NVall: A Crash-Resistant and Kernel-Compatible Memory Allocator for NVRAM

Dustin Nguyen
Ole Wiedemann
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

Jörg Nolte
Brandenburgische Technische Universität Cottbus-Senftenberg (BTU)

Wolfgang Schröder-Preikschat
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

ABSTRACT

Byte-addressable non-volatile memory is essentially persistent, but slower main memory that needs to be managed accordingly. Typical memory allocators for volatile memory are highly efficient today, but usually never had to be designed to keep their state in main memory consistent at all times against the background of system crashes. In this paper we present NVall, a crash-resistant kernel-level memory allocator for non-volatile RAM (NVRAM). The allocator works in a transactional manner, uses existing volatile memory to improve the performance of normal operation and is able to recover its volatile state from persistent data after a system crash. We implemented the allocator for the FreeBSD kernel and compare its performance against the standard (non-crash-resistant) in-kernel allocator of FreeBSD.

KEYWORDS

Operating System, NVRAM, Memory, Storage, FreeBSD

1 INTRODUCTION

Upscaling memory capacity to DRAM can be expensive in terms of performance and cost. DRAM refresh power scales proportionally with memory capacity, also further downscaling of capacitor size is difficult. This speaks in favor of replacing DRAM as main memory. Its high density, low standby power and low cost per bit make NVRAM a promising alternative to DRAM. Nevertheless, the access latency can be much higher than with DRAM, in addition to the lower bandwidth and asymmetric read/write performance [14].

However, these hardware-related handicaps can be compensated to a certain extent, by integrating NVRAM into the virtual memory with problem-aware handling in the operating system (OS). Following the pattern of [15], the latter includes the OS together with all machine programs controlled by it residing directly in NVRAM and executing there—as in the early days of computing technology based only on core memory. But this is only half the truth: Whether such an approach is worthwhile also depends on the overhead that certain OS functions imply in order to be able to run them correctly in the NVRAM in exceptional cases.

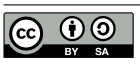
An important and exemplary function in this context concerns the efficient and kernel-compatible dynamic memory management, that is, a crash-resistant NVRAM-based memory allocator (NVall), which is the focus of the paper. The allocator is integrated into the FreeBSD OS and can be used by its modules to store and retrieve data with NVRAM, so that they can also utilize the memories characteristics.

Non-volatile main memory. For the most part NVRAM can be treated like any other volatile RAM, such as DRAM (from here on used as synonym for volatile RAM). The obvious distinction between NVRAM and DRAM are the persistence properties. Any data written by regular `store` instructions on memory locations mapped to NVRAM is deemed persistent. With more recent implementations of Intels Optane Persistent Memory, also all data residing in CPU caches is also considered persistent [17]. Even though NVRAM is reasonably fast as memory, its access speed is slower than DRAM. In addition, performant NVRAM access is more complex, as the read and write latency differ greatly between random and sequential access [19].

Kernel relation. An NVRAM allocator allows the kernel to access a huge amount of storage with very little latency, when compared to any other persistent storage. In addition the minimal dependency of NVRAM allows the kernel to persist data even during the early startup of the OS and the shutdown procedure. The only requirement for accessing NVRAM is a virtual address mapping.

With such a mapping NVall can also be integrated into the early bootstrapping stage, to make data stored by the kernel available to the boot loader. For example, this can be used for suspending the whole OS to NVRAM (as opposed to DRAM), shutting the system down and resuming computation (as viewed by user space) without interruption at a later time.

Particular challenge. The targeted FreeBSD OS already offers a multitude of performant allocators which are build with awareness of NUMA-induced latency [11] and cache efficiency [6] in mind. However none of them can be used to administer NVRAM, if durability of data is desired.



 DOI: https://doi.org/10.18420/fgbs2023h-02. FGBS '23, September 28–29, 2023, Bamberg, Germany

Any allocator managing NVRAM in a durable way has to store its own state in NVRAM as well, otherwise a reconstruction of said state (e.g. after a power failure) would not be possible. This makes it difficult for NVRAM data structures to embed any kind of locking for handling concurrent access. Any mutex becomes a problem during state reconstruction when restarting after a system failure, since locked mutexes cannot be released without potentially leaving guraded the data in an invalid state. Even though locks can be designed to have their state in NVRAM, such as *generational locks* [5], they seem to be tailored towards the specific use case. Others require their own runtime and compiler support [3].

As the allocators are designed for DRAM with low latency access to data and no consequences in case of sudden interruptions of operation with intermittent memory loss (such as with repeated power outages or severe environmental disturbances), some data structures used for management are very large. This can decrease performance with the slower NVRAM, especially if durability has to be guaranteed. Finally, the allocator has to be capable to repair the state of a previous run, in order to be consistent and durable.

All of these aspects are currently not covered by the existing memory infrastructure. Thus, it is necessary to extend the FreeBSD kernel with an allocator filling the gap.

About this paper. We propose NVall, a crash-resistant, persistent memory capable page frame allocator. Its design goals focus on crash consistency, portability and reduced NVRAM accesses per allocation. These goals lead to a versatile allocator with reasonable execution speed that can be used in other system software beyond the FreeBSD kernel.

2 FUNDAMENTALS AND METHODOLOGY

The following describes our assumptions about the runtime environment followed by the goals we want to achieve.

2.1 System Model

We assume that most systems using NVRAM will use multiple kinds of memory simultaneously—meaning both volatile and non-volatile memory. This matches currently available server hardware which can be equipped with NVRAM in addition to mandatory RAM. With mixed kinds of memory, the difference in characteristics, aside from persistence, get more relevant. They may range from strongly varying performance on parallel workloads, to reduced random access performance due to NVRAM-internal write amplification [19]. With bandwidth and latency of Optane PMEM beeing worse than DRAM [9], it is advisable to split data structures in persistent and volatile portions, using both kinds of memory[1, 4].

We assume an NVRAM memory model that ensures that (1) stores of naturally aligned pointer sized types are atomic, (2) cache lines can be written to NVRAM wholly and atomically, (3) the order of cache line flushes can be enforced, and (4) any data written back from cache to NVRAM is considered persistent. These requirements are met by Intel Optane PMEM. There are various machine instructions for writing cached data back to memory (such as `clflush`, `clwb`) and ordering these (`sfence`) [17]. Additionaly there are non-temporal stores that are written directly to the backing (non-volatile) memory, circumventing the cache[17]. However they rely on vector registers [7], which are typically not used within OS software to avoid storage and restoration of their state [2].

With atomic load/stores and controlled write back of cache lines it is possible to model transactions on NVRAM [10]. This approach ensures that any memory requested from the proposed allocator is either served and reachable by the client, or the request failed and must be repeated.

2.2 Design goals

NVall is supposed to be a lightweight allocator with minimal dependencies, so that it can be ported to other system software as well—this includes other kernels and bootloaders. In addition, the implementation has to be robust. It must be able to serve concurrent requests and always be in a valid state, even if allocations are interrupted by unexpected and abnormal system events. More specifically, no memory must be leaked or marked as free, while being in active use on power failure. The lifetime of all served page frames range from allocation until release by the actor, meaning they stay valid across reboots. Our implementation works with Intel Optane, but is supposed to support any byte-addressable non-volatile memory that satisfies the requirements from Section 2.1. NVall should have a minimal state that must be persistent and valid. All other data structures can be held in volatile RAM for increased performance.

Allocations served by NVall should satisfy multiple constraints, such as the amount of allocated page frames, the selected page granularity (i.e. $4KiB$, $2MiB$, $1GiB$) and alignment. Independent of their granularity, it shall be possible to link all allocations together via pointers written to the memory. Resizing of allocated and already used memory is explicitly not within the scope of NVall. Finally, it should be possible to release memory and reseting NVall, dropping all state in NVRAM.

3 IMPLEMENTATION

The page frame allocator is embedded into FreeBSD and accessible from the kernel. Within the kernel it is integrated into the virtual memory subsystem, which is one of the first subsystems within the kernel to be initialised. Thus, it is available very early during the boot process of the OS.

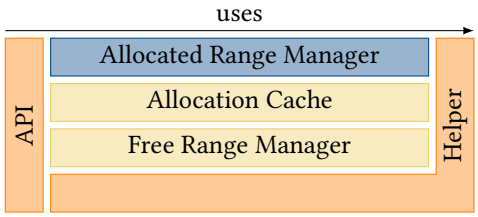


Figure 1: Rough overview of NVall divided into persistent data ( ) and reconstructable data ( ).

NVall is split up in managing specific *root*-objects, identified by name, and allocations which are assigned to either *root*-objects or a non-volatile memory region managed by NVall. As such, any module that uses NVall has to generate a unique name that can be used to (re-)gain access to previous allocations and request more memory. A fundamental concept of NVall's design is the way any allocation result is handled. NVall will not return a plain pointer (as it would be the case with POSIX' `malloc`/`free`), since any such pointer returned to the requesting thread may be lost due to a power failure before the pointer itself can be written to persistent memory. Instead any memory allocation request is modeled as a transaction where the memory reference is written to a logically non-volatile type `nv_ptr`, passed to NVall functions. The transaction either succeeds and stores a non-NULL value in `nv_ptr` or fails. In case of the latter no memory can be leaked.

All allocations have their pointer embedded in non-volatile memory as `nv_ptr`—additionally the very first object in a line of allocations, which serves as *root*-object and can be retrieved by name. Thus, each *root*-object can be the root of an Directed Acyclic Graph (DAG) of allocations.

NVall adopts the notion of *Selective Persistence* for better performance. This concept distinguishes between primary data which must be complete and consistent at all times and any other data which can be rebuilt based on the primary data. The primary data has to reside in NVRAM, while other data structures required for faster access, such as lookup tables and caches, can be stored in volatile RAM [1, 12].

On every boot NVall reserves a configurable amount of NVRAM. Based on the persistent data structures contained therein, NVall performs a self-check for consistency and reconstructs its volatile parts.

3.1 Architectural Overview

As described in Figure 1, NVall offers an API for accessing previously allocated and additional memory. The API functions are built on top of an *Allocated Range Manager*, responsible for managing all previous allocations, as well the transient *Free Range Manager* for fast discovery of free memory. In addition, there is a transient *Allocation Cache* for fast retrieval of currently active objects. The modules use helper functions to achieve persistence. These helper functions are also exposed to the user via the API, to ease the usage of NVRAM.

```
1 int example(void) {
2   struct example {
3     nv_ptr next;
4     unsigned value;
5   } *ptr;
6   nv_ptr parent;
7
8   nv_store_resolve("example", &parent);
9   if (*parent == NULL)
10    nva_alloc(parent, 2, NVA_SIZE_4K, NVA_FLAG_ALIGN);
11  ptr = *(struct example **)parent;
12  if (ptr->next == 0)
13    nva_alloc(&ptr->next, 1, NVA_SIZE_4K, NVA_FLAG_ALIGN);
14  ptr->value++;
15  struct memory_range_nv r = {ptr, ptr+1};
16  nv_persist_range(&r);
17  return ptr->value;
18 }
```

Listing 1: NVall used for a persistent counter value.

An example usage of NVall is given in Listing 1. This example is also the basis for Figure 2. With `nv_store_resolve` it is possible to retrieve a previously prepared *root*-object identified by a key, or to create a new one. The object is stored in a container. However, a retrieved container may not be linked to any memory, since it can either be newly created or the previous run was interrupted before it was filled with memory. A container can be filled with `nva_alloc`. This function requires the amount of memory to be allocated, as well as a persistent pointer to safely store a reference to NVRAM in. If the function succeeds, the persistent pointer `parent` is linked to memory. Upon restart due to an interruption during the allocation, the state can be rolled back so that no memory is lost and the `parent` pointer will be empty (i.e. `NULL`).

3.2 Allocation

NVall offers the function `int nva_alloc(nv_ptr parent, uint64_t pagecnt, int granularity, int flags)` as interface for allocation requests. It can be used to request `pagecnt` number of pages of a predefined granularity. At present, NVall supports requests for $4KiB$, $2MiB$ and $1GiB$ pages, which are the supported (huge) page sizes on *x86_64*. The last parameter `flags` toggles natural alignment of the requested memory, based on the chosen granularity. However, all allocations are at least aligned to the $4KiB$-boundary.

The `parent` parameter, as shown in Listing 1 in Line 10, references a container for storing the result of an allocation. On success, it holds a pointer to the memory, on failure (indicated by the return value) the value is `0`.

Unlike with traditional volatile memory, it is not possible to recover from errors by rebooting. Instead, any data stored in non-volatile memory may lead to persistent bugs. Thus, it

is important to reduce the risk of errors. As a consequence, NVall thoroughly verifies all parameters given to its allocation function. These checks include whether the *parent* object is actually placed in persistent memory and whether is was already managed by NVall. In addition the selected *granularity* and alignment are checked before the allocation starts proper.

On any allocation, the *Free Range Manager* is used to look up ranges of unassigned memory, satisfying all additional requirements. The *Free Range Manager* resides in fast volatile memory and is built from persistent metadata each time NVall is initialised on startup. Whenever a matching range is found, the allocation is stored in a persistent NVall-internal *Allocated Range* table, as depicted in Figure 2. In addition, each allocation is connected to a non-volatile container in which a reference to the allocated memory is stored. This way, it can be assured that all pointers created by `nva_alloc` are also persistent and cannot be lost during a power failure. The container either references a *root*-object, or some memory in a previously allocated object. Details on consistency follow in Section 3.4.

If required by the actor, the allocator accepts an optional flag that requests a zero-initialised memory region. This may be required by some modules, to either simplify working with NVRAM, or for security concerns if the memory is mapped into a user address space and must be cleared of any remnants of kernel data.

Another safeguard makes sure that the containers used for receiving an allocation result are not already in use. Otherwise, memory might be leaked through actors overwriting the only "user-facing" pointer for retrieving the associated memory block.

3.3 *root*-objects

Any kernel module that wants to use NVRAM must generate a unique key that is used by NVall for identifying a *root*-object. These *root*-objects are not distinct from any other memory served by NVall, except they can not be retrieved with a pointer stored to persistent memory, but only by their name. All further allocations which are not *root*-objects must have their allocation result (the identifying pointer) written to non-volatile memory, which is referenced by the original *root* object. Currently, this requirement is checked by NVall at runtime. However, it may also be implemented as an extension to the type system of the programming language to enforce static checking at build time.

The *root*-object can be obtained with `int nv_resolve( const char *name, nv_ptr *parent)`, which uses memory of type `nv_ptr` as location for the pointer to be stored. The parameter `name` serves as a key to identify a single persistent *root*-object which are stored by NVall. It may be used in a

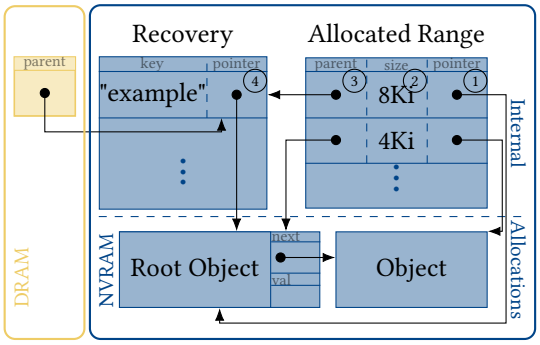


Figure 2: Memory layout example w.r.t. Listing 1.

structured way to build namespaces, similar to paths in file systems. If required, additional access control based on the key and selective mapping into a modules address space is conceivable.

The parameter `parent` serves as a container for a pointer to persistent memory. It can be used for allocation requests with `nva_alloc` and will be filled with a reference to the acquired non-volatile memory. If the dereferenced value of `nv_ptr` is already non-`NULL` prior to any `nv_alloc` call, it was initialised in a previous run of the program and can be used directly.

3.4 Transactions

Based on the system model described in Section 2.1 a three-staged transaction mechanism is implemented. NVall uses `nv_persist(struct memory_range_nv *)` to enforce write-back of all memory within an memory range. It is secured by a `sfence` instruction to ensure that the caches are written to main/persistent memory before the caller proceeds with other operations.

Based on `nv_persist`, NVall implements transactional allocations. In order to maintain consistency at all time, NVall has to be able to cope with any kind of interruption during an allocation. A slightly shortened function for persistence of an allocation is displayed in Listing 2. The code snippet picks up the example in Listing 1 and links the persistence mechanism with metadata displayed in Figure 2. In short, any failure occurring prior to ϟ2 will result in an ongoing allocation to be reverted, while any later interruption can be tolerated and the result is made durable during a recovery phase on the next boot.

The routine for making an allocation durable is separated in three chunks (visually divided by ϟ1 to ϟ3) that must be stored in order. During the first chunk, up to ϟ1 , the start- ① and end-address ② of a newly allocated block is written into the persistent *Allocated Range Table* (as depicted in Figure 2. For a more concise schematic, the end pointer is displayed as size field).

```
1  // ar: Allocated Range, container: nv_ptr
2  ar->range.end = free_range.end;
3  ar->range.start = free_range.start;
4  nv_persist_range(ar);
5  - - - - - - - - - - - - - - - - - - - - - - - - - ϟ 1
6  ar->parent = container;
7  nv_persist_range(ar);
8  - - - - - - - - - - - - - - - - - - - - - - - - - ϟ 2
9  *container = range->start;
10 nv_persist_range(container);
11 - - - - - - - - - - - - - - - - - - - - - - - - - ϟ 3
```

Listing 2: Persistent memory block allocation.

Even after these two pointers have been written persistently into the *Allocated Range Table* by `nv_persist_range` in Line 4, the allocation may still be reverted. Only entries with a valid *parent* entry are considered persistent. Thus, the *parent* is written between ϟ1 and ϟ2. The order (enforced by `nv_persist_range`) is significant, since a valid *parent* field indicates a valid entry.

Finally, the container referenced by the allocation caller is written with a pointer to the requested memory ④. This is necessary to make the allocated memory accessible to the requesting function. However in terms of consistency, this *store* can be done during a recovery phase after a system crash as described next in Section 3.5.

3.5 Recovery

As already indicated in Section 3.4, some states have to be rolled back when resuming execution after an system failure, while others can be completed. Recovery is done during NValls initialisation. This procedure has to handle the transient nature of virtual addresses, that are used in the OS. The FreeBSD kernel is not guaranteed to be mapped at the same virtual addresses in subsequent startups. Constantly changing virtual addresses constitute a problem when using non-volatile memory, since it makes any pointer stored therein possibly invalid. Thus, NVall has to make sure that any virtual address used for referencing memory remains stable across multiple restarts. Therefore, we implemented a direct mapping of physical addresses to their virtual counter part, which results in NVall being indifferent about kernel relocations.

3.6 Concurrent Access

When working with NVRAM, concurrent access comes in multiple flavors. First we have to care about concurrent accesses on shared data in NVRAM. On the other hand there is the *broken time machine* problem, where any sequential code running in NVRAM may be confronted with its leftover, inconsistent state of a previous execution [16].

The first problem is handled with locks, protecting all metadata from inconsistency due to concurrent access. However these locks are only part of the volatile data structures, as their state can only be valid during the current execution. If a lock were to be placed in NVRAM, it would suffer from the time machine problem and may already be in a locked state, when the system restarts. The persistent data structures are protected against parallel access by the volatile locks. Protection against power failures, for example, is based on the transactional behaviour described in Section 3.4.

3.7 Restrictions

NVall does only distribute memory from NVRAM. The recipient of an allocation has to take care to use NVRAM in a consistent, crash resistant manner: He can do this by resorting to transactions at his own level of abstraction. Even though NVall does not give any guarantees regarding consistent NVRAM usage, its functions for persistence can be used by other modules as well to implement transaction-like behaviour.

Its dependencies are an allocator for its volatile data structures, a list of memory ranges covered by NVRAM and a mapping function to set up the identity-mapping.

4 PERFORMANCE CHARACTERISATION

The performance measures were performed on a Dell PowerEdge R650, equipped with two Intel® Xeon® Gold 6330 processors. Each processor has 28 cores and 56 threads, respectively. Their base frequency is 2.0 GHz, even though they can boost to up to 3.10 GHz. The memory is a mix of eight 32 GB DDR4 RDIMMs, plus eight DIMMs of Optane™ Persistent Memory 200 with a capacity of 128 GB each. All tests are performed on a single core during the systems startup with disabled interrupts, to avoid any interference from userspace and devices.

In our tests we compare variations of NVall to determine the additional cost to achieve durability and used different page granularities. Furthermore, we compare the results with the FreeBSD pageframe allocator. All tests are done on a modified FreeBSD 13.1 kernel, that can use NVRAM as its only main memory [15], so that the kernels *.text* and *.data*, as well as all userspace processes reside in NVRAM. This ensures comparability of the FreeBSD allocator, that would otherwise work on faster DRAM. The configuration for NVall uses both NVRAM and DRAM for testing the separation of internal data structures. Our benchmarking is set up according to Intels advisory [13]. As clock we used the processor internal time stamp counter, which is guaranteed to increment at a constant rate on the given CPU [8].

The results in Figure 3 show NVall (■) and NVall with its persistence based on cache line write back and store ordering disabled (■). We compare NVall in different configurations with FreeBSDs `vm_page_alloc_noobj_contig`

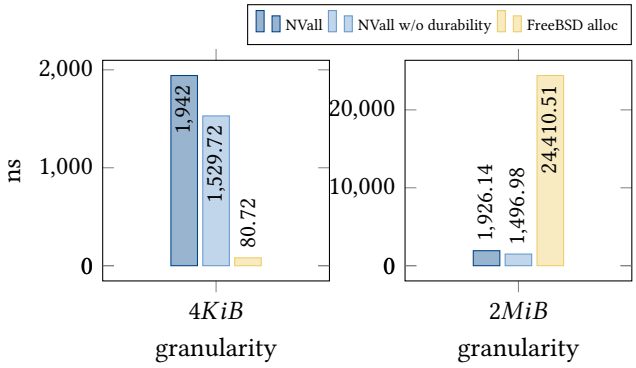


Figure 3: Average allocation performance, 1024 runs

allocator ( ). The different configurations are NVall with all guarantees regarding durability and a version without cache line write back.

The most obvious insight is that the FreeBSD allocator is very well optimised for $4KiB$ pages. However, as soon as the granularity increases, its performance starts to deteriorate. NVall in contrast does not have such a huge fluctuation in behaviour. Instead the time remains fairly similar between the chosen granularities. With NVRAMs great capacity, it is probable that huge pages will be used more frequently, making the huge-page behaviour more important in future.

The greatest performance impact on NVall allocations are associated with the additional safeguards we implemented. With every allocations we check whether the storage for the allocation result resides in NVall-managed NVRAM. In addition, we also make sure that no persistent pointer is overwritten by an allocation. These checks require persistent hash table lookups and, thus, are the main contributor to NVall runtime overhead.

When comparing the different NVall configurations, we can observe the cost of durability due to explicit memory ordering and cache write back. Since one of the goals was to reduce the number of stores to NVRAM for performance, the difference is rather small, even though still measureable.

5 DISCUSSION

With NVall, we have developed an allocator for OS kernels to manage persistent state across reboots and crashes. This can be used for implementing state recovery of a whole system, or for storing kernel data when no other storage media is present—either due to a fault, or due to inherent constraints, such as deactivation of modules during shutdown. The performance, especially for $4KiB$-allocations, can be further improved by using more sophisticated volatile data structures. The comparison of NVall with and without explicit write back shows that the goal of seldom NVRAM access for durability of metadata is met.

6 RELATED WORK

There has been previous work on allocators for NVRAM, such as *Makalu* [1] and NV-Heaps [5]. However, they both require an garbage collector to avoid dangling pointers and leaked memory. *Makalu* has to search for pointers to NVRAM in all memory referenced from specific root objects to avoid memory leaks after recovery from a crash. NV-Heaps, on the other hand, relies on reference counting, which in turn requires programming language support (e.g. operator overloading) that is not available in all programming environments. This is especially true for OS kernel development with *C* as the most prominent language.

Recent work also wants to make OS kernels to be more aware of NVRAM pecularities and proposes *LLFree*, a page-frame allocator with a design that requires neiter locking, nor logging [18]. As a result, there are very few write accesses per allocation on NVRAM, leading to good performance. However, in contrast to NVall, the interface between actor and allocator is prone to leak memory on crashes: “While a crash during a de/allocation would never result in an unrecoverable allocator state, it could lead to a lost frame. This would happen if in an allocation the bit has already been set, but the frame has not yet reached the caller, or if the deallocation has been invoked but not yet cleared the bit.”

7 CONCLUSION

NVRAM can have capacities in the TiB range these days. If memory allocators for NVRAM are not able to survive common system crashes either large amounts of information are lost or still available memory might simply be forgotten. Thus, an allocator for NVRAM needs to operate in a strictly transactional manner and keep its internal state consistent across system crashes and reboots. NVall can tolerate power outages and survive all system crashes that are not caused by or related to memory corruption of the allocator’s metadata. NVall utilises DRAM whenever possible to speed-up normal operation and accesses NVRAM only when necessary to ensure transactional semantics.

AVAILABILITY

The source code of NVall is available under open source license as a patchset: https://doi.org/10.5281/zenodo.8364439

ACKNOWLEDGMENTS

The author order corresponds to the SDC (*sequence determines credit)* model: The first author gets full credit, the second author half, the third author a third and the fourth author a quarter. This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 501993201.

REFERENCES

[1] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 677–694, New York, NY, USA, 2016. Association for Computing Machinery.

[2] Daniel Pierre Bovet, Marco Cassetti, and Andy Oram. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., USA, 2000.

[3] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. Association for Computing Machinery.

[4] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1077–1091, New York, NY, USA, 2020. Association for Computing Machinery.

[5] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. Association for Computing Machinery.

[6] Matthew Dillon. Design elements of the FreeBSD VM system. https://cgit.freebsd.org/doc/tree/documentation/content/en/articles/vm-design/_index.adoc?id=4a95b5409c05615d49b91f155f2fc14bcaa9ba56, 2023.

[7] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual: Instruction Set Reference, A-Z, Volume 2 (2A, 2B, 2C & 2D)*., April 2022.

[8] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual: System Programming Guide, Volume 3 (3A, 3B, 3C & 3D)*., April 2022.

[9] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the Intel Optane DC persistent memory module, 2019.

[10] Marcel Köppen, Jana Traue, Christoph Borchert, Jörg Nolte, and Olaf Spinczyk. Cache-line transactions: Building blocks for persistent kernel data structures enabled by AspectC++. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, PLOS'19, pages 38–44, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Marshall Kirk McKusick, George Neville-Neil, and Robert N.M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, 2nd edition, 2014.

[12] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 371–386, 2016.

[13] Gabriele Paoloni. *White paper: How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures*. Intel, September 2010.

[14] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. System evaluation of the Intel Optane byte-addressable NVM. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, pages 304–315, New York, NY, USA, 2019. Association for Computing Machinery.

[15] Jonas Rabenstein, Dustin Nguyen, Oliver Giersch, Christian Eichler, Timo Hönig, Jörg Nolte, and Wolfgang Schröder-Preikschat. On the performance of NVRAM-based operating systems: A case study with Linux and FreeBSD. Technical Report CS-2023-01, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Department Informatik, March 2023.

[16] Benjamin Ransford and Brandon Lucia. Nonvolatile memory is a broken time machine. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, New York, NY, USA, 2014. Association for Computing Machinery.

[17] Steve Scargall. *Programming Persistent Memory: A Comprehensive Guide for Developers*. Apress Media LLC, 2020.

[18] Lars Wrenger, Florian Rommel, Alexander Halbuer, Christian Dietrich, and Daniel Lohmann. LLFree: Scalable and Optionally-Persistent Page-Frame allocation. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 897–914, Boston, MA, July 2023. USENIX Association.

[19] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.