Dependable Non-Volatile Memory

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

Rouven Scholz
TU Braunschweig
Braunschweig, Germany
scholz@ibr.cs.tu-bs.de

Phil Lindow
TU Braunschweig
Braunschweig, Germany
lindow@ibr.cs.tu-bs.de

Niklas Lehnfeld
TU Braunschweig
Braunschweig, Germany
lehnfeld@ibr.cs.tu-bs.de

Marc A. Kastner
TU Braunschweig
Braunschweig, Germany
kastner@ibr.cs.tu-bs.de

Rüdiger Kapitza
TU Braunschweig
Braunschweig, Germany
kapitza@ibr.cs.tu-bs.de

ABSTRACT

Recent advances in persistent memory (PM) enable fast, byte-addressable main memory that maintains its state across power cycling events. To survive power outages and prevent inconsistent application state, current approaches introduce persistent logs and require expensive cache flushes. Thus, these solutions cause a performance penalty of up to 10× for write operations on PM. With respect to wear-out effects, and a significantly lower write performance compared to read operations, we identify this as a major flaw that impacts performance and lifetime of PM. In addition, most PM technologies are susceptible to soft-errors that cause corrupted data, which implies a high risk of a permanently inconsistent system state.

In this paper, we present ***DNV Memory***, a library for persistent memory management. For securing allocated data against power outages, multi-bit faults that bypass hardware protection and even usage violations, *DNV Memory* introduces *reliable transactions*. Additionally, it reduces writes to PM by offloading logging operations to volatile memory, while maintaining *durability on demand* by an early detection of upcoming power failures. We compare *DNV Memory* to pmemobj, a persistent object-store, and show that our system only causes a moderate overhead. In fact, our benchmark results indicate that *DNV Memory* is even faster than pmemobj for transactions of moderate size.

1 INTRODUCTION

Persistent memory (PM) summarizes various technologies of byte-addressable non-volatile memory (NVM) with access times and throughput comparable to DRAM. By allowing direct access to persistent data through in-memory file systems and experimental libraries like the Non Volatile Memory Library (NVML) [44] or Mnemosyne [46], persistent memory can be used as a fast storage that outperforms commodity flash memory-based solutions.

Recent research is centered around the SNIA[1] NVM programming model (NPM) [40], that manages persistent memory through a PM-aware file system [19, 45]. For accessing persistent data directly, files are mapped into the user-space. In order to keep data consistent in case of power failures, library support [14, 44, 46] provides transactional semantics. Internally, all approaches rely on persistent logs and frequent cache flushes to ensure durability. However, this causes up to 10 times more write operations to persistent memory [30]. With respect to the limited write endurance of currently available PM-technologies a significant lifetime degradation is the consequence. Furthermore, Phase Change Memory (PCM) and resistive random-access memory (RRAM) based PM have higher write latencies than DRAM, thus, frequent write operations to persistent memory further decrease performance.

Additionally, transient fault resilience is an important and so far largely neglected issue for persistent data in system software. For instance, PCM suffers from resistance drifts, RRAM has sneak currents and battery backed DRAM is still susceptible to environmental radiation during runtime. The latter also causes transient bit flips in the circuits and the static random-access memory (SRAM) buffers that contribute to more than 40% of a memory chip's area [50].

While proposals for hardware solutions exist [7, 39, 50], they negatively impact the circuit space, energy demand and latency, and on top lead to increased device cost. Although these are not limiting factors for high-end server hardware, we assume that due to aggressive pricing, no extensive hardware fault tolerance measures will appear in entry level servers as well as consumer and embedded devices. However, for applications using persistent memory,

SYSTOR '18, June 4–7, 2018, HAIFA, Israel

[1] Solid State Storage Initiative

state corruptions need to be considered as severe since these corruptions are persistent and plain system restarts cannot mitigate such issues.

In this paper we present *DNV Memory*, a system architecture for efficient, yet robust allocation and access of persistent memory. With *DNV Memory* application programmers can build static and dynamic data structures of any kind that survive system restarts. To keep data consistent in the presence of power failures, *DNV Memory* employs software transactions. On top of this basic functionality, *DNV Memory* makes three core contributions:

- *First*, we identify that applying frequent flushes and persistent logs highly increases the number of write operations on persistent memory, which has a severe negative impact on dependability, lifetime and performance. As a countermeasure, *DNV Memory* enforces *durability on demand* by utilizing a hardware power failure detector and leveraging hybrid systems that are equipped with volatile and persistent memory. All write overhead caused by transaction logs is offloaded to volatile memory and copied to persistent memory only in case of an imminent power failure. Additional cache flushes are only performed when needed, i.e. when a power failure demands it. Our evaluation proves that even under high load, durability can be achieved on demand and in time, with more than 32 ms left before the final power outage.
- Our *second* contribution is focused on securing persistent data from transient faults. For all persistent data in user-space, *DNV Memory* transparently stores redundancy information and uses *reliable transactions* to verify data whenever they are accessed.
- Omitting transactions when accessing persistent memory is essentially a usage error that may lead to state corruption in case of power failures, transient faults or concurrent access. Therefore, our *third* contribution provides programming support that enforces transactions for persistent memory access.

This paper is structured as follows: we present the system model of *DNV Memory* in Section 2. The main contributions can be found in Section 3 which highlights our concepts in detail. Section 4, briefly describes implementation details of *DNV Memory* and is followed by our evaluation in Section 5. In Section 6 we discuss related work followed by concluding remarks in Section 7.

2 SYSTEM MODEL

We believe that hybrid system architectures equipped with both, volatile and persistent main memory, will dominate the market in the near future. This means, processes still have a volatile and a persistent state. While PCM is the most promising PM technology today, PM modules can also be built using RRAM, spin-transfer-torque magnetoresistive random access memory (STT-MRAM) or even battery-backed DRAM. Thereby, all processes in a system should be able to access persistent memory directly through load and store operations in order to achieve optimal performance.

CPU caches can be used to further speed up access to persistent data, however, in order to survive power failures, cache lines containing data from persistent memory must be flushed and the data must reach the *Durability Domain* of the persistent memory module before the machine shuts down due to a power loss. This requires platform support in form of an asynchronous DRAM refresh (ADR) [3] or a *Flush Hint Address* [4]. Under these premises, we assume that word-level power failure atomicity is reached.

Currently, hybrid systems are built mainly for server environments, however, there is also a great opportunity for hybrid systems in consumer and embedded devices. Here, persistent memory can be used to speed up access to databases [6], can enable very efficient sleep modes [12] or can simply be used to speed up access to frequently used persistent data. We assume that both, new and legacy applications, would benefit from utilizing persistent memory. In order to limit the refactoring overhead for legacy applications, accessing persistent memory should be designed with the concepts of traditional volatile memory in mind.

The importance of persistent data may vary depending on the use case. Errors in self-contained persistent data like documents or media files can be acceptable if the effect is not severe or backup copies are available. However, persistent meta data must be extremely robust in order to prevent a corrupted application state, as this could only be repaired by purging and recreating the entire persistent state. Thus, we argue that selective protection of persistent state is favorable.

Depending on the used main-memory technology, various effects exist that may cause transient faults. The main source of faults in DRAM cells is environmental radiation, leading to state transitions over time. This means that the longer the data is stored, the more likely it is affected by such an error. Novel persistent memory technologies like PCM or RRAM are inherently robust to radiation, as their state transitions demand a higher energy, but at the same time they introduce new sources of transient errors. RRAM cells, for instance, suffer from sneak-currents [39] that cause non-uniform distributed cell transitions. For PCM cells the *short-term resistance drift* is the dominant source of transient faults [48, 50]. It corrupts the information that is stored in the cell right after it was written. In contrast to DRAM, the probability of errors does not increase over time but with the number of write operations. Additionally, all PCM and RRAM have a limited write endurance that lies in the range

of 10^6 up to 10^{10} operations [6, 19, 25]. Once worn out, the cell's value can only be read but not modified anymore.

We assume that all SRAM cells inside the CPU are guarded by hardware fault tolerance and are sufficiently reliable to ensure correct operation. Of course reliable DRAM supporting hardware error correction code (ECC) exists and persistent memory can be protected by hardware solutions too [35, 37–39, 48, 50]. However, the common hardware ECC mechanisms only provide singe-bit-error correction, double-bit-error detection (SECDED) capabilities, which has been reported to be not always sufficient [41, 42]. We assume that due to economic reasons not every persistent memory module will support the highest possible dependability standard, leaving a fraction of errors undetected. Some persistent memory modules may even lack any hardware protection. This paves the way for software-based dependability solutions.

3 CONCEPTS OF *DNV MEMORY*

The main goal of our design is to provide the familiar *malloc* interface to application developers for direct access to persistent memory. At the same time, we want data stored in persistent memory to be robust against power failures, transient faults, and usage errors.

Our core API functions (see Table 1 (a) and (b)) resemble the interface of `malloc` and `free`. As developers can use native pointers or references to interconnect data, this provides the opportunity to build all persistent data structures such as lists, vectors, trees, or hash-tables. The only additional requirement for persisting legacy volatile structures with *DNV Memory* is using our API functions and wrapping all persistent memory accesses in atomic blocks (see Table 1 (e)).

These atomic blocks provide ACID[2] guarantees for thread safety, and additionally preserve consistency in case of power failures. Furthermore, *DNV Memory* combines software transactional memory (STM) with the allocator to manage software-based ECC. Every data word that is accessed during a transaction is validated and can be repaired if necessary.

In order to store entry points to persistent data structures that survive process restarts, *DNV Memory* provides the possibility to create static persistent variables (Table 1 (c) and (d)). In the next sections we will discuss each of our features in detail.

3.1 Tolerating Power Failures

If a power failure occurs meanwhile persistent data structures are modified they might be in an inconsistent state after restart. In the example presented in Figure 1, that shows two core functions of a linked list, persistent memory will leak if a power failure occurs right after `dnv_malloc` or directly

[2]Atomicity, Consistency, Isolation, Durability

```
1  void push_front(widged_t widged) {
2    __transaction_atomic {
3      node_t* node = dnv_malloc(sizeof(node_t));
4      node->pload  = widged;
5      node->next   = head_;
6      head_ = node;
7    } //transaction commit
8  }
9
10 widged_t pop_front(void) {
11   __transaction_atomic {
12     widged_t widged = head_->pload;
13     node_t* node    = head_;
14     head_           = head_->next;
15     dnv_free(node);
16     return widged;
17   } //transaction commit
18 }
```

Figure 1: Example of a persistent linked list

before `dnv_free`. Even more severe would be a power failure during the `dnv_*` functions as internal structures may become inconsistent.

To prevent inconsistent data or persistent memory leaks, *DNV Memory* follows the best practices from databases and other PM-allocators [14, 44, 46] and wraps operations on persistent memory in atomic blocks. This can be achieved with STM provided by modern compilers or libraries like TinySTM [22, 23]. In the example (Figure 1) everything that is wrapped in the `__transaction_atomic{. . .}` block is done atomically. The transactions must also be applied to the allocator itself, as its internal state must be stored in persistent memory too.

State of the Art. Power failures must not be able to interrupt a transaction in a way that leaves persistent data in an inconsistent state. A common state of the art approach to prevent this is redo logging [47]. During the transaction commit, all modifications to persistent data are first written to a *persistent* log. Then, in a second step, the actual data is modified. After each step a memory fence is needed and it must be ensured that the data reaches the durability domain before the next step is executed. Otherwise, log and data modifications may be partially durable when a power failure occurs. For persistent memory, this approach has several negative implications that affect performance, lifetime and dependability.

First, to ensure durability, a memory fence combined with cache line flushing is required. This enforces not only a slow write operation to persistent memory, but also invalidates the cache line. Thus, follow-up reads will have to pay the latency of a full memory access. *Second*, logs do not only

Table 1: Overview of the *DNV Memory* application programming interface (API)

Category	Function	Description	Ref.
Core API	void* dnv_malloc(size_t sz)	allocates persistent memory like **malloc(3)**	(a)
	void dnv_free(void* ptr)	releases persistent memory like **free(3)**	(b)
Static Variables	DNV_POD *variable*	statically places *plain old data* in PM at definition	(c)
	DNV_OBJ *variable*	statically places the object in PM at definition	(d)
Transactions	__transaction_atomic{...}	atomic block with ACID guarantees and reliability	(e)

store the data that is modified but also contain additional information like the address, timestamps etc. For instance, the write-back log in tinySTM [22, 23], that is essentially a redo log, occupies a full cache line for every written word during a transaction. Even if applied only on persistent memory, this increases the write operations by 8× and likewise reduces the lifetime of persistent memory. Fortunately, writes are typically sparse [30], thus write-back logs can be held well in caches. However, flushing the cache, on every commit prevents taking any advantage from data locality. While Intel proposed new instructions like `CLWB` to overcome this problem, a significant overhead on the memory bus still remains that will likely impact the performance. The fact that log-memory is typically allocated once during a process aggravates the wear-out even more. Because all write operations take place on a local memory range, wear leveling algorithms like bank rotation [37] cannot efficiently distribute writes across memory cells, thus permanent faults may arise quickly. *Finally*, increasing the amount of writes will equally increase the probability of transient faults on PCM-based persistent memory due to the short term resistance drift effects that may occur when data is written.

Durability on Demand. Other than the previous work, *DNV Memory* aims at minimizing the writes to persistent memory. We store all transaction logs in volatile memory and utilize a power failure detection to enforce duarility on demand. When a power outage is imminent the operating system copies the write-back logs back to persistent memory in order to prevent state inconsistency. Therefore every thread has to register its volatile memory range for the write-back log at our kernel module, which in turn reserves a persistent memory range for a potential backup copy. After restart, the write-back logs are restored from persistent memory, and every unfinished commit is eventually repeated. If too many registrations occur, so that copying all volatile write-back logs to persistent memory cannot be ensured within a given time frame, the kernel module may respond with a decline. In that case, the thread has to fall back to using persistent memory for its write-back log.

Since durability is actually required only in case of a power failure or process termination, memory fences and cache flushing can be performed on demand. This preserves persistent data inside the CPU cache and consequently reduces writes to persistent memory. Additionally, according to our fault model, persistent data inside the cache is less susceptible to transient faults and can be accessed faster.

Enforcing durability on demand, requires the ability to detect power failures in advance. For embedded devices the power outage detection is a part of the *brownout*-detection and state of the art [34]. On servers and personal computers power outages can be detected via the PWR_OK signal according to the ATX power supply unit (PSU) design guide [1]. Although the PWR_OK signal is required to announce a power outage at least 1 ms in advance, much better forecasts can be achieved in practice. For instance some Intel machines provide a power failure forecast of up to 33 ms [31]. An even better power failure detection can be achieved by inspecting the input voltage of the PSU with a simple custom hardware [24]. With this approach power failures can be detected more than 70 ms in advance which leaves more than enough time to enforce durability and prevent further modification of persistent data.

System Crashes and Hardware Failures. Crashes that are not caused by power failures can be handled just like power failures if durability can be secured. For instance, our kernel module is aware of any process using persistent memory that terminates and enforces durability in that case. Crashes in the operating system kernel, can be handled either as part of a kernel panic procedure or by utilizing a system like Otherworld [18].

3.2 Reliable Transactions

In order to protect persistent data from corruption, *DNV Memory* reserves additional memory in each allocation that is meant to store ECC data. Afterwards fault tolerance is provided through reliable transactions.

As described in the previous section, all accesses to persistent memory should be wrapped by atomic blocks in order to protect persistent data from power failures. These atomic

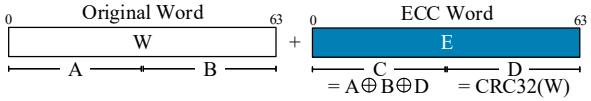

Figure 2: *DNV Memory* ECC

blocks simply wrap all read and write operations in TM_LOAD and TM_STORE functions provided by the STM-library. For instance, line 5 in Figure 1 will be transformed into:

```
5    TM_STORE(&node->next, TM_LOAD(&head_));
```

This way, every word access is controlled by the STM-library. In combination with support from the memory allocator this can be exploited to provide transparent fault tolerance.

Essentially, any ECC can be used to provide fault tolerance in software. For instance, we considered the SECDED hamming code that is common in hardware protected memory. It protects 64-bit words with additional 8 bit, resulting in a 12,5% memory overhead. However, if implemented in software, the hamming code would highly impact the performance of the application. Additionally, as already mentioned, we do not think that SECDED is enough to protect persistent data. Consequently, we decided to implement an ECC that provides a high multi-bit error correction with a memory overhead no more than dual modular redundancy. In addition, we want a fast error detection in software by exploiting commonly available hardware support. This requirements lead to the following algorithm:

Whenever a *data word W* is written inside an atomic block, an *ECC word E* is created and stored in the additional space that the allocator has reserved. In theory, any fault-tolerant encoding is possible as long as error detection can be conducted in a few CPU cycles.

Error Correction Code. For *DNV Memory* we combine cyclic redundancy check (CRC) for fast error detection with an error location hint. Thus, we subdivide E into two halves C and D as shown in Figure 2. The *error detection* half word D is generated with CRC32c ($D = CRC32c(W)$). We have chosen CRC because hardware support is available on many architectures, including most commodity CPUs. Additionally, with CRC32c that is supported by *SSE 4.2*, a hamming distance of 8 is achieved on a word length of 64 bit [26]. Without further assistance, error correction of up to 3 bits can be achieved by guessing the error location. However, by augmenting the CRC-based error detection with an *error location hint C*, less trials are needed and more bit-errors can be corrected. Inspired by RAID level 5 [33], we subdivide the data word W into two halves A and B and compute C according to Equation 1.

$$C = A \oplus B \oplus D \tag{1}$$

Error Detection and Correction. The data validation takes place during a transaction whenever a word W is read for the first time. At that point, we recompute E' from W and compare its value with E. Normal execution can continue if both values match. Otherwise error correction is initiated.

Since errors can be randomly distributed across W and E we start the error correction by narrowing the possible locations of errors. Therefore we compute the *error vector* F via Equation (2) that indicates the bit position of errors.

$$F = A \oplus B \oplus C \oplus D \tag{2}$$

This information is, however, imprecise as it is unknown whether the corrupted bit is located in A, B, C or D. Thus, for f *errors detected by* F, 4^f repair candidates R_i are possible that are computed via Equation (3). The *masking vectors* M_a, M_b, M_c, M_d are used to partition F between all four half words.

$$\begin{aligned} R_i &= W_i \| E_i \\ W_i &= A \oplus (F \wedge M_a) \| B \oplus (F \wedge M_b) \\ E_i &= C \oplus (F \wedge M_c) \| D \oplus (F \wedge M_d) \end{aligned} \tag{3}$$

To find the repair candidate R_s that contains the right solution, each R_i needs to be validated by recomputing E'_i from W_i and compare it to E_i. In order to repair all errors, exactly one R_s must be found with matching E'_i and E_i. For instance, if all errors are located in A the repair candidate using $M_a = F$ and other masking vectors set to zero will be the correct result. Additionally, all combinations need to be considered that have an error at the same bit position in two or all half words, as these errors extinguish each other in C.

Please note, that the set of repair candidates may yield more than one solution that can be successfully validated if more than three errors are present. To prevent a false recovery, all repair candidates must be validated for up to n errors. As an optimization step, we estimate n by counting the population in $E \oplus E'$ and limit the result to a maximum of $n = 7$.

Prevention of Usage Errors. To optimize the performance in a cache-aware way, we store the ECC words interleaved with the original words W as presented in Figure 3. However, this interleaved data layout cannot be accessed correctly outside atomic blocks because the original layout is always expected here. Unfortunately, omitting atomic blocks around persistent memory access is a very common mistake. We encountered such usage errors in every single STAMP benchmark and whenever we ported or wrote persistent applications ourselves. Since the access to PM outside atomic blocks should be prevented to keep data consistent during power failures, we introduce the concept of a *transaction staging (TxStaging) section* as shown in Figure 3. All memory that is allocated by *DNV Memory* returns addresses residing in the TxStaging section. The same applies to the location of persistent static

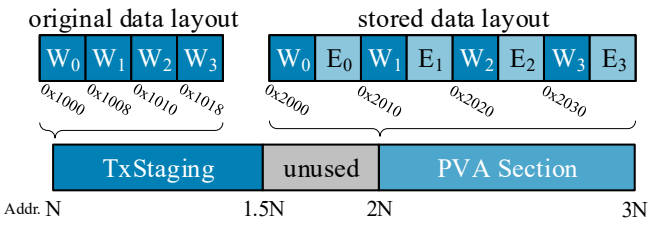

Figure 3: *DNV Memory* persistent data layout and memory sections

variables. The TxStaging section is only a reserved virtual address space without any access rights. Consequently, any access to this segment will cause a segmentation fault that is easy to debug. However, inside an atomic block every access to the TxStaging section is intercepted by the STM library and redirected to the persistent virtual address (PVA) section where the actual persistent data is stored. To simplify the address transformation, the PVA section should be located at the address of the TxStaging section multiplied by 2. For instance, assuming the TxStaging section begins at address `0x1000` the PVA section should be placed at `0x2000`. In that case a 32 byte object that is located in the address range from `0x1000` to `0x101f` will be transformed into the address space `0x2000` to `0x203f` as shown in Figure 3.

4 ARCHITECTURE

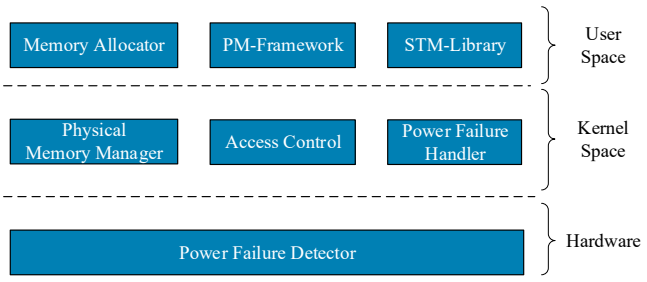

Figure 4: Architecture

We implemented *DNV Memory* on Linux in the form of a user space library with a small companion kernel module and a hardware power failure detector as shown in Figure 4. Our design does not require any changes to the operating systems kernel or the machine itself. All components are pluggable and can be replaced by more extended solutions if needed.

All user-space code is written in C++ and compiled with an unmodified GCC 5.4.0. A small linker script extension provides additional sections like the TxStaging or the PVA section as shown in Figure 3. In the following sections we discuss each component of our architecture in detail.

4.1 User Space Library

Our user-space library consists of three components: the *memory allocator*, the *PM-framework* and the *STM-library*.

Memory allocator. For the memory management we use an own memory allocator that implements the concept of binning similar to jemalloc [21] or hoard [9]. The allocator essentially provides multiple heaps which manage the requested memory in blocks of equal size of 2^n bytes. Whenever a user calls `dnv_free`, the released block is attached to a *free-list* that is stored in place of the freed memory. While this design is relatively straight forward, it eases the integration of the proposed persistence and dependability features.

PM-framework. While the memory allocator is responsible for managing memory in blocks of 2^n bytes, our PM-framework manages persistent memory mappings in chunks that combine a coherent set of 2^n pages. Whenever a heap requires more memory a new chunk is allocated by our kernel module. In order to control the memory mapping of chunks, *DNV Memory* has its own persistent *virtual address manager* that is based on the buddy algorithm. It is implemented as an array of linked lists and the dynamic memory is managed within our memory allocator.

To be able to restore persistent memory on application startup, each memory chunk is identified with a location Id (LID). This is stored together with the virtual address and the size of the chunk, inside a red-black tree (rb-tree) called *Chunk Table*. When a process is restarted the Chunk Table is traversed and every chunk that has its information stored in that table is restored.

STM-library. In order to provide an easy accessible API for atomic blocks we use the language extensions from GCC and redirect the transformed code to our STM-library. Similar to [46] we use TinySTM 1.0 a lightweight, word-based STM implementation [22, 23] to provide power failure tolerance. Our reliable transactions are currently implemented as an extension of TinySTM. However, they do not rely on TinySTM and can be implemented on top of any other STM-library. Please note, that due to the provided durability on demand, the entire STM-library remains in volatile memory, including the write-back log.

4.2 Kernel Module

Current state of the art [40] suggests using a specialized PM-aware file system like PMFS [19] or BPFS [15] to manage physical persistent memory. However, according to [45] the features of file systems are not always needed and performance can be improved when data structures are tailored

Table 2: Hardware setup

Component	Description
PC Model	Viking ArxCis-Developers-Kit
PSU	2× Ablecom PWS-702A-1R (700 W)
CPU	2× Intel Xeon E5-2640 (6 cores)
Main Memory	4 GiB DDR3 @ 1333 MHz
Persistent Memory	4 GiB Viking ArxCis-NV [2]
OS	Ubuntu 16.04.3 (4.4.0-119-generic)

for the task in the user space. Therefore, we provide only a lightweight solution that manages persistent memory on chunk granularity via the buddy algorithm. This has a second advantage in minimizing the need for persistent memory inside the kernel that also needs to be secured from power failures and transient faults. Our implementation is based on a binary tree that is stored as an array and all state transitions are implemented in a fault-tolerant, atomic way.

To isolate chunks from other applications that use the same kernel module, each chunk is stored under a key that must be provided by the user when it is mapped. For a production version of *DNV Memory*, there are no fundamental issues to integrate standard access control mechanisms provided by the operating system.

Power failure forecasts are implemented with a small custom external hardware *Power Failure Detector* that is plugged in the same power source as the computers PSU. For comparability with Rapilog [24], we have chosen a similar hardware layout. It emits a signal via serial port that in turn triggers an interrupt. Then, the *Power Failure Handler* of the kernel module is responsible to enforce durability. It stops all but one cores, transfers the write back logs of running commits to persistent memory and finally flushes all caches.

5 EVALUATION

With the evaluation of *DNV Memory* in this section, we aim to answer the following questions:

(1) Is it possible to detect power failures sufficiently early to perform all necessary clean-up operations and save the write back log to persistent memory?
(2) How much time do we need to repair a word with multiple bit errors and how reliable is our ECC approach when more than 3 bits have flipped?
(3) What is the performance of *DNV Memory* compared to other persistent memory libraries?
(4) How large is the performance overhead implied by reliable transactions?

For the investigation of our system we use micro benchmarks, applications from the STAMP benchmark suite [29] and the key-value store Memcached with retrofitted transactions. All benchmarks were conducted on the same machine with the specification shown in Table 2.

Table 3: Critical tasks related to a power failure

Measurement	Workload	Time in ms		
		avg	min	max
Stop CPU and flush cache	kernel compilation	2.6	2.3	3.3
	idle	4.9	4.4	5.6
Store write back log	kernel compilation	4.2	3.8	4.8
	idle	8.0	7.4	8.6
Machine shutdown	kernel compilation	36.8	34.6	39.4
	idle	32.8	25.2	36.8

5.1 Timing Analysis

DNV Memory relies on an early detection of power failures in order to avoid flushing the CPU caches whenever durability of persistent data needs to be ensured. Therefore, we investigated the time between the detection of a power failure and the eventual machine shutdown as well as the duration of critical cleanup operations.

To measure the time between power failure detection and the machine shutdown, we repeatedly wrote timestamps into persistent memory after detecting a power failure and derived the duration from the first and the last timestamp. In addition we stored a timestamp after stopping all cores and flushing their caches and when the write back log was fully stored in persistent memory. For the computation of the timestamp we use the `RDTSC` assembly instruction [5] to keep the effort as low as possible.

The timings for 100 enforced power failures, with and without workload, are presented in Table 3. All numbers presented in this table are relative to the detection of a power failure and include all previous steps. With a power drain of approximately 100 W when idling and 250 W during kernel compilation with 24 threads, one would expect that the timings are better when the machine is idling. The reality however shows the opposite. As the CPU frequency is scaled down to 1.2 GHz when idling, the cleanup operations take twice the amount of time. Furthermore, the more power is drained, the more residual energy remains in the PSU's inductor and capacitors on power failure, which explains the extended time until the machine shuts down. We also measured the time until the machine shutdown on a laptop and a desktop machine with a 365 W PSU. Both achieved more than 70 ms in every measurement. All in all, the PSU provides sufficient residual energy to fulfill all cleanup operations, despite the workload.

5.2 Error Correction Analysis

To investigate the capabilities of our ECC algorithm we conducted one billion fault injection experiments on random words with an error of up to seven random bits in a word. In a total of 12,163 cases, we failed to repair a seven-bit error, while no successful repair returned a wrong result. We also conducted the time to repair an error for 200 million fault injections per bit error length. As can be seen in Figure 9 the time to repair an error increases exponentially with the number of flipped bits. However, even for correcting seven-bit errors, the the mean repair time is less than 1.4 ms which is acceptible considering the low probability of such errors. In contrast, without any error, the validation only takes 34 ns.

5.3 Performance Analysis

We evaluated our system with the following library configurations to estimate the performance of *DNV Memory*.

- **RTx**: *DNV Memory* with reliable transactions
- **Tx**: *DNV Memory* with plain STM
- **NoTx**: *DNV Memory* without transactions
- **NVML**: pmemobj v1.1, with C++ bindings v1.0.0 [44]

All experiments were repeated 100 times.

Micro Benchmarks. For investigating the performance of transactional memory allocation, deallocation, writes and reads under controlled conditions we implemented a transactional, circular doubly linked list.

As part of a transaction, *pushBack* allocates persistent memory for a new list element and adds it to the tail of the list. Similarly, *popFront* removes an element from the lists head and releases the persistent memory. The results presented in Figure 5 and 6 show average values for a list of 500 elements with varying payloads. As can be seen in the figures, the reliable transactions approximately increase the runtime of plain transactional execution by factor 1.2 to 1.4×. However, even with with reliable transactions, *DNV Memory* outperforms pmemobj by factor 4 to 7× in the pushBack and by factor 3× in the popFront benchmark. This outcome is partially explained by the cache flushes that pmemobj uses to establish durability. For instance in the case of pushBack, cache flushes increase runtime by factor 1.4 to 2.6×. Another source that affects performance are the smart pointers, which are also copied to the redo log when modified.

We also investigated the pure persistent data access without involving the allocator. In *iterate & write* the persistent list is traversed in one single transaction modifying 8 byte in each element. Accordingly, *iterate & read* traverses the list in one big transaction and reads 8 byte from each element. This time we kept the payload size fixed at 128 bytes and varied the number of list elements which correlates with the transaction log size.

The results are shown in Figure 7 and 8. For pure read access, we observe that raising the size of transactions increases the performance gap between reliable and plain transactions from factor 1.2 to 1.9×. Since large read operations heavily benefit from the cache, the doubled memory demand for dependability takes its toll here. With larger write sets, however, the performance impact of dependability becomes less prevalent as cache inefficiency effects become the dominant factor.

Comparing to pmemobj, *DNV Memory* is again faster by factor 2 to 3× with plain transactions in the iterate & read benchmark. This performance difference is caused by the persistent smart pointers that are used in pmemobj to interconnect the list elements. In case of the iterate & write benchmark, pmemobj additionally suffers from the object granularity and as it always stores the full payload in its redo log instead of the modified 8 byte. For a list of 10 elements this leads to an extreme 67× runtime increase compared to *DNV Memory* with reliable transactions. However, the write-back log of tinySTM scales worse than the redo log of pmemobj, because every modified word leads to an entry with a size of a cache line.

In summary, as long as write sets stay moderate, *DNV Memory* greatly benefits from durability on demand and the word-based transaction model provided by tinySTM [23]. However, comparing the transactional execution with uninstrumented code reveals potential for optimizations.

For the application benchmarks we omit the comparison with pmemobj and focus on the performance impact of dependability and transactions in general.

Application Benchmarks. To test the performance impact under realistic conditions we applied *DNV Memory* to the STAMP benchmark applications [29]. Additionally, we retrofitted *DNV Memory* into the Memcached key-value store. All key-value pairs were stored in persistent memory with our allocator and every access was wrapped into reliable transactions. The bars depicted in Figure 10 show the mean runtime of each benchmark. All values are relative to plain transactional execution (Tx) and the error bars represent the 95% and the 5% quantile. Over all applications, a median runtime of 106,5% is achieved with reliable transactions. We observed that applications above this median have a workload that is dominated by reads or short transactions, hence the overhead of data validation has a higher impact here.

We also looked at the general runtime impact of transactions. Our results are roughly in line with the original STAMP publication [29]. On our machine, transactions moderately increase the runtime by zero to 73% in most cases. Please note, that benchmarks suffering significant performance impact from reliability, are similarly affected by standard transactions too. In contrast to our microbenchmarks however, the

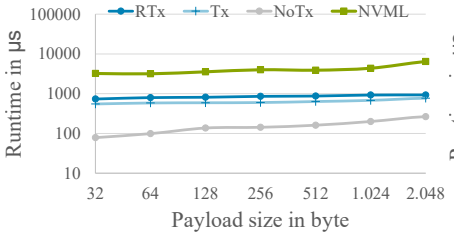

Figure 5: PushBack

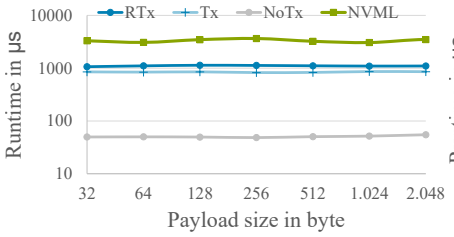

Figure 6: PopFront

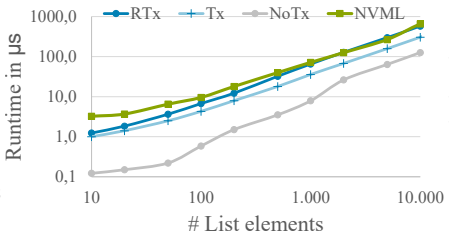

Figure 7: Iterate & read

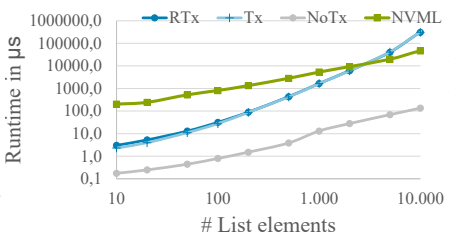

Figure 8: Iterate & write

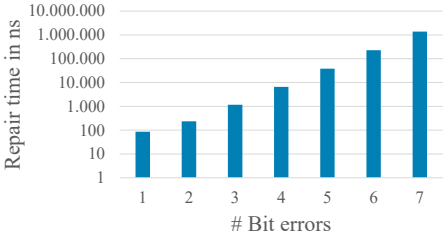

Figure 9: Time to repair

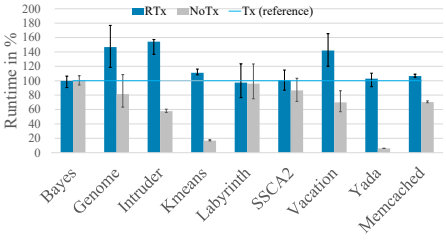

Figure 10: Applications

results indicate that transactions can be used efficiently in practice. Kmeans and Yada, however, exhibit different behavior with a very high transactional impact on the runtime. For Yada, we introduced many additional transactions in order to access all persistent memory reliably, which led to performance penalties. We assume, the result can be improved with code optimizations and a more rigorous partitioning between persistent and non persistent data. In the case of Kmeans we identified caching effects as a likely cause for the high performance impact, since the transactions are small in this case and most of the time is spent outside transactions. In summary, our results indicate a very acceptable performance impact for reliable transactions.

Performance Impact of Durability on Demand. We also investigated the performance gain from applying the durability on demand. The results highly correlate with the cache efficiency of the benchmark. For instance, no performance impact was achieved for our linked list iteration with large write sets, Bayes, Labyrinth and Yada. However, in the benchmarks pushBack, popFront, Genome and Kmeans, as well as in iterating our linked list with moderate read sets, a performance increase of up to 4,4× was observed.

6 RELATED WORK

Among different ways of using persistent memory like process wide persistence [12, 28, 31, 36] and specialized file systems [16, 20, 45, 49] *DNV Memory* shares the most similarities with libraries that provide safe access to a persistent heap [10, 13, 14, 46]. Mnemosyne [46] shows the overall steps that are needed to build a persistent heap while NV-Heaps [14] focuses mainly on usability aspects. Both Libraries rely on a transactional memory model that stores logs in persistent memory and executes expensive flush operations to ensure data consistency in presence of power failures. In order to improve performance, the memory allocator of Makalu [10] guarantees the consistency of its own meta data without the the need of transactions. However, it does not extend this ability to the data stored within. Thus, library support, similar to Mnemosyne[46] or Atlas [13] is still needed to enforce durability.

Although *DNV Memory* shares the transactional model and the goal to provide a persistent heap, our approach differs in several ways. First, *DNV Memory* aims at improving the performance and lifetime of persistent applications by reducing the amount of writes to persistent memory without sacrificing durability guarantees. Second, our architecture is modular and does not require pervasive kernel or hardware modifications. Our components are standalone and can be integrated easily in other systems. Finally, *DNV Memory* provides transparent dependability guarantees that none of the previous work has covered.

Other important works that loosely fall into this area are pVM [25], Atlas [13] and NVML [44]. The first, aims at efficient scaling of volatile memory into persistent memory regions by using the non-uniform memory access (NUMA) interface of Linux. However, durability and consistency is only guaranteed inside the OS kernel and not extended to the user-space. In Atlas, power failure atomicity is enabled for critical sections, that are guarded by synchronization objects (e.g. mutexes). However, applications typically try to minimize critical sections to cover only concurrent access. Therefore, not necessarily all persistent memory is accessed

concurrently and a power failure between two critical sections may still result in an inconsistent state. We believe that in general a transactional memory model better fits the idea of a transition between two consistent states. Finally, NVML grounds on the NVM programming model and provides a pack of libraries all related to accessing persistent memory. The core of this library is *pmemobj* an object store that is very similar to a persistent heap at first glance. However, technically this heap is placed in a file and suffers from all the drawbacks of the file system interface. Nevertheless, we use this library for comparison in our evaluation since its high level programming model shares some similarities with *DNV Memory*.

Dependability. All previous approaches have in common that they neglect dependability. As explained in Section 2, this is risky because without countermeasures, any error in persistent memory can render the whole persistent data useless. This is especially critical for persistence models that span whole applications [12, 28, 31, 36] as without additional measures every crash will become permanent.

In order to improve dependability of persistent memory, recent research focused on hardware solutions. With ECP [37], Safer [38] and Pay-As-You-Go [35], wear out-related hard errors can be corrected. ReadDuo [48] and Shevgoor et. al. [39] provide solutions only for soft errors, while FREE-p [50] aims at tolerating both, hard and soft errors. However, hardware protection increases access latency, adds a memory overhead and usually leads to increased device cost. This price may be too high if dependability is not always needed, as it is the case for persistent data that is backed up on a secondary storage, for example. Moreover, the available hardware protection may not always be sufficient for some mission critical tasks, as indicated by recent studies on DRAM memory in high performance computing clusters [41, 43].

Software solutions are a good alternative if the dependability needs to be improved in a tailored way. Plenty of work has been done to secure commodity main memory state [8, 11, 17, 32]. However, the proposed solutions are either limited to specific programming models (Samurai [32], PASC [17], GOP [11]), impose a high overhead (SEP [8]), or only cover bit-flips inside the CPU (HAFT [27]). Despite previous work, the dependability measures provided by *DNV Memory* are fully transparent to the application and do not imply any programming model beside using transactional memory semantics, which anyway are required to protect persistent data from inconsistencies that can be caused by power outages. Moreover, *DNV Memory* heeds the fault model of emerging persistent memory technologies and is fully applicable to concurrent applications.

Power Failure Detection. The detection of power failures is used in [12, 24, 31, 36] to flush the state of the CPU to storage or NVM. From these works, Rapilog [24] deserves a special mention as it combines volatile logs with a power failure detector hardware in order to speed up transactions in a database. We adapt this solution to persistent memory for reducing the write pressure.

7 CONCLUSION

With the advent of persistent memory in commodity systems, data that is traditionally stored in volatile main memory is about to become persistent. However, when stored persistently, data is also more exposed to bit flips that may render it unusable without further measures.

In this paper we presented *DNV Memory*, system support for dependable non-volatile memory allocator. Unlike previous approaches, *DNV Memory* enforces durability on demand, which in turn reduces write operations on persistent memory and therefore improves reliability, lifetime and performance. For tolerating power failures, *DNV Memory* uses software transactions that also include and secure the allocator itself. Our system even goes one step further and provides fault tolerance via software transactional memory. In essence *DNV Memory* protects data at word granularity, with an ECC word that is capable of detecting and correcting a random distributed seven-bit error, which is by far more than common hardware protection offered by server-class volatile main memory.

We implemented *DNV Memory* as a lightweight user space library with a fault-tolerant companion kernel module that does not rely on any pervasive kernel or compiler modifications. Our evaluation shows that power failures can be detected early on to conduct all necessary cleanup operations. Furthermore, we validate our ECC approach, investigate the performance impact of fault tolerance and compare *DNV Memory* to pmemobj. The results show that fault tolerance, can be achieved with less than 6.5% increase in runtime. Finally, with durability on demand *DNV Memory* outperforms pmemobj by a factor of 2 up to 67× for moderate transaction sizes.

ACKNOWLEDGMENT

This work was partly supported by the German Research Foundation (DFG) under priority program SPP1500 grant no. KA 3171/2-3.

REFERENCES

[1] 2005. ATX12V Power Supply Design Guide. (2005). http://formfactors.org/developer%5Cspecs%5CATX12V_PSDG_2_2_public_br2.pdf

[2] 2012. Viking Technology. ArxCis-NV (TM) Non-Volatile Memory Technology. http://www.vikingmodular.com/products/arxcis/arxcis.html. (2012).

[3] 2014. SNIA NVDIMM Messaging and FAQ. (2014).

[4] 2016. Advanced Configuration and Power Interface Specification (Version 6.1). http://www.uefi.org/sites/default/files/resources/ACPI_6_1.pdf. (2016).

[5] 2016. Intel 64 and IA-32 Architectures Software Developer's Manual. (2016).

[6] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 707–722. https://doi.org/10.1145/2723372.2749441

[7] Manu Awasthi, Manjunath Shevgoor, Kshitij Sudan, Bipin Rajendran, Rajeev Balasubramonian, and Viii Srinivasan. 2012. Efficient Scrub Mechanisms for Error-prone Emerging Memories. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/HPCA.2012.6168941

[8] D. Behrens, S. Weigert, and C. Fetzer. 2013. Automatically Tolerating Arbitrary Faults in Non-malicious Settings. In *Proc. of the Latin-American Symp. on Dependable Comp.*

[9] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/378993.379232

[10] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. 2016. Makalu: Fast Recoverable Allocation of Non-volatile Memory. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[11] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. [n. d.]. Generic Soft-Error Detection and Correction for Concurrent Data Structures. *IEEE Transactions on Dependable and Secure Computing* PP, 99 ([n. d.]).

[12] Björn Cassens, Arthur Martens, and Rüdiger Kapitza. 2016. The Neverending Runtime: Using new Technologies for Ultra-Low Power Applications with an Unlimited Runtime. (2016). http://www.ewsn.org/file-repository/ewsn2016/325_330_cassens.pdf?attredirects=0

[13] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proc. of the International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*.

[14] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ACM SIGARCH Comp. Arch. News*.

[15] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

[16] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proc. of the Symposium on Operating Systems Principles*.

[17] Miguel Correia, Daniel Gómez Ferro, Flavio P Junqueira, and Marco Serafini. 2012. Practical hardening of crash-tolerant systems. In *Proc. of the 2012 USENIX Annual Technical Conf.*, Vol. 12.

[18] Alex Depoutovitch and Michael Stumm. 2010. Otherworld: giving applications a chance to survive OS kernel crashes. In *Proc. of the European conference on Computer systems (EuroSys)*.

[19] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/2592798.2592814

[20] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proc. of the European Conference on Computer Systems (EuroSys)*.

[21] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the BSDCan Conference, Ottawa, Canada*.

[22] Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proc. of Symposium on Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/1345206.1345241

[23] Pascal Felber, Christof Fetzer, Torvald Riegel, and Patrick Marlier. 2010. Time-Based Software Transactional Memory. *IEEE Transactions on Parallel and Distributed Systems* 21 (2010). https://doi.org/doi.ieeecomputersociety.org/10.1109/TPDS.2010.49

[24] Gernot Heiser, Etienne Le Sueur, Adrian Danis, Aleksander Budzynowski, Tudor-loan Salomie, and Gustavo Alonso. 2013. RapiLog: Reducing System Complexity Through Verification. In *Proc. of the European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/2465351.2465383

[25] Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. 2016. pVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 13, 16 pages. https://doi.org/10.1145/2901318.2901325

[26] P. Koopman. 2002. 32-bit cyclic redundancy codes for Internet applications. In *Proc. of Dependable Systems and Networks (DSN)*. https://doi.org/10.1109/DSN.2002.1028931

[27] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2016. HAFT: Hardware-assisted Fault Tolerance. In *Proc. of the European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/2901318.2901339

[28] Xu Li, Kai Lu, Xiaoping Wang, and Xu Zhou. 2012. NV-process: A Fault-tolerance Process Model Based on Non-volatile Memory. In *Proc. of the Asia-Pacific Conference on Systems (APSys '12)*.

[29] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *Proc. of the International Symposium on Workload Characterization (IISWC)*.

[30] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*.

[31] Dushyanth Narayanan and Orion Hodson. 2012. Whole-system persistence. In *ACM SIGARCH Computer Architecture News*, Vol. 40.

[32] Karthik Pattabiraman, Vinod Grover, and Benjamin G. Zorn. 2008. Samurai: Protecting Critical Data in Unsafe Languages. In *Proc. of European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/1352592.1352616

[33] David A Patterson, Garth Gibson, and Randy H Katz. 1988. *A case for redundant arrays of inexpensive disks (RAID)*. Vol. 17. ACM.

[34] Philips Semiconductors [n. d.]. *Protecting Microcontrollers against Power Supply Imperfections*. Philips Semiconductors. AN468, published in May 2001.

[35] Moinuddin K Qureshi. 2011. Pay-As-You-Go: low-overhead hard-error correction for phase change memories. In *Microarchitecture (MICRO), 2011 44th Annual IEEE/ACM International Symposium on*. IEEE, 318–328.

[36] V. A. Sartakov, A. Martens, and R. Kapitza. 2015. Temporality a NVRAM-based Virtualization Platform. In *Proc. of the Symposium on Reliable Distributed Systems (SRDS)*. https://doi.org/10.1109/SRDS.2015.42

[37] Stuart Schechter, Gabriel H. Loh, Karin Strauss, and Doug Burger. 2010. Use ECP, Not ECC, for Hard Failures in Resistive Memories. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 141–152. https://doi.org/10.1145/1815961.1815980

[38] Nak Hee Seong, Dong Hyuk Woo, Vijayalakshmi Srinivasan, Jude A. Rivers, and Hsien-Hsin S. Lee. 2010. SAFER: Stuck-At-Fault Error Recovery for Memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, Washington, DC, USA, 115–124. https://doi.org/10.1109/MICRO.2010.46

[39] Manjunath Shevgoor, Naveen Muralimanohar, Rajeev Balasubramonian, and Yoocharn Jeon. 2015. Improving memristor memory with sneak current sharing. In *Computer Design (ICCD), 2015 33rd IEEE International Conference on*. IEEE, 549–556.

[40] SNIA. 2015. NVM Programming Model. (2015). http://www.snia.org/tech_activities/standards/curr_standards/npm

[41] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. 2015. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[42] Vilas Sridharan and Dean Liberty. 2012. A Study of DRAM Failures in the Field. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 76, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389100

[43] V. Sridharan and D. Liberty. 2012. A study of DRAM failures in the field. In *Proc. of High Performance Computing, Networking, Storage and Analysis (SC)*.

[44] PMDK team at Intel Corporation. [n. d.]. NVM Library. ([n. d.]). http://pmem.io/nvml/

[45] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M Swift. 2014. Aerie: flexible file-system interfaces to storage-class memory. In *Proc. of the European Conference on Computer Systems (EuroSys)*.

[46] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*, Vol. 39.

[47] Hu Wan, Youyou Lu, Yuanchao Xu, and Jiwu Shu. 2016. Empirical study of redo and undo logging in persistent memory. In *Proc. of Non-Volatile Memory Systems and Applications Symposium (NVMSA)*.

[48] Rujia Wang, Youtao Zhang, and Jun Yang. 2016. ReadDuo: Constructing Reliable MLC Phase Change Memory through Fast and Robust Readout. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*. IEEE, 203–214.

[49] Xiaojian Wu and AL Reddy. 2011. SCMFS: a file system for storage class memory. In *Proc. of High Performance Computing, Networking, Storage and Analysis*.

[50] Doe Hyun Yoon, Naveen Muralimanohar, Jichuan Chang, Parthasarathy Ranganathan, Norman P Jouppi, and Mattan Erez. 2011. FREE-p: Protecting non-volatile memory against both hard and soft errors. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 466–477.