NV-Hypervisor: Hypervisor-based Persistence for Virtual Machines

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

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

***Abstract*—Power outages and subsequent recovery are major causes of service downtimes. This issue is amplified by the ongoing trend of steadily growing in-memory state of Internet-based services which increases the risk of data loss and extends recovery time. Protective measures against power outages, such as uninterruptible power supply are expensive, maintenance-intensive and often fragile. With the advent of non-volatile random-access memory (NVRAM) provided by commodity servers, there is a scalable, less costly and robust alternative to recover from power outages and other failures. However, as of today, off-the-shelf software is not ready for benefiting from NVRAM.**

We present NV-Hypervisor a lightweight hypervisor extension that transparently provides persistence for virtual machines. NV-Hypervisor paves the way for utilizing NVRAM in virtualized environments (i.e., infrastructure-as-a-service clouds) and protects stateful services such as key-value stores and databases from data loss and time-consuming recovery.

***Keywords*-NV-RAM; Hypervisor; Operating Systems; Cloud Computing;**

I. Introduction

In today's rapidly progressing information society, we rely on the availability of Internet-based services of all kinds. Increasingly often services are delivered on top of virtualized environments such as provided by infrastructure-as-a-service clouds. To cope with the demand for providing services that are available 24/7, service implementations as well as hosting infrastructure should be resilient to faults. One cause of service disruptions is power outages that can be addressed by fault tolerance features of service implementations (i.e., a crash-tolerant design) and additional infrastructure, such as uninterruptible power supply. In the first case, typically, a time-consuming recovery operation is required once power resumes and there is still a risk of data loss. In the second case, additional infrastructure is required which is expensive, maintenance-intensive and nevertheless fragile [1].

With the advent of non-volatile random-access memory (NVRAM) provided by commodity servers, in-memory data can be retained without an external source of power. It not only enables to tolerate power outages without data loss but also provides additional benefits such as preserving data in case of crashes in general. Thus, it offers the opportunity to implement measures for persistence and fault-tolerance in main memory. The latter simplifies service implementations and provides potential for performance improvements, e.g. write ahead logging to disk can be skipped [2].

Over the recent years multiple technical variants of how to implement NVRAM have been proposed. For example, byte-addressable memory, based on phase-change random-access memory [3] or spin-transfer torque random-access memory [4], has a read/write latency similar to commodity DRAM but promises beside persistence as well as higher capacity. While these technologies are already in a stable state, initial solutions that are already available on the market utilize commodity memory technology [5]. For example, in form of Non-Volatile Dual In-line Memory Modules (NVDIMMs) which are DRAM memory modules that are backed by a flash memory of the same size and a capacitor. In case of a voltage drop the module uses the capacitor energy to mirror DRAM state to the flash memory. At recovery time, the state stored in flash memory is written back to volatile memory.

The availability of persistent main memory needs to be reflected throughout the whole software stack. So far a number of different approaches have been proposed: at the user-space level as a new memory allocator [6], at the kernel level [7], where persistence becomes a new feature of processes [8], and system-wide [9]. All of these approaches require software modifications to utilize the support of NVRAM, thus legacy systems cannot profit right away from the introduction of NVRAM.

In this paper, we propose *hypervisor-based persistence* as a means to enable NVRAM-usage for legacy virtual machines. Thereby, neither system nor application software of a virtual machine which has to be adapted as NVRAM-support is transparently provided by the virtualization layer. We have implemented hypervisor-based persistence as part of *NV-Hypervisor*, which builds a lightweight extension to the QEMU [1] virtualization platform. For evaluating our approach, we have used a market available capacitor-backed NVDIMM solution [5] and measured the time to recover a database after a power outage. Results are promising, while a server without NVRAM support recovers slightly faster, our NV-Hypervisor-based implementation enables services to continue request

[1] QEMU - www.qemu.org

Table I
COMPARISON OF NV-RAM INTEGRATION ABSTRACTIONS

Type of Persistence	Memory Connection	NV-RAM Abstraction	Modification
Language/library-based	Hybrid, Parallel	Variables and objects	Kernel, libc, programs
Process-based	Hybrid, Parallel	Whole programs	Kernel
System-wide	NV-RAM	All programs and kernel	Kernel
Hypervisor-based	NV-RAM	All programs and kernel	No modification for VMs

processing at full speed, immediately after recovery, without any data loss.

In the remainder of this paper, we first provide an overview of related approaches. Next, Section III and Section IV describe the design and implementation of NV-Hypervisor, respectively. In Section V we present initial evaluation results and in Section VI we discuss about future improvements. Finally, Section VII concludes the paper.

II. USES OF NON-VOLATILE MEMORY

As NVRAM support builds on hardware that is integrated by software, we give a brief introduction to NVRAM. Next, we focus on related approaches by detailing how to utilize NVRAM to provide persistence at certain abstraction levels.

A. *Basics of non-volatile memory*

Data that are stored in NVRAM persist without an external source of power. Thus, it is preserved in case of a power outage or a system crash that causes a reboot.

As of today, NVRAM is implemented by multiple competing technologies. In particular spin-transfer torque random-access memory and phase-change random-access memory as well as capacitor-based solutions combined with traditional memory technology [5] are gaining ground. As each of them having individual strengths and weaknesses, it is unclear which technology will gain wide-spread acceptance. Due to its read/write performance and because it is byte addressable, it can be integrated as main memory or as an extension/replacement for classical storage. In this paper, we focus on how NVRAM can be utilized as main memory. Fortunately most existing proposals about how to integrate NVRAM support in system and application software are fairly independent from the actual hardware implementation and demand only for support of ordered, atomic writes and persistence.

Besides the technical realization of NVRAM, its hardware integration is an important aspect. Some researchers envision future systems as a hybrid architecture, where NVRAM and DRAM share the same system bus and the responsible memory controller maps the different types of memory to distinct address ranges [10]. Alternatively, a system could purely rely on NVRAM and omit volatile memory altogether [9].

In our work and in the midterm, we consider a hybrid architecture as more realistic. Despite recent progress, NVRAM comes as attached with additional costs that hinder an immediate and complete supersession of conventional volatile memory.

B. *Software-based integration*

The software support of NVRAM determines at which abstraction level main memory persistence is provided.

Language- and library-based persistence: Persistent memory can be offered to user-space applications as a new type of memory provided by a special allocator that manages the available persistent memory. In this way, programs only benefit from non-volatile memory, if they are explicitly programmed against a specific API offered by approaches implementing *language- or library-based persistence*. Such an approach is beneficial for systems where NVRAM and DRAM co-exist, e.g. due to limited availability of non-volatile memory.

NV-Heaps [7] and *Mnemosyne* [6] are examples for NVRAM abstractions at the application level. While the former provides a set of primitives to manage persistent objects offered by so-called *NV-Heaps*, the latter introduces beside other things a special keyword to C in order to make variables persistent.

Such a language- or library-based persistence requires modification to the kernel, the system libraries (i.e, the `libc`-environment) and the applications themselves.

Process-based persistence: More coarse-grained is the idea of providing the abstraction of *process-based NVRAM* support as proposed by *NV-process* [8]. At creation time of a process, it is either mapped to volatile or non-volatile memory (i.e. as a NV-process). In case of a power failure, the NV-process instances persist in NVRAM and can continue running from where they were left off as soon as power returns and the operating system reboots. NV-process uses independent virtual and physical memory organization mechanisms implemented by the operating system, which guarantees the same mapping between physical and virtual addresses of the process after reboots.

Legacy and proprietary user-space software can take advantage of process-based persistence, but it requires modifications to the kernel.

System-wide persistence: Alternatively, an entire system can run solely based on non-volatile memory. *Whole-system persistence* [9] involves all parts of the system being executed directly in non-volatile memory. Volatile data in CPU is protected against loss by use of a *flush-on-fail* mechanism which works at the time of power failure and saves volatile

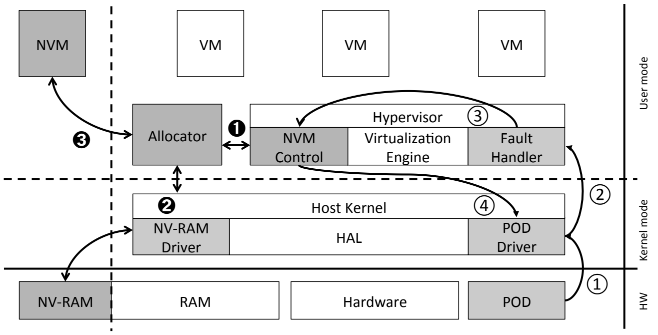

Figure 1. System overview

data such as registers and caches stored in the CPU to NVRAM.

System-wide persistence transparently supports legacy and proprietary user-space software, but it requires modifications to the kernel.

Hypervisor-based persistence: In this paper, we propose *hypervisor-based persistence*. It introduces NVRAM at the level of virtual machines. On creation of a virtual machine it is decided to be volatile or non-volatile. In the latter case it can be recovered from a power outage or a system crash.

Hypervisor-based persistence requires integration of non-volatile memory at the virtualization layer, but otherwise it is transparent to virtual machines.

Comparison: Table I summarizes the discussed related approaches. While language- and process-based persistence explicitly introduces a hybrid system, where volatile and non-volatile state co-exist, whole-system and hypervisor-based persistence only relies on non-volatile memory. In the latter case this is of course restricted to virtual machines. Finally, hypervisor-based persistence is fully transparent to virtual machines, as NVRAM is integrated at the virtualization layer. Therefore, it is well-suited for legacy systems and can be easily integrated in virtualized environments, such as infrastructure-as-a-service clouds.

III. NV-Hypervisor

In order to provide hypervisor-based persistence for virtual machines, we developed NV-Hypervisor as a lightweight layer that extends a commodity hypervisor by integration of non-volatile memory. In the following, we detail our system assumptions and the architecture of NV-Hypervisor as well as basic operations like handling a power outage and the subsequent recovery procedure.

A. Assumptions and hardware support

We assume a commodity server system that is equipped with DRAM and NVRAM, both connected to a shared system bus. Such a hybrid system architecture is in line with market available system designs. Furthermore, in our system model all CPU state such as registers and caches are volatile and will be lost in case of a power outage.

To preserve recent execution results that only reside in the volatile CPU state, as well as to implement essential housekeeping functionalities regarding the management of non-volatile memory, we assume the availability of a power outage detector (POD). This detector measures the current voltage of the external power supply and generates an interrupt if power fails (see Heiser et al. [11] for a possible design). Thereby, we assume that the residual energy contained in capacitors of the power supply is enough to save relevant volatile CPU state to NVRAM after detecting a power outage. In fact, recent experiments indicate that these capacitors provide residual energy for generating stable voltage between 40 to 60 ms after detecting a power outage [11], which suffices our demands (see Section V).

B. System architecture

NV-Hypervisor builds a lightweight extension to an existing virtualization platform. Our architecture assumes a host operating system that manages the hardware and a hypervisor providing support for executing virtual machines.

Figure 1 outlines a system overview aligned with our QEMU-based implementation. The host operating system is extended by a POD-driver. This driver handles interrupts indicating an imminent power outage, detected by the POD. The NVRAM-driver provides support for non-volatile memory and is responsible for adding physical NVRAM addresses to the memory map (❷).

Beside these kernel-level extensions, a special memory allocator for managing the NVRAM is provided as a library (❶). The hypervisor uses this library to allocate VMs in NVRAM (❸). Furthermore, the hypervisor is equipped with a power outage handler that is triggered by the POD-driver in case of power drop. It is responsible for saving volatile management state of persistent virtual machines, e.g. CPU state, to NVRAM. Finally, internal management functions and the user interface of the hypervisor are extended to enable the management of persistent virtual machines.

C. Creation of a persistent virtual machine

Creation and startup of a persistent virtual machine is very similar to an ordinary volatile virtual machine but requires two modifications:

- If the user instructs the hypervisor to create a persistent virtual machine, our memory allocator has to assign space in NVRAM.
- Information about the assigned memory and other long-lived state needs to be made persistent, e.g. by using a configuration file, to enable a recovery.

D. Handling of a Power Fault

Imminent power outage is handled by the following four consecutive steps:

① The POD fires a non-maskable interrupt, once a drop of the input voltage is detected.

② The interrupt is caught by the POD-driver, which notifies the NV-Hypervisor about the upcoming power outage.

③ The NV-Hypervisor saves volatile state of virtual devices and virtual CPU states of persistent VMs to NVRAM.

④ The POD-driver regains control and stops any memory operations, freezes CPUs and flushes the caches.

After this procedure all persistent VMs and their environment are saved and can be recovered once the system is restarted. Finally, it has to be noted that step number three can be omitted if all management state of persistent VMs is directly stored in NVRAM.

E. Recovering

In our current design we do not make further assumption than that VMs and their management state are stored in NVRAM. Thus, the host operating system performs an ordinary boot process like a system without support for non-volatile memory. Once the host operating system and the NV-Hypervisor are up and running, persistent VMs have to be recovered. This is achieved by retrieving information about persistent VMs that were running when the power outage occurred. Taking this information into account, persistent VMs are recovered by reassigning their memory and integrating virtual device information. Finally, the virtual CPU state of the recovered VMs is restored and the VMs are marked as ready for execution.

IV. Implementation

Our NV-Hypervisor prototype extends the QEMU virtualization platform and integrates NVDIMMs [5], a NVRAM-solution provided by Viking Technology. The NVDIMMs are implemented by DRAM memory modules that are backed by a flash memory of the same size and a capacitor. In case of a power drop, the module uses the capacitor energy to mirror the DRAM state to the flash memory and vice versa at reboot time.

A. Integration of NVDIMMs.

In line with the proposed architecture, we have implemented an allocator for managing NVRAM. It consists of two parts: a kernel module that adds NVRAM into the system memory map and provides operations for reading and writing configuration registers of NVDIMMs; and a user-space library (`libnvram.so`) that provides functions (i.e. `nv_alloc()`, `nv_free())`, `nv_init()`) for allocating and freeing regions in NVRAM.

B. NV-Hypervisor core services.

The detection of a power outage is implemented by a POD handler that comes attached with the NVDIMMs. Communication between the POD-handler and our QEMU extension is implemented as a blocking `ioctl`-syscall. We added a thread to our QEMU extension that issues the `ioctl`-syscall and blocks inside the POD-driver until a power outage is detected. For managing persistent VMs, we have implemented two additional QEMU Monitor commands: `dump-devices` and `nv-restore`. The former saves the environment of the VM into storage, the latter performs the recovery of persistent VMs by merging the VM image in NVRAM with virtual devices that are restored from storage. Two additional flags were added to QEMU: `nv-restore` and `-nvm`. The first one tells QEMU that `do_nv_restore` function should be performed after start automatically. The second flag obliges QEMU to use NVRAM for allocation of VM.

V. Evaluation

As evaluation platform we used a NVDIMM equipped server (2 Xeon CPUs and 8 GB RAM, with 4GBs of it being non-volatile memory) provided by Viking [5], running Linux (kernel version 3.4.12) as a host operating system. As an implementation basis for NV-Hypervisor we utilized QEMU (version 1.4.2). Our initial evaluations focused on the timing behavior of NV-Hypervisor during a power outage and a comparison between NV-Hypervisor and a vanilla Linux server when recovering a memory-heavy VM.

A. Handling of a power fault

As detailed in Section IV our current implementation is based on the assumption that the system has enough residual energy to continue execution for 30-50ms after detecting a power outage [11], [9]. To ensure that we are under the limit, we instantiated a large VM with eight virtual cores and a default set of devices including graphic and network support. Next, we measured the time for processing the non-maskable power-outage-interrupt provided by the POD and saved all volatile state that belongs to the persistent VM. The size of volatile state was 80 KB. Saving took only 8 ms and ensures that a limited set of persistent VMs can be preserved in case of a power outage.

However, this limitation can be overcome by allocating memory for virtual devices of VMs in non-volatile memory but requires additional changes to QEMU. Moreover, the use of hardware virtualization techniques like VT-x can reduce volatile state in the hypervisor. According to Intel's specification [12], the NMI driver performs a VM exit event if an interrupt arises when the CPU is occupied by a VM. The event stops the VM execution and places the VM state in NVRAM automatically. In this case it is no longer necessary to send a message from the NMI handler to the Hypervisor. These additional modifications of the current NV-Hypervisor implementation remove the limitation on the count of VMs that can be saved during the power outage.

Regarding the implementation of the POD, we evaluated the Viking-provided detector which utilizes the *Power Good* signal generated by the power supply unit. However, this

Table II
BOOT PROCESS COMPARISON

Boot phase	Commodity system (sec)	NV-Hypervisor (sec)
DB warm up	566	n/a
DB recovery	54	n/a
GuestOS boot	31	n/a
QEMU start	0.2	8
Host boot	108	108
BIOS	15	15
NVDIMMs init	n/a	109
Server boot	36	36
$\sum$	**810.2**	**276**

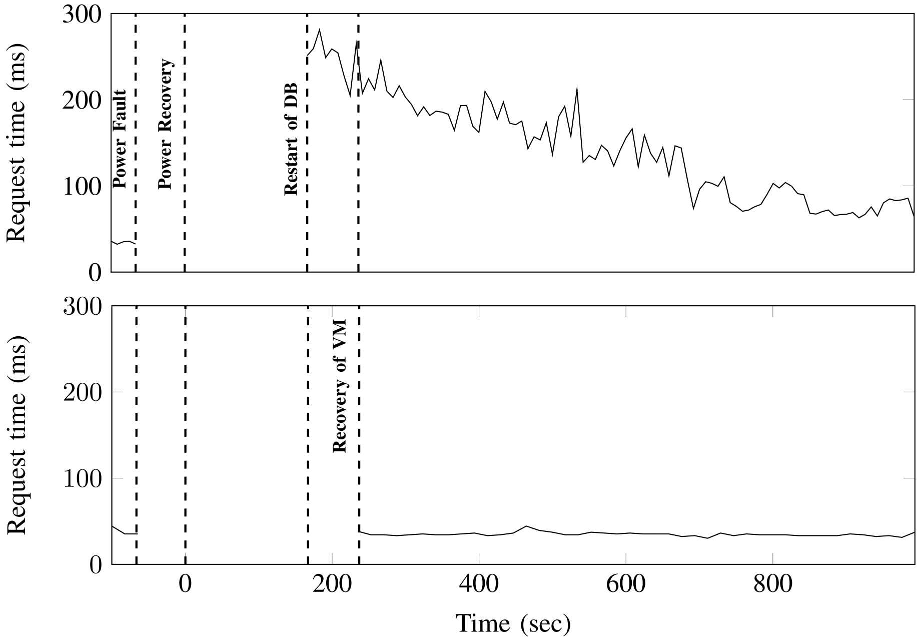

Figure 2. Process of a database recovery: Commodity system versus NV-Hypervisor

solution provides a smaller advance warning time than documented by related approaches [11], [9]. In fact, it provides only enough time to quiescent the memory bus for securing all ongoing operations regarding the NVDIMMs, but is too small for saving the volatile state of multiple persistent VMs. Therefore, in the remainder of our evaluation, we assume the availability of a POD similar to the one proposed by Heiser et. al [11].

B. Recovery of a database server

To evaluate hypervisor-based persistence, we were interested in the recovery behavior of a virtual machine hosting a memory-heavy service.

We created a virtual machine containing a typical Web-based application, composed of an Apache web server instance as front end and a MySQL database as back end. As workload we selected *sysbench oltp test suite* [13].

Next, we compared the recovery behavior of an unmodified vanilla Linux system running plain QEMU with our NV-Hypervisor prototype.

Table II details the different phases during the boot process. The actual boot process of the host operating system is quite similar. In fact, the NV-Hypervisor-based system is even slower as the NVDIMMs have to be initialized and checked. As we used an early evaluation platform, this boot step might get faster once NVDIMMs reach the final product stage. After QEMU/NV-Hypervisor is running, the situation changes as for the commodity system: the VM and its services have to be started while in case of NV-Hypervisor this is not necessary. Still, up to this point, the NV-Hypervisor-based solution is about 13% slower.

However, the picture changes once the actual runtime behavior of both implementation is taken into account. A relational database typically has a long warm-up phase until queries can be answered at full speed. Accordingly, we measured the time until both settings were fully operational. This was performed using the *sysbench* utility which creates a table with 1000000 lines and measures request response time for a random query applied to this table. While for the NV-Hypervisor-based solution, there is virtually no warm-up time, and it took 566 seconds for the commodity system.

Figure 2 details the warm-up process. First, we see normal operations for both system. After normal operation we induce a power outage and start a recovery at time zero. While the commodity solution continues operation after 244.2 seconds the NV-Hypervisor-based requires 31.8 seconds more. Furthermore, we see that initially the queries of the commodity system are about a factor of 5 slower than NV-Hypervisor-based instance.

Our evaluation shows that NV-Hypervisor has a constant time for any VM recovery, which heavily depends on the hardware support of NVRAM. When taking into account the actual service response time, the commodity system demands for a factor of 2.9 longer until the recovery is fully finished.

VI. Future work

Both types of virtualization techniques, hardware-based as well as binary translation, use virtual memory for VM allocation. Our current implementation instead directly allocates NVRAM for VMs, i.e. we use a one-to-one mapping thereby omitting the use of virtual memory. Unused virtual pages of common VMs can be stored in a swap file to free physical space for other VMs. Since NVRAM is attached to the memory controller like ordinary DRAM, our approach could also utilize virtual pages, and hence, some fragments of VMs could be placed in a swap file if physical memory is not enough. However, swapped out pages could be buffered in a storage cache, and this volatile cache is lost in case of power fault. Recovering of swapped out pages of persistent VMs is not the only obstacle when utilizing virtual memory. Virtual to physical memory mappings are placed in the memory management unit (MMU). The MMU state is also volatile and needs to be preserved in case of a power failure. As identified in the context of mobile devices, mixed volatile/non-volatile memory settings might even have some more pitfalls [14]. In summary, virtual memory support for persistent memory, hardware virtualization support, interaction of volatile hardware and non-volatile software - all of those are issues for future research.

VII. Conclusion

In this paper, we introduced hypervisor-based persistence as a means to integrate NVRAM to provide persistent virtual machines. Our NV-Hypervisor builds a lightweight realization of this abstraction and initial evaluation results based on the recovery of memory-heavy services are promising. For the future, we envision NV-Hypervisor to support the use of virtual memory and extend persistence to the host operating system thereby shortening the overall recovery time of persistent VMs. With the widespread availability of NVRAM in commodity servers, hypervisor-based persistence provides the basis to immediately utilize it for legacy VMs, especially in the context of infrastructure-as-a-service clouds.

Acknowledgments

We would like to thank Thomas Knauth and anonymous reviewers for their helpful comments. Also we thank Bertil Munde and Viking Technology for access to hardware.

References

[1] “2013 Cost of Data Center Outages,” Ponemon Institute, Tech. Rep., 2013.

[2] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, “Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46, 2013, pp. 421–432.

[3] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, “Architecting phase change memory as a scalable dram alternative,” in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 2–13.

[4] H. Li and Y. Chen, “An overview of non-volatile memory technology and the implication for tools and architectures,” in *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE’09.* IEEE, 2009, pp. 731–736.

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

[6] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight persistent memory,” in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 91–104.

[7] 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 Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 105–118.

[8] X. L. K. L. X. Wang and X. Zhou, “NV-process: A Fault-Tolerance Process Model Based on Non-Volatile Memory,” 2012.

[9] D. Narayanan and O. Hodson, “Whole-system persistence,” in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1. ACM, 2012, pp. 401–410.

[10] G. Dhiman, R. Ayoub, and T. Rosing, “PDRAM: a hybrid PRAM and DRAM main memory system,” in *Design Automation Conference, 2009. DAC’09. 46th ACM/IEEE.* IEEE, 2009, pp. 664–669.

[11] G. Heiser, E. Le Sueur, A. Danis, A. Budzynowski, T.-l. Salomie, and G. Alonso, “RapiLog: reducing system complexity through verification,” in *Proceedings of the 8th ACM European Conference on Computer Systems.* ACM, 2013, pp. 323–336.

[12] Intel, “Intel 64 and IA-32 Architectures Software Developer’s Manual,” *Volume 3B: System Programming Guide, Part*, vol. 2, 2011.

[13] “Sysbench,” http://sysbench.sourceforge.net/.

[14] S. Kannan, A. Gavrilovska, and K. Schwan, “Reducing the Cost of Persistence for Nonvolatile Heaps in End User Devices,” in *Proceedings of the 20th IEEE International Symposium On High Performance Computer Architecture*, 2014.