Multi-Level Analysis of Compiler-Induced Variability and Performance Tradeoffs

Michael Bentley
Ian Briggs
Ganesh Gopalakrishnan
mbentley@cs.utah.edu
ianbriggsutah@gmail.com
ganesh@cs.utah.edu
University of Utah

Dong H. Ahn
Ignacio Laguna
Gregory L. Lee
Holger E. Jones
ahn1@llnl.gov
lagunaperalt1@llnl.gov
lee218@llnl.gov
jones19@llnl.gov
Lawrence Livermore National Laboratory

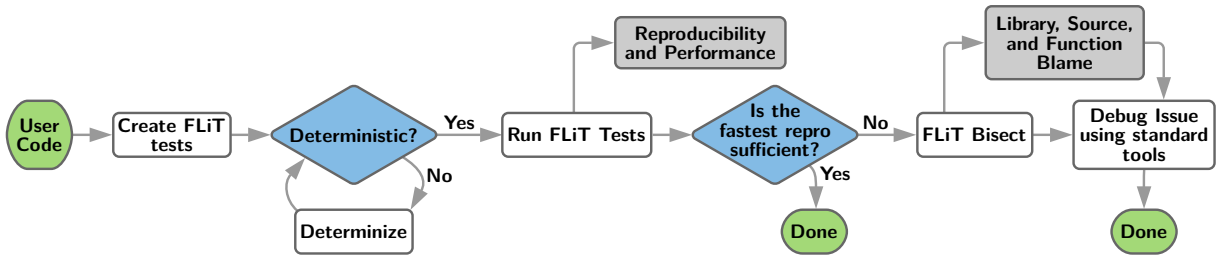


Figure 1: Multi-level workflow. Levels are (1) determine variability-inducing compilations, (2) analyze the space of reproducibility and performance, and (3) debug variability by identifying files and functions causing variability.

ABSTRACT

Successful HPC software applications are long-lived. When ported across machines and their compilers, these applications often produce different numerical results, many of which are unacceptable. Such variability is also a concern while optimizing the code more aggressively to gain performance. Efficient tools that help locate the program units (files and functions) within which most of the variability occurs are badly needed, both to plan for code ports and to root-cause errors due to variability when they happen in the field. In this work, we offer an enhanced version of the open-source testing framework FLiT to serve these roles. Key new features of FLiT include a suite of bisection algorithms that help locate the root causes of variability. Another added feature allows an analysis of the tradeoffs between performance and the degree of variability. Our new contributions also include a collection of case studies. Results on the MFEM finite-element library include variability/performance tradeoffs, and the identification of a (hitherto unknown) abnormal level of result-variability even under mild compiler optimizations. Results from studying the Laghos proxy application include identifying a significantly divergent floating-point result-variability and successful root-causing down to the problematic function over as little as 14 program executions. Finally, in an evaluation of 4,376 controlled injections of floating-point perturbations on the LULESH proxy application, we showed that the FLiT framework has 100% precision and recall in discovering the file and function locations of the injections all within an average of only 15 program executions.

HPDC '19, June 22–29, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6670-0/19/06…$15.00
https://doi.org/10.1145/3307681.3325960

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Software maintenance tools*; Object oriented frameworks; Process validation; Compilers.

KEYWORDS

debugging, compilers, code optimization, reproducibility, performance tuning

ACM Reference Format:
Michael Bentley, Ian Briggs, Ganesh Gopalakrishnan, Dong H. Ahn, Ignacio Laguna, Gregory L. Lee, and Holger E. Jones. 2019. Multi-Level Analysis of Compiler-Induced Variability and Performance Tradeoffs . In *The 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19), June 22–29, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3307681.3325960

1 INTRODUCTION

Tools and techniques that mitigate the effects of compiler-induced result-variability are increasingly important to preserve the value of our investments in scientific software. As a specific example, long-lived scientific applications must be able to take advantage

of different (or newer) machines and their compilers (as well as their optimization flags) while maintaining result integrity and achieving higher performance. Unfortunately, there are currently no techniques and tools that help designers debug field issues that arise during such code ports, especially in the context of large codebases and thousands of functions. At present, designers end up wasting their time by manually debugging field issues. Also, code that is shipped without portability testing may harbor the potential to generate unacceptably significant result variations even under standard optimization. An incident of this type was reported by designers of the Community Earth System Model (a large-scale climate simulation) [5] where the problem was noticed while porting code to a new machine. After weeks of painstaking investigations, the problem turned out to be the *introduction* of fused-multiply-add instructions by the compiler, taking advantage of this new capability offered by their target architecture. This and other incidents reported in this paper underscore the need for an ecosystem of freely available tools that can help scientific programmers. To the best of our knowledge, FLiT is the first such tool.

Definition of Reproducibility. Given the growing heterogeneity of hardware and software, one cannot always define reproducibility as achieving bitwise reproducible results. Instead, we view a reproducible computation as one that produces a result within an "acceptable range" of a trusted baseline answer. In FLiT, we rely on the application developer to provide an acceptance testing function that (indirectly) defines this range.

A Motivating Problem. Scientific HPC applications can be large and complex, often simulating physical phenomena for which expected outcomes are not known. As a result, there is a particular compilation configuration that is trusted because it has passed the test of time (*i.e.*, it is believed to be correct from the first version), and is considered the *baseline compilation configuration*.

When developers port applications to a different compiler or a new version of the same compiler, all *acceptably good* compilation configurations must deliver answers empirically close to the baseline, either based on designer experience or in a more rigorous mathematical sense, such as meeting an error norm. When results deviate from acceptable levels, support tools must help locate the issue within a short distance of the root cause.

Our Contributions.

(1) A significantly extended version of the FLiT [34] testing tool[1] that is now capable of handling real applications that are either sequential or contain deterministic OpenMP or MPI code.
(2) Results that capture how performance varies versus reproducibility on non-trivial applications.
(3) A suite of novel bisection algorithms that help identify code locations responsible for result-variability,
(4) A workflow (Figure 1) providing steps for practitioners to analyze result and performance variability.
(5) Experimental validation using real-world HPC miniature applications that include Laghos [13] and LULESH [21], and the MFEM library [1]. These studies quantitatively evaluate the effectiveness of our Bisect algorithms as well as empirically assess the real-world applicability of the workflow.

New FLiT features. The new FLiT features are:

(1) A multi-level analysis workflow supported by FLiT (Figure 1), resulting in root-cause analysis of compiler-induced result-variability down to individual source files and functions. Root-causing is achieved by FLiT's Bisect algorithms (§2).
(2) An assessment of the efficacy of Bisect on real applications and a fault injection study (§3);
(3) The results of applying FLiT, for the first time, on two real-world systems: MFEM and Laghos (§3).

Compiler-Induced Variability Example. Compiler-induced variability is widely experienced but seldom systematically solved. We provide an example to help the reader better understand the utility of a tool such as FLiT. At one stage of the development of Laghos, an open-source simulator of compressible gas dynamics [14], the project scientists were seeking higher optimizations provided by the IBM compiler, `xlc`. Moving from optimization level `-O2` to `-O3`, the ℓ_2 norm of the energy over the mesh went from 129,664.9 to 144,174.9 *in a single iteration* — an 11.2% relative difference caused merely by the optimizations. One would expect variability around 10^{-8}% or less. Also, the density of the simulated gas became negative — a physical impossibility. Even more striking was the runtime difference: from 51.5 seconds to 21.3 seconds for the first iteration, which is a speedup by a factor of 2.42. In Section 3, we describe how FLiT came to the Laghos designers' rescue.

Paper Organization and Result Highlights. In Section 2, we introduce our multi-level analysis workflow and tooling spread over three phases. The first phase identifies which compiler optimizations cause reproducibility problems. The second phase helps to analyze the performance resulting from the optimizations, thus assisting the programmer in arriving at the most performant of acceptable solutions. The third phase helps characterize which functions within the code exhibit variability under compiler optimizations, sorted by the most influential. The last phase involves our suite of bisection algorithms.

We contribute two key assumptions that help make bisection practical: (1) The *Unique Error* assumption, meaning for a particular value of variability, the set of responsible application functions is unique. This assumption frequently holds in practice, as demonstrated by our results. Without this assumption, we will have an exponential search problem to solve. (2) The *Singleton Blame Site* assumption, which means that a single file or function, by itself, causes variability. In other words, it is not necessary to have two or more files or functions to be jointly acting to induce variability. This assumption also holds in practice, as demonstrated by our results. The Bisect algorithm has a built-in dynamic verification assertion that verifies this assumption. Section 2.2 explains how these assumptions are central to achieving an overall $O(k \log(N))$ runtime complexity (for k "problematic" files/symbols) as opposed to the $O(2^N)$ complexity, if we were to relax these assumptions.

In navigating performance and reproducibility in the MFEM library (Section 3), we found that 14 of 19 examples exhibited the highest speedups with compilations that are bitwise reproducible. Two of those 14 showed bitwise reproducibility across all tested compilations. These results indicate reproducibility need not always be sacrificed for performance gains. We demonstrate our Bisect algorithm on *all* found variability-inducing compilations from MFEM

[1] This version of FLiT source code is available at https://github.com/PRUNERS/FLiT.git

to evaluate the effectiveness of Bisect and to empirically characterize the proclivity of a compiler to introduce variability. For MFEM, we provide the "best average compilation" for each compiler over the set of 19 MFEM examples, along with a rough idea of how often each compiler induces variability. Also, thanks to FLiT, we have located an unexpected result deviation in one test of MFEM which resulted in a 180% relative error under a mild compiler optimization. FLiT could root-cause this failure to a single function.

FLiT could also discover and root-cause a known reproducibility bug in the Laghos proxy application. The benefit of FLiT is the automated re-discovery of this critical bug (first located through a two week *ad hoc* manual search). This automated re-discovery took only 14 application runs under Bisect, taking only 40 minutes.

To quantify the efficacy of our Bisect algorithm even more sharply, we implemented a custom LLVM pass to inject floating-point perturbations in the LULESH proxy application. We achieved precision and recall of 100% at identifying the source of variability, or reporting that the injection was benign and caused no variability. Each injection took only 15 application executions on average during the Bisect search to find the function exhibiting variability.

2 WORKFLOW FOR MULTI-LEVEL ANALYSIS

Key to the design of FLiT is a choice of approaches and algorithms that are essential to making an impact in today's HPC contexts. We now present some of these choices and describe the workflow in Figure 1.

We define a **compilation** as a triple (Compiler, Optimization Level, Switches) applied to a subset of source files in an application. This triple contains the full configuration of how to compile a source file – as far as optimizations and compiler options are concerned. Our work helps hunt down compilations that cause result-variability.

Handling vendor-specific and general-purpose compilers. Vendor-provided compilers are vital to achieving high performance, especially within newly delivered HPC machines. Given this, FLiT cannot rely on technologies that do not generalize to many compilers and architectures. Some such technologies are binary instrumentation tools such as PIN (for Intel architectures) and instrumentation passes based on LLVM (for LLVM-based compilers only).

Applicability in HPC build systems. Productivity-oriented approaches in HPC critically depend on infrastructures such as Kokkos [15] and RAJA [17] that synthesize efficient code, naturally affect loop optimizations, and smoothly incorporate parallelism. Framework-specific annotations burden static analysis based approaches because each framework requires separate support and implementation. FLiT avoids this by dealing with compiled object files directly.

Use designer-provided tests and acceptance criteria. A generic tool such as FLiT cannot have pre-built notions of which results are acceptable. Therefore FLiT engineers its solutions around C++ features to require a minimal amount of customization. For each test, the user creates a class and defines four methods:

- **`getInputsPerRun`:** Simply returns an integer – The number of floating-point values taken by the test as input (between 0 and the maximum value of `size_t`)
- **`getDefaultInput`:** Returns a vector of test input values. If there are more values here than specified in `getInputsPerRun`, then the input is split up, and the test is executed multiple times, thus allowing data-driven testing [6].
- **`run_impl`:** The actual test that takes a vector of floating-point values as input and returns a test result. The test result can either be a single floating-point value, or a `std::string`. FLiT provides the return type of `std::string` so that the user can use more complex structures returned, such as arbitrary meshes.
- **`compare`:** Takes in the test values from the baseline and testing compilations, and returns a single floating-point value. If the two values are considered equal, then this function should return 0. Otherwise, this function should return a positive value. This function behaves as a metric between the two values and is how FLiT determines if there is variability in a compilation compared to the baseline.
  There are two variants of this compare function, one for `long double` values and another for `std::string` values. The user is only required to implement the associated variant for the return type of their test.

FLiT requires deterministic executions, as shown in Figure 1. On a given platform and input, we must be able to rerun an application and obtain the same results as measured by the user-provided `compare` function. There are many deterministic HPC applications, even many MPI and OpenMP applications that provide run-to-run reproducibility. Therefore, FLiT supports the use of deterministic MPI and OpenMP. As depicted in Figure 1, if an application is not deterministic, then external methods can be used to make it deterministic. For example, one can identify and fix races with a race detector such as Archer [4], or directly determinize an execution using a capture-playback framework such as ReMPI [33].

Currently, support for GPUs does not exist in FLiT. With GPUs, the scheduling of warps can cause floating-point reassociations, thus changing execution results. [2]. Given the rapid evolutions in the GPU-space, this is future work

2.1 Bisect Problem

The Bisect problem handled by FLiT is multifaceted: it must help locate variability-inducing compilations while also checking for acceptable execution results. Unfortunately, modern compilers are quite sophisticated, and their internal operation involves many decisions such as link-time library substitutions, the ability (or lack of) to leverage new hardware resources, and many more such options that affect either performance or the execution results. This richness forces us to adopt an approach that is as generic as possible and consists of compiling different files at different optimizations and drawing a final linked image from this mixture. The granularity of mixing versions in our case is either at a file level, or (by using weak symbols and overriding) at a function level[3]. When we encounter a numerical result difference during

[2] There is little external control one can exert on GPU warp schedulers.

[3] The approach of searching by overriding symbols is one that potentially creates "Frankenbinaries." For example, we may link together an Intel-compiled function with a GCC-compiled function at differing optimization levels. Our symbol-based search consists of first creating various binaries (a one-time cost) and merely going through different linkage combinations - which typically takes far less time than a compilation.

Algorithm 1 Bisect Algorithm

```
1:  procedure BisectAll(Test, items)
2:      found ← { }
3:      T ← Copy(items)
4:      while Test(T) > 0 do
5:          G, next ← BisectOne(Test, T)
6:          found ← found ∪ next
7:          T ← T \ G
8:      assert Test(items) = Test(found)
9:      return found
1:  procedure BisectOne(Test, items)
2:      if Size(items) = 1 then                    ▷ base case
3:          assert Test(items) > 0
4:          return items, items
5:      Δ1, Δ2 ← SplitInHalf(items)
6:      if Test(Δ1) > 0 then
7:          return BisectOne(Test, Δ1)
8:      else
9:          G, next ← BisectOne(Test, Δ2)
10:         return G ∪ Δ1, next
```

our bisection search, we allow existing tools to help with root-causing. Thus FLiT's task is to isolate the problem down to a file or a function.

An essential practical reality is that hundreds of functions comprise a large application spread over multiple files. It is possible that the compiler optimization may have affected any subset of these functions to cause the observed variability. The objective of FLiT's Bisect algorithm is to identify and isolate all functions that have contributed to result-variability.

In a general sense, one faces the daunting prospect of identifying those functions that are "coupled," meaning they must be optimized together in a certain way to cause result-variability. The need to identify "coupled" functions would lead to a search algorithm that considers all possible subsets of files or functions — an exponential problem that, if implemented as such, would result in a very slow tool. The singleton blame site assumption alluded to earlier reduces the search space considerably, as discussed in more depth in Section 2.4.

2.2 Bisect Algorithm

The Bisect algorithm (Algorithm 1) follows a simple divide and conquer approach. It takes two inputs: (1) *items*, which is a **set** of files/functions in the compilations to be searched over; and (2) A test function Test that maps *items* to a real value that is greater than or equal to 0. A non-zero output indicates the existence of result variability and also helps us sort the problematic items (files and functions) in order of the *degree of variability* they induce by themselves. It also allows us to formulate the BisectBiggest algorithm (discussed in Section 2.5). A zero output indicates that there is no result-variability.

Notice that procedure BisectOne (helper to procedure BisectAll) does not merely return the next found element. It instead returns a pair of two sets. The first set contains elements that can safely be removed from future search steps. The second is a

Step	*items* fed to Test in Algorithm 1										result
1	1	2	3	4	5	6	7	8	9	10	✘
2	1	2	3	4	5	·	·	·	·	·	✘
3	1	2	·	·	·	·	·	·	·	·	✘
4	1	·	·	·	·	·	·	·	·	·	✔
5	·	②	·	·	·	·	·	·	·	·	✘
6	x	x	3	4	5	6	7	8	9	10	✘
7	x	x	3	4	5	6	·	·	·	·	✔
8	x	x	·	·	·	·	7	8	·	·	✘
9	x	x	·	·	·	·	7	·	·	·	✔
10	x	x	·	·	·	·	·	⑧	·	·	✘
11	x	x	x	x	x	x	x	x	9	10	✘
12	x	x	x	x	x	x	x	x	⑨	·	✘
13	x	x	x	x	x	x	x	x	x	10	✔
Result		②						⑧	⑨		

Figure 2: Illustrative example of BisectAll (Algorithm 1). The numbers represent tested elements. The dots represent elements within the current search space, but not being tested. The small x's represent elements that have been removed from the search space because of previous iterations of Bisect. The ✘ means Test(*items*) > 0 and ✔ means Test(*items*) = 0. The found variability-inducing items are {2, 8, 9}. Each row represents a separate executable by linking together the items under test from the variable compilation and all others from the baseline compilation.

singleton set — the "found element" in essence. As line 2 of BisectOne indicates, this means that Test (*items*) is greater than 0, *i.e.*, the presence of this singleton set, namely *items*, in a compilation causes result-variability. That means we have successfully located one variability-inducing file/function. We now return the pair *items*, *items* indicating: (1) that we found *items*, and (2) we can exclude *items* in future searches (line 7 of BisectAll). These elements are then removed from the search space in future Bisect searches (as seen on line 7 of procedure BisectAll in Algorithm 1). This removal is not necessary for the algorithm to work correctly, or even for the complexity, but it is merely an optimization that allows us to prune the search space if we happen to find elements which cause the given test to pass. This optimization is one significant deviation from Delta debugging [41] — a point discussed under the heading **Assumption 2** of Section 2.4.

As a specific example of this strategy, notice what we do on line 9 of BisectOne which is when Test(Δ_1) = 0. Then we suppress future testing on $G \cup \Delta_1$.

The Test function that is passed to the Bisect algorithms is a user-defined metric that has the following attributes:

- Maps a set of items to a non-negative value, $[0, \infty)$.
- Test(*items*) = 0 $\Rightarrow$ there are no variability causing items
- Test(*items*) > 0 $\Rightarrow$ there is at least one variability causing item

In Figure 2, we can see an example of running Algorithm 1. The ✔ symbol indicates an instance when Test(*items*) = 0 and the ✘ symbol indicates Test(*items*) > 0. Horizontal lines separate individual invocations of BisectOne. The small X's in Figure 2

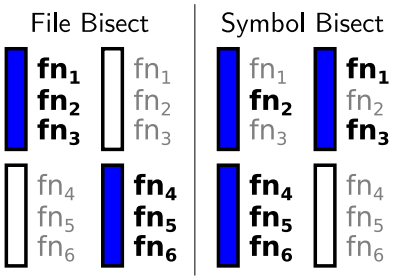


Figure 3: Highlights the difference between File Bisect and Symbol Bisect. File Bisect mixes compiled object files. Symbol Bisect marks some symbols as weak within object files and links in both copies of the object file. The functions in bold are strong symbols that are available in the final executable. Only Symbol Bisect requires the `-fPIC` flag so that we can match up functions arbitrarily.

refer to the extra set of elements returned by procedure BisectOne indicating a set of elements to discard for future search.

Although it is true that for this example, it would be cheaper to do a linear search over the elements, a linear search would always be $O(n)$, where n is the total number of elements. This Bisect algorithm has worst-case complexity $O(k \log n)$ and best-case complexity $O(k \log k)$ where k is the number of variability-causing elements to find. Section 2.4 discusses these bounds in more detail.

2.3 Implementation of Bisect

The Bisect search algorithm utilizes a well-known divide and conquer technique but applying it to find the functions causing variability is nontrivial. Note, the terms "function" and "symbol" are used interchangeably, although symbol usually refers to a compiled version of the function. Since the problem is to find all functions causing variability, we could group all functions of the application and apply the Bisect algorithm. But, for anything more substantial than small applications, the search space becomes too large to search effectively. Instead, akin to how Delta Debugging [41] was extended to work on hierarchical structures [27], we perform this Bisect algorithm on a dual-level hierarchy, first by searching for the files where the compiler caused variability, and then searching the functions within each found file. This hierarchical approach allows us to reduce the search space considerably, by splitting up the full Bisect search into much smaller separate searches.

The Test function used for File Bisect links together the object files generated from the two different compilations, some from the variability-inducing compilation, and the rest from the baseline compilation. The Test function passed into the Bisect algorithm is generated from the baseline compilation, the variable compilation, and the full list of source files. When a set of source files are passed into the Test function, those files are compiled with the variable compilation with all others compiled with the baseline compilation, and then the two sets of object files are linked together. We provide a visualization of File Bisect in the left half of Figure 3.

It is possible that the baseline and variable compilations use different compilers, in which case this approach depends heavily on binary compatibility between the two compilers [2, 18]. Since many compilers implement their own C++ standard library (since C++ 11), one achieves binary compatibility only by forcing all compilers to use a common implementation. In our experiments, we chose to have all compilers use the GCC implementation of the C++ standard library.

In the File Bisect phase, the Bisect algorithm finds all variability-contributing object files when compiled with the variable compilation. Each compiled object file comes from a single source file, and therefore can indicate each responsible source file.

Having finished finding all variability-contributing object files, we move on to finding the variability-inducing symbols within the found object files (*i.e.*, methods and functions). This second pass over symbols, called Symbol Bisect, is performed individually on all symbols within each found variability-producing object file.

Exploiting Linker Behavior and Objcopy: The method for selecting functions from two different versions of the same object file is done by making use of strong and weak symbols and is shown in the right half of Figure 3. At link time, if there is more than one strong symbol, the linker reports a duplicate symbol error. If there is more than one weak symbol, then the linker is allowed to choose which one to keep and discards the rest. In the case there is one strong symbol and one or more weak symbols, the linker keeps the strong symbol and discards all weak symbols. It is the last case we utilize to select functions. Using `objcopy`, we can duplicate an object file, and change a subset of the strong symbols into weak symbols. The other object file is then treated similarly, but marking the complement set of symbols as weak. At this point, both object files can be successfully linked together into the executable.

However, when a compiler generates an object file, it works under the assumption that the object file, also known as a single translation unit, is indivisible [20], and therefore perform many optimizations based on that assumption. This problem of switching the implementation of a function has been solved in the domain of shared libraries, with the use of `LD_PRELOAD` and is called interposition. To successfully replace all instances of one function, one must use the `-fPIC` flag, thus disabling inlining of functions that are callable from other translation units (*i.e.*, the globally exported symbols). When the search reaches the Symbol Bisect phase, the target file is recompiled with this flag, and the result is checked. If variability is removed by using `-fPIC`, then the search cannot go deeper; we must be content with reporting the file containing the variability. We are limited, therefore, to search within the space of globally exported symbols, since those are the only ones we can guarantee can be replaced entirely with the desired version.

Our File Bisect and Symbol Bisect approaches are not the only ways to combine functions from two different compilations. For example, some compilers allow turning on and off compiler optimizations using `#pragma` statements. This approach would work only for compilers with such a capability, and would not be able to handle the situation of mixing compilations that have two different compilers, such as GCC and the Intel compiler, or even two different compiler versions. Another strategy is to split the functions into separate source files. However, this approach is non-trivial to implement and has the potential to disable many of the optimizations that cause variability. The final approach we considered was compiler intermediate representation, such as LLVM IR. This approach will work only with the compilers with which we can perform such a

pass, at the very least excluding the use of closed source compilers such as the Intel compiler, the IBM compiler, and the PGI compiler. For these reasons, we chose to work on combining object files after compilation to conduct our search in File Bisect and Symbol Bisect.

We autogenerate the Test function for Symbol Bisect using the full set of source files, the one source file to search, and the full list of globally-exported symbol names from that source file. It then marks certain symbols as weak from the two versions of the variability-inducing object file (compiled by FLiT with `-fPIC`) and links together these two object files with the rest of the object files compiled with the baseline compilation.

2.4 Bisect Analysis

Stated in a general manner, our objective is to find all functions that contribute to the observed variability. The Bisect algorithm is used for both symbols and files, so here we use a set of elements for which a Test function can quantify the observable variability.

Bisect is based on Delta Debugging, whose explicit goal is to find a single **minimal set** that causes Test to fail [41].

Definition 1. *Y is a **minimal set** of X, denoted by the boolean relation $MS(Y, X)$, if* $\forall Z, [Y \subseteq X \wedge \textsc{Test}(Y) > 0 \wedge Z \subsetneq Y \implies \textsc{Test}(Z) = 0]$.

Such a minimal set is not guaranteed to be unique. Furthermore, Delta Debugging only approximates minimal sets. Instead of finding an arbitrary minimal set, we seek to find all elements that contribute to the variability observed when we test all elements. We start out by defining the elements we do not care about.

Definition 2. *x is a **benign element** of X, denoted by the boolean relation $B(x, X)$, if* $\forall Y \subseteq X, [\textsc{Test}(Y) = \textsc{Test}(Y \cup \{x\})]$.

In other words, a benign element has no effect on the outcome of Test within the set X. Using this definition of a benign element, we define a variable elements as not benign.

Definition 3. *The set of **all variable elements** of X, denoted $AV(X)$, is* $AV(X) \triangleq X \setminus \{x : B(x, X)\}$

This set $AV(X)$ represents the smallest set that fully explains $\textsc{Test}(X)$. Specifically, by the definition of benign elements, we see $\textsc{Test}(AV(X)) = \textsc{Test}(X)$. Finding this set $AV(X)$ is the goal of this paper and of the Bisect algorithm. Without any assumptions or restrictions on the search space, just identifying a single benign element x requires testing against every subset of X to certify that x is truly benign. The complexity to evaluate $B(x, X)$ is $O(2^N)$ for just one element, where $N = |X|$.

Assumption 1. *Errors from different sets of variable elements are distinct in magnitude. That is,* $\textsc{Test}(X) = \textsc{Test}(Y)$ *if and only if* $AV(X) = AV(Y)$.

This assumption states that the only way for Test values to match is if the same underlying variable elements are present. Given the nature of floating-point arithmetic, it is very unlikely for compiler-induced variability to have the exact same magnitude. Without this assumption, we could not do any better than brute-force search or some approximation technique.

It is noteworthy to mention that given this assumption, we can formulate this problem to be solved by Delta Debugging, as follows. Let U be the universal set of all elements. Define a new Boolean function $\textsc{Test}'(Y) \triangleq [\textsc{Test}(Y) = \textsc{Test}(U)]$.

Theorem 1. *Let* $MS'(Y, X) \triangleq \forall Z, [Y \subseteq X \wedge \textsc{Test}'(Y) \wedge Z \subsetneq Y \implies \neg\textsc{Test}'(Z)]$. *If Assumption 1 holds, then* $MS'(AV(U), U)$ *and* $\forall X, [X \neq AV(U) \implies \neg MS'(X, U)]$. That is, $AV(U)$ is the unique minimal set of U.

Proof. By the definition of AV, we have $\textsc{Test}'(AV(U))$ is true because $AV(AV(U)) = AV(U)$. From Assumption 1, if $Z \subsetneq AV(U)$, then $\neg\textsc{Test}'(Z)$, since $AV(Z) \neq AV(U)$. Therefore $MS'(AV(U), U)$ is true. Now, assume $AV(U)$ is non-unique. Then there exists an $X \subseteq U$ such that $X \neq AV(U)$ and $MS'(X, U)$ is true. This leads to a contradiction:
Case 1: $AV(X) = AV(U) \subsetneq X$.
But $AV(X) \subsetneq X$ and $\textsc{Test}'(AV(X))$, therefore $\neg MS'(X, U)$. ↯
Case 2: $AV(X) \subsetneq AV(U)$.
But $\textsc{Test}(X) \neq \textsc{Test}(U)$ because $AV(X) \neq AV(U)$ by Assumption 1. Therefore $\neg\textsc{Test}'(X)$ and subsequently, $\neg MS'(X, U)$. ↯
□

Since Delta Debugging finds minimal sets and this minimal set is unique, we could use Delta Debugging at this point to solve for $AV(U)$. The complexity of the Delta Debugging algorithm is $O(k^2 \log N)$, where $k = |AV(U)|$ and $N = |U|$. We can do better.

Assumption 2. ***Singleton Blame Site Assumption**. Each variability element contributes individually.*

$$\forall x \in AV(X), \textsc{Test}(\{x\}) > 0$$

This assumption claims there is no situation where two or more elements need to be tested together in order to generate a measurable variability. In general, this is not always true. However, we found in the domain of compiler-induced variability, it is true in practice – as demonstrated by the experimental use cases in this paper. With Assumption 2, we can now do Bisect search to find each element of $AV(U)$ individually. Each call to BisectOne is a logarithmic search with complexity $O(\log N)$. This function is called once for each element to find from $AV(U)$. Therefore, the complexity of the Bisect algorithm is $O(k \log N)$, again with $k = |AV(U)|$. If k is proportional to N (which for this problem we have not seen to be the case), then a linear search may outperform both Bisect search and Delta Debugging.

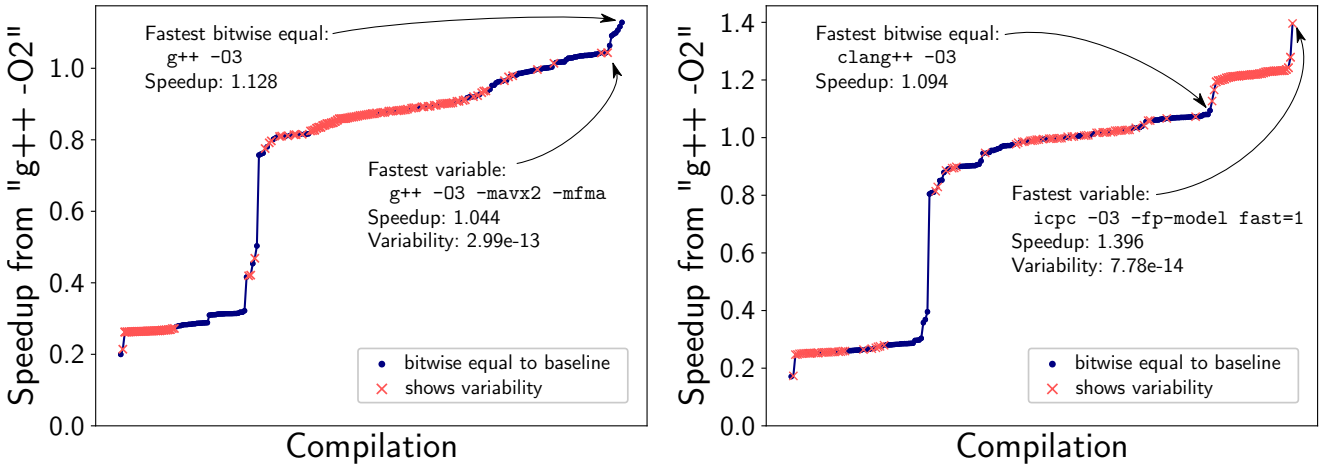
What if Assumption 2 is not true? We would generate false negatives. Except, false negatives are formally checked using the assertions found in the Bisect algorithm. The assertion on line 3 of BisectOne verifies against the case when more than one element is required to cause Test to be positive. It ensures that the list of found elements are each individual contributors to variability. The assertion on line 8 of BisectAll guarantees that $found = AV(items)$.

Proof. By Assumption 1, since $\textsc{Test}(found) = \textsc{Test}(items)$, we have $AV(found) = AV(items)$. Furthermore, because of the assertion on line 3 of BisectOne, we know that each element of *found* is a variable element. Therefore, $found = AV(items)$. □

Despite this simple proof, the result is profound. If Assumption 1 holds, and the assertions in the Bisect algorithm pass, then there are no false negatives, meaning we have found all variability elements.

Table 1: Compilers used in the MFEM study with summary statistics. The best flags are chosen by the best average speedup across all MFEM examples. The average speedup over all 19 MFEM examples is reported and is calculated relative to the speed of `g++ -O2`.

Compiler	Released	# Variable Runs	Best Flags	Speedup
gcc-8.2.0	26 July 2018	78 of 1,288 (6.0%)	`-O2 -funsafe-math-optimizations`	1.097
clang-6.0.1	05 July 2018	24 of 1,368 (1.8%)	`-O3 -funsafe-math-optimizations`	1.042
icpc-18.0.3	16 May 2018	984 of 1,976 (49.8%)	`-O2 -fp-model fast=2`	1.056



(a) Example 5 **(b) Example 9**

Figure 4: MFEM examples, speedup vs. compilation with compilations sorted by speedup. Both bitwise equal and variable compilations are shown. In (a), the fastest bitwise equal compilation was the fastest overall. In (b), the opposite is true.

And this dynamic verification requires $2 + k$ extra calls to Test (though really $1 + k$ calls because Test(*items*) can be memoized). However, if the assertion fails, then either Assumption 1 or Assumption 2 are false, in which case the user is notified that there may be false negative results. Also worth noting is that because of the assertion on line 3, we guarantee that *found* are all variable elements, meaning it is impossible to get false positive results.

2.5 The Bisect Biggest Algorithm

Along with the Bisect algorithm that finds all variability-inducing files and functions, we developed an algorithm that can search for the biggest k contributors where the user can choose the value for k. This variant is based on Uniform Cost Search and can exit early. Upon finding the largest contributing file, it immediately recurses to find the k largest contributing symbols. When a file or symbol is found to have a smaller Test value than the k^{th} found symbol's Test value, it exits early. It is not able to dynamically verify assumptions, but can significantly improve performance if only the top few most contributing functions are desired, and there happen to be many more than that to find.

3 EXPERIMENTAL RESULTS

We performed three evaluations of FLiT: MFEM, Laghos, and LULESH. We applied FLiT to MFEM to view the speed and variability space; then we applied FLiT Bisect on all found variant compilations. The second evaluation is a real-world case study running FLiT Bisect on the Laghos codebase with an unknown issue with variability. Finally, we used an LLVM pass to modify floating-point operations in the compilation of the LULESH miniapp to evaluate precision and recall of the Bisect algorithm.

3.1 Performance vs. Reproducibility Case Study

MFEM is a finite element library poised for use in high-performance applications. FLiT was used with three mainstream compilers to view the tradeoff between reproducibility and speed, as seen in Figure 4. In Figure 5 we examine the fastest non-variant compilations given by each compiler with the fastest variant overall.

The MFEM library comes with 19 end to end examples of how to use the framework, which is what we used as test cases in FLiT. These examples include the use of MPI, which FLiT now supports. Each example produces calculated values over a full mesh or volume. The comparison function used the ℓ_2 norm of the mesh difference, $||baseline - actual||_2$.

Using FLiT, we compiled MFEM using the `g++`, `clang++`, and `icpc` compilers as listed in Table 1. For these compilers, we paired a base optimization level, `-O0` through `-O3`, with a single flag combination, taken from the list used in [34]. This cartesian product leads to 244 compilations, and with 19 test cases results in a total of 4,636

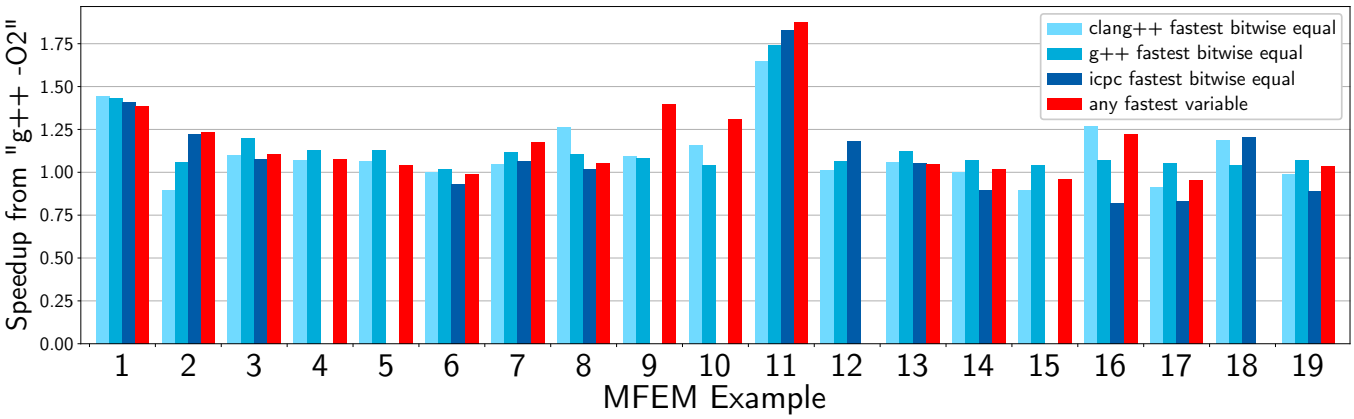


Figure 5: Performance histogram of the fastest compiled executable from each category for each MFEM test. The left three blue bars for each example represent the most performant bitwise equal execution, with the right red bar being the most performant execution exhibiting variability (combined from the three compilers). Missing bars mean there were no results in that category. Examples 12 and 18 had no compilations that produced variability. Examples 4, 5, 9, 10, and 15 are missing the Intel compiler bar, because variability was introduced by the Intel link step, regardless of optimization level or switches.

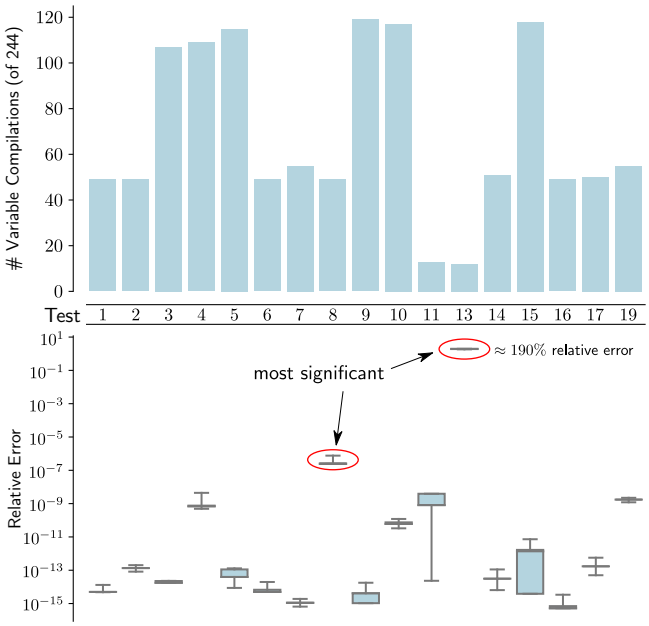


Figure 6: MFEM found variability. For each test, the top bar chart shows the number of variability-inducing compilations out of 244 found by FLiT. The bottom boxplot has a vertical logscale and shows the range of relative ℓ_2 errors induced by the different compilations. Tests 12 and 18 are omitted because they had no found variabilities.

experimental results. Looking at a single MFEM example and ordering the compilations from slowest to fastest, we get graphs similar to those found in Figure 4. The points marked with a blue circle compare equal to the baseline results from `g++ -O0`, and those with a red X exhibit variability. For MFEM example 5 (Figure 4a), the fastest compilation with bitwise equal results was 12.8% faster than `g++ -O2`. This example was not an outlier; we find similar results in 14 of the 19 examples (Figure 5). This finding contrasts with Figure 4b, which has the variant compilations grouped near the top and showing a significant speedup over the fastest functionally equivalent compilation.

While these plots give detail to individual experiments, Figure 5 shows a bigger picture. Each grouping shows the fastest non-variant compilation and the fastest variant compilation in regards to a single experiment. Once again, 14 out of 19 experiments show non-variant compilations to be also the fastest. Variant compilations are noticeably faster than non-variants in only 2 of the groupings.

The magnitude of the observed result-variability is also important to consider. In Figure 6, we see the min, median, and max of the relativized errors observed by the different compilations of each MFEM example. The errors were normalized by dividing by the ℓ_2 norm of the baseline mesh values. Examples 8 and 13 showed significant variability, and are examined further using Bisect.

Table 2: Compiler characterization of Bisect with MFEM. Only those runs that succeeded with File Bisect went on to perform Symbol Bisect. A failure here means the resulting mixed executable crashed.

	g++	clang++	icpc	total
average test executions	64	29	27	30
File Bisect successes	78/78	24/24	778/984	880/1,086
Symbol Bisect successes	51/78	24/24	585/778	660/880

3.2 Bisect

FLiT found 1,086 compilations which lead to variant results, each of which were explored by FLiT Bisect. These searches were over a non-trivial codebase. An overview of the success rate of Bisect is available in Table 2.

The MFEM library contains almost 3,000 functions which are exported symbols, as seen in Table 3. The FLiT Bisect approach depends only on the number of source files and functions, as opposed to static and dynamic analysis approaches that rely on the depth and breadth of the call tree. While this size of 3,000 functions is daunting for a linear search, the Bisect approach used an average of 30 executions including the verification assertion. FLiT was able to isolate the variability to the file level 80% of the time, and of those was able to isolate the variability to the symbol level 75% of the time.

Two findings were significant enough to be reported back to the MFEM team and are currently under further investigation.

Finding 1: MFEM example 8 is an iterative algorithm with a stopping criteria of 10^{-12}, yet converges to a value that has an absolute error of 10^{-6}, meaning it converged differently because of compiler optimizations. FLiT Bisect found all nine functions causing the variability for example 8, each performing matrix and vector operations. The compilations were `icpc -O2`, `icpc -O3`, `g++ -O2 -mavx2 -mfma`, `g++ -O3 -mavx2 -mfma`, and `g++ -O3 -funsafe-math-optimizations`. FMA is a likely culprit as well as vectorization.

Finding 2: Example 13 had the most substantial variability by far, having between 183% to 197% relative error. FLiT Bisect found only one function to contribute to variability, a function that calculates $M = M + aAA^{\top}$ with a being a scalar, and M and A being dense square matrices. This function is implemented in a straightforward manner using nested for loops. The compilations responsible enable AVX2, FMA, and higher precision intermediate floating-point values. Therefore, we suspect FMA, vectorization, and higher precision intermediates to be the reasons for the variability.

3.3 Characterization of Compilers

From this two-part experiment, we can assess the compilers predilection for speed, variability, and compatibility.

The maximum available speedup for a single example ranges from a factor of 1.02 to 1.87 relative to the `g++ -O2` compilation. But each example has its own best compilation. Since MFEM is a library, it is better to see which compilation lead to the best average speedup across all examples to cover all use cases. The best average compilations, separated by compiler, can be seen in Table 1, in which `g++` comes in first with a speedup factor of 1.097. Note, all three of these fastest average compilations have variability induced on at least one example.

In that same Table is the percentage of compilations which caused variability. The most invariant compiler is `clang++` with only 1.8% of compilations deviating from the baseline. The most variant compiler, producing almost half variable compilations (at 49.8%), is the Intel compiler, `icpc`. Intel's compiler went from a distant second in speed to last in variability.

By examining the Bisect results more closely, we discovered some issues that drove the 20% failure rate of File Bisect. When `icpc` and `g++` object files were linked together, the resulting executable would sometimes fail with a segmentation fault. While Intel claims compatibility with the GNU compiler [18], this does not seem to always hold.

Table 3: General statistics of code used by the MFEM examples.

source files	97
average functions per file	31
total functions	2,998
source lines of code	103,205

3.4 Penetration into Laghos

The issue found by the developers of Laghos manifested when they compiled with IBM's `xlc++` compiler at `-O3`. Given the code, Bisect was able to find an issue not related to floating-point that was already fixed in another branch. After fixing that problem, we were able to isolate the problem down to the function level.

The tool developers trusted the results from both `g++ -O2` and `xlc++ -O2` when using their branch of the code. We used a public branch of the code in an attempt to reproduce the results they had. In our runs, all results were the special floating point value NaN. Using Bisect, we narrowed this down to the two visible symbols closest to the issue. The source code in question was `#define xsw(a,b) a^=b^=a^=b`, which evokes undefined behavior in C++. Bisect identified these two function in 45 program executions. The developers confirmed the bug, which they had fixed in their version. While this may appear to be a case of finding a bug yet again, the fact that our automated Bisection-based search found this issue must be viewed as a step forward, considering that the manual process by which the developers initially found this issue is "hit or miss" and requires expert's time to be spent.

After fixing this issue, we achieved results agreeing with the developer-stated results for both the trusted compilation and the variant `xlc++ -O3` compilation. We ran many variants of Bisect to evaluate the speed and effectiveness of BisectAll and BisectBiggest, as can be seen in Table 4. By limiting either the digit sensitivity of our `compare` function or the k value of BisectBiggest (k = all refers to using the traditional Bisect algorithm), the number of runs varied from 69 to 14, all of which were able to identify the most significant variability-inducing function. In the function pointed to was an exact comparison to 0.0 in an if statement. The value compared against 0.0 had small variability, but the difference in branching resulted in significant application variability. Changing this to an epsilon based comparison gave results close to the trusted results, even under `xlc++ -O3`.

3.5 Injection Study

We performed controlled injections of floating-point variability at all floating-point code locations to quantify the accuracy of our tool.

Our injection framework is based on the LLVM compiler [23] and introduces an additional floating-point operation in a given floating-point instruction of the LLVM intermediate representation (IR). More formally, given a target floating-point instruction of the form $x\ OP\ y$, where x and y are floating-point operands, and OP is a basic floating-point operation (+,-,*,/), we introduce an additional operation $x\ OP'\ \epsilon$, where OP' is also a basic floating-point operation and ϵ is chosen from a uniform distribution between 0 and 1. For

Table 4: Bisect statistics of the Laghos experiment. The baseline compilation is provided, with the compilation under test being `xlc++ -O3` versus the result of FLiT Bisect. The *strict* qualifier refers to the additional flag `-qstrict=vectorprecision`. We restrict the comparison to compare only the number of digits in the digits column. The k value is how many of the most contributing functions Bisect is asked to find.

baseline	digits	# files			# funcs			# runs		
	k:	1	2	all	1	2	all	1	2	all
`g++ -O2`	2	1	1	1	1	1	1	18	18	14
	3	1	1	1	1	1	1	18	18	14
	5	1	1	1	1	1	1	18	18	14
	all	2	3	5	1	2	7	28	37	57
`xlc++ -O2`	2	1	1	1	1	1	1	18	18	14
	3	1	1	1	1	1	1	18	18	14
	5	1	1	1	1	1	1	18	18	14
	all	2	3	6	1	3	7	28	37	69
`xlc++ -O3` strict	2	1	1	1	1	1	1	18	18	14
	3	1	1	1	1	1	1	18	18	14
	5	1	1	1	1	1	1	18	18	14
	all	2	3	5	1	2	5	28	39	60

example, assuming that the target instruction is

$$z = x * y,$$

after the injection, the resulting operation is:

$$z = (x + \text{1e-100}) * y.$$

In this example, *OP'* is the addition operation and ϵ is 1e-100.

Our variability injection framework requires two passes. The first pass identifies potential *valid injection locations*; an injection location is defined by a file, function and floating-point instruction tuple in the program. The second pass injects in a user-specified location, using a specific ϵ and operation *OP'*. We perform the injections at an early stage during the LLVM optimization step. Our goal is to introduce variability before optimizations take place.

For our evaluation, we used the benchmark called Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). This LULESH benchmark contains 5,459 source lines of code, in which there are 1,094 floating point operations. For each of these operations, we did four injection runs, one for each possible *OP'*.

Under our evaluation criteria as seen in Table 5, we deem a symbol reported by FLiT Bisect to be exact of the source function where the injection occurred; this occurred 2,690 times. We also count indirect finds, which is when the source function is not a visible symbol but Bisect was able to find the visible symbol which used the injected function, what happened 984 times. This indirection can occur for several reasons, with the majority coming from functions which were inlined or otherwise not exported as a strong symbol. We also count wrong finds and missed finds, which are false positives and false negatives. Both of these categories yielded no results in our runs. The final category is when the injection was not measurable. A non-measurable result is when the injection did not change the output of LULESH, which account for 702 of the runs. A non-measurable result can occur when the injection was in code that was not run, or if the injected variation did not affect the final result.

Table 5: Success statistics of the LULESH compiler perturbation injection experiment. Indirect finds are when the injected function is not in the search space but we successfully report the closest global function that calls it. Wrong finds are when the reported function does not induce variability. Missed finds are when variability occurs, but we do not report the functions responsible. Not measurable indicates a benign injection.

Category	Count
exact finds	2,690
indirect finds	984
wrong finds	0
missed finds	0
not measurable	702
total	4,376

3.6 MPI Support

All experiments described in this paper were run sequentially. However, in Figure 1, we specify runtime determinism as the only prerequisite, meaning that FLiT can be extended to run on a deterministic platform.

Currently, FLiT supports deterministic MPI. To test this path, we repeated a randomized sampling of the MFEM experiment with MPI running under 24 processes.

The first step was to give us high confidence that MFEM under MPI is deterministic for the 19 provided examples. This evaluation was done by performing 100 executions of each test and checking the full matrix output for bitwise equivalence. Unfortunately, only 17 of the 19 tests were able to be easily wrapped so that the FLiT framework could call `MPI_Init` and `MPI_Finalize` (tests 17 and 18 could not be accommodated). All 17 converted parallel tests passed this verification, so we have high confidence that FLiT would work well with MFEM under MPI.

Next, we wanted to determine the effects of adding parallelization to the MFEM examples. That is to say, how do the results from the parallel execution compare against the sequential run? We found that in the 17 used tests, increasing the parallelism changed the result, as measured by the ℓ_2 norm of the result. We believe this is due to increasing or decreasing the grid density when performing domain decomposition. Regardless of the reason for the difference, FLiT was able to identify this difference, and if the comparison function can handle different domain sizes, then it would be able to quantify the variability induced by changing the parallelism configuration.

Finally, we wanted to verify if the Bisect algorithm can identify the same files and functions under MPI as it did sequentially. For this step, we took a single random sample from a successful sequential Bisect run for each test (except for tests 7, 12, and 19, which had no successful sequential Bisect runs). Each random sample was able to isolate the same sets of files and functions, regardless of

the variability introduced by the parallelism. This approach may not work all the time (since the variability produced by parallelism may cause the code to branch differently). In the case of MFEM, this case did not arise; it is highly encouraging that FLiT generates identical results despite the parallelism.

4 RELATED WORK

4.1 Reproducibility

The general areas of floating-point error analysis and result reproducibility have been receiving a lot of attention [7, 11, 24, 31, 35]. There have also been some efforts in understanding performance and reproducibility in the setting of GPUs [40]. The study of deterministic cross-platform floating point arithmetics was reported a decade ago in [36] by Seiler. Our initial work on FLiT was inspired by this work.

In [5], the authors discuss the impact of nonreproducibility in climate codes. The tooling they provide (KGEN) is home-grown, not meant for external use [22]. Their work does not involve any capability similar to Bisect. Their focus is on large-scale Fortran support (and currently FLiT does not handle Fortran; it is a straightforward addition and is future work for us).

A tool called COSFID [25] was used to take climate codes and analyze them more systematically. Their work realizes file-level bisection search, albeit through a single recursive bash script. Their work does not perform symbol-level bisection to isolate problems down to individual functions, as we do.

The issue of designing bitwise reproducible applications is discussed in [3]. Their work focuses on the design of efficient reduction operators, improving on prior work on deterministic addition. It does not support capabilities such as compilations involving different optimizations, and bisection search.

A recent study has discussed the relative lack of understanding about floating-point arithmetic amongst practitioners [12]. The issues we encountered in Laghos (the swap macro that turned out to be undefined behavior according to C++, and the non-robust comparison against 0.0) are both indicative of this observation. Doug James et al. stress the need for widespread education in this area [19].

4.2 Performance Tuning

This work implements a very rudimentary performance tuning model of running all flag combinations within the search space and measure each one. The novelty here is to allow navigation between performance and reproducibility.

There is extensive work in the community with more sophisticated performance tuning techniques. For example, Profile Guided Optimization (PGO) [16], also known as Profile-Directed Feedback (PDF), is implemented in most mainstream compilers [30, 32, 37–39]. PGO uses an instrumented compilation to log the places in the code that are most used and in what order. This log is then used in a later compilation step to optimize the executable specifically for that trace. This approach is useful if you expect your application to follow almost the same path every time.

Other work has tuned the specific parameters within compilers such as the TACT tool [29]. This tool tunes the internal parameters of the GCC compiler optimizations for one particular application. One could take a similar approach with any compiler, but each would contain its own internal optimization parameters.

This work primarily focuses on reproducibility and identifying sources of variability, which at first seems orthogonal to performance. However, we recognize that one often changes architecture, compiler, or compiler optimization flags when seeking performance, and it is at these times that reproducibility can become an issue. We made an initial attempt to incorporate performance and performance tuning without detracting from the primary goal of reproducibility. Involving work from the vast performance tuning community into the FLiT work is left as future work.

5 CONCLUDING REMARKS

The case studies reported in this paper demonstrate that porting applications *even across today's machines and compilers/flags* can be quite problematic in the field in terms of result-variability. For HPC applications developed over decades, the problem worsens. This observation is especially true at "the end of Moore's law" where heterogeneity (CPUs, accelerators, and a plethora of compilers) is the rule and not the exception. Our work through FLiT has already impacted state-of-the-art projects at Lawrence Livermore labs, as we previously described. Our algorithms have yielded results concerning actual projects, as well as in the context of fault injection studies on the LULESH proxy application.

Without tools such as FLiT, a programmer may end up adopting draconian measures such as prohibiting the project-wide use of optimizations higher than, say, `-O2`—something that would be counterproductive. Tools such as FLiT will become increasingly important in supporting new proposals [8] for mixing the use of the fast-math and precise-math modes [28] in the same LLVM compilation. Such mixings can help relax numerical precision in sub-modules where speed matters (and result variability does not matter as much). With FLiT, one can identify which modules can be optimized under fast math, thereby supporting the use of these new LLVM options.

In addition to discovering variability, FLiT can help exercise compiler flag combinations and discover bugs. One such bug we discovered during the course of using FLiT involved using `-Ofast` and `-ffloat-store` has been reported and fixed in GCC 8.2.0 [9].

We have already begun applying FLiT to popular libraries such as CGAL [10] that find applications in 3D printing and other critical applications. It was encouraging for us to discover the (relative) ease of integrating FLiT into the building and testing infrastructure of CGAL. We also have identified specific instances of when it is unsafe to apply higher levels of optimization, as these can drastically change the computed results (*e.g.*, even discrete answers such as the number of points on a mesh). This study also revealed some limitations of Bisect that we plan to overcome. As one example, if an application heavily uses inlining, the granularity of file bisection search can often reduce to a single file, which is insufficient for precisely root-causing variability. Therefore, alternative methods (*e.g.*, dynamic execution based) must be developed. The community also needs to better address the issue of communicating the intended levels of optimizations between developers and users. Our experience is that without this information, we can overly optimize an

application, only to find it throwing exceptions or not converging properly.

Going forward, one significant limitation of FLiT, namely its inability to handle application-level non-determinism, must be addressed. We plan to extend FLiT to work under OpenMP, MPI, accelerator/GPU programming, and other forms of concurrency, with support for result determinization provided in an easy-to-use manner. Where determinization is infeasible, we may have to employ ensemble-based approaches such as proposed in [26]. Last but not least, we will continue to enhance the robustness of FLiT. We continue to maintain the open-source status of FLiT, and invite contributions as well as usage of FLiT in others' projects, providing us feedback.

6 ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by LLNL under contract DE-AC52-07NA27344 (LLNL-CONF-759867), and supported by NSF CCF 1817073, 1704715.

REFERENCES

[1] 2018. MFEM: Modular Finite Element Methods Library. mfem.org. https://doi.org/10.11578/dc.20171025.1248
[2] 2019. Using the GNU Compiler Collection (GCC): Compatibility. https://gcc.gnu.org/onlinedocs/gcc/Compatibility.html
[3] Andrea Arteaga, Oliver Fuhrer, and Torsten Hoefler. 2014. Designing Bit-Reproducible Portable High-Performance Applications. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*. 1235–1244. https://doi.org/10.1109/IPDPS.2014.127
[4] Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H. Ahn, Ignacio Laguna, Martin Schulz, Gregory L. Lee, Joachim Protze, and Matthias S. Müller. 2016. ARCHER: Effectively Spotting Data Races in Large OpenMP Applications. In *IPDPS 2016*. 53–62. https://doi.org/10.1109/IPDPS.2016.68
[5] A.H. Baker, D.M. Hammerling, M.N. Levy, H. Xu, J.M. Dennis, B.E. Eaton, J. Edwards, C. Hannay, S.A. Mickelson, R.B. Neale, D. Nychka, J. Shollenberger, J. Tribbia, M. Vertenstein, and D. Williamson. 2015. A new ensemble-based consistency test for the community earth system model. *Geoscientific Model Development* 8 (2015), 2829–2840. doi:10.5194/gmd-8-2829-2015.
[6] Paul Baker, Zhen Ru Dai, Jens Grabowski, Øystein Haugen, Ina Schieferdecker, and Clay Williams. 2008. Data-driven testing. In *Model-Driven Testing*. Springer, 87–95.
[7] Pavan Balaji and Dries Kimpe. 2013. On the reproducibility of MPI reduction operations. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*. IEEE, 407–414.
[8] Michael Berg and Steve Canon. 2019. LLVM Numerics Improvements. https://llvm.org/devmtg/2019-04/talks.html#Talk_22.
[9] Ian Briggs. 2019. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=90187
[10] cgal-library 2019. The Computational Geometry Algorithms Library. https://www.cgal.org/.
[11] Martyn J Corden and David Kreitzer. 2009. *Consistency of floating-point results using the intel compiler or why doesn't my application always give the same answer.* Technical Report. Technical report, Intel Corporation, Software Solutions Group. https://software.intel.com/sites/default/files/article/164389/fp-consistency-102511.pdf.
[12] Peter A. Dinda and Conor Hetland. 2018. Do Developers Understand IEEE Floating Point?. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018*. IEEE Computer Society, 589–598. https://doi.org/10.1109/IPDPS.2018.00068
[13] Veselin A Dobrev, Tzanio V Kolev, and Robert N Rieben. 2012. High-order curvilinear finite element methods for Lagrangian hydrodynamics. *SIAM Journal on Scientific Computing* 34, 5 (2012), B606–B641.
[14] Veselin A. Dobrev, Tzanio V. Kolev, and Robert N. Rieben. 2012. High-order curvilinear finite element methods for Lagrangian hydrodynamics. *SIAM Jounal on Scientific Computing* 34, 5 (2012), B606–B641.
[15] H Carter Edwards, Christian R Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing* 74, 12 (2014), 3202–3216.
[16] Rajiv Gupta, Eduard Mehofer, and Youtao Zhang. 2002. Profile guided compiler optimizations. (2002).
[17] Richard D Hornung and Jeffrey A Keasler. 2014. *The RAJA portability layer: overview and status.* Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).
[18] Intel. 2018. GCC Compatibility and Interoperability. https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-gcc-compatibility-and-interoperability
[19] Doug James. 2014. Standing Together for Reproducibility in Large-Scale Computing: Report on reproducibility@XSEDE. *CoRR* abs/1412.5557 (2014). arXiv:1412.5557 http://arxiv.org/abs/1412.5557 There are 54 additional co-authors of this article.
[20] ISO Jtc. 2011. SC22/WG14. ISO/IEC 9899: 2011. *Information technology—Programming languages—C*. (2011). http://www.iso.org
[21] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes.* Technical Report LLNL-TR-641973. 1–9 pages.
[22] Youngsung Kim, John Dennis, Christopher Kerr, Raghu Raj Prasanna Kumar, Amogh Simha, Allison Baker, and Sheri Mickelson. 2016. KGEN: A python tool for automated fortran kernel generation and verification. *Procedia Computer Science* 80 (2016), 1450–1460.
[23] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.
[24] Miriam Leeser and Michela Taufer. 2016. Panel on Reproducibility at SC'16. http://sc16.supercomputing.org/presentation/?id=pan109&sess=sess177.
[25] R Li, L Liu, G Yang, C Zhang, and B Wang. 2016. Bitwise identical compiling setup: prospective for reproducibility and reliability of Earth system modeling. *Geoscientific Model Development* 9, 2 (2016), 731–748.
[26] Daniel J. Milroy, Allison H. Baker, Dorit M. Hammerling, Youngsung Kim, Elizabeth R. Jessup, and Thomas Hauser. 2018. Making root cause analysis feasible for large code bases: a solution approach for a climate model. *CoRR* abs/1810.13432 (2018). http://arxiv.org/abs/1810.13432
[27] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*. ACM, 142–151.
[28] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. 2010. *Handbook of Floating-Point Arithmetic*. Birkhäuser. https://doi.org/10.1007/978-0-8176-4705-6
[29] Dmitry Plotnikov, Dmitry Melnik, Mamikon Vardanyan, Ruben Buchatskiy, Roman Zhuykov, and Je-Hyung Lee. 2013. Automatic tuning of compiler optimizations and analysis of their impact. *Procedia Computer Science* 18 (2013), 1312–1321.
[30] Dino Quintero, Sebastien Chabrolles, Chi Hui Chen, Murali Dhandapani, Talor Holloway, Chandrakant Jadhav, Sae Kee Kim, Sijo Kurian, Bharath Raj, Ronan Resende, et al. 2013. *IBM Power Systems Performance Guide: Implementing and Optimizing*. IBM Redbooks.
[31] Carlos R and Michael Steyer. 2018. Intel® MPI Library Conditional Reproducibility. (Jan. 2018). https://software.intel.com/en-us/articles/tuning-the-intel-mpi-library-basic-techniques.
[32] Vinodha Ramasamy, Paul Yuan, Dehao Chen, and Robert Hundt. 2008. Feedback-Directed Optimizations in GCC with Estimated Edge Profiles from Hardware Event Sampling. In *Proceedings of GCC Summit 2008*. 87–102. http://www.capsl.udel.edu/conferences/open64/2008/Papers/113.pdf
[33] Kento Sato, Dong H. Ahn, Ignacio Laguna, Gregory L. Lee, and Martin Schulz. 2015. Clock delta compression for scalable order-replay of non-deterministic parallel applications. In *Supercomputing (SC)*. 62:1–62:12. https://doi.org/10.1145/2807591.2807642
[34] Goef Sawaya, Michael Bentley, Ian Briggs, Ganesh Gopalakrishnan, and Dong H Ahn. 2017. FLiT: Cross-platform floating-point result-consistency tester and workload. In *Workload Characterization (IISWC), 2017 IEEE International Symposium on*. IEEE, 229–238.
[35] SC15-Repro-BOF 2016. SC15 BoF on Reproducibility of High Performance Codes and Simulations – Tools, Techniques, Debugging. https://gcl.cis.udel.edu/sc15bof.php Organized by Miriam Leeser, Dong H. Ahn and Michela Taufer.
[36] Christian Seiler. 2008. http://christian-seiler.de/projekte/fpmath/.
[37] Clang Developer Team. 2019. Clang Compiler User's Manual. https://clang.llvm.org/docs/UsersManual.html.
[38] Visual CPP Team. 2008. Visual C++ Team Blog: POGO. https://blogs.msdn.microsoft.com/vcblog/2008/11/12/pogo/.
[39] Xinmin Tian, Aart Bik, Milind Girkar, Paul Grey, Hideki Saito, and Ernesto Su. 2002. Intel® OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance. *Intel Technology Journal* 6, 1 (2002).
[40] Nathan Whitehead and Alex Fit-Florea. 2012. Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. Presented at GTC 2012.
[41] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.