Procedia Computer Science
Volume 80, 2016, Pages 30–40
ICCS 2016. The International Conference on Computational Science

Checkpointing of Parallel MPI Applications using MPI One-sided API with Support for Byte-addressable Non-volatile RAM

Piotr Dorożyński[1], Paweł Czarnul[1], Artur Malinowski[1], Krzysztof Czuryło[2], Łukasz Dorau[2], Maciej Maciejewski[2], and Paweł Skowron[2]

[1] Faculty of Electronics, Telecommunications and Informatics
Gdansk University of Technology, Poland
piotr.dorozynski@pg.gda.pl, pczarnul@eti.pg.gda.pl,
artur.malinowski@pg.gda.pl

[2] Intel Technology Poland Sp. z o.o., Gdansk, Poland
krzysztof.czurylo@intel.com, lukasz.dorau@intel.com,
maciej.maciejewski@intel.com, pawel.skowron@intel.com

Abstract
The increasing size of computational clusters results in an increasing probability of failures, which in turn requires application checkpointing in order to survive those failures. Traditional checkpointing requires data to be copied from application memory into persistent storage medium, which increases application execution time as it is usually done in a separate step. In this paper we propose to use emerging byte-addressable non-volatile RAM (NVRAM) as a persistent storage medium and we analyze various methods of making consistent checkpoints with support of MPI one-sided API in order to minimize checkpointing overhead. We test our solution on two applications: HPCCG benchmark and PageRank algorithm. Our experiments showed that NVRAM based checkpointing performs much better than traditional disk based approach. We also simulated different possible latencies and bandwidth of future NVRAM and our experiments showed that only bandwidth had visible impact onto application execution time.

Keywords: NVRAM, parallel MPI one-sided extension, checkpointing of parallel applications, performance optimization

1 Introduction

In the recent years, performance growth of computational clusters has been possible mainly due to considerable increases in the numbers of cores in computational devices – both multicore CPUs as well as engagement of accelerators such as GPUs and coprocessors such as Intel® Xeon Phi™. The current Intel® Xeon® Processor E7-8890 v3 features 18 cores (36 threads) at TDP 165 W, Intel® Xeon Phi™

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016

doi:10.1016/j.procs.2016.05.295

Coprocessor 7120A (16GB, 1,238 GHz) features 61 cores (244 threads) at TDP 300 W, Tesla K80 (2x Kepler GK210) features 4992 CUDA cores.

Such increase in complexity of a system may result in a potentially higher percentage of failures. As an example, numbers such as 1.25 failures per day were reported for the Sequoia cluster [6]. Consequently, there is a need for programming solutions that would enable surviving failures. Checkpointing of parallel applications has been widely studied so far in e.g. [3, 8] with transparent, coordinated checkpointing/restart possible in the widely popular OpenMPI [11, 12, 13]. Some new developments in non-volatile memories have driven us to propose a new solution in this regard. Specifically, within this paper, we propose a solution for MPI applications that incorporates MPI one-sided API and a collection of persistent memories in cluster nodes for efficient checkpointing.

2 Related Work

In this paper we propose a new MPI one-sided based persistent memory enabled application for emerging memory technologies. These technologies have the following common features: byte-addressability, random access, non-volatility and limited endurance. Kryder and Kim [14] reviewed thirteen non-volatile memory technologies most of which have the aforementioned features and thus they are of our interest. The performance of various non-volatile memory technologies compared to DRAM and NAND flash is presented in Table 1.

Tablica 1: Performance of memory technologies.

Technology	Read latency	Write latency
DRAM [14]	6-10 ns	6-10 ns
FRAM [14, 19]	8-75 ns	8-75 ns
MRAM [21]	1-10 ns	1-10 ns
STT-RAM [21]	1-10 ns	1-10 ns
NRAM [18]	<10 ns	<10 ns
RRAM [14]	10 ns	20 ns
CBRAM [10]	<50 ns	<50 ns
PRAM [4]	10-100 ns	100-1000 ns
NAND flash [14]	25,000 ns	200,000 ns

The idea of using NVRAM in high performance computing was already investigated, especially in the area of data-intensive architectures, where usage of only DRAM is costly and power-intensive [20]. One of domains that require high memory capacities and good memory performance is graph processing. In this case the emerging NVRAM technologies allow to store much larger graphs with not so big performance loss [17].

NVRAM was also already used for checkpointing of distributed parallel applications. Dong et al. [5] proposed to use hybrid local and global checkpointing using phase-change memories (PCM). Narayanan and Hodson [15] proposed to use NVRAM to make whole-system checkpoints by keeping all data in NVRAM. This approach is similar to our proposed double buffering scheme as the data is already in place when restarting. However, they require hardware modification in order to make sure that data is flushed to persistence on failure. Gao et al. [7] created their own checkpointing system that creates partial checkpoints during application execution. It utilizes runtime idle periods to copy data from DRAM to NVRAM in order not to interfere with application execution.

3 Motivations

Driven by the aforementioned developments in both:

1. growth of HPC systems through the considerable increase of the number of compute devices and cores and consequently a potentially high rate failures of such systems during application execution,

2. non-volatile memory technologies including features such as byte level access, relatively high performance and large sizes compared to RAM

we decided to incorporate non-volatile RAM into wrappers over MPI one-sided API, in order to provide persistence of data stored in MPI windows and consequently apply this solution for continuous checkpointing of an application at the code level. Specifically, checkpointing is realized by provision of transactions in persistent memory. When synchronization is enforced the current state can be considered as a consistent state of the application.

4 Proposed Solution

In this paper we present an MPI one-sided communication based checkpointing in byte-addressable non-volatile RAM. One-sided communication functions enable to specify regions of memory (called windows) of one process to be available for remote read and write by other processes of an MPI application and thus create abstraction of distributed shared memory. The proposed solution extends these functions in order to provide an easy application level checkpointing when communicating processes synchronize. We create local checkpoints i.e. every process saves only data it is responsible for and thus requires all machines to be up and running when restarting after failure.

In our solution we implemented wrappers for MPI one-sided communication functions in order to extend their functionality for transactional access to underlying memory areas. We analyzed MPI one-sided communication API and found out that the only moments when we can be certain that processes finished communication and thus hold a consistent application state are the moments of synchronization. Consequently we decided that synchronization calls should commit a previously started transaction and start a new one, but we allowed the programmer to decide which synchronization calls should do so.

Provided transactional access creates a new programming model by allowing processes to communicate freely using standard one-sided communication functions and fall back to a state saved during synchronization.

4.1 Data consistency

In order to keep data consistent and be able to restart an application after a failure we analyzed three possible methods of keeping consistent data at all times:

- transaction logging — is a method used in databases for making transactional changes. It keeps a log of all changes applied to data, so that these may be reverted when transaction is rolled back;

- checkpointing — is a classical method used in high performance computing for storing an application state and recovery after a failure. It works by copying necessary data into a separate location (actually the data is saved into two locations alternately in order to have at least one proper checkpoint even if a failure occurs during checkpoint creation) at predefined moments in time;

- double-buffering — is a modified version of checkpointing in which the application itself uses two data buffers: one used as a source (i.e. all read accesses will go from this buffer) and the second used as a destination (i.e. all write accesses will go into this buffer). After every iteration semantics of the buffers is swapped i.e. the previous source buffer becomes a new destination buffer and the previous destination buffer becomes a new source buffer for the next iteration. Since the source buffer is not modified during a single iteration, if anything fails we can restart application by running the same iteration once again. Now it is only needed to ensure that the data in the destination buffer is flushed to persistence before it becomes a source for the next iteration.

The first method requires to keep an entry for every modification of the data. If we modify the whole memory area, which is a common case in high performance computing, we will have to keep in a log a full copy of the original data along with additional metadata. The logging operation will also add an additional overhead for storing every modification in a transaction log. These issues make this method of little use in many high performance applications and consequently we did not focus on it in this work.

The second method is very flexible as it does not impose any limitation on the algorithms that may use it for storing an application state. This method also has much less memory overhead than transaction logging as it requires much less metadata to be saved.

The last method has almost no performance overhead, and thus it may achieve much better results than the other two, as data is only written once to one memory (as opposed to the checkpointing approach that requires to save data in the destination area and then copy the same data to a separate location) and the only overhead comes from flushing data to persistence. However, this method requires an algorithm to overwrite whole destination area in every iteration and thus it is less flexible than classical checkpointing. This method also requires fast byte-level access to persistent memory both for reading and writing and thus it was difficult to apply efficiently in traditional persistent memories technologies.

We implemented two of the above methods (ommiting transaction logging due to its aforementioned drawbacks): the checkpointing method as a flexible solution for the MPI one-sided API wrappers we implemented and double-buffering at the application level in our test applications in order to see what performance we could obtain. Sources of the solution will soon be available on GitHub.[1]

5 Experiments

5.1 Testbed Environment

The testbed environment consisted of a cluster with 8 identical nodes. One of the nodes was used as a front-end of the cluster, and so it did not take part in computations, and the other nodes were prepared for simulation of NVRAM.

Every node had 2 Intel® Xeon® CPU E5-4620 v2 @ 2.60GHz processors each with 8 cores (they were modified to simulate latencies of possible NVRAM technologies and did not use Hyper-Threading) giving the total of 16 cores with 2.6 GHz clock per node. Every node also had 32 GB of RAM, from which 17.2 GB was used for simulating NVRAM and the rest was used as normal RAM. Additionally, every node storage included a 240 GB Intel DC 3500 series SSD and a Seagate Barracuda 500 GB 7200 RPM 16MB cache disks.

The nodes were interconnected with 40Gb/s InfiniBand and were running Rocks 6.1.1 cluster distribution[2] (based on CentOS release 6.5). The applications were compiled using GCC 4.4.7 with OpenMP enabled. The MPI implementation used was MVAPICH2[3] 2.1 for its InfiniBand support.

[1] `https://github.com/pmem/mpi-pmem-ext`
[2] `http://www.rocksclusters.org/`
[3] `http://mvapich.cse.ohio-state.edu/`

The NVRAM simulation was done using The Persistent Memory Driver with ext4 Direct Access (DAX)[4], which is an extension to the Linux kernel. This extension allows to create a virtual disk in reserved memory (i.e. part of RAM that is marked as persistent by Linux kernel when it starts) and create a modified ext4 file system on top of it. The modification of ext4 creates direct mapping between file system block numbers and virtual memory addresses, so that the memory is directly accessed bypassing the page cache.

We simulated various possible latencies of emerging NVRAM technologies using emulation platform that is realized in hardware. The emulation platform had the following parameters:

- memory latency — supplementary latency over DRAM of every access to NVRAM,
- commit latency — latency of operation that flushes data in processor caches into NVRAM and makes sure it is stored persistently,
- bandwidth.

5.2 Results

We implemented two applications in order to test our solution: HPCCG [9] and PageRank [16]. The former is described by its authors as "best approximation to an unstructured implicit finite element or finite volume application in 800 lines or fewer" and it was chosen as it is a common type of problem in physical simulations. The latter is the algorithm that stands behind the Google search engine. The algorithm assigns rank values to web pages, organized as a graph, depending on their importance based on pages referencing them. This application was chosen, because of its natural requirement for double buffering (updating the rank of a node in a graph requires knowledge of ranks from a previous iteration of many neighbors).

5.2.1 HPCCG

The HPCCG mini-application measures system performance by solving a system of linear equations $\mathbf{Ax} = \mathbf{b}$ using the conjugate gradient method. The system solved by HPCCG is a finite difference matrix with a 27-point stencil. The size of the matrix is defined by three values nx, ny and nz that define the size of a grid of measurement points that is assigned to every process. Grids assigned to processes are then stacked on the OZ axis so that the final size of the grid for the whole domain is nx by ny by $proc\_count * nz$. In our experiments we set parameters nx, ny and nz to 256, 256 and 512 respectively. This size of domain resulted in the size of the data kept in memory equal to about 80% of available RAM. The size of each checkpoint file for this domain size is about 769 MB. In every test the application was run for 100 iterations.

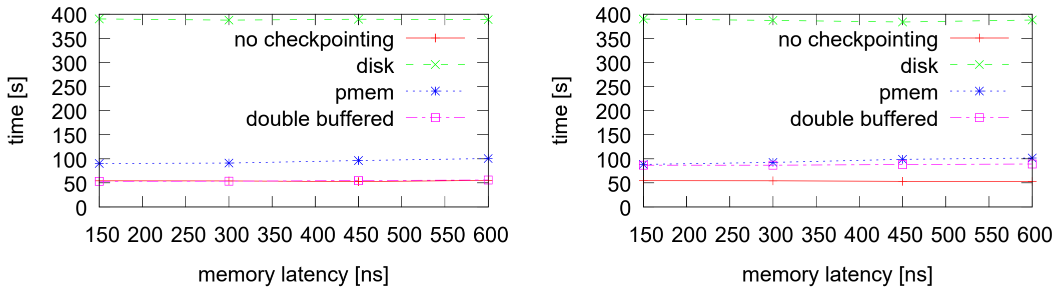
We implemented four versions of this application:

- no checkpointing — the original implementation modified to use MPI one-sided API for communication so it may serve as base for the next implementations with checkpointing,
- disk — modification of the first implementation with typical application level checkpointing that uses standard C file I/O to save the data and saves it on the SSD disk,
- pmem — modification of the first implementation with checkpointing done using implemented wrappers over MPI one-sided functions,

[4] https://www.kernel.org/doc/Documentation/filesystems/dax.txt

- double buffered — implementation of double-buffered checkpointing at the application level as described in Section 4.1.

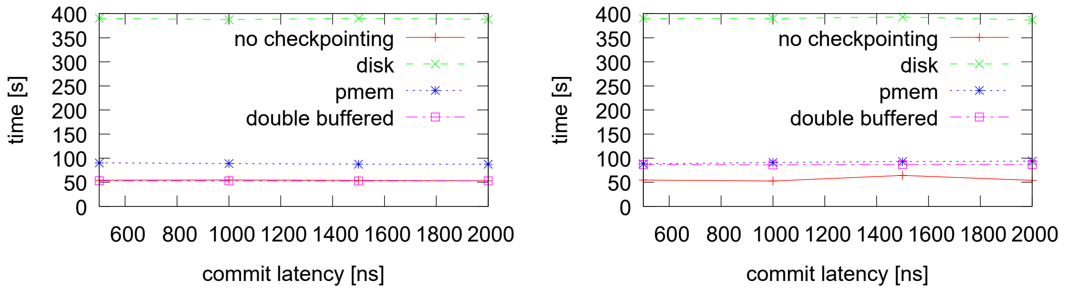
Figure 1 presents execution times over memory latency for different implementations and two values of memory bandwidth. First of all we can see that all NVRAM based methods achieve much better results than the standard checkpointing on disk even though we used SSD. Secondly, we can see that double buffered implementation is always faster than the other ones with checkpointing. We can also see that latency of the memory had almost no impact on final performance, however we can see a small linear growth. The memory bandwidth on the other hand has significant impact on performance as it increased the execution time of double buffered solution from around 55 seconds to around 90 seconds. What is really interesting is that for the pmem version the decreased memory bandwidth did not increase execution time, even though it makes block access to the memory and thus we expected that decreased bandwidth would increase execution time.



(a) Memory bandwidth = 37GB/s (b) Memory bandwidth = 9.5GB/s

Rysunek 1: Chart of HPCCG execution time over memory latency for commit latency = 500 ns

Figure 2 presents execution time versus commit latency for two values of memory bandwidth. As can be seen in the figure the chart is flat i.e. execution time does not depend on commit latency. The commit operation is issued only when data is flushed to persistence and such flush occurs only 3 times (for double buffered) and 6 times (for pmem) per iteration. The largest used commit latency is 2000 ns = 2 μs, iteration count is 100, so the sum of commit latency in the worst case scenario is $2 * 100 * 6\mu s = 1200\mu s = 1.2ms$, which is unnoticeable when compared to the application execution time.



(a) Memory bandwidth = 37GB/s (b) Memory bandwidth = 9.5GB/s

Rysunek 2: Chart of HPCCG execution time over commit latency for memory latency = 150 ns

Table 2 presents start and restart times (measured in seconds) of all implementations for the worst tested NVRAM performance i.e. commit latency equal to 2000 ns, memory latency equal to 600 ns and memory bandwidth equal to 9.5 GB/s. The restart times are shorter than start times as start times also include some conjugate gradient operations that are done before the algorithm main loop. As can be seen in Table 2 the double buffered implementation has the best performance. The pmem implementation has worse restart performance than the disk implementation even though it works in much faster memory, but it also has a worse start time, so the reason for this may lay elsewhere e.g. in memory allocation.

Tablica 2: HPCCG start and restart times.

	no checkpointing	disk	pmem	double buffered
start	1.14	1.20	1.80	1.20
restart	—	0.58	1.53	0.13

5.2.2 PageRank

The PageRank algorithm is used for ranking of nodes in a directed graph dependent on the rank of their neighbors. It is mostly used for ranking web pages dependent on the pages that link them. The definition of PageRank is as follows [16]:

$$R'(u) = c \sum_{v \in B_u} \frac{R'(v)}{N_v} + cE(u) \tag{1}$$

where $R'(u)$ is PageRank of node u, c is a constant used to keep $||R'||_1 = 1$, B_u are neighbors of node u that are connected to u by an edge ending in u, N_v is the outdegree of vertex v and $E(u)$ is some vector that corresponds to a source of rank.

The actual algorithm for calculating PageRank [2] works iteratively by updating every node's rank dependent on the values of the neighbors' ranks from the previous iteration. This imposes the need for double buffering data, so we implemented three variants of this application:

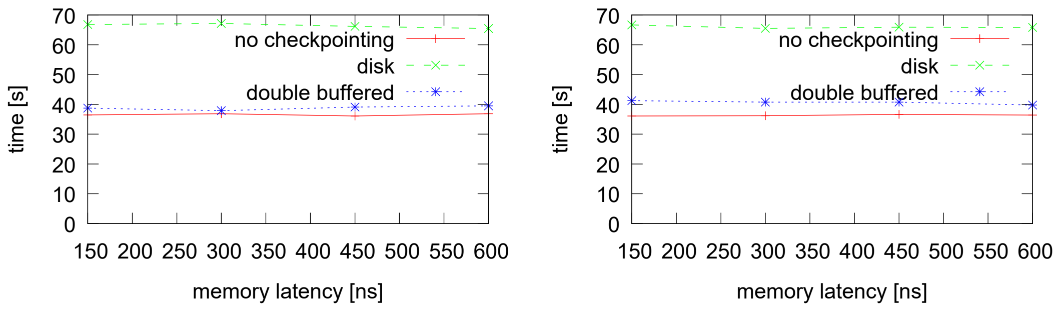
- no checkpointing — a basic implementation without checkpointing,
- disk — modification of the first implementation with typical application level checkpointing that uses standard C file I/O to save data and it saves data on disk (in our case SSD),
- double buffered — implementation of double-buffered checkpointing at the application level as described in Section 4.1.

We ran our experiments on the actual web graph obtained from a 2005 crawl of the .sk domain by UbiCrawler [1]. This graph has 50 636 154 nodes and 1 949 412 601 arcs and it was chosen, because it is the biggest one of the available web graphs in the Laboratory for Web Algorithmics at the University of Milan datasets[5] that will fit into the available memory of our cluster. For this graph the size of the data kept in memory of every process equals to about 10 - 13% of available RAM (depending on process). The size of each checkpoint file for this size of graph was about 56 MB. The application was run for 100 iterations.

Figure 3 presents execution times over memory latency for different implementations and two values of memory bandwidth. The results are similar to the results for HPCCG. The double buffered implementation achieves results comparable to the implementation without checkpointing and much better

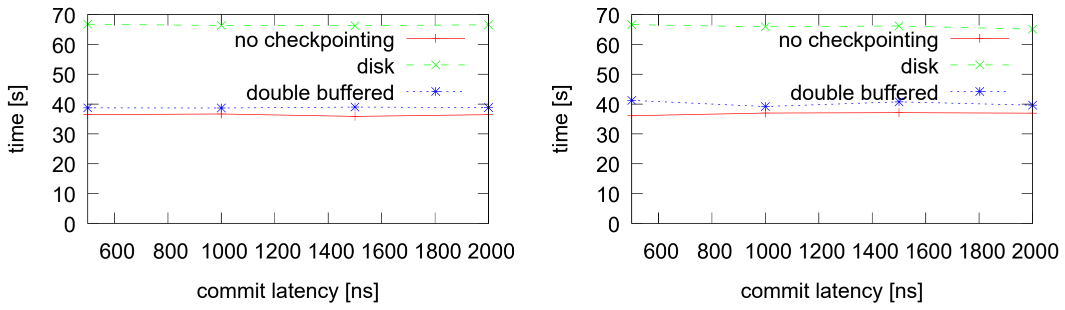
[5]http://law.di.unimi.it/datasets.php

than the disk implementation. The memory latency once again has little or no impact onto final performance and memory bandwidth increased execution time of the double buffered implementation, but the difference is much smaller than for HPCCG. Figure 4 presents execution time versus commit latency for two values of memory bandwidth and once again it shows that commit latency has no impact onto execution time.



(a) Memory bandwidth = 37GB/s (b) Memory bandwidth = 9.5GB/s

Rysunek 3: Chart of PageRank execution time over memory latency for commit latency = 500 ns



(a) Memory bandwidth = 37GB/s (b) Memory bandwidth = 9.5GB/s

Rysunek 4: Chart of PageRank execution time over commit latency for memory latency = 150 ns

Table 3 presents start and restart times (measured in seconds) of all implementations for the worst tested NVRAM performance i.e. commit latency equal to 2000 ns, memory latency equal to 600 ns and memory bandwidth equal to 9.5 GB/s. As can be seen the start and restart times are comparable in all implementations.

Tablica 3: PageRank start and restart times.

	no checkpointing	disk	double buffered
start	0.10	0.12	0.12
restart	—	0.11	0.11

6 Summary and Future Work

In this paper, we presented a new field of application for emerging NVRAM technologies. We showed the main features of these memories and presented how they may be used for checkpointing distributed applications. Two methods of checkpointing were analyzed: classical method with copying data onto persistent media and double buffered approach in which data is kept directly on persistent media. We implemented our solution as an extension to MPI one-sided API using NVRAM and tested it on two different applications: HPCCG benchmark and PageRank algorithm. The results showed that the double buffered approach makes it possible to checkpoint applications with little or no checkpointing overhead.

For future work we consider integrating the double buffered approach into MPI one-sided extensions we implemented as it gave the best results. Both of our test applications represent one type of distributed applications i.e. Single Program Multiple Data (SPMD) applications, so we are now investigating how the proposed solution may be applied to other application paradigms e.g. master-slave or divide-and-conquer applications.

Acknowledgments

The research in the paper was supported by a Grant from Intel Technology Poland.

Literatura

[1] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[2] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107 – 117, 1998. ISSN 0169-7552. doi: http://dx.doi.org/10.1016/S0169-7552(98)00110-X. URL http://www.sciencedirect.com/science/article/pii/S016975529800110X. Proceedings of the Seventh International World Wide Web Conference.

[3] Paweł Czarnul and Marcin Frączak. New user-guided and ckpt-based checkpointing libraries for parallel mpi applications,. In Beniamino Di Martino, Dieter Kranzlmüller, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3666 of *Lecture Notes in Computer Science*, pages 351–358. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-29009-4. doi: 10.1007/11557265_46. URL http://dx.doi.org/10.1007/11557265_46.

[4] Xiangyu Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Yuan Xie. Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems. In *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, pages 1–12, Nov 2009. doi: 10.1145/1654059.1654117.

[5] Xiangyu Dong, Naveen Muralimanohar, Norm Jouppi, Richard Kaufmann, and Yuan Xie. 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*, SC '09, pages 57:1–57:12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-744-8. doi: 10.1145/1654059.1654117. URL http://doi.acm.org/10.1145/1654059.1654117.

[6] Jack Dongarra. Emerging heterogeneous technologies for high performance computing, May 2013. Heterogeneity in Computing Workshop, http://www.netlib.org/utk/people/JackDongarra/SLIDES/hcw-0513.pdf.

[7] Shen Gao, Bingsheng He, and Jianliang Xu. Real-time in-memory checkpointing for future hybrid memory systems. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 263–272, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3559-1. doi: 10.1145/2751205.2751212. URL http://doi.acm.org/10.1145/2751205.2751212.

[8] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physics: Conference Series*, 46(1):494, 2006. URL http://stacks.iop.org/1742-6596/46/i=1/a=067.

[9] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, September 2009.

[10] H. Honigschmid, M. Angerbauer, S. Dietrich, M. Dimitrova, D. Gogl, C. Liaw, M. Markert, R. Symanczyk, L. Altimime, S. Bournat, and G. Muller. A Non-Volatile 2Mbit CBRAM Memory Core Featuring Advanced Read and Program Control. In *VLSI Circuits, 2006. Digest of Technical Papers. 2006 Symposium on*, pages 110–111, 2006. doi: 10.1109/VLSIC.2006.1705334.

[11] Joshua Hursey, Jeffrey M. Squyres, and Andrew Lumsdaine. A checkpoint and restart service specification for open mpi. Technical Report TR635, Indiana University, Bloomington, Indiana, USA, July 2006. URL http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR635.

[12] Joshua Hursey, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 03 2007.

[13] Joshua Hursey, Timothy I. Mattox, and Andrew Lumsdaine. Interconnect agnostic checkpoint/restart in open mpi. In *HPDC '09: Proceedings of the 18th ACM international symposium on High Performance Distributed Computing*, pages 49–58, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-587-1. doi: http://doi.acm.org/10.1145/1551609.1551619.

[14] M.H. Kryder and Chang Soo Kim. After Hard Drives — What Comes Next? *Magnetics, IEEE Transactions on*, 45(10):3406–3413, Oct 2009. ISSN 0018-9464. doi: 10.1109/TMAG.2009.2024163.

[15] Dushyanth Narayanan and Orion Hodson. Whole-system persistence with non-volatile memories. In *Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*. ACM, March 2012. URL http://research.microsoft.com/apps/pubs/default.aspx?id=160853.

[16] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, November 1999. URL http://ilpubs.stanford.edu:8090/422/. Previous number = SIDL-WP-1999-0120.

[17] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Scaling techniques for massive scale-free graphs in distributed (external) memory. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 825–836, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4971-2. doi: 10.1109/IPDPS.2013.72. URL http://dx.doi.org/10.1109/IPDPS.2013.72.

[18] R.F. Smith, T. Rueckes, S. Konsek, J.W. Ward, D.K. Brock, and B.M. Segal. Carbon nanotube based memory development and testing. In *Aerospace Conference, 2007 IEEE*, pages 1–5, March 2007. doi: 10.1109/AERO.2007.353104.

[19] D. Takashima, Yasushi Nagadomi, and Tohru Ozaki. A 100 MHz Ladder FeRAM Design With Capacitance-Coupled-Bitline (CCB) Cell. *Solid-State Circuits, IEEE Journal of*, 46(3):681–689, March 2011. ISSN 0018-9200. doi: 10.1109/JSSC.2010.2098210.

[20] B. Van Essen, R. Pearce, S. Ames, and M. Gokhale. On the Role of NVRAM in Data-intensive Architectures: An Evaluation. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 703–714, May 2012. doi: 10.1109/IPDPS.2012.69.

[21] K L Wang, J G Alzate, and P Khalili Amiri. Low-power non-volatile spintronic memory: STT-RAM and beyond. *Journal of Physics D: Applied Physics*, 46(7):074003, 2013. URL http://stacks.iop.org/0022-3727/46/i=7/a=074003.