DOI:10.1145/3382037

An approach to reproducibility problems related to porting software across machines and compilers.

BY DONG H. AHN, ALLISON H. BAKER, MICHAEL BENTLEY, IAN BRIGGS, GANESH GOPALAKRISHNAN, DORIT M. HAMMERLING, IGNACIO LAGUNA, GREGORY L. LEE, DANIEL J. MILROY, AND MARIANA VERTENSTEIN

Keeping Science on Keel When Software Moves

HIGH PERFORMANCE COMPUTING (HPC) is central to solving large problems in science and engineering through the deployment of massive amounts of computational power. The development of important pieces of HPC software spans years or even decades, involving dozens of computer and domain scientists. During this period, the core functionality of the software is made more efficient, new features are added, and the software is ported across multiple platforms. Porting of software in general involves the change of compilers, optimization levels, arithmetic libraries, and many other aspects that determine the machine instructions that actually get executed. Unfortunately, such changes do affect the computed results to a significant (and often worrisome) extent. In a majority of cases, there are not easily definable a priori answers one can check against. A programmer ends up comparing the new answer against a trusted baseline previously established or checks for indirect confirmations such as whether physical properties such as energy are conserved. However, such non-systematic efforts might miss underlying issues, and the code may keep misbehaving until these are fixed.

In this article, we present real-world evidence to show that ignoring numerical result changes can lead to misleading scientific conclusions. We present techniques and tools that can help computational scientists understand and analyze compiler effects on their scientific code. These techniques are applicable across a wide range of examples to narrow down the root-causes to single files, functions within files, and even computational expressions that affect specific variables. The developer may then rewrite the code selectively and/or suppress the application of certain optimizations to regain more predictable behavior.

Going forward, the frequency of required ports of computational software will increase, given that performance gains can no longer be obtained by merely scaling up the clock frequency, as used to be possible in prior decades. Performance gains are now hinged on the use of multicore CPUs, GPUs and other accelerators, and above all, advanced compilation methods. While reproducibility

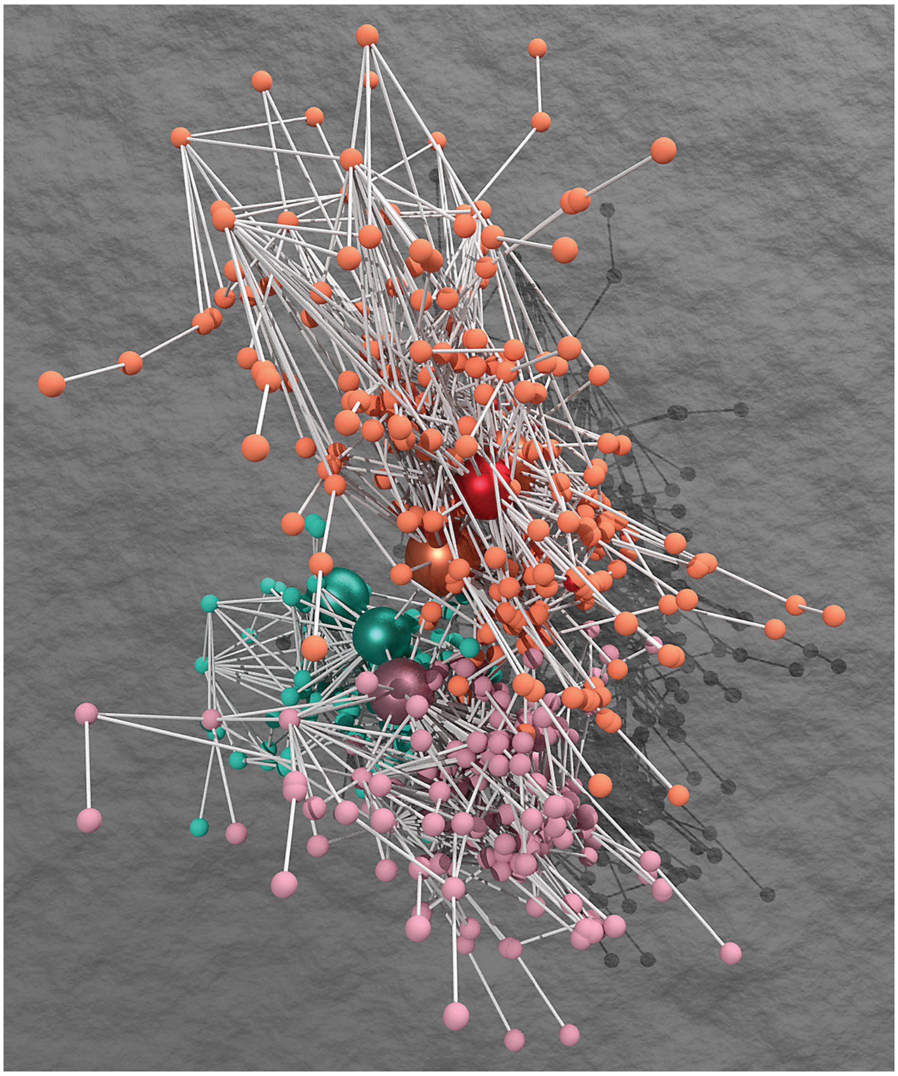
» key insights

- **Even seemingly small changes to scientific software or its build and runtime environment can create large, unexpected changes in floating-point results.**
- **Bisection search helps locate code sites that sow variability, often with firm guarantees.**
- **Combining statistical consistency testing with graph-based code analysis provides important insight into sources of floating-point variability in the Community Earth System Model (CESM™).**

across compilers and platforms in this sense is a problem that hasn't grabbed headlines in discussions centered around reproducibility, the problem is real (see sidebar "Is There a Reproducibility Problem?") and threatens to significantly affect the trustworthiness of critical pieces of software.

It may seem that all the difficulties described thus far can be solved by ensuring that compilers adhere to widely accepted rigorous standards of behavior spanning machines and optimization levels. Unfortunately, this goal is extremely difficult to realize in principle as well as in practice. Modern compilers must exploit not only advanced levels of vectorization but also the characteristics of heterogeneous computing platforms. Their optimizations in this complex space are triggered differently—even for the same compiler flags—based on the compiler's projection of the benefits of heeding the flags. This behavior is very difficult to characterize for all cases. While vendor compilers are often preferred for their superior performance—especially with respect to vectorization—they also present a challenge in terms of intervention in case issues arise.

In this article, we describe the extent of this challenge, and what is actionable in terms of equipping developers with practical tools (FLiT, CESM-ECT, and CESM-RUANDA). Some of these tools are already usable today for important codes such as hydrodynamics simulation codes and finite element libraries. We then take up the more challenging problem of climate simulation codes where much more work is needed before an adequate amount of tooling support will be developed. We describe the progress already made in this area by describing our solutions that address Earth system models (ESMs) that are central to climate simulation.


A three-dimensional, undirected representation of the example from Figure 6. Nodes are colored by community membership and sized based on a threshold centrality value. The red nodes represent model variables sensitive to specific CPU instructions. All nodes with eigenvector centrality ≤ 0.4 have a constant size, and those above the threshold are scaled and highlighted by increased reflectance. Credit: Liam Krauss of LLNL.

"Climate-changing" compiler optimizations. Earth system models (ESMs) simulate many physical, chemical, and biological processes and typically feature a complex infrastructure that couples separate modular representations of Earth system components (for example, atmosphere, ocean, land, river, ice, and land ice). ESMs are characterized by exceedingly large code bases that have resulted from decades of development, often containing a mix of both legacy code and more modern code units. Further, most ESMs are in a state of near constant development as advancing scientific discovery requires the continual addition of new features or processes, while rapidly evolving HPC technology requires new optimizations of the code base. Needless to say, software engineering for ESMs is challenging, and quality assurance is particularly critical for maintaining model credibility given that output may have policy and societal impact as future climate scenarios are considered.[7,10,23]

The popular Community Earth System Model (CESM™)[13] is a fully coupled community global climate model that enjoys widespread use across a range of computational platforms, including cutting-edge HPC architectures. With a code base of nearly two million lines across approximately 13,000 subroutines and 3,000 functions, it is

Is There a Reproducibility Problem?

- **In the Community Earth System Model (CESM™) software, the compiler introduced fused multiply add (FMA) instructions that resulted in "climate changing" differences from the baseline simulations.[3]**
- **Compiling Laghos (https://github.com/CEED/Laghos), a hydrodynamics simulation, under the IBM compiler `xlc` with optimization level `-O3`, there were negative densities created and energy was not conserved after just one iteration.[5]**
- **FLiT-based testing of the MFEM finite element library revealed that even reasonable compiler optimization levels can change the result by as much as 190%.[5]**

Statistical Ensemble Consistency Testing

When a climate simulation code is ported to a new platform, the output on the new platform will not be bit-identical to the original. This difference makes answering the question of consistency non-trivial. Instead, we ask a more tangible question: *Is the new output statistically distinguishable from the original?*

The CESM Ensemble Consistency Test (CESM-ECT) was developed to answer this new question. Ensemble methods are common in climate studies, as a collection of simulations are needed to describe the internal variability in the climate model system. (Climate models are inherently chaotic, meaning that even tiny perturbations or differences can cause large effects.) CESM-ECT generates a large "baseline" ensemble on a trusted machine and software stack and utilizes a testing framework based on the popular technique of Principal Component Analysis (PCA) to determine whether a set of new simulations (for example, from a new machine, compiler upgrade, optimization, and so on) is statistically distinguishable from the baseline ensemble. This ensemble-based approach to verification serves as a powerful classification tool when bit-identical requirements are too restrictive.

critical to ensure that changes made during the CESM development life cycle do not adversely affect the model results. A CESM simulation output is only bit-reproducible when the exact same code is run using the *same* CESM version and parameter settings, initial conditions and forcing data, machine, compiler (and flags), MPI library, and processor counts, among others. Unfortunately, control over these quantities to this degree is virtually impossible to attain in practice, and further, because the climate system is nonlinear and chaotic, even a double-precision roundoff-level change will propagate rapidly and result in output that is no longer bit-identical to the original.[21,a] As an example, a port of CESM to a new architecture is a common occurrence that perturbs the model's calculations (all of which are carried out in double-precision) and requires an evaluation for quality assurance. While the output on a new machine will not be bit-identical, one would reasonably expect there to be some degree of consistency across platforms, as the act of porting should not be "climate-changing." We would expect the same scientific conclusions to be reached when analyzing output from model runs that were identical in all but compute platform.

In the past, such CESM consistency checks were costly undertakings that required climate science expertise and multi-century simulations, as there is not a simple metric for what defines climate changing. However, statistical testing techniques have recently been developed that define consistency in terms of statistical distinguishability, leading to the creation of the CESM Ensemble Consistency Test (ECT)[1,2,21] suite of tools (see the sidebar "Statistical Ensemble Consistency Testing"). The simple and efficient CESM-ECT tools are regularly used by CESM software engineers for evaluating ports to new machines, software upgrades, and modifications that should not affect the climate. In practice, CESM-ECT has proven effective in exposing issues in the CESM hardware and software stacks, including large discrepancies caused by fused multiply-add (FMA) optimizations, an error in a compiler upgrade, a random number generator bug specific to big-endian machines, and an incorrect input parameter in a sea ice model release. In addition, by relaxing restrictive bit-identical requirements, CESM-ECT has allowed greater freedom to take advantage of optimizations that violate bit reproducibility but result in statistically indistinguishable output. Note that optimizing performance for climate models has long been of interest due to their computational expense. For example, a fully coupled "high-resolution" CESM simulation (that is, atmosphere/land at 0.25° grid spacing and ocean at 0.1°) can easily cost on the order of 250,000 core hours per simulated year.[28] While lower resolution simulations consume fewer core hours per simulated year (a 1.0° grid costs ≈ 3,500 core hours), these simulations are often run for a large number of years. For example, CESM's contribution to the current Coupled Model Comparison Project (Phase 6)[11] (used by the Intergovernmental Panel on Climate Change[15] for their assessment reports) is expected to consume nearly 125 million core hours.

Flitting Behaviors

Compiler optimizations do have the capability to change the result of floating-point computations. However, it is possible, even likely, that these optimizations can generate an answer closer to the scientist's underlying model. Unfortunately, in general, it is hard to know which of two answers is better. Therefore, the best we can do is to try to reproduce a trusted implementation on trusted hardware. Thus, we focus on reproducibility and consistency of the program's output compared to the baseline generated from the trusted configuration.

It is clear that manual testing to locate the absence of reproducibility does not scale: any subset of the software submodules could be responsible for the observed result change. Projects that maintain rigorous unit testing may

a Bitwise reproducibility is a coveted goal in general (not just for CESM), as it greatly facilitates regression testing.

already be able to utilize them to locate some problems, however many large projects have insufficient unit testing. Furthermore, floating-point rounding is non-compositional: decreased error in one component can sometimes increase the overall roundoff error.[18,29] It violates some of the basic algebraic laws such as associativity (See the sidebar "Floating-point Arithmetic and IEEE).

Sources of floating-point behavioral changes are also too numerous. Sometimes hardware implementations have fewer capabilities, such as not supporting subnormal numbers in their floating-point arithmetic.[14] Some strange behaviors can be observed when subnormal numbers are abruptly converted to zero. Other times, there are additional hardware capabilities the compiler may utilize, such as replacing a multiply and an add with a single FMA instruction. While FMA can reduce floating-point rounding error locally (because there is only one rounding step instead of two), care must still be taken. A lower local error does not necessarily equate to lower global error, particularly for a code that is sensitive to roundoff.

Under heavy optimizations, compilers can change the associativity of arithmetic operations such as reductions (especially when code is vectorized). For example, an arithmetic reduction loop whose trip-count is not an integral multiple of the vector lane width must involve an extra iteration, handling the remaining elements. The manner in which this iteration is incorporated can change overall associativity. Given the increasing use of GPUs and other accelerators, one must take into account how they deviate from IEEE floating-point standard in an increasing number of ways. The use of mixed-precision arithmetic where later iterations change precision[6,20,27] can exacerbate all these behaviors.

When a simulation code is affected by any one of these reasons and the computational results are deemed unacceptable, how does a developer proceed? The first step would typically be to find the source(s) of floating-point divergence and try to narrow down the root-causes based on one's best guess or experience. Next, it seems logical to identify the sites and involved variables that play a part in the numerical inconsistency. Once inconsistent configurations and the associated code sites are identified, there may be many approaches that can be used to mitigate the inconsistency. For example, one could employ numerical analysis techniques to improve the stability of the underlying algorithm; compile the affected units with fewer optimizations; or, rewrite the units to behave similarly under the two different configurations.

Here, we present a collection of techniques that can be used on realistic HPC codes to investigate significant differences in calculated results.

FLiT: Tool for Locating Sources of Variability

FLiT is a tool and testing framework meant to analyze the effect of compilers and optimizations on user code. It allows users to compare the results between different compilers and optimizations, and even locate the code sites to the function level where compilation differences cause results to differ.

Logarithmic search. Suppose the code is contained in a collection of N files and a new compilation produces inconsistent results. We cannot know there is only one variability site or that errors are not canceled out in strange ways. To make any progress, we make the assumption that floating-point differences are unique (for example, no two variability sites exactly cancel out each other). Without this assumption, to be sure we found all variability sites, it would require an exponential search. With this assumption, we can utilize Delta Debugging[30] with complexity $O(N \log N)$. However, in practice, we have found most variability sites to act alone, meaning they contribute variability by themselves and not in concert with other components. We then make a further assumption that each site acts alone in contributing variability (call this the singleton assumption). This assumption allows for an efficient logarithmic search as illustrated in Figure 1 with complexity $O(k \log N)$ where k is the number of variability sites. Speedometers are also displayed in Figure 1 to represent performance of our partially optimized executable, demon-

Floating-Point Arithmetic and IEEE

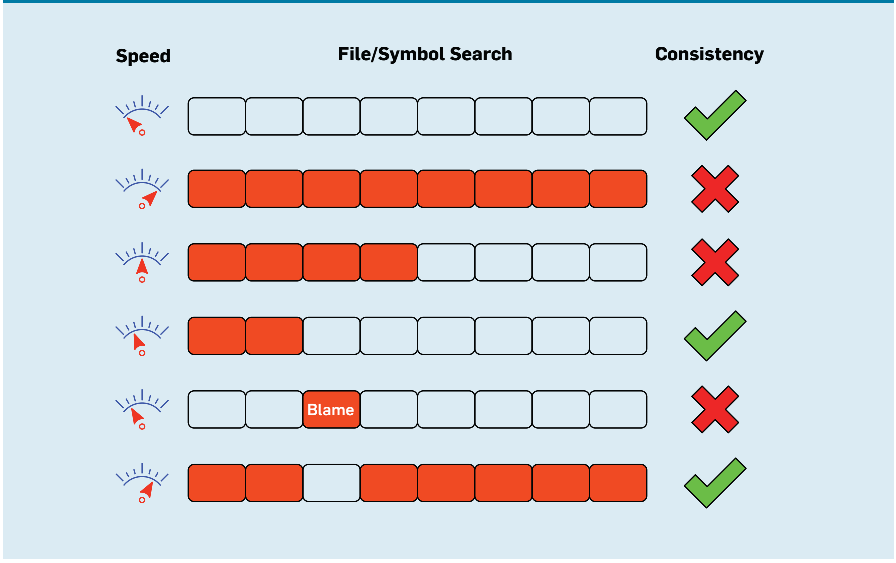
Under most circumstances FMA yields more accurate results than a multiplication and addition computed separately. However, this need not be the case. For example, given the expression $a*b + ac$, one expects the expression to evaluate to 0 when $b == -c$. However, with FMA, the calculation carried out might be $a \ast c + ac$ where ac represents the result of $a \ast c$ with rounding. Kahan[17] presents another example where the multiplication of a complex number by its complex conjugate using FMA might not produce a real number.

Climate Models Are Important

Given recent warming trends and increases in extreme events, understanding present, past, and future climate scenarios is increasingly a global priority. Models such as CESM that perform state-of-the science climate simulations are particularly vital for addressing otherwise intractable "what if?" climate questions (for example, "What if all of the ice in the Arctic melts?" or "What if ocean temperatures rise by N degrees?"), enabling better societal preparation for the future.

CESM 2.0 was released in the summer of 2018, and its popularity is a result of collaborations over several decades between scientists at the National Center for Atmospheric Research (NCAR) and various universities and research institutions. CESM is a true community model that is accompanied by a robust, extensible and portable workflow and code base that provides users a standard way to readily create model experiments and customize the experimental setup. The infrastructure allows users to easily explore and evaluate proposed science changes by creating simplified model configurations (for example, via lower resolutions or disabled feedback). Climate models are well known for pushing the limit for what is computationally feasible, and CESM's infrastructure permits the extensive testing of the model, thereby ensuring its reliability and efficiency on a broad spectrum of modern computational platforms. Establishing the trustworthiness of a code like CESM is paramount given its critical role in exploring important climate questions and defining consistency separately from bit-reproducibility is a practical necessity.

Figure 1. Example of the Bisect logarithmic search where shaded blocks represent optimized files or symbols. Unshaded blocks are from the trusted baseline compilation.



strating that the more files are optimized, the more performance it yields.

The logarithmic search in Figure 1 proceeds as follows. With all of code optimized (all the rectangles shaded), the computation runs quite fast (the speedometer is at its highest), but the results are inconsistent. Even with the left half optimized, the result is still inconsistent. Logarithmic search subdivides the left half, keeping the first two files of the left half optimized, which results in consistency. We then divide the remaining two files from the left half to test file 3 by itself. This file optimized by itself causes inconsistency and is therefore given blame. Removing file 3 from the search, we start over. In this case, we see optimizing all except for file 3 obtains consistency, therefore we have found all sites.

We framed this problem in terms of files, but after blaming files, we can perform this search again over symbols in each file (representing individual functions). The algorithm is the same but the implementation for symbols is a bit more complicated, as outlined in Bentley et al.[5]

Verifying the singleton assumption. A check is inserted in the search that provably verifies whether the singleton assumption holds.[5] In fact, as shown in this illustration, it may be possible to judiciously add back some units in an optimized mode (the last row from Figure 1) to finally leave the code highly optimized and producing acceptable answers. It would also be advantageous to obtain an overall speedup profile of one's simulation code. One such profile can be seen in Figure 3. This was obtained for an example supplied with a widely used finite element library, MFEM. From this profile, one can observe that it is possible to attain a speedup of 9.4% (compared with `gcc –O2`) with exact reproducibility, or a speedup of 39.6% with a small amount of variability.

FLiT workflow. The FLiT workflow is shown in Figure 2. A full application or a piece of it may be converted into a FLiT test. The given FLiT test, sequential or parallel (OpenMP or MPI), must be run-to-run deterministic. One must attempt to make their code as deterministic as possible before using FLiT. For example, random number generators can be seeded and MPI applications can use capture-playback (using tools like ReMPI[24]).

The FLiT test can now be compiled in various ways and run to find the compilations that cause significant differences. If one of the compilations delivers results within tolerance and has acceptable performance, the configuration search can end. For example, in Figure 3, we obtain a 9% speedup with a bitwise equal result on MFEM example 9, and if some variability can be tolerated, then the compilation with 40% speedup can be used. But when significant speedups are accompanied with unacceptable differences, the FLiT Bisect search can be used to locate the sites of variability. The FLiT Bisect search proceeds as previously described.

FLiT is a publicly released tool.[4] It has been applied to production codes within the Lawrence Livermore National Laboratory (LLNL) and has successfully located issues in the MFEM library and the Laghos application, as described earlier. FLiT Bisect first performs *File Bisect*, which proceeds as follows:

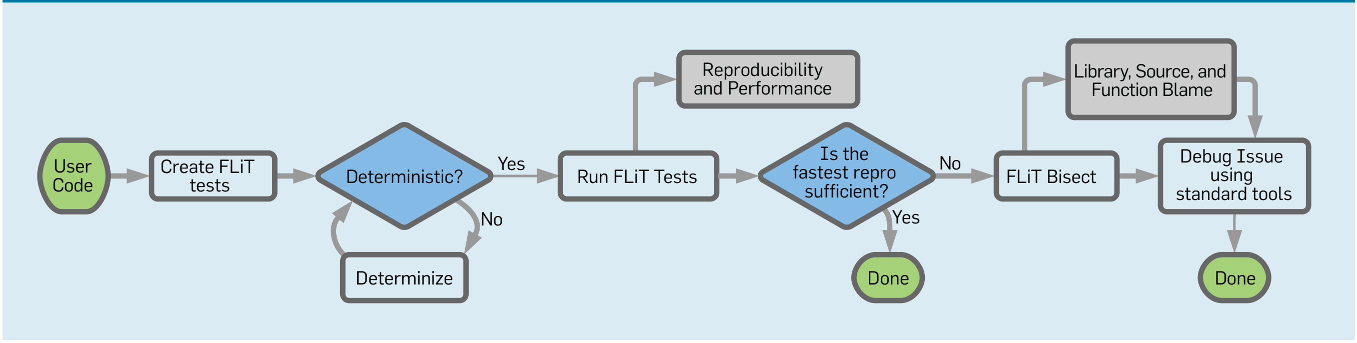
1. compile each source file into an object file using the trusted baseline compilation, and another object file using the optimization compilation under test.

2. get the next file combination to try from the logarithmic search.

3. link together the chosen object files from the two compilations to make a single executable (see File Bisect in Figure 4).

4. run this generated executable and

Figure 2. FLiT workflow.


User Code
Create FLiT tests
Deterministic?
Yes
No
Determinize
Run FLiT Tests
Reproducibility and Performance
Is the fastest repro sufficient?
No
Yes
Done
FLiT Bisect
Library, Source, and Function Blame
Debug Issue using standard tools
Done

compare with the baseline run results.

5. return the comparison to the search algorithm and repeat from (2).

The runtime of FLiT Bisect is the time it takes to run the test code times the number of file combinations and symbol combinations to be evaluated. Notice that compilation into object files happens only at the beginning. After that, FLiT simply does a link step and run for each search step. It is worth noting that FLiT Bisect also includes the capability to report how much each site is estimated to contribute to the overall result divergence.

Function-level Bisect. While File Bisect is quite useful in narrowing down the reasons for a software's non-portability, we often have to locate problems at a finer level of resolution—meaning, a single function within a file. FLiT supports this via its *Symbol Bisect* feature. As seen in Figure 4, Symbol Bisect mixes compiled functions from two different compilations of the same source file. This is performed by demoting some symbols to be weak symbols. During link-time, if there is a duplicate symbol but one is weak, then the strong symbol is kept while the weak symbol is discarded. This approach allows FLiT to search over the symbol space *after optimizations have been performed*. However, for this to be effective, the -fPIC compilation flag must be used (only on the object file to be mixed) to ensure no inlining between functions that we might want to replace occurs. FLiT checks whether using -fPIC interferes with the optimization that causes the result difference.

In practice, this modality of search has helped us successfully attribute root causes down to a small set of functions. For example, in the case of Test-13 within the MFEM library, FLiT-based testing revealed that a compiler optimization level that involved the use of AVX2, FMA, and higher precision intermediate floating-point values produced a result that had a relative difference of 193% from the baseline of g++ –O2. The L_2 norm over the mesh went from approximately 5 to 15 after the optimizations. Using Symbol Bisect, the problem was located to be within one simple function that calculates $M = M + aAA^T$, with a being a scalar, and M and A being dense square matrices. This case wasn't known to the developers of MFEM.

Figure 3. Performance profile of compilations of Example 9 from MFEM. The compilations with the fastest bitwise equal and fastest overall speeds are labeled.

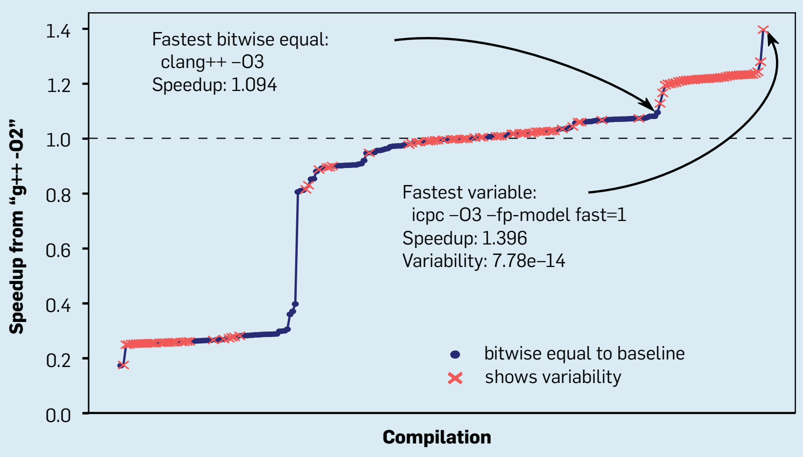
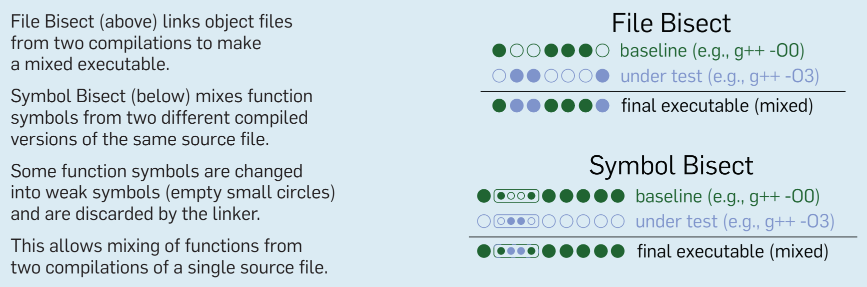


Figure 4. File Bisect and Symbol Bisect.


File Bisect (above) links object files from two compilations to make a mixed executable.

Symbol Bisect (below) mixes function symbols from two different compiled versions of the same source file.

Some function symbols are changed into weak symbols (empty small circles) and are discarded by the linker.

This allows mixing of functions from two compilations of a single source file.

Conversation with the developers of MFEM is under way to resolve this issue. This finding may indicate numerical instability of the underlying finite element method employed, or with its implementation.

Addressing the identified and located issue is outside of FLiT's scope. It is then the responsibility of the scientific sofware developer to solve the issue in order to obtain consistency and numerical stability. A designer may then choose to solve the identified non-portability either by tuning precision, rewriting the computation differently (perhaps employing more numerically stable approximations), or avoiding the problematic optimization for the whole application or the affected files.

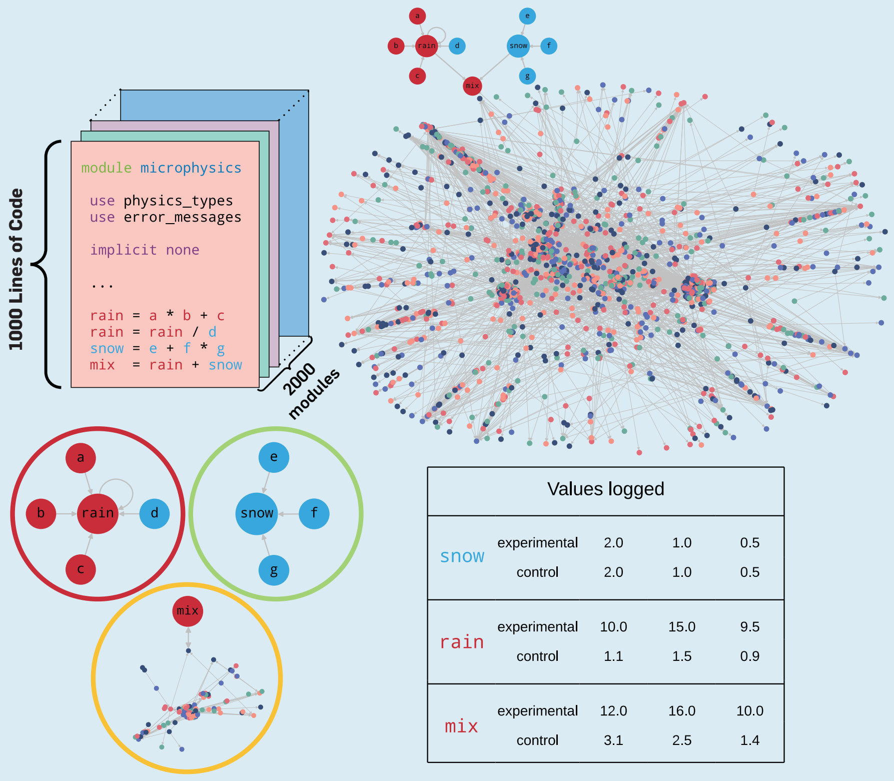
CESM

FLiT's tolerance-based approach to consistency will work for many code bases, but for applications that model complex and chaotic systems, a more nuanced method may be needed. For CESM, statistical consistency between a baseline ensemble and a set of new runs is determined by the CESM-ECT quality assurance framework. Extending the CESM-ECT to help understand why new runs are inconsistent is crucial for comprehensive quality assurance for CESM. Retaining in mind our long-term goal of impacting other large, critical applications, we now describe our recent efforts to tackle the challenge of root cause analysis of inconsistency in CESM.

The CESM-ECT has proven to be useful in terms of detecting inconsistencies that were either introduced during the process of porting the CESM software or by a new machine platform itself, both of which are not uncommon. Such sources of inconsistency can be true errors (for example, resulting from a compiler bug) or new machine instructions. However, while CESM-ECT issues a "fail" when a statistical discrepancy is identified in the new output, little useful information is provided about the possible cause.

Figure 5. The CESM root cause identification workflow.

In this example, the computation of the fictitious CESM output variable rain causes CESM-ECT failure due to an operation in the notional "microphysics" module. To find the cause, CESM-RUANDA first converts the CESM source code (top left) to a directed graph (top right). The subgraph responsible for computing rain is partitioned into communities (bottom left). Then nodes within the communities are selected for runtime sampling by their centrality. The table (bottom right) illustrates how runtime value comparison between an experimental and control case at three logged execution points (columns) can reveal the cause of the CESM-ECT failure.



```
module microphysics

  use physics_types
  use error_messages

  implicit none

  ...

  rain = a * b + c
  rain = rain / d
  snow = e + f * g
  mix  = rain + snow
```

Values logged				
snow	experimental	2.0	1.0	0.5
	control	2.0	1.0	0.5
rain	experimental	10.0	15.0	9.5
	control	1.1	1.5	0.9
mix	experimental	12.0	16.0	10.0
	control	3.1	2.5	1.4

This lack of fine-grained information can be quite frustrating for the user, who would like to know why the new run failed so that the problem can be addressed. And while debugging a large and complex code like CESM is challenging in general, some hope generally exists when the code crashes or stalls or the numerics blow up. In these situations, we often have enough information (from a large-scale debugging tool or software output) to roughly determine the source of the error. However, when trying to determine the cause of a statistical discrepancy in CESM output, it may be far from clear where (or even how) to start looking for the root cause.

Automating root cause analysis for CESM. The need for an automated tool that enables developers to trace a problem detected in CESM output to its source was felt acutely shortly after CESM-ECT was first put into use for verifying ports to other platforms (against simulations on the NCAR supercomputer). Only one of many CESM-supported platforms failed the CESM-ECT and determining the cause of the failure took several frustrating months of effort from a number of scientists and engineers to identify FMA instructions as giving rise to inconsistency (for example, see Baker et al.[3]). Ideally, a companion tool to CESM-ECT would identify which *lines of code or CPU instructions* were responsible for the failure. While tools do exist to find differences at this level, we were not aware of any that we could directly apply to a code the size and complexity of CESM. Approaches based on SAT or Satisfiability Modulo theories are precise, but often cannot handle large code bases.[25] Debugging and profiling toolkits are capable of detecting divergent values in individual variables, but the sampling process can be expensive as well. Furthermore, identifying which variables to sample is a formidable challenge. Therefore, we adopted the strategy of reducing the search space for the root cause(s) to a tractable quantity of code that would facilitate the use of tools like FLiT or KGEN[19] or runtime sampling.

We have successfully progressed toward our goal via a series of developed techniques that we collectively refer to as the CESM Root caUse Analysis of Numerical DiscrepAncy (CESM-RUANDA).[22] This toolkit parses the CESM source code and creates a directed graph of internal CESM variables that represents variable assignment paths and their properties. Based on its determination of which CESM output variables are most affected (using information from CESM-ECT), it then extracts a subgraph responsible for calculating the output variables via a form of hybrid program slicing. Next, the subgraph is partitioned into communities to facilitate analysis, and nodes are ranked by information flow within the communities using centrality. The centrality-based ranking enables either runtime sampling of critical nodes or the identification of critical modules that can be individually extracted from CESM and run as an independent kernel (for example, via KGEN). See Figure 5 for a visual depiction of CESM-RUANDA. Translating the CESM source code into a directed graph representation enables fast, hybrid analysis of information flow making it easier for other existing tools or techniques to locate problematic lines of CESM code.

As an example, CESM-RUANDA can identify internal CESM variables whose values change markedly when computed with FMA. CESM built by the Intel 17 compiler with FMA enabled generates output on the NCAR supercomputer that is flagged as a failure by CESM-ECT. After pinpointing the output variables most affected by enabling FMA instructions, CESM-RUANDA narrows the root cause search space to a subgraph community corresponding to the model atmosphere microphysics package. Examining the top nodes ranked by centrality yields several of the internal variables that take very different values with FMA enabled (Figure 6), allowing us to reach the same conclusion as the manual investigation into the failing CESM port in a fraction of the time (less than an hour on a single CPU socket). The automated identification of the root causes of discrepancies detected in CESM output provided by CESM-RUANDA will tremendously benefit the CESM community and developers.

It is important to highlight that while a CESM-ECT "fail" has a negative connotation, it is simply an indicator of statistically differentiable output. While the negative connotation is warranted for bugs, it masks a subtlety in the case of FMA. In keeping with the sidebar on floating-point arithmetic, we note that CESM-ECT does not indicate which output (with FMA or without) is more "correct" (in terms of representing the climate state). While domain experts might be able to make such a determination, the model should ideally return consistent results regardless of whether FMA machine instructions are executed. In this case, our tools seem to indicate an instability or sensitivity in portions of the code that ideally could be corrected with a redesign.

Concluding Remarks

Computational reproducibility has received a great deal of (well-deserved) attention, with publications emphasizing the reproducibility of experimental methods in systems[8] through summaries of workshops covering scientific and pragmatic aspects of reproducibility.[16] While the problems due to non-reproducibility are amply clear, there is a dearth of tools that help solve day-to-day software engineering issues that impact software developers as well as users.

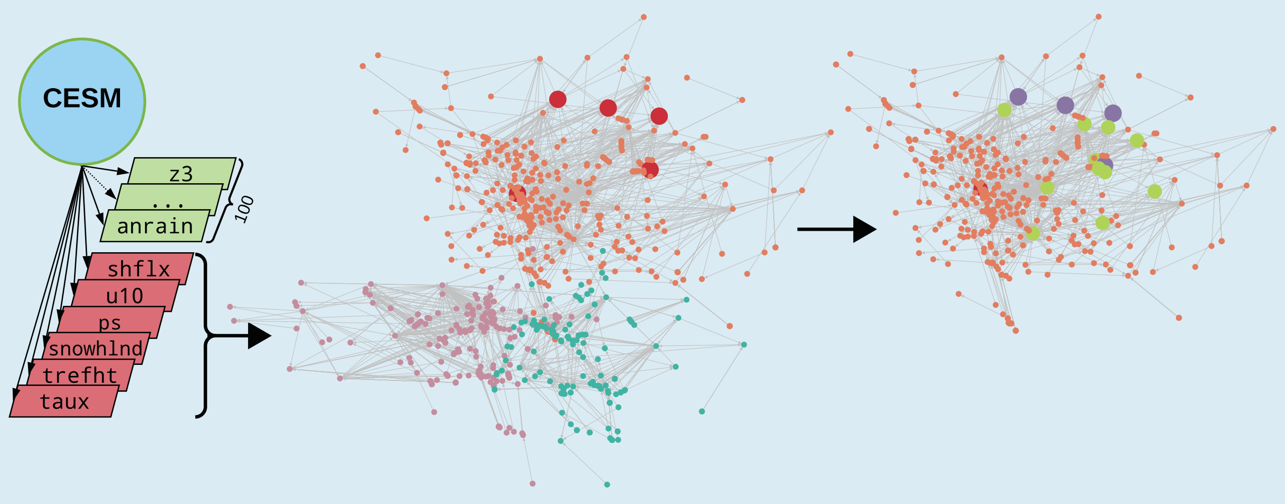
In this context, our specific contribution in this paper has been a two-pronged approach that allows domain scientists to act on reproducibility problems related to porting software across machines and compilers. Our first specific contribution is FLiT—a tool that can be applied to real-world libraries and applications when they exhibit non-reproducible behavior upon changing compilers or optimization flags. Our second contribution are the CESM-ECT and CESM-RUANDA tools that have approached the problem on a very large scale and in the context of climate simulation software.

While much more work remains to be done on both tools, the anticipated usage model is to first use CESM-ECT to determine that a discrepancy exists, then employ CESM-RUANDA to narrow down the scope of the problem (in codebases exceeding several million lines of code) to specific variables whose values differ significantly, and finally attribute the root cause to individual files or functions via tools such as FLiT. The efficacy of FLiT was demonstrated on the MFEM code which occupies over 100K lines of code and consists of 2,998 functions spread over 97 source files. With such non-trivial code sizes already handled via FLiT, we believe that a combination of these tools will quite naturally lead to an overall superior diagnostic process.

Building a community is essential. To help increase the list of tools and approaches in this area, we are eager to engage in collaborations in two primary directions. First, the FLiT tool is available publicly at https://github.com/PRUNERS. We are open to developing

Figure 6. A schematic representation of CESM-RUANDA applied to the problem of finding variables most affected by FMA instructions.[22]

Of the more than 100 atmosphere output variables used in the CESM-ECT, six are related to the failure (left). The CESM subgraph that computes these six variables is represented by the center plot, where node color designates community membership. Note that we render a smaller subgraph than that produced in Milroy et al.[22] for illustrative purposes. The large red nodes in the center plot represent five variables most affected by FMA instructions. In the rightmost plot the community containing these five variables is isolated and nodes are selected for runtime sampling by their centrality. Large green nodes are those chosen for sampling and purple nodes are variables sensitive to FMA which are also selected for sampling. All but one red node from the center plot would be identified by CESM-RUANDA.



FLiT with external input, collaborations, and feature requests. Second, ideas centered around the CESM-RUANDA are ripe for re-implementation, and at NCAR, we are open to supplying computational kernels from the publicly available CESM code to the community. The ideas as well as results behind FLiT and CESM-RUANDA are described in greater detail in Bentley et al.[5] and Milroy et al.,[22] respectively. To further help with community building, we have recently contributed a collection of open-source tools as well as conference tutorials that help pursue many of the issues surrounding floating-point precision analysis, tuning, and exception handling; these are available for perusal at http://fpanalysistools.org.[12]

In summary, the integrity of computational science depends on minimizing semantic gaps between the source level representation of simulation software and its executable versions. Such gaps arise when hardware platforms change, libraries change, and compilers evolve. These changes are necessitated by the need to maintain performance in the present post Dennard scaling era. Furthermore, the pace of these changes is only bound to increase as the designer community is highly engaged in squeezing out the last drop of performance from current generation (as well as upcoming) machines and runtimes. Therefore, the onus of computer science researchers is not only to minimize or avoid these gaps through formally verified compilation methods (for example, Compcert[9]), develop tools that discover and bridge these gaps, and also make fundamental advances that contribute to reproducibility (for example, recent contributions to the IEEE-754 standard in support of reproducible arithmetic operations.[26]).

Digital content available for inclusion with this article. Sources and detailed instructions to install and use the FLiT software system on a worked-out example of debugging a scenario within the MFEM finite element library is available from http://fpanalysistools.org.

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. The CESM project is supported primarily by the National Science Foundation (NSF). This material is based upon work supported by the National Center for Atmospheric Research, which is a major facility sponsored by the NSF under Cooperative Agreement No. 1852977. Computing and data storage resources, including the Cheyenne supercomputer (doi:10.5065/D6RX99HX), were provided by the Computational and Information Systems Laboratory (CISL) at NCAR. C

Refer ences

1. Baker, A.H. et al. A new ensemble-based consistency test for the community earth system model. *Geoscientific Model Development 8*, 9 (2015), 2829–2840; doi:10.5194/gmd-8-2829.
2. Baker, A.H. et al. Evaluating statistical consistency in the ocean model component of the Community Earth System Model (pyCECT v2.0). *Geoscientific Model Development 9*, 7 (2016), 2391–2406; https://doi.org/10.5194/gmd9-2391-2016.
3. Baker, A.H., Milroy, D.J., Hammerling, D.M., and Xu, H. Quality assurance and error identification for the Community Earth System Model. In *Proceedings of the 1st Intern. Workshop on Software Correctness for HPC Applications*. ACM, New York, NY, USA, 8–13; https: //doi.org/10.1145/3145344.3145491.
4. Bentley M. and Briggs, I. FLiT Repository, 2019; https://github.com/PRUNERS/FLiT.git
5. Bentley, M. et al. Multi-level analysis of compiler-induced variability and performance trade-offs. In *Proceedings of the 28th Intern. Symp. High-Performance Parallel and Distributed Computing*. ACM, 2019, 61–72; https://doi.org/10.1145/3307681. 3325960
6. Chiang, W-F, Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G. and Rakamaric, Z. Rigorous floating-point mixed-precision tuning. In *Proceedings of the 44th ACM SIGPLAN Symp. Principles of Programming Languages*. G. Castagna and A.D. Gordon (Eds.). (Paris, France, Jan. 18–20, 2017). ACM, 300–315; https://doi.org/10.1145/3009837
7. Clune, T. and Rood, R. Software testing and verification in climate model development. *IEEE Software 28*, 6 (2011), 49–55; https://doi.org/10.1109/MS.2011.117
8. Collberg, C.A. and Proebsting, T.A. Repeatability in computer systems research. *Commun. ACM 59*, 3 (Mar. 2016), 62–69; https://doi.org/10.1145/2812803
9. Compcert. The Compcert Project, 2019; http://www.compcert.inria.fr
10. Easterbrook, S.M., Edwards, P.N., V. Balaji, V. and R. Budich, R. Climate change: Science and software. *IEEE Software 28*, 6 (2011), 32–35.
11. Eyring, V., Bony, S., Meehl, G.A., Senior, C.A., Stevens, B., Stouffer, R.J., and Taylor, K.E. Overview of the Coupled Model Intercomparison Project Phase 6 (CMIP6) experimental design and organization. *Geoscientific Model Development 9*, 5 (2016), 1937–1958; https://doi.org/10.5194/gmd-9-1937-2016.
12. fpanalysistools.org. Tutorial on Floating-Point Analysis Tools; http://fpanalysistools.org/
13. Hurrell, J. et al. The Community Earth System Model: A framework for collaborative research. *Bulletin of the American Meteorological Society 94* (2013), 1339–1360; https://doi.org/10. 1175/BAMS-D-12-00121.1
14. Intel. BFLOAT16—Hardware Numerics. White Paper, Document Number: 338302-001US, Revision 1.0, 2018; https://intel.ly/36IJ37r.
15. IPCC 2019. Intergovernmental Panel on Climate Change; http: //www.ipcc.ch/about.
16. James, D. et al. Standing Together for Reproducibility in Large-Scale Computing: Report on reproducibility@XSEDE. CoRR abs/1412.5557 (2014), 16. arXiv:1412.5557; http://arxiv.org/abs/ 1412.5557.
17. Kahan, W. Lecture notes on the status of IEEE Standard 754 for binary floating-point arithmetic,1997; https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF
18. Kahan, W. How futile are mindless assessments of roundoff in floating-point computation? 2006; https://people.eecs.berkeley.edu/~wkahan/Mindless.pdf
19. Kim, Y et al. KGEN: A Python tool for automated Fortran kernel generation and verification. In *Proceedings of the 2016 Intern. Conf. Computational Science 80*, 1450–1460; https://doi.org/10.1016/j.procs.2016.05.466.
20. Menon, H., et al. ADAPT: Algorithmic Differentiation Applied to Floating-point Precision Tuning. In *Proceedings of the 2018 Intern. Conf. High Performance Computing, Networking, Storage, and Analysis*, Article 48. IEEE Press, Piscataway, NJ, USA; https://doi.org/10.1109/SC.2018.00051.
21. Milroy, D.J., Baker, A.H., Hammerling, D.M. and Jessup, E.R. Nine time steps: Ultra-fast statistical consistency testing of the Community Earth System Model (pyCECT v3.0). *Geoscientific Model Development 11*, 2 (2018), 697–711; https://doi.org/10. 5194/gmd-11-697-2018.
22. Milroy, D.J., Baker, A.H., Hammerling, D.M., Kim, Y., Jessup, E.R., and Hauser, T. Making root cause analysis feasible for large code bases: A solution approach for a climate model. In *Proceedings of the 28th Intern. Symp. High-Performance Parallel and Distributed Computing*. ACM, 2019, 73–84; https://doi.org/10.1145/3307681.3325399.
23. Pipitone, J. and Easterbrook, S. Assessing climate model software quality: A defect density analysis of three models. *Geoscientific Model Development 5*, 4 (2012), 1009–1022; https: //doi.org/10.5194/gmd-5-1009-2012.
24. PRUNERS. FLiT and ReMPI Projects page, 2019; https://pruners.github.io/flit/
25. Biere, A., Huele, M., van Maaren, H. and Walsh, T. *Handbook of Satisfiability*. IOS Press, 2008.
26. Riedy, E.J. and Demmel, J. Augmented arithmetic operations proposed for IEEE-754 2018. In *Proceedings of the 25th IEEE Symp. Computer Arithmetic* (Amherst, MA, USA, June 25–27, 2018), 45–52; https://doi.org/10.1109/ ARITH.2018.8464813.
27. Rubio-González, C. et al. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the Intern. Conf. High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA, Nov. 17–21, 2013). W. Gropp and SS. Matsuoka (Eds.). ACM, 27:1–27:12; https: //doi.org/10.1145/2503210.2503296.
28. Small, R.J. et al. A new synoptic scale resolving global climate simulation using the Community Earth System Model. *J. Advances in Modeling Earth Systems 6*, 4 (2014), 1065–1094; https: //doi.org/10.1002/2014MS000363.
29. Solovyev, A., Baranowski, M.S., Briggs, I., Jacobsen, C., Rakamarić, Z. and Gopalakrishnan, G. Rigorous estimation of floating-point round-off errors with Symbolic Taylor Expansions. *ACM Trans. Program. Lang. Syst. 41*, 1, Article 2 (Dec. 2018); https://doi.org/10.1145/3230733.
30. Zeller, A. and Hildebrandt, R. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Engineering 28*, 2 (2002), 183–200.

Dong H. Ahn is a computer scientist at the Lawrence Livermore National Laboratory, Livermore, CA, USA.

Allison Baker is Project Scientist III in the Computational Information Systems Laboratory, National Center for Atmospheric Research, Boulder, CO, USA.

Michael Bentley is pursuing a Ph.D. at the School of Computing, University of Utah, Salt Lake City, UT, USA.

Ian Briggs is pursuing a Ph.D. at the School of Computing, University of Utah, UT, USA.

Ganesh Gopalakrishnan is a professor at the School of Computing, University of Utah, Salt Lake City, UT, USA.

Dorit Hammerling is an associate professor in the Department of Applied Mathematics and Statistics, Colorado School of Mines, Golden, CO, USA.

Ignacio Laguna is a computer scientist at the Lawrence Livermore National Laboratory, Livermore, CA, USA.

Gregory L. Lee is a computer scientist at the Lawrence Livermore National Laboratory, Livermore, CA, USA.

Daniel Milroy is a postdoctoral researcher at the Lawrence Livermore National Laboratory, Livermore, CA, USA.

Mariana Vertenstein leads the CESM Software Engineering Group (since 2004) in the Climate and Global Dynamics Laboratory, National Center for Atmospheric Research, Boulder, CO, USA.