Debugging and optimization of HPC programs in mixed precision with the Verrou tool

François Févotte, Bruno Lathuilière

To cite this version:

François Févotte, Bruno Lathuilière. Debugging and optimization of HPC programs in mixed precision with the Verrou tool. 2019. ⟨hal-02044101⟩

HAL Id: hal-02044101
https://hal.science/hal-02044101v1

Preprint submitted on 21 Feb 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Debugging and optimization of HPC programs in mixed precision with the Verrou tool

François Févotte[1] and Bruno Lathuilière[1]
francois.fevotte@edf.fr, bruno.lathuiliere@edf.fr
[1]EDF R&D, Palaiseau, France

November 12, 2018

Abstract

The analysis of Floating-Point-related issues in HPC codes is becoming a topic of major interest. First, parallel computing and code optimization often break the reproducibility of numerical results across machines, compilers and even executions of the same program. Second, mixed precision is more and more used to reduce memory footprint and bandwidth wherever possible, thereby benefiting more from SIMD units.

This paper presents how the Verrou tool can help during all stages of the Floating-Point analysis of HPC codes: diagnostic, debugging and optimization. Recent developments of Verrou are presented, along with examples illustrating the interest of these new features for industrial codes.

As an example, Verrou allows debugging code_aster, an industrial, parallel, mixed-precision code for the simulation of structural mechanics. Re-using existing infrastructure from the non-regression testing process, setting up Verrou requires limited effort. The first useful debugging information can be obtained after only 25 minutes, while the full automated debugging process takes 10 hours to complete.

Keywords: Floating-point arithmetic, Verrou, mixed precision, Verification & Validation

1 Introduction

Floating-Point (FP) arithmetic is becoming a hotter and hotter topic in High-Performance Computing (HPC). First, high computational performance is often achieved at the expense of a loss of control over the order in which FP operations are executed. Vectorizing code is a crucial part of the necessary optimizations to leverage the full power of modern CPUs [22]. However, loop vectorization (especially for reductions) often requires using algorithms which have the same semantics as sequential ones for real, but not FP arithmetic.

Even without changing the algorithms and source code of a program, aggressive compiler optimizations often achieve greater performance by changing the order in which FP instructions are executed (*e.g.* automatic vectorization) or simplifying arithmetic expressions (using real arithmetic semantics).

Moreover, the exact same binary executable is not always executed in the same way on modern hardware: whether parallelism is used at the scale of multi-core processors, multi-processor nodes or multi-node clusters, it involves running concurrent computations, often without much control

on the order in which they are executed (the cost of a synchronization ensuring repeatability in the order of computation is often considered too large).

Since FP arithmetic is not associative, this change in the order of operations leads to a loss of the reproducibility of the results, which can in turn have a wide array of consequences, ranging from benign difficulties in the debugging process, to more serious issues with the Verification & Validation (V&V) process.

Second, optimizing the use of FP precision is often key to achieving high performance. Using smaller-precision FP numbers allows reducing the memory bandwidth usage as well as increasing the number of simultaneous FP operations performed by a single SIMD instructions. Therefore, high FP precision should only be used where it is needed to ensure the required results quality; FP precision should be reduced everywhere possible. With the current trend of ever increasing SIMD register widths (for example, AVX-512 introduces 256-bit registers), and the large number of available FP formats since IEEE 754-2008 [14], such code optimizations are not to be neglected. Over the last few years, the use of mixed precision has therefore become a point of major interest for the HPC community [18, 24, 21]. However, it is important to keep in mind that optimizing mixed precision programs should always be considered as a search for the optimal balance between results accuracy and run times. The quantification of FP-related losses of accuracy is an essential part of this process.

In this paper, we present how the Verrou tool [9] can help deal with these issues in the context of large, industrial, high-performance scientific computing codes such as the ones developed and used by leading actors in the industry. Verrou is a free tool based on Valgrind, available under an open-source licence at `http://github.com/edf-hpc/verrou`.

Verrou is mainly developed by EDF, the main French electric utility, which heavily relies on Scientific Computing Codes (SCC) and numerical simulation to operate its industrial processes. As such, Verrou primarily targets development teams which need techniques and tools to address FP-related issues, both for the development and debugging of SCCs and for their Verification & Validation.

Part 2 of this paper presents an overview of the FP analysis techniques and tools. In part 3, the Verrou tool and its architecture will be briefly presented. Emphasis will be put on the recent development of new features, released in version 2.1.0 and specifically targeting high-performance codes. This part will also concentrate on the core features of Verrou, which are related to the diagnostic of FP issues. This is in contrast to part 4, where the debugging features of Verrou and its ecosystem will be detailed. Part 5 will be devoted to more methodological topics and show how Verrou can be used to debug large industrial scientific computing codes and help optimize their use of mixed precision.

The interest of the various features described in this paper will be illustrated by examples taken from the analysis of code_aster [2], a structural mechanics simulation tool of more than 1.2M lines of code. A previous study [11] details the complete analysis and debugging of code_aster, and shows how the FP analysis process with Verrou can greatly complement a more traditional Quality Assurance (QA) process. In the case of code_aster, this QA process relies on around 4 000 test cases, of which around 2 000 are run daily.

2 Overview of FP analysis techniques & tools

Studying the impact of Floating-Point arithmetic on the quality of results computed by scientific computing codes is not a new topic. Even after it was standardized in IEEE-754, FP arithmetic has been considered a tricky and error-prone subject [12], and much work has therefore been

devoted to tools and techniques helping to diagnose FP-related losses of accuracy in computing codes.

2.1 Guaranteed methods

Some of these techniques, such as Interval or Affine Arithmetic [23, 15] give strong guarantees on the results, but can produce largely overestimated error bounds and are often limited to small programs. Such techniques and tools are therefore much used for the verification of small but crucial parts of larger programs (such as for example the computation of mathematical function in the standard `libm` [7]) or of full programs which can have large impacts (such as all Cyber-Physical Systems [3, 1]).

An important advantage of these guaranteed methods is their ability to provide guarantees for a range of inputs, in contrast to other techniques presented hereafter, which often provide only unwarranted indications that are only valid for a single input dataset.

2.2 Comparison between several precisions

Other techniques can be used, such as the comparison between an analyzed result and a reference computed in higher precision. Such techniques offer far less guarantees, but are far simpler. For example, fpDebug [4] performs the Dynamic Binary Instrumentation (DBI) of an input executable, in order to compare each floating-point value in the program with one obtained with higher precision. FpDebug outputs a graph indicating, for each floating-point operation, whether inaccuracies come from input data or from the operation itself.

The rpe [6] library uses the same idea in a reversed way: it emulates the use of reduced floating-point precision in order to estimate whether some parts of the program might be re-developed in reduced precision without entailing too much change in the results.

Tools like Precimonious [24] also rely on comparisons between single-, double-, and quadruple-precision results in order to automatically determine which variables in a program can benefit from using a lower precision without perturbing the results too much. This results in an optimized mixed-precision implementation. In this case, program instrumentation is performed at the compilation level (based on LLVM).

2.3 Stochastic arithmetic

Between theses two classes of techniques, the broad family of stochastic methods give far less guarantees, but are much more adapted to large and complex programs such as those developed by the HPC community to meet the industrial need for high-fidelity simulations of physical systems. Stochastic methods model the inaccuracies introduced by round-off errors as uncertainties on the results of computations. These uncertainties are estimated by transforming inaccurate results into random variables, which can be sampled and studied in a statistical way. Two main stochastic methods have emerged.

On the one hand, the CESTAC methodology [27] models round-off errors by random rounding: the result of every inexact computation in a program is randomly rounded upward or downward. This effectively transforms the global result of a complete computation into a random variable, of which the standard deviation can be linked (under some hypotheses) to the inaccuracy of the results. CADNA [16, 19] is the most used library implementing Discrete Stochastic Arithmetic (DSA), a synchronous variant of CESTAC in which all random samples of a given operation are computed at once. Using CADNA requires accessing and modifying the source code of the analyzed program, which is not always easy or even possible for large code bases. Nevertheless,

successful uses of CADNA have been reported on large simulation codes [20], which validates the adequateness of the CESTAC method in industrial contexts.

PROMISE [13] is a mixed-precision optimization tool which leverages the same kinds of algorithms as Precimonious to find optimal sets of variables which could use lower-precision floating-point numbers. In contrast to Precimonious, PROMISE relies on CADNA to assess the validity of the mixed-precision configurations that it considers.

On the other hand, Monte-Carlo Arthmetic (MCA) [26] models several types of FP-related problems while allowing the user to choose an arbitrary “virtual precision”. This allows estimating the behavior of a given program with a different precision than what appears in its source code, in contrast to CESTAC, which always models round-off errors at the precision which is implemented in the program. MCA is implemented in various libraries such as `mcalib` [10]. Tools such as Verificarlo [8] allow leveraging the full power of MCA for the analysis of large, industrial code bases. Verificarlo presents itself as a custom compiler based on LLVM, and implements MCA by instrumenting programs at the end of the compilation, after most optimization passes.

This paper focuses on Verrou, an FP analysis tool which performs the Dynamic Binary Instrumentation (DBI) of any given program. This eliminates the need for an instrumentation of the program sources or even a recompilation.

Verrou originally addressed diagnostics and debugging, and implemented several forms of random rounding: an asynchronous CESTAC method, and a variant of MCA with virtual precision equal to the real precision. We describe hereafter how recent versions of Verrou have been extended to implement other forms of FP analysis, and also address mixed-precision-related issues.

3 Presentation of the Verrou tool

Verrou is a tool aiming at helping diagnose, debug and optimize FP-related issues in large, industrial scientific computing codes. An earlier version of the Verrou tool has been presented in [9], where internal arithmetic-related algorithms were presented in detail. Only the most important aspects of Verrou will be recalled here, and this paper will rather focus on recent developments in Verrou, released in version 2.1.0. Specifically, the features related to the use of Verrou for the diagnosis of mixed precision programs will be detailed. Also, this paper will focus more on the presentation of the features themselves, and less on the internal implementation.

From a user standpoint, Verrou instruments the program (in its binary form), replacing each FP instruction in it with a variant implementing another type of FP arithmetic. Results of the computation are output like in any normal execution, except that they are affected by the cumulative effect of all perturbed FP instructions. Analyzing the observed change in the results allows estimating the global impact of FP arithmetic for the code.

In its recent versions, the architecture of Verrou closely follows this description, with 3 major components which will be described in more details in the remainder of this section.

- The *front-end* is the part of Verrou responsible for running the user-given program, replacing each FP instruction with a variant.
- The *back-end* implements variants of all FP instructions, effectively defining a new arithmetic to be used in place of the original one. Different back-ends can be used in Verrou, implementing various arithmetics.
- A *post-processing* stage is needed in order to analyze the effect resulting from the use of an alternate arithmetic. This is mostly left for the user to implement, but advice can be given as to what kind of post-processing should be performed.

This architecture of Verrou in three parts stems from the Interflop[1] initiative, which aims at providing a uniform API allowing interoperability between front-ends and back-ends of several FP intrumentation tools such as Verrou and Verificarlo [8].

3.1 Front-end

Verrou is based on Valgrind to perform a Dynamic Binary Instrumentation (DBI) of the given program, and replace each FP instruction by a variant implemented by the user-chosen back-end.

The main advantage of using Valgrind is the ease of use of Verrou for end-users: there is no need to manually change the program or even recompile it. All that is needed is to prefix the usual command-line with in order to invoke Valgrind with the Verrou tool:

```
valgrind --tool=verrou [VERROU_ARGS] PROGRAM [ARGS]
```

The development teams of many scientific codes already rely on Valgrind for their memory debugging, so that defining such command-line prefixes might even already be part of the standard development-test-debugging routine.

It is also worth noting that relying on DBI makes the core parts of Verrou independent of the tools or languages used for the development of the program. Verrou is therefore language-agnostic: it works for multi-language applications such as code_aster [11], which internally uses Fortran, C, C++ and Python. Successful uses of Verrou have also been reported with languages less commonly used by the HPC community, such as JAVA or Rust. For the same reasons, Verrou works with any compiler and set of compilation options, which allows instrumenting programs exactly as they will later be used for industrial purposes. A consequence of great importance for the HPC community is that Verrou is naturally compatible with any MPI implementation:

```
mpirun [MPI_ARGS] valgrind --tool=verrou [VERROU_ARGS] PROGRAM [ARGS]
```

Also, programs analyzed with Verrou are not restricted in the libraries that they use: Verrou can analyze any part of the program as a whole, even third-party libraries, even closed-source. Again, it might be of interest for the HPC community to note that Verrou can also instrument multi-threaded applications using any framework (pthreads, tbb, OpenMP...). But an important limitation of Valgrind in this case is that threads will be "sequentialized", which may incur a large performance overhead.

In the commands above, `VERROU_ARGS` allows changing several aspects of Verrou, particularly the arithmetic back-end to be used (described in the next paragraph). It is also worth noting that the Verrou front-end can be instructed to only instrument part of a program. This feature is key to the debugging methodologies described in part 4.

3.2 Back-ends

Several back-ends in Verrou define variants of FP arithmetic to be used in place of the standard FP arithmetic. The back-end to be used in a given analysis can be selected using the `--rounding-mode` command-line option to Verrou.

Before entering the details of what type of arithmetic is implemented in each back-end, we discuss here a generic feature which all back-ends implement. When a Not-a-Number (NaN) appears in the results, Verrou emits a Valgrind error, which is reported to the user, along with the current back-trace. When combined with the usual Valgrind debugging features, this allows for an integrated debugging process.

[1]https://github.com/interflop/interflop

Another interesting point to mention here is the verification process of Verrou itself. Recent versions rely on the UCB Floating-Point tests database[2] to check the correctness of all arithmetic backends in all cases. This includes tricky corner cases such as denormalized numbers, non-finite values, positive and negative zeros, *etc.* Provided that Verrou has access to Fused Multiplication and Addition (FMA) operations, its back-ends correctly pass all UCB tests.

The verification process of Verrou also relies on a set of small programs which include various types of operations (including FMA). For these programs, tests are performed to check that rounding modes emulated by Verrou produce the exact same results as when the program is run without Verrou and the corresponding roundings are performed by the hardware.

3.2.1 Deterministic Rounding (DR)

In order to debug Verrou itself and check for instrumentation-induced non-reproducibilities, a Deterministic Rounding back-end re-implements within Verrou any of the 4 standard rounding modes defined by IEEE-754. This back-end can also be used as a cheap way of assessing the impact of round-off errors in FP computations. As discussed in [17], some insight on FP errors can be obtained from the comparison of a few results obtained with the same program, using different rounding modes.

As an extension to this scheme, an additional 5$^{\text{th}}$ deterministic rounding mode can be emulated by Verrou. Named "farthest" (`--rounding-mode=farthest`), this mode always rounds inexact computations in the opposite way to the standard round-to-nearest mode.

3.2.2 Random Rounding (RR)

The historical Verrou backend implements a stochastic arithmetic variant in which the result of each inexact FP instruction is randomly rounded either upward or downward. Depending on the chosen probability law, this can be similar to an asynchronous CESTAC arithmetic [27] or a variant of Monte-Carlo Arithmetic [26].

- `--rounding-mode=average`: in this mode, the probability of rounding upward or downward is determined such that the expected value of the randomly rounded result equals the real mathematical result. Suppose an operation $z = x \circ y$ is to be computed, where x and y are the (FP) operands, z is the real result, and $\circ \in \{+, -, \times, \div\}$ is the operator. The randomly rounded result Z is defined with probabilities given by

  $$\mathbb{P}[Z = \underline{z}] = \frac{\bar{z} - z}{\bar{z} - \underline{z}} \qquad \text{and} \qquad \mathbb{P}[Z = \bar{z}] = \frac{z - \underline{z}}{\bar{z} - \underline{z}},$$

  such that

  $$\mathbb{E}[Z] = z.$$

  In the equations above, $\underline{z}$ and $\bar{z}$ respectivey denote the value of z rounded downward and upward. With the terminology defined in [26], this mode is equivalent to an "output precision bounding" MCA in which the virtual precision is set to the actual FP precision.

- `--rounding-mode=random`: in this mode, roundings are upward or downard equiprobably. With the notations defined above, we have

  $$\mathbb{P}[Z = \underline{z}] = \mathbb{P}[Z = \bar{z}] = \frac{1}{2}.$$

  This is equivalent to an asynchronous CESTAC arithmetic.

[2] http://www.netlib.org/fp/ucbtest.tgz

```
---------------------------------------
 Operation           Instruction count
  `- Precision
      `- Vectorization
---------------------------------------
 add                  216251889
  `- flt                    4761556
      `- llo                    2074760
      `- vec4                    328316
      `- vec8                   2358480
  `- dbl                  211490333
      `- llo                  210679861
      `- vec2                    808672
      `- vec4                      1800
---------------------------------------
 sub                   32673605
  `- flt                    6115214
      `- llo                    6115214
  `- dbl                   26558391
      `- llo                   26558391
---------------------------------------
 mul                  327064720
  `- flt                   12151586
      `- llo                    9464790
      `- vec4                    328316
      `- vec8                   2358480
  `- dbl                  314913134
      `- llo                  313498574
      `- vec2                   1412796
      `- vec4                      1764
---------------------------------------
```

```
---------------------------------------
 Operation           Instruction count
  `- Precision
      `- Vectorization
---------------------------------------
 div                    23350051
  `- flt                      597054
      `- llo                      597054
  `- dbl                    22752997
      `- llo                    22752997
---------------------------------------
 mAdd                   71551834
  `- flt                    71543338
      `- llo                    71543338
  `- dbl                        8496
      `- llo                        8496
---------------------------------------
 conv                    2227250
  `- dbl=>flt                 614010
      `- scal                     614010
  `- flt=>dbl                1594854
      `- scal                    1594854
  `- dbl=>int                    707
      `- scal                        707
  `- dbl=>sht                  17679
      `- scal                      17679
---------------------------------------
```

Figure 1: Number of FP operations output by Verrou for a run of code_aster on test case `ttlv300a`

Changes to this back-end have recently been introduced in order to correctly verify mixed-precision programs: type conversions subject to rounding (*e.g.* `double` → `float`) are now instrumented and perturbed when applicable. To the authors' knowledge, Verrou is the first tool based on stochastic arithmetic which correctly handles type conversions.

Figure 1 illustrates the need for correctly handling these conversions. This listing presents the number of FP operations, as output by Verrou at the end of the execution of a code_aster calculation. This calculation (performed on the `ttlv300a` test case) makes use of complex third-party libraries, such as the PETSc double-precision fGMRES solver preconditioned by a single-precision direct linear solver from the MUMPS library. As can be seen in the number of operations, a significant fraction of the FP operations have been performed in single precision. For example, around 19% of the subtractions, and more than 99% of the FMAs (Fused Multiplication and Addition) have used single precision in this case. Around 28% of the type conversions are potentially inexact `double` → `float` conversions.

3.2.3 Reduced Precision (RP)

A Reduced-Precision back-end has been introduced in newer versions of Verrou. As its name suggests, this back-end emulates the use of an IEEE-754-compliant FP arithmetic with reduced precision. It is currently implemented in only one variant, which reduces double precision FP numbers to single precision. This is similar to what is performed in the rpe library [6], except that Verrou performs the analysis directly at the level of the executable binary, rather than by instrumenting the sources like what is needed for rpe.

For legacy Fortran codes, which often feature a monolithic architecture, such a feature can be an easy way to foresee any problem which might arise when reducing the FP precision of variables. For more modern codes, for example developed in C++ with templated FP type, developers are often better served by simply recompiling their project with the reduced precision.

This back-end really shines when it is combined with the Verrou features allowing to restrict the scope of perturbations to only part of a program: functions and/or source code lines. In this

case, Verrou allows emulating any mixed-precision variant of the program, where an arbitrary part uses single-precision arithmetic, while the rest remains in double precision. Debugging and optimization methodologies relying on such techniques are detailed in part 5.

3.3 Post-processing

Due to the changes introduced by the arithmetic emulated by Verrou, each execution of a given program (with a given dataset) produces perturbed results. An analysis of these results should be performed in order to assess the impact of FP arithmetic on the results quality.

Such post-processing is highly dependent on the nature and format of the results and, as such, it is mostly left for the user to implement for their own code. However in most instances, only a limited amount of work is needed when a Quality Assurance process with automated non-regression testing has been put in place. For example, it is possible to simply check whether the non-regression test suite succeeds when running the program within Verrou with the reduced-precision arithmetic back-end. If so, this gives a good indication that the program can be changed to use single-precision FP arithmetic while remaining "valid".

Post-processing can get more complicated in the case of stochastic arithmetic, where several sampled results are needed to compute statistics. This aspect is detailed in [25]. It is shown there that several statistics can be computing, yielding different accuracy estimators along with the corresponding confidence intervals. In particular, if it can be assumed (or verified in practice) that perturbed results follow a Gaussian distribution, the authors recommend computing the quantity

$$\hat{s}_{\text{CNH}} = -\log_2\left(\frac{\hat{\sigma}}{\hat{\mu}}\right) - \delta_{\text{CNH}},$$

where $\hat{\mu}$ and $\hat{\sigma}$ respectively denote the sample average and variance, and δ_{CNH} is a tabulated quantity depending on the number of samples and the targeted level of confidence. The computed quantity $\hat{s}_{\text{CNH}}$ gives a lower bound for the number of significant bits (*i.e* bits which can be expected to be found in agreement between successive runs of the same computation).

4 Debugging HPC codes with Verrou

The core features of Verrou which have been presented above aim at diagnosing and quantifying FP-related losses of accuracy. However, these features can also be leveraged to set up methodologies aiming at debugging FP-related issues in industrial codes.

This debugging methodology is based on two techniques, aiming at detecting different kinds of FP-related issues.

1. *Unstable tests* are tests for which the branch taken depends on previous round-off errors). These can be detected by combining a Verrou analysis to a code coverage analysis. The analyzed program can be recompiled so that it generates code coverage data[3]. After a standard run and another run perturbed by Verrou, code coverages can be compared in order to determine where execution flows differ.

2. *Unstable functions* (or source code lines) are parts of the code which, when perturbed, produce large perturbations in the results. The detection of such unstable code parts uses

[3] With `gcc`, this is performed for example using the `-fprofile-arcs -ftest-coverage` command-line switches during compilation. Later, code-coverage data can be analyzed using the `gcov` utility.

Data: Δ_{search}: search space (set of symbols or lines)
Data: Δ_{fixed}: list of validated deltas (default: empty)
Data: g: granularity (default value 2)

$\{\Delta_1, \dots, \Delta_g\} \leftarrow \text{split}(\Delta_{\text{search}}, g)$;

for $j \in [\![1, g]\!]$ **do**
  if $\texttt{test}(\Delta_{\mathit{fixed}} + \Delta_j) = ✓$ **then**
    return $\text{DDmax}(\Delta_{\text{search}} - \Delta_j, \Delta_{\text{fixed}} + \Delta_j, g - 1)$;
  end
end

if $g = \text{size}(\Delta_{\mathit{search}})$ **then**
  return Δ_{fixed} ;
end

return $\text{DDmax}(\Delta_{\text{search}}, \Delta_{\text{fixed}}, \min(2\,g, \text{size}(\Delta_{\text{search}})))$;

Algorithm 1: DDmax, as implemented in `verrou_dd`

the Delta-Debugging [28] algorithm to perform a binary search in the source code. It heavily relies on the Verrou ability to restrict the scope of FP perturbations to a part of the source code. A dedicated utility in the Verrou ecosystem, `verrou_dd`, automates most of the Delta-Debugging process.

No specific development has been made recently regarding unstable tests detection, and this feature will not be detailed further here; the interested reader can get the details in [11]. However, recent improvements have been incorporated into `verrou_dd`. They are described in more details in the following paragraphs.

4.1 Delta-Debugging algorithms

The Delta-Debugging methodology [29, 28] is built in the Verrou ecosystem thanks to the `verrou_dd` utility, which automates the full process. The Delta-Debugging process is based on two key ingredients:

- A search space, *i.e.* a set Δ of program parts in which instabilities are to be searched for. In our case, this is a list of functions (or source code lines), which is automatically built by `verrou_dd`.
- A $\texttt{test}$ function, which determines whether introducing Verrou-induced perturbations in a subset of Δ produces instabilities. Most of this testing process is automated, but two user-defined scripts are required in order to specify how to run the analyzed program and how to validate its results. Based on these two user-defined scripts, the automatically-generated $\texttt{test}$ function either validates (✓) a given subset of Δ, or detects instabilities (✗) in it.

As an illustration, at the beginning of the Delta-Debugging process, the $\texttt{test}$ function should fulfill the two following requirements:

- $\texttt{test}(\Delta) = ✗$: perturbing the whole program leads to instabilities (*i.e.* there are problems to look for);
- $\texttt{test}(\{\}) = ✓$: perturbing no part of the program should produce an acceptable result.

Data: Δ_{search}: search space (set of symbols or lines)
Data: g: granularity (default value 2)

$\{\Delta_1, \ldots, \Delta_g\} \leftarrow \text{split}(\Delta_{\text{search}}, g)$;

```
for j ∈ ⟦1, g⟧ do
    if test(Δ_j) = ✗ then
        return DDmin(Δ_j, 2);
    end
end
for j ∈ ⟦1, g⟧ do
    if test(Δ_search − Δ_j) = ✗ then
        return DDmin(Δ_search − Δ_j, g − 1);
    end
end
if g = size(Δ_search) then
    return Δ_search ;
end
return DDmin(Δ_search, min(2g, size(Δ_search))) ;
```

Algorithm 2: Zeller's DDmin

Initial versions of the Delta-Debugging used the DDmax algorithm (*cf.* alg. 1), which returns a 1-maximal stable subset $\Delta_{\max} \subset \Delta$, characterized by:

$$\begin{aligned} &\mathtt{test}(\Delta_{\max}) = \checkmark, \\ &\mathtt{test}(\Delta_{\max} \cup \{\delta\}) = \text{✗}, \quad \forall \delta \in \Delta. \end{aligned}$$

In other words, all program parts (functions or source code lines) in $\Delta_{\max}$ can be perturbed together, without producing invalid results. But as soon as another program part is perturbed, instabilities appear. For practical use, `verrou_dd` returns the set $\Delta - \Delta_{\max}$.

The DDmax algorithm is traditionally described in the literature, and implemented, in terms of its DDmin counterpart (see below). However, `verrou_dd` implements a standalone DDmax algorithm, presented in algorithm 1, which avoids some of the work and is therefore a bit more efficient.

Recent versions of Verrou include extensions to the `verrou_dd` tool, which leverage Zeller's DDmin algorithm (*cf.* alg. 2). This algorithm returns a 1-minimal subset of symbol $\Delta_{\min} \subset \Delta$, characterized by

$$\begin{aligned} &\mathtt{test}(\Delta_{\min}) = \text{✗}, \\ &\mathtt{test}\,(\Delta_{\min} - \{\delta\}) = \checkmark, \quad \forall \delta \in \Delta_{\min}. \end{aligned}$$

In other words, instabilities appear as soon as all parts in $\Delta_{\min}$ are perturbed together; no single part of $\Delta_{\min}$ can be left unperturbed without making instabilities disappear. DDmin is used by `verrou_dd` in the following recursive process (*cf.* alg. 3):

1. search for a minimal subset generating instabilities,
2. remove this subset from the search space,

Data: Δ: search space (set of symbols or lines)

$\Delta_{\text{current}} \leftarrow \Delta$;
$R \leftarrow \{\}$;

while $\texttt{test}(\Delta_{current}) = ✗$ **do**
  $\Delta_{\text{min}} \leftarrow \text{DDmin}(\Delta_{\text{current}})$;
  $R \leftarrow R \cup \{\Delta_{\text{min}}\}$;
  $\Delta_{\text{current}} \leftarrow \Delta_{\text{current}} - \Delta_{\text{min}}$;
end

return R;

Algorithm 3: Recursive DDmin (rDDmin)

3. restart the process if there are still instabilities to find.

This recursive DDmin (rDDmin) algorithm has nice properties. First, rDDmin produces a set of unstable subsets. This is a richer information than DDmax algorithm, which would in principle[4] return the union of all these subsets.

This is illustrated in Table 1, which shows the output of `verrou_dd` using the rDDmin algorithm for code_aster on the same test-case as in section 3.2.2 and Figure 1. In this case, a total of 32 functions (or rather, symbols) are detected as unstable (in the sense of 1-minimality). A DDmax algorithm could have returned the same 32 unstable symbols, but rDDmin results are structured in a finer way. 30 of these symbols are unstable by themselves, but 2 are coupled together, forming rDDmin subset #2: these two symbols produce instabilities only if they are perturbed together.

Second, each DDmin step produces results which are immediately useful; the user can start analyzing them even before the full algorithm terminates. In the example of Table 1, around 10 hours are needed for the whole rDDmin algorithm to terminate its search fro unstable symbols. However, the first unstable symbol (`KSPFGMRESBuildSoln` from PETSc) is found after only 25 minutes, which allows the user to start investigating much sooner.

These results also highlight the role played by third-party libraries and the interaction between them. Unsurprisingly, a significant fraction of the unstable symbols are found in linear algebra libraries such as PETSc or OpenBLAS, for algorithms related to dot products. In these cases, additional information like Call Site Paths (CSP) would be very useful to help determine whether which parts of the program itself (code_aster in this case) should be investigated. Such information is already available with tools like Veritracer [5], and are an important perspective for the development of Verrou.

4.2 Delta-Debugging in stochastic contexts

The algorithms mentioned above were designed for deterministic *test* functions. When the *test* function involves Stochastic Arithmetic, as it is the case with Verrou, successes and failures get non-deterministic. A practical way to deal with this issue consists in performing several Verrou samples within each call to the *test* function. As soon as one Verrou sample fails the user-defined validation criteria, the test is considered to be failing. Conversely, the test succeeds if all Verrou samples meet the user-defined criteria. By increasing the number of samples[5] the *test*

[4] The set of unstable program parts is not unique; it depends on the order in which subsets are tested.
[5] The `VERROU_DD_NRUNS` environment variable allows controlling this parameter.

Table 1: DDmin subsets output by verrou_dd for code_aster on test case ttlv300a with 10 Verrou samples.

rDDmin subset	Symbol name	Library
1	KSPFGMRESBuildSoln	libpetsc.so.3.8.2
2	KSPFGMRESCycle KSPGMRESClassicalGramSchmidtOrthogonalization	libpetsc.so.3.8.2 libpetsc.so.3.8.2
3	MatMult_SeqAIJ	libpetsc.so.3.8.2
4	VecMAXPY_Seq	libpetsc.so.3.8.2
5	VecMDot_Seq	libpetsc.so.3.8.2
6	__smumps_fac_asm_master_m_MOD_smumps_fac_a...	asterd
7	__smumps_fac_front_aux_m_MOD_smumps_fac_mq...	asterd
8	__smumps_fac_front_aux_m_MOD_smumps_fac_mq...	asterd
9	amumpm_	asterd
10	amumpp_	asterd
11	ascopr_	asterd
12	assvec_	asterd
13	daxpy_k_HASWELL	libopenblasp-r0.2.12.so
14	ddot_k_HASWELL	libopenblasp-r0.2.12.so
15	dfdm3d_	asterd
16	dscal_k_HASWELL	libopenblasp-r0.2.12.so
17	elraga_	asterd
18	elrfdf_	asterd
19	elrfvf_	asterd
20	op0024_	asterd
21	sgemm_kernel_HASWELL	libopenblasp-r0.2.12.so
22	smumps_facto_send_arrowheads_	asterd
23	smumps_gather_solution_	asterd
24	smumps_ldlt_asm_niv12_._omp_fn.0	asterd
25	smumps_solve_driver_	asterd
26	smumps_solve_node_._omp_fn.7	asterd
27	strsm_kernel_LN_HASWELL	libopenblasp-r0.2.12.so
28	strsm_kernel_LT_HASWELL	libopenblasp-r0.2.12.so
29	te0051_	asterd
30	te0054_	asterd
31	te0057_	asterd
32	te0061_	asterd

Table 2: Comparison of ddmax and rddmin algorithm for an evaluation of test function with 2,5 and 10 samples

Algorithm (#samples)	Total #tests	Total #runs	First #tests	First #runs	%Success	%Failure
DDmax(2)	17.0	24.2	17.0	24.2	8.6	70.6
rDDmin(2)	18.2	29.4	6.9	10.9	30.6	49.0
DDmax(5)	21.1	50.2	21.1	50.2	72.0	20.0
rDDmin(5)	18.0	49.9	6.5	17.9	86.0	10.0
DDmax(10)	22.3	81.5	22.3	81.5	99.4	0.4
rDDmin(10)	17.7	75.6	6.1	26.4	99.6	0.2

function can *approach* a deterministic behavior. However, this often increases prohibitively the computational cost of the Delta-Debugging.

The numerical experience reported in table 2 aims at evaluating the effectiveness and robustness of the DDmax and rDDmin algorithms with respect to the non-determinism of the `test` function. In this experiment, the Delta-Debugging is performed with a fake Verrou, which simulates a program comprising 10 symbols. 3 of these symbols, randomly distributed in the Δ list, independently produce failures with probability 0.5.

Table 2 reports a comparison of the DDmax and rDDmin algorithms, each tested with 2, 5 and 10 Verrou samples per evaluation of the `test` function. The same experiment is performed 500 times, and averaged results are presented in the table columns:

- **Total #tests** (resp. **#runs**): total number of `test` function calls (resp. number of individual runs of the analyzed program) for the whole DD algorithm;
- **First #tests** (resp. **#runs**): number of `test` function calls (resp. number of individual program runs) needed to obtain the first useful result;
- **%success**: fraction of experiments where the unstable set produced contains the 3 unstable symbols and only these;
- **%fail**: fraction of experiments where false positive are found. Finding a strict subset of the expected result is neither considered a fail nor success, which explains why %fail + %success $\neq$ 100%.

The main result reported table 2 is that rDDmin performs almost always better that DDmax for all metrics. In terms of computational cost, rDDmin needs approximately 3 times less runs of the analyzed program before it produces its first useful result. Although both algorithms perform similarly for a relatively large numbers of samples (10), the results relevance is improved by rDDmin when using a reduced number of samples (2 or 5). And this metric does not take into account the structure of the partition of 1-minimal set.

4.3 Filtering

A last improvement, which is valid for both rDDmin and DDmax, consists in reducing the search space (*i.e.* the Δ set in the DD terminology). In previous versions of Verrou, the search space contained all symbols during the execution of Verrou. In recent versions of Verrou, the generation of the Δ set only considers symbol which contain instrumented FP operations.

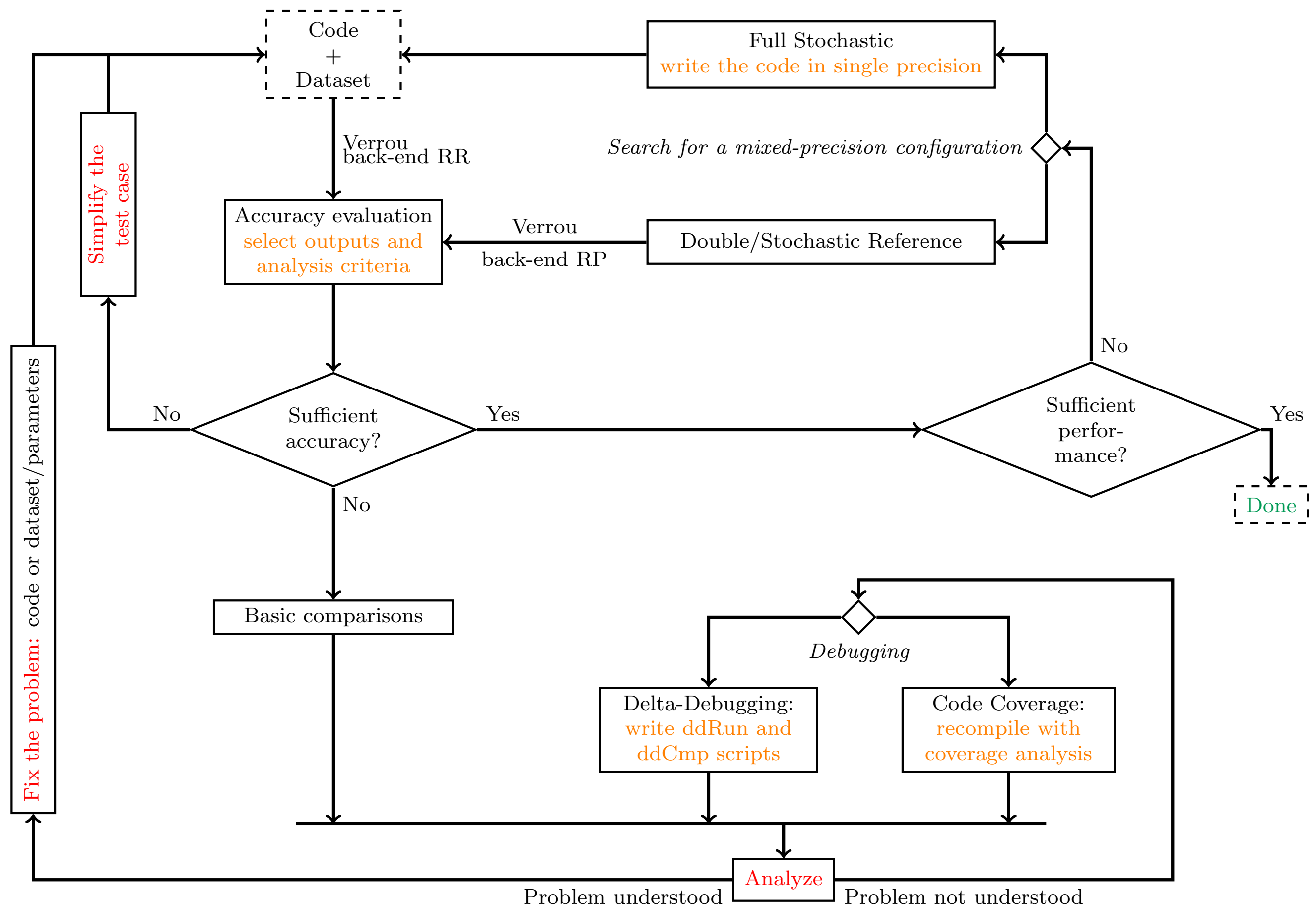

Figure 2: Full process for the analysis and optimization of a scientific computing code with Verrou

For code_aster, with the test case `ttlv300a`, this pre-filtering of the Δ set leads to a drastic reduction of the number of symbols considered by the Delta-Debugging: from an initial list comprising 4459 symbols, the filtered set is reduced to only 154 symbols. With such a small resulting search space, one could wonder whether using a Delta-Debugging algorithm is still an optimal choice.

5 Optimizing mixed precision with Verrou

The debugging techniques presented above can be used to optimize mixed-precision implementations of a given code. Indeed, two complementary workflows can be devised, which help developers decide where high FP precision is needed and, conversely, for which variables a low FP precision might be sufficient.

5.1 Double/Stochastic Reference

The "stochastic reference" methodology used by PROMISE [13] can provide the basis for a first Verrou-based way of finding optimal uses of mixed precision in a given program.

This methodology relies on the assumption that the program produces "valid" results in double precision (*i.e.* results which meet user-defined quality criteria). The validity of these results can either be postulated (in which case double-precision results are taken as reference without further questioning), or verified using Verrou with its Random Rounding back-end (in which case reference results could be obtained by averaging the results produced by all double-precision Verrou samples).

The same test case then can be run again, this time using Verrou with its Reduced Precision back-end (see §3.2.3). A comparison between reduced-precision results and the reference allows determining whether more work needs to be done. If reduced-precision results meet the user-defined quality criteria, then the whole program could benefit from using single precision variables, without affecting results quality too much. Conversely, if reduced-precision results are too different from the reference, then the Delta-Debugging techniques described above can be used to determine which parts of the program have to keep using double-precision variables. Everything else should be considered for the introduction of single-precision variables.

5.2 Full Stochastic

A second optimization methodology can be devised, which is closer in spirit to the so-called "Full Stochastic" methodology of the PROMISE tool.

In contrast to the method presented above, this methodology is based on a single-precision version of the analyzed program. As such, it is probably more adapted to newer scientific computing codes, where the possibility to (uniformly) change the FP precision is often built-in (for example *via* `typedefs` or C++ templates).

This time, a direct application of the Verrou Delta-Debugging techniques exhibits the program parts which have to be switched back to double precision (or redesigned with more accurate algorithms).

5.3 Overview of a full debugging and optimization process

Figure 2 presents a full Verrou-based workflow which can be used to analyze, debug and optimize industrial HPC codes in mixed precision. Actions printed in orange in the figure correspond to setup work, which needs to be done once in order to take into account the specifics of the analyzed program. Actions printed in red correspond to manual steps which have to be performed by the user.

The process starts with a program to analyze and the associated dataset. The user may then proceed to the standard diagnostics procedure, based on the Random Rounding Verrou backend. At this stage, the only work required consists in extracting the results of interest for the code output, and decide on relevant analysis criteria. Usually, for industrial codes with an emphasis on Quality Assurance, much of this preliminary work has already been performed, for example in order to setup non-regression testing.

At the end of the diagnostics, the user should have enough insight to determine whether code results are sufficiently accurate. If not, a first good option might be to try and find another smaller unstable test case, which will simplify and speed-up the debugging process.

Once a simple enough test case has been identified, the user may proceed to a debugging stage. Experience shows that a simple analysis and comparison of the program output between Verrou samples sometimes allows to quickly find the origin of errors. This comparison might

consider not only the results themselves, but also the full log of the computation (number of iterations...) and the number of FP operations counted and output by Verrou. If such a simple analysis is not fruitful, the user may use more complex debugging methodologies, relying either on code coverage comparisons in order to find unstable tests, or on Delta Debugging in order to find unstable functions or source code lines. In both cases, some work is required in order to setup these debugging processes.

After the debugging stage, the user should be able to try and implement fixes in the program (whether by changing the code itself, its parameters, or the test-case).

Once all FP-related issues have been fixed and the program produces sufficiently accurate results, the optimization stage may begin. Depending on the program being analyzed, the user may choose between the "Full Stochastic" and "Double/Stochastic Reference" methodologies described above.

With the "Full Stochastic" methodology, some work needs to be done in order to reduce the FP precision in the program (or in a part of it). The same diagnostics & debugging process can then be started over with this reduced-precision code. However, the debugging process should be easier since a natural fix consists in increasing the precision for all detected unstable code parts. At the end of this process, the developer should have an optimized mixed-precision version of the program, which meets the accuracy criteria on the given test case.

The "Double/Stochastic Reference" methodology might be more adapted to legacy code, where it is often difficult – if possible at all – to change the FP precision. Indeed, no additional work is needed for this optimization methodology. The diagnostics process just needs to be started over, this time using the Reduced Precision Verrou backend. Unstable parts of the program that are detected this way are parts of the code which have to remain in double precision. At the end of this process, developers should have enough insight to guide the development of an optimized mixed-precision version of the program.

6 Conclusions

Verrou is an open-source tool which aims at helping development teams during the whole Floating-Point analysis process, from diagnosing FP-related issues, to debugging them, and finally optimize the code for mixed-precision. Verrou specifically targets large, industrial, high-performance codes, and is freely available under an open-source licence at

`http://github.com/edf-hpc/verrou`

In this paper, we presented various new features which have been recently been implemented in Verrou and released in version 2.1.0. These features extend the original capabilities of Verrou, and target more specifically the HPC needs.

The diagnostics stage of the Verrou analysis mainly relies on Random Rounding. This feature has been adapted and extended correctly handle mixed precision codes, which nowadays represent a fair share of the HPC codes.

As for the debugging stage of the analysis, the Verrou ecosystem provide a dedicated `verrou_dd` utility which automates most of the search for unstable code parts. This utility, which relies on the Delta-Debugging family of algorithms by Zeller, has been extensively reworked. It now includes a new Recursive DDmin (rDDmin) algorithm, which is more robust and provides faster and more accurate results than the original DDmax algorithm. This is especially advantageous in the case of FP-related issues which appear infrequently and are thus hard to reproduce. When

searching for unstable parts of the source code (functions/symbols or source code lines), a pre-filtering feature also allows dramatically reducing the size of the search space by considering only functions (or source code lines) which actually perform FP operations.

In terms of the mixed-precision optimization of HPC codes, Verrou now includes a new backend which emulates the use of single precision in the program (or any part of it). In combination with the debugging features of Verrou, this can be used to identify which parts of the source code are sensitive to FP issues and should remain in double precision.

The interest of these features was illustrated throughout the paper by examples taken from the analysis of code_aster, an industrial structural mechanics simulation tool. As an example, the Delta-Debugging process now produces its first useful results after only 25 minutes, which allows developers to start analyzing and debugging the code much sooner.

These new features make it possible to use Verrou not only as a diagnostic tool, but also as an help and guide during the development.

References

[1] Erika Ábrahám. Techniques and tools for hybrid systems reachability analysis. In *Rigorous Systems Engineering of Cyber Physical Systems, RISE4CPS 2017*, Heidelberg, Germany, July 2017.

[2] Code_Aster: Structures and thermomechanics analysis for studies and research. http://www.code-aster.org/.

[3] Stanley Bak and Parasara Sridhar Duggirala. Simulation-equivalent reachability of large linear systems with inputs. In *Proceedings of the 29th International Conference on Computer Aided Verification*. Springer, 2017.

[4] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. A dynamic program analysis to find floating-point accuracy problems. In *33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 453–462, New York, NY, USA, June 2012. ACM.

[5] Yohan Chatelain, Pablo de Oliveira Castro, Eric Petit, David Defour, Jordan Bieder, and Marc Torrent. VeriTracer: Context-enriched tracer for floating-point arithmetic analysis. In *IEEE Symposium on Computer Arithmetic (ARITH)*, Amherst, MA, USA, June 2018.

[6] A. Dawson and P. D. Düben. rpe v5: an emulator for reduced floating-point precision in large numerical simulations. *Geoscientific Model Development*, 10(6):2221–2230, 2017.

[7] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Transactions on Computers*, 60(2):242–253, 2011.

[8] Christophe Denis, Pablo de Oliveira Castro, and Eric Petit. Verificarlo: Checking Floating Point Accuracy through Monte Carlo Arithmetic. In *23nd IEEE Symposium on Computer Arithmetic, ARITH 2016, Silicon Valley, CA, USA, July 10-13, 2016*, page 55–62, 2016.

[9] François Févotte and Bruno Lathuilière. VERROU: Assessing Floating-Point Accuracy Without Recompiling. October 2016.

[10] Michael Frechtling and Philip H.W. Leong. Mcalib: Measuring sensitivity to rounding error with monte carlo programming. *ACM Transactions on Programming Languages and Systems*, 37(2):5, 2015.

[11] François Févotte and Bruno Lathuilière. Studying the Numerical Quality of an Industrial Computing Code: A Case Study on code_aster. In *10th International Workshop on Numerical Software Verification (NSV)*, page 61–80, Heidelberg, Germany, July 2017.

[12] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1), March 1991.

[13] Stef Graillat, Fabienne Jézéquel, Romain Picot, François Févotte, and Bruno Lathuilière. Auto-tuning for floating-point precision with discrete stochastic arithmetic. working paper or preprint, June 2016.

[14] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[15] Maxime Jacquemin, Sylvie Putot, and Franck Védrine. A reduced product of absolute and relative error bounds for floating-point analysis. In Andreas Podelski, editor, *Static Analysis*, pages 223–242, Cham, 2018. Springer International Publishing.

[16] Fabienne Jézéquel, Jean-Marie Chesneaux, and Jean-Luc Lamotte. A new version of the CADNA library for estimating round-off error propagation in Fortran programs. *Computer Physics Communications*, 181(11):1927–1928, 2010.

[17] William Kahan. How futile are mindless assessments of roundoff in floating-point computation? Technical report, 2006.

[18] Michael O Lam and Jeffrey K Hollingsworth. Fine-grained floating-point precision analysis. *The International Journal of High Performance Computing Applications*, 32(2):231–245, 2016.

[19] Jean-Luc Lamotte, Jean-Marie Chesneaux, and Fabienne Jézéquel. CADNA_C: A version of CADNA for use with C or C++ programs. *Computer Physics Communications*, 181(11):1925–1926, 2010.

[20] Sethy Montan. *Sur la validation numérique des codes de calcul industriels*. PhD thesis, Université Pierre et Marie Curie (Paris 6), France, 2013. in French.

[21] Ralph Nathan, Bryan Anthonio, Shih-Lien Lu, Helia Naeimi, Daniel J. Sorin, and Xiaobai Sun. Recycled Error Bits: Energy-Efficient Architectural Support for Floating Point Accuracy. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*, pages 117–127. IEEE Press, nov 2014.

[22] Laurent Plagne and Kavoos Bojnourdi. Portable vectorization and parallelization of C++ multi-dimensional array computations. In *ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY)*, pages 33–39, 2017.

[23] Nathalie Revol. *Introduction to the IEEE 1788-2015 Standard for Interval Arithmetic*, pages 14–21. Springer International Publishing, Cham, 2017.

[24] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious: Tuning assistant for floating-point precision. In *Proc. of SC13*, 2013.

[25] Devan Sohier, Pablo De Oliveira Castro, François Févotte, Bruno Lathuilière, Eric Petit, and Olivier Jamond. Confidence Intervals for Stochastic Arithmetic. Preprint.

[26] D. Stott Parker. Monte Carlo arithmetic: exploiting randomness in floating-point arithmetic. Technical Report CSD-970002, University of California, Los Angeles, 1997.

[27] Jean Vignes. A stochastic arithmetic for reliable scientific computation. *Mathematics and Computers in Simulation*, 35:233–261, 1993.

[28] Andreas Zeller. *Why Programs Fail.* Morgan Kaufmann, Boston, second edition, 2009.

[29] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.