Fixing Numerical Inconsistencies Due to Different Optimization Levels in Scientific Applications

Dr. Md. Alam Hossain
Dept. of Computer Science and Engineering
Jashore University of Science and Technology
Jashore-7408, Bangladesh
alam@just.edu.bd

Md. Istiaque Hossain
Dept. of Computer Science and Engineering
Jashore University of Science and Technology
Jashore-7408, Bangladesh
istiaquehossainshubho@gmail.com

Mehedi Hasan
Dept. of Computer Science and Engineering
Jashore University of Science and Technology
Jashore-7408, Bangladesh
imehedi01@gmail.com

***Abstract*—Due to the usage of approximate representations of numbers, such as floating-point, numerical errors are notoriously difficult to diagnose and fix. The representation of floating-point types is infamous for its complexity. PLINER is a program that finds code lines that cause compiler-induced variability, employs a revolutionary method to improve floating-point precision, and uses a guided search to locate areas where numerical inconsistencies exist in order to make high performance on programs using with long double variables. But in some cases PLINER does not perform with double variables accurately. So, this paper is going to demonstrate the usage of double variables on various programs instead of using long double variables that will enhance the performance as well as make the programs faster in order to reduce the time.And this paper also shows on how to make a performance gain using double variables rather than using long double variables on different types of programs that contain numerical inconsistencies.**

I. Introduction

Floating-point arithmetic is widely employed in numerical software, although it has serious implications for program dependability and performance. Floating-point arithmetic is used in practically all scientific and technical applications, and it is the foundation for these programs' numerical computations. A floating point number is a regularly used representation of a real number. [3] Because of their nature, floating-point computations are intrinsically inaccurate. In order to talk about numerical inconsistency, we are asked a lot about the most common numerical issues. Which numerical flaws are the most common? How common are these bugs? Are there any existing technologies that can detect them? What are the methods for resolving such issues? Are there any existing software repair tools that can correct them? [3] Because essential decisions are made based on the numerical outputs of these programs, their repeat-ability and numerical consistency are critical. Floating-point computation is notoriously intuitive, and floating-point applications suffer from a slew of accuracy concerns, including a high risk of large rounding mistakes and improperly handled numerical exceptions. Because HPC systems can run on multiple architectures in the same system and different architectures' compilers can generate different floating-point programs, numerical consistency and repeat-ability become a problem. [2] Compilers have a lot of leeway when it comes to writing code for floating-point programs because floating-point is frequently under-defined in language specifications. When numerous compilers optimize the same code, the numerical result varies. Additionally, the usage of several optimization levels by the compiler results in numerical discrepancies. Hence, this paper is talking about numerical discrepancies caused by the compiler. When discrepancies in floating point applications are caused by compilers, programmers spend a lot of time attempting to figure out what's wrong.

II. Motivation

A. Dot Product

Dot products are vastly used in solar systems as well as the game industry and plagiarism process. Solar panels must be precisely installed so that the roof's inclination and direction to the sun can produce the maximum amount of electrical power in the solar panels. To receive the most energy from solar panels, their direction must be precise, and due to numerical irregularities in dot product programs, this might be difficult to determine. And in the game industry, dot products are vital tools for measuring direction, projecting vectors and measuring planes. Also in plagiarism, dot product programs are used as vector space model. These motivate us to work with these issues in order to reduce numerical inconsistencies and inspire us to show a good performance gain.

A dot product over two arrays of floating-point numbers produces 16877216 when compiled with g++ -O2 and 16877222 when compiled with g++ -O3.

gcc	**-02**	**16877222.0000**
gcc	-03	16877216.0000

Fig. 1. Dot Product comparison

B. Laghos Program

Scientists at the Lawrence Livermore National Laboratory (LLNL) were looking for higher optimizations provided by the IBM compiler, xlc, in the development of a new hydrodynamics application, Laghos(LAGrangian High-Order Solver is a mini-app that solves the time-dependent Euler equations of compressible gas dynamics in a moving Lagrangian frame). [1]

xlc	-O1	129664.9230611104
xlc	-O2	129664.9230611104
xlc	-O3	144174.9336610391

Fig. 2. Laghos comparison

III. Related Works

A. Real-World Numerical Bug Characteristics [3]

Due to the usage of approximate representations of numbers, such as floating-point, numerical errors are notoriously difficult to diagnose and fix. Which numerical flaws are the most common? How common are these pests? Are there any existing technologies that can detect them? What are the methods for resolving such issues? Are there any existing software repair tools that can correct them? Numerical bugs could be classified as accuracy bugs, special-value bugs, convergence bugs, or correctness bugs, according to our proposal. In the data set showing that correctness bugs are the most common (37%) followed by special-value bugs (28%) and convergence bugs (21%).

B. Floating-Point Precision Tuning HiFPTuner [2]

This method formulates precision tuning as an optimization problem and dynamically searches through alternative precision to reach a local optimum, using a static performance and accuracy model. This paper introduce a community detection approach for floating-point variables in this research, and use combination with dynamic precision adjustment to increase the search's scalability as well as the quality of the recommended solutions. A comprehensive search for floating-point precision adjustment is presented. Based on the dependence analysis for the target program, this paper develops a variable community hierarchy that organizes dependent variables that may need the same level of accuracy.

C. Isolating Lines of Floating-Point Code [1]

PLINER is a program that can automatically identify code lines that cause compiler-induced variability, employs a revolutionary method to improve floating-point precision at various levels of code granularity, as well as a guided search to find sites where numerical discrepancies exist.

PLINER is a novel method for automatically isolating the lines of code that generate numerical discrepancies in floating-point applications due to compiler-induced inconsistencies. PLINER is made up of a precision transformer and a search engine. To improve precision for a succession of code sections, the search engine repeatedly separates the code and runs the transformer. This paper introduce about recompiling the changed program to evaluate result consistency, and if it provides consistent results presuming the source of the compiler-induced numerical inconsistency is included in the modified code sequence.

In summary, this study makes the following contributions:

- By combining precision enhancement with hierarchical search, this paper develops a method for finding the cause of compiler-induced numerical differences in floating-point applications.
- This paper provides a PLINER tool that implements the method as an extension to the clang/LLVM compiler. PLINER is a tool for dealing with fine-grained numerical discrepancies caused by compilers (lines of code).
- PLINER is successful at identifying discrepancies caused by real-world compilers.
- PLINER is compared to other applications for discovering the origins of floating-point mistakes, and it is shown that PLINER outperforms them in isolating compiler-induced numerical disparities.

Previous work enhances the precision of the variables or regions, it transforms both float and double variables to long double variables but does not transform float variables to double variables. As it transforms variables to long double directly instead of using double variables, it takes more time to execute the inconsistent programs. That's this paper we showed a good work with in this area of limitations.

IV. Methodology

This section describes the working methodology of our proposed solution.
This paper is going to deal with applying a search engine in compiler induced inconsistency program and this search engine uses hierarchical code isolation techniques to identify the functions, loops, blocks and lines. It transforms the variables in those isolated areas by doing precision enhancement and then compile and test according input data. Hierarchical code isolation identifies functions that are affected by numerical inconsistencies. After that this paper shows to automatically identify loops inside those functions and basic blocks inside those loops. Single line is the last part of the hierarchical code isolation that transforms the precision of the variables in those lines.And end of the paper, we are going to show a performance gain using double variables.

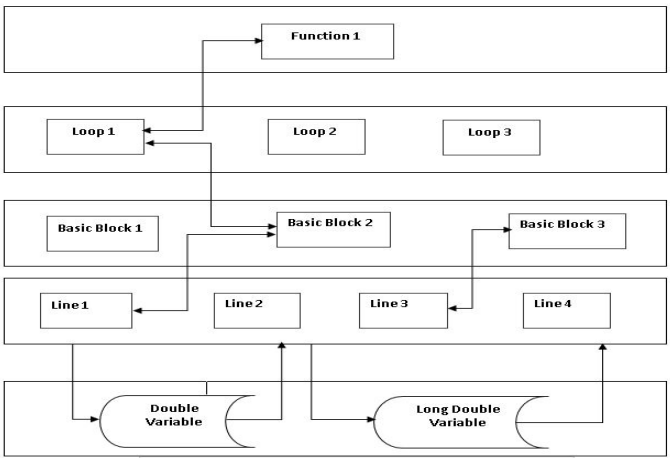

Fig. 3. Heirarchical Isolation

A. Function Transformation

A program consists of multiple functions. To isolate region, loops, lines, first work is to isolate the function. A function consists of parameters and return statements. To isolate the function, next step is to identify the parameters and return statement.

1: **if** $parameter\_type \leftarrow float$ **then**
2: $parameter\_type \leftarrow double$
3: **else if** $parameter\_type \leftarrow double$ **then**
4: $parameter\_type \leftarrow longdouble$
5: **end if**

B. Region transformation

A region is a group of basic blocks having a single point of entry and exit from another block outside the area. Here it is defined a new variable with increased precision v_H for each variable toReplace at the region's entrance, which v is initialized with the value of its equivalent. Within the area v, this paper will show how to replace each usage of such variables with its higher-precision variant. Finally, it is assigned the value of back to its v_H equivalent variable at the region's v_H exit.

1: $typeTransto \leftarrow double$
2: **for** var in varlist **do**
3: **if** $var\_type \leftarrow float$ **then**
4: $var\_type \leftarrow double$
5: $v\_H = var\_name + "\_H"$
6: **else if** $var\_type \leftarrow double$ **then**
7: $var\_type \leftarrow longdouble$
8: $v\_H = var\_name + "\_H"$
9: **end if**
10: **end for**

C. Statement Transformation

A region consists of many statements including while, for, if and switch statements. This paper helps to dynamically type casting these statements to enhance their precision for both double and long double.

1: **if** $type \leftarrow Whilestmt$ **then**
2: $type \leftarrow dyn\_cast$
3: **else if** $type \leftarrow ForStmt$ **then**
4: $type \leftarrow dyn\_cast$
5: **else if** $type \leftarrow SwitchStmt$ **then**
6: $type \leftarrow dyn\_cast$
7: **end if**

V. EXPERIMENTAL EVALUATION

A. Hardware Specification

In order to complete this process, what are used for experiments and algorithms are Intel Core i5-6200 CPU @ 2.40 GHz x64-based processor with 8 GB RAM.

B. Algoritms

To make this task efficiently, hierarchical code isolation algorithms with enhance precision techniques are used, and this algorithm helps to identify the lines of codes in hierarchical order from function to basic blocks.

C. Software

Docker container, code-blocks, visual studio as software requirements.

VI. EXPERIMENTAL RESULTS AND COMPARISON

This paper works on 5 different programs LU Decomposition, Bisection Method, Adaline algorithm, Dot Product and rounding number for experimental purpose.At first, we find out the numerical inconsistent areas by applying search and test engine and then change those areas and compare the time complexity for both double and long double variables, and show the performance gain.

A. LU Decomposition [16]

After applying search & test engine to this program,line 18, 23, 25, 65, 105, 394, 404 & 405 need to be enhanced in order to remove the inconsistencies.
We apply this program on a matrix of containing 500 row and 500 columns and found time variation between normal and enhanced programs.

Double variable programs take **910ms** and Long Double variable programs takes **950ms** to run a single iteration.
After applying search & test engine both for double and long double variables it takes 10-15ns for each portion. So we have **25-30ms** performance gain.

Double Time	Long Double Time	Search and Test Time	Performance Gain
910ms	950ms	10-15ns	3.15%

B. Adaline Algorithm [17]

After applying search & test engine to this program,line 46, 47, 73, 199, 353, 354,& 355 need to be enhanced in order to remove the inconsistencies.

Double variable programs take **85-90ms** and Long Double variable programs takes **125-130ms** to run a single iteration.
After applying search & test engine both for double and long double variables it takes 10ms for each iteration. So we have **30-35ms** performance gain.

Double Time	Long Double Time	Search and Test Time	Performance Gain
85-90ms	125-130ms	10ms	26.92%

C. Bisection Method [18]

After applying search & test engine to this program,line 83, 84, 85, 154 & 155 need to be enhanced in order to remove the inconsistencies.

Double variable programs take **20ms** and Long Double variable programs takes **20ms** to run a single iteration.
After applying search & test engine for both double and long double variables, it takes 15-20ns for each section. So we have **0ms** performance gain.

Double Time	Long Double Time	Search and Test Time	Performance Gain
20ms	20ms	15-20ns	0%

D. Dot Product

After applying search & test engine to this program,line 6, 7 & 19 need to be enhanced in order to remove the inconsistencies.

Double variable programs take **30ms** and Long Double variable programs takes **50ms** to run a single iteration.
After applying search & test engine for both double and long double variables, it takes 10-15ms for each section. So we have **5-10ms** performance gain.

Double Time	Long Double Time	Search and Test Time	Performance Gain
30ms	50ms	10-15ms	10%

E. Rounding Number

After applying search & test engine to this program,line 6, 8, 9 & 10 need to be enhanced in order to remove the inconsistencies.

Double variable programs take **60ms** and Long Double variable programs take **90ms** to run a single iteration.
After applying search & test engine for both double and long double variables, it takes 15-20ms for each section. So we have **10-15ms** performance gain.

Double Time	Long Double Time	Search and Test Time	Performance Gain
60ms	90ms	10-15ms	16.67%

In this paper, the numerically inconsistent programs are changed by enhancing the precision of the variables from long double to double. During this implementation, variations of execution time between the double and long double programs have found out.

According to the results, it is clearly shown the time difference between the double and long double programs. In most of the cases, the use of double variable in programs run quickly than the use of long double programs.

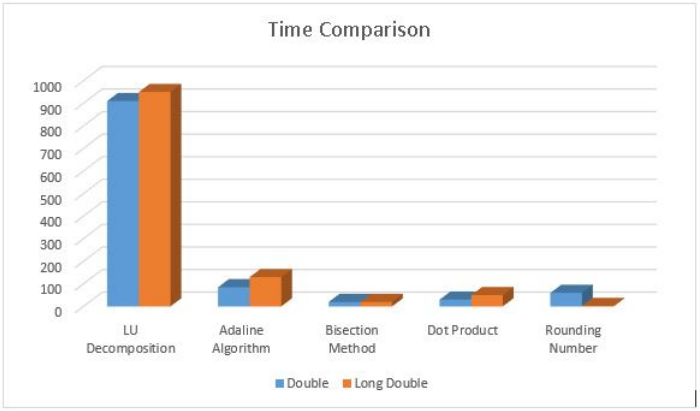

Fig. 4. Time Comparison Graph

VII. Conclusions and Future Work

Floating point is heavily used in scientific applications, and small variations of floating point numbers can cause a vital error. The usage of long double variables make the programs large and take more time to execute. So this paper prefer to using double variables rather than using long double variables to make the programs faster and show how the performance gain deals with these inconsistencies. In Future, this paper will help to work more in this field on the limitations of numerical inconsistencies and contribute more in those floating point areas.

References

[1] Guo, Hui, Ignacio Laguna, and Cindy Rubio-González. "pLiner: isolating lines of floating-point code for compiler-induced variability." SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2020.

[2] Guo, Hui, and Cindy Rubio-González. "Exploiting community structure for floating-point precision tuning." Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2018.

[3] Di Franco, Anthony, Hui Guo, and Cindy Rubio-González. "A comprehensive study of real-world numerical bug characteristics." 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2017.

[4] Variety, floating-point program random generator. https://github.com/llnl/varity, Accessed: 2020-08-25.

[5] Bao, Tao, and Xiangyu Zhang. "On-the-fly detection of instability problems in floating-point program execution." Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications. 2013.

[6] The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.The llvm compiler infrastructure. http://llvm.org/.

[7] Benz, Florian, Andreas Hildebrandt, and Sebastian Hack. "A dynamic program analysis to find floating-point accuracy problems." ACM SIGPLAN Notices 47.6 (2012): 453-462.

[8] GSL - GNU scientific library - GNU project - free software foundation (FSF).http://www.gnu.org/software/gsl/.

[9] M. O. Lam and J. K. Hollingsworth. Fine-grained floating-point precision analysis.The International Journal of High Performance Computing Applications, 2016.

[10] Michael Bentley, Ian Briggs, Ganesh Gopalakrishnan, Dong H. Ahn,Ignacio Laguna, Gregory L. Lee, and Holger E. Jones. Multi-Level Analysis of Compiler-Induced Variability and Performance Tradeoffs. In Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC), pages 61–72, 2019.

[11] Veselin A Dobrev, Tzanio V Kolev, and Robert N Rieben. High-order curvilinear finite element methods for lagrangian hydrodynamics. SIAM Journal on Scientific Computing, 34:B606–B641, 2012.

[12] D. Zou, R. Wang, Y. Xiong, L. Zhang, Z. Su, and H. Mei. A genetic algorithm for detecting significant floating-point inaccuracies. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), pages 529–539, 2015.

[13] W. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev. Efficient search for inputs causing high floating-point errors. In PPoPP, pages 43–52, 2014.

[14] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In PLDI, pages 77–88, 2012.

[15] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In ASPLOS, pages 329–339, 2008.

[16] Github.com, 2021, Available: https://github.com/makkader/LU-Decomposition/blob/master/ludecomposition.cpp

[17] Github.com, 2021, Available: https://github.com/TheAlgorithms/C/blob-/master/machine_learning/adalin_learning.c

[18] Github.com, 2021, Available: https://github.com/Lehmannhen/Bisection-and-Newton-method/blob/master/src/bisect_newton.c