Avoiding silent data corruption in checkpoint files

Leonardo Bautista Gomez[1,a)] Satoshi Matsuoka[1,b)]

Abstract: Silent data corruption is one of the sources of inaccurate results for scientific simulations. As supercomputers grow from tens to hundreds of thousands of computing cores, errors are becoming the norm, therefore it is critical to guarantee the correctness of future scientific applications. In this paper we propose a technique to verify data on the fly just before taking a checkpoint in order to guarantee the correctness of the saved data. We evaluate the efficiency of our approach injecting random failures in a HPC application and demonstrate its low overhead while checkpointing at high frequency.

1. Introduction

Scientific applications running in current supercomputers observe silent data corruption (SDC) in a regular basis. SDC are involuntary bit alterations that are not detected by the system and therefore introduce errors into the application execution, causing wrong results. Most current computing systems are protected against bit-flips using Error Correcting Codes (ECC), which are usually capable of detecting and correcting one bit-flip. However, in many cases this is not enough to protect the system. The validity and accuracy of scientific simulations deployed on high performance computing (HPC) systems is paramount. Hence, it is mandatory to guarantee the correctness of every result obtained trough scientific simulations.

Moreover, future large scale systems, featuring smaller transistor sizes and higher density are likely to observe SDC several times more often than current machines [3], [15]. Some studies predict that in the coming years, the large majority of errors will be soft-errors instead of hardware crashes [7]. Protecting scientific applications against SDC is usually expensive in energy (i.e. ECC) and it can also produce an important overhead, in particular at large scale, where large data sets need to be protected or analyzed against data corruption. Nowadays, the most popular fault tolerance technique in the HPC community is checkpointing. However, checkpoints do not protect the system against SDC. In fact, errors can *squeeze* inside the saved state, corrupting the checkpoints.

For this reason, novel techniques to protect checkpoints from corrupted data must be proposed and evaluated. Proposed techniques should minimize the execution overhead and should cover a large percentage of errors. Many current HPC applications implement periodical data consistency checkings, such as mass conservation or others. This tests depend strongly on the application and cannot be performed by the system without help from the user. In this article we present a user-friendly way to perform data consistency checkings and hide its overhead using fault tolerance dedicated threads. Our technique can detect an important percentage of data corruptions imposing only about 1% of overhead to the applications. Unfortunately our technique cannot correct the corrupted data but it does *transforms* an important number of SDCs into soft errors for a negligible price. Such soft errors can be tolerated using checkpointing.

1.1 Contributions

The main novelty of this research work is the way we off-load the data consistency checkings so that its cost is negligible. This can be decomposed in the following contributions:

- In order to substantially decrease the chance of SDC, we propose a technique to off-load data consistency checkings to dedicated threads that are executed in the background.
- We implement our proposed technique and develop a statistical analysis to predict the percentage of detected errors.
- We evaluate the overhead of our proposed technique and demonstrate its efficiency using synthetic benchmarks.

The rest of this article is organized as follows: Section 2 presents the background and motivations for this work and in Section 3 we presents some related work. Section 4 explains our proposed techniques and some details about its implementation while Section 5 shows our evaluation and finally we present our conclusions in Section 6.

2. Background

As explained in the previous section, data corruption may occur at any moment in any of the components of a system [16]. There are two types of data in a computer: the instructions of the program executed by the machine and the data modified by those instructions. When data corruption affects instructions of any of

[1] Tokyo Institute of Technology
[a)] leobago@matsulab.is.titech.ac.jp
[b)] matsu@is.titech.ac.jp

the procedures executed by the machine, most of the times the error will cause the given program to crash. This is usually observed as a transient failure. Transient failure can be easily recovered assuming periodic checkpointing. However, when the data corrupted makes part of the data used as input in the application, the program may not fail and continue its execution, sometimes even until the expected termination of the application. The data and results produced by the execution will not be correct as the data has been involuntarily altered.

Since such errors are not detected or detected very late in the execution, they are called silent errors. Silent errors are very difficult to deal with because of their very nature. Most of the fault tolerance techniques used in HPC can not address wrong results produced by silent errors. Scientists usually use application level techniques, such as data checking, during the execution. For instance, basic physic laws, such as mass conservation or positive time values, are checked at a certain frequency to guarantee correctness. These techniques cannot be easily standardized because *correct* value ranges are application dependent and only researches know the expected value ranges for the variables in their applications. Moreover, reactive techniques may not guarantee a proper recovery because the data may have been corrupted for a long period of time.

As systems grow in transistors count, noise levels increase and feature sizes decrease future supercomputers are more likely to be affected by silent errors than current machines. Moreover, silent errors can affect nodes, networks, disks, etc. increasing the probability of silent error in systems with many more components than current platforms. This lead us to a situation where future fault techniques will require solutions to avoid data corruption. However, not all the applications are similarly affected by silent errors, some applications are significantly more sensitive than others. This is because not all undetected faults necessarily lead to a silent error and the probability of this happening depends in the way the application stores its data and the corrupted data.

3. Related Work

There is a large literature in fault tolerance for HPC systems. System-level checkpointing [8] for instance makes checkpointing transparent to the user. Several works have proposed techniques to make checkpointing faster in large scale systems [1], [5], [6], [10], [11], [13], [14]. Also, there have been substantial work on improving PFS write troughput for large scale checkpointing [2], [17]. New hardware devices such as Phase Change Memory (PCM) have been studied to decrease fault tolerance overhead [7]. However, none of these works tackle the problem of SDC. All these improvement in checkpointing only cover fail-stop failures and assume that failures are detected in order to be recovered. Therefore, the problem of SDC remains open and it can be critical for future exascale machines.

Most of the work done to deal with involuntary bit alterations is done at the hardware level. Those works are focused on developing more elaborated ECC mechanisms to substantially decrease the chance of data corruption. However, it is very difficult to completely remove data corruption in very large scale systems. In addition, ECC strategies require extra space to store the checksum used for corrections. For instance, the new Fermi GPU can activate ECC on the GPU memory to guarantee higher confidence on the data correctness but it also requires about 12% of the GPU memory for the checksums. Another important element to take into account is the power consumption. Indeed, ECC requires a significant amount of energy to protect the data. Such energy consumption is tolerable at current scale, but it is not clear whether it will still be tolerable at exascale. The power limitations are exascale are so important that it is not clear today if we still can afford ECC in future system components.

Many scientific applications perform data consistency checks by themselves inside the application. This is a flexible way of checking data because the user can develop elaborated functions to check different parameters. However, this functions are usually executed by the same applications processes, imposing an important overhead to the application. Algorithm-Based Fault Tolerance (ABFT) [4] is one of the emerging solutions to deal with data corruption. In ABFT, the algorithms are implemented with additional computations and extra data, to make them tolerant to errors, including bit-flips or other involuntary alterations. However, if the corruption is not detected the corruption might propagate to the checksums kept during the execution, making the extra work useless.

Another way to protect large scale systems against SDC is to use process replication. For instance, replicating every process twice and periodically checking that both replicas are producing the same results is a way to detect data corruption. In addition, using three replicas the system can, not only detect but also correct data corruption. For instance, recent work [9] has proposed to use three way process replication with a voting system implemented at the communication level, to detect and automatically correct data corruption. This technique has the advantage of giving a high degree of confidence in the data correctness but unfortunately it also imposes a very expensive price, since three way data replication implies that the machine is only being used at about 30% of its maximum performance.

4. Pre-checkpoint data-checking

In our previous work, we have proposed to use idle CPU cores to encode the checkpoint files in parallel with the application execution. This encoding is performed after the application is checkpointed. This implies that the fault tolerance dedicated processes work periodically and if the encoding is not too long they might see some idle time between encodings. In addition, in the multi-level scheme that we have proposed, not every checkpoint is followed by some encoding, therefore the encoding processes are very likely to have idle time to perform other tasks.

We propose to use the idle time of the fault tolerance dedicated processes to perform data checkings in the background. The idea is to check that all the values of a given data vector are with-in the possible physical boundaries of the simulated model. Of-course, only the user knows which are such limits, therefore is the user who gives the vector to analyze and the minimum and maximum values that any item of the vector should respect. The data vector is then transfered to the fault tolerance dedicated thread, who will verify that every single element of the vector respects the mini-

mum and maximum specified by the user. As shown in Figure 1, this verification will be done in parallel with the application execution, which is one of the advantages of using the idle time of the fault tolerance dedicated thread.

The data checking could be performed at random time points during the execution with some periodicity. However, to increase the confidence in that the data saved in the checkpoint files is not corrupted, we decided to perform the data checking just before taking a checkpoint. While it is clear that the data can be corrupted at any point in time, for instance just after the checking and before the checkpoint, performing the data checking just before the checkpoint does increase the probability of having accurate results.

It is also important to notice that the data can be corrupted and still have a value with-in the specified boundaries, In particular, for physical data that has a wide range of possible correct values. While the data is corrupted and still with-in the given limits, the application is very likely to produce wrong results and our technique will not detect the corruption. The probability of such scenario taking place depends on the range of possible values that the application data can have. Very wide ranges will increase the probability of undetected corruption, while narrow ranges of possible values will make undetected corruption very unlikely. Such range depends on the application and its physic characteristics.

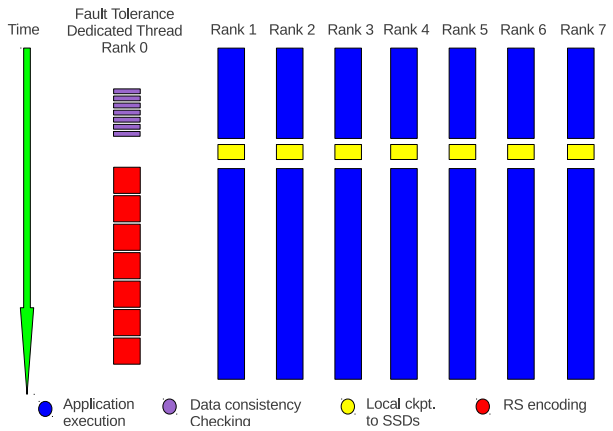

Fig. 1 Concurrent data consistency checking

If we analyze the way data is stored in scientific simulations we find that most applications use floating point representation of values, either in single precision or in double precision. Floating point representations use a certain number of bits (32 for single precision and 64 for double precision) and the bits are organized in three parts: the sign, the mantissa and the exponent. Depending on which of these three parts the corruption happens, the corruption might be more or less easy to detect. For instance, if the difference between the minimum and maximum possible values is no larger than one order of magnitude then every corruption in the exponent of the floating point representation should be detected by our technique.

Table 1 shows the internal representation of floating point numbers for both cases: single and double precision. As we can see, they reserve more than 70% of bits to the mantissa and the rest for exponent and sign. When one of the bits belonging to the exponent or the sign is corrupted, the application is very likely to be highly perturbed by such corruption because the data values are not within the expected limits. In such cases our technique will detect the corruption. Statistically, bit-flips are more likely to hit the mantissa because it takes the majority of the space in the floating point representation. When a bit is corrupted in the mantissa, depending on the position of the corrupted bit and the sensitivity of the application, the corruption can be ignored without perturbing the execution because the alteration is observed as a insignificant change in the decimal on the value in question. These alteration will not be detected by our technique because the values will be comprised between the corresponding limits but at the same time such errors can be simply ignored as they do not significantly change any data. The last case, is when one of the *important* bits of the mantissa is corrupted and the result is within the boundaries given by the user. Our technique will be unable to detect the corruption and the alteration will be significant, causing wrong results. For most applications this last scenario is unlikely to happen, as most alterations will corrupt either the sign or exponent, changing dramatically the values, or the right decimals of the mantissa that will not have a significant impact and can be ignored. However, there are some applications that are very sensitive to any minimal alteration and even the lightest alteration in the right part of the mantissa will cause the application to diverge.

Type	Sign	Mantissa	Exponent	Total
Single	1	8	23	32
Double	1	11	52	64

Table 1 Floating point representation

Finally, it is important to point out that this technique can only detect data corruption but cannot correct it. This is an important limitation of this technique. However, it importantly decreases the probability of SDC and when a data corruption is detected the application can restart from the last saved checkpoint, with a high probability that the previous checkpoint is not corrupted. In other words, our technique does not guarantee that SDC cannot happen, but it transforms an important number of SDCs in soft-errors, which are significantly less harmful failures.

5. Evaluation

In this section we present our evaluation for the proposed technique. In this evaluation we use synthetic benchmarks to measure the overhead on the application execution while performing the data checking before checkpointing and compare it with an execution without data checking. The synthetic benchmark corresponds to a program that allocates a certain amount of memory as a grid and then performs computations into that grid. The program is composed by a main loop in which at every time step we call a kernel. The kernel includes some data transfer between processes and it performs some dummy computation on the grid cells.

The program is modified to perform checkpointing with our fast checkpointing library FTI. The checkpoints are stored in local SSDs and some of them encoded using Reed-Solomon encod-

ing. We set up to versions of the code, one that only performs the checkpoint without data consistency checking, and a second version in which data consistency is performed before checkpointing. The data consistency is performed in the background by the head of the fault tolerance dedicated thread that is spawned with FTI and that also encodes the checkpoint files.

CPU	2 Intel Westmere-EP 3.20GHz 12Cores/node
Memory	16 GB
SSD	120 GB
Network	InfiniBand

Table 2 Cluster specifications

For our evaluation we use a medium size cluster which specifications are given in Table 2. We launch our synthetic benchmark with a total of 64 processes. Each process allocates the same amount of memory and performs the same computation and communications, in the same way that stencil applications works [12]. The allocated grid is a vector of floats (single precision). During the execution we perform one checkpoint and for the version of the code with data consistency the checking is performed before the checkpoint. We measure and compare the execution time for both versions for different grid sizes. The results are given in Figure 2. Every point in the figure is the average of three executions.

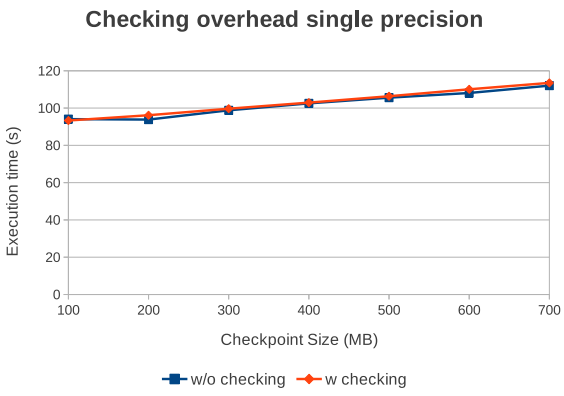

Fig. 2 Overhead comparison for single precision data

We increase the data size per process from 100MBs to 700MBs, which is almost the saturation point of the node memory launching 16 processes per node. As we can see, the execution time increases with the size of the data, which is normal because more data implies more computation. What is really important to notice in this experiment is that both versions (w/ and w/o checking) have almost the same execution time in most of the cases. The largest difference between both versions is not more than 1%. This was expectable because the data consistency checking is performed in the background before the checkpoint is taken.

We perform the same evaluation for double precision data, moving again from 100MBs to 700MBs. It is important to notice that in double precision the number of grid elements is lower than for single precision assuming the same storage size, because double precision data requires more space. This means that the number of data verifications is lower than in single precision. Once again, the overhead of performing data consistency checking in the background is negligible, as we can see in Figure 3.

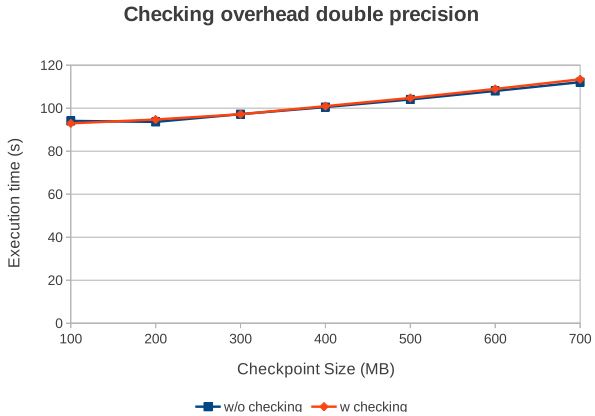

Fig. 3 Overhead comparison for double precision data

6. Conclusions

In this research work we have proposed a technique to detect SDC. Although our technique has several limitations, it has the benefit of being almost transparent in that the overhead imposed by this technique is negligible. This is because the data consistency checkings are performed in the background by the same fault tolerance dedicated threads that perform the checkpoint files encoding. Our technique does no guarantee to detect all the SDC and it cannot correct any, but it does *transform* an important number of SDCs into less harmful soft errors. In the future, we plan to allow the user to perform more complex data consistency verifications more than just giving two boundaries for the data vectors.

References

[1] Bautista-Gomez, L. A., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N. and Matsuoka, S.: FTI: High performance fault tolerance interface for hybrid systems., *SC*, ACM, p. 32 (2011).

[2] Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M. and Wingate, M.: PLFS: a checkpoint filesystem for parallel applications, *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC09, New York, NY, USA, ACM, pp. 21:1--21:12 (2009).

[3] Borkar, S.: Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation, *IEEE Micro*, Vol. 25, pp. 10--16 (2005).

[4] Bosilca, G., Delmas, R., Dongarra, J. and Langou, J.: Algorithm-based fault tolerance applied to high performance computing, *J. Parallel Distrib. Comput.*, Vol. 69, No. 4, pp. 410--416 (2009).

[5] Chen, Z. and Dongarra, J.: A Scalable Checkpoint Encoding Algorithm for Diskless Checkpointing, *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE*, pp. 71 --79 (2008).

[6] da Lu, C.: Scalable Diskless Checkpointing for Large Parallel Systems, Technical report, Ph.D. Dissertation, Univ. of Illinois at Urbana-Champain (2005).

[7] Dong, X., Muralimanohar, N., Jouppi, N., Kaufmann, R. and Xie, Y.: Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems, *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, ACM, pp. 57:1--57:12 (2009).

[8] Duell, J., Hargrove, P. and Roman., E.: The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart, Technical Report LBNL-54941, Future Technologies Group (2002).

[9] Ferreira, K., Stearley, J., Laros, III, J. H., Oldfield, R., Pedretti, K., Brightwell, R., Riesen, R., Bridges, P. G. and Arnold, D.: Evaluating the viability of process replication reliability for exascale systems, *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, ACM, pp. 44:1--44:12 (2011).

[10] Gomez, L., Nukada, A., Maruyama, N., Cappello, F. and Matsuoka, S.: Low-overhead diskless checkpoint for hybrid computing systems, *High Performance Computing (HiPC), 2010 International Conference on*, pp. 1 --10 (2010).

[11] Gomez, L. A. B., Maruyama, N., Cappello, F. and Matsuoka, S.: Distributed Diskless Checkpoint for Large Scale Systems, *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pp. 63 --72 (2010).

[12] Maruyama, N., Nomura, T., Sato, K. and Matsuoka, S.: Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers, *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, ACM, pp. 11:1--11:12 (2011).

[13] Moody, A., Bronevetsky, G., Mohror, K. and Supinski, B. R. d.: Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System, *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, Washington, DC, USA, pp. 1--11 (2010).

[14] Plank, J. S., Li, K. and Puening, M. A.: Diskless Checkpointing, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, No. 10, pp. 972--986 (1998).

[15] Schroeder, B. and Gibson, G. A.: A large-scale study of failures in high-performance computing systems, *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, Washington, DC, USA, IEEE Computer Society, pp. 249--258 (2006).

[16] Schroeder, B. and Gibson, G. A.: Understanding failures in petascale computers, *Journal of Physics: Conference Series*, Vol. 78, No. 1, p. 012022 (2007).

[17] Thakur, R., Gropp, W. and Lusk, E.: On Implementing MPI-IO Portably and with High Performance, *In Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, ACM Press, pp. 23--32 (1999).