The first class occurs when a given application $A$ is compiled with two different compilers, $C_1$ and $C_2$, that are available using an equivalent optimization level in both compilers (e.g., -O2); when the two resulting programs are executed with a given input, they produce different numerical results. The second class occurs when $A$ is compiled with two different optimization levels (e.g., -O2 and -O3) using the same compiler (e.g., $C_1$); when the outputs of the two resulting programs are compared, they are different.
B. Floating Point and Optimizations

Most compilers provide at least four optimization levels: -O0, -O1, -O2, and -O3. When code is highly optimized (e.g., with -O3), floating-point optimizations can provide significant performance improvements, but at the same time, they can generate code that is not compliant with the IEEE 754-2008 standard for floating-point arithmetic.

The philosophy of open-source compilers, such as gcc and clang, is by default to be compliant, or strict, with respect to IEEE 754-2008, unless the user indicates the opposite; the -ffast-math option can be used in gcc and clang to produce programs that are not compliant with IEEE. GPU compilers, such as nvcc, also provide a similar option, --use_fast_math. On the other hand, the philosophy of commercial compilers, such as xlc, is different: in the xlc compiler, -O3 by default performs aggressive optimizations that alter the semantics of the program and are not compliant with IEEE, i.e., by default with -O3 it is non-strict with respect to IEEE. The -qstrict option in xlc generates code that is compliant with IEEE.

C. Laghos Case

The case presented in Section I is an example of both of the two classes of inconsistencies described above, i.e., the energy that results from the program compiled with xlc with -O3 is different from the energy that results from the clang and gcc compilations and it is also different between the various optimization levels for xlc. In addition to the numerical inconsistencies, the execution time of the xlc -O3 version was significantly smaller than any of the other versions.

Using a number of methods to compile functions at different optimization levels, the issue was isolated to the function rUpdateQuadratureData2D. It was observed that when this function was compiled with -O2 and the rest of the program was compiled with -O3 the numerical inconsistency disappears. However, the function was composed of several loops and branches, and no insight lead to the conclusion that a specific line of code originated the inconsistency. Once the function had been isolated, a number of programmers still spent several days of effort diagnosing this case to no avail.

Instead of compiling function rUpdateQuadratureData2D with -O2, we increased the precision of this function to long double and recompiled the code with xlc -O3—this yields consistent results with other compilers or xlc -O2. In the Power8 xlc compiler, a long double data type is referred as IBM extended double type which uses a pair of double values for more bits of precision. We further minimized the code area that uses long double from function rUpdateQuadratureData2D to one line of code in the function. Specifically, we increased the precision of the operations on line 130 to long double and recompiled the code with -O3—this change ultimately eliminated the numerical inconsistency. Figure 1 highlights the line of code we isolated from Laghos.

After analyzing the assembly code for the function rUpdateQuadratureData2D, we observed that the division operation on line 130 is converted to a reciprocal operation and a multiplication in the -O3; we speculate that the compiler performs this transformation to improve performance while the -O2 version (or lower) performs the floating-point division operation. The reciprocal operation easily introduces exceptions when the values of gradv10, gradv00, gradv11 are subnormal numbers that are close to zero while division operation performs well. The goal of pLINER is to automatically find cases such as this where increasing the floating-point precision of lines of code can help programmers isolate compiler-induced numerical inconsistencies.

D. Overview of pLINER

Figure 2 gives an overview of pLINER to isolate the origin of compiler-induced numerical inconsistencies. Suppose the floating-point program P yields a different output when compiled by compiler C1 with high compiler optimization level compared to the case when P is compiled with low optimization (or compiler with C2). Taking the source code of P and the triggering inputs, pLINER returns the code lines that are responsible for this inconsistency.

pLINER comprises two modules (see Figure 2): (1) a search-&-test engine that consists of hierarchical code isolation and compile-&-test, and (2) a program transformer that performs precision enhancement. Hierarchical code isolation identifies the minimal code areas that are suspicious, P_t, and precision
such as a transformation source code. Precision enhancement is achieved by abstract syntax tree (AST) of the program, and rewrites the such as LLVM or GCC IR. Our algorithm manipulates the instead of transforming a specific intermediate representation, sider enhancing floating-point precision at the source level our goal is to ensure that the transformed code is accurate with region. Theoretically, all operations on reals are accurate and operations to a user-specified higher precision in a given code enhancement replaces the floating-point types of variables and that produces the same result as the original code. We refer to inconsistencies), we emulate real arithmetic using high pre-

III. Approach

This section describes P LINER’s approach for precision enhancement, and our hierarchical algorithm for code isolation.

A. Source-to-Source Precision Enhancement

With the purpose of optimization, the compiler reorders and transforms program instructions. The optimized code is guaranteed to be equivalent to the original code on the condition that the arithmetic used in the computation satisfies the commutative, associative and distributive laws. Unfortunately, floating-point arithmetic does not satisfy associative and distributive laws, therefore, a compiler optimization could induce numerical inconsistencies on floating-point code.

To eliminate the impact of compiler optimizations on a floating-point code region (i.e., compiler-induced numerical inconsistencies), we emulate real arithmetic using high precision. Ideally, such emulation will result in optimized code that produces the same result as the original code. We refer to the process of transforming a given code region to use high precision as precision enhancement. Specifically, precision enhancement replaces the floating-point types of variables and operations to a user-specified higher precision in a given code region. Theoretically, all operations on reals are accurate and our goal is to ensure that the transformed code is accurate with compiler optimizations.

To allow our algorithm operate on all compilers, we consider enhancing floating-point precision at the source level instead of transforming a specific intermediate representation, such as LLVM or GCC IR. Our algorithm manipulates the abstract syntax tree (AST) of the program, and rewrites the source code. Precision enhancement is achieved by function transformation that enhances precision for a function, and region transformation that enhances precision for code regions such as a for loop.

1) Function Transformation: Figure 3 illustrates the transformation with a simple function foo—Figure 3a shows the original definition of function foo, and Figure 3b describes the transformed definition of foo in which floating-point operations are performed in higher precision. Assume the user has specified the long double type to emulate reals. Changes in the code are highlighted. We describe these changes separately in the following paragraphs.

Scalar Parameters/Variables. For each of the floating-point scalar parameters such as arg1 in function foo, we first declare a new local variable with enhanced precision, which is initialized with the value of the associated parameter. As shown on line 2 in Figure 3b, a local variable var_arg1 is declared for parameter arg1 using higher precision, and assigned the value of arg1. Next, each use of such floating-point scalar parameter in an expression is replaced by its corresponding higher-precision variable (see variable replacement on line 5). For floating-point scalar local variables such as var1 in function foo, we rewrite each declaration to use higher precision, e.g., the first change on line 4 in Figure 3b.

Pointer and Array Parameters/Variables. For floating-point pointer and array parameters/variables such as arg2 in foo, we manually cast their values to higher precision when dereferenced (see line 4 in Figure 3b). We choose to keep floating-point array parameters/variables in their original precision to prevent a significant increase in memory usage.

Remarks. In function transformation, we do not intend to modify the function signature. Using the original signature can potentially cause downcasting when, for example, returning the value of var1 in double precision, as shown on line 6 in Figure 3b. However, we did not observe a necessity in our evaluation to transform function signatures. Finally, we transform the precision of math library function calls, such as sin and cos, for which a higher-precision implementation is available (e.g., sinl and cosl in C, respectively).

2) Region Transformation: We define a region as a sequence of basic blocks that has a single entry point from a basic block outside the region and a single exit point to a

There are cases where simply replacing a variable with its corresponding higher-precision variable would cause an error. For example, replacing variable x in statement swap(&x, &y) with its corresponding higher-precision variable will cause a compilation error because function swap takes the reference of double variable. Such cases are handled separately.

Fig. 3: Function Transformation Example.

(a) Original implementation for function foo.

(b) Function foo with enhanced precision.

```c
1  double foo(double arg1,double* arg2,int arg3){
2    double var1 = *arg2;
3    var1 = (arg1 - var1) / arg3;
4    return var1;
5  }
```

```c
1  double foo(double arg1,double* arg2,int arg3){
2    long double var_arg1 = arg1;
3    long double var1 = (long double) *arg2;
4    var1 = (var_arg1 - var1) / arg3;
5    return var1;
6  }
```
Double the compiler may still downcast the register and would not be downcast to described in the third condition.

Note that to avoid such precision loss by increasing the precision of lost due to the previous downcast.

Addition operation reads the operand is then downcast to perform the operation with enhanced precision.

Consider the following two operations as an example. Suppose floating-point operation in region any floating-point operations outside region are performed with enhanced precision, (2) there are as few floating-point downcasts as possible inside region, and (3) typecast. Algorithm 1 describes the categorization process. We define three sets to hold the variables in each category (line 1), and two additional sets: curr_reads and visited (line 2). curr_reads stores the collection of variables whose value has been read so far in the region, and visited contains the variables that have been categorized up to this point. The algorithm traverses the statements in the given region in reverse order (line 3). For each statement, we retrieve operand variables, i.e., set reads (line 4), and the variables

Data: Code Region : region
Result: Variable Sets: var_toReplace, var_toRevise, var_toCast
1 var_toReplace, var_toRevise, var_toCast = set(), set(), set();
2 curr_reads, visited = set(), set();
3 for st in reversed(region) do
4     reads = obtainVarReads(st);
5     writes = obtainVarWrites(st);
6     curr_reads += reads;
7     for v in curr_reads ∧ writes and not in visited do
8         visited.add(v);
9         if obtainDeclSt(v) in region then
10            var_toRevise.add(v);
11         else
12             var_toReplace.add(v);
13         end
14     end
15 end
16 var_toCast = curr_reads - visited;
17 return var_toReplace, var_toRevise, var_toCast

Algorithm 1: FP Variable and Parameter Categorization.

Basic block outside the region. In region transformation, we synchronize data at the entry/exit points.

Goals. Enhancing the precision of a region R requires the following: (1) all floating-point operations inside region R are performed with enhanced precision, (2) there are as few floating-point downcasts as possible inside region R, and (3) all floating-point operations outside region R are performed in their original precision. Upcasting all operands for every floating-point operation in region R suffices to achieve the first goal. However, the result of the operation is then downcast by the compiler when stored in a variable of lower precision. Consider the following two operations as an example. Suppose all variables are declared in double precision. We upcast each operand of the two arithmetic operations, which forces to perform the operation with enhanced precision.

\[ v_1 = \frac{(\text{long \ double})v_2}{(\text{long \ double})v_3}; \]
\[ v_1 = ((\text{long \ double})v_1 + (\text{long \ double})v_2); \]

First, v2 and v3 are cast to long double precision. The division is therefore performed in long double, but the result is then downcast to double before being stored in v1. The addition operation reads the operand v1, but precision has been lost due to the previous downcast.\(^2\) The second condition aims to avoid such precision loss by increasing the precision of v1. Note that v1’s precision must be reestablished once outside region R. Any floating-point operations outside region R that involve v1 should be performed in their original precision, as described in the third condition.

Variable Categorization. To achieve the above goals, we divide local floating-point variables and parameters into three categories: (1) revise declaration, (2) replace declaration, and (3) typecast. Algorithm 1 describes the categorization process. We define three sets to hold the variables in each category (line 1), and two additional sets: curr_reads and visited (line 2). curr_reads stores the collection of variables whose value has been read so far in the region, and visited contains the variables that have been categorized up to this point. The algorithm traverses the statements in the given region in reverse order (line 3). For each statement, we retrieve operand variables, i.e., set reads (line 4), and the variables

Data: Code Region : region
Result: Revised Code Region: region_revised
1 /* Insert sync statements at the entry. */
2 entryStmts = NULL;
3 for v in var_toReplace do
4     v_HP = v.name() + “_HP”;
5     entryStmts += Declare(v_HP); entryStmts += Assign(v_HP, v);
6 end
7 /* Revise statements of the region. */
8 for st in region do
9     reads, writes, decls = obtainReads(st), obtainWrites(st), obtainDeclare(st);
10    uses = reads + writes;
11    for v in uses do
12        if v in var_toReplace or var_toRevise then
13            replace(v.name(), v.name() + “_HP”);
14        end
15    if v in reads and var_toCast then
16        castUp(v);
17    end
18    for v in decls and var_toRevise do
19        v_HP = v.name() + “_HP”;
20        replaceDecl(v, v_HP);
21 end
22 /* Insert sync statements at the exit. */
23 exitStmts = NULL;
24 for v in var_toReplace do
25     v_HP = v.name() + “_HP”;
26     exitStmts += Declare(v_HP);
27 end
28 for v in var_toRevise do
29     v_HP = v.name() + “_HP”;
30     exitStmts += Assign(v, v_HP);
31 end
32 region_revised = entryStmts + region + exitStmts;
33 return region_revised

Algorithm 2: Region Transformation.
that are overwritten in the statement, i.e., set \textit{writes} (line 5). If a variable \texttt{var} is overwritten in the current statement and it will be read by an operation in or after the current statement (recall that statements are traversed in reverse order), i.e., \texttt{var} \in \textit{curr\_reads} \land \textit{writes}, then the variable is added to categories \texttt{var\_toRevise} or \texttt{var\_toReplace} based on the location of the variable’s definition. Each variable in \texttt{var\_toRevise} is defined inside the region (line 9-10) while each variable in \texttt{var\_toReplace} is defined prior to the start of the region (line 12). Lastly, once all statements are traversed, the remaining uncategorized floating-point variables whose value has been read, i.e., \texttt{curr\_reads} – \textit{visited}, are added to the set \texttt{var\_toCast} (line 16).

The categorization algorithm ensures that \texttt{var\_toCast} includes variables for which there are no write operations across the region, or for which there are no read operations following a write; \texttt{var\_toRevise} consists of variables for which there is at least one read operation following a write, and the variable is declared inside the region; and \texttt{var\_toReplace} includes the remaining variables for which there is at least one read operation following a write but the variable is declared prior to the entry of the region.

\textbf{Code Transformation}. Finally, we describe how we transform a given region based on the categorization of floating-point variables and parameters. The transformation involves inserting statements for data synchronization at the entry and exits points of the region, and rewriting variables and types in the statements within the region.

Algorithm 2 describes code transformation. At the entry of the region, for each variable \texttt{v} \in \texttt{var\_toReplace}, we declare a new variable with enhanced precision \texttt{v\_HP}, which is initialized with the value of its corresponding \texttt{v} (lines 3-6). Next, we replace each use of such variables \texttt{v} within the region, with its higher-precision version \texttt{v\_HP} (line 13). Lastly at the exit of the region, we assign the value of \texttt{v\_HP} back to its corresponding variable \texttt{v} (lines 26-29).

Variables in category \texttt{var\_toRevise} are declared inside the region, thus we simply revise their existing declarations. As shown on lines 8-23 in Algorithm 2, for each statement in the region we capture the variables it declares into the set \textit{decls}. If a declared variable \texttt{v} is in the category of \texttt{var\_toRevise}, we replace its definition with the definition of \texttt{v\_HP}, which denotes variable \texttt{v} declared with enhanced precision (lines 19-22). For any use in the region of variable \texttt{v} \in \texttt{var\_toRevise}, we replace \texttt{v} with \texttt{v\_HP} (line 13). Finally, at the exit of the region, we insert the original declaration of \texttt{v} initialized with the value of \texttt{v\_HP} (line 30-33) so that later operations outside the region access \texttt{v} in the original precision. Finally, for the category of variables \texttt{v} \in \texttt{var\_toCast}, we add a cast for any use of \texttt{v} in the region to enhance the precision of the corresponding operation (lines 15-17).

Consider our earlier example introduced when explaining the goals of region transformation,

\begin{verbatim}
1  var1 = var2 / var3;
2  var1 = var1 + var2;
\end{verbatim}

the transformed region for the two operations is as follows:

\begin{verbatim}
1  long double var1_HP = var1;
2  var1_HP = (long double)var2 / (long double)var3;
3  var1_HP = var1_HP + (long double)var2;
4  var1 = var1_HP;
\end{verbatim}

Variable \texttt{var1} is replaced with \texttt{var1\_HP} that is declared with enhanced precision, and two assignments synchronizing \texttt{var1} and \texttt{var1\_HP} are inserted at the entry and exit of the region. Moreover, variables \texttt{var2} and \texttt{var3} are upcast. In the revised code, (1) all operations are performed with enhanced precision, (2) there are no downcast instructions between the two operations, and lastly (3) the precision of statements prior to or after this region is not affected.

\textbf{Remarks}. The complexity of both the categorization algorithm and the transformation algorithm is \(O(nm)\) while \(n\) indicates the number of statements and \(m\) denotes the number of variables in the specified region. Similar to function transformation, we do not modify the declaration of pointer or array variables and parameters, i.e., these will not be categorized into \texttt{var\_toReplace} or \texttt{var\_toRevise}. Instead, these variables or parameters will be categorized into \texttt{var\_toCast} if the floating-point values they point to or store are used as operands in the given region.

\footnote{The synchronization of \texttt{var1} and \texttt{var1\_HP} at the entry of the region is not necessary because \texttt{var1\_HP} is rewritten right away in the first operation. Future analysis can help remove such unnecessary data synchronization.}
B. Hierarchical Code Isolation

To isolate the lines of code that are responsible for compiler-induced numerical inconsistency, we apply binary search on the code hierarchically. As shown in Figure 4, we first divide the code into functions and use binary search to isolate the functions that induce inconsistencies. In the search, we enhance the precision of a function by performing function transformation, as described in the previous section. For the isolated functions (shown in red in the figure), we further examine the loops, basic blocks (BBs), and lines to hierarchically isolate the problematic lines of code in each function. The precision of loops, BBs and lines are enhanced by applying region transformation. In the rest of this section, we first give an overview of the search algorithm, then elaborate on each phase of the search.

Code Search. During the search, we iteratively divide the program into regions, enhance the precision of each such region, and test whether the numerical inconsistency persists or disappears.

As an example, consider a program \( P \) for which a numerical discrepancy is observed between the results produced by \( P_{O0} \) and \( P_{O3} \), where \( P_{O0} \) is the executable of \( P \) generated with compiler optimization level \(-O0\), and \( P_{O3} \) is the executable generated with \(-O3\), i.e., there exists an input for which \( \text{Execute}(P_{O0}, \text{input}) \neq \text{Execute}(P_{O3}, \text{input}) \). Let \( P(r) \) denotes the program generated by enhancing the precision of the code region \( r \) in \( P \). The purpose of our search algorithm is to minimize \( r \) with respect to \( \text{Execute}(P_{O0}, \text{input}) = \text{Execute}(P_{O3}(r), \text{input}) \). We achieve this by performing binary search on the list of code regions. In this paper, we use the bisection algorithm followed by a 1-minimal [29] check. The bisection algorithm iteratively bisects the region list and selects the sublist in which regions have to be transformed to high precision to prevent inconsistencies. The search terminates when regions in both sublists have to be transformed to high precision, and returns the minimal sublist it isolated. We further minimize this sublist by attempting to remove each individual region in it. The final set the isolated regions is guaranteed to be 1-minimal, which indicates that removing any individual region will cause inconsistencies.

Function Isolation. First we collect all functions in the program, and discard those that do not contain floating-point operations. The remaining functions are organized in a list as shown at the top of Figure 4. The search is performed over this list to identify problematic functions. The subsequent phases in the search zoom into each problematic function individually.

Loop Isolation. This step is optional, and only applies to functions that include loops. Unsurprisingly, the origin of numerical inconsistencies is often found within loops. This is because rounding errors can accumulate in each loop iteration, which ultimately may cause a numerical inconsistency in the final result. Thus, the nature of loops makes them susceptible to compiler optimizations.

Figure 5 shows an example of the tree structures of loops. In this example there are six loops \( a-f \), while loop \( d \) and \( e \) are nested in loop \( a \), and loop \( f \) is nested in loop \( b \). In the search for loop isolation, we first perform binary search on the list of outermost loops (roots of trees, i.e., \( a \ b \ c \) in the example). Once a loop is identified as problematic, we zoom into its inner loops and repeat the process until the problematic loop is no longer divisible or no inner loops exist. As shown in the second and third rows in Figure 4, for function \( \text{Fun2} \), we first isolate Loop2 from the outermost loops and iteratively zoom into inner loops until we reach \( \text{inLoop1} \). Transforming all inner loops of \( \text{inLoop1} \) does not remove the numerical inconsistency and therefore \( \text{inLoop1} \) is returned at this stage.

Basic Block Isolation. A basic block (BB) consists of a straight-line code sequence with single entry and exit points. A function can be decomposed into one or more BBs. Specific to our algorithm, we may further break a BB into two depending on the last statement in the BB. If the last statement is a condition statement in an \textit{if} or \textit{while-do} statement, e.g., \text{if} \((a > 0)\), we simply divide the BB so that the last statement is a separate BB. This is to facilitate region transformation when inserting synchronization statements at the exit point of a BB. We then organize the BBs as a list for binary search.

Before the search, if a minimal list of problematic loops is returned from the Loop Isolation phase, we use it to filter the BBs so that we only search from BBs that are within the problematic loops. As shown in Figure 4, the problematic loop \( \text{inLoop1} \) in function \( \text{Fun2} \) is further decomposed into four BBs and binary search in Basic Block Isolation identifies BB1 and BB2 to be problematic. Specifically, if transforming all BBs in the search space to high precision at once does not remove the numerical inconsistency, it indicates that the region transformation on the problematic loop cannot be divided into multiple BB transformations. This is because at the end of a BB transformation, all floating-point values are truncated back to the original precision, and the truncation among BBs contributes to the inconsistency in the result. In this case, the hierarchical search terminates and returns the list of problematic loops. If no loops are identified, it returns the whole function as problematic.

Line Isolation. Finally, we break the problematic BBs into
lines\textsuperscript{5} and apply an algorithm similar to the Basic Block Isolation phase to identify the problematic code lines. If the line isolation succeeds, it returns the minimal code lines that induce the discrepancy, and if it fails, the minimal list of BBs isolated from the above phase is returned.

IV. IMPLEMENTATION

\textsc{pLiner} comprises two main modules: (1) program transformer and (2) search-&-test engine. The program transformer is implemented as a Clang tool built with LibTooling\textsuperscript{2}, a library for writing standalone Clang tools. This module analyzes the Abstract Syntax Tree (AST) of C/C++ programs to perform source-to-source transformation via a Clang rewriter buffer. The search-&-test engine is implemented in Python. The search component divides the program (based on functions, loops, blocks, and lines) to perform a systematic search to isolate the code region responsible for the observed numerical inconsistency. The testing component launches the program transformer module to increase the precision of the specified code regions, compiles the transformed program and evaluates it on inputs known to cause numerical inconsistencies to determine whether the problem persists. \textsc{pLiner} is publicly available on GitHub.\textsuperscript{6}

V. EXPERIMENTAL EVALUATION

Our evaluation aims to answer the following questions:

\textbf{RQ1} How effective is \textsc{pLiner} at isolating code responsible for compiler-induced numerical inconsistencies?

\textbf{RQ2} How does \textsc{pLiner} compare to the state of the art in finding the root cause of numerical errors?

A. Experimental Setup

In the evaluation, we use a set of 100 floating-point synthetic programs, the SNU NPB Suite\textsuperscript{26} (a set of the NAS Parallel Benchmarks (NPB)\textsuperscript{5} implemented in C, OpenMP C, and OpenCL), and a real-world case of compiler-induced inconsistency in a hydrodynamics application—Laghos, [1, 11]. All programs use uniform double precision. The program transformer emulates reals via \texttt{long double} precision. The floating-point synthetic programs were automatically generated by \textsc{Varity}\textsuperscript{4, 17}, a framework to generate random floating-point programs that produce numerical variations when compiled with different optimization levels. We compare \textsc{pLiner} with the baseline \textsc{Herbgrind}\textsuperscript{23}, the state-of-the-art tool to identify root causes of floating-point errors. Below we provide more details on our experiment with \textsc{Varity} programs and \textsc{Herbgrind}.

\textbf{Varity Programs}. First, we used \textsc{Varity} to generate 50 floating-point programs that yield inconsistent results when compiled by IBM XL C/C++ compiler (\texttt{xlc}, Version: 16.01) with -O3 optimization level in contrast to -O0. These 50 programs were generated and tested on a workstation with 160 POWER8 2.06GHz CPUs and 250GB RAM. Second, we used \textsc{Varity} to generate 50 programs on a workstation with 40 Intel(R) Xeon(R) 2.20GHz CPUs and 250GB RAM. These 50 programs produce inconsistent results when compiled by the GCC compiler (Version: 5.4) on the Intel platform with -O3 -ffast-math options in contrast to GCC -O0.

All the test programs are written in C. In addition to the source code, each program is associated with a test input, i.e., a set of floating-point numbers that induce a numerical inconsistency in the program. When executed with the given input, the program will produce inconsistent results.

Each program consists of two functions: \texttt{main} and \texttt{compute}. Function \texttt{main} transforms the program arguments passed from the command line into floating-point values (occasionally to integer values for a few arguments), and calls function \texttt{compute} with all floating-point (and integer) values as parameters. Function \texttt{compute} performs computation over the parameters and outputs a single floating-point number in double precision. The computation involves very intensive floating-point arithmetic operations and math library calls such as \texttt{sqrt} and \texttt{sin}. Moreover, there can be multiple if-else and for-loop statements in function \texttt{compute}. Loop iterations are bounded by an integer parameter. Table II presents size characteristics of the \textsc{Varity} programs used in our evaluation: number of lines of code (LOC), floating-point operations, parameters, conditionals, and loops.

In our experiment with \textsc{Varity} programs, \textsc{pLiner} compiles each transformed program both with -O0 and -O3 (or -O3 -ffast-math), and compares their results. If the results match, then the transformed program is considered free of the numerical inconsistency.

\textbf{Baseline: \textsc{Herbgrind}}. To the best of our knowledge, \textsc{pLiner} is the first tool that identifies the origin of compiler-induced numerical inconsistencies at a fine granularity, i.e., lines of code. However, tools that identify the root cause of floating-

<table>
<thead>
<tr>
<th>LOC</th>
<th>#FP Operations</th>
<th>#Parameters</th>
<th>#Stmts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Arithmetics</td>
<td>Parameters</td>
<td>Loads</td>
</tr>
<tr>
<td>average</td>
<td>48.2</td>
<td>17.8</td>
<td>5.4</td>
</tr>
<tr>
<td>min</td>
<td>29</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>max</td>
<td>83</td>
<td>48</td>
<td>20</td>
</tr>
</tbody>
</table>

\textsuperscript{5}The source code of the given program is normalized so that there are no multiple BBs in a single line.

\textsuperscript{6}https://github.com/LLNL/pLiner
In these operations, by the two compiler optimization levels. As shown, the first results and column "Output" shows the specific outputs given and 11 out of 13 programs, the inconsistencies examined the root causes of the numerical inconsistencies. We found that, in 11 out of 13 programs, the inconsistencies in the programs. To understand why precision enhancement is insufficient, we manually examined the root causes of the numerical inconsistencies. We observed that, in 11 out of 13 programs, the inconsistencies are related to operations on signed zeros and NaNs.

Table IV shows examples in which PLINER failed to isolate the origin of the compiler-induced inconsistency that PLINER failed to isolate. The operation "atan2" violates the IEEE 754 rules while -00 in both compilers yields IEEE 754 compatible results.

Precision enhancement cannot fix the above inconsistencies because +0, -0, and NaN are fixed floating-point special numbers irrespective of precision, and the operations on floating-point special numbers follow IEEE 754 rules. In the last row of Table IV, -00 and -03 of the xlc compiler produce inconsistent results in a call to the math library atan2. Specifically, in the computation of atan2's first parameter, -00 and -03 generate inf and NaN, respectively. Precision enhancement cannot fix the above inconsistencies because +0, -0, and NaN are fixed floating-point special numbers irrespective of precision, and the operations on floating-point special numbers follow IEEE 754 rules. In the last row of Table IV, -00 and -03 of the xlc compiler produce inconsistent results in a call to the math library atan2. Specifically, in the computation of atan2's first parameter, -00 and -03 generate inf and NaN, respectively. Precision enhancement cannot fix the above inconsistencies because +0, -0, and NaN are fixed floating-point special numbers irrespective of precision, and the operations on floating-point special numbers follow IEEE 754 rules. In the last row of Table IV, -00 and -03 of the xlc compiler produce inconsistent results in a call to the math library atan2. Specifically, in the computation of atan2's first parameter, -00 and -03 generate inf and NaN, respectively. Precision enhancement cannot fix the above inconsistencies because +0, -0, and NaN are fixed floating-point special numbers irrespective of precision, and the operations on floating-point special numbers follow IEEE 754 rules. In the last row of Table IV, -00 and -03 of the xlc compiler produce inconsistent results in a call to the math library atan2. Specifically, in the computation of atan2's first parameter, -00 and -03 generate inf and NaN, respectively.
TABLE V: Compiler-Induced Inconsistencies in NPB and PLINER Results. The inconsistencies in NPB programs appear when compiled with gcc -O3 --ffast-math in comparison to gcc -O0. Column “Epsilon” indicates the error threshold used in the result verification process, which has been reduced intentionally. Column “LOC” shows the number of lines of code for each program. Column “Isolated Region” identifies the code region PLINER identified as the origin of the inconsistency. Column “#Trans” denotes the number of transformations PLINER performed to isolate the code region. Lastly, the total running time of PLINER is presented in Column “Time”.

<table>
<thead>
<tr>
<th>Program</th>
<th>Input Class</th>
<th>Epsilon</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>B</td>
<td>3.0e-14</td>
<td>900</td>
</tr>
<tr>
<td>SP</td>
<td>A</td>
<td>1.2e-11</td>
<td>3242</td>
</tr>
<tr>
<td>SP</td>
<td>B</td>
<td>1.0e-12</td>
<td>3242</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Isolated Region</th>
<th>File</th>
<th>Function</th>
<th>Line(s)</th>
<th>#Trans</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>g.c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ninvr.c</td>
<td></td>
<td>sparse</td>
<td>-</td>
<td>16</td>
<td>31m43s</td>
</tr>
<tr>
<td>ninvr</td>
<td></td>
<td></td>
<td>59</td>
<td>23</td>
<td>11m07s</td>
</tr>
<tr>
<td>exact_solution.c</td>
<td></td>
<td>exact_solution</td>
<td>44 - 47</td>
<td>17</td>
<td>39m22s</td>
</tr>
</tbody>
</table>

C. Comparing PLINER with Herbgrind on Synthetic Programs

While we evaluated the effectiveness of PLINER on 50 programs on the Power8 platform and 50 programs on the Intel platform, we only compared PLINER to Herbgrind on the 50 programs on Intel —unfortunately, Herbgrind failed to install on our Power8 system, and its authors confirmed that Power8 is not officially supported. The 50 programs on the Intel platform produce inconsistent results when compiled with gcc -O3 --ffast-math compared to gcc -O0. We use Herbgrind to detect floating-point errors for the executable file of each program compiled with gcc -O3 --ffast-math.

Herbgrind only reported the root cause of errors for 5 out of 50 programs—in three programs, Herbgrind reports the same lines as PLINER; in one program, Herbgrind reports a different line than PLINER; and in one program, Herbgrind reports a line while PLINER fails to isolate the origin of the inconsistency. For the two programs for which Herbgrind and PLINER disagree, we manually transformed the operations on the line that Herbgrind reported to long double precision and both transformations failed to remove the inconsistencies. The line that PLINER isolated, however, was shown to be the origin of the inconsistency; the inconsistency disappears when the line is transformed to high precision. For the program for which PLINER does not report a line, we found that the reason was signed zeros, as discussed in Section V-B.

In summary, Herbgrind successfully isolated the origin of compiler-induced inconsistencies in 3 programs. We believe that Herbgrind cannot identify all floating-point errors because it only traces a limited number of operations for root cause identification. Also, Herbgrind performs its analysis at the binary level where the compiler optimizations have been performed, and therefore is not able to directly identify compiler-induced problems—Herbgrind has no notion of the behavior of the unoptimized code.

Finding 2: PLINER outperforms Herbgrind (the state-of-the-art tool to identify root-causes of floating-point errors) in isolating compiler-induced numerical inconsistencies in the synthetic programs. PLINER isolated the origin of inconsistencies in 87 out of 100 programs while Herbgrind only isolated the origin in 3 programs.

D. SNU NPB Suite

We investigated the serial C version of the SNU NPB Suite [26] (Version 3.3) to identify compiler-induced inconsistencies. The suite consists of 10 programs—BT, CG, DC, EP, FT, IS, LU, MG, SP, and UA, and is derived from the serial Fortran code in “NPB3.3-SE” developed by NAS [5]. Each program includes 6 input classes—S, W, A, B, C, D. We compiled all NPB programs both with gcc -O3 --ffast-math and gcc -O0 on the Intel platform described earlier, and did not find any inconsistencies between the two compiler options for any of the input classes.

However, we observed that 6 out of the 10 NPB programs use an error threshold in their verification routine—BT, CG, FT, LU, MG, and SP. Specifically, the verification routine computes the relative error of the program results comparing to the ground truth for an input class; if the relative error is within the specified error threshold \( \epsilon \), then the verification fails, otherwise the verification is successful. Using the original error thresholds, the verification is successful for both the programs compiled with gcc -O3 --ffast-math and the programs compiled with gcc -O0.

In an attempt to trigger compiler-induced inconsistencies in the NPB programs, we tested various error-threshold values for each of the 6 programs. In other words, we searched for a smaller error-threshold for which the program compiled with gcc -O3 --ffast-math failed to verify whereas the program compiled with gcc -O0 verified successfully. Specifically, we iterated through error-threshold values starting with the original error threshold (1.0e-08 in MG, LU, BT and SP; 1.0e-10 in CG; 1.0e-12 in FT) down to 1.0e-16 by reducing the exponent by 1 each time. We tested each program with input classes A, B and C, and identified three compiler-induced numerical inconsistencies between gcc -O3 --ffast-math and gcc -O0 with the modified error thresholds.

Specifically, when the error threshold is 1.0e-14 or 1.0e-15, program CG with input class B (abbreviated as CG.B) produces an inconsistent result when compiled with gcc -O3 --ffast-math compared to gcc -O0. We also observed an inconsistency for SP with input class A (abbreviated as SPA) when the error threshold is 1.0e-11 or 1.0e-12 and SP with input class B (abbreviated as SP.B) when the error threshold is 1.0e-12. We manually maximized the error
threshold for each inconsistency case, and the results are shown in Column “Epsilon” in Table V.

We evaluated PLINER on the 3 NPB numerical inconsistencies. As described on the right of Table V, PLINER successfully isolated a code region for each inconsistency in 10 to 40 minutes. PLINER isolated a function in CG.B out of 7 functions in file cg.c (900 LOC), one line in file ninvr.c of SPA, and an assignment statement broken into 4 lines in file exact_solution.c of SP.B out of 17 files (3242 LOC).

Finding 3: We triggered three compiler-induced inconsistencies in the SNU NPB Suite—CG.B, SPA and SP.B by modifying the error threshold used in the verification routine. PLINER successfully isolated the origin of numerical inconsistency for each program within minutes.

E. Laghos Case Study

Finally, we applied PLINER to the Laghos [1] case described in Sections I and II. PLINER successfully isolated the root-cause line of code in the function where the numerical inconsistency originates within 19 minutes. In other words, after transforming from double to long double the precision of the operation at the line pinpointed by PLINER, the numerical inconsistency disappears. The transformed program produces 129664.9230609672 for the computed |e| in contrast to 144174.9336610391 that is yielded by the original program compiled by xlc -O3. Using clang, gcc and xlc -O2 as the ground truth (|e| = 129664.9230611104), the transformed program satisfies the accuracy threshold 10⁻¹² that is given to PLINER for isolation.

We created a test case to showcase this inconsistency in Laghos (see Figure 6), where Line 6 is isolated by PLINER. We use input values for variables gradv11, gradv00, gradv10 from program profiling. Compiling the test with xlc with optimization level -O2 yields output -1.3507 while compiling the same test using xlc -O3 outputs -inf. By examining the assembly code of Laghos, we found that xlc -O3 converts the division operation from Line 6 to a reciprocal operation and a multiplication. This optimization introduces a floating-point exception because the value of gradv10 is a subnormal number that is close to zero and the reciprocal of 2*gradv10 overflows. We propose two possible fixes: (1) using higher precision such as long double for the operations; or (2) using a threshold \( \sigma \) instead of 0 on Line 127 in Figure 1 to exclude the floating-point values whose reciprocal overflow.

Finding 4: PLINER successfully isolated a line that the compiler-induced inconsistency originates in the real-world application, Laghos. We manually transformed the isolated line to a test case. It shows that the inconsistency appears because xlc -O3 converts a division operation to a reciprocal operation and a multiplication which causes a floating-point exception in program execution.

F. Discussion and Threats to Validity

Compiler-induced inconsistencies are not uncommon in the real world. Besides the Laghos case that is discussed in this paper, there are other compiler inconsistency cases reported in the literature [7, 24]. We tried to use the MFEM case reported in [7], but we were not able to reproduce the numerical inconsistency in our system. As code is run in heterogeneous architectures, we believe the existence of compiler-induced inconsistencies will become even more frequent among compilers for different architectures.

In addition to the Laghos case and SNU NPB Suite programs, PLINER was evaluated on 100 synthetic programs to investigate its applicability to many different compiler inconsistencies. In program generation, Varity explores many combinations of floating-point operations (including math function operations), conditionals and loops. These programming constructs are common in HPC codes. The program and the input space are very large, and it is unlikely that two Varity programs are equivalent. As shown in Table II, the code characteristics (LOC, number of floating-point operations and parameters, etc.) of the programs vary widely. Moreover, to avoid pure randomness, the programs are generated based on the code structure of existing benchmarks such as the NAS Parallel Benchmarks [5]. We believe that the Varity programs used in our evaluation include a representative set of code patterns that expose compiler-induced inconsistencies, which are likely to occur in real-world programs.

PLINER may not always be effective at isolating code when the computation involves inf, -inf, 0, -0 and NaNs. Compiler optimizations that violate the IEEE 754 rules could yield a different result for operations on the above values, such as a comparison to a NaN. From our observation in the 100 synthetic programs across two architectures and compilers, there are 13% such cases that PLINER cannot handle.

PLINER is currently implemented to transform float/double precision uniformly to a higher precision type that the user specifies, such as long double in our evaluation. PLINER provides a command-line argument for the user to specify what precision should be used to emulate reals. For code that originally uses long double or higher precision types, we recommend to use float128 or arbitrary precision. We leave it, however, to the user to make sure that the specified precision type is available. Finally, PLINER does not explore mixed-precision configurations, such as

```c
#include<stdio.h>
int main()
{

double gradv11=-3.935e-309;

double gradv00=1.430e-309;

double gradv10=1.986e-309;

double zeta = (gradv11−gradv00)/(2*gradv10);

printf(“zeta = %g\n”, zeta);

return 0;
}
```

Fig. 6: Test case extracted from Laghos.

1. #include<stdio.h>
2. int main()
{
3. double gradv11=-3.935e-309;
4. double gradv00=1.430e-309;
5. double gradv10=1.986e-309;
6. double zeta = (gradv11−gradv00)/(2*gradv10);
7. printf(“zeta = %g\n”, zeta);
8. return 0;
9. }
enhancing precision in the order of float, double, and long double. If the program uses mixed precision, PLINER uniformly transforms all floating-point types to the precision type used to emulate reals.

VI. RELATED WORK

Floating-Point Testing. FLiT [7, 24] investigates the impact of compiler optimizations on floating-point programs by testing various compiler flags and identifying result variations—it supports several compilers, such as gcc, clang, the Intel compiler, and CUDA. FLiT can detect compiler-induced numerical inconsistencies that impact files/functions, but unlike PLINER it cannot detect inconsistencies at a fine granularity, such as code lines. It is worth mentioning that if the origin of an inconsistency is a very small function, and FLiT identifies that function, FLiT would be as effective as PLINER; however, in our experience with several realistic HPC programs, functions that induce inconsistencies can be large, in which case PLINER can identify the exact line that originates the inconsistency. Finally, FLiT and PLINER could be used together, where FLiT could first find functions/files that induce inconsistencies and PLINER could be used to identify at a fine granularity which specific regions are the origin of the problem in those files/functions identified by FLiT.

Herbgrind [23] performs an error analysis on the binary code of a floating-point program to find the root causes of floating-point errors. Herbgrind instruments the binaries using Valgrind [3] to perform each floating-point operation in both the original and higher precision to calculate the error. Function calls, conditional branches, and conversions from floating-point values to integers are referred as spots. The error is dynamically traced through operations that have an influence on a spot. However, for the sake of performance, Herbgrind only traces a limited number of operations (5 by default), and therefore is not able to report all problematic operations. Moreover, Herbgrind identifies root causes of floating-point errors at the binary level, and thus it is difficult to recognize the errors that are induced by compiler optimizations. In our evaluation, Herbgrind is not as effective as PLINER at identifying the origin of compiler-induced inconsistencies.

Similar to Herbgrind, both FpDebug [8] and Verrou [12] use Valgrind to detect floating-point errors. FpDebug also performs high-precision execution in the shadow to calculate errors. Verrou perturbs the floating-point rounding modes to generate an output variation for each run of the instrumented code. A statistical analysis on the output variations can be performed to estimate the error in the result. However, Verrou does not report the root causes of the floating-point errors.

Lee et al. [19] identify output variations by executing the program using multiple copies of the floating-point values varied by rounding errors. Fu et al. [13] consider the backward error besides the forward error, and automate the calculation of backward error and condition number, which can be used to measure how sensitive the code is to errors in the input and rounding errors in the finite precision arithmetic. Bao and Zhang [6] monitor the program execution on-the-fly to detect and track cancellation errors until their propagation is cut off or a critical execution point such as a predicate is reached. FPDiff [27] automatically identifies equivalence classes of function synonyms from a chosen ensemble of numerical libraries and performs testing to discover points of divergence. Moreover, some efforts [10, 15, 28, 30] have been made to generate high error-inducing inputs which provide a conservative estimation of the error and, more importantly, testing inputs for code debugging and optimization.

Floating-Point Program Transformation. AutoRNP [28] detects floating-point errors by approximating the condition number of the program to guide a search for high error-inducing inputs, and provides a patch that satisfies a given accuracy threshold for these inputs. However, AutoRNP is limited to transforming numerical programs with one floating-point input. It is therefore not applicable to our problem in which most applications and tests have multiple floating-point inputs. Herbie [20] uses a database of rewrite rules and performs series expansion to rewrite floating-point expressions to improve floating-point accuracy. Rewrite candidates are generated for different input regions and combined to produce a single program that improves accuracy across input regions. Herbie complements our work by providing a rewrite suggestion for the code lines that PLINER isolates.

A large portion of transformation work focuses on mixed precision tuning (e.g., [9, 14, 18, 21, 22, 25]). Instead of using one precision type uniformly, the program is transformed to use mixed precision with the aim to improve performance.

VII. CONCLUSION

Diagnosing compiler-induced floating-point inconsistencies can be difficult and can significantly impact the productivity of programmers of HPC systems. With HPC systems moving to heterogeneous architectures with multiple compilers and platforms, it becomes crucial to have diagnosing tools to isolate such issues quickly. Our new framework, PLINER is able pinpoint lines of code that originate compiler-induced variability. Our approach uses a novel method that combines precision enhancements at different levels of code granularity and an efficient guided search to identify locations that can be affected by numerical inconsistencies. In the course of developing PLINER, we have used it to isolate floating-point inconsistencies that affected real-world applications. In our evaluation with 100 synthetic programs, PLINER isolates the affected lines 87% of the time.

ACKNOWLEDGMENT

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-808938), the National Science Foundation under award CCF-1750983, and the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under award DE-SC0020286.


Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED
We run Laghos v1.1 on a PowerNV 8335-GCA workstation compiled with IBM XL C/C++ compiler v16.01; we run Varity tests v0.1.0 on both a PowerNV 8335-GCA workstation with IBM XL C/C++ compiler v16.01 and an Intel(R) Xeon(R) workstation with GCC compiler v5.4.

ARTIFACT AVAILABILITY

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: No author-created artifacts are proprietary.

Author-Created or Modified Artifacts:

Artifact name: pLiner

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: PowerNV 8335-GCA workstation with 160 POWER8 2.06GHz CPUs and 250GB RAM; a second workstation with 40 Intel(R) Xeon(R) 2.20GHz CPUs and 250GB RAM

Operating systems and versions: Red Hat Enterprise Linux Server release 7.6 running Linux kernel 3.10.0-957.21.3.1chaos.ch6.ppc64le

Compilers and versions: IBM XL C/C++ compiler v16.01; gcc v5.4; LLVM/clang v9.0.1

Applications and versions: Laghos v1.1; NAS Parallel Benchmarks v3.3 - C version

Libraries and versions: JSON for Modern C++ v3.5.0

Key algorithms: bisection algorithm

Input datasets and versions: Varity Tests v0.1.0