Master-Seminar: Hochleistungsrechner - Aktuelle Trends und Entwicklungen
Winter Term 2017/2018

Fault Tolerance in High Performance Computing

Simon Klotz
Technische Universität München

Abstract

Fault tolerance is one of the essential characteristics of High-Performance Computing (HPC) systems to reach exascale performance. Higher failure rates are anticipated for future systems because of an increasing number of components. Furthermore, new issues such as silent data corruption arise. There has been significant progress in the field of fault tolerance in the past few years, but the resilience challenge is still not solved. The most widely used global rollback-recovery method does not scale well enough for exascale systems which is why other approaches need to be considered. This paper gives an overview of established and emerging fault tolerance methods such as rollback-recovery, forward recovery, algorithm-based fault tolerance (ABFT), replication and fault prediction. It also describes how LAIK can be utilized to achieve fault tolerant HPC systems.

1 Introduction

Fault tolerance is the ability of an HPC system to avoid failures despite the presence of faults. Past HPC systems did not provide any fault tolerance mechanisms since faults were considered rare events [1]. However, with the current and next generation of large-scale HPC systems faults will become the norm. Due to power and cooling constraints, the increase in clock speed is limited, and it is expected that the number of components per system will continue to grow to improve performance [2] following the trend of the current Top500 HPC systems [3]. A higher number of components makes it more likely that one of them fails at a given time. Furthermore, the decreasing size of transistors increases their vulnerability.

The resilience challenge [4] encompasses all issues associated with an increasing number of faults and how to deal with them. Commonly used methods such as coordinated checkpointing do not scale well enough to be suited for exascale environments. Thus, the research community is considering other approaches such as ABFT, fault prediction or replication to provide a scalable solution for future HPC systems. This paper presents existing and current research in the field of fault tolerance in HPC and discusses their advantages and drawbacks.

The remainder of this paper is structured as follows: Section 2 introduces the terminology of fault tolerance. Fault recovery methods such as rollback-recovery, forward recovery and silent error detection are covered in Section 3. Section 4 discusses fault containment methods such as ABFT and replication. Section 5 is concerned with fault avoidance methods including failure prediction and migration. Finally, Section 6 presents LAIK, an index space management library, that enables all three classes of fault tolerance.

2 Background

First, essential concepts of the field have to be introduced for an efficient discussion of current fault tolerance methods. This paper relies on the definitions of Avizienis [5], who defines faults as causes of errors which are deviations from the correct total state of a system. Errors can lead to failures which imply that the external and observable state of a system is incorrect. Faults can be either active or dormant whereas only active faults lead to an error. However, dormant faults can turn into active faults by the computation or external factors. Furthermore, faults can be classified into permanent and transient faults. Transient faults only appear for a certain amount of time and then disappear without external intervention in contrast to permanent faults which do not disappear. Errors can be classified as detected, latent or masked. Latent errors are only detected after a certain amount of time and masked errors do not lead to failures at all. Errors that lead to a failure are also called fail-stop errors whereas undetected latent errors that only corrupt the data are called silent errors. Silent errors have been identified to be one of the greatest challenges on the way to exascale resilience [1]. As an example, cosmic radiation impacting memory and leading to a discharge would be a fault. Then, the affected bits flip which would be an error. If an application now uses these corrupted bits the observable state would deviate from the correct state which is a failure.
An important metric to discuss fault tolerance mechanisms is the Mean Time Between Failures (MTBF). This paper is following the definition of Snir et al. [6] who define it as

$$MTBF = \frac{\textit{Total runtime}}{\textit{Number of failures}}$$

Several studies [7, 8] were conducted to classify failures and determine their source in HPC systems. In order to analyze failures researchers define different failure classes based on their cause. Schroeder et al. [7] distinguish between hardware, software, network, environment, human, and undetermined failures whereas Oliner et al. [8] only distinguish between software, hardware, and undetermined failures. Depending on the studied system the occurence of different failure classes differ significantly. A possible explanation is the discrepancy in system characteristics and used failure classes. Nevertheless, most researchers agree that software and hardware faults are the most frequent causes of failures [7]. Hardware faults can occur in any component of an HPC system, such as fans, CPUs, or memory. One of the most important classes of hardware faults are memory bit errors due to cosmic radiation. Software faults can occur at any part of the software stack. The most common faults include unhandled exceptions, null reference pointers, out of memory error, timeouts and buffer overflows [6]. Since hardware and software faults are most common in HPC systems, the remainder of this paper will focus on these two classes since also most fault tolerance methods apply to them.

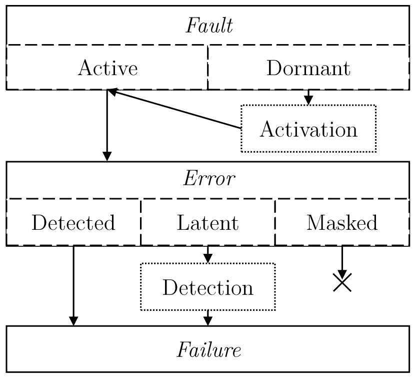

Figure 1: Classification of faults, errors, and failures

3 Fault Recovery

The objective of fault recovery is to recover a normal state after the application experienced a fault, thus preventing a complete application failure. Fault recovery methods can be divided into rollback-recovery, forward recovery, and silent error detection. Rollback-recovery is the most widely investigated approach and commonly used in many large-scale HPC systems. Rollback-recovery methods periodically take snapshots of the application state and in case of failure the execution reverts to that state. Forward recovery, avoids this restart by letting the application recover by itself using special algorithms. A novel research direction is silent error detection which can be combined with rollback-recovery and forward recovery to recover from faults.

3.1 Rollback-recovery

Rollback-recovery, which is also called checkpoint-restart, is the most widely used and investigated fault tolerance method [9, 10]. Checkpointing methods can be classified into multiple categories according to how they coordinate checkpoints and at which level of the software stack they operate. Each class has different approaches trying to solve the challenge of saving a consistent state of a distributed system which is not a trivial task to achieve since large-scale HPC systems generally do not employ any locking mechanisms. The two major approaches to checkpoint a consistent state of a HPC system are coordinated checkpointing and uncoordinated checkpointing with message-logging [10] as depicted in Figure 2.

Coordinated protocols store checkpoints of all processes simultaneously which imposes a high load on the file system. They enforce a synchronized state of the system such that no messages are in transit during the checkpointing process by flushing all remaining messages. The advantage of coordinated checkpointing is that each checkpoint captures a consistent global state which simplifies the recovery of the whole application. Furthermore, coordinated protocols have a comparatively low overhead since they do not store messages between processes [9]. However, if one process fails all other processes need to be restarted as well [11]. Also, with a growing number of components and parallel tasks the cost of storing checkpoints in a coordinated way steadily increases.

Message-logging protocols store messages between nodes with a timestamp and the associated content, thus capturing the state of the network [9]. They are based on the assumption that the application is piecewise deterministic and that the state of a process can be recovered by replaying the recorded messages from its initial state [9]. In practice, each process periodically checkpoints its state so that recovery can continue from there instead of the initial state [12]. In contrast to coordinated protocols, only the processes affected by failure need to be restarted which makes it more scalable. However, each process needs to maintain multiple checkpoints which increases memory consumption and inter-process dependencies can lead to the so-called Domino effect [9]. Furthermore, all messages and their content need to be stored as well.

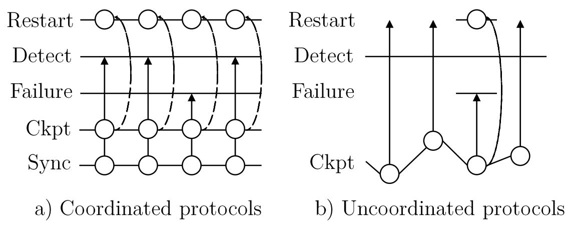

Figure 2: Difference of coordinated protocols and uncoordinated protocols with message-logging [11]

Message-logging protocols can be further divided into: optimistic, pessimistic and causal protocols [13]. Optimistic protocols store checkpoints on unreliable media, e.g. the node running a process itself. Furthermore, a process can continue its execution during logging [11]. This method has less overhead compared to others but can only tolerate a single failure at a time and if a node itself fails the whole application has to be restarted. Pessimistic protocols store all checkpoints on reliable media and processes have to halt execution until messages are logged. This enables them to tolerate multiple faults and complete node failures at the cost of a higher overhead using reliable media and halting execution. Causal protocols also do not use reliable media but log the messages in neighboring nodes with additional causality information. This omits the need for reliable media and an application is still able to recover if one node experiences a complete failure.

Checkpointing methods can operate on different levels of the software stack which has an important impact on transparency and efficiency. Checkpointing can be implemented by the operating system, a library, or the application itself [14]. Operating system extensions can do global checkpoints that provide transparent checkpointing of the whole application state and do not require modifications of the application code. There are also libraries that act as a layer on top of MPI and enable global and transparent checkpointing. However, global checkpointing has a large overhead since the storage costs are proportional to the memory footprint of the system [15] which often includes transient data or temporary variables which are not critical for recovery. Approaches such as compiler-based checkpointing intend to solve this issue by automatically detecting transient variables and excluding them from checkpoints which reduces the total size of checkpoints.
All of the previously mentioned approaches automatically store checkpoints which does not allow any control of the procedure. User-level fault tolerance libraries assume that developers know best when to checkpoint and what data to store which can significantly decrease the checkpoint size. However, this also increases application complexity [15] and an application developer can be mistaken and forget about critical data.
Researchers introduced different variations and additions to the general checkpointing techniques to reduce the number of needed resources to reach feasible performance for exascale HPC systems. One of these approaches is diskless checkpointing [16] which stores checkpoints in memory. Other approaches include multi-level checkpointing [17] which hierarchically combines multiple storage technologies with different reliability and speed allowing a significantly higher checkpointing frequency since checkpointing on the first layer is generally fast.

3.2 Forward recovery

Forward recovery avoids restarting the application from a stored checkpoint and lets the application recover by itself. It is only applicable for algorithms that can accept faults but ultimately converge to a correct state, sometimes at the cost of additional computations. Forward recovery algorithms are non-masking since they can display erratic behavior due to transient faults but finally display legitimate behavior again after a stabilization period [18]. One class of algorithms having this property are self-stabilizing algorithms.
Forward recovery can also be manually implemented by application developers. Resilient MPI implementations provide an interface which can be used by application developers to specify behavior in case a fault occurs. Using this, developers can implement custom forward recovery routines that recover a consistent, fault-free application state. The advantage of forward recovery methods is that they often allow faster recovery compared to rollback-recovery. However, they are application specific and not applicable to all algorithms.

3.3 Silent Error Detection

The objective of silent error detection methods is to detect latent errors such as multi-bit faults. Silent error detection is a relatively novel research area of increasing importance since smaller components are more vulnerable to cosmic radiation [1]. It is application specific and uses historic application data to detect anomalous results during the computations of an application which indicates a silent data corruption (SDC).
Several methods can be used to detect anomalies in the application data. Vishnu et al. [19] use different machine learning methods able to correctly detect multi-bit errors in memory with a 97% chance. After a silent error is detected it has to be corrected. Gomez et al. [20] just replace corrupted values with their predictions. Since it is not always desirable to use approximate predictions, rollback-recovery can be used, to revert to a consistent application state instead. Application developers can also implement application-specific routines to recover a normal state.
One drawback of silent error detection is that it introduces computational overhead. Another limita-

tion is the false positive rate which has a significant impact on the overall performance of an application. Furthermore, it takes significant effort to develop silent error detection methods since they are application specific. Nevertheless, silent error detection is one of the few methods able to deal with silent errors and has less resource overhead compared to replication [21].

4 Fault Containment

Fault containment methods let an application continue to execute until it terminates even after multiple faults occur. Compensation mechanisms are in place to contain the fault and avoid an application failure. They also ensure that the application returns correct results after termination. The most commonly used fault containment methods are replication and ABFT. Replication systems run copies of a task on different nodes in parallel and if one fails others can take over providing the correct result. ABFT includes all algorithms which can detect and correct faults, that occur during computations.

4.1 Replication

Replication methods replicate all computations on multiple nodes. If one replica is hit by a fault the others can continue computation and the application terminates as expected.

Replication methods can be distinguished into methods that either deal with fail-stop or silent errors. MR-MPI [22] deals with fail-stop errors and uses replicas for each node. As soon as one node fails the replica is taking over and continues execution. Only if both nodes fail the application needs to be restarted. In contrast, RedMPI [23] is not addressing complete node failures but silent errors. It compares the output of all replicas to correct SDCs.

A novel research direction is to use approximate replication to detect silent errors. A complementary node is running an approximate computation which is then compared to the actual result to detect SDCs. Benson et al. [24] use cheap numerical computations to verify the original results which reduces the overhead compared to other replication methods.

Replication generally has a high overhead because multiples of all resources are needed depending on the replication factor. Furthermore, it is costly to check the consistency of all replications and it is difficult to manage non-deterministic computations [25]. Also, an overhead of messages is needed to communicate with all replicas. Nevertheless, Ferreira et al. [25] showed that replication methods outperform traditional checkpointing methods if the MTBF is low.

4.2 ABFT

ABFT was first proposed by Huang and Abraham in 1984 [26] and includes algorithms able to automatically detect and correct errors. ABFT is not applicable to all algorithms but researchers are continuously adapting new algorithms to be fault tolerant because of the generally low overhead compared to other methods. The core principle of ABFT is to introduce redundant information which can later be used to correct computations.

ABFT algorithms can be distinguished into online and offline methods [11]. Offline methods correct errors after the computation is finished whereas online methods repair errors already during the computation. The most prominent examples for offline ABFT are matrix operations [27] where checksum columns are calculated which are later used to verify and correct the result. Online ABFT can also be applied to matrix operations [28] whereas they are often more efficient than their offline counterparts since they can correct faults in a timely manner, when their impact is still small, compared to offline ABFT methods that correct faults only after termination.

The main drawback of ABFT is that it requires modifications of the application code. However, since most applications rely on low-level algebra libraries the methods can be directly implemented in those libraries hidden from an application programmer. Another disadvantage of ABFT is that it cannot compensate for all kinds of faults, e.g.

complete node failures. Furthermore, if the number of faults during the computation exceeds the capabilities of the algorithm it cannot correct them anymore. ABFT also introduces overhead in the computations which is however comparatively small to other fault tolerance methods.

5 Fault Avoidance

In contrast to fault recovery and fault containment, fault avoidance or proactive fault tolerance methods aim at preventing faults before they occur. Generally, fault avoidance methods can be split in two phases. First, predicting the time, location and kind of upcoming faults. Second, taking preventive measures to avoid the fault and continue execution. Proactive fault tolerance is a comparatively new but promising field of research [1]. The prediction of fault occurrences mainly relies on techniques from Data Mining, Statistics and Machine Learning which use RAS log files and sensor data. The most common preventive measure is to migrate tasks running on components which are about to fail to healthy components.

5.1 Fault Prediction

It is important to consider which data and model to use to accurately predict faults. Engelmann et al. [29] derived four different types of health monitoring data that can be used to predict faults whereas most prediction methods rely on Type 4 which utilizes a historical database that is used as training data for machine learning methods. It mostly includes sensor data such as fan speed, CPU temperature and SMART metrics but also RAS logfiles containing failure histories.
The success of fault avoidance methods mainly depends on the accuracy of the predictions. It is important that predictors can predict both location and time of a fault in order to successfully migrate a task [30]. Furthermore, the accuracy of the prediction is of high importance. If too many faults are missed the advantage of a prediction-based model is small compared to other approaches such as rollback-recovery. On the other hand, if the predictor detects too many false positives too much time is spent on premature preventive measures. Models used for fault prediction mainly include Machine Learning models [31] and statistical models [32].

5.2 Migration

Once a fault is predicted measures have to be taken to prevent it from happening. The most common approaches for prevention are virtualization-based migration and process migration which both move tasks from failing nodes to healthy ones. Both approaches can be divided into stop&copy and live migration [33]. Stop&copy methods halt the execution as long as it takes to copy the data to a target node which depends on the memory footprint, whereas live migration methods continue execution during the copy process reducing the overall downtime.
Both virtualization and process-based methods need available nodes for migration. They can either be designated spare nodes, which are expected to become a commodity in future HCP systems, unused nodes, or the node with the lowest load [33]. Two tasks sharing the same node can however result in imbalance and an overall worse performance. Most approaches generally use free available nodes first, then spare nodes, and as a last option the least busy node [33].
Nagarajan et al. [34] use Xen-virtualization-based migration while there approach is generally transferrable to other virtualization techniques as well. Each node runs a host virtual machine (VM) which is responsible for resource management and contains one or more VMs running the actual application. Xen supports live migration of VMs [35] which allows MPI applications to continue to execute during migration.
Process-based migration [36] has been widely studied on several operating systems. It generally follows the same steps as virtualization-based migration but instead of copying the state of a VM it copies the process memory [33]. Process-based migration also supports live migration.

Applications do not need to be modified for both process-based and VM-based migration. Virtualization offers several advantages such as increased security, customized operating system, live migration, and isolation. Furthermore, VMs can be more easily started and stopped compared to real machines [37]. Despite of these advantages virtualization-based techniques have not been widely adopted in the HPC community because of their larger overhead.

Fault avoidance cannot fully replace other approaches such as rollback-recovery since not all faults are predictable. Nevertheless, fault avoidance can be combined with other approaches such as ABFT, replication and rollback-recovery. Since it can significantly increase the MTBF it allows an application to take less checkpoints which is shown by Aupy et al. [38] who study the impact of fault prediction and migration on checkpointing.

6 LAIK

LAIK (Leichtgewichtige Anwendungs-Integrierte Datenhaltungs Komponente) [39] is a library currently under development at Technical University Munich which provides lightweight index space management and load balancing for HPC applications supporting application-integrated fault tolerance. Its main objectives are to hide the complexity of index space management and load balancing from application developers, increasing the reliability of HPC applications.

LAIK decouples data decomposition from the application code so that applications can adapt to a changing number of nodes, computational resources and computational requirements. The developer needs to assign application data to data containers using index spaces. Whenever required, LAIK can trigger a partitioning algorithm which distributes the data across nodes where the respective data container is needed. This allows higher flexibility in regard to fault tolerance and scheduling mechanisms. It also avoids application specific code which can become increasingly complex and hard to maintain. Furthermore, LAIK is designed to be portable by defining an interface which supports multiple communication backends and allowing application developers to incrementally port their application to LAIK. Figure 3 shows an overview of how LAIK is integrated in HPC systems.

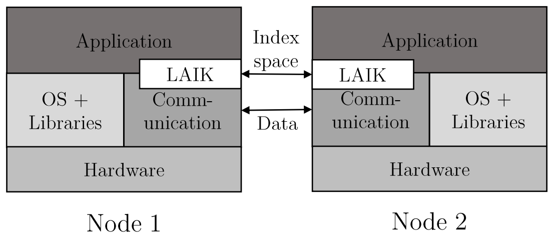

Figure 3: Integration of LAIK in HPC systems [39]

The capabilities of LAIK can be utilized for application-integrated fault tolerance. It is possible to implement fault recovery, fault containment and fault avoidance methods as layers on top of LAIK using its index space management and partitioning features.

LAIK can be used for local checkpointing by maintaining copies of relevant data structures on neighboring nodes [39]. Once a node failure is detected, the application can revert to the checkpointed state of these copies and resume execution from there on a spare node using LAIKs partitioning feature.

Furthermore, it is possible to implement replication in a similar way as checkpointing using LAIK. Again multiple redundant copies of the data containers are maintained on different nodes whereas one node acts as the master node. Instead of only using the redundant data after a failure as in checkpointing, tasks can be run in parallel on each replicated data container. Additional synchronization mechanisms have to be utilized to keep the replicas in a consistent state. In case the master node experiences failure one of the replicas becomes the new master node and execution continues without interruption.

LAIK also allows proactive fault tolerance. Once a prediction algorithm predicts a failure the nodes are synchronized, execution is stopped, and the failing node is excluded. Then, LAIK triggers the repartitioning algorithm and execution can be continued on a different node preventing a complete application failure.

7 Conclusion

Fault tolerance methods aim at preventing failures by enabling applications to successfully terminate in the presence of faults. Several distinct fault tolerance methods were developed to solve the resilience challenge. Since the most common method global rollback-recovery is not feasible for future systems new methods have to be considered. However, currently no other method is able to fully replace rollback-recovery. Nevertheless, methods such as user-level fault tolerance, multi-level checkpointing and diskless checkpointing aim at increasing the efficiency of checkpointing. Also, additional methods such as replication, failure prediction, and process migration can be used complementary to rollback-recovery to decrease the MTBF and application specific methods such as ABFT, forward recovery and silent error detection can further improve the fault tolerance.
It is challenging to use the various existing fault tolerance libraries complementary since they rely on different technologies. In order to reach exascale resilience an integrated approach, which is able to combine fault tolerance methods, is of high importance. LAIK has the capabilities to act as a basis for such an approach. Researchers could build layers on top of LAIK implementing new methods in a standardized and compatible way utilizing LAIKs features. Furthermore, additional partitioning algorithms, data structures and communication backends could be implemented because of LAIKs modular architecture. This would enable a complementary use of multiple fault tolerance methods to significantly increase the overall fault tolerance of HPC systems.

References

[1] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, “Toward exascale resilience: 2014 update,” *Supercomputing frontiers and innovations*, vol. 1, no. 1, pp. 5–28, 2014.

[2] J. Shalf, S. Dosanjh, and J. Morrison, “Exascale computing technology challenges,” in *International Conference on High Performance Computing for Computational Science*, pp. 1–25, Springer, 2010.

[3] E. Strohmaier, “Top500 supercomputer,” in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC ’06, (New York, NY, USA), ACM, 2006.

[4] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, *et al.*, “Exascale computing study: Technology challenges in achieving exascale systems,” 2008.

[5] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004.

[6] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, *et al.*, “Addressing failures in exascale computing,” *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[7] B. Schroeder and G. A. Gibson, “Understanding failures in petascale computers,” in *Journal of Physics: Conference Series*, vol. 78, p. 012022, IOP Publishing, 2007.

[8] A. Oliner and J. Stearley, “What supercomputers say: A study of five system logs,” in *Dependable Systems and Networks, 2007. DSN’07. 37th Annual IEEE/IFIP International Conference on*, pp. 575–584, IEEE, 2007.

[9] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, “A survey of rollback-recovery

protocols in message-passing systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375–408, 2002.

[10] G. Bosilca, A. Bouteiller, E. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Herault, Y. Robert, F. Vivien, and D. Zaidouni, "Unified model for assessing checkpointing protocols at extreme-scale," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 17, pp. 2772–2791, 2014.

[11] F. Cappello, "Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities," *The International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 212–226, 2009.

[12] S. Rao, L. Alvisi, and H. M. Vin, "The cost of recovery in message logging protocols," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 2, pp. 160–173, 2000.

[13] L. Alvisi and K. Marzullo, "Message logging: Pessimistic, optimistic, causal, and optimal," *IEEE Transactions on Software Engineering*, vol. 24, no. 2, pp. 149–159, 1998.

[14] E. Roman, "A survey of checkpoint/restart implementations," in *Lawrence Berkeley National Laboratory, Tech*, Citeseer, 2002.

[15] J. Dongarra, T. Herault, and Y. Robert, "Fault tolerance techniques for high-performance computing," in *Fault-Tolerance Techniques for High-Performance Computing*, pp. 3–85, Springer, 2015.

[16] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, 1998.

[17] N. H. Vaidya, "A case for two-level distributed recovery schemes," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 23, pp. 64–73, ACM, 1995.

[18] S. Tixeuil, "Self-stabilizing algorithms," in *Algorithms and theory of computation handbook*, pp. 26–26, Chapman & Hall/CRC, 2010.

[19] A. Vishnu, H. van Dam, N. R. Tallent, D. J. Kerbyson, and A. Hoisie, "Fault modeling of extreme scale applications using machine learning," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pp. 222–231, IEEE, 2016.

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

[21] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello, "Lightweight silent data corruption detection based on runtime data analysis for hpc applications," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 275–278, ACM, 2015.

[22] C. Engelmann and S. Böhm, "Redundant execution of hpc applications with mr-mpi," in *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*, pp. 15–17, 2011.

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

[24] A. R. Benson, S. Schmit, and R. Schreiber, "Silent error detection in numerical time-stepping schemes," *The International Journal of High Performance Computing Applications*, vol. 29, no. 4, pp. 403–421, 2015.

[25] K. Ferreira, J. Stearley, J. H. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pp. 1–12, IEEE, 2011.

[26] K.-H. Huang *et al.*, "Algorithm-based fault tolerance for matrix operations," *IEEE transactions on computers*, vol. 100, no. 6, pp. 518–528, 1984.

[27] Z. Chen and J. Dongarra, "Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 10–pp, IEEE, 2006.

[28] P. Wu, C. Ding, L. Chen, F. Gao, T. Davies, C. Karlsson, and Z. Chen, "Fault tolerant matrix-matrix multiplication: correcting soft errors on-line," in *Proceedings of the second workshop on Scalable algorithms for large-scale systems*, pp. 25–28, ACM, 2011.

[29] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott, "Proactive fault tolerance using preemptive migration," in *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pp. 252–257, IEEE, 2009.

[30] A. Gainaru, F. Cappello, M. Snir, and W. Kramer, "Fault prediction under the microscope: A closer look into hpc systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 77, IEEE Computer Society Press, 2012.

[31] E. W. Fulp, G. A. Fink, and J. N. Haack, "Predicting computer system failures using support vector machines.," *WASL*, vol. 8, pp. 5–5, 2008.

[32] L. Yu, Z. Zheng, Z. Lan, and S. Coghlan, "Practical online failure prediction for blue gene/p: Period-based vs event-driven," in *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, pp. 259–264, IEEE, 2011.

[33] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive process-level live migration and back migration in hpc environments," *Journal of Parallel and Distributed Computing*, vol. 72, no. 2, pp. 254–267, 2012.

[34] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive fault tolerance for hpc with xen virtualization," in *Proceedings of the 21st annual international conference on Supercomputing*, pp. 23–32, ACM, 2007.

[35] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pp. 273–286, USENIX Association, 2005.

[36] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *ACM Computing Surveys (CSUR)*, vol. 32, no. 3, pp. 241–299, 2000.

[37] W. Huang, J. Liu, B. Abali, and D. K. Panda, "A case for high performance computing with virtual machines," in *Proceedings of the 20th annual international conference on Supercomputing*, pp. 125–134, ACM, 2006.

[38] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni, "Checkpointing algorithms and fault prediction," *Journal of Parallel and Distributed Computing*, vol. 74, no. 2, pp. 2048–2064, 2014.

[39] J. Weidendorfer, D. Yang, and C. Trinitis, "Laik: A library for fault tolerant distribution of global data for parallel applications," in *Konferenzband des PARS'17 Workshops*, p. 10, 2017.