Exploring Partial Replication to Improve Lightweight Silent Data Corruption Detection for HPC Applications

Eduardo Berrocal[1], Leonardo Bautista-Gomez[2], Sheng Di[2], Zhiling Lan[1], and Franck Cappello[2]

[1] Illinois Institute of Technology, Chicago, IL, USA,
{eberroca,lan}@iit.edu
[2] Argonne National Laboratory, Lemont, IL, USA,
{leobago,sdi1,cappello}@anl.gov

Abstract. Silent data corruption (SDC) poses a great challenge for high-performance computing (HPC) applications as we move to extreme-scale systems. If not dealt with properly, SDC has the potential to influence important scientific results, leading scientists to wrong conclusions. In previous work, our detector was able to detect SDC in HPC applications to a certain level by using the peculiarities of the data (more specifically, its "smoothness" in time and space) to make predictions. Accurate predictions allow us to detect corruptions when data values are far "enough" from them. However, these data-analytic solutions are still far from fully protecting applications to a level comparable with more expensive solutions such as full replication. In this work, we propose partial replication to overcome this limitation. More specifically, we have observed that not all processes of an MPI application experience the same level of data variability at exactly the same time. Thus, we can smartly choose and replicate only those processes for which our lightweight data-analytic detectors would perform poorly. Our results indicate that our new approach can protect the MPI applications analyzed with 49–53% less overhead than that of full duplication with similar detection recall.
Index Terms—Silent Data Corruption Detection; Partial Replication; Data Analysis; HPC Applications.

1 Introduction

Silent data corruption (SDC) involves corruption to an application's memory state (including both code and data) caused by undetected soft errors, that is, errors that modify the information stored in electronic devices without destroying the functionality [13]. If undetected, these errors have the potential to be damaging since they can change the scientific output of HPC applications and mislead scientists with spurious results.

External causes of transient faults are usually rooted in cosmic ray particles hitting the electronic devices of the supercomputer [22]. As systems keep scaling

up, the increasing number of devices will make these external faults appear more often. Other techniques introduced to deal with excessive power consumption, such as aggressive voltage scaling or near-threshold operation, as well as more complex operating systems and libraries, may also increase the number of errors in the system [7].

Substantial work has been devoted to this problem, both at the hardware level and at higher levels of the system hierarchy. Currently, however, HPC applications rely almost exclusively on hardware protection mechanisms such as error-correcting codes (ECCs), parity checking, or chipkill-correct ECC for RAM devices [19, 10]. As we move toward the exascale, however, it is unclear whether this state of affairs can continue. For example, recent work shows that ECCs alone cannot detect and/or correct all possible errors [16]. In addition, not all parts of the system, such as logic units and registers inside the CPUs, are protected with ECCs.

With respect to software solutions, full process replication provides excellent detection accuracy for a broad range of applications. The major shortcoming of full replication is its overhead (e.g., $\geq$ 100% for duplication, $\geq$ 200% for triplication). Another promising solution is data-analytic-based (DAB) fault tolerance [26, 9, 2, 6], where detectors take advantage of the underlying properties of the application data (the smoothness in the time and/or space dimensions) in order to compute likely values for the evolution of the data and use those values to flag outliers as potential corruptions. Although DAB solutions provide high detection accuracy for a number of HPC applications with low overhead, their applicability is limited because of an implicit assumption—the application is expected to exhibit smoothness in its variables all the time.

In this work, we propose a new adaptive SDC detection approach that combines the merits of replication and DAB. More specifically, we have observed that not all processes of some MPI applications experience the same level of data variability at exactly the same time; hence, one can smartly choose and replicate only those processes for which lightweight data-analytic detectors would perform poorly. In addition, evaluating detectors solely on overall single-bit precision and recall may not be enough to understand how well applications are actually protected. Instead, we calculate the probability that a corruption will pass unnoticed by a particular detector. In our evaluation, we use two applications dealing with explosions from the FLASH code package [12], which are excellent candidates for testing partial replication. Our results show that our adaptive approach is able to protect the MPI applications analyzed (99.999% detection recall) replicating only 43–51% of all the processes with a maximum total overhead of only 52–56% (compared with 110% for pure duplication).

The rest of the paper is organized as follows. In Sect. 2 we describe how DAB SDC detectors work. In Sect. 3 we introduce our adaptive method for SDC detection. In Sect. 4 we describe the probabilistic evaluation metric used. In Sect. 5 we present our experimental results. In Sect. 6 we discuss related work in this area. In Sect. 7 we summarize our key findings and present future directions for this work.

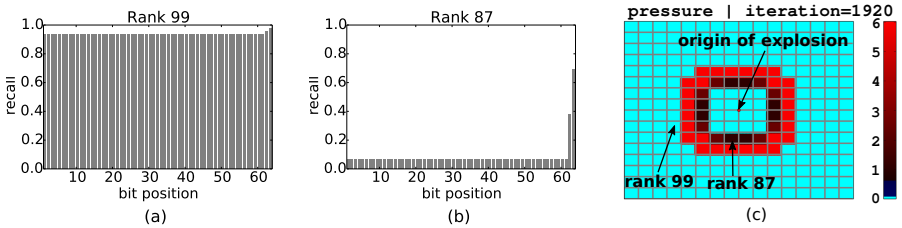

Fig. 1. Detection recall for two different processes in Sedov during 100 iterations.

2 Data-Analytic-Based SDC Detectors

In this section we describe how DAB detectors work. We also point out their major limitations.

Lightweight DAB SDC detectors are composed of two major parts. The *predictor* component computes a prediction for the next value of a particular data point. The prediction takes advantage of the underlying physical properties of the evolution of the data, since we have observed that this evolution is *smooth* in the time and/or space dimensions for a wide range of variables in HPC scientific applications. After the prediction is done, the *detector* component decides whether the current value of the data point is corrupted.

We have implemented our lightweight DAB SDC detectors inside the Fault Tolerance Interface (FTI) [4], an MPI library for HPC applications to perform fast and efficient checkpoint/restarts (C/R). We can add SDC detection support by taking advantage of the fact that iterative applications already provide (to FTI) the data variables representing their *state*. An HPC application needs to perform only one extra call to FTI: a call at the end of every iteration to allow our detectors to check for SDC in the data.

In our previous work we showed that one can detect a large number of corruptions by using simple and lightweight predictors. For the time dimension, we found that quadratic curve fitting (QCF) outperformed all the other considered options [6] with a memory overhead of less than 90% for all the applications studied. Another way to do predictions is by using the spatial information instead of the temporal information. In [3], 3D linear interpolation was used succesfully to predict values in a computational fluids dynamics (CFD) miniapplication.

Once a prediction $X(t)$ has been made, our detector decides whether the current value of the data $V(t)$ is a normal value by checking whether it falls inside a particular *confident interval* determined by a parameter δ: $[X(t)-\delta, X(t)+\delta]$. We calculate δ using the maximum prediction error from all data points in a process at $t-1$ multiplied by some constant: $\delta = \lambda \cdot e_{max}(t-1)$. This constant λ determines a tradeoff between detection *recall* (how many real corruptions can we actually detect) and *precision* (how many of the detected corruptions are actually real corruptions). In our case, the value for λ is chosen to have zero false positives given a particular execution size (i.e., to maximize *precision*).

When data values change too abruptly in a particular process, our δ becomes far too big to detect barely any corruption. Two examples of this kind of application dealing with sharp changes in the data are *Sedov* and *BlastBS*. Sedov is a hydrodynamical test code involving strong shocks and nonplanar symmetry [21]. BlastBS, on the other hand, is a 3D version of the magnetohydrodynamical spherical blast wave problem [27]. Both are part of the FLASH simulation code package.

To illustrate the problem at hand, we show in Fig. 1 (a) and (b) detection recall rates for single bit-flips injected on each bit position over two different processes in the variable *pressure* of Sedov during a particular period of time (100 iterations). One can see how the wave of the explosion passing through rank 87 is making detection recall rates decrease substantially for this variable[3]. In contrast, detection recall is high in rank 99. To get a glimpse of how this data looks like, consider Fig. 1 (c). Here, we show the state of the maximum of variable *pressure* right after the window of 100 time steps has passed.

3 Adaptive Method

Full replication is generally considered too costly for HPC because of its high overhead both in the time and the space dimensions. Partial replication, however, is worth considering for applications for which sharp changes in the data occur only in a small subset of the processes, such as those involving explosions or collisions (e.g., Sedov). Considering again the example introduced in Sect. 2, we can see that duplicating rank 99 in this situation is a major waste of resources, while rank 87 can surely benefit from replication, making detection recall go from below 10% in the majority of bits to 100% in all of them automatically.

One way to detect corruptions efficiently by using replication, proposed by Fiala et al. [11], is by comparing messages in MPI. The idea is that any corruption in the data of a particular process will ultimately produce corrupted messages that will be sent to other process. By comparing messages from replicas of the same process, one can determine whether that process (or any of its replicas) got its data corrupted. In this paper we adopt an adaptive strategy. For some processes (replication set), we use partial replication based on the method of Fiala et al. For the other processes, we use our lightweight DAB detectors.

We implement the following strategy in order to select our replication set and to dynamically adapt it over time. After the first iteration, we choose a subset of processes to replicate, given the maximum prediction error in that first iteration. The number of processes to replicate is determined by the *replication budget* B. During the following w application time steps (where w defines a *window*), we create an array S of size n, which is the number of processes in the application. After every time step, we sort all processes in ascending order given their maximum prediction errors and add their positions in S. For example, if at

[3] The rank of a process in MPI is its ID inside a group of processes. In this paper we consider only the rank of the general group to which all processes belong. In this sense, we use rank(s) or process(es) interchangeably.

a particular time step, rank 12 is the one with the highest prediction error and there are 128 processes, then $S[12] \mathrel{+}= 128$. When w steps have passed, the score S represents an aggregation of the relative positions of each rank with respect to the others given their prediction errors during the window w. At this point we sort all processes by their score S, pick the top B (which is the allocated budget) as the new replication set, and reset S to start a new window again.

4 Probabilistic Evaluation Metric

In order to understand why this metric is needed, consider the case where we have a mechanism with perfect detection recall for the 22 most significant bits of 32-bit numbers. What is the probability that, in this particular example, a corruption will evade our detector? The answer to this question will actually depend on how many bits can get "flipped" in the memory state of the application. Assuming 1-bit-flip corruptions only, we could say that $10/32 = 0.3125$ (31% of corruptions will pass undetected). For 2-bit-flips, the probability would be $(10/32)\times(9/31) = 0.0907$ (9.07% corruptions will pass undetected). For our detector to be unaware of a 2-bit-flip corruption in this case, all flips would always need to hit bits in the 10 less significant positions of the mantissa. We could continue with 3-bit-flips, 4, and so on. An interesting observation from this example is that, generally, the fewer bits that can get "flipped" in a system, the harder it is to detect corruptions using software mechanisms. Furthermore, another interesting question appears: What is the distribution of corruption sizes (in terms of the number of bits) in the system? Is a corruption affecting a large number of bits more or less common than one affecting just a few? The key idea is that protecting the data of simulations at this level is not so much protecting against particular bit-flips as it is protecting against numerical deviations from the original data.

In this work we use an evaluation metric based on the probability that a corruption will pass unnoticed by a particular detector. Since we aim at designing general SDC detectors, we cannot assume that bit-flips in the less significant bits of the mantissa are harmless. For example, we performed a sensitivity study where we injected different corruption sizes on multiple applications (the full study is omitted because of space limitations). In that study we observed that the same exact corruption produces different impacts[4] on different applications, i.e., there are applications that can absorb the corruption effortlesly while there are others that suffer big data deviations.

The evaluation metric, which we also call the *probability of undiscovered corruption*, is defined as

$$P_{\mathrm{f}} = \sum_{i=1}^{N} \left[P(\#\mathrm{bits} = i) \times \left(1 - \left(\frac{1}{N} \times \sum_{j=1}^{N} r_j\right)\right)^i \right], \tag{1}$$

[4] Impact is defined as the rate of deviation over the variable's total data range during the execution. For example, a deviation of 10 on a [0,200] range produces an impact of 0.05.

where N is the number of bits per data point (i.e., 64), $\frac{1}{N} \times \sum_{j=1}^{N} r_j$ represents the average recall rate for all bit positions collected during our injection studies, and $P(\#\text{bits} = i)$ represents the probability that the corruption is exactly i bits long.

The distribution $P(\#\text{bits} = x)$ depends on how corruptions in the whole system ultimately affect the numerical data of simulations. Because of the impossibility of calculating this distribution for a system as massive as a supercomputer, we assume four distributions representing the following four cases: (1) the number of bits affected is usually small, with 1 bit being the most common size (for this case, we use a Poisson distribution with $\lambda = 1.0$); (2) all bit sizes are equally probable (i.e., $P(\#\text{bits} = x) = 1/N$); (3) all possible corruptions (2^N) are equally probable (e.g., P$\sim \mathcal{N}(32.5, 13.05)$ for $N = 64$); and (4) the number of bits affected is usually big, with N bits being the most common size (for this case, we use the inverse of distribution (1)).

5 Evaluation

We use two applications from the FLASH code package in our experiments—Sedov and BlastBS—representing two different types of explosions. These applications are excellent candidates for testing the effectiveness of partial replication for data experiencing sharp changes due to explosions and collisions. Implementations of MPI allowing replication at the process level, such as RedMPI [11], do not yet support partial replication; we simulate partial replication by considering precision and recall to be 100% for those processes that are part of the *replication set*[5]. For the others, we use our lightweight SDC detectors.

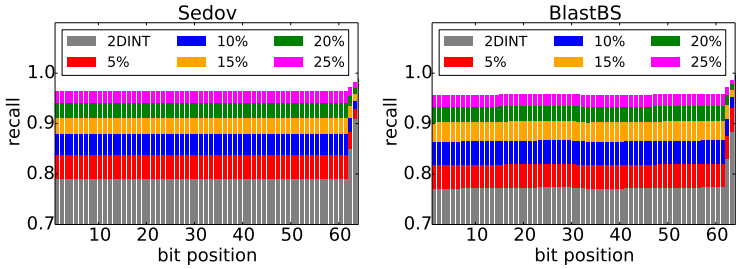

Fig. 2. Single-bit detection recall results from our injection study. We use two applications (Sedov and BlastBS) running 256 processes, and we set $w = 100$. Five partial replication rates (5-25%) are compared with nonreplication (2DINT).

Detection recall for each bit position is calculated by averaging the results over hundreds of random injections on the pressure variable in every process over

[5] Of course, this only holds for deterministic applications (which is the case here).

thousands of time steps. For all our experiments, we set our λ parameter, which controls our dynamic *detection range* δ (see Sect. 2) to have exactly zero false positives. In all the experiments, we run the applications using 256 processes, while the data domain is configured to be a two-dimensional grid. We report the results only of those experiments using linear interpolation (spatial) as our predictor. Similar results were obtained using our temporal predictor (QCF), so we omit them here.

Figure 2 presents the results of our injection study. Here, we fix the window w (see Sect. 3) to be 100 time steps. One can see the added benefit of using partial replication for improving single-bit detection recall rates. For example, we observe an overall improvement for Sedov from 6% for 5% replication to 18% for 25% replication. For BlastBS we also see significant gains, with improvements from 5% for 5% replication to 24% for 25% replication.

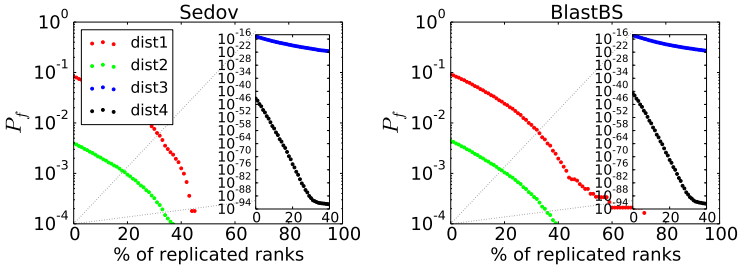

Fig. 3. Probability of undiscovered corruption when replicating a particular percentage of processes using the four distributions $P(\#\text{bits}= x)$ described in Sec. 4. Note that the y-axis is plotted using logarithmic scale. The small subplots represent the data zoomed below 10^{-15}.

Figure 3 presents the results when using the probabilistic evaluation metric P_f with the four distributions presented in Sect. 4. We note that the y-axis is plotted by using a logarithmic scale. We can categorize the distributions in a spectrum from *difficult* (dist1) to *easy* (dist4) (as discussed in Sect. 4, the fewer the number of bits that can get corrupted, the harder it is to detect corruptions at the software level). In this case, only distributions 1 and 2 are of further interest to us, since distributions 3 and 4 represent easy detection cases; that is, the probability of undiscovered corruption using our DAB detectors is already below 10^{-15} without even considering replication. For distribution 1, we need over 43% of the processes replicated in order to achieve a 99.999% protection (i.e., $P_f < 0.001$) level in the case of Sedov and 51% in the case of BlastBS. Recall that distribution 1 is the most *difficult* one, representing an upper bound in the number of replicated processes needed.

Apart from the obvious performance overhead incurred by using replication (i.e., extra hardware needed to run extra processes, or spatial overhead), there is also an overhead introduced by extra messages sent throughout the network,

which ultimately enlarges the runtime of applications. Another source of temporal overhead is our own DAB detector, which needs to run on every iteration and check all the data points for all the protected variables. From the system's point of view, both dimensions—temporal and spatial—contribute equally to the overall overhead, so both should be included. In partial replication we also need to consider the extra temporal overhead introduced by process migration when changing the replication set. We calculate the total overhead, then, using the following model:

$$O(r, w) = T(r) \times (r + 1) + \frac{M \times r \times n}{W \times T_w}. \tag{2}$$

where r is the replication rate (e.g., 0.5 when replicating half of the processes) and $T(r)$ is the runtime overhead introduced by the DAB detector and partial replication when running with a replication rate equal to r (e.g., $T(r) = 1.1$ if there is a 10% increase in running time). The right-hand side of the summation represents the overhead introduced by changing the replication set every w steps[6]. In this part, M is the memory used per process, $r \times n$ is the replication budget (n is the number of processes), W represents the aggregate network bandwidth in the system, and T_w is the time taken to run w steps in the original application. Note that this is an upper bound, since in some cases the number of processes to replicate is less than the budget, namely, when some processes replicated in the previous window are chosen again for the current one. We find that in only a few cases does the replication set change completely.

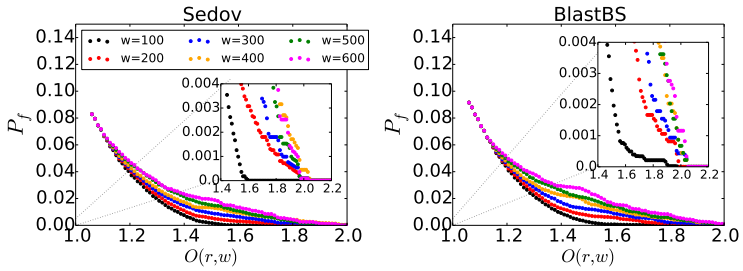

Fig. 4. Total overhead introduced by partial replication using different values of w. Distribution 1 used for $P(\#\text{bits}= x)$; the small subplots represent the data zoomed between $[0.0, 0.004]$.

The temporal overhead $T(r)$ may vary depending on the communication-to-computation ratio of the application. For those cases where computation dominates communication, the extra overhead is usually small. Fiala et al. [11] show

[6] Wang et al. [25] show that calculating process migration time as $process\_memory/network\_bandwidth$ is a fairly good estimate.

that temporal overhead for full duplication (i.e., $r = 1.0$) is not a concern (around 1–2%) for those applications that can maintain a well-balanced communication-to-computation ratio as they scale (applications exhibiting weak scalability). On the other hand, temporal overheads can reach 30% for network-bound applications and kernels. Since we are simulating partial replication, we are unable to measure exactly the value of $T(r)$ for the applications used. In this case, we assume the temporal overhead introduced by the extra network messages never to be above 5%, given that the stencil codes evaluated are not network-bound. Moreover, our experiments indicate that the temporal overhead introduced by using our DAB detectors is never above 6%[7]. Thus, we set $T(r = 1.0) = 1.11$ (i.e., 5+6=11% temporal overhead introduced by replicating all processes and using our DAB detector on every process). We estimate $T(r)$ for $r < 1.0$ assuming a balanced communicaton pattern between processes (which is the case in the stencil codes evaluated, where processes communicate mainly with their neighbors): $\hat{T}(r < 1.0) = 1 + r \times 0.05 + 0.06$.

In order to get an idea of how much overhead would be introduced by partial replication, we compute the values of P_f in Fig. 4 based on $O(r)$, for different values of the parameter w. Moreover, we assume distribution 1 for $P(\#\text{bits}= x)$. All the injection experiments are run on the Fusion cluster at Argonne National Laboratory [1], which has an InfiniBand QDR network with a bandwidth of 4 GB/s per link, per direction, arranged on a fat tree topology. Since we are not taking into account network contention issues in our overhead model, we set W to the lowest possible aggregate bandwidth in order to get an upper bound on the effect that the network bandwidth has on the overhead. That is, we set W=4 GB/s.

As one can see, a window of a 100 time steps is the best choice among all the considered possibilities. The reason is that the smaller temporal overheads can not compensate for the accuracy loss incurred when using larger window sizes. Our analyses show that we can have a 99.999% protection (i.e., $P_f < 0.001$) with $w = 100$ with a total overhead of around 1.52 (52%) for Sedov and 1.56 (56%) for BlastBS, with a replication rate of 43% and 51% respectively. This is an improvement of 53% and 49%, respectively, over full duplication (considering 5% in temporal overhead due to the extra network messages, full duplication has a total overhead of 2.1, or 110%) with a detection recall close to 100%. For easy comparison, these results are listed in Table 1.

6 Related Work

Software solutions for SDC detection can be grouped in four main categories: (1) full replication [11, 18], which is the most general but also the most expensive; (2) algorithm-based fault tolerance (ABFT) [14]; (3) approximate computing [5];

[7] The memory overhead of our DAB detectors is practically 0% given that we are using spatial-based predictors only in this study. For that reason, we do not include extra memory usage in the overhead calculation.

Table 1. Detection recall and overhead for DAB-only detectors, 2x replication, and our adaptive solution. In the latter, two cases are shown corresponding to two protection levels: 97% and 99.999% recall, respectively.

		DAB-only	**Duplication**	**Adaptive (case 1)**	**Adaptive (case 2)**
Sedov	*Overhead*	6%	110%	25%	52%
	Recall	92%	100%	97%	99.999%
BlastBS	*Overhead*	6%	110%	26%	56%
	Recall	91%	100%	97%	99.999%

and (4) data-analytic-based (DAB) fault tolerance [26, 9, 2, 6]. ABFT and approximate computing are not general enough and have limited applicability, since kernels need to be adapted manually and only a subset of them can be protected. In the case of DAB, detectors take advantage of the underlying properties of the applications' data (their smoothness in the time and/or space dimensions) in order to compute likely values for the evolution of the data and use those to flag outliers as potential corruptions. In this work we combine replication-based and DAB in order to avoid some of their individual shortcomings (i.e., the high cost of replication and the limited applicability of DAB).

Replication mechanisms for fault tolerance have been studied extensively in the past, especially in the context of aerospace and command and control systems [8]. Traditionally, the HPC community has considered replication to be too expensive to be applicable; and, to the best of our knowledge, it has not been implemented in any real production system.

Liu et al. [17] propose partial replication *in time* by taking advantage of the fact that soft errors in the first 60% of iterations of some iterative applications are relatively tolerable. The idea is to duplicate all processes only during the last 40% of iterations. Nakka et al. [20], Subasi et al. [24], and Hukerikar et al. [15]—by introducing new programming language syntax—propose to make the programmers responsible for identifying those parts of the code or data that are critical and need to be replicated. In contrast to these solutions, which are application dependent, our work is more general in the sense that we do not require any specific knowledge of tolerability to errors of particular iterations, variables, or code regions.

Partial replication in HPC where processes are chosen at random has also been investigated. Research has shown, however, that such an approach does not pay off [23]. In this work we choose the processes to replicate based on their data behavior.

7 Conclusions and Future Work

In this paper we have shown that combining partial replication along with DAB detectors allows us to get SDC protection levels that are close enough to those achieved by duplication at a lower overhead price. Our results show that we can get an overall SDC protection level, or recall, of 99.999% replicating only between

43% and 51% of all the processes with a maximum total overhead (upper bound) of 52–56% (compared to 110% for duplication) for the applications analyzed.

As future steps for this work, we want to consider the situation where the replication budget B is "elastic" during the length of the computation—for example, a situation where we can replicate a small number of processes (say, 10%) during the majority of the computation but increase the rate to a higher number (say, 60%), for a short period of time. This strategy can be useful for situations where sharp data changes are concentrated not only in a particular place in space but also in time. One can imagine an scenario in exascale where systems will have spare resources, in our case nodes, which will be allowed to be requested "on the fly" by applications and libraries in order to perform fault tolerance tasks.

Acknowledgments

This material was based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, under Contract DE-AC02-06CH11357, and by the ANR RESCUE and the INRIA-Illinois-ANL- BSC-JSC-Riken Joint Laboratory on Extreme Scale Computing. The work at the Illinois Institute of Technology is supported in part by U.S. National Science Foundation grants CNS-1320125 and CCF-1422009.

References

1. Fusion cluster at Argonne National Laboratory. [online] Available at http://http://www.lcrc.anl.gov/guides/Fusion
2. Bautista-Gomez, L.A., Cappello, F.: Detecting silent data corruption through data dynamic monitoring for scientific applications. In: PPoPP'14. pp. 381–382 (2014)
3. Bautista-Gomez, L.A., Cappello, F.: Detecting and correcting data corruption in stencil applications through multivariate interpolation. In: 1st International Workshop on Fault Tolerant Systems (part of Cluster'15). pp. 595–602 (2015)
4. Bautista-Gomez, L.A., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., Matsuoka, S.: Fti: High performance fault tolerance interface for hybrid systems. In: SC'11. pp. 32:1–32:32 (2011)
5. Benson, A.R., Schmit, S., Schreiber, R.: Silent error detection in numerical time-stepping schemes. International Journal of High Performance Computing Applications pp. 1–20 (2014)
6. Berrocal, E., Bautista-Gomez, L., Di, S., Lan, Z., Cappello, F.: Lightweight silent data corruption detection based on runtime data analysis for hpc applications. In: HPDC'15 (short paper) (2015)
7. Borkar, S.: Major challenges to achieve exascale performance. Intel Corp. (April 2009)
8. Briere, D., Traverse, P.: AIRBUS A320/A330/A340 electrical flight controls – a family of fault-tolerant systems. In: Proceedings of the IEEE International Symposium on Fault-Tolerant Computing. pp. 616–623 (1993)
9. Chalermarrewong, T., Achalakul, T., See, S.C.W.: Failure prediction of data centers using time series and fault tree analysis. In: ICPads'12. pp. 794–799 (2012)

10. Dell, T.J.: A white paper on the benefits of chipkill-correct ecc for pc server main memory. In: IBM Microelectronics Division. pp. 1–23 (1997)
11. Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., Brightwell, R.: Detection and correction of silent data corruption for large-scale high-performance computing. In: SC'12. pp. 78:1–78:12 (2012)
12. Fryxell, B., Olson, K., Ricker, P., Timmes, F.X., Zingale, M., Lamb, D.Q., MacNeice, P., Rosner, R., Truran, J.W., Tufo, H.: Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. The Astrophysical Journal Supplement Series (ApJS) 131, 273–334 (2000)
13. Hengartner, N.W., Takala, E., Michalak, S.E., Wender, S.A.: Evaluating experiments for estimating the bit failure cross-section of semiconductors using a colored spectrum neutron beam. Technometrics 50(1), 8–14 (Feb 2008)
14. Huang, K.H., Abraham, J.A.: Algorithm-based fault tolerance for matrix operations. IEEE Transactions on Computers 100(6), 518–528 (1984)
15. Hukerikar, S., Diniz, P.C., Lucas, R.F., Teranishi, K.: Opportunistic application-level fault detection through adaptive redundant multithreading. In: HPCS'14 (2014)
16. Hwang, A.A., Stefanovici, I.A., Schroeder, B.: Cosmic rays don't strike twice: Understanding the nature of dram errors and the implications for system design. In: ASPLOS'XVII. pp. 111–122 (2012)
17. Liu, J., Kurt, M.C., Agrawal, G.: A practical approach for handling soft errors in iterative applications. In: Cluster'15. pp. 158–161 (2015)
18. Mukherjee, S., Kontz, M., Reinhardt, S.: Detailed design and evaluation of redundant multi-threading alternatives. In: ISCA'02. pp. 99–110 (2002)
19. Mukherjee, S.S., Emer, J., Reinhardt, S.K.: The soft error problem: An architectural perspective. In: HPCA'05 (2005)
20. Nakka, N., Pattabiraman, K., Iyer, R.: Processor-level selective replication. In: DSN'07. pp. 544–553 (2007)
21. Sedov, L.I.: Similarity and Dimensional Methods in Mechanics (10th Edition). Academic Press, New York (1959)
22. Snir, M., et. al.: Addressing failures in exascale computing. International Journal of High Performance Computing 28(2), 129–173 (March 2014)
23. Stearly, J., Ferreira, K., Robinson, D., Laros, J., Pedretti, K., Arnold, D., Bridges, P., Riesen, R.: Does partial replication pay off? In: DSN'12 (2012)
24. Subasi, O., Arias, J., Unsal, O., Labarta, J., Cristal, A.: Programmer-directed partial redundancy for resilient hpc. In: CF'15 (2015)
25. Wang, C., Mueller, F., Engelmann, C., Scott, S.L.: Proactive process-level live migration in hpc environments. In: SC'08 (2008)
26. Yim, K.S.: Characterization of impact of transient faults and detection of data corruption errors in large-scale n-body programs using graphics processing units. In: IPDPS'14. pp. 458–467 (2014)
27. Zachary, A.L., Malagoli, A., Colella, P.: A higher-order godunov method for multidimensional ideal magnetohydrodynamics. SIAM Journal of Scientific Computing 15(2), 263–284 (1994)