Challenges in Empirically Testing Memory Persistency Models

Vasileios Klimis
Queen Mary University of London
v.klimis@qmul.ac.uk

Alastair F. Donaldson
Imperial College London
alastair.donaldson@imperial.ac.uk

Viktor Vafeiadis
MPI-SWS
viktor@mpi-sws.org

John Wickerson
Imperial College London
j.wickerson@imperial.ac.uk

Azalea Raad
Imperial College London
azalea.raad@imperial.ac.uk

ABSTRACT

Memory persistency models provide the foundational rules for software engineers to develop applications that take advantage of non-volatile memory (NVM), dictating which (and when) writes to NVM are deemed persistent. Though formalised for Intel-x86 and Arm architectures, these models remain empirically unvalidated on actual machines. Conventional validation methods for memory *consistency* models fall short as test programs cannot differentiate between volatile cache reads and those from NVM. To address this, we employed a commercial device designed to intercept and log data on a system's memory bus in their order of arrival. We used this device to conduct a campaign using *litmus tests*—small programs designed to assess specific memory persistency behaviours—aimed at empirically validating Intel-x86 and Arm machine persistency guarantees.

We noted out-of-order memory writes and ensured they were not merely artifacts of our test setup. Analysis revealed Intel-x86's architecture *cannot* be validated via memory bus interception due to legitimate early subsystem reordering. Intel engineers confirmed the absence of dependable validation methods for the persistency claims of their architectures. Meanwhile, an expert-recommended Arm machine did not align with the formal persistency model due to a specification loophole, and further investigation suggests that no market-available Arm machine fully supports NVM.

Our finding for Intel highlights a major concern for software developers wishing to take advantage of NVM: currently there is, to our knowledge, *no viable way* to confirm the persistency guarantees claimed by Intel. Our results for Arm suggest that our interception-based approach is viable for reliably detecting reorderings in the memory subsystem, which will be valuable for empirical validation once NVM-supporting machines become available.

ACM Reference Format:
Vasileios Klimis, Alastair F. Donaldson, Viktor Vafeiadis, John Wickerson, and Azalea Raad. 2024. Challenges in Empirically Testing Memory Persistency Models. In *New Ideas and Emerging Results (ICSE-NIER'24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3639476.3639765

ICSE-NIER'24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0500-7/24/04…$15.00
https://doi.org/10.1145/3639476.3639765

1 INTRODUCTION

Non-volatile memory (NVM) combines the speed of DRAM with the durability and capacity of SSD, retaining data even after system crashes. As such, NVM (a.k.a. persistent memory) has the potential to radically change the way we build fault-tolerant software by optimising traditional and distributed file systems [6, 10, 12, 14, 29, 31], transaction processing systems for high-velocity real-time data [19], distributed stream processing systems [28], and stateful applications organised as a pipeline of cloud serverless functions interacting with cloud storage systems [25, 30].

Modern architectures, including Intel-x86 and Arm, provide write-back instructions for cache-to-memory operations. Specifically, Intel-x86 uses *clflush*(·) while Arm utilises *dc_cvap*(·) followed by *dsb*(*sy*). These can be harnessed by expert programmers to orchestrate the order in which writes to NVM are made persistent.

Formal *persistency* models such as Px86 [21] for Intel-x86 and PArm [22] for Arm, co-created with vendor experts, offer rigorous insights for NVM programs. While they are consistent with architectural specifications, they remain untested on real machines.

Empirical validation is vital for two reasons: holding vendors accountable for hardware adherence to specs in deployed machines, on which software developers depend (and which historically have been inconsistent in the related area of memory consistency [1, 3]) and to verify accurate persistency specifications that software developers can trust. Our goal is to validate their *soundness* – ensuring specs don't ban plausible behaviours. We also strive to avoid undue weakness, where specs allow behaviours not witnessed in machines.

In the context of memory *consistency* models, empirical validation has become commonplace. Tools such as `litmus` [4] have been widely used to run large numbers of small multi-threaded test cases (called *litmus tests*) on machines-under-test, in order to see which threads can see which writes in which order. However, memory *persistency* is more intricate. A `litmus`-like approach will not work since programs can't directly sense data persistence – distinguishing volatile cache from persistent memory is challenging.

A naive approach would involve causing a crash (e.g., by powering off) while the program is running, then reading persistent data from NVM upon restart. However, this only lets us observe the latest persisted write for each memory location, not the order of earlier writes. Additionally, frequent power-cycling makes it impractical to run the numerous tests needed for comprehensive validation, and scheduling precise 'crashes' becomes nearly impossible.

This brings us to the approach we put forward in this paper: monitoring writes reaching the NVM using a specialised interposer positioned between the motherboard and memory module. The data sensed by the interposer is then relayed to a device called the

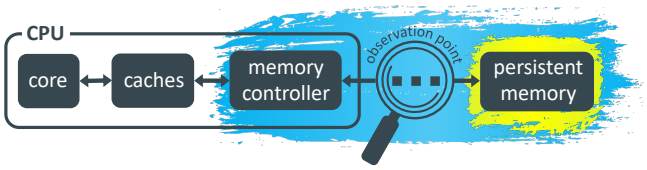

Figure 1: the persistence domain in a typical Intel-x86; and in an Arm system

DDR Detective [7] (see Fig. 2). By recording all memory accesses during a litmus test and analysing the resulting log, we can deduce the order in which writes become persistent.

We have applied our approach to Intel-x86 and Arm machines. Simple litmus tests revealed that writes can indeed enter memory out-of-order (i.e. disagreeing with the order prescribed by the formal persistency models), even when they are 'persist-separated'; that is, separated by explicit (sequences of) instructions that a programmer can use to persist writes in a certain order. Nevertheless, our results remain inconclusive due to intricacies in both architectures.

For Intel-x86, data headed to memory first traverses a battery-backed *write-pending queue* (WPQ) in the memory controller [24]. This data persists before leaving the processor, so that our post-processor observations are too late and are not relevant to order of persistence (Fig. 1). Still, we investigated whether the WPQ reorders writes, and indeed found that litmus tests showed out-of-order data propagation from processor to memory. Due to the battery-backed WPQ, these findings have no bearing on conformity of the Intel-x86 platform to the Px86 persistency model [21], but they raise two important points: (1) our findings are relevant for software engineers working with *remote direct memory access* (RDMA). In this framework, machines connect and share memory directly through the network interface card (NIC), bypassing the operating system and CPU (including its WPQ). This setup poses a challenge for developers: it lacks assurance for the persistence of remote pending writes that sidestep the WPQ. (2) the results emphasise a transparency concern in Intel CPU designs. While users ought to be able to independently verify Intel CPUs, for accountability, there is currently no method to do so, a fact even recognised by an Intel persistency engineer. Our research brings this issue to the forefront.

On Arm machines, without a WPQ-like component, DDR Detective should recognise reorderings against the Arm persistency model. Although we acquired an Arm v8.2 compliant machine, a loophole in the Arm manual means there is no clear point-of-persistency.[1] Technically, the out-of-order NVM writes identified by our tests do not breach the persistency specification, due to this loophole. As of now, no commercial Arm systems support NVM, but if one does, our method is primed to validate its assertions.

Contributions We claim two research contributions:

(1) an experimental setup demonstrating the potential to validate memory persistency models, aiding software engineers in understanding system behaviour; and

(2) empirical evidence revealing memory access reordering between the CPU and memory in both Intel-x86 and Arm machines, underscoring considerations for software engineers working with RDMA technologies.

Supplementary Material: Replicating our results, given the commercial nature of DDR Detective, may pose challenges. However, our automation scripts, litmus tests, and raw data are accessible as supplemental material at: https://zenodo.org/records/10427213.

2 EMPIRICAL INFRASTRUCTURE

We have devised a novel testing infrastructure to validate memory persistency models, outlined in Fig. 2. Using the FS2800 DDR Detective interposer [7], we tap into memory signals, analysing them through its Probe Manager software. This setup also enables remote SSH access to the target system, aiding automation.

The DDR Detective presented four challenges:

- Our litmus tests use virtual addresses, whereas DDR Detective records geometric addresses, reflecting the chip's internal organisation like rank, bank, row, and column.
- The DDR Detective can log the address of each memory write, but not the actual data (value) written. This means that we cannot infer the order of writes to the same location, only the order of writes to different locations.
- Configuring the DDR Detective to log all write operations quickly fills its storage. Though address logging can be restricted using a wildcard pattern, selecting the right pattern without prior knowledge is challenging.
- The DDR Detective is controlled by a GUI application with no command-line support, complicating automation.

We overcome these challenges as follows, where the numbers refer to the steps depicted in Fig. 2.

❶ The first step involves crafting a litmus test: a succinct program focusing on specific memory interactions, inspired by Raad et al. [21, 22]. While litmus tests can be auto-generated using Alloy methods [18, 21], we opted for a basic single-thread test with two distinct writes (Fig. 3) since it readily reveals evident issues, making complex tests redundant. Yet, our method can accommodate advanced multi-threaded testing scenarios.

❷ We now begin the process of identifying active geometric addresses. The logging process has been streamlined by instructing the DDR Detective to exclusively log 'writes' (effectively, *all* 'writes' *system-wide*) upon detecting the first one.

❸ Upon instructing DDR Detective to log, the litmus test's 'preamble' is launched. This allocates locations and carries out distinct write counts for each – e.g., 100 to one, and 200 to another if two locations are used. Subsequently, the test pauses for approximately 20 seconds (reason in step ❹).

❹ Meanwhile, once the DDR Detective observes any memory write, it starts logging all writes across the system. We set it to log a maximum of 8,000 writes, saved in a CSV file for analysis in step ❺.[2] The 20-second pause in step ❸ allows ample time for generating and analysing the CSV log file.

[1] A Point of Persistence (PoP) designates a location within the persistence domain where data is guaranteed to be retained, even if power is lost.

[2] The log size, referred to as 'trace memory depth' in FuturePlus terminology, is adjustable. For our experiments, 8,000 writes sufficed.

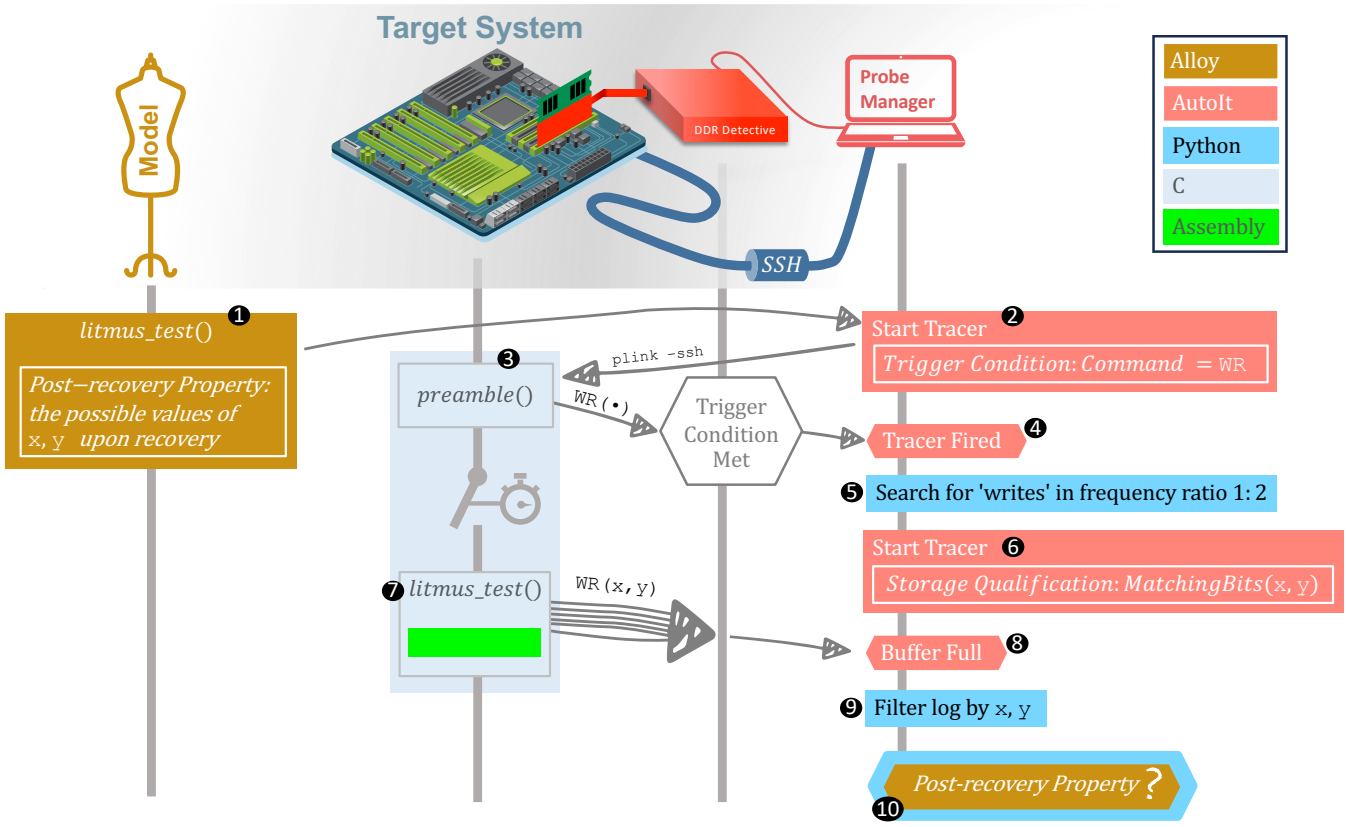

Figure 2: Our flow for validating persistency models; a colour nested inside another denotes an embedded program within outer code; a hexagon allows the flow to continue if the condition inside is true.

❺ Our Python script then scans the log to identify the two memory locations with 100 and 200 writes. This allows pinpointing the relevant geometric addresses.

❻ With the relevant geometric addresses identified, we reset the DDR Detective log. Subsequently, we direct DDR Detective to resume logging, exclusively for accesses matching a particular wildcarded pattern (since specific address sets are not supported, only one pattern can be used). The pattern is the most specific one that matches all geometric addresses determined in step ❺.

❼ The litmus test script is then reactivated, running multiple iterations within a 60-second timeframe as shown in Fig. 3. We currently do not check the post-recovery condition during each iteration; this becomes evident upon log inspection.

❽ The probe keeps collecting data until the log hits a set size.

❾ Another Python script filters the log, removing entries linked to addresses other than those reverse-engineered in step ❺.

❿ Lastly, we inspect the log to check the post-recovery condition for each litmus test iteration. In Fig. 3, this involves ensuring that writes to x and y strictly alternate in the log. Consecutive writes to the same location indicate reordering, addition, or deletion of writes, all signaling a persistency model violation.

Automation The DDR Detective probe manager lacks a command-line interface (CLI), relying solely on a Windows GUI. This hinders automation and remote access. We addressed this with AutoIt [5], a tool that automates Windows GUI tasks.

3 EXPERIMENTAL EVALUATION

We now present experiments that put the process depicted in Fig. 2 into practice. For Intel-x86, we utilised an Intel® Xeon® CPU E5-1630 v3 at 3.70GHz with 128 KiB L1, 1 MiB L2, and 10 MiB L3 caches. On the ARM side, we employed an Ampere® Altra™ M96-30 SOC with 96 Arm v8.2+ Cores and caches of 64 KB L1 I-cache, 64 KB L1 D-cache, and 1 MB L2.

Our experiments rely on the program in Fig. 3. It accesses addresses x and y in 2,000-batch runs, with the DDR Detective logging the memory bus transmission order. The program has a post-recovery condition indicating allowable values for x and y upon recovery. Since it's not programmatically checkable, we use the DDR Detective to analyse the logged memory instruction order.

Metrics Our experimental evaluation employs two key metrics: *reorderings* and *deviations*. A *reordering* occurs when the observed instruction sequence diverges from the vendor-prescribed order. *Deviation* (Δ) measures the gap between the actual persisted writes, as logged by the interposer, and the test program's issued writes.

Key Observations For Intel-x86, our results showed slight deviations and rare reorderings. We believe that, in their current state,

```
preamble
1: x ← posix_memalign(size_≤L1 Cache, CACHE_LINE_SIZE)
2: y ← posix_memalign(L1 Cache≤size_≤L2 Cache, CACHE_LINE_SIZE)

3: reg ← 0
4: for x ← 1 to 100 do
5:     x ← 1; asm(dc_cvap(x)); asm(dsb(sy));
6:     y ← 1; asm(dc_cvap(y)); asm(dsb(sy));
7:     y ← 1; asm(dc_cvap(y)); asm(dsb(sy));
8: end
9: sleep(20)
10: while time < 60 do
11:     ++reg
litmus test
12:     x ← reg;
13:     asm(dc_cvap(x)); asm(dsb(sy));   // persist(x)
14:     sleep(·)   // suspend execution
15:     y ← reg;
16:     asm(dc_cvap(y)); asm(dsb(sy));   // persist(y)
17:     sleep(·)   // suspend execution
18:     // persistency property
19:     // y_pd = reg ⟹ x_pd = reg
20: end
```

Figure 3: A litmus test in ARMv8.2 running in a loop and preceded by the corresponding preamble; the `asm` keyword declares inline assembler instructions that are embedded within the C code; x_{pd} denotes the value of memory location x read from the persistence domain utilising physical probing.

Intel machines cannot be reliably and independently validated for persistency behaviour. Our study is the first to spotlight this issue.

In the context of Arm, we identified 2.5% deviations and 5.78% reorderings. Ampere shared that their Altra™ SOC doesn't support fine-grained persistent memory and is devoid of a Point of Persistence. Due to an Arm8.2 loophole, `DC CVAP`[3] can mimic `DC CVAC` without a defined PoP. At present, programmers lack the means to ascertain the presence of a PoP or the 'meaningful' implementation of `DC CVAP` within a system, without liaising with the designer.

On the ARM machine, a notable concern was the frequent interception of more writes by DDR Detective than those generated by the litmus test. Explaining this phenomenon remained elusive, though fewer propagated writes might be attributed to compiler optimisations, for instance.

We also pursued some 'curiosity-driven' experiments, exploring:

DC CVAP vs. DC CVAC: The data from Figs. 4a and 4b doesn't offer sufficient statistical evidence to indicate that `DC CVAP` defaults to a different behaviour than `DC CVAC`.

Reordering & Memory Size: Larger memory chunks exhibit increased reorderings. Interestingly, a marked transition occurs beyond 64 KB, hinting at the influence of the L1 cache size (Fig. 4c).

Repeated Persists & Reordering: As the *persist* (`DC CVAP`) repetitions grow, reorderings wane, implying a tighter adherence to expected specifications (Fig. 5a).

Processor Suspension's Effect: Delays following the *write-persist-barrier* in the litmus test decrease reorderings, implying a closer – but not exact – alignment with `DC CVAP` specs (Fig. 5b). The presence of a delay, rather than its specific duration, is crucial for this shift.

[3]The ARM system instructions `DC CVAP` and `DC CVAC` stand for Data or unified Cache line Clean by Virtual Address to the point of Persistence and Coherency, respectively.

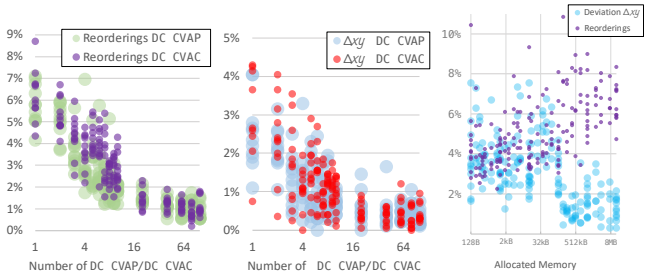

Figure 4: (a) *reorderings* and (b) *deviations* corresponding to varying `DC CVAP`/`DC CVAC` counts; (c) *deviations/reorderings* against uniform memory size allocation. Each data point represents a 2K-iteration test outcome on the Arm machine.

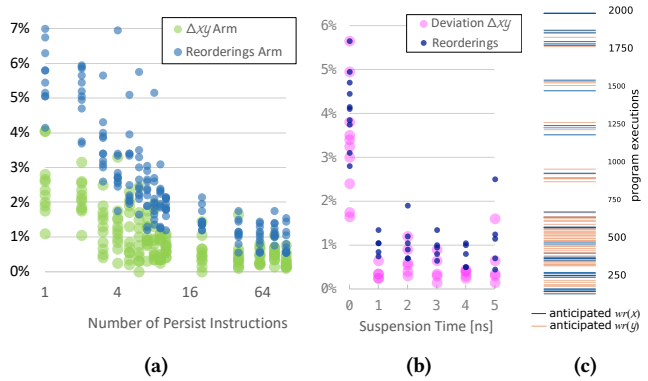

Figure 5: (a) *deviations/reorderings* based on changing persist counts between writes; (b) effects of intermittent suspensions post each write (each mark reflects a 2K-iteration test outcome); (c) random *distribution* of anomalies from a single 2K-iteration run.

Distribution of Reorderings: Besides the sheer *number* of anomalies, we assessed their distribution. Figure 5c reveals that, while anomalies spread across all executions, a denser concentration appears in the first half of the dataset.

Overall, our inquisitive tests underscore our setup's ability to provide insights into memory write dynamics and their interplay with various factors. Our method also sets the stage for future validations as persistency support becomes available in Arm systems.

4 RELATED WORK

Our work is situated within a rich lineage of research that has systematically validated memory *consistency* models across an array of architectures, including x86 [4, 26], IBM Power [23], Arm [2], GPUs [1], and hybrid CPU/FPGA systems [8]. The landscape of these models has evolved, recently integrating facets like virtual memory [27] and non-temporal accesses [20]. While traditional consistency models are amenable to software-centric validation, the

nuanced interplay of caches in modern architectures necessitates a tailored validation paradigm for *persistency* models.

Alongside our persistency model validation, there's a significant focus on testing persistent programs. PMTest [17] and similar tools like Pmemcheck [11] and PMAT [9] detect bugs through user annotations and dynamic analysis. XFDetector [16] delves deeper, pinpointing unforeseen interactions surrounding crashes. PMFuzz [15], a test-case generator, emphasises path coverage, especially where persistency-related commands appear. These tools rely on the vendor's architectural specification as the ground truth, underscoring the significance of our efforts in enhancing their reliability.

5 CONCLUSION AND FUTURE PLANS

We have introduced a litmus-testing campaign to empirically validate persistency guarantees of Intel-x86 and Arm systems. Traditional memory validations fall short in this realm, emphasising the significance of our tailored approach. Our experiments showcase discrepancies between observed behaviours and vendors' specifications, underscoring the need for hands-on testing over mere reliance on documentation. We provide a reusable methodology to validate persistency guarantees if vendors pledge new promises in their forthcoming releases.

We see significant potential in advancing smarter, bespoke techniques for memory persistency validation. Our upcoming efforts will involve: (1) auto-generating 'interesting' test cases from Alloy models to spotlight corner scenarios (marginal weak behaviours), using an approach akin to [13]; (2) refining generalised strategy to automatically extract intricate violation patterns from logs (representative of the permissible persistent memory state) applicable to any test case, with a keen focus on the nuances of complex multi-threaded litmus tests.

As another step forward, we aim to pinpoint the exact location of the stored value within the memory hierarchy on Intel architectures through a methodical memory hierarchy timing attack using timestamped instructions. We plan to leverage inherent timing variations for insightful inferences. Additionally, we intend to explore systems-on-chip, such as Xilinx's Zynq Ultrascale+, which integrates a multicore Arm processor with programmable logic.

ACKNOWLEDGEMENT

This work is partially supported by EPSRC project EP/R006865/1 and has additionally received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101003349).

REFERENCES

[1] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *ASPLOS*.

[2] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* (2021).

[3] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Computer Aided Verification*.

[4] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running Tests against Hardware. In *Tools and Algorithms for the Construction and Analysis of Systems*.

[5] AutoIt. 2022. AutoIt Downloads. https://www.autoitscript.com/site/autoit/downloads/

[6] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. 2021. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In *19th USENIX Conference on File and Storage Technologies, FAST*.

[7] FuturePlus Systems. 2017. DDR Detective FS2800. https://www.futureplus.com/ddr-detective/

[8] Dan Iorga, Alastair F. Donaldson, Tyler Sorensen, and John Wickerson. 2021. The semantics of shared memory in Intel CPU/FPGA systems. *Proc. ACM Program. Lang.* 5, OOPSLA (2021).

[9] Louis Jenkins and Michael L. Scott. 2020. Persistent Memory Analysis Tool (PMAT). In *11th Annual Non-Volatile Memories Workshop*. https://louisjenkinscs.github.io/publications/PMAT_EA.pdf.

[10] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. 2021. WineFS: A Hugepage-Aware File System for Persistent Memory That Ages Gracefully. In *SOSP*.

[11] Tomasz Kapela. 2015. An introduction to pmemcheck. https://pmem.io/blog/2015/07/an-introduction-to-pmemcheck-part-1-basics/.

[12] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. 2021. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *SOSP*.

[13] Vasileios Klimis, Jack Clark, Alan Baker, David Neto, John Wickerson, and Alastair F. Donaldson. 2023. Taking Back Control in an Intermediate Representation for GPU Computing. *Proc. ACM Program. Lang.* 7, POPL (2023).

[14] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *SOSP*.

[15] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Manabi Khan. 2021. PMFuzz: test case generation for persistent memory programs. In *ASPLOS*.

[16] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas F. Wenisch, Aasheesh Kolli, and Samira Manabi Khan. 2020. Cross-Failure Bug Detection in Persistent Memory Programs. In *ASPLOS*.

[17] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Manabi Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *ASPLOS*.

[18] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. 2017. Automated Synthesis of Comprehensive Memory Model Litmus Test Suites. In *ASPLOS*.

[19] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, Michael Stonebraker, Kristin Tufte, and Hao Wang. 2015. S-Store: Streaming Meets Transaction Processing. *Proc. VLDB Endow.* (2015).

[20] Azalea Raad, Luc Maranget, and Viktor Vafeiadis. 2022. Extending Intel-x86 consistency and persistency: formalising the semantics of Intel-x86 memory types and non-temporal stores. *Proc. ACM Program. Lang.* 6, POPL (2022).

[21] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2020. Persistency semantics of the Intel-x86 architecture. *Proc. ACM Program. Lang.* 4, POPL (2020).

[22] Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019. Weak persistency semantics from the ground up: formalising the persistency semantics of ARMv8 and transactional models. *Proc. ACM Program. Lang.* 3, OOPSLA (2019).

[23] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *PLDI*.

[24] Steve Scargall. 2020. *Persistent Memory Architecture*. Apress.

[25] Srinath Setty, Chunzhi Su, Jacob R. Lorch, Lidong Zhou, Hao Chen, Parveen Patel, and Jinglei Ren. 2016. Realizing the Fault-Tolerance Promise of Cloud Storage Using Locks with Intent. In *OSDI*.

[26] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010).

[27] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. 2022. Relaxed virtual memory in Armv8-A. In *ESOP*.

[28] Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J. Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta, Varun Madan, and Jun Rao. 2021. *Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka*.

[29] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *FAST*.

[30] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *OSDI*.

[31] Bohong Zhu, Youmin Chen, Qing Wang, Youyou Lu, and Jiwu Shu. 2021. Octopus$^+$: An RDMA-Enabled Distributed Persistent Memory File System. *ACM Trans. Storage* 17, 3 (2021).