Temporality a NVRAM-based virtualization platform

Vasily A. Sartakov
TU Braunschweig
sartakov@ibr.cs.tu-bs.de

Arthur Martens
TU Braunschweig
martens@ibr.cs.tu-bs.de

Rüdiger Kapitza
TU Braunschweig
rrkapitz@ibr.cs.tu-bs.de

***Keywords*-NVRAM; Virtualization; Cloud Computing;**

***Abstract*—Power failures in data centers and Cloud Computing infrastructures can cause loss of data and impact revenue. Existing best practice such as persistent logging and checkpointing add overhead during operation and increase recovery time. Other solutions like the use of an uninterruptable power supply incur additional costs and are maintenance-intensive.**

Novel persistent main memory, i.e. memory that retains stored data without an external source of power, firstly prevents data loss in case of a power outage, secondly reduces the time for a system reboot and thirdly enables to continue operation at full-speed after a recovery. Yet new architectures and programming models are required to utilize persistent main memory.

We present *Temporality* a virtualization layer that runs virtual machines in persistent memory and offers virtual persistent memory. It can be used as a basis for future Cloud platforms to allow applications the utilization of persistent memory without any changes. It provides safety of volatile data, significantly decreases overall recovery time and prevents subsequent performance degradation.

I. Introduction

In recent years the number of Internet services hosted by Cloud platforms has been steadily growing and this trend still continues. Due to economic reasons and ease of use many consumer applications like Netflix, Spotify or Dropbox are backed by Cloud infrastructures. Therefore, it is not surprising that outages of Cloud infrastructure can cause a potentially high loss of revenue.

A recent study [1] assessed the most common failures (55 %) in data centers are related to power outages. Power failures often cause long downtimes: according to the study an average duration of a partial outage is 56 minutes, while a complete unplanned outage takes 119 minutes. The average cost of an unplanned power outage per incident is almost $8,000 per minute. Main factors are indirect costs for example due to service level agreement misses and the loss of runtime data (i.e. state that is not secured on persistent storage).

To mitigate the latter, techniques like logging and checkpointing [2] are applied. However these mechanisms require additional resources and delay recovery. Protective technologies, which specifically address power outages, such as uninterruptable power supply (UPS) can prevent the loss of runtime data, by extending the time until the inevitable shutdown. This can be used to finalize running processes but once power returns it requires a time-consuming restart. In some cases an UPS enables to bridge the power gap. Nevertheless, UPSs are expensive, maintenance-intensive and often fragile [3].

Despite numerous measures power outages in data centers occur and existing best practice to handle their effects have serious drawbacks. Therefore, we investigated alternative ways to handle power outages and how to decrease the downtime after such events. Novel non-volatile random-access memory (NV-RAM) can retain in-memory data without an external source of power. The benefit of a system equipped with NV-RAM lies in the ability to store all runtime data of an application inside persistent memory without a significant decrease in performance. Hence after system restart all runtime data is still available or can be recovered swiftly without the need of time-consuming data retrieval from background storage or the rerun of requests.

Despite the benefits, NV-RAM is a technology which is still under development and has sparse system support. However, adapting virtualization technology to NV-RAM enables off-the-shelf software appliances to utilize it for fast power outage recovery without any changes to the appliance. This creates for cloud providers the opportunity to offer persistence as a value-added service for Infrastructure-as-a-Service clouds.

In this paper, we present *Temporality* a virtualization layer that adopts NV-RAM to enable *persistent virtual machines (pVMs)*. A pVM is an persistent entity without any volatile state and the use of NV-RAM is transparent to the hosted virtual appliances. Temporality extends our preliminary design [4] by introducing *virtual persistent memory (VPM)*. This allows to run multiple pVMs in parallel, which together or individually exceed the physical NV-RAM capacity, by on-demand outsourcing of persistent pages to background storage. Thereby, for applications with a linear access pattern Temporality can virtually increase the amount of persistent memory by 40% at a performance penalty of 10%.

The remainder of the paper first introduces the architecture of Temporality. Next, our implementation of the VPM is explained in detail together with the support for multiple virtual machines. Section V is devoted to evaluation. Finally, Section VI describes related work, followed by a conclusion.

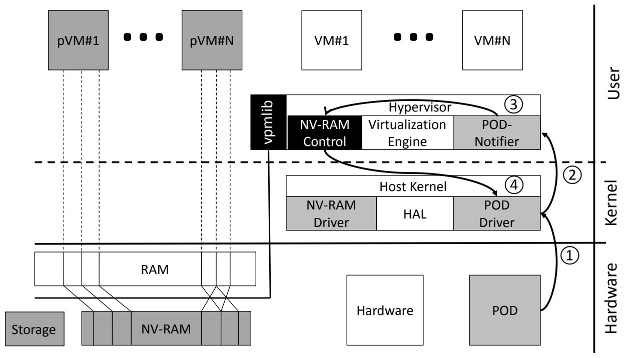


Figure 1. Architecture of the NV-Hypervisor with Temporality extensions indicated in black

II. Architecture

Temporality is an extension of our preliminary work [4] on NV-Hypervisor. There, we presented an architecture for a hypervisor that is capable of running persistent virtual machines (VMs). In this section we give a brief overview of the NV-Hypervisor architecture which forms the basis for Temporality.

A. Persistence Model

The minimum persistent executable unit in Temporality is a VM, which can be subdivided into the following components: a RAM image of the guest system, a CPU state and virtual device states. The RAM image is a flat chunk of host memory and is used as a RAM emulation for the guest system. It holds all runtime data of the VM. The CPU state is a structure that describes the current state of the guest systems virtual CPU. Finally, the virtual devices are described as independent stateful modules that communicate via the virtual bus. The state of the virtual devices and CPU can be combined as a *virtual environment*.

Persistence in Temporality is data-centric and is provided by the NV-Hypervisor. Data-centric specifically means we do not continue execution of the hypervisor process in case of a recovery. Instead we restart the hypervisor and reconnect the pVMs with the RAM image and the *virtual environment* that is stored persistent in NV-RAM. The host system with its drivers and the NV-Hypervisor itself stay volatile and shall be restarted after every failure. In summary, we have volatile devices in the kernel connected to non-volatile devices in the pVM and both need to be coherent, even at time of recovery.

B. Components

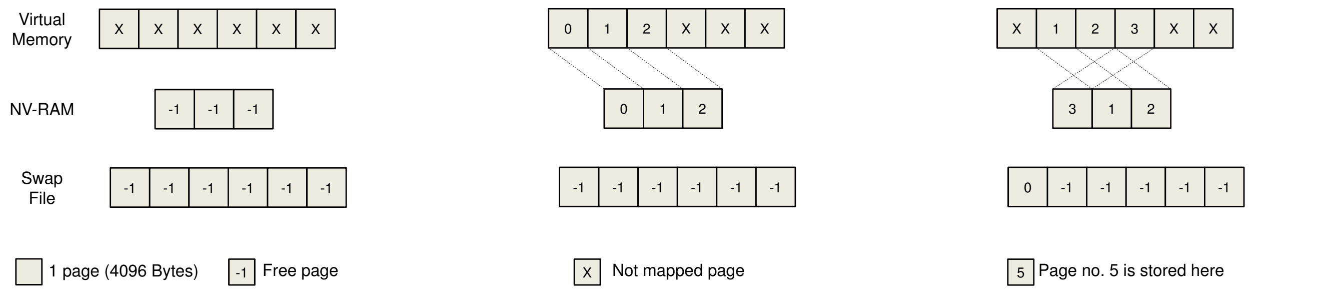
As Figure 1 depicts, Temporality follows a multilayer architecture involving hardware and software components. All components provided by NV-Hypervisor are colored in gray while Temporalitys modifications are shown in black.

1) Hardware: Our hardware consists of a commodity server platform with a general-purpose CPU, DRAM main memory and hard drives. Additionally NV-RAM and a power outage detector (POD) are included in this system. NV-RAM and DRAM are used simultaneously, sharing the same memory controller. We assume that the CPU and peripherals will utilize current volatile technologies in the near future. Accordingly the state of these components will be lost in case of a power failure. The *virtual environment* of a pVM is located in the cache and registers of the CPU during active operation and is therefore volatile. In order to continue operation after recovery the *virtual environment* needs to be saved before a power failure shuts down the machine. Detection of a power failure is the purpose of the POD in our system. A POD continuously monitors the input voltage and immediately sends a signal to the system when it drops below a defined threshold. On detection of a power failure residual energy, i.e. all the energy accumulated in the electrical circuits, is spent to store the *virtual environment* inside NV-RAM. In the remainder of the paper we will refer to this process as *fixation*.

2) Software: The software components of our system are a general-purpose operating system with a POD-driver and a POD-notifier. On top of this software stack we run our NV-Hypervisor who manages all VMs, including initialization, execution, stopping and recovery. Access and management to the NV-RAM is provided through the NV-RAM Control and a library called *vpmlib*. The operating system kernel performs only usual functions related to the management and sharing of standard system resources. In case of a power outage all components work together to reliably save the pVMs as presented in Figure 1. Any interrupt from the POD is processed by the POD-driver ① and forwarded to the POD-Notifier located in the NV-Hypervisor ② that stops all running VMs. Afterwards, it initiates the *fixation* and notifies the kernel about the end of the saving process ③. Finally, the kernel clears all caches and stops memory operations ④.

III. A library for virtual persistent memory

Our initial prototype [4] did not involve any specific memory management service for NV-RAM. Free memory was simply indicated by a pointer and whenever a memory region was allocated in NV-RAM this pointer was moved to a subsequent free region. When pVMs are restored all virtual memory regions need to be reconnected to the exact same locations as assigned before the shutdown. However modern systems operate in symmetric multiprocessing mode which introduces nondeterminism during normal operation and recovery. In consequence the order of allocations is not deterministic which leads to inconsistencies.



(a) Initial state (b) Full NV-RAM (c) Swapped out page

Figure 2. Simplified memory allocation example with VPM: six persistent pages shall be allocated but only three pages can be stored in NV-RAM at once

For Temporality we extended the NV-Hypervisor with capabilities to allocate and restore memory of multiple concurrently operating pVMs. Additionally, we added support for virtual persistent memory thereby enabling pVMs to utilize more than the physically available NV-RAM. All capabilities are provided by our *vpmlib* library that, for now, is placed in user space of the virtualization layer.

For the application developer, *vpmlib* provides a minimal API to allocate (*nvalloc*) and free (*nvfree*) memory regions of a given size. Internally this is achieved through the system calls *mmap* and *munmap* to map pages from the persistent memory located in */dev/mem* into the virtual memory of the running process. In between these levels *vpmlib* coordinates to persistent memory, provides name-based allocation, and implements the swapping mechanism. Furthermore, *vpmlib* stores its state completely in persistent memory to enable a recovery of allocated regions after restart.

According to best practice, we implemented a lazy page allocation for persistent pages and coupled it with swapping. The whole mechanism is explained by the allocation example presented in Figure 2. When a pVM is started, it initializes the *vpmlib* and registers a signal handler (*vpm_fault_handler*) for the page fault signal *SIGSEGV*. At the end of initialization the virtual memory and the swap file are in a state without any mappings (see Figure 2a).

In Figure 2 we want to allocate six consecutive persistent pages of virtual memory but the NV-RAM in this example can only store three pages at once. For the page allocation we call the *nvalloc*-function and create a flat region without any permissions in the virtual memory by means of the *mmap* system call. As a result *nvalloc* returns a pointer to a set of six continuously mapped pages in the virtual memory. At this point none of the allocated pages is mapped to the NV-RAM or to the swap-file (see Figure 2a). All following access to any of these mapped pages causes a page fault that is processed by the *vpm_fault_handler*. The *vpm_fault_handler* maps the first free page in NV-RAM to the accessed virtual memory page. We also have to add some meta data with information about the mapping to the persistent-page table (PPT) located in the header of the memory region (see Figure 3).

Eventually the *vpm_fault_handler* changes the mapping of the accessed page to read-write and the page may be accessed. Figure 2b shows the state of the virtual memory after three different pages have been accessed. Now the NV-RAM is fully occupied. When we access any of the other virtual pages, we again get a page fault because there is no free page left in the NV-RAM, which we can map directly. Hence we need to swap-out a page first. According to a simple first-in first-out strategy we look for the oldest allocated page and move its data to the swap-file. Afterwards we unmap the swapped-out page from virtual memory and map it to the currently accessed virtual memory page. Again we have to store some meta data in the PPT to be able to restore the mapping later. The result is shown in Figure 2c, with one page in the swap file (0), three pages (1, 2 and 3) in NV-RAM and two pages (4 and 5) are still unmapped.

With multiple concurrently running pVMs, page allocation and recovery may happen in different order. In this case the traditional allocation techniques are not sufficient to enable a recovery of allocated regions. Therefore we implemented a name-based allocation of regions in our *nvalloc*-function. Every time we call *nvalloc* to allocate a region of persistent memory, we assign a unique identifier to it. This identifier is a hash value generated from the pVM name provided by the NV-Hypervisor and an allocation sequence number. Figure 3 shows an example with two pVMs, each with one region.

To be able to find regions by their unique identifier we store the physical persistent memory address together with the size of the region in the table of contents (TOC) which is located at the beginning of the persistent memory. The identifiers are used here as a key to find the corresponding location of the region. In addition we also have a general configuration section (CONF) in persistent memory that contains information like total size, number of descriptors, page size and other management related information.

In summary, the allocation of persistent pages via the *nvalloc*-function consists of two parts. We have the name-based allocation of persistent memory regions and the VPM. This architecture provides competitive access to persistent memory by multiple applications without blocking. Every application

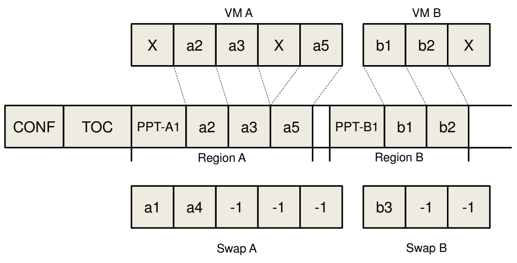


Figure 3. Name-based allocation

works with its own region of memory and any change to the state of a page is stored locally inside the PPT. Global locks are only required in case of allocation or freeing of regions. Since these tasks happen only sporadically the performance is not affected. For the deallocation of persistent memory regions we have implemented the *nvfree*-function. A call to this function removes all previously allocated pages, belonging to one region from persistent memory and the swap file. In this procedure we retrieve the initial address of the region from the TOC and consecutively read every page from it. Regardless of the location, by reading we ensure that the page is stored in persistent memory afterwards. Next, we set all data inside the page to zero, remove its descriptor from the PPT and unmap it from virtual memory. We used this basic approach to decouple the implementation of allocation and deallocation, however a tighter integration would lead to a more efficient solution.

IV. Persistent virtual machine life cycle

A pVM is created, executed, can be snapshotted and loaded again. Next, we describe the implementation of mechanisms, which manage the life cycle of a pVM and how the *vpmlib* is linked into this process.

Creation and execution. Initially every pVMs is equipped with a user-provided unique identifier. Next, during the creation of the pVMs our persistent memory allocator provided by *pvmlib* is used to place all sections in NV-RAM.

Fixation. The *fixation* is initiated by a signal (i.e. number 44 from the range of real-time signals) sent from the POD-driver to the NV-Hypervisor once a power outage is detected. Locks are applied to ensure that any initialized swapping operations is finished beforehand. During the fixation of a pVM we save the state of the *virtual environment* like it is accomplished for QEMU-based snapshots. This way we can apply the existing serialization capabilities of virtual devices in QEMU. However, contrary to taking a snapshot, we store the *virtual environment* as a file in persistent memory.

pVM Recovery. First, the NV-Hypervisor creates an «empty» pVM that has a default state for the *virtual environment* but does not have any virtual memory. Second, this pVM is connected with all regions of persistent memory, which were mapped to this pVM before the system went down. In a third step the default state of the *virtual environment* is replaced with the state we have stored previously in a file inside persistent memory. Next, the pVM is ready for operation.

V. Evaluation

We addressed the following questions in our evaluation:

- Does VPM influence the time to recover?
- Does VPM influence runtime performance? E.g. due to swapping of pages.

For all tests we used a uniform hardware stetting, as described next, either with or without NV-RAM enabled.

A. Environment

As previously mentioned, Temporality follows a multilayer architecture consisting of specialized hardware and software.

1) Hardware: We use a server platform from Viking Technology equipped with two six-core Xeon CPUs, 8 GB RAM and 4 GBs of Non-Volatile Dual In-line Memory Modules (NVDIMMs) as persistent memory technology. NVDIMMs are DRAM memory modules that are backed by a flash memory of the same size and a capacitor [5]. In case of a voltage drop the module uses the capacitor energy to mirror the DRAM state to flash memory. At system restart, the state stored in flash memory is written back to the volatile DRAM.

2) Software: Our software runs on a Linux (kernel version 3.4.12) as a host operating system. For simplicity only the POD and NVDIMM drivers reside in the kernel while all other parts of Temporality are implemented in the user space. As virtualization layer we used QEMU (1.4.2). However, there are no restrictions to implement Temporality within other hypervisors such as Xen and NOVA.

B. Influence on Recovery

In our preliminary work [4], we demonstrated that our NV-Hypervisor ensures safety of runtime data and reduces the time to restore the original performance of the system. To test the ability of a quick performance recovery we used one pVM with a typical Apache web server setup, including a MySQL database. As a workload we have chosen the *Sysbench OLTP Test Suite* [6] that creates a table of one million lines and measures the request response time for a random query. At a random point in time we initiated a power outage and observed the recovery procedure of the pVM. In summary, after booting the host OS we needed 93 s less to start the services with original throughput. For comparison, our reference system needed (566 s) to achieve the same throughput after database startup. In contrast to the benefits we had to add an overhead of 117 s which can be split up into 109 s to restore the state of our persistent memory and 8 s for connecting the NV-Hypervisor with all persistent pages. We were unable to influence the persistent

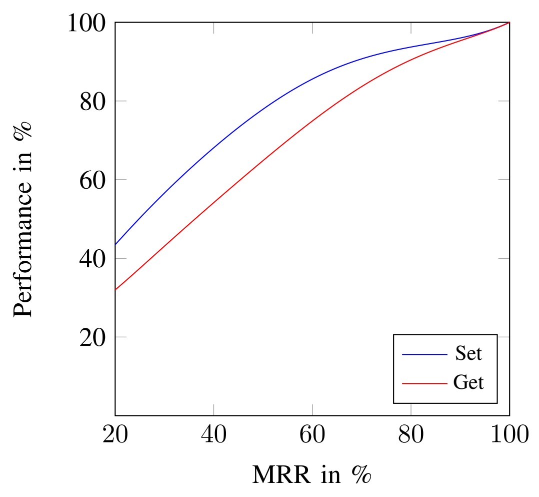


Figure 4. Redis performance for set/get requests as a function of MRR

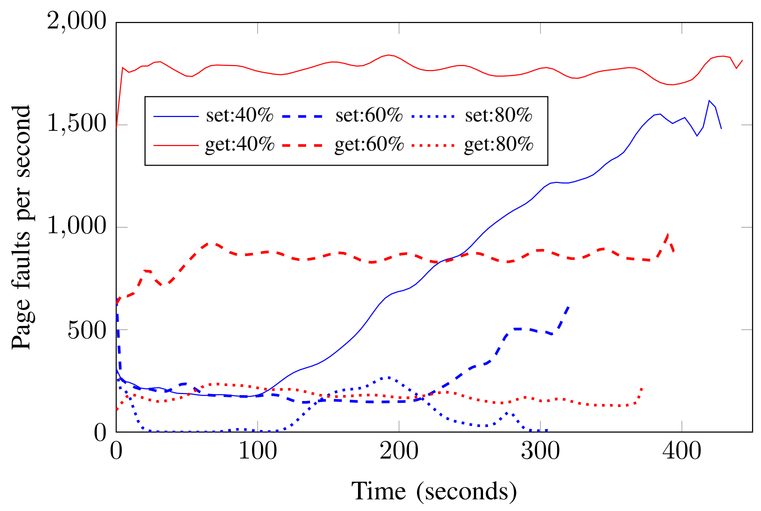


Figure 5. Number of page faults for different MRR (fixed workload)

memory initialization but we assume that persistent memory will not have this issue in near future. However with lazy page allocation and VPM we could remove the entire startup overhead of the NV-Hypervisor.

C. Runtime performance

To evaluate the performance impact of our implementation we used the key-value store Redis. For load generation the *set* and the *get* test have been used as provided by the *redis-benchmark* tool.

Initially, we have conducted experiments with three pVMs running in parallel. Each pVM was allocated a fixed amount of persistent memory, which was filled by the database. We experimented with different memory sizes of up to one GB. Since the results were similar, we present in this paper the numbers for small pVMs with 256 MB memory.

As long as no pages need to be swapped-out, we couldn't observe any performance degradation. This is not surprising since VPM allows any mapping or access of persistent pages to be executed in parallel. Synchronization is only required when we request new regions of persistent memory, which happens only on pVM startup.

However with increased persistent memory demand, we noticed a decrease in performance relative to the amount of swapping. Therefore, in a second test run, we used the same setup of three pVMs but changed the amount of swapped out pages. We present the configuration of our pVM by the memory residence ratio (MRR):
$MRR = InMemory/(InMemory + InSwap) * 100$
MRR describes the data size located in persistent memory ($InMemory$) relative to the total amount of allocated memory ($InMemory + InSwap$).

Our results presented in Figure 4 show the performance of a pVM. Performance is given as the percentage value of requests per second relative to a plain VM. As can be seen, without swapping (MRR=100%) the pVM performs at the same speed as a usual VM, both for *get* and *set* requests. However by decreasing the MRR the performance drops near linear for the *get* requests while *set* request performance stays at 90% until the MRR reaches 60%. This difference can be explained by the access pattern used in these benchmarks. During the *set* request benchmark, new memory is allocated linearly for storing keys. With a MRR of 90% the system will have to swap-out pages only when the amount of free memory falls below 10%. Opposed to this, the benchmark for *get* requests performs random *get* requests on a prefilled database. Therefore, the probability for *get* requests to access a swapped out page is very linear to the MRR.

To further analyze our swapping mechanism, we have investigated the amount of page-faults during the *get* and *set* benchmark. The results for a fixed workload of one million requests are presented in Figure 5. For random access pattern, during the *get* benchmark, the page-fault rate stays constant on one level, depending on the MRR. In contrast, for the linear access pattern of the *set* benchmark, the page-fault rate always starts low. When the access reaches the memory limit the page-fault rate begins to rise linearly since every new access involves a page swap. Thereby lower MRR leads to an earlier increase of page-faults since the memory limit is reached faster.

VI. Related work

Our work was inspired by *rapilog* [7] where residual energy was used to reactively store transaction logs of a database only in case of a power outage on persistent background storage. We use a similar approach but for a different purpose: to store the *virtual environment* in persistent memory.

Persistence can be provided at different levels. The most extensive approach is the idea of *whole-system persistence* [8] where everything becomes non-volatile. Another way is to make processes persistent like proposed in *NV-process* [9]. These approaches do not consider persistence of devices internal state, which we circumvented by a volatile hypervisor.

On a finer granularity only parts of an application can be made persistent. In particular *NV-Heaps* [10] and *Mnemosyne* [11] provide name-based allocators for objects with transactional semantics. For Temporality we have also implemented a name-based allocation in order to support the NV-Hypervisor. However our main goal is not to offer software primitives to create new persistent applications from scratch. We provide a platform which enables proprietary and legacy software to take advantage of persistence without any changes to the virtual appliances.

Since NV-RAM is byte-wise addressable, but current file systems are designed for block-wise access, there are several projects like BPFS [12], SCMFS [13], PMFS [14], and Aerie [15] which are devoted to develop new in-memory file systems. These projects have in common that they use persistent memory as a storage in combination with improved performance of the file system interface. This is achieved by means of removing layers in the Linux kernel (BPFS, SCMFS, PMFS) or by increasing the performance by direct access of persistent memory [15]. To make whole applications persistent with these file systems, substantial refactoring is needed. In contrast, our approach is a customized hypervisor, which provides persistence transparently to any application running inside a pVM.

The idea to use VMs in conjunction with persistent memory is presented in Ex-Tmem [16]. Here, persistent memory is used as a fast caching system for swapped out pages. However, this only improves performance during runtime and does not decrease recovery time.

VII. Conclusion

Power outages in Infrastructure-as-a-Service clouds may cause a loss of data and after a system restart the performance is often degraded. In this paper, we presented Temporality an virtualization layer that is robust against power outages by the utilization of NV-RAM and provides transparent persistence to virtual machines. By introducing VPM and name-based memory regions, provided by our *vpmlib*, we are able to run multiple pVMs in parallel. Moreover with VPM we can swap-out pages to the background storage and safely allocate more persistent memory than physically available.

References

[1] "2013 Cost of Data Center Outages," http://www.emersonnetworkpower.com/documents/en-us/brands/liebert/documents/white%20papers/2013_emerson_data_center_cost_downtime_sl-24680.pdf, Ponemon Institute, 2013.

[2] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under Unix," in *USENIX Winter Technical Conference*, 1995.

[3] "Calculating the Cost of Data Center Outages," http://www.emersonnetworkpower.com/documentation/en-us/brands/liebert/documents/white%20papers/data-center-costs_24659-r02-11.pdf, Ponemon Institute, 2011.

[4] V. A. Sartakov and R. Kapitza, "Nv-hypervisor: Hypervisor-based persistence for virtual machines," in *Dependable Systems and Networks (DSN)*, 2014.

[5] "Viking Technology. ArxCis-NV (TM) Non-Volatile Memory Technology," http://www.vikingmodular.com/products/arxcis/arxcis.html, 2012.

[6] "Sysbench," https://github.com/akopytov/sysbench/.

[7] G. Heiser, E. Le Sueur, A. Danis, A. Budzynowski, T.-l. Salomie, and G. Alonso, "RapiLog: reducing system complexity through verification," in *Proc. of the 8th ACM European Conference on Computer Systems*, 2013.

[8] D. Narayanan and O. Hodson, "Whole-system persistence," in *ACM SIGARCH Computer Architecture News*, vol. 40, 2012.

[9] X. Li, K. Lu, X. Wang, and X. Zhou, "Nv-process: A fault-tolerance process model based on non-volatile memory," in *Proceedings of the Third ACM SIGOPS Asia-Pacific Conference on Systems*, ser. APSys '12, 2012.

[10] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories," in *ACM SIGARCH Comp. Arch. News*, 2011.

[11] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ACM SIGARCH Computer Architecture News*, vol. 39, 2011.

[12] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009.

[13] X. Wu and A. Reddy, "Scmfs: a file system for storage class memory," in *Proc. of Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, 2011.

[14] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proc. of the 9th European Conference on Computer Systems*, 2014.

[15] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: flexible file-system interfaces to storage-class memory," in *Proc. of the 9th European Conference on Computer Systems*, 2014.

[16] V. Venkatesan, W. Qingsong, and Y. Tay, "Ex-tmem: Extending transcendent memory with non-volatile memory for virtual machines," in *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), 2014 IEEE Intl Conf on.* IEEE, 2014, pp. 966–973.