Report from Dagstuhl Seminar 17352

Analysis and Synthesis of Floating-point Programs

Edited by

Eva Darulova[1], Alastair F. Donaldson[2], Zvonimir Rakamarić[3], and Cindy Rubio-González[4]

1 **MPI-SWS – Saarbrücken, DE,** eva@mpi-sws.org
2 **Imperial College London, GB,** alastair.donaldson@imperial.ac.uk
3 **University of Utah – Salt Lake City, US,** zvonimir@cs.utah.edu
4 **University of California – Davis, US,** crubio@ucdavis.edu

Abstract

This report summarises the presentations, discussion sessions and panel that took place during the Dagstuhl seminar on "Analysis and Synthesis of Floating-point Programs" that took place during August 27 – 30, 2017. We hope that the report will provide a useful resource for researchers today who are interested in understanding the state-of-the-art and open problems related to analysing and synthesising floating-point programs, and as a historical resource helping to clarify the status of this field in 2017.

Seminar August 27–30, 2017 – http://www.dagstuhl.de/17352
1998 ACM Subject Classification Design and analysis of algorithms, Approximation algorithms analysis, Numeric approximation algorithms, Logic, Automated reasoning
Keywords and phrases energy-efficient computing, floating-point arithmetic, precision allocation, program optimization, rigorous compilation
Digital Object Identifier 10.4230/DagRep.7.8.74

1 Executive Summary

Eva Darulova
Alastair F. Donaldson
Zvonimir Rakamarić
Cindy Rubio-González

This report documents the program and the outcomes of Dagstuhl Seminar 17352 "Analysis and Synthesis of Floating-point Programs".

Floating-point numbers provide a finite approximation of real numbers that attempts to strike a fine balance between range, precision, and efficiency of performing computations. Nowadays, performing floating-point computations is supported on a wide range of computing platforms, and are employed in many widely-used and important software systems, such as high-performance computing simulations, banking, stock exchange, self-driving cars, and machine learning.

However, writing correct, and yet high-performance and energy-efficient, floating-point code is challenging. For example, floating-point operations are often non-associative (contrary to their real mathematical equivalents), which creates problems when an ordering of operations is modified by either a compiler or due to nondeterministic interleavings of concurrent executions. Furthermore, the underlying floating-point hardware is often heterogeneous,

hence different results may be computed across different platforms or even across components of the same heterogeneous platform. Given the underlying complexity associated with writing floating-point code, it is not surprising that there have been numerous software bugs attributed to incorrectly implemented floating-point computations.

Research related to floating-point computation spans a multitude of areas of computer science, ranging from hardware design and architecture, all the way to high-performance computing, machine learning, and software analysis and verification. The objective of this seminar was thus to bring together researchers from several of these areas, which have either traditionally been considered as non-overlapping, or which have arguably enjoyed insufficient interaction despite a clear overlap of interests. The goal in mind here was to provide opportunities to brainstorm new theoretical advances and practical techniques and tools for making floating-point computations performant and correct, and to help foster long term collaborations.

The seminar involved brief presentations from most participants, interspersed with a lot of informal technical discussion, in addition to four breakout sessions based on common themes that arose during informal discussion. In addition, a joint panel session was held between this seminar and the concurrently running "Machine Learning and Formal Methods" seminar. This report presents the collection of abstracts associated with the participant presentations, notes summarising each discussion session, and a transcript of the panel session. We hope that the report will provide a useful resource for researchers today who are interested in understanding the state-of-the-art and open problems related to analysing and synthesising floating-point programs, and as a historical resource helping to clarify the status of this field in 2017.

2 Table of Contents

3 Overview of Talks

3.1 Algorithm – Architecture Codesign

George A. Constantinides (Imperial College London, GB)

Joint work of George Constantinides, Peter Cheung, Xitong Gao, Wayne Luk
URL http://cas.ee.ic.ac.uk/people/gac1

This talk takes a look at the changing face of computer architecture and the implications for numerical compute. I argue that the drive in computer architecture in recent years has been to give more silicon (and more energy) back to computation, and I note that the rise of FPGA-based computation results in many possibilities for non-standard, customised arithmetic. I review work from my group in the early 2000s, which first considered the question of precision tuning for such customisation, for LTI systems implemented in fixed-point arithmetic. I then move to look at some more recent work from my group which automatically refactors code to explore the Pareto tradeoff between accuracy, area, and performance of algorithms specified in floating-point and implemented in FPGA hardware. Finally, I pose some automation challenges to the community – success in these challenges is likely to require the combined efforts of architects, computer arithmetic researchers, numerical analysts and programming language researchers.

3.2 Salsa: An Automatic Tool to Improve the Numerical Accuracy of Programs

Nasrine Damouche (University of Perpignan, FR)

Joint work of Nasrine Damouche, Matthieu Martel
Main reference Nasrine Damouche, Matthieu Martel, "Salsa: An Automatic Tool to Improve the Numerical Accuracy of Programs", Automated Formal Methods (AFM), 2017.
URL http://fm.csl.sri.com/AFM17/AFM17_paper_4.pdf

This talk describes Salsa, an automatic tool to improve the accuracy of the floating-point computations done in numerical codes. Based on static analysis methods by abstract interpretation, our tool takes as input an original program, applies to it a set of transformation rules and then generates a transformed program which is more accurate than the initial one. The original and the transformed programs are written in the same imperative language. This talk is a concise description of former work on the techniques implemented in Salsa, extended with a presentation of the main software architecture, the inputs and outputs of the tool as well as experimental results obtained by applying our tool on a set of sample programs coming from embedded systems and numerical analysis.

3.3 Algorithms for Efficient Reproducible Floating Point Summation and BLAS

James W. Demmel (University of California – Berkeley, US)

Joint work of James W. Demmel, Peter Ahrens, Hong Diep Nguyen, Greg Henry, Peter Tang, Xiaoye Li, Jason Riedy, Mark Gates
Main reference J. Demmel, I. Dumitriu, O. Holtz, P. Koev, "Accurate and Efficient Expression Evaluation and Linear Algebra," Acta Numerica, v. 17, 2008.
URL https://tinyurl.com/y8yxq94n

We briefly present 3 topics: (1) Reproducibility. We define reproducibility to mean getting bitwise identical results from multiple runs of the same program, perhaps with different hardware resources or other changes that should ideally not change the answer. Many users depend on reproducibility for debugging or correctness. However, dynamic scheduling of parallel computing resources, combined with non-associativity of floating point addition, makes attaining reproducibility a challenge even for simple operations like summing a vector of numbers, or more complicated operations like the Basic Linear Algebra Subprograms (BLAS). We describe two versions of an algorithm to compute a reproducible sum of floating point numbers, independent of the order of summation. The first version depends only on a subset of the IEEE Floating Point Standard 754-2008, and the second version uses a possible new instruction in the proposed Standard 754-2018. The algorithm is communication-optimal, in the sense that it does just one pass over the data in the sequential case, or one reduction operation in the parallel case, requiring a "reproducible accumulator" represented by just 6 floating point words (more can be used if higher precision is desired), enabling standard tiling techniques used to optimize the BLAS. The arithmetic cost with a 6-word reproducible accumulator is 7n floating point additions to sum n words using version 1, or 3n using version 2, and (in IEEE double precision) the final error bound can be up to 10^8 times smaller than the error bound for conventional summation on ill-conditioned inputs. We also describe ongoing efforts to incorporate this in a future BLAS Standard. We seek feedback on this proposed new instruction, which is also intended to accelerate many existing algorithms for double-double, compensated summation, etc. For details see [1]. (2) New BLAS Standard. The BLAS standards committee is meeting again to consider a new version, motivated by industrial and academic demand for new precisions, batched BLAS, and reproducible BLAS. This is also an opportunity to improve floating point exception handling in the BLAS, which currently propagates exceptions, e.g. NaNs, inconsistently. We seek feedback on the draft standard, available at [2]. (3) Automatic generation of accurate floating point formulas. This was a topic at Dagstuhl 05391, 25-30 Sept, 2005, when Olga Holtz presented joint work on the decidability of whether an expression guaranteeing high relative accuracy exists for a given algebraic expression (in the 1 + delta model, so real numbers). We give an example of this (the Motzkin polynomial) and point out more recent (prize-winning) results in [3].

Topic (1) is joint work with Peter Ahrens (former UCB undergrad, now a grad student at MIT) and Hong Diep Nguyen (former UCB postdoc, now at a startup) [1].

Topic (2) is joint work with Greg Henry (Intel), Peter Tang (Intel), Xiaoye Li (LBNL), Jason Riedy (GaTech) and Mark Gates (U Tenn) [2].

Topic (3) is joint work with Olga Holtz (UCB), Ioana Dumitriu (U Wash), and Plamen Koev (former UCB grad student, now at SJSU) [3].

References

1 J. Demmel, P. Ahrens, H.-D. Nguyen. *Efficient Reproducible Floating Point Summation and BLAS*. UC Berkeley EECS Tech Report UCB/EECS-2016-121, June 2016.
2 J. Demmel, M. Gates, G. Henry, X. S. Li, J. Riedy, P. T. P. Tang. *A proposal for a Next-Generation BLAS*. Writable document (at time of writing) for submission of comments: `https://tinyurl.com/y8yxq94n`, August 2017.
3 J. Demmel, I. Dumitriu, O. Holtz, P. Koev. *Accurate and Efficient Expression Evaluation and Linear Algebra*. Acta Numerica, v. 17, 2008.

3.4 A Comprehensive Study of Real-World Numerical Bug Characteristics

Anthony Di Franco (University of California – Davis, US)

Joint work of Anthony Di Franco, Hui Guo, Cindy Rubio-González
Main reference Anthony Di Franco, Hui Guo, Cindy Rubio-González: "A comprehensive study of real-world numerical bug characteristics", in Proc. of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, pp. 509–519, IEEE Computer Society, 2017.
URL http://dx.doi.org/10.1109/ASE.2017.8115662

Numerical software is used in a wide variety of applications including safety-critical systems, which have stringent correctness requirements, and whose failures have catastrophic consequences that endanger human life. Numerical bugs are known to be particularly difficult to diagnose and fix, largely due to the use of approximate representations of numbers such as floating point. Understanding the characteristics of numerical bugs is the first step to combat them more effectively. In this paper, we present the first comprehensive study of real-world numerical bugs. Specifically, we identify and carefully examine 269 numerical bugs from five widely-used numerical software libraries: NumPy, SciPy, LAPACK, GNU Scientific Library, and Elemental. We propose a categorization of numerical bugs, and discuss their frequency, symptoms and fixes. Our study opens new directions in the areas of program analysis, testing, and automated program repair of numerical software, and provides a collection of real-world numerical bugs.

3.5 Testing Compilers for a Language With Vague Floating-Point Semantics

Alastair F. Donaldson (Imperial College London, GB)

Joint work of Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, Paul Thomson
Main reference Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, Paul Thomson: "Automated testing of graphics shader compilers", PACMPL, Vol. 1(OOPSLA), pp. 93:1–93:29, 2017.
URL http://dx.doi.org/10.1145/3133917

The OpenGL shading language (GLSL) deliberately has vague semantics when it comes to floating-point arithmetic, stating (see p. 90 of the OpenGL Shading Language 4.50 specification):

> *"Without any [precision] qualifiers, implementations are permitted to perform such optimizations that effectively modify the order or number of operations used to evaluate an expression, even if those optimizations may produce slightly different results relative to unoptimized code."*

As a result, it is hard to test compilers for GLSL: there is no predefined image that a shader should render for a given input, and differential testing across shader compilers for different platforms is not straightforward because one can legitimately expect differences in rendering results.

We have been investigating an alternative testing approach using *metamorphic testing*, inspired by recent work on compiler validation using *equivalence modulo inputs* testing. We take an initial shader and make from it a family of *variant* shaders by applying semantics-preserving transformations, or more precisely transformations that would have no impact on semantics if floating-point operations were replaced with real number operations. We then render, on a single platform, all the images generated by compiling and executing each shader in the family. While we do expect to see pixel-level differences between rendered images, since our transformations may have impacted on floating-point computation, if the initial shader is numerically stable these differences should not be large. Using an image comparison metric, such as the chi-squared distance between image histograms, we can automatically flag up variant shaders that give rise to large rendering differences. To such *deviant* shaders we then apply a form of delta-debugging whereby we iteratively reverse the semantics-preserving transformations that were initially applied, converging on a shader that differs minimally from the original shader such that the source-level difference – which should be semantics-preserving – causes a large difference in rendering. Inspection of this difference then sheds light on whether there is a compiler bug, or whether the original shader is in fact numerically unstable.

We have implemented this method in a tool, GLFuzz, which we have successfully applied to a number of commercial shader compilers, finding a large number of defects that we track via a repository (see https://github.com/mc-imperial/shader-compiler-bugs/), with highlights from the work summarized via a series of online posts (see https://medium.com/@afd_icl/689d15ce922b).

3.6 Floating-Point Cadence

Theo Drane (Cadence Design Systems – Bracknell, GB)

Main reference Theo Drane, "Lossy Polynomial Datapath Synthesis", PhD thesis, Imperial College London, 2014.
URL https://spiral.imperial.ac.uk/handle/10044/1/15566

Cadence's logic synthesis and high level synthesis divisions, Genus and Stratus respectively, both come with floating-point IP offerings. Our IP development is done in tandem with our synthesis tools and thus provide a unique opportunity for co-optimisation. Our verification draws upon other Cadence divisions in equivalence and model checking, Conformal and Jasper divisions respectively; as well as reaching out into the use of academic theorem provers. This talk will cover the offerings, challenges and potential future avenues of research and development in this area.

3.7 Floating-point result-variability: sources and handling

Ganesh L. Gopalakrishnan (University of Utah – Salt Lake City, US)

Joint work of Geof Sawaya, Michael Bentley, Ian Briggs, Ganesh Gopalakrishnan, Dong Ahn
Main reference Geoffrey Sawaya, Michael Bentley, Ian Briggs, Ganesh Gopalakrishnan, Dong H. Ahn: "FLiT: Cross-platform floating-point result-consistency tester and workload", in Proc. of the 2017 IEEE International Symposium on Workload Characterization, IISWC 2017, Seattle, WA, USA, October 1-3, 2017, pp. 229–238, IEEE Computer Society, 2017.
URL http://dx.doi.org/10.1109/IISWC.2017.8167780

Understanding the extent to which computational results can change across platforms, compilers, and compiler flags can go a long way toward supporting reproducible experiments. In this work, we offer the first automated testing aid called FLiT (Floating-point Litmus Tester) that can show how much these results can vary for any user-given collection of computational kernels. Our approach is to take a collection of these kernels, disperse them across a collection of compute nodes (each with a different architecture), have them compiled and run, and bring the results to a central SQL database for deeper analysis.

Properly conducting these activities requires a careful selection (or design) of these kernels, input generation methods for them, and the ability to interpret the results in meaningful ways. The results in this paper are meant to inform two different communities: (a) those interested in seeking higher performance by considering "IEEE unsafe" optimizations, but then want to understand how much result variability to expect, and (b) those interested in standardizing compiler flags and their meanings, so that one may safely port code across generations of compilers and architectures.

By releasing FLiT, we have also opened up the possibility of all HPC developers using it as a common resource as well as contributing back interesting test kernels as well as best practices, thus extending the floating-point result-consistency workload we contribute. This is the first such workload and result-consistency tester underlying floating-point reproducibility of which we are aware.

3.8 Hierarchical Search in Floating-Point Precision Tuning

Hui Guo (University of California – Davis, US)

Joint work of Hui Guo, Cindy Rubio-González

Floating-point types are notorious for their intricate representation. The effective use of mixed precision, i.e., using various precisions in different computations, is critical to achieve a good balance between accuracy and performance. Unfortunately, reasoning about the impact of different precision selections is a difficult task even for numerical experts. Techniques have been proposed to systematically search over floating-point variables and/or program instructions to find a profitable precision configuration. These techniques are characterized by their "black-box" nature, and face scalability limitations mainly due to the large search space.

In this talk, we present a semantic-based search algorithm that hierarchically guides precision tuning to improve performance given an accuracy constraint. The novelty of our algorithm lies in leveraging semantic information to reduce the search space and find more profitable precision configurations. Specifically, we perform dependence analysis and edge

profiling to create a weighted dependence graph that presents a network of floating-point variables. We then formulate hierarchy construction on the network as a community detection problem, and present a hierarchical search algorithm that iteratively lowers precision with regard to communities. Our evaluation on real world floating-point programs shows that hierarchical search algorithm is in general more efficient than the state of the art in finding profitable configurations.

3.9 Auto-tuning Floating-Point Precision

Jeffrey K. Hollingsworth (University of Maryland – College Park, US)

Joint work of Mike Lam, Jeffrey K. Hollingsworth
Main reference Mike O. Lam, Jeffrey K. Hollingsworth, "Fine-grained floating-point precision analysis," International Journal of High Performance Computing Applications, SAGE Publications, 2016.
URL http://doi.org/10.1177/1094342016652462

In this talk I will describe the CRAFT tool for automatically exploring the required floating point precision for specific programs. The tool operates on compiled binary programs. Using different analysis modules, it can explore reduced precision (single vs. double) as well as variable precisions (specific numbers of bits of precision for each operation). I will also present results of using the CRAFT tool for different benchmark programs.

3.10 Floating Point Computations in the Multicore and Manycore Era

Miriam Leeser (Northeastern University – Boston, US)

Joint work of Miriam Leeser, Thomas Wahl, Mahsa Bayati, Yijia Gu
Main reference Yijia Gu, Thomas Wahl, Mahsa Bayati, Miriam Leeser: "Behavioral Non-portability in Scientific Numeric Computing", in Proc. of the Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings, Lecture Notes in Computer Science, Vol. 9233, pp. 558–569, Springer, 2015.
URL https://doi.org/10.1007/978-3-662-48096-0_43

The results of numerical computations with floating-point numbers depend on the execution platform. One reason is that, even for similar floating point hardware, compilers have significant freedom in deciding how to evaluate a floating-point expression, as such evaluation is not standardized. Differences can become particularly large across (heterogeneous) parallel architectures. This may be surprising to a programmer who conflates the portability promised by programming standards such as OpenCL with reproducibility.

In this talk, I present experiments, conducted on a variety of platforms including CPUs and GPUs, that showcase the differences that can occur even for randomly selected inputs. I present a study of the same OpenCL code run on different architectures and analyze the sources of differences, and what tools are needed to aid the programmer. The code is taken from popular high performance computing benchmark suites. I also introduced challenges to the research community. A major one is being able to handle programs that run on large parallel machines while providing useful information to the user regarding the importance of these differences.

3.11 JFS: Solving floating point constraints with coverage guided fuzzing

Daniel Liew (Imperial College London, GB)

Joint work of Daniel Liew, Cristian Cadar, Alastair Donaldson
Main reference Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F. Donaldson, Rafael Zähl, Klaus Wehrle: "Floating-point symbolic execution: a case study in n-version programming", in Proc. of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, pp. 601–612, IEEE Computer Society, 2017.
URL http://dx.doi.org/10.1109/ASE.2017.8115670

In this talk I will present the work-in-progress design and implementation of a constraint solver called JIT Fuzzing Solver (JFS). JFS is designed to investigate the use of "coverage guided fuzzing" as an incomplete tactic for solving SMT queries that use any combination of the Core, BitVector, and FloatingPoint theories defined by SMT-LIB.

JFS takes as input an SMT query, from which it generates a C++ program with a sequence of "if" branches, each corresponding to an assert from the query. The program is constructed such that finding an input to the program that traverses all the "true" edges of the branches is equivalent to finding a satisfying assignment to all the free variables of the SMT query. To find such an input, a high-performance coverage-guided fuzzer is used. We conclude with a brief discussion of initial results.

This work was motivated by a project on extending symbolic execution with support for floating-point reasoning; the "main reference" is a paper on that project.

3.12 Autotuning for Portable Performance for Specialized Computational Kernels

Piotr Luszczek (University of Tennessee – Knoxville, US)

Joint work of Piotr Luszczek, Jakub Kurzak, Mark Gates, Mike (Yaohung) Tsai, Matthew Bachstein, Jack Dongarra
Main reference Mark Gates, Jakub Kurzak, Piotr Luszczek, Yu Pei, Jack J. Dongarra: "Autotuning Batch Cholesky Factorization in CUDA with Interleaved Layout of Matrices", in Proc. of the 2017 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2017, Orlando / Buena Vista, FL, USA, May 29 - June 2, 2017, pp. 1408–1417, IEEE Computer Society, 2017.
URL http://dx.doi.org/10.1109/IPDPSW.2017.18

High performance Exascale libraries for numerical algorithms, data analytics, and statistical inference require specialized implementations of computational kernels that progressively become harder to create due to the increasingly divergent designs of extreme-scale hardware platforms. We present our ongoing efforts in automated performance engineering that encompasses code autotuning within a comprehensive framework of integrated tools that include domain specific languages, testing/deployment infrastructure, and analysis modules. These components assist in continuous development, deployment, and improvement of these essential scientific kernels.

3.13 Interval Enclosures of Upper Bounds of Roundoff Errors using Semidefinite Programming

Victor Magron (VERIMAG – Grenoble, FR)

Joint work of Victor Magron, George Constantinides, Alastair F. Donaldson
Main reference Victor Magron, George A. Constantinides, Alastair F. Donaldson: "Certified Roundoff Error Bounds Using Semidefinite Programming", ACM Trans. Math. Softw., Vol. 43(4), pp. 34:1–34:31, 2017.
URL http://dx.doi.org/10.1145/3015465

Roundoff errors cannot be avoided when implementing numerical programs with finite precision. The ability to reason about rounding is especially important if one wants to explore a range of potential representations, for instance in the world of FPGAs. This problem becomes challenging when the program does not employ solely linear operations as non-linearities are inherent to many computational problems in real-world applications.

We present two frameworks which can be combined to provide interval enclosures of upper bounds for absolute roundoff errors.

The framework for upper bounds is based on optimization techniques employing semidefinite programming (SDP) and sparse sums of squares certificates, which can be formally checked inside the Coq theorem prover.

The framework for lower bounds is based on a new hierarchy of convergent robust SDP approximations for certain classes of polynomial optimization problems. Each problem in this hierarchy can be exactly solved via SDP.

Our tool covers a wide range of nonlinear programs, including polynomials and transcendental operations as well as conditional statements. We illustrate the efficiency and precision of this tool on non-trivial programs coming from biology, optimization and space control.

3.14 Verification for floating-point, floating-point for verification

David Monniaux (Université Grenoble Alpes – Saint-Martin-d'Hères, FR)

Joint work of David Monniaux, Antoine Miné, Jérôme Feret
Main reference Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, Xavier Rival: "A static analyzer for large safety-critical software", in Proc. of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003, pp. 196–207, ACM, 2003.
URL http://dx.doi.org/10.1145/781131.781153

Verification for floating-point: Verifying the safety of floating-point programs poses specific challenges. The usual assumptions about arithmetic types (addition and multiplication form a commutative ring, etc.) are now false, making it difficult to apply standard abstractions (e.g. zones, octagons, polyhedra). In the Astrée system, we applied interval analysis as well as relaxation of floating-point into reals so as to use the standard abstractions. We also developed specific abstract domains for filters.

Floating-point for verification: It is tempting to use floating-point inside decision procedures and other algorithms inside analysis tools, instead of exact precision arithmetic, which can be much slower. But how can we be sure of the results? We present here methods for e.g. using an off-the-shelf implementation of linear programming inside a sound tool.

3.15 Alive-FP: Automated Verification of Floating Point Optimizations in LLVM

Santosh Nagarakatte (Rutgers University – Piscataway, US)

Joint work of David Menendez, Santosh Nagarakatte, Aarti Gupta
Main reference David Menendez, Santosh Nagarakatte, Aarti Gupta: "Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM", in Proc. of the Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings, Lecture Notes in Computer Science, Vol. 9837, pp. 317–337, Springer, 2016.
URL https://doi.org/10.1007/978-3-662-53413-7_16

Peephole optimizations optimize and canonicalize code to enable other optimizations but are error-prone. This talk presents Alive-FP, an automated verification framework for floating point based peephole optimizations in LLVM. Alive-FP handles a class of floating point optimizations and fast-math optimizations involving signed zeros, not-a-number, and infinities, which do not result in loss of accuracy. This talk will describe multiple encodings for various floating point operations to account for the various kinds of undefined behavior and under-specification in the LLVM's language reference manual. We have discovered seven wrong optimizations in LLVM using Alive-FP.

3.16 Debugging Large-Scale Numerical Programs with Herbgrind

Pavel Panchekha (University of Washington – Seattle, US)

Joint work of Pavel Panchekha, Alex Sanchez-Stern, Zachary Tatlock, Sorin Lerner
URL http://herbgrind.ucsd.edu

It is hard to find and fix numerical errors in large numerical programs because it is difficult to detect the error, determine which instructions are responsible for the error, and gather context describing the computation that lead to the error. Herbgrind is a dynamic binary analysis tool to address these issues. Herbgrind uses higher-precision shadow values to compute a ground truth, correct value for every floating-point value in the program, thus making it possible to detect that program outputs are erroneous. Herbgrind then uses the local error heuristic to identify program instructions that introduce numerical errors and then propagates information about those instructions to affected program values using a taint analysis. Finally, to gather context describing the computations that lead to instructions with high local error, Herbgrind uses anti-unification to summarized expression trees into an abstract form that describes the computation. Herbgrind has been used on large numerical programs, and has proven valuable for identifying errors and providing the information necessary to fix them.

3.17 Utah Floating-Point Toolset

Zvonimir Rakamarić (University of Utah – Salt Lake City, US)

Joint work of Zvonimir Rakamarić, Ganesh Gopalakrishnan, Alexey Solovyev, Wei-Fan Chiang, Ian Briggs, Mark Baranowski, Dietrich Geisler, Charles Jacobsen
Main reference Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamarić, Ganesh Gopalakrishnan: "Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions", in Proc. of the FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings, Lecture Notes in Computer Science, Vol. 9109, pp. 532–550, Springer, 2015.
URL https://doi.org/10.1007/978-3-319-19249-9_33

Virtually all real-valued computations are carried out using floating-point data types and operations. With the current emphasis of system development often being on computational efficiency, developers as well as compilers are increasingly attempting to optimize floating-point routines. Reasoning about the correctness of these optimizations is complicated, and requires error analysis procedures with different characteristics and trade-offs. In my talk, I present both dynamic and rigorous static analyses we developed for estimating errors of floating-point routines. Finally, I describe how we extended our rigorous static analysis into a procedure for mixed-precision tuning of floating-point routines.

3.18 Condition Number and Interval Computations

Nathalie Revol (ENS – Lyon, FR)

Main reference Nathalie Revol, "Influence of the Condition Number on Interval Computations: Illustration on Some Examples," in honour of Vladik Kreinovich' 65th birthday, Springer Festschrift, El Paso, United States, 2017.
URL https://hal.inria.fr/hal-01588713

The condition number is a quantity that is well-known in "classical" numerical analysis, that is, where numerical computations are performed using floating-point numbers. This quantity appears much less frequently in interval numerical analysis, that is, where the computations are performed on intervals.

In this talk, three small examples are used to illustrate experimentally the impact of the condition number on interval computations. As expected, problems with a larger condition number are more difficult to solve: this means either that the solution is not very accurate (for moderate condition numbers) or that the method fails to solve the problem, even inaccurately (for larger condition numbers). Different strategies to counteract the impact of the condition number are discussed and, with success, experimented: use of a higher precision, iterative refinement, bisection of the input.

3.19 Dynamic Analysis for Floating-Point Precision Tuning

Cindy Rubio-González (University of California – Davis, US)

Joint work of Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Iancu, Costin; Hough, David
Main reference Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, David Hough: "Precimonious: tuning assistant for floating-point precision", in Proc. of the 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.
URL http://dx.doi.org/10.1145/2503210.2503296

Given the variety of numerical errors that can occur, floating-point programs are difficult to write, test and debug. One common practice employed by developers without an advanced background in numerical analysis is using the highest available precision. While more robust, this can degrade program performance significantly. In this paper we present Precimonious, a dynamic program analysis tool to assist developers in tuning the precision of floating-point programs. Precimonious performs a search on the types of the floating-point program variables trying to lower their precision subject to accuracy constraints and performance goals. Our tool recommends a type instantiation that uses lower precision while producing an accurate enough answer without causing exceptions. We evaluate Precimonious on several widely used functions from the GNU Scientific Library, two NAS Parallel Benchmarks, and three other numerical programs. For most of the programs analyzed, Precimonious reduces precision, which results in performance improvements as high as 41%.

3.20 FPBench: Toward Standard Floating Point Benchmarks

Zachary Tatlock (University of Washington – Seattle, US)

Joint work of Zachary Tatlock, Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern
Main reference Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, Zachary Tatlock: "Toward a Standard Benchmark Format and Suite for Floating-Point Analysis", in Proc. of the Numerical Software Verification - 9th International Workshop, NSV 2016, Toronto, ON, Canada, July 17-18, 2016, [collocated with CAV 2016], Revised Selected Papers, Lecture Notes in Computer Science, Vol. 10152, pp. 63–77, 2016.
URL http://dx.doi.org/10.1007/978-3-319-54292-8_6

FPBench is a standard format and common set of benchmarks for floating-point accuracy tests. The goal of FPBench is to enable direct comparisons between competing tools, facilitate the composition of complementary tools, and lower the barrier to entry for new teams working on numerical tools. FPBench collects benchmarks from published papers in a standard format and with standard accuracy measures and metadata. As a single repository for benchmarks, FPBench can be used to guide the development of new tools, evaluate completed tools, or compare existing tools on identical inputs, all while avoiding duplication and the manual effort and inevitable errors of translating between input formats. Please see http://fpbench.org for more.

3.21 Impacts of non-determinism on numerical reproducibility and debugging at the exascale

Michela Taufer (University of Delaware – Newark, US)

Joint work of Dylan Chapp, Travis Johnston, Michela Taufer
Main reference Dylan Chapp, Travis Johnston, Michela Taufer: "On the Need for Reproducible Numerical Accuracy through Intelligent Runtime Selection of Reduction Algorithms at the Extreme Scale", in Proc. of the 2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015, pp. 166–175, IEEE Computer Society, 2015.
URL http://dx.doi.org/10.1109/CLUSTER.2015.34

In message-passing applications, one notable technique is the use of non-blocking point-to-point communication, which permits communication and computation to be overlapped, leading to an increase in scalability. The price paid however, is the loss of determinism in applications' executions. Non-determinism in high performance scientific applications has severe detrimental impacts for both numerical reproducibility and debugging. As scientific simulations are migrated to extreme-scale platforms, the increase in platform concurrency and the attendant increase in non-determinism is likely to exacerbate both of these problems (i.e., numerical reproducibility and debugging). In this talk, we address the challenges of non-determinism's impact on numerical reproducibility and on debugging. Specifically, we present empirical studies focusing on floating-point error accumulation over global reductions where enforcing any reduction order is expensive or impossible. We also discuss techniques for record and replay to reduce out-of-order message rate in non-deterministic execution.

3.22 An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point

Laura Titolo (National Institute of Aerospace – Hampton, US)

Joint work of Marco A. Feliu, Aaron Dutle, Laura Titolo, Cesar A. Muñoz
Main reference Mariano M. Moscato, Laura Titolo, Aaron Dutle, César A. Muñoz: "Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis", in Proc. of the Computer Safety, Reliability, and Security - 36th International Conference, SAFECOMP 2017, Trento, Italy, September 13-15, 2017, Proceedings, Lecture Notes in Computer Science, Vol. 10488, pp. 213–229, Springer, 2017.
URL http://doi.org/10.1007/978-3-319-66266-4_14

In this work, an abstract interpretation framework for the round-off error analysis of floating-point programs is presented. This framework defines a parametric abstract semantics of the floating-point round-off error of functional programs. Given a functional floating-point program, the abstract semantics computes, for each execution path of the program, an error expression representing a sound over-approximations of the accumulated floating-point round-off error that may occur in that execution path. In addition, a boolean expression representing the input values that satisfy the path conditions of the given path is also computed. This boolean expression characterizes the input values leading to the computed error approximation. A widening operator is defined to ensure the convergence of recursive programs. The proposed framework provides the infrastructure to correctly handle the so-called unstable tests, which occur when the round-off errors of conditional floating-point expressions alter the execution path of the ideal program on infinite precision real arithmetic.

3.23 Stabilizing Numeric Programs against Platform Uncertainties

Thomas Wahl (Northeastern University – Boston, US)

Joint work of Mahsa Bayati, Yijia Gu, Miriam Leeser, Thomas Wahl
Main reference Yijia Gu, Thomas Wahl: "Stabilizing Floating-Point Programs Using Provenance Analysis", in Proc. of the Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings, Lecture Notes in Computer Science, Vol. 10145, pp. 228–245, Springer, 2017.
URL https://doi.org/10.1007/978-3-319-52234-0_13

Floating-point arithmetic (FPA) is a loosely standardized approximation of real arithmetic available on many computers today. The use of approximation incurs commonly underestimated risks for the reliability of numeric software, including reproducibility issues caused by the relatively large degree of freedom for FPA implementers offered by the IEEE 754 standard. If left untreated, such problems can seriously interfere with program portability.

In this talk I demonstrate how, using information on the provenance of platform dependencies, reproducibility violations can be repaired with low impact on program efficiency, resulting in stabilized program execution. I illustrate the use of these techniques on decision-making and purely numeric programs.

This is direct joint work with my student Yijia Gu, and is based on longer collaboration with Miriam Leeser and her students, particularly Mahsa Bayati.

4 Working groups

4.1 Notes from Breakout Session: "Analysis Tools for Floating-Point Software"

Pavel Panchekha (University of Washington – Seattle, US)

The discussion on tools focused on two central issues: the benefits and challenges of running a tools competition (heavily reliant on Zvonimir's experience with SVComp) and then on incremental steps that can be taken toward those benefits. The key conclusions were that a tools competition provides visibility for participants, including a full list of entrants; a deadline by which entrants must support a common format, which force participants to implement that functionality; ease of entry by new groups; and the opportunity to write up an analysis on the entrants, their strengths and weaknesses, and the benchmarks covered. This final benefit seemed particularly beneficial to industry users, as Sam attested. To achieve some of these benefits incrementally, a series of four steps were chosen: first, a central list of floating point research tools; second, support for common input and output formats; third, regular composition of floating-point tools; and fourth, regular competitions to compare tools in the same space.

The discussion started with Zvonimir's experience with the SVComp competition, where software verification tools compete to find assertion violations on a set of tens of thousands of benchmarks formatted as C code. The competition works by having a deadline by which entrants must submit an analyzer. Each analyzer is then run on each benchmark without any interactions, and the results are tallied, scored, and the results made available two weeks

later or so. The whole evaluation is fully-scripted, though other competitions are structured differently (with interaction and a limited benchmark set). The results of the competition are then discussed at a dedicated sessions at TACAS, which also provides 50 pages of conference proceedings to publish the competition analysis (written by the competition organizer) as well as write-ups by winners. A key challenge in organizing the competition is in resolving disagreements between tools and within the community. For example, a dedicated witness and proof format was eventually invented to make it possible to resolve disagreements between tools (since not every benchmark program is labeled with correct answers), and the weighting of results has been changed over time as the emphases of the community have shifted (mistakes, for example, are now more heavily penalized than earlier). Within the community, managing different perspectives, like the difference between bug-finding (where false positives are penalized but false negatives may be acceptable) and verification (where the emphasis is reversed) has been a challenge throughout the life of the competition.

This experience informed a discussion of the dangers of a tools competition. The chief among them is the possibility of over-fitting to the benchmarks. As the community's goals change, some benchmarks may become irrelevant, and their presence in the competition distorts the competitions goals. Benchmarks are also drawn from many sources, and this leads to disagreements in the community; for example, the SVComp benchmarks range from five to fifty-thousand lines of C – disagreements over the relative importance of programs of different sizes can be a problem. A committee is also needed to handle disputes, plus organizational effort put in yearly and money to set up the competition and reward winners. In SVComp's case, this is done by a single professor, who secured a European grant for the work; over the long term, this may be necessary. Especially valuable in SVComp's case has been the support of a conference. Publishing proceedings and tool reports is an important way to reward competition entrants, and so extremely important in organizing one. (A conference such as TACAS allowing proceedings space is especially good, but in a pinch it seemed a workshop proceedings could also work. A workshop proceedings could also allow the competition to move from conference to conference, exposing the competition to new audiences.)

Some miscellaneous thoughts on the competition asked whether the competition should also involve submitting new benchmarks (whether that would be fair, and also whether or not it would lead to good benchmark submissions); and how to set the objective of the competition: highest accuracy, tightest error bounds, assertion verification, or something else? There would also need to be consideration for tools that verify specifications and tools that help users explore the design space. Since a persistent issue is accuracy improvement by increasing the domain of accurate input values, evaluation by that metric would also be valuable. With as diverse a community as floating-point is, a large suite of benchmarks and several independent competitions seemed necessary.

With the idea of a tools competition having been deeply explored, the discussion turned to composing tools. Though competitions would force tool developers to adopt common formats, composing tools would be the most valuable benefit of that change. The most valuable compositions seem to be refactoring (as in Salsa or Herbie) together with precision tuning (as in Precimonious or FPTuner); tuning and refactoring with verification (as in Rosa or FPTaylor); and input generation together with tools (like Precimonious or Herbgrind) that rely on user-supplied inputs. These compositions would yield more complicated tools with more complicated trade-offs. It would become important to show the Pareto frontier (between, for example, accuracy and speed), and to evaluate pairs of tools together.

Finally, to make the discussion even more concrete, Pavel spoke for three strategies the FPBench project had been considering. The first was a list of floating-point research tools – a website could be maintained, with information about each tool and links to project web pages, source code, and papers. Second would be an auto-evaluation capability, to download supported tools and run them on the benchmarks automatically. Third would be a series of workshop papers on the benefits and uses combining two tools together. The universal consensus was that a list of tools would be most immediately valuable; Theo described this approach as "Visibility First". The other two suggestions could follow on from that work. The list of tools could also indicate support for FPBench, which could be a form of leverage to encourage tools to support the common format. Overall, the order would be to focus on listing tools, then encouraging a common input format, then composing tools, and only then comparing them.

Additional suggestions for FPBench included an exact real evaluation mode (using, for example, the computable reals) and support for simple assertion tests (such as sign or convergence tests). For full accuracy specification, a much richer assertion language would be needed, which FPBench did not develop due to a lack of examples, but simple tests could be easily described and then verified.

4.2 Notes from Breakout Session: "Specifications for Programs Computing Over Real Number Approximations"

Cindy Rubio-González (University of California – Davis, US)

We are interested in specifications for programs computing over reals.

What is the notion of accuracy? According to the machine learning panel, and the discussions throughout the seminar, there is a mismatch between worst-case error case and programmer's expectations. Programmers want programs to be approximately correct most of the time.

What does it mean to be approximately correct? What do programmers mean by 'most of the time'? We really need to take the application into account.

What do we mean by specifications? We also need to be more specific about the kinds of specifications we are interested in. What level of specifications do we refer to? Specifications at the user level, or at the component level? Furthermore, tools work at different levels of abstraction? Are we interested in hardware specifications, library specifications, application specifications? Do we need DSLs for expressing specifications?

What do 'most of the time' mean? For many, most of the time may mean 'for most inputs'. If we consider classification in machine learning, positive/negative results can be massively wrong. On the other hand, a large different in a number may be OK for applications where accuracy is not as important. 'Most of the time' is definitely relative, and heavily dependent on the application under study.

'Most of the time' may also require for us to be aware of special cases. If we assume all inputs are independent, then we can sampling over reals/floats. However, users need to be educated in this respect. For dynamic tools, test inputs are already samples, and a main goal is for user inputs not to break the tool.

What is a good specification for verification? We really need an expert to provide the specifications. For example, the control theory part of embedded systems would provide specifications, but do they? There is just no general way to gather specifications because these depend on the application. Specifications are domain specific.

But, what does it mean to be most of the time correct? For machine learning the specification is "approximately correct". But "most of the time" is not part of the equation. We are not able to prove "all of the time", and we don't know what "most of the time" means. Perhaps what we want to find is under what conditions an algorithm is stable. That is, under what inputs is an algorithm stable? Thus, we should perhaps focus on the measure of robustness and the discovery of preconditions.

How about probabilistic assertions? Can we use these to describe the input space? We want the probability distribution. When is the environment hostile?

How about benchmarks of probabilistic assertions? Do they include floating point? Are they large enough? Are there any other floating-point issues these do not consider? For example, floating-point roundoff errors are not considered in the context of probabilistic programming.

Are there simpler specifications we can find? Perhaps we can synthesize simpler optimizations such as loop optimization with respect to precision required, or assertions to insert in the code.

In general, there is a need for better examples. There are many different application domains facing different problems. An example is radiotherapy. The main block to verification is the lack of formal semantics.

Scientists just "know" what is a correct/expected answer.

What can we do in computation that provides self certification? An example is calculating the residual for a certain computation. But, to what degree can we use self certification?

What to do when specifications are not available? How about testing without oracles? A very successful approach is to compare alternative ways to compute an answer, and then compare the results. Although no formal guarantees are provided, it is very useful in practice to detect unexpected behavior.

4.3 Notes from Breakout Session: "Compilers: IEEE Compliance and Fast Math Requirements"

Daniel Schemmel (RWTH Aachen, DE)

Prompted not only by problems in traditional languages, such as C, C++ or Fortran, but also by upcoming languages intended for other platforms, such as OpenCL or OpenACC, the questions of IEEE-754 compliance and specification of fast math requirements was discussed in this breakout session. Current state of the art is very compiler-dependent behavior, whose guarantees (where they exist at all) are not trusted very far.

Ideally, a user would be able to always rely on strong guarantees such as: specified error bounds will be met, the control flow of the program remains unchanged and/or FP optimizations will never introduce undefined behavior to an otherwise correct program. Any of these strong guarantees would however require strong whole-program analyses, which are

usually infeasible. Additionally, many library operations are not just out of scope for the compiler, but may only exist as compiled code or even be written directly in assembly.

The consensus in this breakout session was that floating point optimizations should be compiled as a well-specified set of contracts, in which the developer explicitly agrees to certain restrictions in order to increase performance. Examples for such contracts could be a guarantee to never use certain kinds of values such as NaNs or Infinities (compare e.g. GCC's `-ffinite-math-only`) or the explicit permission to ignore association, which may improve vectorization at the cost of precision. In general, IEEE-754 non-compliant behavior should never be the default.

While modern C compilers are already going in a similar direction, there are big weaknesses w.r.t. IEEE rounding modes. For example, GCC does not even claim to be IEEE-754 compliant in its default mode, as it always assumes round to nearest, tie to even. The reasoning behind this is that it often inhibits even basic operations like constant folding, if the rounding mode is not known statically. During this session, the participants agreed that it would be useful to be able to statically define rounding modes per instruction or per block. This would enable static analysis while still being useful in most cases. Two additional rounding modes were suggested: First, a "native" rounding mode would be useful in the case where the developer does not really care, and maybe only a single (possibly non-standard) rounding mode is available on the target platform. This "native" rounding mode would allow maximum optimization. Second, a "dynamic" rounding mode could enable a dynamic setting, as is currently mandated by changing the floating point environment programmatically (cf. `#pragma STDC FENV ACCESS ON`). By statically expressing a desire for a specific rounding mode, it becomes more natural for compilers to either comply with the request, or to deny compilation, which would elegantly deal with the current problem of compilers that silently ignore it.

Finally, the question of how to enforce floating point contracts came up. To check and/or enforce that the contracts which the programmer agreed to are being kept, it would be useful to dynamically check for violations, which would then allow easy debugging. To deal with that, an extension to UBSan, which already includes some small FP checks, was suggested.

4.4 Notes from Breakout Session: "Reproducibility in Floating-Point Computation"

Thomas Wahl (Northeastern University – Boston, US)

Sources of non-reproducibility

- Not necessarily related to FP: exists way beyond FP.
- Test question: is integer addition associative (and hence reproducible, irrespective of evaluation order)? No, due to "one-sided overflows"

Dangers of non-reproducibility

- Do people even notice that there is non-reproducibility? Unlike classical crashes, you won't run into such issues by chance: as long as you play only with your production machine, they will not emerge.
- Nondeterminism in debugging: print instructions, assertions can change memory layout of your program and prevent/enable use of FMA.

- More generally known as "Heisenbugs": they disappear only because you try to find them. Agreed that we need reproducibility beyond just the debugging phase.
- Non-reproducibility can also be a valuable indicator for problems in the code: if results depend on computation platform, they are bound to be numerically unstable, irrespective of the accuracy question.

Enforcing reproducibility

- Easy if you ignore the other numeric quality criteria (performance, accuracy), but probably these cannot/should not be separated.
- Even just enforcing reproducibility alone raises questions: you can determinize your code e.g. towards "strict evaluation" (supported by many compilers), but such arbitrary choice is likely not the best for accuracy and performance. "We don' know what to reproduce."
- Better/more realistic definition than bit-precise reproducibility: preserve control flow; allow exception: ignore differences in bit-patterns for NAN. Need a specification language of reproducibility: what am I willing to accept.
- Could specify that I allow optimizations like constant folding to ignore differences in L/R association and just fold `X * 3.5 * 4.2` to `X * 14.7`.
- Separating sources of non-reproducibility: architecture differences from compiler decisions: maybe it makes sense to determinize compiler decisions for a given fixed architecture.

Future of reproducibility

- (Exponentially) increasing diversity of architectures and compilers will make it harder, rather than easier.
- Domain-specific applications (matrix operations, HPC) will always produce highly specialized/customized architectures/compilers that have very little expectations of (and need for) reproducibility.
- Important to realize/accept that reproducibility is certainly not the top concern everywhere (we'd be happy if it got sufficient attention in SOME areas).
- For other domains (controllers, decision-making programs) we can afford more expensive techniques to worry about R.
- Similar like in verification: it is expensive, so rather than verifying everything, we focus on safety-critical applications.

We did not discuss much the relationship with other numeric quality criteria (performance, accuracy), other than observing that we cannot really look at these aspects in isolation.

5 Panel discussions

5.1 Notes from Joint Panel Discussion between the "Analysis and Synthesis of Floating-Point Programs" and "Machine Learning and Formal Methods" Seminars

Alastair F. Donaldson (Imperial College London, GB)

The following notes aim to capture a panel discussion held between concurrently held Dagstuhl seminars on "Analysis and Synthesis of Floating-Point Programs" (17352), and "Machine Learning and Formal Methods" (17351)

The panelists were:

- Eva Darulova (MPI-SWS – Saarbrücken) – representing the Floating-Point seminar
- Miriam Leeser (Northeastern University) – representing the Floating-Point seminar
- Charles Sutton (University of Edinburgh) – representing the Machine Learning and Formal Methods seminar
- Sasa Misailovic (University of Illinois – Urbana Champaign) - representing the Machine Learning and Formal Methods seminar

The panellists gave a brief introduction, which was followed by a 40-minute discussion. Alastair Donaldson took notes on the questions and answers during this discussion, which are reproduced below. These are merely notes summarising broadly the points that were made and questions asked - they by no means represent verbatim what was said by the various participants.

Alastair Donaldson To what extent is reproducibility of floating-point computations important in machine learning?

Charles Sutton

- Of course floating-point reproducibility would be nice in principle
- However, there are so many sources of approximation in the domain already, so that this is one of many problems
- Given that there are a long list of things that introduce approximation, it would be useful to know when round-off error is moving up this list of problems – techniques to help with that would be useful.

Miriam Leeser Given that the "Machine Learning and Formal Methods" seminar has clear links with verification, I'd be interested in a perspective on numerical issues in the context of verification.

Sasa Misailovic My view, which may not be representative of the entire formal methods community, is that properties are divided into two categories: (1) properties related to safety of computation (no crashes, no out-of-bounds accesses, that mass and gravity do not become negative, etc.), and (2) accuracy of the result, which can sometimes be verified, but can sometimes be let go or estimated, depending on the application domain.

Charles Sutton You can put the arrow in "Machine Learning and Formal Methods" in either direction, so that another approach is to apply machine learning as a tool in formal methods.

Ganesh Gopalakrishan A colleague is going to give a talk in Salt Lake City's public library to members of the public on the topic of the role of computer algorithms in making decisions that might "decide their fate". I'd like a perspective on what machine learning practitioners can say to reassure the general public that certain levels of guarantees, from formal to pragmatic to social, can be provided?

Charles Sutton

- People are starting to think more and more about this, but with no final answers.
- On a social level, if a machine learning method is making sentencing recommendations to judges, how can you argue that it is unbiased? There are several approaches that could be taken – not yet clear which way is best.
- There is also the point of view that the method may not be biased, but rather that the learned model has been trained on biased data.

- In cyber-physical systems the guarantees required are different – in a deep learning model trained in one situation, might a small perturbation cause very different behaviour?
- People are well aware that these issues exist, but it is not clear what to do about it.
- Also there is a pedagogical need to train our students regarding to what we should not do with machine learning and why – a case in point being a paper on arxiv that upset people by trying to predict criminal records based on photos of faces.

Miriam Leeser What do you verify in the context of machine learning applications if you have no ground truth? We have a similar issue in reasoning about software that uses real number approximations.

Sasa Misailovic From a compiler's perspective you can treat the input program as a gold standard, with the rule that whatever transformations you apply you should get something equivalent. It's not clear what the analogue of this would be in machine learning.

Susmit Jha Floating-point round-off can insert biased noise into a machine learning model. Would machine learning experts prefer floating-point arithmetic that doesn't have a deterministic semantics, so that when you do some kind of operation that involves rounding the direction of round-off is random? To the floating-point guys: do you believe non-deterministic semantics to be limiting from a hardware point of view, or can you provide some probabilistic guarantees? To machine learning guys: do you agree that it would be useful to have unbiased rounding?

Jim Demmel A certain civil engineering firm wanted a reproducible parallel linear system solver because their customer was required by lawyers to have reproducibility. Will that come up in e.g. autonomous driving?

Charles Sutton This comes up more in the context of "explainability" rather than "reproducibility" – there is a need to be able to give an explanation for why, though not clear what form that should be.

George Constantinides I'm aware of work in machine learning on using very low precision arithmetic. Usually papers say that they can recover all or almost all of performance by incorporating this into training. Question is: if I could provide you with a 100 fold speedup but far less rigorous error guarantees would you take it? [Laughter in the room.] Secondly, if you could incorporate this in the training, what kind of spec would you like me to provide you with?

Charles Sutton For the first question: all of these methods are intensive to train. Part of reason for impressive DeepMind papers is they have 1000 times more GPUs than anyone else – this gives them a very fast development cycle. If you give me something 100x faster you've made my development cycle much faster. Can you clarify what you mean regarding a spec?

George Constantinides Training relies on properties of the network that is being trained, so what is it that you'd want?

Charles Sutton I'm maybe not best person to ask, but e.g. that noise is unbiased. Might not be a requirement, but seems natural.

Thomas Wahl Coming back to explainability: presumably one aspect of this is providing a level of confidence that I'm aware of noise, and can therefore say that confidence is not high. In floating-point we are good at knowing that a result is likely unstable. My understanding is that there are lots of examples where learners report a wrong decision with high confidence. Is that a problem for you guys?

Charles Sutton It's something we care about. Many learning algorithms we use can report a confidence, but are just learning confidence from data so can be over-confident about wrong answers.

Thomas Wahl This reminds me of a situation in verification where we generate a proof script that can be independently checked by another tool, increasing confidence substantially. Is there an analogue in machine learning?

Charles Sutton That reminds me of "ensemble" methods in machine learning. One thing we know about machine learning is that if we average together different learners we will be better, but it's still just a learner – there is no free lunch.

Eva Darulova are there cases where you have instabilities in training or classification, such that debugging tools would help you get a handle on what is going wrong?

Charles Sutton Could be – often with a deep network I don't even know and blissfully pretend they are not – bursting my bubble might make me sad! [Laughter in the room]. There are cases where we know things are bad – certain linear algebra operations for example. If working with probabilities over structured objects, we may know these things are affected by numerical precision so we do things with logarithms.

Sasa Misailovic My question to you guys [the floating-point seminar participants]: is it possible to check whether an algorithm would converge to a solution after doing some kind of perturbation, e.g. a gross numerical approximation?

George Constantinides What we want is a ranking function argument, and there is some work on introducing round-off error into ranking function arguments.

Miriam Leeser A question for verification people who use machine learning: how worried are you about issues with machine learning being unstable, etc.?

Armando Solar-Lezama Generally we don't use the output of a machine learning component as part of proof. We use it to guide the search process, e.g. searching for a proof or witness or program. Machine learning can be very useful in guiding that search, but ultimately if it is wrong it doesn't matter – it might just slow things down – but you're going to check that the artifact you're looking for is correct independent of machine learning.

[Did not catch speaker] In specification learning you might get false positives as a result of inaccuracy in machine learning, but there are false positives in program analysis anyway.

[Did not catch speaker] I'd add a bit: I'm concerned by this. The solvers we use in the verification community often don't terminate, but if they do we have high confidence in their answers. When we use e.g. belief propagation this don't terminate sometimes; we don't know why – we inject our own hacks and retry. We're not using these for things for which we need proofs – but we might in the future and then this would be a concern.

[Did not catch speaker] In tools for dynamical systems, proofs can come from semi-definite programming. We have experienced proofs that are very numerically sensitive – if you change a digit in an insignificant place, the proof no longer holds. You need to be very careful about such things. If you use machine learning to do invariant learning this problem may also persist.

Martin Vechev What's your take on being able to see a program that [something about neural automata – did not catch the full question]?

Charles Sutton

- You're alluding to two different areas.
- Neural Turing machines – let me unpack that. I wouldn't say they are particularly interpretable. There are a lot of people doing research on interpretability in machine learning, so it's a hot topic.
- You're asking whether it is real or a fad?
- Seems like it would be real – hard to see people trusting deep learning in e.g. precision medicine unless for an individual prediction you could give some evidence.
- You could argue there are cases where you don't care – perhaps with self-driving cars you're happy if they work and satisfy stability properties – but don't want your car to explain everything that it's doing. [Laughter in the room.]

Susmit Jha I've seen work in fixed-point where I can give accuracy bound [for an algorithm] and it can figure out what width to use [to meet this bound in an efficient manner]. Are there similar things in floating-point where I can tell you the expected accuracy and you can figure out most efficient floating-point types?

George Constantinides The question implies that you now have a specification for accuracy requirement – what would such a specification look like?

Susmit Jha As a an extreme I might say that output error should be bounded by actual evaluation with infinite precision. I'd like a compiler that can take my program, compile it into floating-point operations, such that it consumes less energy but meets the spec.

George Constantinides A lot of people in our seminar working on that.

Miriam Leeser But we almost never get specs from users.

Jerry Zhu We typically specify this with an evaluation stack or test set. We can at least say: "on this data you should maintain accuracy to some degree".

Cindy Rubio-González Would you care if it turned out that on other data sets the guarantee would not hold? I.e., we make this work for your training set, but there is no proof it will work in general.

Jerry Zhu Our basic assumption is that things are distributed in an even manner; might be parts of the world where things don't work so well. I would then need to redefine my loss function.

George Constantinides The benefit right now for machine learning applications is to talk about expectation of some error over a data set or some space, rather than worst case error, which is what most tools do.

Susmit Jha In terms of hardware accelerators: is it possible to have an accelerator tune its precision to meet needs of application?

Miriam Leeser there are many accelerators. GPUs are fixed. FPGAs provide the freedom to define what you want. The ability is there, but tool flows are not sophisticated. I don't know about the new tensor processor, but they've picked something and fixed it.

Susmit Jha What are the options?

Miriam Leeser On an FPGA you can do anything, which makes it so hard to write good tools! It depends on what you're targeting.

Sam Elliott Going back to GPUs: if it's needed, the next generation will have it – so GPU vendors can be influenced.

Alastair Donaldson [To Susmit Jha] Did you mean tuning precision in real time?

Susmit Jha Yes – is it possible to reconfigure the FPGA on the fly so that I can do something in this precision then that precision with very short latency?

Miriam Leeser Switching time between designs can be milliseconds, but design time can be days.

Jim Demmel Some problems are well conditioned and can be solved at various precisions, others are not. Are the problems important in machine learning well conditioned?

Charles Sutton By and large yes.

Zvonimir Rakamarić I'm curious what people are doing in this area. For me one interesting question is: once you put a neural network in a safety critical system, how do people analyse this? Are there already capable verification approaches?

Sanjit Seshia There are verification approaches for cases where decisions made by a system are based on outputs from a neural network. Some approaches treat this component as a black box. There is work within the verification community on trying to analyse neural networks in isolation; some of that scales less well than networks used in industrial applications, so there is a gap. I don't think any of it is doing a detailed analysis at the floating-point level.

Martin Vechev There are 3-4 papers I'm aware of but they usually work with small feed forward networks.

Eva Darulova There was a recent paper on CAV related to this, in which verification was over reals while the implementation was over floats.

Martin Vechev [Asked a question about the use of fp16, which I did not catch]

Charles Sutton People must have done experiments on this, as lots of people use fp16 – people have results for particular applications where they use extreme low precision.

George Constantinides I've seen cases where severe roundoff error is akin to a regularizer and can avoid over-fitting.

Miriam Leeser There must be an optimization point there which you can aim for.

Participants

- Erika Abraham
RWTH Aachen, DE
- George A. Constantinides
Imperial College London, GB
- Nasrine Damouche
University of Perpignan, FR
- Eva Darulova
MPI-SWS – Saarbrücken, DE
- James W. Demmel
University of California – Berkeley, US
- Anthony Di Franco
University of California – Davis, US
- Alastair F. Donaldson
Imperial College London, GB
- Theo Drane
Cadence Design Systems – Bracknell, GB
- Sam Elliott
Imagination Technologies – Kings Langley, GB
- Ganesh L. Gopalakrishnan
University of Utah – Salt Lake City, US
- Hui Guo
University of California – Davis, US
- Jeffrey K. Hollingsworth
University of Maryland – College Park, US
- Miriam Leeser
Northeastern University – Boston, US
- Daniel Liew
Imperial College London, GB
- Piotr Luszczek
University of Tennessee – Knoxville, US
- Victor Magron
VERIMAG – Grenoble, FR
- Matthieu Martel
University of Perpignan, FR
- Guillaume Melquiond
INRIA – Gif-sur-Yvette, FR
- David Monniaux
Université Grenoble Alpes – Saint-Martin-d'Hères, FR
- Magnus Myreen
Chalmers University of Technology – Göteborg, SE
- Santosh Nagarakatte
Rutgers University – Piscataway, US
- Pavel Panchekha
University of Washington – Seattle, US
- Sylvie Putot
Ecole Polytechnique – Palaiseau, FR
- Zvonimir Rakamarić
University of Utah – Salt Lake City, US
- Nathalie Revol
ENS – Lyon, FR
- Cindy Rubio-González
University of California – Davis, US
- Daniel Schemmel
RWTH Aachen, DE
- Oscar Soria Dustmann
RWTH Aachen, DE
- Zachary Tatlock
University of Washington – Seattle, US
- Michela Taufer
University of Delaware – Newark, US
- Laura Titolo
National Institute of Aerospace – Hampton, US
- Thomas Wahl
Northeastern University – Boston, US