On the Detection of Silent Data Corruptions in HPC Applications Using Redundant Multi-threading

Diego Pérez¹(✉), Thomas Ropars², and Esteban Meneses¹,³

¹ School of Computing, Costa Rica Institute of Technology, San José, Costa Rica
diperez@estudiantec.cr, emeneses@cenat.ac.cr
² Univ. Grenoble Alpes, CNRS, Grenoble INP, Grenoble, France
thomas.ropars@univ-grenoble-alpes.fr
³ Advanced Computing Laboratory, Costa Rica National High Technology Center, San José, Costa Rica

Abstract. This paper studies the use of Redundant Multi-Threading (RMT) to detect Silent Data Corruptions in HPC applications. To understand if it can be a viable solution in an HPC context, we study two software optimizations to reduce RMT performance overhead by reducing the amount of data exchanged between the replicated threads. We conduct experiments with representative HPC workloads to measure the performance gains obtained through these optimizations, and the error detection coverage they achieve. In the best case, when running on a processor that features Simultaneous Multi-Threading, our results show that the overhead can be as low as 1.4× without significantly reducing the ability to detect data corruptions.

Keywords: HPC · Silent data corruptions · Redundant multi-threading

1 Introduction

Silent Data Corruptions (SDCs) are alterations of data that go undetected by the hardware. These soft errors are usually attributed to external causes, such as high energy particles that may strike transistors and induce a bit-flip in the memory, the caches or the registers of the processor [12]. Such data corruptions may lead to a crash failure (e.g., a pointer corruption leading to an invalid memory access), or lead to a wrong application output in the worst case.

As supercomputers keep increasing in scale, concerns about SDCs become more serious [1,12,13]. This problem is exacerbated by the current technology trends in processor architectures (e.g., increased number of computing cores or near-threshold voltage) that aim at improving energy efficiency, but also increase the probability of SDCs. Error Correcting Codes (ECC) are usually used to detect and correct soft errors in main memory. However, such solutions...
are probably too expensive, both with respect to energy and performance, to be generalized to all caches and processor registers [3].

Several directions are explored to deal with SDCs in supercomputers, each of them having advantages and pitfalls. Some works propose to apply replication at the level of processes [5]. Solutions based on data analysis have also been proposed [2]. In this paper, we study an alternative approach: Redundant Multithreading (RMT). RMT detects SDCs at the level of threads. Hence, it can be cheaper than process replication. However, existing software-based RMT solutions [8,14,16] have major limitations. Either they induce a high performance overhead or they assume very infrequent interactions with other threads or processes, which is typically not a valid assumption with MPI applications.

The main factor that limits the performance of existing RMT solutions is the cost of communicating between threads to compare the output of their execution [8]. In software-based RMT, a leading thread has to send the result of each operation it executes to its sibling trailing thread that detects soft errors by comparing the values it receives with the result of its own operations. The single-producer/single-consumer (SPSC) queue involved in this communication is central to the performance of RMT.

This paper studies three directions to improve the performance of RMT approaches for HPC. First, it explores the possibility of using Simultaneous Multithreading (SMT) to improve the performance of RMT. The evolution of the hardware (e.g., increased number of cores in processors) makes it difficult for some applications to make good use of all the processors resources [10], thus we think it is worth considering this possibility. Second, we evaluate two solutions to reduce the amount of data that are exchanged between the leading and the trailing RMT threads: i) aggregating multiple produced values together before testing them for SDCs; ii) only testing a subset of the program variables, the ones that are identified as having the potential to corrupt a state outside of the thread context (we call this approach selective checking).

We present a detailed evaluation of these optimizations using two representative benchmarks. Our experiments evaluate the performance of the optimizations. Experiments with fault injection are also used to estimate the impact on the coverage offered by RMT. The main conclusion from our experiments is that RMT could be a viable solution for SDC detection at least for some HPC applications. The results obtained with the selective checking optimization show that, when taking advantage of SMT hardware threads, the overhead compared to a non-protected execution can be as low as 1.4×, without significantly degrading the error coverage compared to a standard RMT approach.

The paper is organized as follows: Sect. 2 presents the problem of soft-error detection and the related work. Section 3 describes the studied RMT technique and the optimizations. Section 4 presents the evaluation results.
2 Background

2.1 Detecting Soft Errors

This paper considers the case of soft errors due to external causes, such as high energy particles that may invert the state of transistors. These errors are unpredictable by nature. More specifically, we consider soft errors that are not detected by the hardware. Such a soft error might lead to a crash (e.g., a memory reference corruption may generate an invalid memory access), or might go totally undetected: these are silent data corruptions (SDCs). Soft errors can affect different components of a computing node including the main memory, the processor caches, and the processor registers.

Commonly, mechanisms at the hardware level, such as Error Correcting Codes (ECC) and parity bits, are used to protect the main memory and the processor caches [13]. Hence, most often, DRAM and caches are considered safe contrary to the processor registers and the combinational logic [12]. This is the kind of soft errors we are dealing with in this paper.

In today’s systems, detecting SDCs has to be done at the software level. In the HPC context, approaches based on process replication have been proposed [5]: each process of the application is replicated and the outputs of sibling processes are compared to detect differences. Such a technique can detect SDCs with high precision, but it is costly as it doubles the amount of resources required to run the application. An alternative is to rely on data analysis to detect unexpected variations in the values of some program variables [2]. Such solutions induce much less overhead than process replication. However, they are only applicable if the data changes in a predictable way, which is not always the case [3].

In this paper, we explore solutions based on thread-level replication to detect SDCs. Such a solution is an alternative to process replication that is expected to be cheaper. Namely, thread-level replication only replicates the computation and not the data. On the other hand, this implies that thread-level replication can only work under the assumption that the memory of the processor is reliable.

Two approaches to thread-level replication exist, ILR (Instruction Level Redundancy) and RMT (Redundant Multi-Threading). ILR duplicates the instructions inside a single thread and compares the results before every write to memory [6]. RMT duplicates each application thread into a leading and a trailing thread and compares the output to detect SDCs [8,14,16]. Both approaches have advantages and drawbacks. The most appropriate time to run a comparison in ILR is always right after the values have been computed when they are still in the registers of the processor. Since values are produced by two different threads in RMT, comparison checks are more costly. Hence, state-of-the-art ILR techniques are more efficient than best RMT solutions today [6]. However, RMT offers a higher degree of flexibility regarding when to run comparison checks compared to ILR [16], and more opportunities for performance optimizations. Hence, our goal is to explore the potential of RMT approaches in an HPC context.
2.2 Redundant Multi-threading

In RMT approaches, the sibling threads use a Single-Producer-Single-Consumer queue (SPSC) to exchange the values they are generating, to be able to compare them. Hence the performance of RMT depends on the performance of the queue and on how the threads synchronize through this queue.

The synchronization over the queue can be synchronous, asynchronous or semi-synchronous. In a synchronous approach, the trailing thread needs to acknowledge the leading thread before any write is made to the memory, which induces a huge performance overhead [16]. On the other hand, in a fully asynchronous approach, performance is promoted over safety [8]. But leaving SDCs undetected for some period of time could result in an uncontrolled propagation of the error to other parts of the application, and lead to an unrecoverable state (for instance, if corrupted data would be saved in a checkpoint).

The semi-synchronous approach aims at limiting the synchronization between threads without compromising safety [14]. In this approach, the threads progress most of the time asynchronously and an acknowledgment is sent by the trailing thread to the leading thread only when volatile variables, i.e., variables that must never get corrupted, are modified. A volatile variable is a variable that may be modified in ways unknown to the soft-error detection technique or have unknown side effects [16]. HPC applications include many accesses to volatile variables (I/O operations, MPI function calls) which limits the performance improvements that can be gained from the semi-synchronous technique.

Another direction to improve RMT performance is to work on the SPSC queue efficiency. Hence, batching has been proposed to improve the performance of SPSC queues [9]. However, fixed-size batching can only be applied with an asynchronous technique. Using batches with a semi-synchronous communication pattern could lead to a deadlock when the leading thread is waiting for an acknowledgment from the trailing thread, while the trailing thread is waiting for the current batch to be full to start processing data. Hence the most suitable solution for semi-synchronous RMT is to use a dynamic batching strategy where the leading thread can decide to terminate a batch early if a synchronous communication is required. According to our experiments\(^1\), the queue algorithm proposed by Wang et al. [14] is the best queue to implement semi-synchronous RMT. Hence, we consider this solution as baseline for our evaluation.

3 Solutions to Improve RMT Performance

To assess the potential of RMT with HPC applications, our study aims at exploring the impact in terms of performance and reliability of different optimizations that could be applied to such a technique. At the hardware level, we consider the impact of Simultaneous Multithreading (SMT). We leverage SMT’s ability to run sibling threads on the same physical core. At the software level, we study two optimizations that can be applied to the RMT approach to reduce the stress

\(^1\) Results not included in the paper due to the limited space.
on the SPSC queue and, thus, improve performance. In this section, we start by providing details about the vanilla RMT algorithm that we use as baseline for our study. Then we present the two optimizations that we propose to evaluate.

### 3.1 Vanilla Semi-synchronous RMT Algorithm

Our baseline algorithm is designed based on the solution proposed by Wang et al. [14]. To obtain efficient communication between sibling threads, they use delayed buffering and lazy synchronization. Delayed buffering means that data are buffered on the producer with the help of a local index and only when $K$ values have been enqueued, the data are made visible to the other thread. Lazy synchronization means that the algorithm avoids checking directly shared variables on each enqueue/dequeue operation, but iterates based on local indexes when possible [14]. Our implementation of this solution, that we call vanilla-RMT hereafter, defines the following API that includes 6 methods:

- **Produce**() is called by the leading thread to send a value to the trailing thread. It does not require an explicit acknowledgment. This method is used for asynchronous communication.
- **Consume**() is called by the trailing thread to read the next value from the queue and check for a soft error. This method is also used for asynchronous communication.
- **Produce**$\_\text{Volatile}$() is a blocking method called by the leading thread to enqueue a value and wait for an acknowledgment from the trailing thread before continuing.
- **Consume**$\_\text{Volatile}$() is called by the trailing thread to dequeue a value that requires an explicit acknowledgment and sends that acknowledgment.
- **Produce**$\_\text{Drt}$() is used to sent data directly from the leading to the trailing thread without any soft-error verification.
- **Consume**$\_\text{Drt}$() is used by the trailing thread to receive a value that is not computed locally from the leading thread.

We illustrate the use of this API through the example presented in Fig. 1. Figure 1a shows the original code snippet. The `foo` function computes the sum of the elements of a vector `x` passed as parameter. The result is stored in the local variable `l`. Then a MPI call is made with `l` as input parameter and `g` as output. Finally, `g` is returned by the function.

Figure 1b and c illustrates the transformed code that is executed by the leading thread and the trailing thread respectively with RMT. The first thing to mention is that since `l` is an input parameter of a library function call at line 9, it makes it a `volatile` variable. Hence, our RMT approach has to guarantee that this variable is not altered by a SDC before calling the MPI function, and so the leading and the trailing threads respectively have to call `Produce_Volatile()` and `Consume_Volatile()` at line 8. The intermediate values of variable `l` produced at line 5 are also checked for correctness, but since these intermediate values can only alter the local state of the thread, they are checked asynchronously using `Produce()` and `Consume()` calls at line 6.
We should point out that each synchronous exchange (calls to \_*Volatile()\ functions) makes the leading thread spin-wait for the trailing thread acknowledgment. Since the former might have already enqueued several \textit{non volatile} values, it has to hold until all of them are verified for correctness. Therefore, a \textit{volatile store} implies that the whole execution so far is checked for integrity.

In a RMT approach, instructions and functions calls that correspond to writes to volatile variables should not be replicated \cite{6,8,14,16}. We would not want to print twice a value to the console, or send data twice to a neighbor node. Hence, such calls are only made by the leading thread, as illustrated in line 9 of Fig. 1c. Since the outputs of such calls are required by the trailing thread to continue its execution, these outputs are directly transferred from the leading thread using \texttt{Produce\_Drt()} and \texttt{Consume\_Drt()} calls, as illustrated in line 10.

### 3.2 Optimization Techniques for RMT

We present the three techniques studied to improve the performance of RMT.

**Leveraging Simultaneous Multi-threading.** Most recent processor architectures implement simultaneous multithreading to increase the degree of parallelism in multicore processors. SMT allows having 2 or more threads concurrently executing in the same physical core \cite{11}. It makes a single physical processor appear as (at least) two logical processors. The physical execution resources are shared and the architecture state is duplicated for the logical processors.

The limited scalability of some HPC workloads with SMT threads has been highlighted in several studies \cite{10}. On the other hand, placing a leading thread and its corresponding trailing thread on SMT hardware threads of the same physical core can help to improve the performance of the communication between them, as they will share the same L1 cache. Leveraging SMT for soft-error detection has been proposed in the seminal paper by Reinhardt and Mukherjee \cite{11} but this solution was relying on dedicated hardware. The impact of SMT on RMT has been evaluated in \cite{14} with negative results. As we show in Sect. 4, the conclusions can be different on more recent hardware.
Aggregation of Values. The first software technique we propose to evaluate aims at reducing the contention of the queue between the leading and the trailing thread by aggregating several values together and to compare the aggregated values to detect SDCs. This approach was originally proposed by Mitropoulou et al. [8]. Obviously this approach can only be applied to asynchronous communication. We evaluate this idea using the sum (+) operator as aggregation operator. In [8], the authors propose to aggregate 2 values together. We study a generalization of this idea where $K$ values can be aggregated, with $K$ being a power of 2 to be able to compute a modulo very efficiently using bit-wise operations\(^2\).

It should be pointed out that using aggregation could potentially weaken soft-error detection. First, if more than one value is corrupted, there is a small chance that 2 bit-flips or more would compensate each other and go undetected. Second, there is a risk of overflow when computing the aggregated value.

Selective Checking. The second software technique that we evaluate aims at reducing the contention on the queue by limiting the number of values that are checked for correctness. This idea has already been proposed in the context of ILR [7, 15]. By definition, writes to volatile variables are the points where a thread could make a data corruption visible to the outside world. As such, we propose to only check values that are stored in these variables. We refer to this optimization as selective checking.

Applying this technique to the example of Fig. 1 would lead to only check the final value of $l$ for correctness at line 8. Obviously, with such an approach, there is also a risk of weakening the detection capabilities of the RMT technique. We evaluate this point during our experiments.

4 Evaluation

Our evaluation aims at assessing the efficiency that can be achieved when using an RMT approach to detect SDCs in HPC applications. More specifically, for the three optimizations described in Sect. 3.2, we want to evaluate: i) the performance improvement that can be obtained compared to the vanilla RMT approach presented in Sect. 3.1; ii) the reduction of the capacity of RMT to detect SDCs that they may induce.

4.1 Software Used for Evaluation

RMT Implementation. The implementation of the RMT technique described in Sect. 3.1 has been made in C. The optimizations presented in Sect. 3.2 have been applied to this code base. Special care has been taken to properly align variables and to pad the data structures to avoid any false sharing.

\(^2\) The results presented in Sect. 4 are obtained with $K = 16$. Our tests showed that this the value that leads to the best performance for our applications.
To evaluate the performance of RMT with existing applications, we directly modified the code of the applications to create the leading and trailing threads and to insert API calls when required. Automatically replicating an application for RMT using a compiler approach has already been done in other works [8,14,16]. Our work focuses on evaluating the efficiency of RMT approaches.

**Benchmarks.** To run our analysis, we select two applications, HPCCG and CoMD from the Mantevo benchmark suite. HPCCG is representative of finite element methods while CoMD is an implementation of classical molecular dynamics algorithms. We work with the MPI version of these applications.

### 4.2 Performance Evaluation

The first part of our evaluation focuses on the performance of RMT techniques. We start by presenting the experimental setup used for the evaluation. Then we evaluate the impact of the two software optimizations described in Sect.3.2. Finally, we evaluate the impact of using SMT threads.

**Experimental Setup.** All experiments presented below are run on a node equipped with 2 Intel Xeon E5-2630L v4 processors (Broadwell architecture) and 128 GB of RAM. Each processor features 10 physical cores with two Hyper-Threads per core. The applications are compiled using the highest level of optimization activated (-O3). The presented results are average execution times over 10 runs (with error bars representing the standard deviation).

For the two applications, the problem size considered for the evaluation is the following: a global problem size of $1.28 \times 10^8$ for HPCCG and a total number of atoms of $2.4565 \times 10^6$ for CoMD.

**Performance Impact of the Software Optimizations.** Figure 2 presents the performance of the different versions of RMT with the two tested applications. The applications are executed with 20 MPI ranks. This corresponds
to running one process per rank in the non-replicated execution and leads to
the best performance for these applications with the considered problem sizes\(^3\). The presented results are normalized execution times with the non-replicated execution as baseline.

We first focus on the results when the leading and the trailing threads of
each process are executed on different cores (labeled **Different Cores**). When
RMT is applied, we have 40 threads to execute in total\(^4\), which we pin in such a
way that the leading and the trailing threads are on a different core but on the
same NUMA node.

Comparing the performance of the *vanilla* RMT approach to the perform-
ance when optimizations are applied, we observe that the optimizations lead
to significant performance improvements. Aggregation leads to 18% and 20% of
improvements compared to the default RMT approach with CoMD and HPCCG
respectively. The *selective checking* achieves even better performance: 63% for
CoMD and 50% for HPCCG.

The obtained performance also show that the *vanilla* RMT approach induces
a very high overhead: a 6.7× slowdown for CoMD and a 3× slowdown for
HPCCG. We conclude from these results that such an approach is not viable
for SDC detection in HPC applications. On the other hand, with *selective check-
ing*, the slowdown reduces to 2.5× for CoMD and to 1.5× for HPCCG. The
slowdown could still be too high with CoMD but with HPCCG, the overhead
becomes promising and might also even be competitive against ILR techniques
[6], where the overall overhead across all applications\(^5\) is 2×. Indeed, in an app-
roach based on full process replication [5], one can expect to achieve a smaller
slowdown but at the cost of duplicated the amount of computing resources used
to run the application.

**Performance Impact of SMT.** In Fig. 2, the data labeled **Hyper-threads**
correspond to the configuration where the leading and trailing threads of each
rank are collocated on the same physical core. The results show that a (limited)
performance improvement is always observed. It ranges from 6.2% to 11.6%.
Hence, when this approach is combined with *selective checking*, the slowdown
reaches 2.3× for CoMD and 1.4× for HPCCG\(^6\).

### 4.3 Reliability Evaluation

In a second set of experiments, we evaluate the ability of RMT to detect soft
errors when the proposed performance optimizations are used. We identify two

\(^3\) Trying to take advantage of the SMT threads by running 40 ranks in the non-
replicated run does not provide any performance improvement.

\(^4\) We also tested configurations with 10 ranks when RMT was used to have one thread
per core. In most cases the performance were equivalent to the results with the
default configuration.

\(^5\) Tested applications in [6] are part of different benchmark suites than ours.

\(^6\) These results are not specific to the selected problem size. They remain equivalent
for other problem sizes in both applications.
aspects that can be impacted by changes in the RMT technique with respect to soft errors: i) the capacity of detecting errors might be altered; ii) the probability of experiencing an error might change.

**Ability to Detect Soft Errors.** To test soft-error detection, we use *FlipIt*, an LLVM-based soft-error injection tool for HPC applications [4]. For these experiments, we consider the HPCCG application.

We configured *FlipIt* to inject a single fault (bit flip) in each execution and we ensure that one fault will be randomly injected in each execution. We allow faults to be injected in arithmetic and control instructions, but not in pointer type instructions because inserting bit flips in these instructions often simply leads to a crash, which is not an interesting case for our evaluation.

In all experiments, HPCCG is run with a single rank, and we deactivate all compiler optimizations to be able to simply relate the executed instructions to the source code described in C. Furthermore, to have a comparable distribution of injected errors in the tests of the RMT techniques and simplify the analysis, we run the tests in the following way: i) We use the same seed for the random number generator that is used to define the sequence of errors that are injected over the different tests for each technique; ii) We configure *FlipIt* to inject errors only in the application code (and not in the RMT code that can vary in number of instructions depending on the optimization that is used).

In a first step, we focus on the HPCCG routine that computes a dot product between two vectors. This routine is very similar to the one presented in Fig. 1 with local values computed during each iteration of a loop, and the final result of being a volatile variable. Table 3a presents the results of the test. For each configuration, 5000 runs of HPCCG have been executed. We identified 5 possible outcomes for a run: **Benign** (the execution terminated correctly despite the bit flip), **Error detected** (by the RMT technique), **Corrupted** (the application returned a wrong result – a SDC occurred), **Crashed** (for instance due to an illegal memory access), **Timeout** (the execution did not terminate).

Several observations can be made based on the results presented in Fig. 3a. First, it should be noted that HPCCG is rather robust to soft errors injected in that part of the code since about 30% of the executions lead to the correct results in all configurations. On the other hand, 66% of the injected errors lead to an
SDC that corrupted the final result of the application with the non protected version of HPCCG. Our default RMT implementation was able to detect all the SDCs as expected. When applying the optimizations, it is important to notice that the RMT version using aggregation was able to detect all SDCs, while 13% of the executions were corrupted with selective checking (SC). This illustrates the trade-off that SC provides: a significant performance improvements at the cost of a reduced error coverage.

Some results presented in Fig. 3a can be a bit surprising at first sight. For instance, there are a bit more executions that terminate correctly with the default RMT approach or when the SC optimization is applied. Also, the version of RMT with aggregation detects more errors than the version without optimizations. We found out that all these inconsistencies have the same root cause. They relate to the case where a corruption leads to access an incorrect memory address when loading a value of the vectors used as input of the dot product. Depending on the layout of the data in memory, the incorrect memory address might be not valid (leading to a crash) or might be a valid address which contains a value (which might corrupt the result of the computation or not).

Figure 3b presents the results of the tests when we allow bit flips to be injected in any instruction of the application code. The first thing to observe is that most of the executions of the non-protected version terminate correctly. This implies that HPCCG is in general robust to soft errors. We can also notice that in this case, the SC approach does not experience any corrupted result. The results with the default RMT approach and with the aggregation optimization show that many of the benign errors are detected as soft errors. These are not false positive in the sense that real soft errors were detected, but it is still counter-productive since executions that would have otherwise terminated correctly got interrupted. The results obtained with the SC optimization illustrate another advantage of this approach. Since SC only verifies writes done to volatile variables, it almost never detects these benign errors, allowing more often the execution to terminate correctly. We also ran tests where the bit flips are only inserted in the most significant bits. The obtained results are qualitatively equivalent the one of Table 3b with less benign faults being observed.

Probability of Experiencing a Bit-Flip. The results obtained during our evaluation illustrate the trade-offs that are obtained with the different RMT approaches. The SC approach allows reducing the performance compared to the other approaches. On the other hand, it has a negative impact on the ability of RMT to detect SDCs. This is a drawback as it provides a weaker guarantee regarding the validity of the final result of an execution. But it can also be a big advantage as it much less often interrupts an execution because of a detected soft error that would not have corrupted the final result.

To have a full comparison of the RMT approaches, we should also take into account the impact of each approach on the probability to experience a soft error. The duration of the execution and the total number of instructions executed by the applications, are two parameters that can influence this probability. From
this point of view, the SC approach is a clear winner. The results presented in Sect. 4.2 show the execution with SC can be up to 63% faster than the default RMT approach. Using the Linux tool Perf, we also measured to total number of instructions executed during several runs of HPCCG with the three approaches. We observed that the SC solution reduces the number of instructions to be executed by about 12% compared to the default approach (while the aggregate optimization has not impact on this number). A dedicated analysis would be required to understand the impact of these numbers on the real probability of experiencing soft errors in HPC applications.

5 Conclusion

The paper studies the use of RMT to detect SDCs in HPC applications. To improve the performance of RMT, we evaluate two techniques to reduce the amount of data that need to be exchanged between the replicated threads: aggregation and selective checking. Our evaluation with representative workloads shows that selective checking, which consists in only verifying for correctness the content of variables that can impact the global state of the application, is a promising solution. When running on a multicore processor that features SMT threads, it leads to a performance overhead that can be as low as 1.4× while still ensuring a good error coverage. As future work, we plan to study in details the impact of these optimizations on the probability of experiencing a soft error, to better understand the trade-offs that they offer.

References

1. Bautista-Gomez, L., Zyulkyarov, F., Unsal, O., McIntosh-Smith, S.: Unprotected computing: a large-scale study of DRAM raw error rate on a supercomputer. In: IEEE Supercomputing (2016)
2. 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: ACM HPDC (2015)
3. Berrocal, E., Bautista-Gomez, L., Di, S., Lan, Z., Cappello, F.: Toward general software level silent data corruption detection for parallel applications. IEEE TPDS 28(12), 3642–3655 (2017)
4. Calhoun, J., Olson, L., Snir, M.: FlipIt: an LLVM based fault injector for HPC. In: EuroPar (2014)
5. 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: IEEE Supercomputing (2012)
6. Kuvaiskii, D., Faqeh, R., Bhatotia, P., Felber, P., Fetzer, C.: HAFT: hardware-assisted fault tolerance. In: ACM Eurosys (2016)
7. Laguna, I., Schulz, M., Richards, D.F., Calhoun, J., Olson, L.: IPAS: intelligent protection against silent output corruption in scientific applications. In: CGO (2016)
8. Mitropoulou, K., Porpodas, V., Jones, T.M.: COMET: communication-optimised multi-threaded error-detection technique. In: ACM CASES (2016)
9. Mitropoulou, K., Porpodas, V., Zhang, X., Jones, T.M.: Lynx: using OS and hardware support for fast fine-grained inter-core communication. In: ACM ICS (2016)
10. Porter, L., et al.: Making the most of SMT in HPC: system-and application-level perspectives. ACM TACO 11(4) (2015)
11. Reinhardt, S.K., Mukherjee, S.S.: Transient fault detection via simultaneous multithreading. In: IEEE Symposium on Computer Architecture (2000)
12. Snir, M., et al.: Addressing failures in exascale computing. IJHPCA 28(2), 129–173 (2014)
13. Sridharan, V., et al.: Memory errors in modern systems: the good, the bad, and the ugly. ACM SIGPLAN Notices 50, 297–310 (2015)
14. Wang, C., Kim, H.S., Wu, Y., Ying, V.: Compiler-managed software-based redundant multi-threading for transient fault detection. In: ACM CGO (2007)
15. Yu, J., Garzaran, M.J., Snir, M.: ESoftCheck: removal of non-vital checks for fault tolerance. In: CGO (2009)
16. Zhang, Y., Lee, J.W., Johnson, N.P., August, D.I.: DAFT: decoupled acyclic fault tolerance. Int. J. Parallel Prog. 40(1), 118–140 (2012)