VeriTracer: Context-enriched tracer for floating-point arithmetic analysis

Yohan Chatelain*¶, Pablo de Oliveira Castro*¶, Eric Petit†¶, David Defour‡,
Jordan Bieder§¶, Marc Torrent§¶
* *University of Versailles, {yohan.chatelain, pablo.oliveira}@uvsq.fr*
† *Intel, eric.petit@intel.com* ‡ *University of Perpignan, david.defour@univ-perp.fr*
§ *CEA, DAM, DIF, F-91297 Arpajon, France, {jordan.bieder, marc.torrent}@cea.fr*
¶ *Exascale Computing Research, France*

Abstract

VeriTracer automatically instruments a code and traces the accuracy of floating-point variables over time. VeriTracer enriches the visual traces with contextual information such as the call site path in which a value was modified. Contextual information is important to understand how the floating-point errors propagate in complex codes. VeriTracer is implemented as an LLVM compiler tool on top of Verificarlo. We demonstrate how VeriTracer can detect accuracy loss and quantify the impact of using a compensated algorithm on ABINIT, an industrial HPC application for Ab Initio quantum computation.

1. Introduction

The recent evolution of HPC system – massive parallelism, large SIMD vectors, asynchronicity, complex memory hierarchy – and the constant growth in computational power allow for higher resolution simulations of complex physical phenomena. Because HPC optimizations change the order of the operations, they can lead to numerical accuracy bugs.

The need for tools and documented best practices to address numerical bugs such as inherent instability of the simulated phenomena, reproducibility of results across architectures, compilers, and languages, are becoming of prime importance.

The floating-point arithmetic (FPA) models real numbers with finite precision. Consequently, the result of an FPA operation may be truncated to fit in a finite number of digits. The accumulation of round-off errors, absorption, or cancellations along a computational flow may lead to accuracy and precision loss [1].

In order to understand and prevent such numerical bugs raised during a run of a given implementation, it is important to propose numerical debugging tools to assist users. In this paper we tackle this problem through the following contributions:

- VeriTracer, a visualization tool that brings temporal dimension to a graphical Floating-Point analysis.
- A methodology to reduce the search space for floating-point bugs by detecting critical functions according to a user-defined numerical criterion.
- The use of source location and Information Flow Analysis to enrich and help the analysis of the precision traces.
- A co-design study to detect and improve loss of accuracy on ABINIT [2], an academic and industrial physics simulation code.

2. Motivating example

Evaluating the numerical reliability of a given results computed after a long run of a program involving numerous floating-point operations is most of the time limited to the observation of the final results, which might not be sufficient, as shown in the next example.

Let consider the following series proposed by J.-M. Muller [3] :

$$u_{n+1} = 111 - \frac{1130}{u_n} + \frac{3000}{u_n u_{n-1}} \tag{1}$$

The fixed-points of this series are roots of the polynomial: $u^3 - 111u^2 + 1130u - 3000 = (u - 5)(u - 6)(u - 100)$. Given the selected starting point $u_0 = 2, u_1 = -4$ the mathematical limit of this series is the root 6. Nevertheless, when computed with single or double FPA, it converges toward the dominant root 100. Furthermore, as shown by Parker [4, p. 67], if one is focusing on the final result, one can conclude that the computation is fully precise and correct, which is not the case.

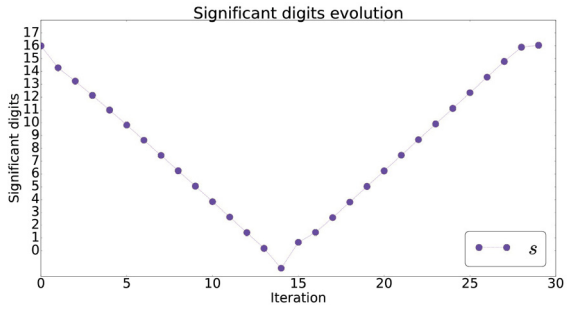

Figure 1: Evolution of the number of significant decimal digits (s) over time for the sequence u_n in equation 1. For $n = 14$, s below 0 means that u_{14} has no correct decimal digits. Only checking the final results is not enough to detect the accuracy loss.

Figure 1 plots the evolution of the number of significant digits over time. One can observe, that the numerical precision slowly degrades until it reaches zero significant digits and then increases until it reaches sixteen, the maximum number of significant digits in double precision. The final result has been generated from an intermediate result with no significance alerting the developer about the correctness of the final result. All accuracy has been lost, and the series converge precisely to 100 instead of 6.

The goal of VeriTracer is to display the numerical history as a temporal story, thus helping understand the numerical behavior of programs. It allows focusing on a program's region of interest, in a given execution context, and visualize its numerical behavior. Figure 1 was automatically generated with VeriTracer.

3. Background

Floating-point arithmetic (FPA) refers to an approximation of real numbers and operations on them. FPA is defined by the IEEE-754 standard [5]. The discretization of real numbers by FP leads to the so-called error of representation. For example, the decimal number 0.1 has a finite representation in base 10 while it has an infinite representation in base 2. Also, the result of an operation on FP may have more digits than available in the format, leading to round-off errors. Finally, when subtracting two numbers that are very close, part of the mantissa is canceled which reduces the number of significant bits in the result. Goldberg [1] reviews the floating point representation and its dangers.

Stochastic Arithmetic proposes automatic methods to estimate the number of significant digits in a result. Numerical errors are approximated by introducing random perturbations at each FP operation. By repeating this process many times and studying the spread of the output results one can stochastically approximate the significance of a computation. Two main methods have been proposed: CESTAC [6] and Monte Carlo Arithmetic (MCA) [4].

CADNA [7] is a library implementing CESTAC. It provides special C and Fortran types to simulate floating-point inaccuracies by randomly changing the IEEE rounding mode. Verrou [8] is a Valgrind tool that automatically replaces all the floating-point operations by CESTAC operations. A recent extension [9] of CADNA leverages Delta-Debug [10] to pinpoint the parts of a source code that can be rewritten using single precision. Verificarlo [11] is a compiler based on LLVM which replaces the floating-point operations by their MCA counterparts. FpDebug [12] uses shadow memory for detecting accuracy problems with Valgrind by computing in higher precision. Similary, Herbgrind [13] is a tool based on Valgrind that can automatically localize floating-points errors and find the causes of inaccuracies by tracking operations dependencies. Craft HPC [14] instruments codes for detecting catastrophic cancellations at runtime. Although efficient, these tools study the error locally. Measuring the accuracy loss for a given variable does not provide information about its story: how the numerical quality of a variable evolves during its life is potentially critical information as section 2 revealed.

Collecting and tracing numerical accuracy information over time is a novel contribution of this paper. Nevertheless, collecting, processing and tracing performance metrics is a well studied problem in the performance characterization research field [15], [16], [17], [18], [19]. Performance characterization must analyze an application to target potential execution overhead. Such tools can display a significant amount of information over time for large multiprocess applications. We aim at bringing solutions to extract and visualize complex information over time from the performance characterization community to the numerical analysis community.

3.1. Monte Carlo Arithmetic

Monte Carlo Arithmetic (MCA) as proposed by Parker [4] corresponds to a stochastic arithmetic that extends the IEEE-754 Floating-Point Arithmetic. MCA models the round-off errors of the FPA by introducing a random noise in each floating-point operation. By sampling a large number of MCA executions of a program, one can simulate the error distribution for a given result and estimate the number of significant digits. Replacing floating-point expression by their

MCA counterparts by hand is a tedious task for real codes. Verificarlo [11] automates this process.

Unlike other stochastic arithmetic, the advantage of MCA is that the magnitude of the error introduced is configurable. Therefore it can simulate the effects of running on lower precision hardware. The precision at which MCA operates is called the *virtual precision*. For a given value x, the noise is modeled as follow:

$$inexact(x) = x + 2^{e_x - t}\xi$$

where e_x is the order of magnitude of x, t the virtual precision and ξ a random variable between $[-\frac{1}{2}, \frac{1}{2}]$.

To ensure that MCA approximation behaves as FPA on average, the probability distribution of ξ must be unbiased and its expected average equal to 0 [4, p. 33]. In the following, we consider ξ to be uniform. We define $\odot$ as the FPA approximation of an arithmetic operation. MCA provides three error modes.

Random Rounding (RR): $x \odot_{RR} y = inexact(x \odot y)$, models round-off error by introducing an error after the operation. From a numerical analysis point of view, it can be seen as a forward error analysis. RR mode preserves exact operations [4, p.33,§6.4] when used with $t = 53$ for double values ($t = 24$ for single).

Precision Bounding (PB): $x \odot_{PB} y = inexact(x) \odot inexact(y)$, models cancellation by disturbing both inputs. It models the backward error.

Full MCA: $x \odot_{MCA} y = inexact(inexact(x) \odot inexact(y))$ is the composition of the previous modes.

By simulating round-off and cancellation errors in a large number of samples, MCA is able to estimate the numerical quality of a computation. The number of significant digits can be estimated by computing the magnitude of the relative error in the resulting sample,

$$s = -\log_\beta \left| \frac{\mu}{\sigma} \right|$$

where σ is the standard deviation and μ the expected value for a large enough number of sampling and β is the base. One can refer to [4] for a complete discussion of the different modes and their link with round-off errors and a proof that s is a good approximation of the number of significant digits of the result.

4. VeriTracer: Principles of design

In this section, we address the design of VeriTracer, a tool built upon Verificarlo[11]. This tool aims at providing a temporal dimension to numerical analyses of real-world programs.

To achieve this goal, VeriTracer automatically instruments a code to replace each floating-point operation with its MCA counterpart. Figure 2 presents the

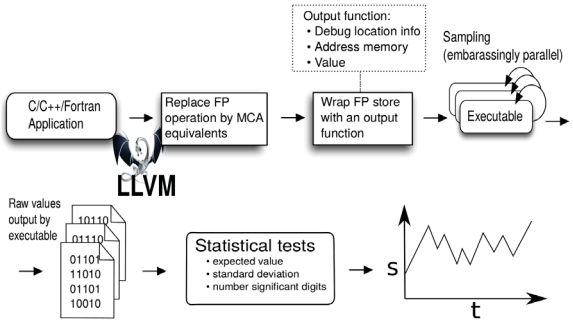

Figure 2: VeriTracer workflow.

workflow of VeriTracer built upon Verificarlo, a tool based on LLVM Compiler [20]. First, VeriTracer takes source code supported by LLVM front-end as input files such as C, C++ or Fortran. Second, VeriTracer replaces every FP operation by its MCA equivalent and inserts tracing probes to gather error and context information. Third, an executable is generated and is run multiples times, such that each sample produces a binary trace. Fourth, traces are merged and a visual representation enriched with context information is produced. One can notice that by operating at the back-end level, the optimizations made by the compiler are preserved and instrumented by VeriTracer.

4.1. Instrumentations

A program manipulates numerical values through variables or constants in various formats on which operations are applied. The instrumentation platform must gather numerical information such as accuracy on any variables at every step during its lifetime and associate this information with its context.

VeriTracer relies on the Verificarlo framework which replaces every FP operations by its MCA counterparts. To gather the tracing and context information, VeriTracer inserts tracing probes after each FP operation. During each execution of the program, data produced by the VeriTracer's probes are saved in binary files, with one file per MCA run.

Instrumenting a program is possible either at compile time as we do in VeriTracer with LLVM or dynamically with a framework such as Valgrind or Pin. The compile time instrumentation has less overhead at runtime since no dynamic patching is necessary. For example, the authors of AddressSanitizer [21] report a smaller slowdown when using LLVM for instrumentation compared to higher overhead with dynamic instrumentation tools such as Valgrind.

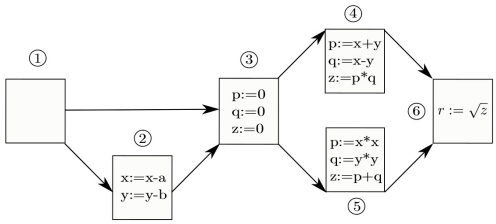

Figure 3: Control-flow analysis

4.2. Information flow analysis

In this article we consider an asynchronous analysis based on a Monte Carlo process. Each run generates a set of data, and many runs are collected. However, since the analysis is asynchronous, we must ensure that variables come from the same control-flow path to build meaningful statistic. Since each execution is slightly different from the others due to the noise added by MCA, they may have divergent flow paths. Moreover, from a temporal point of view, two measures for a given variable may not be correlated if they do not share the same history. This history is related to the execution flow path. To tackle this issue, we need to compare the *information flow* of variables. This is called *information flow analysis*. This analysis can be done at three different granularities: context, control-flow and data-flow. In the following, we outline our design for the three information flow analysis in VeriTracer, currently only context analysis is implemented.

4.2.1. Context analysis. Context-analysis considers the context of a function call where the probe is located. A function can be called by different parent functions which can be called themselves by different functions. The list of the caller ancestors is the *call site path* (CSP). Depending on its CSP, a function may have different behaviors. Not identifying them can lead to wrong interpretations since successive values may not be correlated, from a temporal point of view.

4.2.2. Control-flow analysis. The control-flow analysis considers the path taken within a function where the probe is located. Figure 3 illustrates this problem. On this example, at stage 1, the program can take the branch 2 or not. Among different executions, the execution trace will not exhibit the same number of variables traced, whether they take this branch or not. Another issue with Control-flow analysis arises when considering basic blocks 4 and 5. Both blocks use the same variables but in different sequences of operation.

An efficient solution to control-flow analysis consists in dynamically computing a hash value associated with a path as it is proposed in [22]. The hash sum captures the branches path taken and can differentiate execution paths. It tags each branch with a unique identifier and builds the hash value associated with a path. It applies a hash function between the previous and the current branch visited. The hash function is chosen to minimize collisions, with a small overhead.

4.2.3. Data-flow analysis. The data-flow analysis considers the dependencies between variables across a program. As the errors introduced by MCA can be propagated through calculations, it is interesting to keep track of these dependencies. This class of analysis allows us to observe the interactions between variables, and have information on errors propagation.

Taint analysis [23] checks which computations are affected by a set of variables *tainted* by the user. A *taint policy* defines the propagation rules for tainting. *Tainted* variables would be, in our case, the variables disturbed by MCA. The *taint* flow follows variables affected by the *tainted* variables to construct the *contamination chain* which represents the variables affected by the error introduced.

The forward symbolic execution [23] represents the execution of a program with a logical formula. With a logical formula, it is possible to apply mathematical logic to the data-flow analysis and reason on values domain providing strong proofs. However, the number of possibilities to explore is exponential.

4.3. Pinpointing relevant functions

Introducing MCA errors and tracing each function of a large code is an expensive process. Moreover, disturbing the computation of a function may impact the computation of others functions. We should analyze functions by groups, however, this leads to a combinatorial explosion. To reduce the search space, we propose a two steps heuristic. First, functions are separated according to their numerical impact on the precision based on a criterion given by the user, which can be a reference value computed by the program. Second, functions are classified according to their impact on the selected criterion.

4.3.1. Search space reduction. The solution we propose consists in instrumenting one function of the program at a time. We execute each function with the maximal MCA error that is to say using a virtual precision of 1 bit. If the reference value computed with MCA is different from the IEEE result, the function is tagged as critical and not critical otherwise.

By focusing on executions with a single modified function, our process does not capture coupling effects which would require more costly and sophisticated techniques such as ANOVA multivariate analysis. Nevertheless, it provides a reasonable approximation of the *set of critical functions* that impact the criterion, therefore reducing the search space.

4.3.2. Search space ordering. The identification of interesting functions is done by ordering the *set of critical functions* using *numerical stress resistance*. Numerical stress resistance consists in looking for the minimal virtual precision (t_{min}) such that the criterion is identical to the one computed without MCA. t_{min} is identified in logarithmic time by dichotomy on the level of error introduced during MCA.

Once, t_{min} is gathered for every function, the *set of critical functions* is ordered according to the t_{min} value. Functions with a small (large) t_{min} are considered to have a minimal (maximal) impact on the numerical error of the criterion. In the current stage of development of VeriTracer, this ordered set is displayed on a graph for manual inspection and selection of functions of interest.

4.4. LLVM instrumentation

Instrumentation is done at Intermediate Representation (IR) [20] level. The LLVM IR is a virtual assembly code which abstracts hardware through virtual registers. In this format, there is no strict mapping between virtual registers and physical registers. LLVM transfers values between the memory and virtual registers with the `load` and `store` instructions. The memory is divided into three parts: the stack, the heap, and the global area. The `alloca` instruction allocates memory on the stack. LLVM uses Static Single Assignment (SSA) form to manage virtual registers. This form defines each variable once. An instruction that produces a value, like the `add` instruction, implicitly creates a new virtual register to represent the resulting value.

Without optimizations (`-O0`), LLVM maps each variable to an address in memory. Each memory access to a variable is translated as a `load` or `store` instruction. It is therefore straightforward to track FP value changes since they directly map to `store` instructions. VeriTracer identifies all FP `store` instructions and inserts, immediately after each of them, a `call` instruction to a probe. Figure 5 shows the LLVM IR of the TwoSum algorithm [25] (figure 4) compiled without optimizations (`-O0`).

However, production codes are most of the time compiled with optimizations, which are grouped into optimizations sets such as `-O2` or `-O3`. Those optimizations sets include the `mem2reg` pass. At the IR level, this optimization promotes memory references to register references. In those cases, VeriTracer may miss variables located in virtual registers since `store` instructions never use them as shown figure 6. To overcome this problem, VeriTracer checks every operation on FP numbers, and if the LLVM value is linked to a named variable, it inserts a probe.

LLVM maintains debug information about variables during the compilation process called `metadata`. Metadata are available when the debug mode is enabled (with `-g` flag). VeriTracer accesses the metadata through functions provided by the LLVM API to retrieve the name, the memory address, the calling function and the source line of operation.

VeriTracer handles scalar and derived types such as pointers, arrays, structures and any complex compound types. LLVM uses the `getelementptr` instruction to compute the memory address of a sub-element. As a sub-element can be arbitrarily complex, multiples indexes are required. For pointers, it returns the variable targeted, for structures, it returns a pointer to a given field, and for arrays, it returns a pointer to the element at a given index. VeriTracer parses the intermediate pointer chain recursively until a root variable named in the source code is found.

5. ABINIT: industrial testcase

ABINIT [2] is a program to calculate, from the quantum equations of density functional theory (DFT), the optical, mechanical, vibrational, and others observable properties of materials. ABINIT deduces these properties from the properties of electrons whose number is the dimensioning variable.

It works on any chemical composition ranging from molecules to nanostructures or solids and is widely used by industrial and academics. ABINIT is a Fortran program made of 1307 files, 5835 functions, for 851615 lines of code.

In this use case, ABINIT computes the total-energy E_{total} of a system composed of unit cells of 5 atoms ($BaTiO_3$). This structure is a variant of a type of crystal structure called Perovskite. ABINIT is deterministic: the IEEE result is the same between two executions with the same inputs.

5.1. Selection of functions of interest

In a first step, we use E_{total} accuracy as the criterion for the filtering described in section 4.3.1. Table 1 shows that, for that use-case and solver setting, among

```
void twoSum(double a,double b,
 double *x_ptr,double *e_ptr)
{ double x = a + b;
  double z = x - a;
  double e = (a -(x-z))+(b-z);
  *x_ptr = x; *e_ptr = e; }
```

Figure 4: TwoSum [24] in C

```
define void @twoSum(double %a,double
 %b,double* %x_ptr,double* %e_ptr)
{%5 = load double* %a
 %6 = load double* %b
 %7 = fadd double %5, %6    ; a+b
 store double %7, double* %x ; x=a+b
 ... }
```

Figure 5: IR with -O0

```
define void @twoSum(double %a,double
 %b,double* %x_ptr,double* %e_ptr)
{
  %1 = fadd double %a, %b ; a+b (x)
  ...
}
```

Figure 6: IR with -O3 (after reg2mem)

LLVM IR of the TwoSum function (fig. 4) with (fig. 6) and without (fig. 5) compiler optimizations. Without the `reg2mem` pass, LLVM loads operands from memory and stores the result in memory for each FP operations. With optimizations, intermediate computations transit through registers instead of memory, as computation of x shows.

	#Functions	#FP operations
Functions visited	30410	308704
Functions with FPA	2952	308704
Critical functions	88	19235
`simp_gen`	1	41

Table 1: ABINIT code statistics. Filtering reduces the search space by 33.

Instrumentation	Time (s)	Overhead
Original	201	1
Probes	208	1.03
MCA	236	1.17
VeriTracer (Probes + MCA)	238	1.18

Table 2: Instrumentation overhead on the Occigen cluster when producing the traces in figure 8. VeriTracer combines two instrumentations: tracing *Probes*, and replacing FPA operations replaced by *MCA* operations.

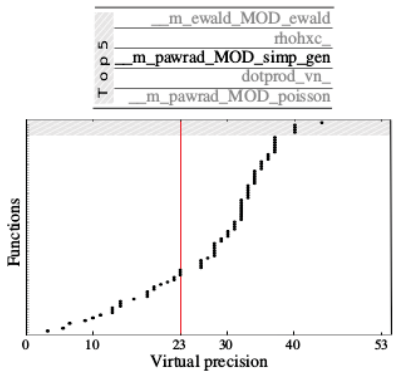

Figure 7: Numerical sensitive analysis on the Perovskite use case. For each function on the y-axis, we plot the minimal virtual precision required to reach machine precision accuracy. Table on the top shows the 5 most sensitive functions of the hatched area.

the 2952 functions which contain FP operations, only 88 critical functions affect E_{total} when noise is added. This filtering step reduces by 33 the search space.

The second step consists of performing a numerical sensitivity analysis sorting functions from the *critical functions set*. Figure 7 computes for each function of the *critical functions set* the minimal virtual precision required to reach machine accuracy on E_{total}. In other words, the virtual precision is the number of bits required to compute the same E_{total} than in IEEE mode. Functions close to zero have a low impact on E_{total}. On the other hand, functions that are close to 53 do not tolerate any loss of precision.

Filtering requires 2952 runs for the first step (the number of functions) and at most 616 runs for the second dichotomy step (88 functions $\times \log_2 53$).

5.2. Tracing analysis of numerical loss

Table in figure 7 corresponds to the five most sensitive functions for the selected criterion. In the typical VeriTracer workflow, each of these sensitive functions would be analyzed in detail. However, for the sake of brevity in the sequel of this article, we will only focus on `simp_gen` which computes an integral by Simpsons' rule over a generalized 1D-grid.

The main loop of `simp_gen` is at its core a dot product between a function to integrate and an input vector. `Simp_gen` is called many times in ABINIT and appears in 31 different call-site paths (CSP).

VeriTracer uses Random Rounding with 53 bits of virtual precision to simulate IEEE round-off error in double precision. We compiled ABINIT with VeriTracer and ran 24 MCA traces. MCA traces are independent and can be executed concurrently. In this case, the 24 runs were launched in parallel on the CINES Occigen cluster. Occigen has 2106 nodes with 2 processors Intel 12-Cores (E5-2690V3@2.6 GHz) by nodes. Each node has 64 Go or 128 Go of RAM.

After post-processing, VeriTracer produces figure 8, which represents the number of significant digits for each call to `simp_gen`. Each point has a color that depends on its CSP. Many CSPs only correspond to

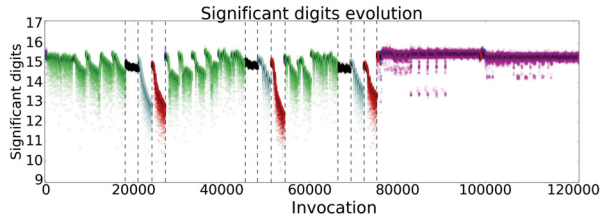

Figure 8: Original

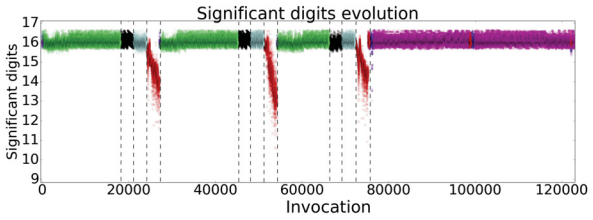

Figure 9: Compensated

Figures show profiles produced by VeriTracer of `simp_gen`. Several accuracy losses present in the original version (fig. 8) are improved by the compensated algorithm Dot2 (fig. 9). Each color maps one of the 31 distinct CSPs. Dot2 improved 30 out of 31 CSPs.Vertical dashes separate the main CSPs.

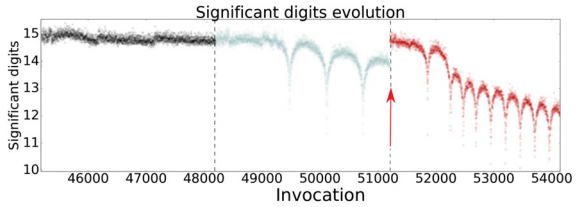

Figure 10: Without CSP, one concludes that the accuracy improves at the marked invocation. With CSP, one knows that the traces belong to different computations.

a single call to `simp_gen` and are not easily identifiable on the figure. Figure 10 is a zoom of figure 8 which illustrates that CSPs are crucial for analyzing dependencies: without CSPs context information, one would misinterpret results and conclude that precision is enhanced after invocation 51000, while in fact, those computations are independent. Furthermore, arches within the same CSP are due to different inputs: when integral results get close to zero, the accuracy lowers.

Among the four main CSPs, we observe several downward spikes that correspond to accuracy loss in the Simpsons' integral computation. Up to six digits of precision are lost in some of the calls.

Table 2 shows the implementation performance of VeriTracer when producing the traces in figure 8. Each VeriTracer execution is on average 1.18× slower than the original execution (only the `simp_gen` function is instrumented).

5.3. Effect of a compensated dot product

In the previous section, VeriTracer identified several instances of accuracy loss during the calls to `simp_gen`, one of the top five sensitive functions.

Ogita [24] introduces the *dot2* algorithm that uses an error correcting term. Since `simp_gen` can be modeled as a large dot product computation, it was rewritten with `libeft` *dot2* implementation [26]. We use VeriTracer to measure the impact in ABINIT and check how much it improves the accuracy losses.

Compensated algorithms work by using exact cancellations to compute exact error terms. In PB mode, each exact cancellation must be protected with calls to VeriTracer to temporarily disable MCA noise. Nevertheless, here RR $t = 53$ mode does not break the exact operations in `libeft` as explained in section 3.1.

After running 24 times the `simp_gen` compensated version of ABINIT, VeriTracer produces figure 9. In 30 out of the 31 CSPs, the compensated algorithm fully fixed the precision loss. Interestingly, one of the CSPs (in red color) was not fixed by *dot2*. The failing CSP computation produces multiple cancellations, but our initial analysis shows that the cancellations are small enough (11 bits are canceled) to be compensated. A dependency analysis of the code shows that `simp_gen`'s inputs in the failing CSP are themselves produced by upstream calls to `simp_gen`. The precision loss seems to be tied to the complex dependencies between the multiple calls and require further study.

In this section, we showed how VeriTracer could measure the impact of a numerical optimization such as using a compensated algorithm. We were able to entirely fix 30 out of 31 CSPs in the Simpson's integral computation in ABINIT.

6. Conclusion

VeriTracer is a tool to visualize the numerical behavior and quality of variables over time automatically. It is built upon Verificarlo, a compiler tool which implements Monte Carlo Arithmetic. In this article, we have demonstrated, on an industrial physics simulator, how VeriTracer helps in the detection of numerical problems, in the co-design by pinpointing parts of the code where instabilities arise and in visualizing the impact of numerical corrections.

This paper exposes several methodologies to automate the information flow analysis. Compilation metadata enriches the floating-point instrumentation and context-sensitive analysis disambiguates the analysis.

VeriTracer is available at **http://www.github.com/verificarlo/tree/veritracer** under the GPL license.

Acknowledgements We thank Exascale Computing Research Lab supported by CEA, Intel, and UVSQ. This work has been granted access to the HPC resources of CINES under the allocation 20XX-A0031010295 made by GENCI.

References

[1] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

[2] X. Gonze, F. Jollet, *et al.*, "Recent developments in the ABINIT software package," *Computer Physics Communications*, vol. 205, pp. 106–131, 2016.

[3] J.-C. Bajard, D. Michelucci, J.-M. Moreau, and J.-M. Muller, "Introduction to the Special Issue "Real Numbers and Computers"," in *The Journal of Universal Computer Science*, pp. 436–438, Springer, 1996.

[4] D. S. Parker, *Monte Carlo Arithmetic: exploiting randomness in floating-point arithmetic*. University of California. Computer Science Department, 1997.

[5] D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo, *et al.*, "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[6] J. Vignes, "Discrete Stochastic Arithmetic for Validating Results of Numerical Software," *Numerical Algorithms*, vol. 37, no. 1-4, pp. 377–390, 2004.

[7] F. Jézéquel and J.-M. Chesneaux, "CADNA: a library for estimating round-off error propagation," *Computer Physics Communications*, vol. 178, no. 12, 2008.

[8] F. Févotte and B. Lathuiliere, "VERROU: a CESTAC evaluation without recompilation," *SCAN 2016*, p. 47, 2016.

[9] S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuilière, "Auto-tuning for floating-point precision with Discrete Stochastic Arithmetic." working paper or preprint, June 2016.

[10] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pp. 1–10, ACM, 2002.

[11] C. Denis, P. de Oliveira Castro, and E. Petit, "Verificarlo: checking floating point accuracy through monte carlo arithmetic," in *Computer Arithmetic (ARITH), 23nd Symposium on*, pp. 55–62, IEEE, 2016.

[12] F. Benz, A. Hildebrandt, and S. Hack, "A dynamic program analysis to find floating-point accuracy problems," *ACM SIGPLAN*, vol. 47, no. 6, pp. 453–462, 2012.

[13] A. Sanchez-Stern, P. Panchekha, S. Lerner, and Z. Tatlock, "Finding root causes of floating point error with herbgrind," *arXiv preprint arXiv:1705.10416*, 2017.

[14] M. O. Lam, J. K. Hollingsworth, and G. Stewart, "Dynamic floating-point cancellation detection," *Parallel Computing*, vol. 39, no. 3, pp. 146–155, 2013.

[15] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[16] J. Reinders, "VTune performance analyzer essentials," *Intel Press*, 2005.

[17] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: transputer and occam developments*, vol. 44, pp. 17–31, IOS Press, 1995.

[18] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The vampir performance analysis tool-set," *Tools for High Performance Computing*, pp. 139–155, 2008.

[19] L. Djoudi, D. Barthou, *et al.*, "Maqao: Modular assembler quality analyzer and optimizer for itanium 2," in *The 4th Workshop on EPIC architectures and compiler technology, San Jose*, vol. 200, 2005.

[20] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization*, p. 75, IEEE, 2004.

[21] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker.," in *USENIX Annual Technical Conference*, pp. 309–318, 2012.

[22] M. Zalewski, "American fuzzy lop," 2015.

[23] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Security and privacy (SP), 2010 IEEE symposium on*, pp. 317–331, IEEE, 2010.

[24] T. Ogita, S. M. Rump, and S. Oishi, "Accurate sum and dot product," *SIAM Journal on Scientific Computing*, vol. 26, no. 6, pp. 1955–1988, 2005.

[25] D. E. Knuth, "The art of computer programming, 3rd edn. seminumerical algorithms, vol. 2," 1997.

[26] F. Févotte and B. Lathuilière, "LibEFT: a library implementing Error-Free transformations," 2017.