Detection of Silent Data Corruption in Adaptive Numerical Integration Solvers

Pierre-Louis Guhur,*† Emil Constantinescu,* Debojyoti Ghosh,‡ Tom Peterka,* Franck Cappello*
*Argonne National Laboratory (USA) {emconsta, tpeterka, cappello}@mcs.anl.gov
†ENS Paris-Saclay (France) pierre-louis.guhur@ens-paris-saclay.fr
‡Lawrence Livermore National Laboratory (USA) ghosh5@llnl.gov

***Abstract*—Scientific computing requires trust in results. In high-performance computing, trust is impeded by silent data corruption (SDC), in other words corruption that remains unnoticed. Numerical integration solvers are especially sensitive to SDCs because an SDC introduced in a certain step affects all the following steps. SDCs can even cause the solver to become unstable. Adaptive solvers can change the step size, by comparing an estimation of the approximation error with an user-defined tolerance. If the estimation exceeds the tolerance, the step is rejected and recomputed. Adaptive solvers have an inherent resilience, because some SDCs might have no consequences on the accuracy of the results, and some SDCs might push the approximation error beyond the tolerance. Our first contribution shows that the rejection mechanism is not reliable enough to reject all SDCs that affect the results' accuracy, because the estimation is also corrupted. We therefore provide another protection mechanism: at the end of each step, a second error estimation is employed to increase the redundancy. Because of the complex dynamics, the choice of the second estimate is difficult: two methods are explored. We evaluated them in HyPar and PETSc, on a cluster of 4,096 cores. We injected SDCs that are large enough to affect the trust or the convergence of the solvers. The new approach can detect 99% of the SDCs, reducing by more than 10 times the number of undetected SDCs. Compared with replication, a classic SDC detector, our protection mechanism reduces the memory overhead by more than 2 times and the computational overhead by more than 20 times in our experiments.**

***Index Terms*—high-performance computing, resilience, fault tolerance, silent data corruption, numerical integration solver**

I. Introduction

Several reports [1], [2], [3], [4] highlight that many scientific applications suffer from corruption without any notification from the firmware or the operating system. Consequences of these corruptions, called silent data corruptions (SDCs), are worrisome: loss of data, untrustworthy results [5], or cascading patterns of corruption [6]. Their sources are also wide, ranging from electromagnetic interference [7] to aging of hardware components. Moreover, the situation is expected to worsen for the next generation of supercomputers: Snir et al. [8] estimated that the SDC rate is likely to increase in future exascale systems.

Resilience to SDC is defined as the ability of a system to achieve its design purpose even in presence of SDCs. In this study, we focus on the resilience of numerical integration solvers. These solvers provide an approximate solution of a differential equation. They are widely used in many different contexts, such as acoustics, heat transfer [9], fluid dynamics, weather prediction, and quantum mechanics [10]. In previous work [11], we showed that solvers can remain unaffected by some SDCs, while other SDCs, which we refer to as significant, impair the user's accuracy expectation and may render the solution unstable. Solvers are particularly sensitive to significant SDCs because an SDC introduced at a certain step might have consequences on the following ones. Solvers are classified into fixed solvers and adaptive solvers. Fixed solvers have a constant step size. At the end of each step, adaptive solvers estimate an approximation error by subtracting two approximated solutions, one of which is less accurate than the other one. This estimate is then compared with user-defined thresholds. This approach allows controlling the step size or even rejecting some steps. In the presence of SDCs, this estimate is corrupted, however, and might be underestimated. Corrupted steps might not be rejected, while the step size might be increased beyond stability bounds.

This paper provides a resilient and lightweight mechanism for detecting SDCs in adaptive solvers. More specifically, the key contributions are the following.

- We show that the rejection mechanism of adaptive solvers is unable to reject all significant SDCs.
- We derive a lightweight and robust mechanism that uses two different strategies for checking the validation of a step.
- The first strategy is inspired by the state-of-the-art *adaptive impact-driven* (AID) SDC detector [12].
- The second strategy computes another estimate of the approximation error based on a second numerical integration solver.
- Because finding two error estimates that agree is difficult for adaptive solvers, we provide an algorithm that automatically selects the best estimate.
- We measure the resilience properties of our mechanism in a high-performance computing (HPC) application with two scientific libraries: HyPar [13], a hyperbolic-parabolic differential equation solver, and PETSc [14], [15], [16], a scalable toolkit for partial differential equations.

The remainder of this paper is organized as follows. In Section II we describe the model of SDCs that we consider and the high-performance computing application used for

our experiments. In Section III we explain how a numerical integration solver works. In Section IV we show that adaptive solvers can naturally reject only a part of the significant SDCs. In Section V we detail our double-checking method and present its implementation. Experiments are described in Section VI. In Section VII we discuss related work, and in Section VIII we draw our conclusions.

II. Detection of Silent Data Corruption

We describe here the SDC model and the application we used.

A. Silent Data Corruption Model

A silent data corruption occurs when a program provides an output that is correctly formatted but is unexpected. For example, in the case of the Pentium FDIV bug [17], $4195835/3145727$ provided 1.333820449136241002 instead of 1.333739068902037589: final outcomes were wrong, but the error might not be detected. In memory, a corruption is more likely to occur in data than in instructions because instructions occupy less memory than data do, and corrupted instructions typically result in crashes and not silent corruptions. At exa-scale, corruptions will likely happen in latches and flip-flops of processors [8]. Other mechanisms besides SDC detection, such as checkpointing, may be employed for protecting an execution against instruction corruptions. We assume here that corruptions affect only data.

An SDC is called *nonsystematic* when it affects a program randomly. Such SDCs typically are triggered when radiation or aging hardware flips one or several bits [18]. On the contrary, a *systematic* corruption is triggered by a repeatable pattern such as a bug, as in the case of the Pentium FDIV bug. The probability of SDCs is low, and it is unlikely that two nonsystematic SDCs occur two times consecutively in the same bits. Therefore, correction can be obtained by recomputing a corrupted step to recover from a nonsystematic SDC. This prevents the execution from infinite recovery loops. In this study, we consider only nonsystematic SDCs.

B. Consequences of SDCs in Numerical Integration Solvers

Numerical integration solvers are particularly sensitive to SDCs: because of the iterative scheme that progresses by successive steps, an SDC affects not only the corrupted step but also the following steps. We illustrate this sensitivity with two examples.

- In nonlinear ordinary differentiel equations (ODEs), the stability region of the ODE method depends on the current step. An SDC can bring the solution outside the stability region. For example, in the equation $\frac{dx}{dt} = (x-1)^2$, an initial point greater than 1 diverges to infinity, while an initial point less than 1 converges to 1.
- Even though the corruption is silent in the solver, it can produce corrupted results in the next stages of the application's workflow. For example, in image processing, feature extraction can be based on solving a partial differential equation (PDE) as shown by Zhou et al. [19]. If the PDE solution is incorrect, the iterative process of level set evolution may not converge.

Solvers require several function evaluations (defined in Section III) to compute a step. Corruptions are more likely to affect the function evaluations because those are the most computationally expensive part of a solver.

C. Objectives of Our SDC Detector

Replication is a generic solution for detecting all nonsystematic SDCs. Hence, a new SDC detector should have lower memory and/or computational overhead than does replication. For a numerical integration solver, an SDC detector is a function of the last steps and last function evaluations. Minimizing the computational overhead means computing as few additional operations as possible. Minimizing the memory overhead is equivalent to storing as little extra data as possible.

Correction can be achieved by recomputing a step that is detected as corrupted. A detector can do a false positive when it asked a noncorrupted step to be recomputed. False positives waste resources; minimizing the overheads requires maintaining the number of false positives at a low level.

D. Components

The numerical integration solvers represent one step in a scientific application. Section II-D shows an overview of a typical HPC workflow composed of a resilient numerical integration solver. The SDC detection is done at each step. When a step is found to be corrupted, it is recomputed in order to allow the solver to continue.

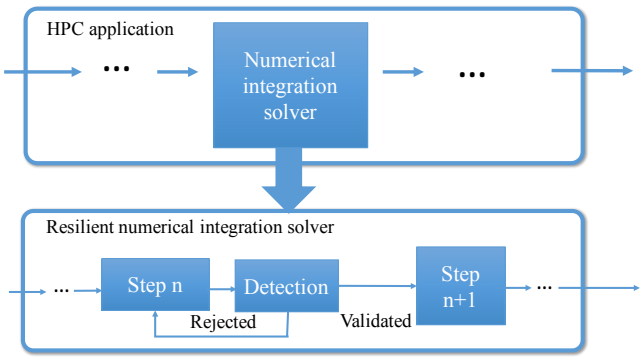

Figure 1. SDC detector for an HPC application with an iterative numerical integration solver. At the end of each step, the SDC detector decides whether to validate or reject the step.

E. Modeling SDC

We consider here SDCs that occur randomly on data. For mathematical discussions, we model an SDC as a random variable ϵ added to a deterministic variable X that is part of the iterative solver. If X^o and X^c are resp. the noncorrupted and corrupted value of X, then $X^o = X$, and $X^c = X^o + \epsilon$. The letter c stands for corrupted and o for original.

Concerning simulations, recent papers on SDC detections propose different ways to inject SDCs. We preferred, therefore,

being exhaustive in our simulations. On the one hand, in several papers [20], [21], injections were done by randomly flipping bits in data items. In the following, we refer to *singlebit* SDCs when one bit is flipped inside a data item, or *multibit* SDCs when several bits are flipped. For example, the IEEE 754 half-precision representation of 1, 0011110000000000, might become $0111110000000000 = \infty$ with a singlebit SDC or $0000010000000000 = 2^{-14}$ with a multibit SDC. The number of bit-flips in multibit SDCs is drawn from a uniform distribution. In our previous work [11], we compared several probability distributions to choose the position of the bit-flip, and we noticed that uniform distribution provides similar results to other distributions. Although multibit SDCs might seem less likely than singlebit SDCs, they are also not protected by error-correcting code memory [22].

On the other hand, a bit-flip on lowest-order positions may not have an impact on the results, whereas a bit-flip in highest-order positions may crash the application or be easy to detect. Consequently, Benson et al. [23] simulated SDC injections by multiplying a data item with a random factor. The factor is drawn from a normal distribution with zero mean and unit variance. We refer to this method as *scaled injections*.

In both cases, we injected SDCs on some function evaluations. Injecting SDCs on all function evaluations would create unrealistic cascading patterns. A function evaluation was thus corrupted with a probability of $1/100$, but a lower probability does not affect the detection performance. In every experiments at least $10,000$ SDCs were injected to provide statistically significant detection performance.

F. Simulations

Our numerical experiments used HyPar [13], a high-order, conservative finite-difference solver for hyperbolic-parabolic PDEs. We also use the time integrators (ODE solvers) implemented in PETSc [14], [15], [16], a portable and scalable toolkit for scientific applications. HyPar and PETSc are written in C and use the MPICH library on distributed computing platforms.

The use case solves the problem of a rising warm bubble in the atmosphere. This problem is used as a benchmark for atmospheric flows [24], [25]. The governing equations are the three-dimensional nonhydrostatic unified model of the atmosphere [26].

The use case is solved with HyPar. The domain is discretized on equispaced Cartesian grids. For solving the hyperbolic-parabolic PDEs, the finite-difference methods, called the fifth-order WENO [27] and CRWENO [28] schemes, were used to compute the spatial derivatives. This computation results in an ODE in time that is solved by using an ODE solver implemented in PETSc. SDCs were injected inside the ODE solver, but they represent also corruptions that could occur inside HyPar. In the following, several ODE solvers are compared: Heun-Euler, Bogacki-Shampine and Dormand-Prince methods [29]. They have an increasing accuracy but also an increasing memory and computational cost.

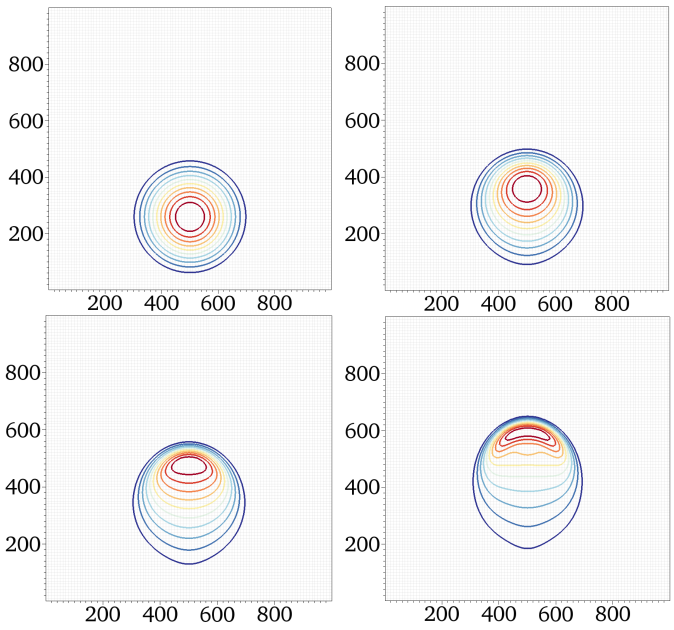

Figure 2. Rising thermal bubble: Density perturbation (ρ') contours at 0 s (initial), 100 s, 150 s, and 200 s (final). Ten contours are plotted between -0.0034 (red) and -0.0004 (blue). The cross-sectional profile is shown at $y = 500$ m.

Figure 2 shows the density perturbation contours for the rising thermal bubble case at 0 s, 100 s, 150 s, and 200 s, solved on a grid with 64^3 points. The bubble rises as a result of buoyancy and deforms as a result of temperature and velocity gradients.

The experiment was done on the Blues cluster at Argonne National Laboratory. The cluster is composed of 310 compute nodes, 64 GB of memory on each node, 16 cores per compute node with the microarchitecture Intel Sandy Bridge and a theoretical peak performance of 107.8 TFlops. PETSc is configured with MVAPICH2-1.9.5, shared libraries, 64-bit ints, and O3 flag.

G. Detection Performance

Detection performances are based on the false positive rate and the true positive rate. The false positive rate (FPR) is defined as the ratio between the number of noncorrupted steps that are rejected and the number of noncorrupted steps. Similarly, the true positive rate (TPR) is the ratio between the number of corrupted and rejected steps and the number of corrupted steps. The false negative rate (FNR) is the ratio between the number of accepted but corrupted steps and the number of corrupted steps.

III. Background and Context

We introduce here the notion of local truncation error (LTE) that is the basis of our new detection method. We also introduce adaptive solvers that are the targets of this work.

A. Numerical Integration Solvers

1) Differential equation: Our study focuses on numerical integration solvers. These solvers approximate the integration of a differential equation. They are iterative, time-stepping methods. For stiff problems, namely, problems that are numerically unstable, the dynamics are so complex that basic properties such as extrapolation, as used in [12], [23], provide limited SDC detection. If the differential equation contains one independent variable, it is called an ODE, whereas with multiple independent variables it is called a PDE. A PDE may be solved with the method of lines, where all but one variable is discretized. In this way, a PDE is solved by solving several ODEs. In this paper, we consider an ODE method, that solves an initial value problem formulated as

$$x'(t) = f(t, x(t)), x(t_0) = x_0,$$

with $t_0 \in \mathbb{R}, x_0 \in \mathbb{R}, x : \mathbb{R} \to \mathbb{R}^m$, and $f : \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m$. [1]

ODE methods approximate the exact solution of the ODE $x(t_n)$ into x_n, with $n \in 1, ..., N$, $t_n = t_0 + nh$, and $h \in \mathbb{R}^*_+$ where the step size.

ODE methods can be explicit or implicit. Explicit methods compute the step n from previous steps, whereas implicit methods also use the current step n. Implicit methods require solving a system of equations. This extra computation is worthwhile when implicit methods can use larger step sizes than explicit methods can. This is the case for stiff problems.

ODE methods are composed of several terms that are computed from the differential equations. We denote those terms $(K_i)_i$. For example, in explicit Runge-Kutta methods,

$$x_{n+1} = x_n + h \sum_{i=1}^{s} b_i K_i$$
$$\forall i \leq s, K_i = f\left(t_n + c_i h, x_n + h \sum_{j=1}^{i-1} a_{ij} f_{n,j}\right).$$

The coefficients $(a_{ij})_{ij}, (b_i)_i, (c_i)_i$ are given by the methods.

2) Control of the approximation error: Numerical integration solvers produce inherent approximation errors. The LTE is the absolute difference between the approximation error introduced at a step $n+1$ and the exact solution started at step n, whereas the global truncation error (GTE) is the absolute difference between difference between the approximation error introduced at a step $n+1$ and the exact solution started at the first step. Given the step size h, an ODE method is said to have an order p if the LTE at step n is $LTE_n = O(h^{p+1})$ and the global truncation error at the last step N is $GTE_N = O(h^p)$.

The choice of step size is a difficult trade-off: with a decreasing step size, the approximation error is decreased; but more steps increase the computational time. The step size cannot exceed a certain region of stability, which depends on the function f and the employed ODE method. Adaptive solvers differ from fixed solvers in that the step size varies according to an *error estimate* that is an estimation of the GTE or LTE. For performance reasons, most of the solvers change the step size only with an estimation of the LTE.

3) Assumption: In the absence of SDCs, we assume that the solver works well. This means that it converges in a limited number of steps and achieves the user's accuracy expectations.

B. Design of Adaptive Solvers

In the case of an adaptive solver, the user explicitly states the maximum acceptable approximation error with a desired absolute $\mathrm{Tol_A}$ or a relative $\mathrm{Tol_R}$ error tolerance. $\mathrm{Tol_A}$ is used to control the error for small values of $||x_n||$, and $\mathrm{Tol_R}$ for larger values. In practice, the error estimate is based on the LTE, so for every step the algorithm verifies that the estimated local truncation error satisfies the tolerances provided by the user and suggests a new step size to be taken.

The adaptive controller at step n forms the error level Err_n and the scaled error $SErr_n$ as

$$Err_n = \mathrm{Tol_A} + ||x_n|| \mathrm{Tol_R},$$
$$SErr_n = m^{\frac{1}{q}} \left\| \frac{|x_n - \tilde{x}_n|}{Err_n} \right\|_q,$$

where the errors are computed componentwise, m is the dimension of x, and q is typically 2 or ∞ (max norm). The error tolerances are satisfied when $SErr_n \leq 1.0$.

1) Estimating the local truncation error: Usually, estimation of the LTE consists of subtracting x_n with an approximation of it, $\tilde{x}_n$:

$$x_n - \tilde{x}_n = x_n - u(t_n, n-1) - (u(t_n, n-1) - \tilde{x}_n), \quad (1)$$
$$= LTE[x]_n - LTE[\tilde{x}]_n. \quad (2)$$

Although estimates based on Richardson's extrapolation can be employed [30], the estimation is generally based on an embedded method. Embedded methods compute at each step two results at two different orders p and q: x_n^p and x_n^q (in general $|q-p| = 1$). The solution is propagated by one of these results, while the second result provides the approximation $\tilde{x}_n$ that is used to compute an estimate of the LTE at step n. If q is at a higher order than LTE^p, then the difference between x_n^p and x_n^q is an estimate of the LTE of x_n^p:

$$x_n^p - x_n^q = LTE[x^p]_n - LTE[x^q]_n \quad (3)$$
$$= LTE[x^p]_n + O(h^{q+1}). \quad (4)$$

2) Control of error estimation: Based on this error estimate, in practice the step size that would satisfy the tolerances is

$$A_{n+1} = \alpha (1/SErr_{n+1})^{\frac{1}{p+1}},$$
$$h_{\mathrm{new}}(t_n) = h_{\mathrm{old}}(t_n) \min(\alpha_{\max}, \max(\alpha_{\min}, A_{n+1})), \quad (5)$$

where $\alpha_{\min}$ and $\alpha_{\max}$ keep the change in h to within a certain factor. We impose $\alpha < 1$ so that there is some margin for which the tolerances are satisfied and so that the probability of rejection is decreased in the SDC-free case.

In this study we use the following settings: $\alpha = 0.9$, $\alpha_{\max} = 10$, $\alpha_{\min} = 0.1$, and $q = 2$. These are usually employed for adaptive solvers and are the default settings of PETSc. Therefore, the scaled error is $SErr = \sqrt{\frac{1}{n} \sum_n \frac{|x - \tilde{x}|^2}{Err^2}}$, and the step size is adjusted as $h_{\mathrm{new}} = h_{\mathrm{old}} \min(10, \max(0.1, 0.9A))$.

[1] f is L-Lipschitz continuous.

3) Scheme of an adaptive controller: The adaptive controller works in the following way. After completing step n, if $SErr_n \leq 1.0$, then the step is accepted, and the next step is modified according to (5); otherwise the step is rejected and retaken with the step length computed in (5).

IV. Resilience of Adaptive Controllers

Chen et al. [31] observed that some solvers have an inherent resilience. In the following, we extend this point to all adaptive solvers. Experimentally, we observe that the adaptive controller rejects some steps where the error estimate exceeds a certain threshold due to an SDC.

However, this assumes that the adaptive controller is not corrupted in the presence of an SDC. This assumption does not hold because the error estimation used by the adaptive controller is computed from corrupted results. In Section IV-B, we observe that the error estimate can be shifted under the threshold of the adaptive controller and leave the SDC-affected step unrejected.

A. Inherent Resilience

1) Significant and insignificant SDCs: Numerical integration solvers have an inherent approximation error depending on the integration method and its order p: the GTE is $O(h^p)$. When the lowest-order bit is flipped, the impact is insignificant with respect to the approximation error, and this SDC does not affect the accuracy of the results. Basically, we call *insignificant* any SDC that does not affect the user's expectation in accuracy. Other SDCs affect higher-order bits, and then they drastically increase the error or may even cause the solver to diverge. These SDCs are referred to as *significant*.

Distinguishing significant and insignificant SDCs in the general case is difficult. In our previous work on fixed solvers [11], the user did not give an explicit expectation of accuracy, and we considered that any SDC higher than a tenth of the LTE was significant. In the context of adaptive solvers, we can consider a corruption is significant when the scaled error is above 1.0. More precisely, we measure the LTE when a step is corrupted. Then, we recompute the step in order to know what would have been the LTE without corruptions. When the scaled error by the tolerances is drifted above 1.0, the step is rejected and the corruption is considered significant. Later, we compute detection performances also for significant SDCs. For example, the significant false negative rate is the ratio between the number of steps that are accepted but corrupted with a significant SDCs, and the number of corrupted steps with a significant SDCs.

2) Rejection of corrupted steps: In Section II, we saw that an error estimate exceeding the tolerances Tol_A and Tol_R is rejected because the approximation error is considered unacceptable for the user.

When an SDC occurs and the approximation error is shifted outside the tolerance because of the SDC, the step is naturally rejected. In this case, the step size is reduced according to equation (4); then the next noncorrupted step observes that the error estimate is too small and increases the step size. Overall, the computation time is increased just during one step, while the accuracy is preserved.

The corrupted step is not rejected in two cases: if the SDC shifts the approximation error below the tolerance or if the SDC is small enough to avoid the approximation error's exceeding the tolerance. Accepting such steps seems dangerous, however. One could object that the approximation error can be higher than it would have been without the SDC; even if the current step is below the tolerance, it might affect the next steps. Nevertheless, an adaptive solver is designed in a way that if all steps are below the tolerances, then the expected accuracy is achieved. Accepting such corrupted steps might increase the approximation error on the next steps, but the expected accuracy will be achieved.

One caveat must be added. The approximation error is only estimated. In the presence of an SDC, the estimate is also corrupted, and its value might differ from the real value of the approximation error. This case is considered in Section IV-B.

We injected SDCs in the use case introduced in Section II. In Table I, we show the detection performances of the adaptive controller.

The false positive rate remains below 0.1% for all considered ODE methods. At the same time, the true positive rate is usually below 50%. Singlebit SDCs are the hardest SDCs to detect (9.3%), whereas multibit SDCs are the easiest (55.1%). The reason is that singlebit SDCs usually have a lower impact on the results, as explained in Section II.

The ODE methods differ in their number N_k of the function evaluations $(K_i)_i$ (for example, $N_k = 7$ for the Dormand-Prince method, $N_k = 4$ for the Bogacki-Shampine method, and $N_k = 2$ for the Heun-Euler method). The true positive rate decreases with the order of an ODE method.

The true positive rate may seem low, but only significant SDCs need to be rejected. Further experiments must thus distinguish significant from insignificant SDCs in order to know whether the inherent resilience of adaptive solvers is reliable enough.

Rate	Injector	Heun-Euler	Bogacki-Shampine	Dormand-Prince
FP	All	0.0	0.0	0.0
TP	Multibit	55.1	46.8	35.3
TP	Singlebit	13.2	11.8	9.3
TP	Scaled	31.1	23.3	20.1

Table I. Detection accuracy of several ODE methods and several SDC injectors. FP: false positive. TP: true positive. Results are given in percentage.

B. Significant SDCs Not Detected

The approximation error is not precisely known but is only estimated. In the presence of a corruption, the estimate is also corrupted. In particular, it may be shifted below the tolerances of the adaptive controller; in such a case, the step would be accepted. We give several examples.

- In the extreme case, the memory of $(K_i)_{i\geq 0}$ could be reset. In this case, the corrupted error estimate is equal to zero; consequently the step is accepted, and the step size is increased by α_{max}. The solution would be the same

as during the last step: $x_n = x_{n-1}$. The approximation error could then be unacceptable with respect to the user's requirements.
- Because any K_i depends on other $(K_j)_{j \neq i}$, the corruption of a certain K_l corrupts the other $(K_j)_{j \neq l}$. Such cascading patterns increase the possibility of underestimating the approximation error.
- The SDCs can affect only the estimate. In this case, the estimate can be completely decorrelated from the approximation error.

Consequences of accepting a corrupted step can be disastrous. Not only will the corrupted step exceed the user's accuracy expectation, but the next steps will be initialized with a corrupted result. Moreover, the step size might be increased after the corrupted step, and it might even exceed its stability region; in such a case, the solution may not converge at all.

In our use case, we observe that this phenomenon can occur with a random corruption. We report rates for significant corruptions, namely, steps whose real scaled LTE is higher than 1.0. The real scaled LTE is computed from the difference between the corrupted solution x_n^c and a noncorrupted approximation solution $\tilde{x}_n^o$. We report the false negative rate of those significant SDCs, called the significant false negative (SFN) rate. We recall that the false negative rate is equal to 1 minus the true positive rate. In Table II, we show the false negative rate of the adaptive controller. The false negative rate with all steps is higher than the significant false negative rate, because insignificant SDCs can have too low an impact on the results to be detectable.

While the SFN rate achieves 13.3% for the Heun-Euler method with scaled SDCs, the rate increases dramatically to 50.4% for the Dormand-Prince method. The reason is that the number N_k of function evaluations $(K_i)_i$ is higher for the Dormand-Prince method. In this case, more patterns of SDCs can lead to an underevaluation of the error estimation, and the probability of nondetection is thus higher. While the false negative rate with all steps is higher with singlebit SDCs than with scaled SDCs, the significant false negative rate is lower with singlebit SDCs than with scaled SDCs. The reason is that a singlebit SDC becomes significant when one of the highest-order bits is flipped. This is easily detectable, whereas a scaled SDC can be significant while being difficult to detect.

Injector	Heun-Euler		Bogacki-Shampine		Dormand-Prince	
	All	Sign.	All	Sign.	All	Sign.
Singlebit	86.8	5.4	88.2	10.1	90.7	15.0
Multibit	44.9	3.9	53.2	4.5	64.7	7.9
Scaled	68.9	13.3	26.7	36.1	79.9	50.4

Table II. False negative rate for several ODE methods and several SDC injectors. Sign. = significant (only steps that are corrupted with at least one significant SDC are considered). Results are given in percentage.

V. Resilience Method for Adaptive Solvers

We saw that adaptive solvers use an estimate to reject or accept a step. In the presence of SDCs, adaptive solvers can underestimate the approximation error because the estimator is using corrupted data; in this case, the adaptive solvers may not reject all significant SDCs. To address this issue, we increase the reliability of the rejection mechanism by adding a second acceptance step. When the adaptive controller accepts a step, we apply a different rejection mechanism to validate the decision. We use this additional mechanism because the rejection mechanism could be underevaluated following its own pattern of corruptions. By selecting two rejection mechanisms with different patterns, the risk of nondetection of a significant SDCs is reduced. We call our method *double-checking*.

We explore here two approaches for computing the double-checking. Both of them compute an extra estimate of the approximation error and then compare the estimate to a threshold function. The step is originally computed by the method with the extra estimate. If the difference exceeds a threshold, the step will be rejected. In order to compute the extra estimate, the first approach uses Lagrange interpolating polynomials as presented in Section V-A, whereas the second approach considers an estimate based on another ODE method as explained in Section V-B. Thereafter, we denote LTE_1 the error estimate's vector from the original rejection mechanism, whereas LTE_2 is the error estimate's vector used by the double-checking.

A. Double-Checking Based on Lagrange Interpolating Polynomials

The SDC detector AID (presented in Section VII) uses an extrapolation method to compute an approximation $\tilde{x}_n$ of the solution x_n. The extrapolation method is computed from one, two, or three previous steps, but it cannot be computed when the step size is variable. Instead, we compute an approximation of the solution using Lagrange interpolating polynomials (LIPs). We provide formulations for order 0, 1, and 2:

$$
\begin{aligned}
\tilde{x}_n^0 &= x_{n-1}, \\
\tilde{x}_n^1 &= x_{n-1}\frac{h_n + h_{n-1}}{h_{n-1}} - x_{n-2}\frac{h_n}{h_{n-1}}, \\
\tilde{x}_n^2 &= x_{n-1}\frac{(h_n + h_{n-1})(h_n + h_{n-1} + h_{n-2})}{h_{n-2}(h_{n-2} + h_{n-1})} \\
&\quad - x_{n-2}\frac{h_n(h_n + h_{n-1} + h_{n-2})}{h_{n-2}h_{n-1}} \\
&\quad + x_{n-3}\frac{h_n(h_n + h_{n-1})}{h_{n-2}(h_{n-1} + h_{n-2})}.
\end{aligned}
$$

In AID, the difference between the approximated and the computed solution $x_n - \tilde{x}_n$ is compared with a threshold based on a user-defined constant. If the difference exceeds the threshold, the solution is considered corrupted. Using a user-defined constant requires a manual setting. Instead, we compute the scaled error $SErr_2$ (defined in Section III-B) from x_n and $\tilde{x}_n$ in the context of an adaptive solver; the step is rejected when $SErr_2$ is higher than 1.0. We call this method *LIP-based double-checking*.

B. Integration-Based Double-Checking

Our second approach consists of computing another approximation of the solution $\tilde{x}_n$ based on a different ODE method from the one used in the solver. In order to reach a low computational overhead, it must not require extra computations. It must also have a larger stability area than the ODE method used by the first method. Because implicit methods usually have a larger stability area than explicit methods have, the latter condition can be followed by employing an implicit method for the double-checking and an explicit method for the solver. Similarly, the scaled error $SErr_2$ is computed from x_n and $\tilde{x}_n$, and the step is rejected when $x_n - \tilde{x}_n > 1.0$. This method is called *integration-based double-checking*. We suggest employing a backward differentiation formula (BDF) for the double-checking because it uses previous computations and has a large stability area. We compute the estimates by storing $(x_{n-k})_{k\geq 0}$. One could also use an Adam-Moulton method: it requires storing $f(t_{n-k}, x_{n-k})$ instead, although it often appears less practical. BDF are multistep and implicit methods. In the literature, several expressions for a variable step size are given. In the following, we will use the expressions of $\tilde{x}_n^1$, $\tilde{x}_n^2$, and $\tilde{x}_n^3$ for the orders 1, 2, and 3:

$$
\begin{aligned}
\tilde{x}_n^1 &= x_{n-1} + hf(x_n),\\
\tilde{x}_n^2 &= \frac{(1+\omega_n)^2}{1+2\omega} x_{n-1} - \frac{\omega_n^2}{1+2\omega} x_{n-2} + hf(x_n),\\
\tilde{x}_n^3 &= h_n \frac{(w_n+1)\left(w_n w_{n-1} + w_{n-1} + 1\right)}{3w_{n-1}w_n^2 + 4w_{n-1}w_n + 2w_n + w_{n-1} + 1} f\left(x_n\right)\\
&+ \frac{\left(w_n+1\right)^2 \left(w_{n-1}\left(w_n+1\right)+1\right)^2}{\left(w_{n-1}+1\right)\left(2w_n + w_{n-1}\left(w_n+1\right)\left(3w_n+1\right)+1\right)} x_{n-1}\\
&- \frac{w_n^2 \left(w_{n-1}\left(w_n+1\right)+1\right)^2}{2w_n + w_{n-1}\left(w_n+1\right)\left(3w_n+1\right)+1} x_{n-2}\\
&+ \frac{w_n^2 \left(w_n+1\right)^2 w_{n-1}^3}{\left(w_{n-1}+1\right)\left(2w_n + w_{n-1}\left(w_n+1\right)\left(3w_n+1\right)+1\right)} x_{n-3},
\end{aligned}
$$

where $\omega_n = \frac{h_n}{h_{n-1}}$ and $\omega_{n-1} = \frac{h_{n-2}}{h_{n-1}}$.

BDF methods have expressions until order 6, but the stability area decreases with the order. At the same time, ODE methods with a small order require less computation and less storage of previous solutions $(x_{n-k})_{k\geq 0}$. In this study, we restrict our work to orders 1, 2, and 3 as to avoid stability issues and to mitigate the overheads. By employing previous solutions $(x_{n-k})_{k.0}$ and the current solution x_n computed by the ODE method, BDF requires only the computation of $f(x_n)$. For most ODE methods, however, $f(x_n)$ is used for the next step. In this case, there is no extra computation when the step is accepted. Certain ODE methods, called first-same-as-last, compute $f(x_n)$ at step n; the Dormand-Prince method is an example. Consequently, first-same-as-last methods require a lower computational overhead for computing the extra estimate.

C. Difficulties in Gathering Two Different Estimates

While estimates of the approximation error provide similar results in fixed solver, they differ significantly in adaptive solvers.

The estimation of the approximation error uses solutions computed at order p. Thus, the error estimate does not exceed an accuracy higher than $O(h^{p+1})$, even if the second ODE method is expressed at an higher order $q > p$. However, $\tilde{x}_n$ tends to be more similar to x_n with an higher value of q. Consequently, the higher q is, the smaller the error estimate tends to be. It makes the detection less sensitive: the second error estimate is less likely to be higher than 1.0, and fewer steps tend to be rejected. Also, the number of false positives decreases: fewer noncorrupted steps are rejected.

Because we want to improve the detection while maintaining a low false positive rate, we adapt the order of the ODE method. We define two constants: γ and Γ. In our experiments, we take $\gamma = 0.05$, $\Gamma = 0.1$, and $q_{max} = 3$. When the false positive rate is higher than γ for an order q, a formula with one higher order $q' \leq q_{max}$ is considered. On the contrary, when the FPR is lower than Γ, the order of the ODE method is decreased to $q' = q - 1 \geq 1$. Γ can be chosen as the maximum false positive rate we can accept; γ must be lower than Γ but in the same order of magnitude as Γ. This procedure is shown in Algorithm 1. The selection of the order is every $c_{max} = 10$ times or when the detector makes a false positive.

D. About Correctness

Let us see under which conditions the double-checking allows to detect SDC that would not have been detected by the adaptive controller. For this article, we consider only the case where the solver is using the Heun-Euler method and the LIP-based double-checking at order 1. At step n, we have the following expressions:

$$
\begin{aligned}
x_n &= x_{n-1} + \frac{h_n}{2}\left(f(x_{n-1}) + f\left(x_{n-1} + h_n f(x_{n-1})\right)\right)\\
LTE_1 &= \frac{h_n}{2}\left(-f(x_{n-1}) + f\left(x_{n-1} + h_n f(x_{n-1})\right)\right)\\
LTE_2 &= \left(x_{n-2} + x_{n-1}\right)\frac{h_n}{h_{n-1}}\\
&\quad + \frac{h_n}{2}\left[f(x_{n-1}) + f\left(x_{n-1} + h_n f(x_{n-1})\right)\right].
\end{aligned}
$$

If the SDC shifts x_{n-1} by ϵ, then $x_n^c = x_n^o + \epsilon$, $LTE_1^c = LTE_1^o$ and $LTE_2^c = LTE_2^o + \frac{h_n}{h_{n-1}}\epsilon$. Here, only the second estimate is affected by the SDC and is able to detect it. The double-checking is thus necessary.

If the SDC shifts K_i by ϵ, then $x_n^c = x_n^o + \epsilon\frac{h_n}{2}$, $LTE_1^c = LTE_1^o - \epsilon\frac{h_n}{2}$ and $LTE_2^c = LTE_2^o + \epsilon\frac{h_n}{2}$. The double-checking is necessary if $x_n^c - x(t_n)$ and LTE_2 exceed the tolerance, whereas LTE_1 does not. By noting $\tau_n = \sqrt{n}.Err_n > 0$, it provides the following inequalities for each component j of the vectors:

$$
\begin{aligned}
2.\frac{\tau_n - x_{n,j}^o + x(t_n)_j}{h_n} > \epsilon_j > 2.\frac{-\tau_n - x_{n,j}^o + x(t_n)_j}{h_n}\\
2.\frac{\tau_n + LTE_{1,j}}{h_n} > \epsilon_j > 2.\frac{-\tau_n + LTE_{1,j}}{h_n}\\
2.\frac{\tau_n - LTE_{2,j}}{h_n} > \epsilon_j > 2.\frac{-\tau_n - LTE_{2,j}}{h_n}.
\end{aligned}
$$

Data: $(x_{n-k})_{k\geq 0}, f(x_n), q, N_{steps}$
Result: Rejection or validation of step n
$rejected = True$;
$SErr_1 = Estimating_1((x_{n-k})_{k\geq 0}, f(x_n))$;
// Eq. (2)
if $c++ == c_{max}$ **then**
  /* Update ODE method's order q */
  $c_{max} = 0$;
  if $FP_q/N_{steps} < \gamma$ **then**
    $q = \max(1, q-1)$
  else if $FP_q/N_{steps} > \Gamma$ **then**
    $q = \min(q_{max}, q+1)$
end
if $SErr_1 == lastSErr$ **then**
  /* Case of a false positive */
  $validation = True$;
  FP_q++;
  $c = c_{max}$
else
  bool validation $= SErr < 1.0$;
  if *validation* **then**
    $SErr_2 = Estimating_2((x_{n-k})_{k\geq 0}, f(x_n), q)$;
    $validation = SErr_2 < 1.0$;
    $lastSErr = SErr_1$;
  end
end
if *validation* **then**
  $n++$;
  $rejected = False$;
  $h = NewStepSize(Serr_1, h)$;
  // Eq. (5)
end

Algorithm 1: Our adaptive-controller's scheme

In this case, all terms can be interpreted as random variables, and thus an evaluation of the probability to achieve all equations is impossible. That is why we do empirical evaluations in Section VI.

E. Implementation

The implementation was done directly inside the adaptive controller. This allows reuse of some allocation in memory to compute the second estimate. We refer to the adaptive controller without the double-checking mechanism as a *classic adaptive controller*. Because x_{n-1} is already stored by the classic adaptive controller, double-checking requires the storage of only x_{n-2} and x_{n-3}.

VI. Experiments

In Section IV, we showed that the rejection mechanism of an adaptive solver is able to correct only a part of the SDCs. Some SDCs, although significant, remained in the solution because the rejection mechanism was corrupted and did not detect any outlier. Therefore, in Section V, we proposed a method that enhances the rejection mechanism by double-checking the acceptance of a step.

In this section, we experimentally validate our method with the use case introduced in Section II. First, we show that our method greatly reduces the risk of accepting a significant SDC. Second, we measure the overheads and the scalability of our double-checkings, in order to compare them with replication and to suggest improvements.

A. Detection Accuracy

	FPR	TPR	Significant FNR
Classic	0.0	31.1	13.3
LBDC	2.3	33.1	4.1
IBDC	4.2	41.9	1.1
Replication	0.0	100.0	0.0

Table III. Our double-checking based on Lagrange interpolation polynomials (LBDC) and on a numerical integration method (IBDC) compared with the expensive state-of-the-art replication and the classic adaptive controller without our enhancement (Classic). FPR = false positive rate. TPR = true positive rate. FNR = false negative rate. Unit is %

We applied the integration-based double-checking and the LIP-based double-checking to the Heun-Euler method. Table III compares their detection performances with replication and the classic adaptive controller. Details on how we defined the performances are given in Section IV.

The LIP-based double-checking reduces the rate of significant false negatives by a factor of 3, whereas the integration-based double-checking decreases the rate by a factor of 10. This difference in accuracy results from the fact that the error estimate used by the integration-based double-checking is more precise than that used by the LIP-based double-checking.

B. Overheads

SDCs impact the convergence rate. For example, by shifting the error estimate to a high value, the next step size is reduced, and the computation takes more time. We measure the computation time ratio (defined as the computational overhead) between our method with injected errors and the classic adaptive controller without injected errors to confirm that the convergence rate does not burst. Table IV presents also the memory overhead, due to the storage of previous step size in the double-checking mechanism.

Overheads	Memory (%)	Computation (%)
Classic	+0.0	+0.0
LBDC	+57.6	+2.4
IBDC	+42.7	+4.5
Replication	+100	+100

Table IV. Overheads between our double-checking based on Lagrange interpolation polynomials (LBDC) and on a numerical integration method (IDBC), replication, and the classic adaptive controller (Classic).

We also added the overheads of replication: the computational overhead of replication is at least $+100\%$ plus the rate of corrected steps, but the rate of corrected steps is below 1%, and thus the overhead is equal to $+100$.

The computational overhead for LBCD and IBCD is partly due to the cost of the double-checking and to false positives, since false positives require recomputing a noncorrupted step. For the integration-based double-checking, the false positive rate is 4.2%, while the computational overhead is $+4.5\%$. Therefore, the computational cost of our method is due mainly to the cost of recomputing a false positive. To a certain extent, the computational overhead can be reduced by decreasing the parameters γ and Γ, although doing so would also decrease the detection accuracy. The memory overhead can appear important but is on average two times lower than the memory overhead of replication. It decreases with the complexity of the ODE method of the solver: in general, the solver requires $N_k + 2$ vectors of data with N_k the number of function evaluations, whereas double-checking requires a fixed number of vectors.

C. Scalability

Table V. Details of the mean execution time computation for the classic adaptive controller (Class.), LIP-based double-checking (LBDC), and integration-based double-checking (IBDC). Results are given in seconds.

Component	512 cores			4096 cores		
Protection	Class.	LBDC	IBDC	Class.	LBDC	IBDC
Double-check	-	$3.8e^2$	$3.9e^2$	–	$1.5e^1$	$1.6e^1$
Step	$1.2e^3$	$1.3e^3$	$1.3e^3$	$4.6e^2$	$4.8e^2$	$4.8e^2$

Table V shows the mean execution time computation for the double-checking methods and the classic adaptive controller over 100 executions. The computational overheads remain below 5%. Double-checking scales similarly to the step itself, mainly because of the collective operation for computing the norms. Moreover, the table shows that double-checking is almost a purely additional cost to the classic adaptive controller, because the time execution of a step with double-checking is almost equal to the addition between the time execution of the step of the classic adaptive controller and the double-checking itself. A better implementation must instead better integrate the double-checking inside the adaptive controller. This could be done by computing the norm of the error estimates used by the classic adaptive controller and the double-checking methods at the same time. Doing so requires allocating an additional vector and thus increasing the memory overheads.

Figure 3 shows the relative performance in time (yellow) and memory (green) compared with the classical adaptive controller of the LIP-based double-checking (square) and the integration-based double-checking (circle) up to 4096 cores. The integration-based double-checking shows better performances than does the LIP-based double-checking in memory, time execution, and detection accuracy. The reason is that the integration-based double-checking is based on mathematical properties of solvers and is more specific than the Lagrange interpolation polynomials. The relative performance is computed as the difference between the performance of the double-checking and the classic adaptive controller, divided by the performance of the classic adaptive controller. The overheads

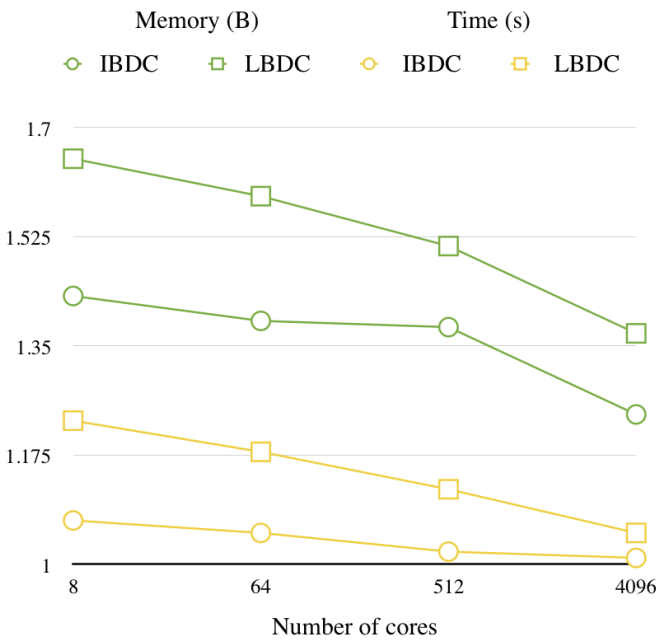

Figure 3. Relative performance in time and in memory of the LIP-based double-checking (LBDC) and integration-based double-checking (IBDC) compared with the classic adaptive controller until 4,096 cores.

tend to decrease with the number of cores, because the SDC detectors provide a better scalability than the rest of HyPar does. Indeed, as the number of cores decreases, parts of HyPar that cannot be parallelized become more and more important with respect to the cost of the double-checking.

VII. Related work

The resilience to SDC has been extensively studied for several years. While some methods were generic, others tend to be more specific to certain contexts.

A. Generic Solutions

The most generic solution for achieving the resilience to SDCs is replication [32]. It duplicates an execution and compares the results of the two executions. An SDC is reported when the results differ. In these cases, the overheads in memory and in computation are at least $+100\%$. Once an SDC is detected, the execution needs to be re-executed in order to provide a correction. Alternatively, a correction can be obtained directly by using a variant called triple-modular redundancy [33]. Triple-modular redundancy executes the simulation three times; if one result differs from the two other results, this result is claimed to be corrupted, and one of the two other results is kept. The overheads thus read $+200\%$. Reducing them is the challenge of the new methods.

B. Algorithmic Resilience

At a higher level, resilience can be achieved by using algorithm properties. Algorithm-based fault tolerance in the context of linear algebra has a large body of work [34], [35]. Several works have highlighted inherent resilient properties

inside algorithms. For example, Pauli et al. [36] showed that even in presence of the nonrecoverable samples, Monte Carlo methods can still converge; and the authors provided recommendations to enhance resilience.

Chen et al. [31] provided an extensive survey of algorithms that can naturally reject some SDCs. In particular, they showed that computing a Runge-Kutta method with two different step sizes allows rejecting some SDCs. Their algorithm is derived from the Richardson extrapolation and computes an error estimate. Our work can be seen as a significant improvement of this method, since we showed in Section IV that not all SDCs are filtered out with an error estimate.

C. Fixed Numerical Integration Solvers

In the context of fixed numerical integration solvers, several methods extract a surrogate function $\mathscr{S}$ and compare $\mathscr{S}$ with a threshold function $\mathscr{T}$. The step is validated when $|\mathscr{S}| < \mathscr{T}$. Correction can be achieved with a rollback to the previous step.

The adaptive impact-driven detector AID [12] developed by Di and Cappello is designed to detect SDCs in an iterative, time-stepping method with a fixed step size. The surrogate function is an error estimate obtained from the difference between x_n, the result at step n, and $\tilde{x}_n$, an extrapolation of previous results. These extrapolation methods are considered: the last value, a linear extrapolation, or a quadratic extrapolation. The best method is calculated by the *best-fitting algorithm* every $p = 5$ steps. Basically, this algorithm chooses the extrapolation method that minimizes the error of extrapolation or the memory cost. $\mathscr{T}$ is computed from η the number of false positives, ϵ the maximum error of extrapolation, r the interval of admissible values, and a user-defined error bound upon which an SDC is considered as unacceptable: $\mathscr{T} = (1 + \eta)(\epsilon + \theta r)$. AID is designed for fixed step-size because of the formulation of the extrapolation methods. However, Lagrangian interpolation polynomials, employed by our method LBDC, is designed for variable step sizes.

The surrogate function of Hot Rode [11] computes the difference between two error estimates, while the threshold function is initialized with first samples and is modified with the number of false positives. Hot Rode is restricted to fixed-solvers: as explained in Section V-C, estimates adapted to variable step sizes may differ significantly in adaptive solvers. Consequently, the surrogate function is more sensitive to stiffness than to SDCs.

VIII. Conclusion

In this study, we showed that two kinds of SDCs can occur in a numerical integration solver. Some, called *significant*, can exceed the user-defined tolerance of the approximation error. They may even hinder the convergence of a solver. Others, called *insignificant*, have no impact on the results with respect to the intrinsic approximation error of a solver.

Some solvers have an adaptive controller that controls the step size from an estimation of the approximation error. We showed that the rejection mechanism of this adaptive controller can correct some SDCs by rejecting corrupted steps. But an important number of significant SDCs are not rejected because the rejection mechanism is corrupted itself in the presence of an SDC.

Our solution consists in double-checking the acceptance of each step. Two strategies are proposed. The first strategy, called LIP-based double-checking, is derived from AID, a state-of-the-art SDC detector. An error estimate is obtained from the difference between the result of the solver and a prediction obtained by Lagrange interpolation polynomials. When the error estimate is higher than a certain threshold function, the step is rejected. The second strategy, called integration-based double-checking, computes an error estimate from the difference between two results: one from the ODE method used by the solver and one from another ODE method. This latter ODE method is a backward differentiation formula, which will usually get a larger stability area than with the ODE method of the solver. A difficulty arises, however, with variable step sizes. Error estimates might not agree, resulting in a high false positive rate. An algorithm is given to automatically select the best error estimate based on the false positive rate. With respect to the LIP-based double-checking, the integration-based double-checking is more difficult to apply for high-order methods, but the error estimate is closer to the error estimate of the adaptive controller.

Consequently, the method detects 99% of the significant SDCs. It reduces by a factor of 10 the number of false negatives of the adaptive controller in our experiments. At the same time, the overheads are smaller than with replication, by a factor of 10 in computation and by a factor of 2 in memory on average. In our experiments, we suggest further improvements to reduce those overheads. We plan also to explore the use of the double-checking mechanism for implicit solvers.

Acknowledgment

This material is based upon work supported in part by the National Science Foundation under Grant No. 1619253, and in part by the US Department of Energy Office of Sciences under contract DE-AC02-06CH11357. Part of this work also was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344.

We gratefully acknowledge the computing resources provided on Blues, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

References

[1] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler, "An analysis of latent sector errors in disk drives," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 35, no. 1. ACM, 2007, pp. 289–300.

[2] B. Panzer-Steindel, "Data integrity," 2007. [Online]. Available: http://indico.cern.ch/event/13797/session/0/contribution/3/attachments/115080/163419/Data_integrity_v3.pdf

[3] "A conversation with Jeff Bonwick and Bill Moore," 2007. [Online]. Available: http://queue.acm.org/detail.cfm?id=1317400

[4] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder, "An analysis of data corruption in the storage stack," *ACM Transactions on Storage (TOS)*, vol. 4, no. 3, p. 8, 2008.

[5] D. Li, J. S. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 57.

[6] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 78.

[7] S. E. Lapinsky and A. C. Easty, "Electromagnetic interference in critical care," *Journal of Critical Care*, vol. 21, no. 3, pp. 267–270, 2006.

[8] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. DeBardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen, "Addressing failures in exascale computing," *IJHPCA*, vol. 28, no. 2, pp. 403–421, 2014.

[9] E. Fehlberg, "Classical fourth- and lower order runge-kutta formulas with stepsize control and their application to heat transfer problems. classic fourth and lower order Runge-Kutta formulas with stepsize control, applying to heat transfer problems," *Computing*, vol. 6, no. 1, pp. 61–71, 1970.

[10] J. Fröhlich and K. Schneider, "An adaptive wavelet–vaguelette algorithm for the solution of pdes," *Journal of Computational Physics*, vol. 130, no. 2, pp. 174–190, 1997.

[11] P.-L. Guhur, H. Zhang, T. Peterka, E. Constantinescu, and F. Cappello, "Lightweight and accurate silent data corruption detection in ordinary differential equation solvers," in *European Conference on Parallel Processing*. Springer, 2016, pp. 644–656.

[12] S. Di and F. Cappello, "Adaptive impact-driven detection of silent data corruption for HPC applications," *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[13] "HyPar," 2015, `https://hypar.github.io`.

[14] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang, "PETSc Web page," http://www.mcs.anl.gov/petsc, 2015. [Online]. Available: http://www.mcs.anl.gov/petsc

[15] ——, "PETSc users manual," Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.6, 2015. [Online]. Available: http://www.mcs.anl.gov/petsc

[16] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*. Birkhäuser Press, 1997, pp. 163–202.

[17] D. Price, "Pentium fdiv flaw-lessons learned," *IEEE Micro*, vol. 15, no. 2, pp. 86–88, 1995.

[18] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, and S. McIntosh-Smith, "Unprotected computing: a large-scale study of dram raw error rate on a supercomputer," in *SC'16 Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2016.

[19] B. Zhou, X.-L. Yang, R. Liu, and W. Wei, "Image segmentation with partial differential equations," *Information Technology Journal*, vol. 9, no. 5, pp. 1049–1052, 2010.

[20] S. Di, E. Berrocal, and F. Cappello, "An efficient silent data corruption detection method with error-feedback control and even sampling for HPC applications," in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2015, pp. 271–280.

[21] L. Bautista-Gomez and F. Cappello, "Detecting and correcting data corruption in stencil applications through multivariate interpolation," in *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2015, pp. 595–602.

[22] S. Ghosh, S. Basu, and N. A. Touba, "Selecting error correcting codes to minimize power in memory checker circuits," *Journal of Low Power Electronics*, pp. 63–72, 2005.

[23] A. R. Benson, S. Schmit, and R. Schreiber, "Silent error detection in numerical time-stepping schemes," *International Journal of High Performance Computing Applications*, pp. 1–19, 2014.

[24] M. Giraldo, F. X.; Restelli, "A study of spectral element and discontinuous Galerkin methods for the NavierStokes equations in nonhydrostatic mesoscale atmospheric modeling: Equation sets and test cases," *Journal of Computational Physics*, vol. 227, no. 8, pp. 3849–3877, 2008.

[25] D. Ghosh and E. M. Constantinescu, "Well-balanced, conservative finite difference algorithm for atmospheric flows," *AIAA Journal*, vol. 54, no. 4, pp. 1370–1375, 2016.

[26] F. X. Giraldo, J. F. Kelly, and E. Constantinescu, "Implicit-explicit formulations of a three-dimensional nonhydrostatic unified model of the atmosphere (NUMA)," *SIAM Journal on Scientific Computing*, vol. 35, no. 5, pp. B1162–B1194, 2013.

[27] G.-S. Jiang and C.-W. Shu, "Efficient implementation of weighted ENO schemes," *Journal of Computational Physics*, vol. 126, no. 1, pp. 202–228, 1996.

[28] D. Ghosh and J. D. Baeder, "Compact reconstruction schemes with weighted ENO limiting for hyperbolic conservation laws," *SIAM Journal on Scientific Computing*, vol. 34, no. 3, pp. A1678–A1706, 2012.

[29] J. C. Butcher, *Numerical Methods for Ordinary Differential Equations*. Wiley Online Library, 2005.

[30] O. Abraham and G. Bolarin, "On error estimation in runge-kutta methods," *Leonardo Journal of Sciences*, vol. 18, pp. 1–10, 2011.

[31] S. Chen, G. Bronevetsky, M. Casas-Guix, and L. Peng, "Comprehensive algorithmic resilience for numeric applications," Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep. LLNL-CONF-618412, 2013.

[32] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *Computer*, no. 4, pp. 68–74, 1997.

[33] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[34] K.-H. Huang and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. 100, no. 6, pp. 518–528, 1984.

[35] Z. Chen and J. Dongarra, "Algorithm-based fault tolerance for fail-stop failures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 12, pp. 1628–1641, 2008.

[36] S. Pauli, P. Arbenz, and C. Schwab, "Intrinsic fault tolerance of multilevel Monte-Carlo methods," *Journal of Parallel and Distributed Computing*, vol. 84, pp. 24–36, 2015.