An NVM Performance Study Towards Whole System Persistence on Server Platforms

Till Miemietz
Barkhausen Institut
Dresden, Germany

Viktor Reusch
TU Dresden
Dresden, Germany

Michael Roitzsch
Hermann Härtig
Barkhausen Institut
Dresden, Germany

Abstract

Whole system persistence (WSP) is a concept for retaining the computational state of a system even in case of a power failure. In the context of server systems, WSP could render it possible to quickly power on and off machines that only need to be used occasionally, thus saving energy. This paper takes on this idea and discusses multiple approaches for implementing WSP on such machines. Our evaluation shows that after starting a system, an NVM-based version of WSP can achieve tail latency improvements of up to 93% compared to booting a system and loading data from an SSD. At the same time, WSP is able to provide suspend and resume times in the order of tens of milliseconds.

CCS Concepts: **• Software and its engineering → Memory management**; **• Hardware → Non-volatile memory**.

ACM Reference Format:
Till Miemietz, Viktor Reusch, Michael Roitzsch, and Hermann Härtig. 2023. An NVM Performance Study Towards Whole System Persistence on Server Platforms. In *1st Workshop on Disruptive Memory Systems (DIMES '23), October 23, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3609308.3625269

1 Introduction

Whole system persistence (WSP) [21] is a concept that allows to cut the power supply of a computer without having the system lose any computational state. With WSP, the system hardware has enough capacitance reserves to write all volatile state back to persistent media when the power goes away. Thus, the traditional procedure of shutting down and booting a system can be shortened to faster suspend and resume operations, ideally enabling users to switch computers on and off just as a light bulb. In such a world, a loss of power would have no harmful effects on a system since it would be equal to a "shutdown" operation triggered by pulling the power plug of a server. Furthermore, WSP not only shields computational state from sudden changes in the power supply, but also preserves ephemeral state that is normally discarded during a reboot.

Due to these properties, WSP could enable interesting power saving options for applications that serve requests infrequently but still need to maintain low response times. For instance, in a data center, this could be a database server that is specialized for answering queries to seldom accessed records. On the one hand, such systems should maintain in-memory caches to achieve a short response time. On the other hand, powering off such systems during idle phases to save energy is of both economic and environmental benefit. However, in a traditional system, a boot cycle of a *physical* machine takes rather long and loses any memory contents. With respect to the requirement of short response times this creates a dilemma that an approach like WSP might solve.

Implementing WSP requires only simple hardware and software modifications. As already pointed out by Narayanan et al. [21], the hardware changes mostly consist of a power event notification mechanism and a suitable provisioning of power reserves, possibly in the form of additional capacitors in the system. Additionally, the software modifications needed to implement WSP are modest, since much of the standard procedures for suspending and resuming a system can be reused.

Today, there exist three candidate technologies for implementing WSP: Using a conventional system design with volatile DRAM and fast SSDs, storing data on remote nodes using the compute express link (CXL) [4], and deploying non-volatile main memory (NVM). In general, the decision between a conventional system and a CXL- or NVM-enabled one is mostly a tradeoff between runtime performance and suspend/resume latency as well as implementation simplicity.

To this extent, we take a look into the time it takes to suspend and resume a system that runs on NVM only and compare these numbers to a system deploying DRAM and SSDs. First, this should answer the question of whether implementing WSP using a traditional system architecture is a viable choice. This is particularly important in the light of Intel having discontinued Optane NVM.

Second, since NVM has worse performance than DRAM in most cases [26], we will enrich the measurements of plain suspend and resume times with an analysis of the runtime costs that are entangled with relying on NVM as main memory.

DIMES '23, October 23, 2023, Koblenz, Germany
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0300-3/23/10.
https://doi.org/10.1145/3609308.3625269

These measurements are intended to investigate whether the performance benefit of resuming a system with warm caches by means of WSP is significant compared to rebooting a server.

Lastly, we investigate whether the design of the operating system can positively influence the ability to implement WSP. We find that compartmentalized systems, such as microkernels, provide better safety for systems running on NVM and additionally alleviate WSP-specific issues such as a restart of repeatedly failing system components.

In conclusion, the paper makes the following contributions:

- A prototype software implementation of WSP on L4Re, a state-of-the-art microkernel
- A detailed evaluation of the tradeoff between runtime performance and suspend / resume latency when using NVM compared to a system deploying DRAM and SSDs
- A discussion of OS design guidelines for supporting the implementation of WSP

2 Background

Starting with an overview of techniques for suspending and resuming an operating system, this section discusses the idea of WSP and candidate technologies for implementing it in detail.

2.1 Suspending and Resuming Operating Systems

Hibernate. Hibernation [2], also called suspend-to-disk, is a method for temporarily halting a system by retaining the volatile system state on disk. On hibernation, the CPU registers and memory contents are bundled into an image and subsequently written to a non-volatile device. Upon next boot, the operating system then tries to discover and load such an image instead of carrying out a standard booting procedure. An OS instance restored from a hibernation image can transparently continue execution without restarting system services or applications.

Suspend-to-RAM. During a suspend-to-RAM, CPUs and other devices are shut down but the main memory is still being powered. This allows to quickly suspend execution as only the CPU state consisting of registers and caches needs to be written to memory. The amount of moved data is small compared to hibernation and only needs to reach main memory. A suspend-to-RAM requires hardware and firmware support, such as the *S3* sleeping state of ACPI [2, p. 767]. On resume, the firmware hands control back to the kernel, which is still present in memory. However, when the power supply is cut, all volatile state such as the DRAM content is lost, too.

2.2 Whole System Persistence

Whole System Persistence (WSP) [21] shields the computational state of a system against an unexpected loss of power by constantly monitoring its power supply, triggering a write-back of volatile state to persistent devices in case of a power outage. Unlike other approaches [7, 16, 19, 25], WSP does not add any runtime overhead to ensure the integrity of a system in case of a power loss. After restarting, a WSP-enabled system neither performs undo or redo operations on persistent state since the system continues execution at the same spot as it was in before the power was cut. This makes the adoption of WSP easy as application code does not need to be modified.

In order to implement WSP, the system hardware needs to be able to quickly recognize a loss of input power. When there is a power failure, the hardware sends an interrupt to the CPU that in turn triggers the suspend routine of the operating system. In order for WSP to work properly, the *residual energy window*, i.e., the time span that the system can still operate after losing power, needs to be sufficiently large to allow a write-back of all volatile state.

On the software side, the modifications required for WSP comprise a handler for the power-fail interrupt, that suspends the system. Upon restoring power, a WSP-enabled system uses dedicated code paths to recognize that it needs to restore a previous system instance. Furthermore, device drivers have to be extended in a way that they can resume the state of peripheral hardware upon a resume operation [12, 14].

2.3 Non-Volatile Memory

Non-volatile memories (NVM) constitute a class of main memory devices that are capable of retaining their information even in an unpowered state. Unlike traditional persistent media, NVM exposes a memory interface and can thus be directly accessed by the CPU. There are various technologies for implementing NVM, such as phase-change memory (PCM) [1] or spin-transfer torque magnetoresistive RAM (STT-MRAM) [27]. While there are types of NVM that have performance characteristics close to that of DRAM (e.g., STT-MRAM), there is no technology that combines the performance and capacity of DRAM with the feature of persistence. So far, the only type of NVM that can to be used in commodity systems is Intel's PCM-based Optane [1].

2.4 CXL

The compute express link (CXL) [4] is a collection of hardware protocols built on top of PCI Express. CXL offers cache coherence across multiple machines as well as between a CPU and peripheral devices. In contrast to current protocols, CXL renders it possible to access device memory as if it was a part of the local main memory.

By means of CXL, one could implement a kind of NVM by moving critical data to remote machines. In this case, if one machine loses power, the data is not lost even if it is stored in volatile memory. However, in order to implement true WSP, this assumes that the machines are located in different fault domains with respect to the power supply.

3 Design Considerations for WSP

For the rest of this paper, we assume that the implementation of WSP is accomplished without the help of external devices such as uninterruptible power supplies in order to reduce the complexity and cost of possible solutions. Hence, the feasibility of WSP in general boils down to two questions, namely: (1) How fast can the system carry out a suspend and resume operation? and (2) Does the use of a particular system layout affect the runtime performance of applications? The first question indirectly also covers the question of how much residual energy needs to be kept in a system, and thus determines whether a particular implementation of WSP is viable from a technical and economical point of view.

3.1 WSP Using NVM

When running the entire system on NVM, the implementation of WSP is simple: As all data except that contained in CPU caches is directly persisted, only the hardware extensions described in section 2.2 (power monitoring and providing enough residual energy) need to be applied. Inside the OS, the suspend and resume operations can be implemented just as when running on DRAM.

However, Optane, the only available implementation of NVM for commodity systems, has a significantly worse performance than DRAM, both in terms of latency and bandwidth [22, 26]. Several studies showed that this negatively impacts the performance of applications that run on NVM. As Koutsoukos et al. pointed out, the performance of a system running on Optane can even drop to the level of flash-based SSDs since the CPU stalls on rather slow accesses to NVM, reducing opportunities to carry out computations as it would be possible when using an asynchronous interface for accessing persistent media [20].

3.2 Working on Remote Memory

With the rise of advanced interconnects like CXL, a possible replacement of NVM could be the use of remote memory. Since CXL offers a memory interface to access memory of remote machines, the advantages with respect to the implementation simplicity of WSP would be the same as for NVM. The main difference between implementing WSP with CXL and NVM is that the CXL variant can not be realized using only a single machine.

Unfortunately, at the time of writing this paper, a publicly available solution for CXL.mem devices does not exist yet. Hence, we do not cover this way of implementing WSP in the evaluation. However, preliminary studies on CXL fabrics indicate that the performance of accessing a remote memory target via CXL might be close to or even slightly better than that of Optane NVM [15].

3.3 WSP on Traditional Systems

When using a traditional system design, suspending a system is not only about writing back CPU caches and registers, but also encompasses the write-back of volatile main memory contents to a persistent device. Writing back the whole DRAM content slows down the suspend procedure significantly. For instance, even when considering the peak write throughput of a modern SSD (e.g. 5 GiB/s for a Kioxia CM6-RI), storing an entire DRAM image of 500 GiB would take a hundred seconds. This is much more than the residual energy window found in standard servers [3] as well as the energy window needed for suspending to NVM [21].

In contrast, most applications only modify a fraction of the RAM that they allocated. For instance with YCSB [8], a popular cloud service benchmark, most of the predefined workloads are read-heavy. Thus, the amount of dirty data that needs to be written back to secondary storage during a suspend operation can be reduced. Moreover, a system could use compression algorithms to shrink the RAM image, as done in OSes like Linux.

3.4 On the Value of Compartmentalization for WSP

In a compartmentalized system design, such as a microkernel, system components such as drivers are strongly separated from each other, interacting using a client-server model. As a part of this philosophy, the code running in privileged CPU modes is minimized.

Compartmentalization is beneficial for systems running on NVM. Unlike traditional systems that can recover from memory corruptions due to software bugs by rebooting, corruptions of NVM can neither be detected nor fixed easily. Hence, a strong separation of system components can increase the safety of a system by confining stray writes and other application bugs into rather small domains, instead of sharing a huge memory space such as the kernel in monolithic systems.

Furthermore, a compartmentalized system design often comes with good modularity, i.e., a clear separation of independent subsystems of an OS. This modularity alleviates restarting of parts of the system in case of an error [10] thus being particularly valuable for WSP in order to get out of repeated failures due to bugs in one subsystem. Additionally, as pointed out by Tsalapatis et al., an aggressive write back of dirty RAM contents is crucial for swiftly saving the state of a system to persistent media when running on a traditional system architecture [25]. The use of several smaller and separated system components facilitates the identification of memory areas that each component has to write back to save an image of itself.

4 Implementation

In order to evaluate the idea of using a compartmentalized system for WSP on NVM, we chose to add the necessary software mechanisms to L4Re [17], a microkernel-based operating system.

As a first step to add support for WSP to L4Re, we constructed a mechanism for discovering and managing different types of main memory. Amongst other things, this involved an extension of the parsing procedure of the ACPI tables, subsequently passing the resulting layout information through the multi-stage boot process of L4Re, as well as providing different page frame allocators for each type of memory. As a result, an application can choose explicitly which type of memory to allocate. Moreover, we added means for a system administrator to transparently override the default memory type that applications are served from. This enables a movement of unmodified binaries from DRAM to NVM, one key property for conducting WSP.

The next step was to move all components of L4Re to NVM. For the suspend and resume routines, we could mostly rely on an existing suspend-to-RAM mechanism of L4Re. However, we had to extend the resume operation to use a trampoline code page for switching to the actual kernel that resides in NVM. This became necessary due the startup procedure of an x86 CPU[1] that does not allow to begin execution in NVM when powering on the system.

Lastly, we replaced the standard `malloc` implementation of L4Re with a customized version of `jemalloc` [5]. This was aimed at improving the performance of L4Re compared to commodity OSes like Linux.

5 Evaluation

During the evaluation, we aim at answering the following questions:

- How fast can WSP be implemented theoretically on modern machines using different approaches for persisting volatile system state?
- What is the performance tradeoff between running an application on DRAM vs. NVM?
- Does the compartmentalized design proposed in this paper incur conceptual performance disadvantages compared to a standard OS?

Note that we only implemented WSP for L4Re running entirely on NVM. For gaining insight into the hypothetical performance of a WSP implementation on a traditional system (DRAM + SSD), we instead used microbenchmarks running on L4Re's and Linux' NVMe stack.

5.1 System Setup

We conducted our measurements on a dual-socket server platform using two Intel Xeon Platinum 8358 CPUs with a clock rate of 2.6 GHz. The CPU features 1.25 MiB of L2 cache for each core as well as an shared L3 cache with a capacity of 48 MiB per socket. For all benchmarks, we disabled hyperthreading (SMT) as well as temporary overclocking (TurboBoost) and set the CPU's pstate to the performance configuration.

Furthermore, the machine is equipped with 500 GB of DRAM as well as one TB of Optane DIMMs (Gen. 2). As a secondary storage, the benchmarks use a Kioxia CM6-RI SSD with a size of 3.84 TB. The SSD is attached to the host system using NVMe Version 1.4 over PCIe 4.0 x4. For all Linux-related benchmarks presented in the following, we deployed kernel version 6.1.27.

[1]Due to the type of available NVM we were forced to use this architecture.

Note that the test system was not capable of sending power-fail interrupts. Instead, we simulated a loss of power supply by executing a dedicated system call. This system call then triggered the L4Re kernel to execute its suspend routine.

Similarly, we report the resume latency as the time between the OS being kicked off by L4Re's bootloader and the OS continuing to execute a system image. In practice, the startup of a server system takes a couple of minutes due to hardware self-tests. However, this is not a conceptual limitation since such checks could be omitted on a dedicated WSP platform.

5.2 Suspend and Resume Performance

Figure 1 shows the latency for suspending an L4Re system to either RAM or NVM as a function of the amount of dirty cache[2], using the `WBNOINVD` instruction for writing back the CPU caches. We chose to restrict the system to a single CPU socket to avoid NUMA effects. For each measurement, we recorded the latency when using a single core (no SMP) as well as when using all cores of the socket (SMP).

In accordance with Narayanan et al. [21], the latency for a suspend-to-RAM operation is independent of the amount of modified cache. When enabling SMP, the latency for a cache write back increases because of the IPIs used for notifying all CPU cores of the suspend operation. Furthermore, while the write-back of core-local L1 and L2 caches runs in parallel on the application processors, the boot CPU (i.e., the core that runs first when the system starts) has to wait for every core to finish the write-back before flushing its own cache and shutting down the system. This wait time is not present in the non-SMP configuration.

However, in contrast to earlier predictions [21], the suspend latency on NVM is higher than on DRAM, increasing linearly with the amount of dirty cache. We believe that this is because a cache flush to NVM waits for the corresponding store operations to complete instead of just forwarding them to the memory controller.

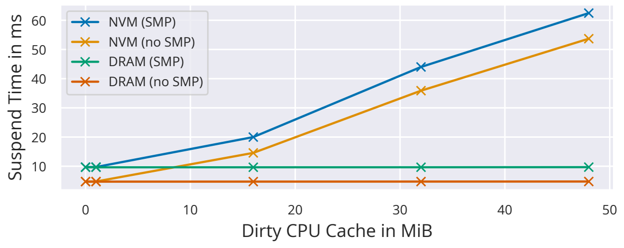

Figure 1. Time needed for suspending L4Re to memory as a function of the amount of dirty cache.

Figures 2 and 3 show a detailed latency breakdown of the suspend and resume operations of an idle L4Re instance that

[2]Since the procedure for a suspend or resume on memory does not depend on the OS architecture, we expect a standard OS like Linux to show similar numbers.

runs entirely on NVM (i.e., for one flavor of WSP). For the suspend operation, the preparation of the kernel, the notification of the application processors, and the write-back of the application processors' caches take less than a millisecond when SMP is disabled (*suspend_prep*). When enabling SMP, the cost of this phase is much higher, again being caused by cross-core communication overhead and the time that the boot CPU has to wait for the other cores to write back their caches. The remainder of the suspend time is consumed by the write-back of the boot CPU's cache (*suspend_wb*). Finally, the latency of marking the main memory contents as a valid image and shutting down the CPU (*suspend_finish*) accounts for roughly 150 µs in both SMP and non-SMP settings.

As depicted in Figure 3, the resume operation is much faster than a suspend, since no data has to be written back to NVM. Here, the time it takes to start up L4Re's second-level bootloader (*resume_trampoline*) as well as to resume the kernel (*resume_kernel*) account for only a couple of microseconds. Then, the startup of the boot CPU (*resume_boot*) takes around 250 µs. Finally, *resume_finish* denotes the delay until all application processors become online again. We believe that adding device drivers to the resume path will not slow down the operation by orders of magnitude since typical operations for initializing devices like NVMe-attached SSDs or RDMA NICs only take a couple of milliseconds [23].

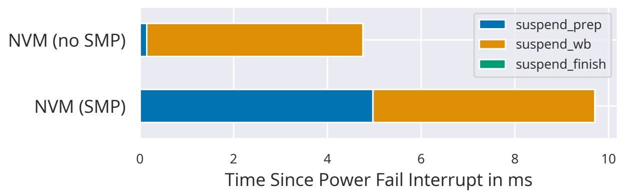

Figure 2. Latency breakdown for a suspend-to-NVM operation on L4Re (minimal cache usage).

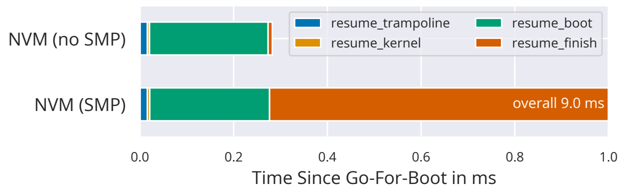

Figure 3. Latency breakdown for a resume-from-NVM operation on L4Re.

Latency for writing RAM images to an SSD. We used Linux for evaluating the performance of a standard suspend-to-disk procedure of an idle system. This operation took 48.5 s (11.1 s to allocate the RAM image, 16.8 s to copy the image in RAM, and 20.6 s to write the image to the SSD), creating an image of roughly 10 GiB in size. A resume-from-disk took 70.1 s overall (17.9 s to find the image on disk, 47 s to read the image, and 5.2 s to continue execution). When suspending a system with large applications that touch more memory, we expect these numbers to grow significantly. Thus, we conclude that using even a modern SSD as a persistent medium for WSP appears to be impractical since the suspend and resume latency is increased by orders of magnitude compared to a system running on NVM, exceeding the available residual energy.

5.3 Application Performance

As an application benchmark we used the built-in `db_bench` latency benchmark of LevelDB [6] to evaluate whether WSP provides performance benefits when resuming a system. We also compared the performance of LevelDB running on a compartmentalized system (here L4Re) to that observed on a standard Linux platform.

Figure 4 shows the resulting histograms of the LevelDB benchmark, with each run performing ten million accesses on a database with one billion entries. The columns group the single benchmark runs by the configuration of the OS (L4Re / Linux) and the benchmark type (random read / sequential read) used. The rows further distinguish the benchmarks by the memory configuration used for each run. *DRAM Uncached* denotes benchmark runs on a newly started LevelDB instance residing in DRAM. Hence, all records that the benchmark reads for the first time have to be fetched from the SSD. In contrast, *NVM Cached* denotes benchmarks on an in-NVM instance of LevelDB with all database contents present in NVM. Thus, no disk operations need to be done in this configuration. Note that for L4Re, *NVM Cached* means that all parts of the system run in NVM (i.e., WSP configuration), whereas on Linux, due to implementation constraints, this is only true for the contents of LevelDB which are mapped using an `fsdax` file system. *DRAM Cached* runs under the same preconditions as *DRAM Uncached*, with the exceptions that the database has all of its contents present in memory. This setup is the best-case configuration from a performance perspective.

In order to evaluate the benefit of deploying WSP for a database scenario, the comparison between the *DRAM Uncached* column (equals a newly booted system) and the *NVM Cached* column (represents a system resumed from NVM) is of particular interest. When suspending and resuming a system that runs entirely on NVM, all database contents and caches are still present while a traditional system would have to load them from secondary storage again.

For a sequential workload pattern on Linux, the average latency of the *NVM Cached* setting was 30% lower than that of *DRAM Uncached* (0.18 µs vs. 0.26 µs). On L4Re *NVM Cached* was even 50% faster (0.22 µs vs. 0.41 µs), probably due to less optimized prefetching routines. During the random read benchmark the performance advantage of *NVM Cached* over *DRAM Uncached* increased to 50% on Linux (8.35 µs vs. 14.96 µs) and

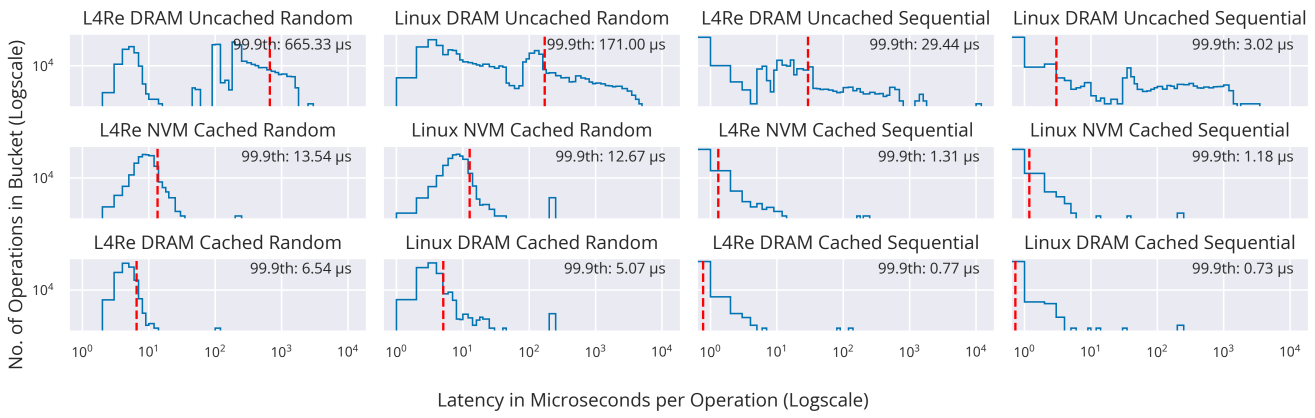

Figure 4. Latency histograms for various configurations of LevelDB. The dashed red lines correspond to the 99.9th percentile specified in the upper righthand corner of each subplot.

95% on L4Re (9.09 µs vs. 133.83 µs), respectively. We assume that the absolute performance gap between L4Re and Linux is not a conceptual problem of microkernel-based systems but rather caused by the fact that L4Re is currently geared towards embedded systems and hence lacks many of the optimizations that Linux has, e.g. on the NVMe I/O path.

When looking at the tail latency (99.9th percentile), the benefit of WSP for running occasionally used systems becomes even more pronounced: On Linux the *NVM Cached* runs had a 99.9th percentile that was 60% lower than that of *DRAM Uncached* for sequential reads. For random reads, this advantage increased to 93%. We attribute the larger latency gap between *NVM Cached* and *DRAM Uncached* for random reads mostly to Optane's internal organization that is better in hiding the PCM latency when facing sequential access patterns.

On L4Re, the differences in tail latency between *DRAM Uncached* and *NVM Cached* were even higher than on Linux. While the 99.9th percentile was similar to Linux concerning the cached runs from NVM, the tail latency for the *DRAM Uncached* runs were 50 times higher than those of *NVM Cached* for both sequential as well as random access patterns. Again, we attribute this difference to an unoptimized implementation L4Re's block I/O path.

6 Related Work

The idea of WSP has first been described by Narayanan et al. [21]. Capri [18] extended this idea, presenting an approach that provides WSP to unmodified applications through a combination of compiler support and architectural extensions. Neverlast [14] demonstrated the feasibility of WSP for embedded systems, also covering device states.

Beside using WSP for creating consistent and persistent images of a system's state, there exists a variety of approaches such as checkpointing [13, 19, 25], transactional programming [7, 16], or CPU extensions [11, 24]. Particularly recent checkpointing approaches are fast in saving an application's state to disk. For instance Aurora [25] can create a snapshot of a process with a memory footprint of 500 MiB in 97 ms. However, in contrast to WSP, these approaches introduce complex programming models, require the modification of application code, and often cover only parts of a system. Moreover, the use of these frameworks comes with additional runtime costs.

Non-volatile processors (NVPs) [9], which are mainly intended for use in energy-harvesting devices, aim at creating a persistent system by saving computational state in the CPU itself. This requires the processor to use non-volatile building blocks for constructing registers and caches. Unfortunately, most NVP approaches suffer from wear effects and involve extensive changes to the CPU.

7 Conclusion

In this paper, we discussed the feasibility and advantages of implementing WSP on server platforms. To this end, we took a look at different candidate technologies and proposed the use of a compartmentalized OS architecture to support the goal of WSP. During the evaluation, we showed that memory-based technologies like NVM are a key factor for enabling WSP. Furthermore, we demonstrated that WSP can reduce the tail latency of a database benchmark running on a seldom used server by up to 93% on Linux and up to 98% on L4Re, compared to booting such a system each time it is needed. In future work, we plan to evaluate the practicability of CXL for WSP as well as to further improve the performance of the microkernel approach used in this paper.

Acknowledgements

This research is funded by the German Research Council DFG through grant 502457159 (FOSSIL). Furthermore, we would like to thank Adam Lackorzynski and Jan Klötzke for supporting us during the implementation of WSP on L4Re.

References

[1] Intel optane persistent memory product brief. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html, 2019. (Accessed on August 18, 2023).

[2] Advanced configuration and power interface (acpi) specification, release 6.5. https://uefi.org/sites/default/files/resources/ACPI_Spec_6_5_Aug29.pdf, 2022. (Accessed on August 18, 2023).

[3] Atx version 3.0 multi rail desktop platform power supply, revision 2.01. https://cdrdv2-public.intel.com/336521/336521_Rev2p01.pdf, 2023. (Accessed on August 18, 2023).

[4] Compute express link. https://www.computeexpresslink.org/, 2023. (Accessed on August 18, 2023).

[5] jemalloc. https://github.com/jemalloc/jemalloc, 2023. (Accessed on August 18, 2023).

[6] leveldb. https://github.com/google/leveldb, 2023. (Accessed on August 18, 2023).

[7] Persistent memory development kit. https://pmem.io/pmdk/, 2023. (Accessed on February 13, 2023).

[8] Yahoo! cloud serving benchmark. https://github.com/brianfrankcooper/YCSB, 2023. (Accessed on August 18, 2023).

[9] Steven C. Bartling, Sudhanshu Khanna, Michael P. Clinton, Scott R. Summerfelt, John A. Rodriguez, and Hugh P. McAdams. An 8mhz 75µa/mhz zero-leakage non-volatile logic-based cortex-m0 mcu soc exhibiting 100% digital state retention at vdd=0v with <400ns wakeup and sleep transitions. In *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 432–433, 2013.

[10] Koustubha Bhat, Dirk Vogt, Erik van der Kouwe, Ben Gras, Lionel Sambuc, Andrew S. Tanenbaum, Herbert Bos, and Cristiano Giuffrida. OSIRIS: efficient and consistent recovery of compartmentalized operating systems. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*, pages 25–36. IEEE Computer Society, 2016.

[11] Abhishek Bhattacharyya, Abhijith Somashekhar, and Joshua San Miguel. Nvmr: non-volatile memory renaming for intermittent computing. In Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang, editors, *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 1–13. ACM, 2022.

[12] Adriano Branco, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. Intermittent asynchronous peripheral operations. In Raghu K. Ganti, Xiaofan Fred Jiang, Gian Pietro Picco, and Xia Zhou, editors, *Proceedings of the 17th Conference on Embedded Networked Sensor Systems, SenSys 2019, New York, NY, USA, November 10-13, 2019*, pages 55–67. ACM, 2019.

[13] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. Fine-grain checkpointing with in-cache-line logging. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 441–454. ACM, 2019.

[14] Christian Eichler, Henriette Hofmeier, Stefan Reif, Timo Hönig, Jörg Nolte, and Wolfgang Schröder-Preikschat. Neverlast: an nvm-centric operating system for persistent edge systems. In Haryadi S. Gunawi and Xiaosong Ma, editors, *APSys '21: 12th ACM SIGOPS Asia-Pacific Workshop on Systems, Hong Kong, China, August 24-25, 2021*, pages 146–153. ACM, 2021.

[15] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, high-performance memory disaggregation with directcxl. In Jiri Schindler and Noa Zilberman, editors, *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 287–294. USENIX Association, 2022.

[16] Swapnil Haria, Mark D. Hill, and Michael M. Swift. MOD: minimally ordered durable datastructures for persistent memory. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 775–788. ACM, 2020.

[17] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μkernel-based systems. In Michel Banâtre, Henry M. Levy, and William M. Waite, editors, *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997*, pages 66–77. ACM, 1997.

[18] Jungi Jeong, Jianping Zeng, and Changhee Jung. Capri: Compiler and architecture support for whole-system persistence. In Jon B. Weissman, Abhishek Chandra, Ada Gavrilovska, and Devesh Tiwari, editors, *HPDC '22: The 31st International Symposium on High-Performance Parallel and Distributed Computing, Minneapolis, MN, USA, 27 June 2022 - 1 July 2022*, pages 71–83. ACM, 2022.

[19] Ana Khorguani, Thomas Ropars, and Noel De Palma. Respct: fast checkpointing in non-volatile memory for multi-threaded applications. In Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis, editors, *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, pages 525–540. ACM, 2022.

[20] Dimitrios Koutsoukos, Raghav Bhartia, Michal Friedman, Ana Klimovic, and Gustavo Alonso. Nvm: Is it not very meaningful for databases? *Proc. VLDB Endow.*, 16(10):2444–2457, aug 2023.

[21] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In Tim Harris and Michael L. Scott, editors, *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 401–410. ACM, 2012.

[22] Ivy Bo 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 2019, Washington, DC, USA, September 30 - October 03, 2019*, pages 304–315. ACM, 2019.

[23] Maksym Planeta, Jan Bierbaum, Leo Sahaya Daphne Antony, Torsten Hoefler, and Hermann Härtig. Migros: Transparent live-migration support for containerised RDMA applications. In Irina Calciu and Geoff Kuenning, editors, *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 47–63. USENIX Association, 2021.

[24] Jinglei Ren, Jishen Zhao, Samira Manabi Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. Thynvm: enabling software-transparent crash consistency in persistent memory systems. In Milos Prvulovic, editor, *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, pages 672–685. ACM, 2015.

[25] Emil Tsalapatis, Ryan Hancock, Tavian Barnes, and Ali José Mashtizadeh. The aurora single level store operating system. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 788–803. ACM, 2021.

[26] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In Sam H. Noh and Brent Welch, editors, *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 169–182. USENIX Association, 2020.

[27] Lingjun Zhu, Lennart Bamberg, Anthony Agnesina, Francky Catthoor, Dragomir Milojevic, Manu Komalan, Julien Ryckaert, Alberto Garcia-Ortiz, and Sung Kyu Lim. Heterogeneous 3d integration for a risc-v system with stt-mram. *IEEE Computer Architecture Letters*, 19(1):51–54, 2020.