FLOAT: Framework for Workflow Analysis, Visualization and Transformation

John Jacobson* University of Utah
Mike Bentley† University of Utah
Cayden Lund‡ University of Utah
Ganesh Gopalakrishnan§ University of Utah
Ignacio Laguna¶ Lawrence Livermore National Laboratory
Gregory L. Lee‖ Lawrence Livermore National Laboratory

ABSTRACT

Continued progress in high-performance computing requires the constant creation of new workflows and frameworks to support important activities such as compilation and linking. Once such frameworks mature, visualization tools can help oversee and ensure the integrity of such frameworks and obtain valuable feedback. This paper describes our experience creating such a light-weight framework out of a previous tool effort FLiT for detecting compiler-induced numerical variability. The resulting framework FLOAT has already helped us better understand and fix performance bugs in FLiT. Our design of FLOAT and the ways in which we anticipate it enabling the adoption and re-purposing of FLiT are described, including support for accelerator-based programming and other heterogeneous build workflows.

Index Terms: Software Testing; Visualization; Floating-Point Arithmetic; Compiler Optimizations; Bisection Search

1 INTRODUCTION

With the increasing scale and heterogeneity of HPC software, the process of producing an executable binary file for a complex application has become quite involved. A developer frequently ends up compiling and linking 1000's of files with various libraries, running the resulting executable across numerous inputs, and then validating the results in a variety of ways. This involves tuning a number of compilation settings, sometimes even at source-file granularity, as well as designing build systems, and reserving the necessary computational resources for both compilation and execution. This workflow becomes even more complicated when porting applications across machines and compilers. In existing prior work [3], the authors identify *result reproducibility* concerns using bisection methods which require many levels of compilation, execution, and diagnosis, adding a new dimension of complexity to this standard workflow. Further, with increasing use of GPUs and other accelerators, these workflows will enter into the murkier territory of multiple vendor-provided GPUs and their looser and highly varied numerical specifications [8].

In many instances of creating an executable, there is an initially planned workflow—*an execution model*—and the resulting concrete code—*an implementation*—that is quite involved. Additionally, automated testing frameworks *generate* their own complex (and often recursive) workflows that add many "hidden details" to the implemented workflow. Many things can (and do) go wrong in the face of such complexity. Accomplishing a satisfactory final executable is now the culmination of a complex collection of tasks orchestrated by a concrete workflow designed from a model that lies implicit within the build system code.

Another aspect to workflows is that the computing environments as well as humans are imperfect. One might have the right mental model to accomplish a compilation/build but produce an inconsistent build script; one might forget to state some dependencies, or at build-time some of the compilations might silently fail resulting in an unintended binary having been created and run. Bugs may creep in (unstated dependencies, duplicated executions, build sub-calls silently dying, etc.) Clearly there is need for automation.

There are two directions to take when considering automation: (1) take an existing workflow system (e.g. Pegasus [7]) and adapt it to one's needs, or (2) organically "grow" a customized workflow system for the immediate needs, and allow it to generalize it to the correct level (overly generalized systems often fail to address specific domain needs well). Taking the latter direction, our contribution is FLOAT, a framework under construction for the capture, analysis, and transformation of workflows manifested during the build, run, test, and debug processes within complex heterogeneous workflows. There are systems out there (e.g., Spack [5] for package management and associated installation challenges) that tend to reinforce the virtues of not over-generalizing.

The need for our framework FLOAT was highlighted by lessons reported by the FLiT project [3, 6]—a tool for multi-level analysis of compiler-induced variability and performance tradeoffs. FLiT has been applied to many in-house applications at LLNL. FLiT has also recently been extended to CUDA applications which introduce another layer of compilation complexity via the proprietary NVCC compilation wrapper.

Through all this, it became clear to us that offering a tool such as FLiT without a complementary tool such as FLOAT would be a poor decision. The main impediment is that tools such as FLiT would be difficult meaningfully to hand over to others: the adopters would want to customize it for their uses, and unfortunately FLiT does not truly expose its inner workings in a manner that can be captured through dependency annotations. It also does not report the details of the work it carries out nor provide opportunities for intervention, repair, and resumption. FLOAT on the other hand reveals exactly what the internal activities of a tool such as FLiT are, and this knowledge can help understand and adapt to a local context. Further, while developing FLOAT, we found a significant performance bug in FLiT's implementation of the conceptual model (we detail this in §4.)

Roadmap: In this paper, we will primarily take the point of view of FLOAT serving the needs of a tool such as FLiT. §2 presents background and related work. §3 presents the overall design of FLOAT, walking through an example. §4 presents some results obtained using FLOAT. Conclusions and future work are in §5, where we also touch on the higher level perspective of FLOAT.

*e-mail: john.jacobson@utah.edu
†e-mail: mbentley@cs.utah.edu
‡e-mail: cayden.lund@utah.edu
§e-mail: ganesh@cs.utah.edu
¶e-mail: ilaguna@llnl.gov
‖e-mail: lee218@llnl.gov

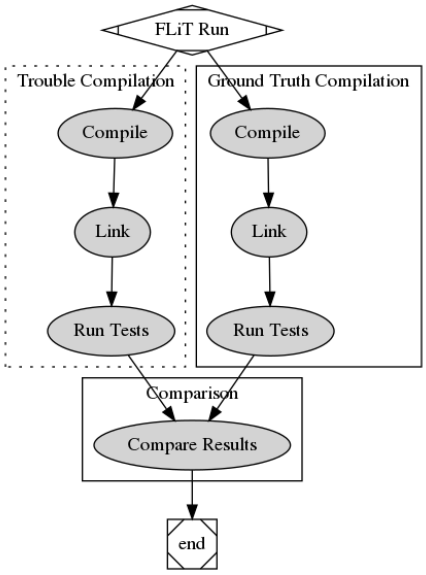

Figure 1: Abstract model of FLiT control flow, capturing all high-level tasks which form a basic FLiT run. Although the implementation involves the use of 3 languages to dynamically create and execute build systems, these few tasks are sufficient for a useful analysis of the programs design.'

```
def match_Linking(e, p):
    return p['obj'] in \
        [f for f in e['src'].split()] \
        and e['compiler'] == p['compiler'] \
        and e['opt']      == p['opt'] \
        and e['switches'] == p['switches']

link_def = {
        "Parent"       : ['root'],
        "Dependencies" : ['Compile',
                          'Baseline Compile',
                          'Compile fPIC']
        "Matching"     : match_Linking
    }

event_definitions['Linking'] = link_def
```

Figure 2: Python code to define the 'Link' task represented in Fig. 1. It is not a sub-task of any other defined task, thus is nested within the root. Linking object files depends on the compilation of those files into object files, hence the possible parent events. Lastly, the "Matching" function is used to determine exact parents of a unique Link event.

2 Background, Related Work

FLOAT was motivated by the need for further development on the FLiT tool. FLiT is a testing framework for identifying floating-point variability caused by compiler optimizations or changes in hardware and execution environments. In particular, FLiT's bisection search allows the user to isolate such variability to symbol (or function) granularity, greatly reducing the scope of troubleshooting large applications. Preliminary implementation of GPU file bisection has also been recently completed. This gives FLiT the ability to analyze heterogeneous CPU/GPU programs in future.

FLiT is realized as a "meta-build system" in which the target application is built and tested under any number of conditions specified by the user. These conditions are specified by the user in the form of a TOML configuration file enumerating the desired build parameters and their own test functions to define the specific inputs and execution parameters. Provided these files, FLiT creates a search space of compilation parameters (here a compilation is defined as a triple of [compiler, optimization level, compiler flags]) and generates a (recursive) GNU Make build system for generating and executing all defined variations of the target application. The user-defined test cases are executed under each compilation, and results are compared with a trusted baseline execution to determine variability in floating-point results and performance.

Variability within a compilation of interest identified through a FLiT execution may then be further dissected with FLiT bisect. Using a variation of delta-debugging, FLiT bisect links object files from a "trouble compilation" with other files from the baseline compilation, creating mixed-compilation executables which are tested for variability from the trusted results. The results guide the search through continued re-linking of trouble and baseline object files until individual variability-inducing files are isolated.

Once a single file is found to induce variability, the bisect search proceeds with another round of delta-debugging inspired search by creating a mixed-compilation object file. This is achieved by linking two variations of the trouble file; one file from the trouble compilation, and one from the trusted baseline. Each symbol within both files is marked as weak within one copy, and conversely as strong within the other file. When linked, the resulting executable will retain only strong symbols from each file, and variability in the resulting executable is blamed on the set of strong symbols from the trouble compilation file. This leads to further bisection of the set of blamed symbols, and the search continues until blame is assigned to individual symbols from the original trouble file.

As all of these steps are achieved through dynamically generated build systems, the resulting execution trace of FLiT runs quickly become complex and difficult to analyze, particularly for large applications. Further, the trusted build system profile for target applications often require significant time on their own so the FLiT search through multiple builds has significant performance concerns for adoption on large HPC applications, where the tool is most needed. FLOAT was developed in order to understand the performance bottlenecks within FLiT and guide further development of the application in the face of this complexity.

Instead of targeting statement or function level analysis within FLiT, FLOAT aims to model the underlying conceptual model of a test framework and project this model onto actual execution traces to verify implementation of the conceptual model, as well as to identify higher-level design enhancement opportunities within the application. This methodology greatly eases the burden of analyzing interoperable programs like FLiT, which since little to no communication between separate language components is necessary. Also, by tracing execution at a higher level of abstraction, the volume of data is significantly reduced and their mapping to well-understood design abstractions allows for more natural digestion and processing by the developer, and suggests a number of pre-defined visualizations and results to be provided to the user. As stated earlier, workflow management systems such as Pegasus [7] and FLUX [4] are popular in scientific computing. Although one might imagine their use in our context, this approach would not result in a light-weight and domain-focused framework.

3 Design of FLOAT

FLOAT is designed to represent a high-level abstraction of a programs design for analyzing performance and correctness from an overall design perspective. The user models their application as a task dependency graph, where tasks are arbitrary blocks of functionality within the application which are not necessarily tied to the structure of code implementation. This model is then mapped to

```
{
    "date" : "Tue Aug 10 13:41:18 MDT 2021",
    "time" : 1628624478570859473,
    "name" : "Compile",
    "type" : "stop",
    "properties" : {
        "file" : "tests/Mfem13.cpp",
        "obj" : "obj/GCC_MFMA_O3/Mfem13.cpp.o",
        "compiler" : "g++-7",
        "switches" : "-mfma",
        "opt" : "-O3"
    }
}
```

Figure 3: Sample event, capturing a specific 'Compile' task instance. The top layer of items are required for all events, while the contents of the "properties" object allow capture of arbitrary fields. These fields are primarily used for building unique event relationships.

the code by manual instrumentation of log calls, after which the resulting logs are processed into a DAG data structure guided by the abstract model. This process allows a high-level view of the implementations conformity to the defined model, concurrency potential and utilization, and critical path analysis.

The model is initially defined by a Python dictionary as shown in Fig. 2.[1] The model defines a set of events by unique string identifiers, each representing a unique task within the application design. (See Fig. 3.)

For example, within the FLiT framework the complete compilation of an executable may be defined as one task including preprocessing, compilation, assembly, and linking as a single event. Alternatively, these sub-tasks may be modeled as individual tasks (as determined by the user.) The actual implementation in FLiT tracks two tasks within compilation; pre-processing, compilation, and assembly of an object file are treated as a single task, while linking is tracked as a second independent task. This particular definition was important in our use-case for FLiT Bisect where initially each file is compiled in two variants (unoptimized versus optimized), but later linked in multiple combinations to detect non-reproducible situations (e.g., one file has a function whose numerical behavior changes unacceptably upon optimization). While a logarithmic search might require only $log(N)$ linkages for N files, a true delta-debug session might potentially require an exponential number of linkages. The latitude to define sub-tasks flexibly may prove to be important for adopters of FLOAT. Here are additional details of our design:

Event Hierarchy For each event defined in our framework, a list of parent events and a "Matching" function containing logic for matching this event to its parent event(s) is expected. As an example, within the FLiT framework source files are compiled multiple times, using different flags, to produce multiple executables. As such, there are several object file linking tasks performed (one for each produced executable), but each linking task depends only on source files compiled with the relevant flags. As such, it is necessary in both the **linking** and its parent **compilation** events to capture unique identifiers (in this case compilation optimization level plus all flags and switches.) The function `match_Linking` shown in Fig. 2 demonstrates the logic necessary to identify which source compilation tasks are linked by a specific linking instance during runtime.

[1]Technically the model is not required, but any events not included in the model will be collected as an unorganized set of events, i.e. plain json logs with no additional functionality.

Event Structure Each of these events is then captured by instrumenting logging within the source files. JSON format is used for its human-readability and ubiquity. The captured logs require only (1) their unique event identifier (such as "Compilation" within FLiT), (2) a timestamp, (3) a start/end identifier, and (4) a JSON string containing relevant properties for uniquely identifying each separate instance of this event and to match this event to its parent(s) and/or children, as shown in Fig. 3.

Log-file Structure and Visual Elements Once the event definitions and collected logfiles have been provided, FLOAT parses the logs and generates a DAG data structure using the Python NetworkX library [1]. During parsing, any events not conforming to the provided model will be explicated to the user. The generated DAG is available to the user for manual processing, as well as a number of built-in analyses. A Gantt chart demonstrating relative runtimes of captured events, such as in Fig. 6, provides an at-a-glance view of utilized concurrency and bottleneck tasks. A graph visualization of the entire graph structure for the underlying model is available, as well as a similar representation including actual runtime event details captured, as shown in Fig.'s 1 & 5. Event specific details captured within the logs are also queryable for isolating runtime variability.

Use-case: FLiT applied to MFEM We now walk through the use of FLOAT using an example of FLiT as applied to MFEM [2], a popular finite-element library. MFEM was one of the examples analyzed using FLiT [6]; in the following, we highlight the added insights to FLiT offered by FLOAT.

The first step in application of FLOAT is design of the abstract model underlying the target application (in this case the FLiT testing framework.) Despite the inherent complexity of the recursive build system FLiT utilizes to explore the compilation search space, the framework relies on a simple conceptual model. As shown in Fig. 1, FLiT may be understood as a simple workflow for compiling and executing the test application. Using this abstraction, a FLiT run may be modeled by only seven unique tasks (and five distinct tasks suffice as ground truth processing may be identified from only one unique task in the compilation branch.)

With the model defined conceptually, we translate to a specification for automated processing. FLOAT works from a Python dictionary whose keys represent unique task names, and associated values are dictionaries of dependency relationship information (see Fig. 2.) Dependency relationships are defined by three parameters:

1. A unique "Nested Parent" - This is a super-task which encapsulates the defined task. This allows for isolated analysis of tasks by relating them to sub-tasks defined within their scope ("nested within" the scope of their Nested Parent.) All tasks are nested within the overall execution (defined in the DAG as the root node, or "FLiT Run" in Fig. 1.)

2. A list of "Parents" - These are tasks on which the defined task depend logically. For example, in Fig. 1 Compile tasks are parents of Link tasks, since the linkage phase of compilation depends on compiled object files from the Compile task.

3. A "Matching" function - This function takes an instance of the defined task (an event) and an instance of a potential parent task, and determines whether the event defined logically depends on the potential parent. For example, in 1 the trouble compilation Link task depends only on Compile tasks for the trouble compilation and not on Compile tasks for the ground truth compilation. The "Match" function distinguishes this relationship using data captured within specific log events.

This specification is defined by a Python script containing the necessary definitions in a dictionary; our implementation of the "Link" task is shown in 2.

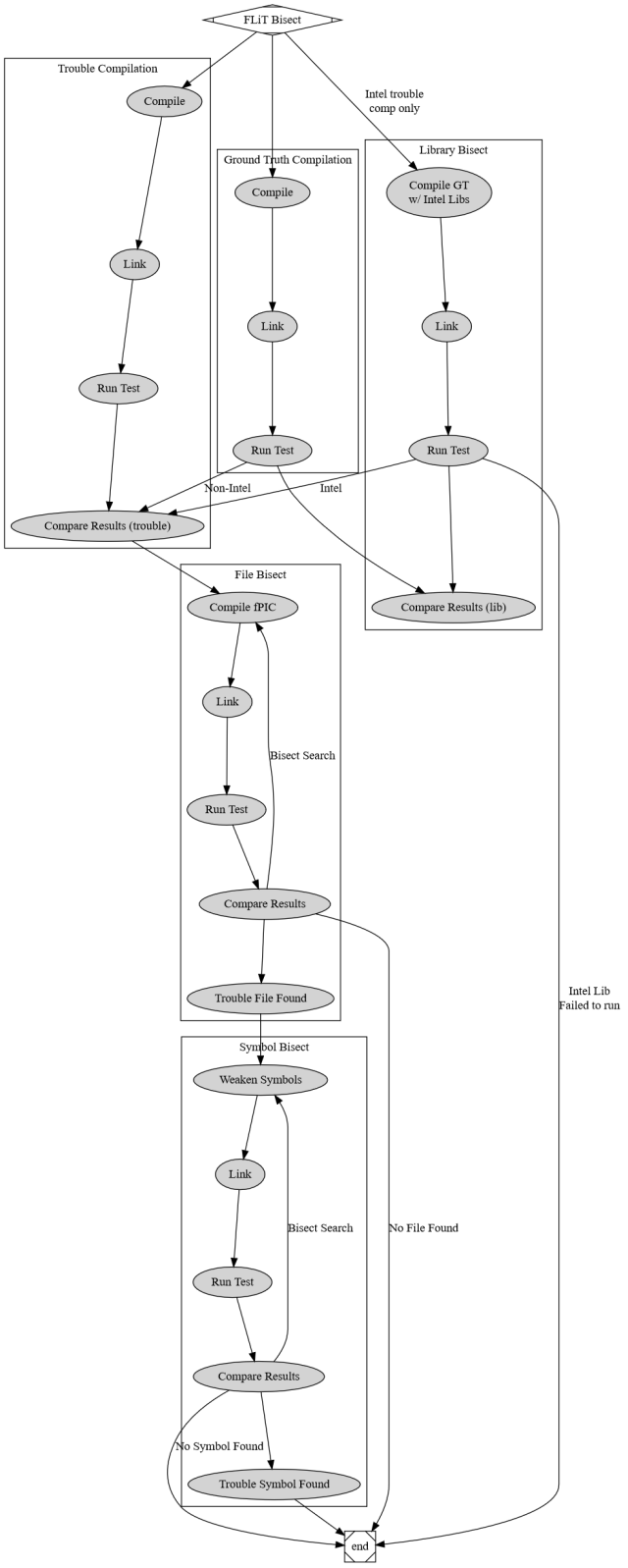

Figure 4: Abstract model of FLiT Bisect control flow. 2 layers of bisection search increase the model complexity a small amount relative to the necessary implementation intricacy.

With the abstract model specified in this way, we instrument the capture of log events representing each logical task defined. Log events are captured in JSON format conforming to a simple specification (as seen in Fig. 3.)

- Date (string)
- Time (integer), a timestamp for performance analysis
- Name (string), a unique task identifier for each task within the model
- Type (string), "start" or "stop" for event duration demarkation
- Properties (JSON string), a JSON object containing arbitrary fields deemed necessary by the user. These should capture necessary runtime details for relating specific events to their runtime dependencies.

The instrumentation of logging in the target application is left to the user. FLiT is implemented using 3 distinct languages; Python is utilized for the command-line interface (as well as implementing the dynamic build system in FLiT Bisect), GNU Make is used to build the test application, and C++ is used for test implementation and floating-point code. As such, our logs for capturing runtime event data were defined with a simple logging function in each language and calls to these functions were manually placed within the source code.[2]

Finally, with a collection of logfiles captured from an execution of FLiT on the MFEM tool, along with the model definition dictionary, FLOAT parses the logs and constructs a NetworkX DAG data structure. The provided model guides the creation of the data structure, so ill-defined dependency relationships are identified and flagged to the user as exceptions. The resulting object is provided to the user for manual analysis, alongside a number of built-in functions for analyzing the data.

In this example test case, we compare the results of executing MFEM's example Test-13 under two compilers (g++ and clang), with 3 flags (`--ffast_math`, `--funsafe-math-optimizations`, and `--mfma`) all under `-O3` forming a small search space of *six distinct compilations*, alongside a ground truth compilation using g++ in `-O2`. In Figure 5 one may see the six distinct compilation branches. This concrete execution trace DAG is provided in the form of a Gephi graph specification for interactive visualization and graph processing.

Additionally, Figures 6 show relative execution time for all events captured in the execution using `-j4` and `-j` respectively.

4 RESULTS

Our use of FLOAT yielded a few interesting results:

1. Defining the abstract model and workflow dependencies becomes more difficult with added functionality as compared to the original design.
2. Capturing the defined model required only a handful of log capture points, but these points were enough to identify a significant performance bug in the implementation.
3. Applying FLOAT even to very simple, pathological test cases for the FLiT tool are enough to identify, and quantify, performance bottlenecks within the implementation and guide future development.

The FLiT framework follows a fairly simple model; search over the space of compilations and run an executable for each. FLiT's Bisect

[2]Practices such as Aspect-Oriented Programming might help automate this in future.

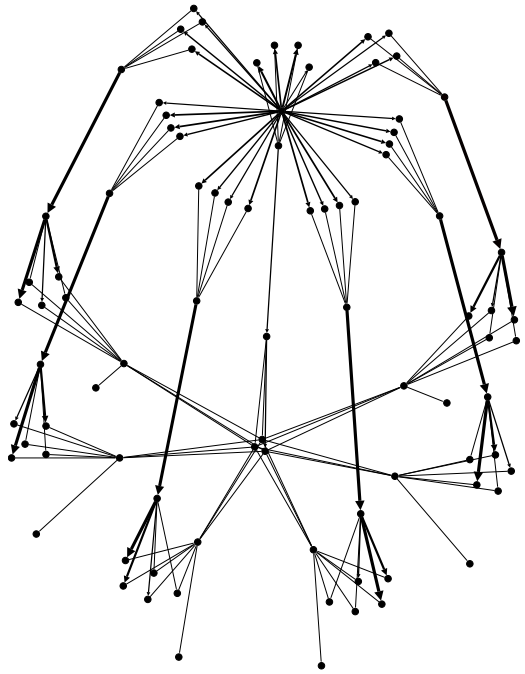

Figure 5: After the user instruments logs to capture run-time events for each task described within the abstract model, the logs are read into FLOAT. An implementation DAG is generated using Gephi displaying unique event relationships for analysis. Edge-weights represent run-times, and edge/node labels are removed here due to size constraints. All event details captured by user-implemented logs are available within the data structure provided by FLOAT.

functionality isn't much more complex from this level and can be viewed as adding two layers of bisection search. Despite this, the implementation required changes to the underlying model within FLOAT which are in some ways unnecessarily complicated. In the case of FLiT Bisect, many components of FLiT testing are reused, but to expedite the search process a number of branch conditions are created which change the dependencies of certain events. As an example, in a single FLiT Bisect run it is necessary to compile files using the `--fPIC` flag for the symbol bisect stage. Depending on whether Bisect is run on a single compilation or against a set of compilations these compilations may depend on different tasks within the FLiT model, where logically the compilation of these files has no dependency within the other defined tasks.

With the model constructed, implementing data capture necessary to feed FLOAT is quite simple; since the captured tasks are high-level abstractions there is generally no need to search through complex stack traces or nested functions and definitions as they are usually launched from top-level function calls. Once captured we immediately found a violation of the constructed model for FLiT in that the ground-truth test compilation was being executed redundantly; due to the complexity of the implemented recursive Make build system, a small technical oversight caused the executable to be forcibly run each time a separate test executable was run. Aside from doubling the number of tests being run, the ground truth executable is compiled as a trusted baseline usually with with fewer compiler optimizations so that it is generally one of the worst-performing compilations in the search space.

By only tracing high-level tasks we were able to quickly identify a general performance model for the FLiT tool. With a goal of improving the overall design of the FLiT framework one may quickly analyze the critical path of a particular execution, or verify at-a-glance in the provided visualizations the tasks which occupy

Figure 6: Event timing from a FLiT run using '-j4' to limit to 4 jobs. A missing bar represents a duration smaller than the charts resolution; this is caused by some tests being disabled in the test run.

significant runtime. This allows one to isolate those tasks which deserve deeper analysis, and targeted profiling of these tasks can take place in a more narrow scope for higher return on invested development time.

5 Conclusions, Future Work

We have motivated the need for a flexible and customized workflow analysis and transformation framework. In addition to helping enhance trust aiding in adoption of complex applications, FLOAT also helps reveal potential deficiencies of an existing workflow. FLOAT has already played an important role in enhancing the FLiT tool as well as detecting serious performance bugs lurking within it. From our walk-through in §3, additional insights provided by FLOAT in enhancing a user's trust should be apparent: capturing the abstract model behind an application and mapping it onto the execution allows for much simpler and more focused visualizations of the applications workflow. As we gain more experience with FLOAT, we will be making improvements to its specification mechanisms and validation mechanisms, as outlined under future work.

Future Work

Some of the anticipated future directions of FLOAT are as follows. First, we are developing ways to optimize floating-point codes through precision tuning (e.g., [11]) and selective expression rewriting (e.g., [10]). The search-space in these cases will be the execution differences between the original (say, higher) precision- and the new (lower precision) codes.

Second, we are interested in input (test) generation for CPU and GPU codes. In this case, the search-space will be the behavior under the initial tests, and (after monitoring coverage) additional tests administered to enhance coverage. Our initial results in this area have been published [9] and are planned to be integrated into FLOAT along with facilities to visualize heterogeneous compilations and optimizations.

Given that compilation itself is a rich domain, it may pay to focus FLOAT to excel in visualization support for this domain. For instance, if static analysis methods were to be used to augment dynamic analysis, additional tasks as well as dependencies and event-types will have to be accommodated into FLOAT. In all these cases, having a framework such as FLOAT will retain the advantages of workflow analysis and improvement.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE- AC52-07NA27344 (LLNL-CONF-832228), DOE ASCR Award Number DE-SC0022252 and NSF CISE Awards 2124100 and 1956106, and LLNL subcontract B640720 (PI Dong H. Ahn, LLNL during 2021).

References

[1] NetworkX.

[2] MFEM: Modular finite element methods library. `mfem.org`, 2018. doi: 10.11578/dc.20171025.1248

[3] D. H. Ahn, A. H. Baker, M. Bentley, I. Briggs, G. Gopalakrishnan, D. M. Hammerling, I. Laguna, G. L. Lee, D. J. Milroy, and M. Vertenstein. Keeping Science on Keel When Software Moves. *Commun. ACM*, 64(2):66–74, Jan. 2021. doi: 10.1145/3382037

[4] D. H. Ahn, N. Bass, A. Chu, J. Garlick, M. Grondona, S. Herbein, J. Koning, T. Patki, T. R. W. Scogland, B. Springmeyer, and M. Taufer. Flux: Overcoming scheduling challenges for exascale workflows. In *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, pp. 10–19, 2018. doi: 10.1109/WORKS.2018.00007

[5] G. Becker, P. Scheibel, M. P. LeGendre, and T. Gamblin. Managing combinatorial software installations with spack. In *2016 Third International Workshop on HPC User Support Tools, HUST@SC 2016, Salt Lake City, UT, USA, November 13, 2016*, pp. 14–23. IEEE Computer Society, 2016. doi: 10.1109/HUST.2016.007

[6] M. Bentley, I. Briggs, G. Gopalakrishnan, D. H. Ahn, I. Laguna, G. L. Lee, and H. E. Jones. Multi-Level Analysis of Compiler-Induced Variability and Performance Tradeoffs. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC 19, pp. 61–72. ACM, June 2019. doi: 10.1145/3307681.3325960

[7] E. Deelman, R. F. da Silva, K. Vahi, M. Rynge, R. Mayani, R. Tanaka, W. R. Whitcup, and M. Livny. The pegasus workflow management system: Translational computer science in practice. *J. Comput. Sci.*, 52:101200, 2021. doi: 10.1016/j.jocs.2020.101200

[8] M. a. d. H. N. Fasi, M. Mikaitis, and P. S. Numerical behavior of nvidia tensor cores, 2021.

[9] I. Laguna and G. Gopalakrishnan. Finding inputs that trigger floating-point exceptions in gpus via bayesian optimization. In *Supercomputing*, 2022. Accepted.

[10] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. *SIGPLAN Not.*, 50(6):1–11, June 2015. doi: 10.1145/2813885.2737959

[11] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: tuning assistant for floating-point precision. In W. Gropp and S. Matsuoka, eds., *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, pp. 27:1–27:12. ACM, 2013. doi: 10.1145/2503210.2503296