Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 136 (2018) 52–61

Procedia Computer Science

www.elsevier.com/locate/procedia

7th International Young Scientist Conference on Computational Science

Three levels of fail-safe mode in MPI I/O NVRAM distributed cache

Artur Malinowski, Paweł Czarnul

Dept. of Computer Architecture,
Faculty of Electronics, Telecommunications and Informatics,
Gdańsk University of Technology, Gdansk, Poland

Abstract

The paper presents architecture and design of three versions for fail-safe data storage in a distributed cache using NVRAM in cluster nodes. In the first one, cache consistency is assured through additional buffering write requests. The second one is based on additional write log managers running on different nodes. The third one benefits from synchronization with a Parallel File System (PFS) for saving data into a new file which allows to keep file history at the cost of space. We have shown that the three level fail-safe mode incorporating these versions does introduce minimal overhead for a random walk microbenchmark application for a 1GB file and checkpoints created every 2000 iterations, computing powers of a graph with 10000 vertices and up to 20% overhead for parallel processing of images up to 1000 megapixels compared to the basic NVRAM cache without fail-safe modes. We also presented times for checkpoint creation and restoring for sizes up to 10GBs.

Keywords: NVRAM, parallel MPI I/O extension, fail safe, distributed cache

1. Introduction

Failures in computer systems are inevitable and we are forced to live with them. As a failure is usually connected with some kind of loss, we should design a system in such a way that the former will be minimized while taking into account time and space constraints of data loss prevention mechanisms. One of the most valuable artifacts, that could be protected from a damage, is data. Depending on how valuable the data is, and how frequent a failure might be, different protection techniques should be employed. Protecting data in personal computers using occasionally created manual backups is sufficient enough for most users. Mobile devices like laptops or phones are often equipped with

E-mail address: artur.malinowski@pg.edu.pl, pczarnul@eti.pg.edu.pl

10.1016/j.procs.2018.08.237

software for automatic backup, because of high risk of damage or theft. Many computers used in business may contain such valuable data that companies even create specialized departments focused on data safety.

In many High Performance Computing (HPC) applications keeping data safe is also a priority task. With increasing sizes of clusters (nowadays clusters contain even thousands of nodes) the probability of failure grows [22, 8], while high computing cost makes data really valuable. Moreover, with some long running applications, keeping data safe makes it possible to rerun computations from a certain point instead of starting over and over from the beginning. And finally, it is also desirable to design applications in a way that they can perform their tasks even if some of the nodes are unavailable [8].

To keep data safe during processing, many HPC applications use checkpointing – saving the state of an application, so it can be restored after a failure has occurred [8]. This technique can be implemented either manually or incorporated into an application using a variety of libraries. Manual implementation is often based on the idea of saving only important data into files located on a Parallel File System (PFS, e.g. Lustre, GPFS, OrangeFS), automatic solutions could be even capable of freezing the whole process (with its memory stack, opened sockets, pending communication etc.) and rerunning it after a failure has occured and the system is restored [8]. More detailed checkpointing description is included in Section 2.

Moreover, many HPC applications, especially data intensive ones, process files. With huge data volumes, a shared file could be used not only for storing the data, but also for communication between processes. Although a file located on a PFS may seem safe during processing, to the best of our knowledge no modern production distributed system provides concurrent transactional file updates. Moreover, pending file operations are often not considered by checkpointing libraries. As a result, a programmer is forced to keep a file safe manually, for example by creating redundant copies.

During our research we focused on safety of files in HPC applications. As Message Passing Interface (MPI) is one of the most popular parallel programming standards, we decided to restrict our solution to its file access API named MPI I/O. In this paper, we propose three levels of fail-safe mode built on top of our previously developed extension of the MPI I/O: NVRAM (non-volatile, random access memory) based distributed cache. The basic fail-safe mode, that was already presented [17], is significantly enhanced making the whole solution valuable due to easy to use and low overhead mechanisms for creating checkpoints and keeping file in consistent state during processing.

2. Related work

There are many techniques to create fault-tolerant applications and they vary depending on selected technology stack. An idea, applicable to programs designed in the master-slave paradigm, is an implementation using the client-server architecture. In such a case the master distributes tasks to all available slaves and when it does not receive response in a given time or communication channel becomes broken, task is assigned to another slave. Although it still requires additional data-safety mechanisms on a master, it is applicable to HPC in general, in partitcular using MPI [10]. Another wide range of methods are connected with replication of computations or error prevention based on fault prediction [6].

Checkpointing, one of the most popular fault tolerance technique, is a general term that encompasses many different solutions. In their work [25], Walters and Chaudhary classified these into four major levels: hardware, kernel, user and application. Because of their cost, hardware based solutions are used mainly in systems where data-safety is extremely valuable, such as spacecraft control computers [26]. A well known example of a kernel-level checkpoint system is Berkeley Lab Checkpoint/Restart (BLCR) – a comprehensive and high quality solution that focuses on MPI based applications [11]. Although transferring responsibility for checkpoint creation to the operating system reduces configuration effort and does not require additional development, it was shown that well-designed user-level and application-level solutions could achive better performance [3]. An example of a user-level library could be ckpt or libckpt tools [20] with further improvements [3]. Application-level solutions are often related more to the program architecture, such as CPPC framework extended to exploit the User Level Failure Mitigation (ULFM) [12] or checkpointing for applications using MPI one-sided API [7].

Nowadays, many research papers focus on checkpointing optimizations for large scale computations. In 2013, Rajachandrasekar et al. proposed a novel user-space file system that, combined with a checkpoint library, achieved a bandwidth of 1 PB/s when checkpointing three million MPI processes [21]. Optimizations designed by Di et al. in

2014, that included proper calculation of checkpoint interval times and the optimal selection of memory levels, improved checkpointing performance up to 50% compared to other solutions [4]. Recent deep analysis of failure distributions and their future predictions results in an idea that checkpointing may be insufficient and it would be beneficial to combine it with process replication [1]. A new direction of research related to checkpointing is also connected to optimizations that care not only about performance, but also take into consideration energy consumption [18].

Checkpointing, as well as other fault tolerance techniques, may benefit from emerging memory technologies. Already in 2009 3D-PCRAM was expected to reduce checkpointing overhead [5], in 2012 feasibility of using NVRAM to store the whole application state was discussed [19]. Moreover, after Intel's press release related to 3D XPoint technology, that suggest appearance of NVRAM devices on the market in the nearest future [2], the topic has became more popular [9, 23]. The new memory is expected to exceed DRAM capacity, be fast enough for direct access and omit page cache, and be persitent [24].

3. Proposed solution

3.1. MPI I/O supported by NVRAM cache

The idea behind the extension to MPI I/O we proposed previously is based on caching the data within buffers located in node-local NVRAM [16]. The motivation was not only achieving better performance compared to unmodified implementation, but also making the development process easier. Acceleration of I/O operations (read from and write to a file located on a PFS) could be evaluated in the following terms:

- high scalability achieved by a fully decentralized architecture and utilizing both NVRAM and computational power of all of the nodes,
- low request processing time that comes from minimal meta-data and prefetching the whole file into the cache during the initialization phase,
- acceptable CPU time overhead – processing does not involve any computationally intensive algorithms,
- simplicity of processing – each node-local cache data is separate, which makes implementation of concurrent file modification easier and allows to skip cache coherence mechanisms.

The entity responsible for file access is the cache manager – a background thread running on each node within a cluster that serves requests and manages NVRAM buffers. Although the solution is universal, to obtain the best performance application ought to be data intensive, access relatively small data chunks (in other words, make frequent file requests), and run long enough to compensate for the overhead of initialization and deinitialization (which is typical for HPC programs). Prefetching the whole file results in the maximum file size limited to the sum of NVRAM devices' capacities in a cluster, but it should not be a problem due to their expected capacities significantly exceeding those of RAM. Our original solution gave promising results in several applications, among others: matrix multiplication applied for graph domain [13], processing large images [14] or – with minor enhancements – simulation of a crowd behavior [15].

In order to ensure that using our solution would not introduce any additional development effort, we made it compatible with MPI I/O API. Moreover, in typical applications, a developer is forced to implement techniques that reduce the number of file requests, e.g. data buffering or data staging. Our solution handles such optimizations automatically, so it is no longer required to implement it manually. Currently, we covered only a subset of MPI I/O methods, but it should be enough for preparing applications of any type.

3.2. Three level fail-safe mode

The previously proposed extension contained only a single-level of fail safety, described in detail in the next subsection. Although with negligible time overhead and fast cache initialization after a failure, the solution was very limited – the data stored within a cache had no redundant copy, so in case of a hardware failure related to an NVRAM device, there was no way to recreate the processed file. Moreover, the previous solution lacks support for file checkpointing. With our contribution presented in this paper, the fail-safe mode is extended to eliminate those disadvantages. The

three level fail-safe mode may be either a comprehensive solution to application checkpointing, if the application stores all of its state within a file, or it may be used as a complementary method for checkpointing techniques that do not put emphasis on safety of file operations. To our best knowledge, it is the only solution that benefits from expected properties of emerging byte addressable NVRAM.

3.2.1. Cache consistency within a node

Cache consistency technique is based on buffering all pending write requests before storing them in the cache. In case a write request ends successfully, the copy of a request is removed from a buffer. When an application crashes, all pending requests stored in a buffer could be resubmitted. This situation is illustrated in Figure 1. An advantage of this solution is keeping the NVRAM cache in a consistent state, which means that after a failure we are able to omit long cache initialization and restart the application almost immediately (it is also useful for files opened in the read-only mode). On the other hand, as the buffer is located on a node-local NVRAM, this method works well only for failures that do not eliminate the node from further processing, i.e. software, network, environment or human failures. As we assume NVRAM to be byte-addressable, the solution is efficient especially for small requests.

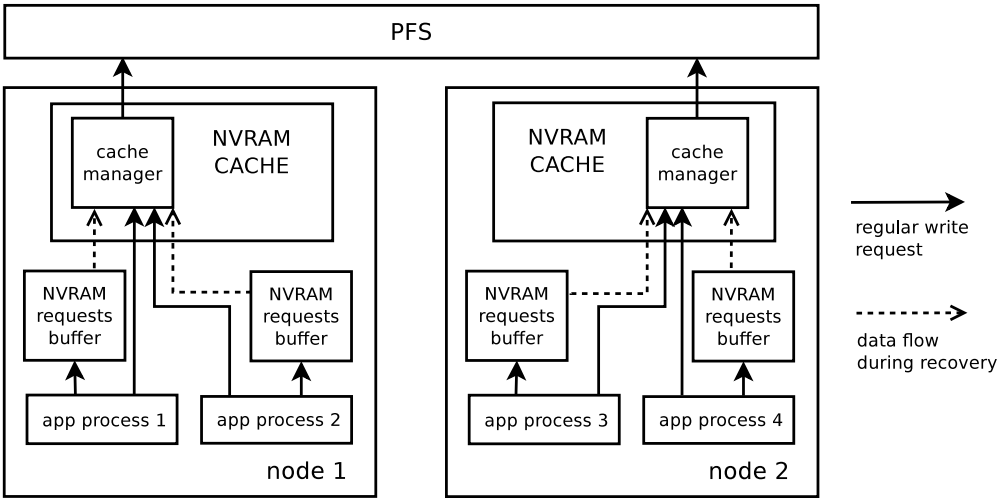


Fig. 1: Architecture of cache consistency assurance. At first each write request is stored in a buffer and only after that sent to the cache manager. It is visible that after a failure only cache is recovered – flushing the data from cache to PFS is the application's responsibility.

The buffer is created with the `libpmem` library from Persistent Memory Development Kit[1] and preallocated to accelerate smaller requests. As already stated, this technique was previously described in detail [17].

3.2.2. Write requests redundant log

To keep a file consistent after a hardware failure, all of the data is required to be stored redundantly. For that reason we introduced a background thread (called the write log manager) responsible for storing the redundant data, which is running on each node within a cluster. When the cache manager starts processing a write request, it sends a copy of a request to the write log manager located on the next neighbor node according to a logical ring topology, and the write log manager adds the data to its journal. In case of a failure that eliminates a node from further processing, application crashes and it is required to start data recovery process. At the beginning, the write log manager applies all of the requests to the file. Then, during the data recovery phase, a file located in the PFS is recreated using either data stored within cache managers, or – if the cache manager is not operational – using data from the write log manager. Figure 2 presents the architecture of this solution. For larger clusters, where there is a possibility to lose more than a single node during one failure, it may be useful to make more than one redundant copy.

[1] http://pmem.io/pmdk/libpmem/

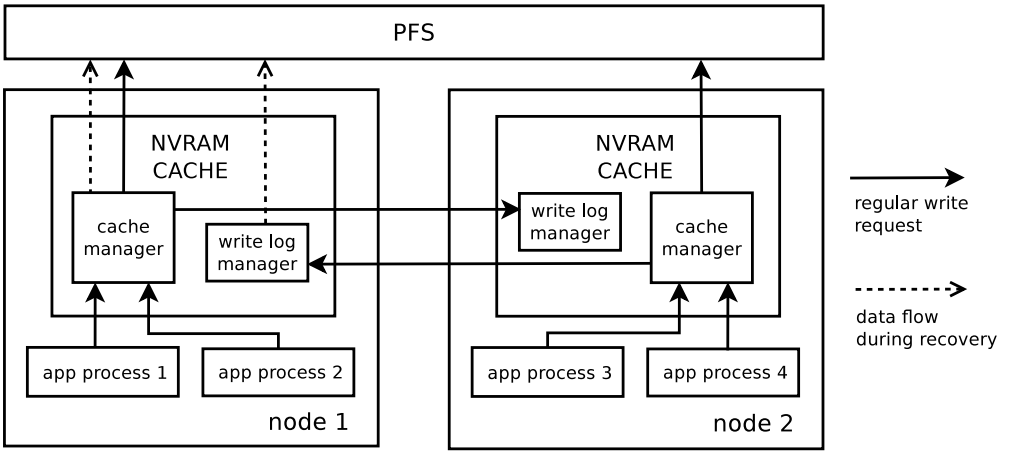


Fig. 2: Architecture of write requests redundant log. It is assumed that the failure occurred on a second node which was unable to perform further processing.

As a journal we used a file stored on NVRAM managed using `libpmem` for data and NVRAM located array managed with atomic operations using the `libpmemblk` library[2] for meta-data. As NVRAM is expected to be byte-addressable, there is no overhead for smaller requests related to block addressing.

A drawback of this method is a quick increase of the journal size, especially for highly data intensive applications. Although NVRAM is expected to highly exceed today's RAM capacity, in some scenarios it may be overflowed, because it is also used to store other cache data and may be involved in operations performed by other processes. In order to safely clear the journal, all of the data has to be pushed to the PFS, which is possible using the `MPI_File_sync` function. For that reason, a developer who wants to use this fail-safe method is forced to synchronize the cache with a PFS periodically. Another possible approach, based on a ring buffer and flushing oldest data automatically, was not used to make sure that PFS would not be unintentionally overloaded by a flood of small requests.

3.2.3. On sync checkpoint

In many cases it is valuable to keep not only a consistent state of a file, but also its history. As the architecture of NVRAM distributed cache is based on the idea of prefetching the whole file into buffers, each `MPI_File_sync` request causes writing a complete file into PFS. The single change introduced into extension is saving the data into a new file instead of replacing the old one. During recovery, the last checkpoint file located on the PFS is used. An advantage of this solution is backward compatibility with all applications that used NVRAM cache and file synchronization – in such a case, after updating the extension, checkpoints are added automatically at no cost. Figure 3 shows minimal impact of the on-sync checkpoint on the application. If the application does not use our MPI I/O extension, designers have to consider a significant overhead for each file sync call. Moreover, frequent synchronizations may cause fast storage space usage – it is required either to customize checkpoint creation frequency, or implement a mechanism of deleting unneeded file versions. The method could also be used for full application checkpointing, only if the whole program state is stored within a file.

3.2.4. Comparison

Although all three methods can be used simultaneously, there is also a possibility to use these separately. Table 1 presents main differences between proposed methods. Consistency within a single node is a perfect option for smaller clusters, where the probability of a hardware failure is low. On sync checkpoint gives a possibility to keep the history of a file, but as it requires the whole cache to be synchronized with a PFS, it may consume a lot of time for bigger files or frequent checkpoints. Using a write request log is a good solution for bigger clusters, but, because of its overhead, it is recommended only in situations for which frequent on sync checkpoint is too expensive.

[2] http://pmem.io/pmdk/libpmemblk/

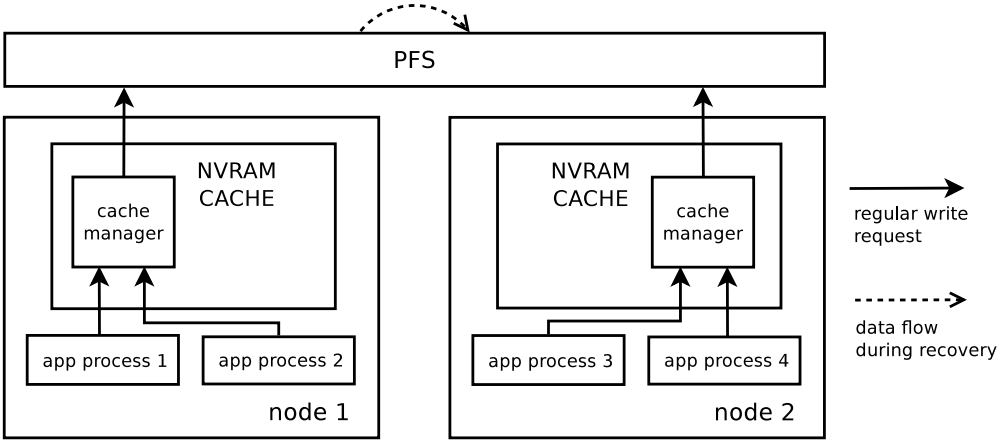


Fig. 3: Architecture of on-sync checkpoint.

Table 1: Differences between three levels of fail-safe mode

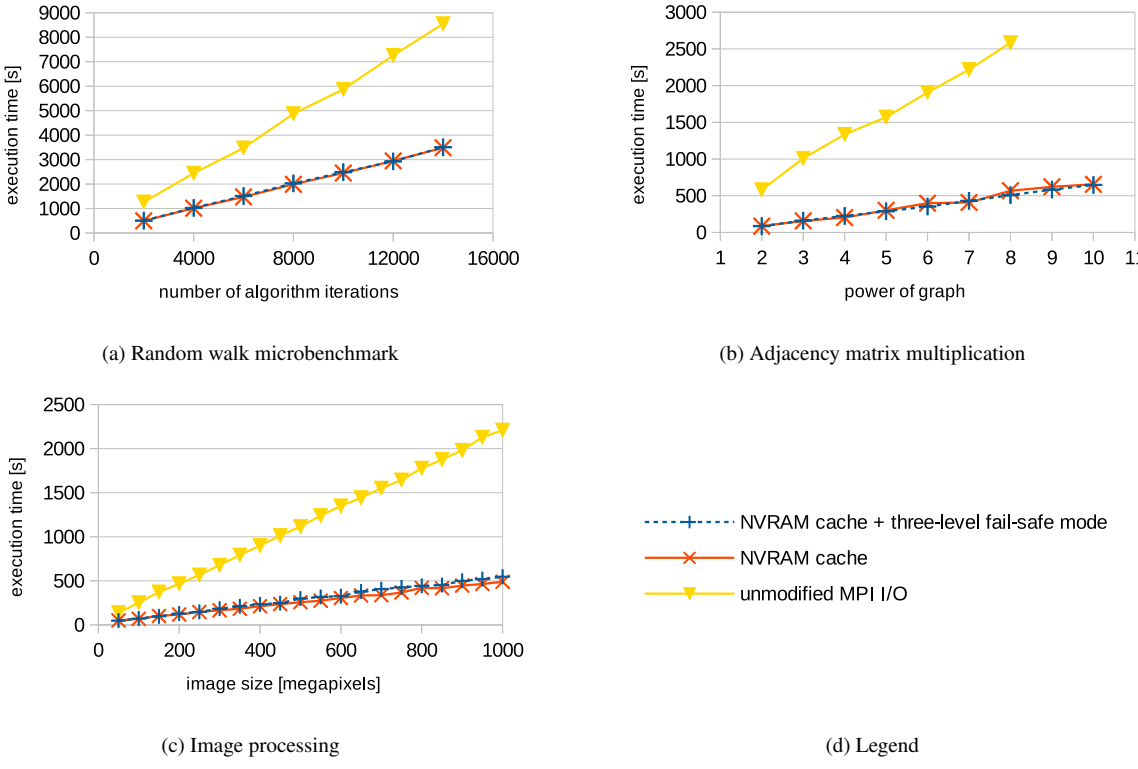
	consistency within a node	write requests log	on sync checkpoint
time overhead	negligible	small, but noticeable	depends on file size and checkpoint frequency
fault tolerance	all failure causes except hardware	all failure causes	all failure causes
restore time	negligible	long	long
additional requirements	none	frequent file synchronization	none
file history support	no	no	yes

4. Experiments

The purpose of the first part of experiments was to show that overhead related to the solution proposed in this paper is acceptable. Results also include execution times of the same application running using unmodified MPI I/O. The second part focuses on on sync checkpoint, demonstrating checkpoint creation time and the cost of cache recovery after rerunning the application from a checkpoint. The solution was also verified using a script with a set of automated functional tests, which were based on the idea of crashing the application during pending file requests and checking consistency of a file after performing the recovery procedure.

All of tests were performed on a six node cluster running MPICH 3.2 as an MPI implementation. Each node was equipped with two Intel Xeon E5-4620 CPUs, 32GB of RAM and 40Gb/s Infiniband connection. Half of the RAM, in order to simulate NVRAM realistically, was configured as follows: the bandwidth level was limited four times, each access request was elongated by 600ns of additional delay, each write request was elongated by 2000ns before the data was considered as stored persistently. As a PFS we have used OrangeFS 2.9.3 running on the two servers of equal parameters as the previously described computing nodes.

The term “three-level fail-safe mode” used in chart labels stands for NVRAM cache working with assurance of cache consistency within a node by buffering each remote request, keeping a single, redundant copy of cache data within write requests log located on another node and creating checkpoints using the MPI sync function. Checkpoint frequency is either a test case parameter, or it is specified within the chart description.



(a) Random walk microbenchmark

(b) Adjacency matrix multiplication

(c) Image processing

(d) Legend

Fig. 4: Overhead of three-level fail-safe mode in different applications

4.1. Demonstration applications

The overhead of the solution was presented for three different applications. Random walk microbenchmark is a synthetic benchmark that reads a data chunk from a random location, processes it and writes it back. In the experiment illustrated in Figure 4a we used a 1GB file and checkpoints created every 2000 iterations. It is clearly visible that in this application there is no significant delay connected to the incorporated fail-safe method.

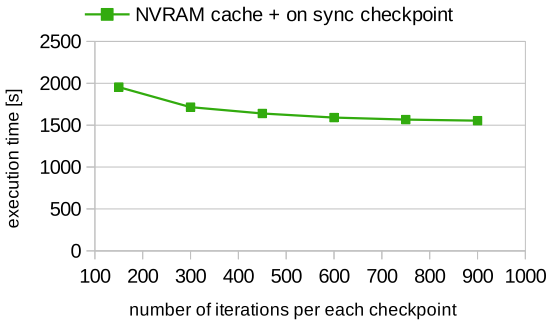
The aim of the second application was calculation the power of a graph using adjacency matrix. Practically, the core of the application is parallel matrix multiplication using Cannon's algorithm. Our input in the experiment presented in Figure 4b was a graph of 10000 vertices. Checkpoints were created after calculation each graph power. Results proved again that with correct usage of three-level fail-safe mode, the overhead is insignificant. Moreover, in this example the fail-safe mode is more stable than the base version of the NVRAM cache. The reason for such advantage is connected to implementation of two background threads instead of one. Using two background threads, synchronous waiting of a cache manager (implemented in MPICH using a busy-waiting technique) was inefficient, therefore asynchronous waiting was introduced to the cache manager and the write log manager.

The third example was an application for large image processing. The variant demonstrated in Figure 4c applied a blur multipass filter on images of different sizes. The algorithm was configured to perform a blur ten times, each pass of the filter ended with creating a checkpoint. As the application was more data intensive than the two previously described, we can observe an overhead related to the proposed fail-safe mode. The maximum overhead we obtained for this application was about 20% compared to the base NVRAM cache – we consider it acceptable, especially when compared also to the unmodified MPI I/O execution times.

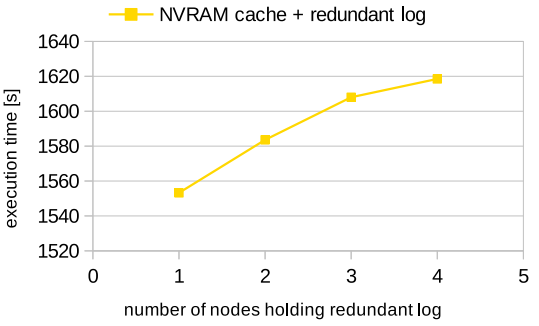
4.2. Solution configurations

The next set of charts present different configurations of the proposed modes and their comparison. All three experiments were performed with the random walk microbenchmark that operated on a 1GB file. The first two tests were prepared with 6000 iterations of the algorithm. Figure 5a shows reduction of application execution time with decreasing frequency of checkpoint creation, while Figure 5b illustrates growth of execution time with an increasing number of nodes holding redundant copies of submitted requests.

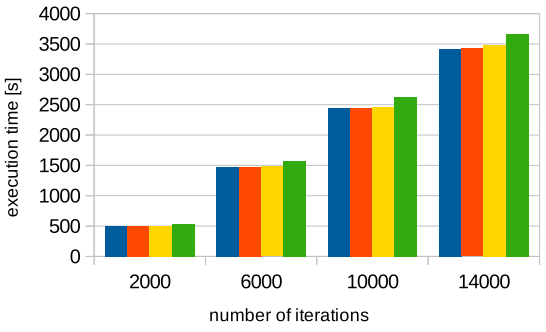
Figure 5c includes comparison of the basic NVRAM cache with three different mode configurations: single node consistency mode, on-sync checkpoint, and redundant log combined with infrequent checkpoint creation (it was required to use the on-sync checkpoint to periodically clear the journal, as discussed in Section 3). As expected, the overhead of the fail-safe mode limited to ensuring consistency within a single node is not visible due to no network communication. Creation of a redundant log on an additional node introduces negligible overhead. The most time-expensive configuration is the on-sync checkpoint because of communication with PFS, which is often a bottleneck.



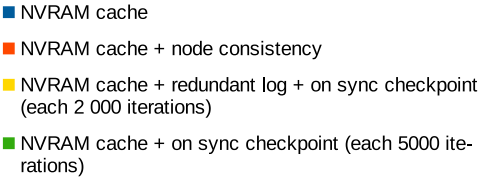
(a) Overhead related to checkpoint frequency



(b) Overhead related to number of redundant copies



(c) Comparison of fail-safe levels

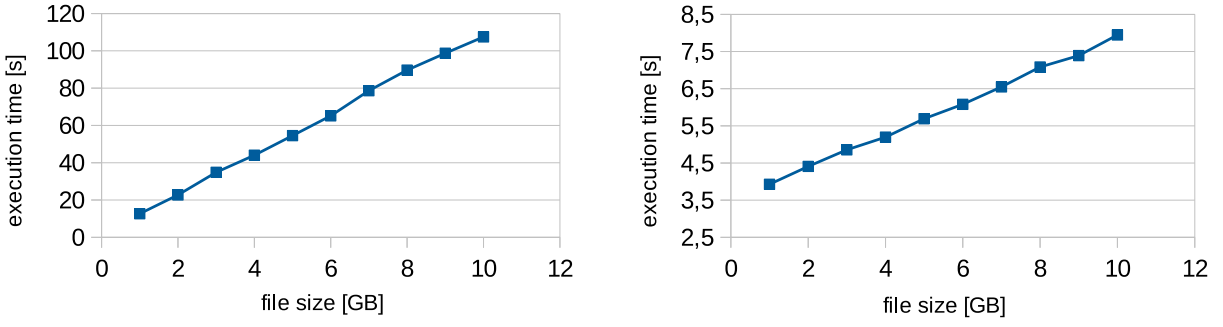


(d) Legend

Fig. 5: Different configurations of the solution for random walk microbenchmark

4.3. On sync checkpoint overhead

Figure 6 presents a linear growth of checkpoint creation (a) and cache restoring from a checkpoint (b). We can see that restoring is much faster than preparing a checkpoint – cache restoring is basically a sequence of cheap read requests, while creating a checkpoint is based on writing the whole file onto the PFS. We can also observe that with a proper configuration of a PFS, execution times are limited only by a hardware configuration. In our example we were able to restore a file at the speed of 1GB/s which is a doubled bandwidth of SSDs installed within a PFS server (and we used two servers). The same situation occurs for checkpoint creation.



(a) On sync checkpoint creation

(b) Cache restore from a checkpoint

Fig. 6: Execution time of on sync checkpoint creation and cache restore

5. Summary and future work

Within this paper we proposed an extension to the previously proposed MPI I/O NVRAM cache that focuses on safety of files. The three presented techniques benefit mainly from expected size of NVRAM, its persistence and byte addressing. If the application stores its complete state within a file, the solution could provide data safety with no other tools. Otherwise, it may be used complementarily to other checkpointing techniques. Experimental results demonstrated acceptable overhead of the solution.

The project of MPI I/O distributed NVRAM cache has a potential for further optimizations in its performance, for instance with collective operations. Another research direction is verification of the solution with more demanding applications in larger environments. We are expecting first NVRAM enabled devices soon which would allow tests on real hardware.

Acknowledgments

The research on NVRAM cache was supported by a grant from Intel Technology Poland. The research in the paper was partially done in collaboration with Intel Technology Poland and partially supported by statutory funds of Dept. of Computer Architecture, Faculty of ETI, Gdansk University of Technology, Poland.

References

[1] Casanova, H., Robert, Y., Vivien, F., Zaidouni, D., 2015. On the impact of process replication on executions of large-scale parallel applications with coordinated checkpointing. Future Generation Computer Systems 51, 7 – 19. URL: http://www.sciencedirect.com/science/article/pii/S0167739X15000953, doi:https://doi.org/10.1016/j.future.2015.04.003.

[2] Corporation, I., 2015. Pushing Storage Forward: Bigger, Faster and in 3-D. https://newsroom.intel.com/editorials/3d-nand-faster-storage/.

[3] Czarnul, P., Fraczak, M., 2005. New User-Guided and ckpt-Based Checkpointing Libraries for Parallel MPI Applications, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer Berlin Heidelberg. pp. 351–358.

[4] Di, S., Bouguerra, M.S., Bautista-Gomez, L., Cappello, F., 2014. Optimization of multi-level checkpoint model for large scale hpc applications, in: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 1181–1190. doi:10.1109/IPDPS.2014.122.

[5] Dong, X., Muralimanohar, N., Jouppi, N., Kaufmann, R., Xie, Y., 2009. Leveraging 3D PCRAM Technologies to Reduce Checkpoint Overhead for Future Exascale Systems, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, New York, NY, USA. pp. 57:1–57:12. URL: http://doi.acm.org/10.1145/1654059.1654117, doi:10.1145/1654059.1654117.

[6] Dongarra, J., Herault, T., Robert, Y., 2015. Fault-Tolerance Techniques for High-Performance Computing.

[7] Dorożynski, P., Czarnul, P., Malinowski, A., Czuryło, K., Dorau, Ł., Maciejewski, M., Skowron, P., 2016. Checkpointing of Parallel MPI Applications Using MPI One-sided API with Support for Byte-addressable Non-volatile RAM. Procedia Computer Science 80, 30 – 40. URL: http://www.sciencedirect.com/science/article/pii/S1877050916306457, doi:https://doi.org/10.1016/j.procs.2016.05.295. int. Conference on Computational Science, 6-8 June 2016, San Diego, USA.

[8] Egwutuoha, I.P., Levy, D., Selic, B., Chen, S., 2013. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. The Journal of Supercomputing 65, 1302–1326. URL: https://doi.org/10.1007/s11227-013-0884-0, doi:10.1007/s11227-013-0884-0.

[9] Fernando, P., Kannan, S., Gavrilovska, A., Schwan, K., 2016. Phoenix: Memory Speed HPC I/O with NVM, in: IEEE Int. Conference on High Performance Computing, pp. 121–131. doi:10.1109/HiPC.2016.023.

[10] Gropp, W., Lusk, E., 2002. Fault Tolerance in MPI Programs. Special Issue of the Journal High Performance Computing Applications 18 , 363–372.

[11] Hargrove, P., C Duell, J., 2006. Berkeley lab checkpoint/restart (blcr) for linux clusters 46, 494.

[12] Losada, N., Cores, I., Martín, M.J., González, P., 2017. Resilient MPI applications using an application-level checkpointing framework and ULFM. The Journal of Supercomputing 73, 100–113. URL: https://doi.org/10.1007/s11227-016-1629-7, doi:10.1007/s11227-016-1629-7.

[13] Malinowski, A., Czarnul, P., 2017. Distributed NVRAM Cache – Optimization and Evaluation with Power of Adjacency Matrix. Springer International Publishing, Cham. pp. 15–26. URL: https://doi.org/10.1007/978-3-319-59105-6_2, doi:10.1007/978-3-319-59105-6_2.

[14] Malinowski, A., Czarnul, P., 2018. A Solution to Image Processing with Parallel MPI I/O and Distributed NVRAM Cache. Scalable Computing: Practice and Experience 19. URL: https://doi.org/10.12694/scpe.v19i1.1389.

[15] Malinowski, A., Czarnul, P., Czuryło, K., Maciejewski, M., Skowron, P., 2017a. Multi-agent large-scale parallel crowd simulation. Procedia Computer Science 108, 917 – 926. URL: http://www.sciencedirect.com/science/article/pii/S1877050917305537, doi:https://doi.org/10.1016/j.procs.2017.05.036. international Conference on Computational Science, ICCS 2017, June 2017, Switzerland.

[16] Malinowski, A., Czarnul, P., Dorozynski, P., Czurylo, K., Dorau, L., Maciejewski, M., Skowron, P., 2016. A parallel MPI I/O solution supported by byte-addressable non-volatile RAM distributed cache, in: Position Papers of Federated Conference on Computer Science and Information Systems, Gdańsk, Poland, September 11-14, 2016., pp. 133–140. URL: https://doi.org/10.15439/2016F52, doi:10.15439/2016F52.

[17] Malinowski, A., Czarnul, P., Maciejewski, M., Skowron, P., 2017b. A Fail-Safe NVRAM Based Mechanism for Efficient Creation and Recovery of Data Copies in Parallel MPI Applications. Springer International Publishing, Cham. pp. 137–147. URL: https://doi.org/10.1007/978-3-319-46586-9_11, doi:10.1007/978-3-319-46586-9_11.

[18] Mills, B., Znati, T., Melhem, R., Ferreira, K.B., Grant, R.E., 2014. Energy consumption of resilience mechanisms in large scale systems, in: 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 528–535. doi:10.1109/PDP.2014.111.

[19] Narayanan, D., Hodson, O., 2012. Whole-system persistence with non-volatile memories, ACM. URL: https://www.microsoft.com/en-us/research/publication/whole-system-persistence-with-non-volatile-memories/.

[20] Plank, J.S., Beck, M., Kingsley, G., Li, K., 1995. Libckpt: Transparent checkpointing under unix, in: Proceedings of the USENIX 1995 Technical Conference Proceedings, USENIX Association, Berkeley, CA, USA. pp. 18–18. URL: http://dl.acm.org/citation.cfm?id=1267411.1267429.

[21] Rajachandrasekar, R., Moody, A., Mohror, K., Panda, D.K.D., 2013. A 1 pb/s file system to checkpoint three million mpi tasks, in: Proc. of International Symposium on High-performance Parallel and Distributed Computing, ACM, New York, USA. pp. 143–154. URL: http://doi.acm.org/10.1145/2462902.2462908, doi:10.1145/2462902.2462908.

[22] Raju, N., Gottumukkala, Y.L., Leangsuksun, C.B., Nassar, R., Scott, S., 2006. Reliability Analysis in HPC clusters, in: Proceedings of the High Availability and Performance Computing Workshop, pp. 673–684.

[23] Ren, J., Hu, Q., Khan, S., Moscibroda, T., 2017. Programming for non-volatile main memory is hard, in: Proceedings of the 8th Asia-Pacific Workshop on Systems, ACM, New York, NY, USA. pp. 13:1–13:8. URL: http://doi.acm.org/10.1145/3124680.3124729, doi:10.1145/3124680.3124729.

[24] Rudoff, A., 2017. Persistent Memory: The Value to HPC and the Challenges, in: Proc. of the Workshop on Memory Centric Programming for HPC, ACM, New York, USA. pp. 7–10. URL: http://doi.acm.org/10.1145/3145617.3158213, doi:10.1145/3145617.3158213.

[25] Walters, J.P., Chaudhary, V., 2006. Application-level checkpointing techniques for parallel programs, in: Distributed Computing and Internet Technology, Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 221–234.

[26] Yang, M., Hua, G., Feng, Y., Gong, J., 2017. Fault-Tolerance Techniques for Spacecraft Control Computers.