Auto-tuning for floating-point precision with Discrete Stochastic Arithmetic

Stef Graillat, Fabienne Jézéquel, Romain Picot, François Févotte, Bruno Lathuilière

Auto-tuning for floating-point precision with Discrete Stochastic Arithmetic

S. Graillat[a], F. Jézéquel[a,b], R. Picot[a,c], F. Févotte[c], B. Lathuilière[c]

[a] *Sorbonne Universités, UPMC Univ Paris 06, CNRS, UMR 7606, Laboratoire d'Informatique de Paris 6 (LIP6), 4 place Jussieu, 75252 Paris CEDEX 05, France*
Email: {Stef.Graillat, Fabienne.Jezequel, Romain.Picot}@lip6.fr
[b] *Université Panthéon-Assas, 12 place du Panthéon, 75231 Paris CEDEX 05, France*
[c] *EDF R&D 7 boulevard Gaspard Monge, F-91120, Palaiseau, France*
Email: {Francois.Fevotte, Bruno.Lathuiliere}@edf.fr

Abstract

The type length chosen for floating-point numbers (e.g. 32 bits or 64 bits) may have an impact on the execution time, especially on SIMD (Single Instruction Multiple Data) units. Furthermore optimizing the types used in a numerical simulation causes a reduction of the data volume that is possibly transferred. In this paper we present PROMISE, a tool that makes it possible to optimize the numerical types in a program by taking into account the requested accuracy on the computed results. With PROMISE the numerical quality of results is verified using DSA (Discrete Stochastic Arithmetic) that enables one to estimate round-off errors. The search for a suitable type configuration is performed with a reasonable complexity thanks to the delta debugging algorithm. The PROMISE tool has been successfully tested on programs implementing several numerical algorithms including linear system solving and also on an industrial code that solves the neutron transport equations.

Keywords: auto-tuning, Discrete Stochastic Arithmetic, floating-point arithmetic, numerical validation, round-off errors

1. Introduction

Nowadays, most numerical simulations are performed in IEEE 754 `binary64` precision (double precision) [11]. This means that a relative accuracy of about 10^{-16} is provided for every arithmetic operation. Indeed, in practice, programmers tend to use the highest precision available in hardware which is the double precision on current processors. This approach can be costly in terms of computing time, memory transfer and energy consumption [1]. A better strategy would be to use no more precision than needed to get the desired accuracy on the computed result. The `binary32` precision (single precision) can be very attractive as it makes it possible to halve the cost of storing and transferring data. Moreover, SIMD instructions like SSE or AVX allow the parallel execution of twice more single precision operations than double precision ones. Using mixed precision arithmetic seems to be a promising way to improve the performance of numerical codes.

The challenge of using mixed precision is to find which variables may be declared in lower precision (let us say single precision) and which ones should stay in higher precision (double precision). The amount of possible configurations is exponential in the number of variables and so the use of a brute-force algorithm is too expensive. To overcome this difficulty, we propose an algorithm and a tool called PROMISE (PRecision OptiMISEd) that provide a mixed precision configuration with a worst-case complexity quadratic in the number of variables. PROMISE returns a subset of variables that can be transformed into single precision, taking into account a required accuracy on the computed result. It is based on the delta debugging search algorithm [28, 29] and Discrete Stochastic Arithmetic (DSA) [27].

Some tools already exist to auto-tune floating-point precision. We briefly review them below.

From an initial double precision program and a data set, CRAFT HPC [16] searches for possible mixed precision configurations that pass a user-provided verification routine and chooses the one that optimizes

the speed-up and the memory consumption. CRAFT HPC implements a breadth-first search algorithm to find in a program the coarsest granularity at which single precision can be used. Single precision operations are actually simulated by modifying the 64 bits of double precision operands. Changes are performed in the binary code. Then the source code can be modified thanks to a graphical configurator editor.

From an initial program and representative data sets, Precimonious [22] proposes a mixed precision configuration that satisfies both accuracy and performance constraints. Precimonious provides an interface where the accuracy requirement can be specified. The configuration search is based on the delta debugging algorithm [28, 29] and several floating-point types can be tested for each variable. Precimonious uses the LLVM [18] compiler to transform the program into a binary file and type modifications are performed on the LLVM bitcode file. Finally Precimonious provides an LLVM bitcode file with optimized variable declarations.

Blame Analysis [21] aims at improving the performance of precision tuning tools. Taking into account an accuracy requirement, it can lower the precision of some variables in the input program, but does not necessarily improve its execution time. Blame Analysis determines which precision configurations satisfy the accuracy constraint by executing instructions that have been instrumented several times with different precisions. These additional executions are performed side-by-side with the concrete execution. Blame Analysis improves the execution time of Precimonious by removing some variables from its search space.

The main contributions of this paper are an algorithm with a reasonable complexity and a tool named PROMISE[1] for auto-tuning floating-point precision. From an initial C or C++ program, PROMISE automatically modifies the precision of variables taking into account an accuracy requirement on the computed result. PROMISE has been successfully tested on several benchmark programs and also on a large industrial code. To estimate the numerical quality of results, PROMISE uses DSA that controls round-off errors in simulation programs. PROMISE automatically provides a code with an optimized type configuration and instrumented or not for DSA, according to the user's choice. The version of the code instrumented for DSA enables one to use all its features: in particular the estimation of round-off errors in any result and the detection of numerical instabilities.

The rest of the paper is organized as follows. In Section 2, we present the principles of DSA and the CADNA library that implements it. In Section 3 we describe the PROMISE algorithm. In Section 4 we present the two versions of the PROMISE tool that have been developed. Section 5 is devoted to experimental evaluations of the PROMISE tool that has been tested on classical benchmark programs but also on an industrial code that solves the neutron transport equations. Finally, we conclude and discuss future works in Section 6.

2. Discrete Stochastic Arithmetic (DSA)

2.1. Principles of DSA

The CESTAC method [26] allows the estimation of round-off error propagation which occurs with floating-point arithmetic. This probabilistic method uses a random rounding mode: at each elementary operation, the result is rounded up or down with the probability 0.5. With this random rounding mode, the same program run several times provides different results, due to different round-off errors. The computer's deterministic arithmetic, therefore, is replaced by a stochastic arithmetic where each arithmetic operation is performed N times before the next one is executed, thereby propagating the round-off error differently each time. The CESTAC method furnishes us with N samples $R_1, \ldots, R_N$ of the computed result R. The value of the computed result $\overline{R}$ is chosen to be the mean value of $\{R_i\}$ and, if no overflow occurs, the number of exact significant digits in $\overline{R}$, $C_{\overline{R}}$, can be estimated as

$$C_{\overline{R}} = \log_{10}\left(\frac{\sqrt{N}\,|\overline{R}|}{\sigma\tau_\beta}\right)$$

[1] URL address: `http://promise.lip6.fr/`

$$\text{with } \overline{R} = \frac{1}{N}\sum_{i=1}^{N} R_i \text{ and } \sigma^2 = \frac{1}{N-1}\sum_{i=1}^{N}\left(R_i - \overline{R}\right)^2.$$

τ_β is the value of Student's distribution for $N-1$ degrees of freedom and a probability level $1-\beta$. In practice $\beta = 0.05$ and $N = 3$. Indeed, the estimation with $N = 3$ is more reliable than with $N = 2$ and increasing the size of the sample does not significantly improve the quality of the estimation. The complete theory can be found in [3, 26].

The validity of $C_{\overline{R}}$ is compromised if the two operands in a multiplication or the divisor in a division are not significant [3]. Therefore the CESTAC method requires, during the execution of the code, a dynamical control of multiplications and divisions, which is a so-called *self-validation* of the method. This self-validation has led to the synchronous implementation of the method, *i.e.* to the parallel computation of the N results R_i, and also to the concept of computational zero [25]. A computed result is a computational zero, denoted by @.0, if $\forall i, R_i = 0$ or $C_{\overline{R}} \leq 0$. This means that a computational zero is either a mathematical zero or a number without any significance, *i.e.* numerical noise.

To establish consistency between the arithmetic operators and the relational operators, discrete stochastic relations are defined as follows. Let $X = (X_1, ..., X_N)$ and $Y = (Y_1, ..., Y_N)$ be two results computed using the CESTAC method, we have from [4]

$$\begin{aligned}
&X = Y \text{ if and only if } X - Y = @.0;\\
&X > Y \text{ if and only if } \overline{X} > \overline{Y} \text{ and } X - Y \neq @.0;\\
&X \geq Y \text{ if and only if } \overline{X} \geq \overline{Y} \text{ or } X - Y = @.0.
\end{aligned}$$

Discrete Stochastic Arithmetic (DSA) [26, 27] is the combination of the CESTAC method, the concept of computational zero, and the discrete stochastic relationships.

2.2. The CADNA software

The CADNA[2] software [3, 7, 12, 17] is a library which implements DSA in any code written in C, C++ or Fortran. It has been successfully used for the numerical validation of academic and industrial simulation codes in various domains such as astrophysics [15], atomic physics [24], chemistry, climate science [2, 13], fluid dynamics, geophysics [14]. CADNA allows one to use new numerical types: the stochastic types. In practice, classic floating-point variables are replaced by the corresponding stochastic variables, which are composed of three floating-point values and an integer to store the accuracy. The library contains the definition of all arithmetic operations and order relations for the stochastic types.

The round-off error that affects any stochastic variable can be estimated with the probability 95%. Only exact significant digits of a stochastic variable are printed or "@.0" for a computational zero. Because all operators are redefined for stochastic variables, the use of CADNA in a program requires only a few modifications: essentially changes in the declarations of variables and in input/output statements. CADNA can detect numerical instabilities which occur during the execution of the code. Such instabilities are usually generated by an operation involving a computational zero. When a numerical instability is detected, dedicated CADNA counters are incremented. At the end of the run, the value of these counters together with appropriate warning messages are printed on standard output.

3. The PROMISE algorithm

3.1. Searching suitable type declarations

From an original program and a requested accuracy on the result, PROMISE provides a transformed program having a maximum number of variables declared with a lower precision and computing a result that satisfies the accuracy constraint. The number of exact significant digits in the result is estimated using

[2]URL address: `http://cadna.lip6.fr`

the CADNA software. In order to simplify the algorithm presentation, we will consider the case where the input program has all its variables declared as *double* (`binary64`) [11] and PROMISE can change the type of any variable into *single* (`binary32`). However both *single* and *double* variables can be declared in the input program. In this case, PROMISE first transforms all *single* variables into *double* ones and then executes the type optimization algorithm.

PROMISE aims at maximizing the number of variables declared as *single*. In order to describe its algorithm, we need the following notations and definitions. We denote by C the set of all variables and C^s (resp. C^d) the set of variables in single (resp. double) precision. So we have $C^d \cap C^s = \emptyset$ and $C^d \cup C^s = C$.

Definition 1. *A* configuration *is a couple of sets* (C^s, C^d) *and the set of all possible configurations is denoted by* $\mathcal{R}$.

Definition 2. *A result is* accurate *if it satisfies the requested accuracy.*

Definition 3. *The function* $\text{test} : \mathcal{R} \to \{\checkmark, \times\}$ *determines for a configuration if the result is accurate (*$\checkmark$*) or not (*$\times$*).*

Definition 4. *A configuration* (C^s, C^d) *is* m-maximal *if* $\text{test}(C^s, C^d) = \checkmark$ *and for any* $(C^{s\prime}, C^{d\prime})$ *with* $C^s \subset C^{s\prime}$, $C^{d\prime} \subset C^d$, *and* $C^{s\prime} \cup C^{d\prime} = C^s \cup C^d$

$$\text{test}(C^{s\prime}, C^{d\prime}) = \times \text{ and } |C^{s\prime}| - |C^s| \leq m.$$

Definition 5. *A configuration* (C^s, C^d) *is* relevant *if it is 1-maximal in the sense of Definition 4. So*

$$\forall \delta \in C^d, \text{test}(C^s \cup \{\delta\}, C^d \backslash \{\delta\}) = \times.$$

In order to find a relevant configuration, PROMISE uses a slightly modified version of the delta debugging algorithm [28, 29] which is based on a *divide and conquer* method. If we consider a program with n variables, searching for a global maximum configuration would require to test 2^n possibilities. As a comparison, PROMISE has the same complexity as the delta debugging algorithm: $O(n^2)$ in the worst case and $O(n \log(n))$ in the average case [28]. Because the delta debugging algorithm has been designed to identify program bugs, it has been adapted to precision optimization. The main modifications are searching for a maximum passing configuration rather than finding a minimum failure modification and taking into account the accuracy instead of program failures.

Two versions of the algorithm used in PROMISE are described: first the recursive version, and then the iterative one which has actually been implemented.

3.2. Recursive algorithm

Algorithm 1 is the recursive version of the algorithm used in PROMISE. Because all variables are initially declared in double precision, it starts with $\text{PROMISE}(\emptyset, C, 2)$. The set C^d of *double* variables is partitioned in p subsets C_i^d such that $\cup_{i=1}^{p} C_i^d = C^d$ with $|C_i^d| \approx \frac{|C^d|}{p}$. Variables are successively taken into account to create each subset. In the sequel, p is referred to as the granularity. Configurations where all elements of a subset C_i^d are transformed in single precision are tested in Algorithm 1. At each recursion, three possibilities may occur.

1. First, if it exists a subset C_i^d with $\text{test}(C^s \cup C_i^d, C^d \backslash C_i^d) = \checkmark$, then we consider that elements from C_i^d should be definitely added to the set C^s. Furthermore, the granularity is set to $p-1$. The granularity must be at least 2. Indeed, if it reaches 1, a redundant test would be performed.
2. The second case occurs when all tests fail and the granularity p is lower than the size of C^d. The algorithm will consider the same couple (C^s, C^d) with a higher granularity. Because we must not have more subsets than the number of elements in C^d, the granularity is then set to $\min(2p, |C^d|)$.
3. The last case occurs when all tests fail and the granularity is equal to the size of C^d. Then all the possibilities have already been tried. Another run is useless, the algorithm stops and returns C^s.

Algorithm 1 PROMISE recursive algorithm

PROMISE(C^s, C^d, p)=

$$\begin{cases} \textbf{1}-\text{PROMISE}(C^s \cup C_i^d, C^d \backslash C_i^d, \max(p-1,2)) \\ \quad \text{if } \exists i \in \{1, \cdots, p\} \text{ test}(C^s \cup C_i^d,\ C^d \backslash C_i^d) = \checkmark; \\ \textbf{2}-\text{PROMISE}(C^s, C^d, \min(2p, |C^d|)) \quad \text{if } p < |C^d|; \\ \textbf{3}-\text{return } C^s \qquad\qquad\qquad\qquad\quad \text{otherwise.} \end{cases}$$

Algorithm 2 test function (determines if a type configuration is valid)

function *state* = test(C^s, C^d)

1: **if** (C^s, C^d) was previously tested **then**
2: return the result on cache
3: **end if**
4: Update the source code according to (C^s, C^d)
5: **if** the source code compilation fails **then**
6: return ✗
7: **end if**
8: Run the modified program
9: **if** the execution fails **then**
10: return ✗
11: **else if** the result is accurate **then**
12: return ✓
13: **end if**
14: return ✗

The function test introduced by Definition 3 is presented in Algorithm 2.

Lines 1-3 refer to the possibility that the same configuration may be tested twice. This could be the case when a configuration (C^s, C^d) fails but another one $(C^{s\prime}, C^{d\prime})$ with $C^{s\prime} \subset C^s$ and $C^{s\prime} \cup C^{d\prime} = C^s \cup C^d$ passes. Figure 1 exemplifies such a situation with a program having 4 variables $(v_i, i = 0, \ldots, 3)$. Figure 1 presents the successive calls to Algorithms 1 and 2. One can notice that step 4 leads to a configuration already tested at step 1. In this case, the appropriate result on cache is returned.

	variables				test	PROMISE(C^s, C^d, p)
steps:	v_0	v_1	v_2	v_3		
1					test$(\{v_0, v_1\}, \{v_2, v_3\}) =$ ✗	PROMISE$(\emptyset,\ \{v_0, v_1, v_2, v_3\},\ 2)$
2					test$(\{v_2, v_3\}, \{v_0, v_1\}) =$ ✗	
3					test$(\{v_0\}, \{v_1, v_2, v_3\}) =$ ✓	PROMISE$(\emptyset,\ \{v_0, v_1, v_2, v_3\},\ 4)$
4					already tested (step 1)	PROMISE$(\{v_0\},\ \{v_1, v_2, v_3\},\ 3)$

Figure 1: Situation where a result on cache is returned in order to avoid a redundant test. A program with 4 variables $(v_0, \ldots, v_3)$ is considered. Single (resp. double) precision variables are represented in grey (resp. white). At step 4 a test already performed at step 1 is avoided.

The compilation is tested (lines 5-7). Indeed, modifying types could lead to compilation errors. Such an error would occur for instance if a function that requires a double precision array as a parameter actually receives a single precision array. This kind of difference between the prototype and the function call would generate a compilation error.

Even if no compilation error occurs, a possible error during the execution must be taken into account (lines 9-10). Such an error may be a division by zero due to a conversion. For example, if the double

precision variable x initialized as $x = 1 + 10^{-14}$ is converted in single precision, its value may become 1, depending on the rounding mode. Then the expression $1/(1-x)$ would generate a division by zero.

Finally the test passes if the result accuracy is satisfactory (lines 11-12). The conditions that must be fulfilled for a result to be considered as accurate are described in 4.1.

Figure 2 exemplifies the evolution of the search tree and the subdivisions of the variable sets. Each grey or white area corresponds to at least one variable. If transforming a subset in single precision leads to a successful test, then the remaining configurations at the same level in the tree are not tested. As mentioned before, the cache mechanism avoids tests already performed.

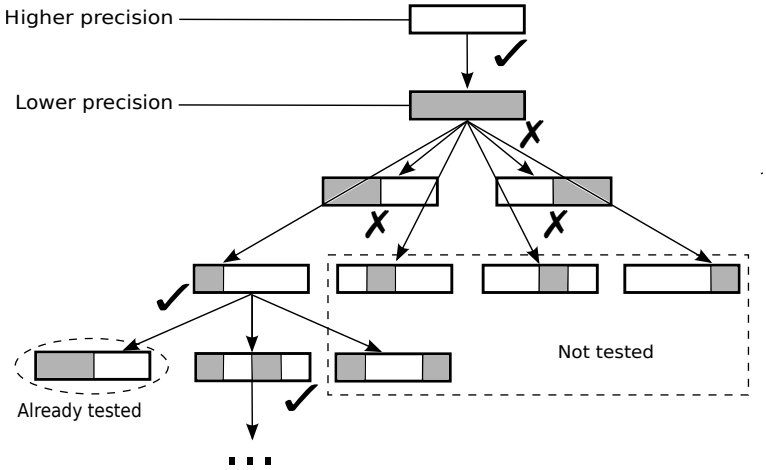


Figure 2: Creation of subsets by PROMISE. Subsets with single (resp. double) precision variables are represented in grey (resp. white).

3.3. Iterative algorithm

Algorithm 3 presents the iterative version of Algorithm 1 that is actually implemented in PROMISE.

The set C of all variables is given as input. As explained before, we consider only *single* and *double* variables and assume that all variables are in double precision at the beginning of Algorithm 3 (line 2). The $found$ variable determines if a correct configuration with more variables declared as *single* has been found or not.

In Algorithm 3 two nested loops are performed. The outer loop is executed until the granularity p reaches $|C^d|$ and no element could be added to C^s at the last iteration (line 4). At each iteration of the outer loop, p subsets are created (line 5).

Iterations of the inner loop are performed until a set C_i^d transformed in *single* passes the test (line 9) or all the subsets previously created have been tested (line 8). When a test succeeds, C^s and C^d are modified accordingly (line 10).

After the inner loop, if a configuration has passed, the granularity is modified (line 16), otherwise if the granularity has not reached the size of C^d yet, it is doubled or set to $|C^d|$ (line 18). Line 19 enables one to execute the outer loop when the granularity equals $|C^d|$.

After this last step, the algorithm returns C^s (line 22) if it has not been modified ($found = false$).

3.4. Example

As an example, Figure 3 presents the execution of Algorithm 1 with a program having ten variables: v_0 to v_9 with the same formalism as Figure 1 except that the steps on cache are not represented. The success of the test function at steps 3, 6, 10, 11 and 14 allows to definitely add element(s) to C^s following the case 1 of Algorithm 1. Following the case 2 of Algorithm 1, after the steps 2, 8 and 13, the size of the subset tested in single precision is reduced. At step 15, a configuration with v_2 in C^s is tested. The test with only

Algorithm 3 PROMISE iterative algorithm

function PROMISE(C)

1: $p \leftarrow 2$
2: $C^s \leftarrow \emptyset$; $C^d \leftarrow C$
3: $found \leftarrow False$
4: **while** $found = True$ or $p < |C^d|$ **do**
5: partition C^d as $\cup_{i=1}^{p} C_i^d = C^d$ with $|C_i^d| \approx \frac{|C^d|}{p}$
6: $found \leftarrow False$
7: $i \leftarrow 1$
8: **while** $i \leq p$ and $found \neq True$ **do**
9: **if** test$(C^s \cup C_i^d, C^d \backslash C_i^d) =$ ✓ **then**
10: $C^s \leftarrow C^s \cup C_i^d$; $C^d \leftarrow C^d \backslash C_i^d$
11: $found \leftarrow True$
12: **end if**
13: $i \leftarrow i + 1$
14: **end while**
15: **if** $found = True$ **then**
16: $p \leftarrow \max(p - 1, 2)$
17: **else if** $p < |C^d|$ **then**
18: $p \leftarrow \min(2p, |C^d|)$
19: $found \leftarrow True$
20: **end if**
21: **end while**
22: return C^s

v_2 in double precision is not executed because it has already been performed at step 12: the result in cache is used. The set C^s returned is the last one leading to a satisfactory accuracy: $\{v_0, v_1, v_3, v_4, v_5, v_7, v_8, v_9\}$ in our example. Therefore variables v_2 and v_6 should be declared in double precision in the program returned by the PROMISE tool.

4. The PROMISE tool

We describe here how the PROMISE tool has been developed. We actually present two versions of the PROMISE tool that differ by the way the accuracy of results is verified.

4.1. Result accuracy verification

For each type configuration considered, PROMISE tests if the associated computed result satisfies the requested accuracy. A comparison with a high precision reference result computed using classical floating-pint arithmetic would not be satisfactory, because this result may be not reliable. Indeed we can mention as an example a numerical expression proposed by Rump [23] that provides the same first digits in single, double and extended precision. But in fact, that result, detected by DSA as numerical noise [9], is totally different from the correct evaluation. Therefore PROMISE uses DSA to control the numerical quality of the computed result. Actually two versions of PROMISE have been developed, hereafter referred to as the "full stochastic" version and the "stochastic reference" one.

With the full stochastic version, all executions are performed with DSA. The result accuracy is considered satisfactory if two conditions are fulfilled. The number of exact significant digits estimated using DSA must be at least the requested number of digits and these digits must be in common with the result obtained in double precision using DSA. The latter condition is due to the fact that if a high number of absorptions occur, the hypotheses of DSA may be violated and the accuracy of the mixed precision result may be

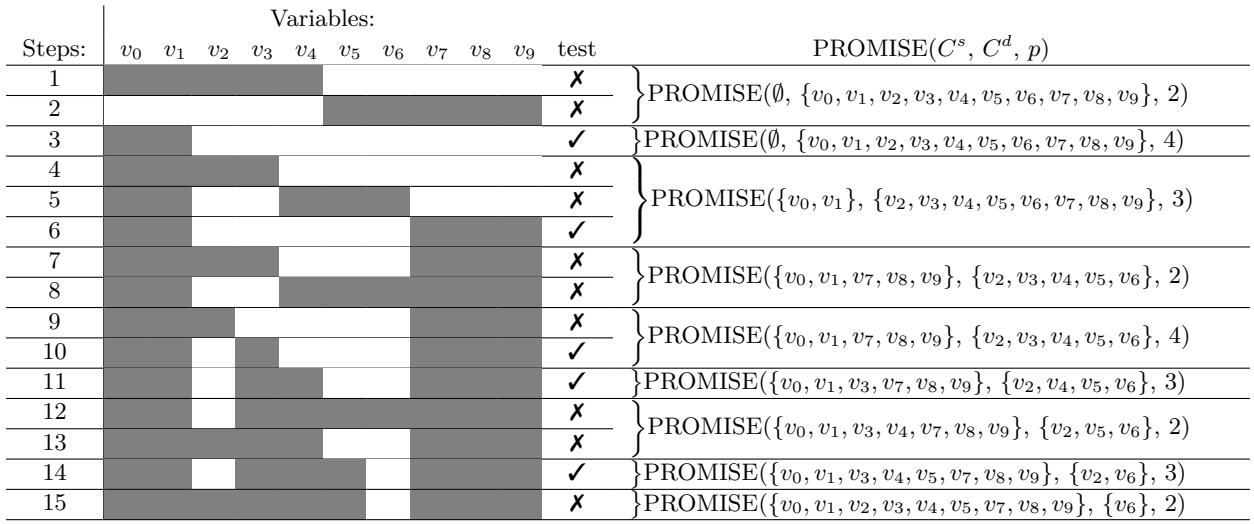


Figure 3: Execution of Algorithm 1 with a program having 10 variables ($v_0, \ldots, v_9$). Single (resp. double) precision variables are represented in grey (resp. white).

incorrectly estimated by DSA. This situation, that happens in one of the programs tested in this paper, is described in detail in [19, p. 29-31].

Because of the cost of DSA in terms of both execution time and memory, another version of PROMISE has been developed. This so-called "stochastic reference" version computes a reference result in double precision with DSA. Then, for each type configuration considered, the program is executed without DSA (i.e. in classical floating-point arithmetic) and the result produced is compared with the reference result obtained with DSA. The reference result used in the floating-point arithmetic programs is actually the mean value of the N samples of the reference result obtained with DSA.

As a remark, the accuracy requirement may concern one scalar result or several computed results, e.g. an entire array. Algorithms 4 and 5 describe how the accuracy of n computed results is verified with the two versions of the PROMISE tool.

Algorithm 4 Result accuracy verification with the full stochastic version of PROMISE

function `result_accuracy`(val, ref, req)
val_i ($i = 1, \ldots, n$): computed results
ref_i ($i = 1, \ldots, n$): reference results
req: requested accuracy

1: **for** $i = 1, \ldots, n$ **do**
2: **if** number of exact significant digits of $val_i < req$ **then**
3: return ✗
4: **end if**
5: **if** $|val_i - ref_i| \geq |ref_i| * 10^{-req}$ **then**
6: return ✗
7: **end if**
8: **end for**
9: return ✓

Algorithm 5 Result accuracy verification with the stochastic reference version of PROMISE

function `result_accuracy`(val, ref, req)
val_i $(i = 1, \ldots, n)$: computed results
ref_i $(i = 1, \ldots, n)$: reference results
req: requested accuracy

1: **for** $i = 1, \ldots, n$ **do**
2: **if** $|val_i - ref_i| \geq |ref_i| * 10^{-req}$ **then**
3: return ✗
4: **end if**
5: **end for**
6: return ✓

4.2. Implementation

The PROMISE tool has been developed in Python and uses a modified version of Zeller's implementation of the delta debugging algorithm[3]. Figure 4 summarizes the execution of the two versions of the PROMISE tool: the full stochastic version and the stochastic reference one. PROMISE automatically transforms an initial C or C++ program to enable its execution with the CADNA software. These modifications are mainly changes in type declarations in order to use stochastic variables. Whatever the version of PROMISE, the user program is executed once with CADNA in double precision to produce a reference result. Then PROMISE verifies that the accuracy of this result (estimated by CADNA) is satisfactory. If the single precision version does not provide a sufficiently accurate result, a mixed precision configuration that satisfies the accuracy requirement is searched using Algorithm 3. The single precision program and the mixed precision ones are executed with CADNA if the full stochastic version of PROMISE has been chosen. With the stochastic reference version, the instructions and declarations specific to the CADNA software are removed, so as the programs are executed with the classical floating-point arithmetic. When an appropriate type configuration is found, the associated program is returned with ou without CADNA statements, depending on the user's choice. By default, PROMISE performs a tuning of all floating-point variables type declarations. It is however possible to specify a list of variable declarations which must not be changed. This allows for faster executions and finer grained tuning when developers have previous knowledge of the numerical stability of their programs.

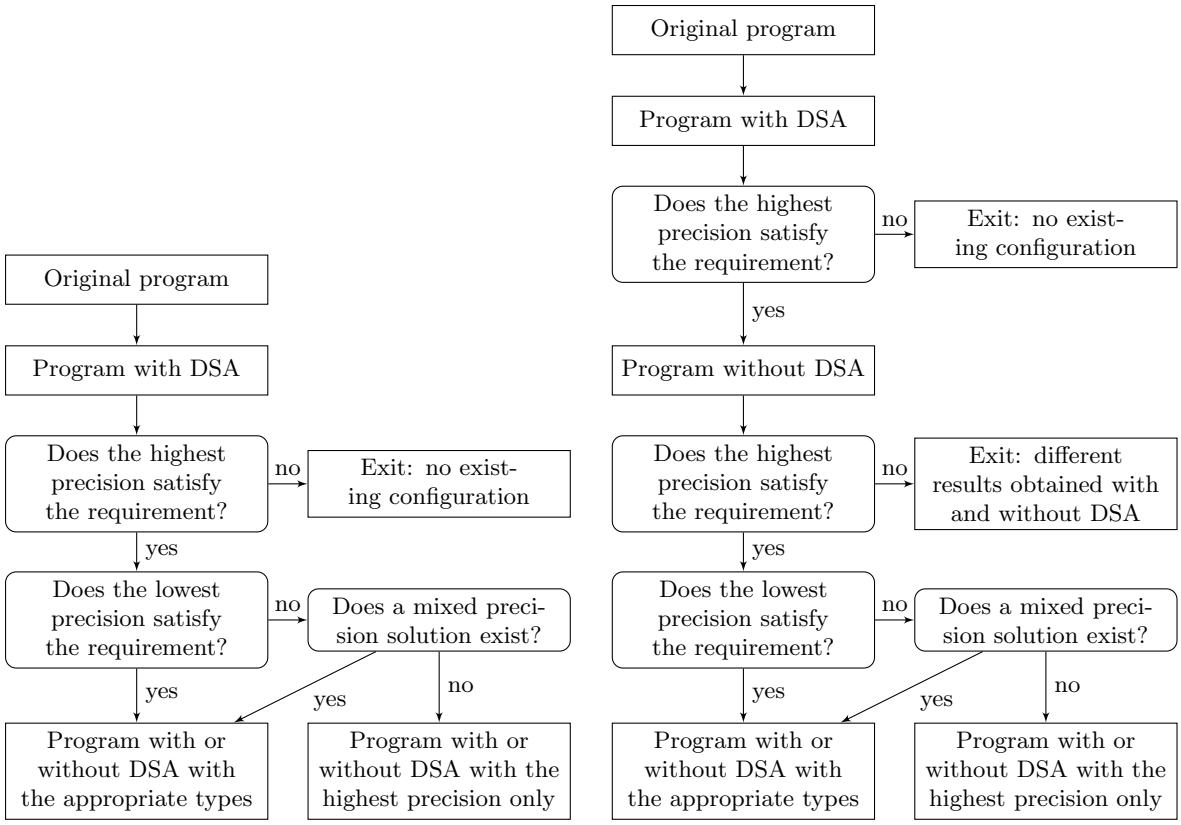
5. Experimental evaluation

5.1. Experiment setup

All tests have been performed on a desktop platform with an Intel Xeon CPU E5-2670 at 2.6 GHz and 128 GB RAM. All codes mentioned in this section are written in C++ and compiled with GCC 4.9.2 and optimization level O3. CADNA for C/C++ codes version 2.0.0 [7] is used. Codes come from several sources, the names in parentheses in the list below will be used in the remainder of this article.

- short test programs: arclength computation (arclength), rectangle method for the computation of integrals (rectangle), Babylonian method (squareRoot) and matrix multiplication (MatMul);
- GNU Scientific Library [5]: Fast Fourier Transform (FFT), sum of Taylor series terms (sum) and polynomial evaluation/solver (poly);
- SNU NPB Suite [6]: Conjugate Gradient method (CG) used with a matrix of size 7,000 with 8 non-zero values per row, the number of iterations being set to 15, and a Scalar Penta-diagonal solver (SP) used to solve a 3D discretization of the Navier-Stokes equation.

[3]URL address: `https://www.st.cs.uni-saarland.de/dd/DD.py`



(a) Full stochastic version of PROMISE

(b) Stochastic reference version of PROMISE

Figure 4: PROMISE execution flowchart

In the tests presented here, all variable declarations are considered for tuning. In all the programs considered, the accuracy of all the results is verified. In particular, in cases where results are matrices or vectors, all elements must fulfill the requested accuracy as mentioned in section 4.1. In order to enable a comparison with Precimonious, another precision auto-tuning tool, some programs implement numerical methods mentioned in [22]. In particular, the instructions of the arclength program are taken from [22], but its variables are here declared in double precision. Nevertheless, some programs could be different or with a different number of tested variables.

The results of these tests are discussed in the sequel of this section. Furthermore, we also tested PROMISE on MICADO, a large-scale industrial computing code which is more complex to analyse and will be the subject of paragraph 5.4.

5.2. Results obtained with several numerical programs

Tables 1a and 1b present results provided by PROMISE with the programs previously mentioned using respectively the full stochastic version and the stochastic reference version (both described in Section 4). For each program, different rows in Table 1 correspond to several accuracy requirements: 4, 6, 8 and 10 exact significant digits. The results reported in Table 1 are:

- the number of compilations performed by PROMISE (# comp); this is also the number of tested configurations;
- the number of executions of the transformed programs (# exec); this number might be different from the previous one when the compilation fails for some configurations;

- the number of variables which must be stored in double precision (# double);
- the number of variables which can be relaxed to single precision (# float);
- the total execution time of PROMISE, including the compilation and the execution of the transformed codes, as well as the time spent in PROMISE's own routines;
- the speed-up of the proposed configuration, when run without CADNA, with respect to the initial configuration (all variables in double precision). The ratio reported is the time of the initial program over the time of the transformed one.

Over the 9 programs and the 4 accuracy requirements tested, both versions of PROMISE always propose a new configuration with variables that can be relaxed in single precision. In 14 of the test cases, the two versions of PROMISE provide different type configurations. Indeed the full stochastic version may lead to configurations with more double precision variables. This is due to differences in the result accuracy verification. In both versions, the computed result is compared with the reference one. However in the full stochastic version, another test is performed: the result accuracy estimated by CADNA is compared to the requested accuracy. Futhermore, in the full stochastic version, the comparison between the computed result and the reference one is performed in DSA. In cases when the difference between these results is numerical noise, the computed result is considered not satisfactory. Such a numerical instability cannot be detected with the stochastic reference version, because the type configurations are tested in classical floating-point arithmetic.

As a remark, when the same type configuration is provided by the two versions of PROMISE, one more compilation and one more execution are performed with the stochastic reference version. This is due to the fact that, with the stochastic reference version, two double precision programs are tested: first in DSA and then in classical floating-point arithmetic.

The run time of PROMISE depends on the version chosen (full stochastic or stochastic reference), the execution time of the program tested, its number of variables and the number of compilations/executions performed. The stochastic reference version performs better than the full stochastic one: from Table 1 the ratio of their execution times can be up to 91. This is mainly due to the cost of CADNA. Furthermore the number of configurations tested may be lower with the stochastic reference version. Indeed a configuration may be considered valid with this version and not with the full stochastic one.

Only the SP program from a certain requested accuracy leads to a total run time greater than 10 minutes. This is mostly due to relatively long compilation and execution times. Such test cases with larger compile and run time also stress the importance of a correct optimization technique: out of the 2^{110} possible different configurations, delta debugging considers in the worst case only 276 possibilities. On the other hand, if we consider smaller and faster programs like arclength, we find that no accuracy constraint requires more than 1 minute to tune the precision of the 9 variables. Performing such tests by hand, even on as few as two configurations, would have been much more time consuming than using PROMISE.

It is also worth noting that in our test cases, compilations can end with errors. In the worst cases, that happen with the SP program, only 7% of the tested configurations are executed. A detailed analysis shows that compilations fail because of type differences in array arguments between the function calls and their prototypes. However compilation errors cause only a minor slowdown on the code since the compilation is a limited amount of the execution time of PROMISE.

The execution times obtained with the new configurations are similar to those of the original programs. PROMISE differs from tools like Precimonious which incorporate execution times in their metrics, as its only objective is to lower the precision of variables and validate the code. Slow-downs sometimes occur, but they are low (at most 2%). They can be explained by the cost of casts between single and double precision values. However the type configurations provided by PROMISE could help developers to improve the performance of vectorized codes by increasing the number of single precision instructions executed by SIMD units.

Program	# digits	# comp - # exec	# double - # float	Time (mm:ss)	Speed-up
arclength	10	20-20	8-1	0:37	1.01
	8				
	6	25-25	7-2	0:48	1.00
	4	18-18	3-6	0:32	1.01
MatMul	10	6-6	2-1	0:05	0.99
	8				
	6				
	4				
rectangle	10	14-14	4-3	0:11	1.00
	8				
	6	15-15	3-4	0:10	1.01
	4	6-6	1-6	0:06	1.02
squareRoot	10	20-20	6-2	0:16	1.00
	8				
	6	2-2	0-8	0:01	1.02
	4				
FFT	10	23-9	3-19	0:18	1.11
	8				
	6	2-2	0-22	0:03	1.10
	4				
poly	10	232-74	53-66	7:07	1.12
	8				
	6	187-28	41-78	5:05	1.11
	4	2-2	0-119	0:05	1.11
sum	10	307-46	54-45	7:56	1.06
	8				
	6	299-38	52-47	7:36	1.04
	4	2-2	0-99	0:05	1.05
CG	10	123-28	24-23	6:54	1.02
	8				
	6	114-22	23-24	6:07	1.03
	4	2-2	0-47	0:15	1.03
SP	10	276-31	64-46	275:24	1.00
	8				
	6	264-18	61-49	209:07	1.01
	4	2-2	0-110	9:59	1.02

(a) Full stochastic version of PROMISE

Program	# digits	# comp - # exec	# double - # float	Time (mm:ss)	Speed-up
arclength	10	21-21	8-1	0:13	1.01
	8				
	6	26-26	7-2	0:15	1.00
	4	16-16	2-7	0:09	0.98
MatMul	10	7-7	2-1	0:03	0.99
	8				
	6				
	4	3-3	0-3	0:02	1.00
rectangle	10	15-15	4-3	0:06	1.00
	8				
	6	16-16	3-4	0:06	1.01
	4	3-3	0-7	0:01	1.01
squareRoot	10	21-21	6-2	0:07	1.00
	8				
	6	3-3	0-8	0:01	1.02
	4				
FFT	10	24-10	3-19	0:07	1.11
	8				
	6	3-3	0-22	0:01	1.10
	4				
poly	10	233-75	53-66	2:16	1.12
	8	229-71	52-67	2:13	1.11
	6	3-3	0-119	0:04	1.11
	4				
sum	10	297-46	50-49	5:40	1.03
	8				
	6	3-3	0-99	0:05	1.05
	4				
CG	10	115-23	23-24	2:18	1.03
	8				
	6	3-3	0-47	0:11	1.03
	4				
SP	10	265-19	61-49	25:04	1.01
	8				
	6	3-3	0-110	5:31	1.02
	4				

(b) Stochastic reference version of PROMISE

Table 1: Experimental results of PROMISE on several test cases.

Program	# digits	# configurations	# `double`	# `float`	Time (mm:ss)	Speed-up
arclength	10 8	40	8	1	2:12	1.01
	6	46	7	2	2:32	1.00
	4	60	6	3	3:19	1.02
rectangle	10 8	44	5	2	0:14	1.00
	6	28	3	4	0:09	1.01
	4	2	0	7	0:01	1.01

Table 2: Experimental results of Precimonious

5.3. Comparison with Precimonious

As previously mentioned, although PROMISE shares some objectives and methodologies of Precimonious, there are important differences which are worth noting. First, Precimonious relies on a reference result computed using classical floating-point arithmetic, whereas PROMISE uses DSA to validate its results. Then, with PROMISE we deliberately do not take into account the execution time of modified configurations as we focus on maximizing the number of single precision variables to enhance vectorization. Precimonious focuses on a speed-up tuning. For each configuration, it validates the result using its accuracy and the execution time. If a configuration is accurate enough but slower than a previous one, it is not considered valid. Therefore the configuration provided by Precimonious may depend on the computer and the compiler used, whereas the solution obtained with PROMISE is reproducible.

We analyse here results provided by Precimonious when the execution time is not taken into account for testing the possible configurations. We present the results obtained with two programs already considered in 5.2 (arclength and rectangle) and initially in double precision. In Table 2 are reported, for each requested accuracy, the number of configurations tested, the number of variables of each type in the configuration proposed, the execution time of Precimonious, and the speed-up obtained with the transformed program with respect to the initial one.

It can be noticed that the number of variables that can be transformed in `float` is lower or equal than the one found by PROMISE. This is due to the implementation of the delta debugging algorithm in Precimonious. If the granularity is greater than the number of remaining variables, Precimonious does not test the remaining possibilities, whereas PROMISE tries to transform those variables one by one in single precision. Consequently PROMISE may produce configurations with more single precision variables.

Considering the execution time, with the arclength program, PROMISE performs better than Precimonious, even if the full stochastic version is used, the execution time ratio being up to 22. With the rectangle program, no such ratio can be noticed, however one can observe that with the stochastic reference version of PROMISE the execution time is lower or equal than with Precimonious. The better performances of PROMISE can be related to the lower number of configurations tested. Indeed the ratio of the number of configurations tested by Precimonious and PROMISE is up to 3. Again this is due to differences in the delta debugging algorithm used. Precimonious implements a version of the delta debugging algorithm from [29]. In this version, if the granularity is sufficiently high, the algorithm tests a subset in single precision, and then its complement. In PROMISE, the implementation of the delta debugging algorithm is based on the version from [28]. In this version, if a subset can be transformed in single precision, its complement is not tested. This reduces the number of configurations tested by PROMISE. Also, it is worth noticing that if the execution time is not taken into account by Precimonious, the speed-up obtained with the transformed program is similar to that observed with PROMISE.

The same arclength program, except variables are initially declared as `long double`, is analysed by Precimonious in [22]. Indeed Precimonious includes `long double` in the possible types of its input and output programs. The results described in [22] have been obtained by taking into account the execution time in the selection of the configuration to propose. One can notice that with the arclength program the configurations presented in [22] have more single precision variables than those in Table 2. The speed-ups,

# digits	# comp - # exec	# double - # float	Time (mm:ss)	Speed up	memory gain
10	75-36	20-31	276:35	1.01	1.00
8	72-33	19-32	255:17	1.01	1.01
6	75-33	17-34	251:57	1.06	1.43
5	61-22	11-40	177:12	1.20	1.43
4	60-21	11-40	171:58	1.24	1.50

(a) With the full stochastic version

# digits	# comp - # exec	# double - # float	Time (mm:ss)	Speed up	memory gain
10	83-51	19-32	88:56	1.01	1.00
8	80-48	18-33	85:10	1.01	1.01
6	69-37	13-38	71:32	1.20	1.44
5, 4	3-3	0-51	9:58	1.32	1.62

(b) With the stochastic reference version

Table 3: Experimental results of PROMISE on the MICADO software.

evaluated with respect to the `long double` version, are also better (from 11.0% to 41.7%). Except for 4 requested digits, the number of configurations tested is higher in [22] than in Table 2, the ratio being up to 4.75. This has an impact on the execution time of Precimonious itself.

All these differences are consequences of the objectives of each tool. Precimonious aims at producing faster configurations; it therefore discards any configuration which does not produce a speed-up. At each stage of the delta debugging algorithm, it also tries all sets to find the fastest configuration when our algorithm stops when one has enough accuracy. This accounts for both the higher speed-up of the configuration proposed, and the higher number of configurations tested.

5.4. Results obtained with an industrial code

We tested our tool on MICADO [8], a simulation code developed by EDF (Electricité De France), the main French electric utility company. MICADO is used to simulate nuclear cores in nuclear plants. It is a C++ software, with 11,000 code lines, that solves the neutron transport equations in 2D or 3D geometries using the method of characteristics [10]. MICADO creates parallel characteristic lines in a two-dimensional plane, and computes intersections between these lines and the regions defining the geometry of a nuclear reactor core. The resulting line segments are then swept to compute the neutron flux within the core. Industrial applications of such codes include for example knowing whether the nuclear reaction is stable (*i.e.* the number of neutrons does not vary over time) or determining the burning rate of nuclear fuel.

The code is composed of a tracker that computes the characteristic lines and a solver based on the inverse power iteration algorithm that produces a scalar λ and an array ϕ. PROMISE is tested on a 2D benchmark modelling a small nuclear reactor and requiring an array ϕ of size 20,000. MICADO internally uses an iterative solver, which stops when the relative difference between λ values at two successive iterations is less than 10^{-5}; this leads to 63 iterations. Since 99% of the execution time occurs in the solver, the PROMISE type optimization is performed only on the 51 floating-point type declarations of the solver and each type declaration in the tracker remains in double precision. The accuracy verification is performed on λ and ϕ with 5 different requirements. The results reported in Table 3 are the number of compilations, the number of executions, the type configurations and the total running times. Then for each configuration the speed-up and the memory gain is measured without CADNA with respect to the original code (each type in double precision).

As already observed in 5.2, with the stochastic reference version of PROMISE, configurations with more single precision variables are obtained than with the full stochastic version. For each accuracy requirement, the number of compilations and executions performed depends on the version of PROMISE. However even if fewer compilations and executions are observed with the full stochastic version, its execution time remains higher because of the cost of DSA.

From Table 3a, the execution time of the full stochastic version is about 3 to 5 hours. As a remark, configurations with the same number of single precision variables are proposed when 4 or 5 correct digits are requested. However the variables transformed in single precision are not the same. As a consequence, the number of compilations, executions, the speed-ups and the memory gains are different.

From Table 3b, the five executions of the stochastic reference version lead to three mixed precision configurations and two single precision ones. When PROMISE produces a mixed precision configuration the execution time is between 70 and 90 minutes. However, single precision configurations satisfying the accuracy requirements are quickly detected (in less than 10 minutes).

As expected, more single precision variables can be used when the accuracy requirements are reduced, which can lead to improvements in both execution time and memory usage. A detailed analysis of the mixed precision configurations generated by PROMISE when 8 or 10 correct digits are requested shows that many of the single precision declarations are located in parts of the code which are not extensively used in this benchmark. In those cases, the speed-up and the memory gain are limited and not significant. However the other configurations lead to a speed-up between 1.06 and 1.32 and a memory gain between 1.43 and 1.62.

A previous analysis of the MICADO solver [20] using SSE single precision instructions showed a speed-up of 1.52 compared to the execution time using scalar single precision instructions. Results presented here now confirm that the single precision configuration is accurate to 5 digits according to the stochastic reference version. This means that speed-ups shown in Table 3 can still be improved using SSE instructions.

PROMISE has been successfully tested on an industrial software, and leads to a performance enhancement. A next step would be to analyse larger benchmarks, that would generate more constraints in the type optimization.

6. Conclusion and perspectives

Taking into account a desired accuracy on the results, PROMISE maximises the number of single precision variables in programs having initially double precision declarations. With PROMISE the numerical quality of results is verified thanks to Discrete Stochastic Arithmetic. PROMISE has been successfully used on several codes including an industrial application. In all the tests performed, a new configuration is proposed with variables relaxed to a lower precision. The execution time of the programs considered in this paper has not always been improved. Nevertheless, the amount of memory required by the programs tested has decreased. Furthermore, the type configurations proposed could help developers get further speed-ups by vectorizing their algorithms. Indeed, SIMD instructions using single precision operands perform twice more operations in parallel than in double precision.

The current version of PROMISE provides a code where double precision declarations may be replaced by single precision ones. However, depending on the numerical stability of the code, an optimal type configuration could include `binary16` or `binary128` variables, these types being defined in the IEEE 754 standard [11]. The CADNA software could be modified to enable the estimation of round-off errors on such variables. This is a work in progress. Then PROMISE could be improved to take into account these types. From an initial program with the highest precision, several executions of the algorithm described in Section 3 could be performed with two precisions each time.

The execution time of PROMISE could be improved in several ways. First, the different partitions of the set of variables considered could be processed in parallel. Second, because of compilation errors, some configurations cannot be executed. Such errors are mainly due to type differences in array arguments between the prototype of functions and their calls. A static analysis of the code could enable one to determine links between some numerical types. This analysis would reduce the number of compilation errors and, as a consequence, it would improve the execution time of PROMISE.

References

[1] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, S. Tomov, Accelerating scientific computations with mixed precision algorithms, Computer Physics Communications 180 (2009) 2526 – 2533.

[2] J. Brajard, P. Li, F. Jézéquel, H.S. Benavidès, S. Thiria, Numerical Validation of Data Assimilation Codes Generated by the YAO Software, in: SIAM Annual Meeting, San Diego, California (USA).

[3] J.M. Chesneaux, L'arithmétique stochastique et le logiciel CADNA, Habilitation à diriger des recherches, Université Pierre et Marie Curie, Paris, France, 1995.

[4] J.M. Chesneaux, J. Vignes, Les fondements de l'arithmétique stochastique, Comptes Rendus de l'Académie des Sciences - Series I - Mathematics 315 (1992) 1435–1440.

[5] Contributors, GSL Project, GSL - GNU scientific library - GNU project - free software foundation (FSF), http://www.gnu.org/software/gsl/, 2010.
[6] Contributors of Center for Manycore Programming, Seoul, SNU NPB Suite, 2010.
[7] P. Eberhart, J. Brajard, P. Fortin, F. Jézéquel, High performance numerical validation using stochastic arithmetic, Reliable Computing 21 (2015) 35–52.
[8] F. Févotte, B. Lathuilière, MICADO: Parallel implementation of a 2D–1D iterative algorithm for the 3D neutron transport problem in prismatic geometries, in: Proceedings of Mathematics, Computational Methods & Reactor Physics.
[9] S. Graillat, F. Jézéquel, S. Wang, Y. Zhu, Stochastic arithmetic in multiprecision, Mathematics in Computer Science 5 (2011) 359–375.
[10] M. Halsall, CACTUS, a characteristics solution to the neutron transport equations in complicated geometries, AEEW / R: AEEW, Atomic Energy Establishment, Winfrith, 1980.
[11] IEEE Computer Society, IEEE Standard for Floating-Point Arithmetic, IEEE Standard 754-2008, 2008.
[12] F. Jézéquel, J.M. Chesneaux, CADNA: a library for estimating round-off error propagation, Computer Physics Communications 178 (2008) 933–955.
[13] F. Jézéquel, J.L. Lamotte, O. Chubach, Parallelization of Discrete Stochastic Arithmetic on multicore architectures, in: 10th International Conference on Information Technology: New Generations (ITNG), Las Vegas, Nevada (USA).
[14] F. Jézéquel, J.L. Lamotte, I. Said, Estimation of numerical reproducibility on CPU and GPU, in: Annals of Computer Science and Information Systems, Proceedings of the 2015 Federated Conference on Computer Science and Information Systems (FedCSIS), volume 5, pp. 675–680.
[15] F. Jézéquel, F. Rico, J.M. Chesneaux, M. Charikhi, Reliable computation of a multiple integral involved in the neutron star theory, Math. Comput. Simulation 71 (2006) 44–61.
[16] M.O. Lam, J.K. Hollingsworth, B.R. de Supinski, M.P. Legendre, Automatically Adapting Programs for Mixed-precision Floating-point Computation, in: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS '13, ACM, New York, NY, USA, 2013, pp. 369–378.
[17] J.L. Lamotte, J.M. Chesneaux, F. Jézéquel, CADNA_C: A version of CADNA for use with C or C++ programs, Computer Physics Communications 181 (2010) 1925–1926.
[18] C. Lattner, V. Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, in: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California.
[19] W. Li, Numerical accuracy analysis in simulations on hybrid high-performance computing systems, Phd thesis, Stuttgart University, Germany, 2012.
[20] S. Moustafa, F. Févotte, B. Lathuilière, L. Plagne, Vectorization of a 2D-1D Iterative Algorithm for the 3D Neutron Transport Problem in Prismatic Geometries, in: Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013 (SNA + MC 2013).
[21] C. Nguyen, C. Rubio-González, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D.H. Bailey, D. Hough, Floating-Point Precision Tuning Using Blame Analysis, Technical Report, LBNL TR, 2015.
[22] C. Rubio-González, C. Nguyen, H.D. Nguyen, J. Demmel, W. Kahan, K. Sen, D.H. Bailey, C. Iancu, D. Hough, Precimonious: Tuning assistant for floating-point precision, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC'13, ACM, New York, NY, USA, 2013, pp. 27:1–27:12.
[23] S.M. Rump, Reliability in Computing: The Role of Interval Methods in Scientific Computing, Academic Press, Boston, 1988, pp. 109–126.
[24] N. Scott, F. Jézéquel, C. Denis, J.M. Chesneaux, Numerical 'health check' for scientific codes: the CADNA approach, Computer Physics Communications 176 (2007) 507–521.
[25] J. Vignes, Zéro mathématique et zéro informatique, Comptes Rendus de l'Académie des Sciences - Series I - Mathematics 303 (1986) 997–1000.
[26] J. Vignes, A stochastic arithmetic for reliable scientific computation, Mathematics and Computers in Simulation 35 (1993) 233–261.
[27] J. Vignes, Discrete Stochastic Arithmetic for validating results of numerical software, Numerical Algorithms 37 (2004) 377–390.
[28] A. Zeller, Why Programs Fail, second ed., Morgan Kaufmann, Boston, 2009.
[29] A. Zeller, R. Hildebrandt, Simplifying and isolating failure-inducing input, IEEE Trans. Softw. Eng. 28 (2002) 183–200.