Facilitating Process Persistence in Hybrid Non-Volatile Memory Systems

A Thesis Submitted

in Partial Fulfilment of the Requirements

for the Degree of

Doctor of Philosophy

by

Arun KP

18111263

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

July 2025

Certificate

It is certified that the work contained in this thesis entitled **"Facilitating Process Persistence in Hybrid Non-Volatile Memory Systems"** by **Arun KP** has been carried out under my supervision and that it has not been submitted elsewhere for a degree.

Dr. Debadatta Mishra
Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur

Dr. Biswabandan Panda
Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Declaration

This is to certify that the thesis titled **“Facilitating Process Persistence in Hybrid Non-Volatile Memory Systems”** has been authored by me. It presents the research conducted by me under the supervision of **Dr. Debadatta Mishra** and **Dr. Biswabandan Panda**.

To the best of my knowledge, it is an original work, both in terms of research content and narrative, and has not been submitted elsewhere, in part or in full, for a degree. Further, due credit has been attributed to the relevant state-of-the-art and collaborations with appropriate citations and acknowledgments, in line with established norms and practices.

Arun KP
Roll No. 18111263
CSE Department
Indian Institute of Technology Kanpur

Abstract

Name of the student: **Arun KP** Roll No: **18111263**

Degree for which submitted: **PhD** Department: **CSE Department**

Thesis title: **Facilitating Process Persistence in Hybrid Non-Volatile Memory Systems**

Thesis supervisors: **Dr. Debadatta Mishra** and **Dr. Biswabandan Panda**

Computing systems with persistent process semantics enable applications to resume execution from a consistent state after an abrupt system restart due to a crash or power failure, saving computation time and energy. Enabling process persistence capability in systems requires storing the state of a process consisting of CPU registers, memory state, and operating system metadata in a persistent device. In this thesis, we focus on saving the state of a process in a hybrid memory system comprising Non-Volatile Memory (NVM) along with volatile DRAM. While NVM allows access through the load/store interface of a CPU, using NVM for persistence requires special care to guarantee consistent data updates in NVM. Therefore, we need mechanisms at the hardware and/or software layers to ensure a consistent memory state in NVM across reboots.

In this thesis, we first analyze the performance overhead associated with different memory persistence mechanisms for NVM on Intel x86-64 and Arm64 systems. We study the performance of conventional persistence barrier primitives to modify data reliably in NVM. We also analyze the performance of advanced mechanisms, such as logging, for memory persistence. We study performance overhead by incorporating memory persistence mechanisms in data structures widely used in modern operating systems to maintain process states. We observe that the performance overhead of different persistence techniques depends upon the nature of the data structure; for example, queue incurs the highest performance overhead compared to other data structures across different persistence barrier primitives on Intel x86-64 and Arm64. We also observe that using memory serialization operations to enforce the order of writes to NVM contributes significantly to the performance overhead of all memory persistence mechanisms.

The design choices to persist different process states determine the performance of achieving process persistence. Thus, we need a simulation framework to study end-to-end performance trade-offs across different design choices for achieving process persistence in a hybrid memory system. As the existing simulation infrastructure requires nontrivial adaptation to incorporate hybrid memory for process persistence study, we create a hybrid memory simulation framework, *Kindle*, that supports process persistence. *Kindle* provides an end-to-end framework consisting of an operating system and a hardware simulator to enable quick prototyping of mechanisms and policies for process persistence in a hybrid memory system. Using *Kindle*, we compare the efficiency of two design choices to persistently maintain the virtual address translation of processes. We also implement prototypes of two state-of-the-art hybrid memory schemes using *Kindle* to show its capability to realize complex designs.

From the empirical analysis, we observe that the working set size and memory access patterns of applications influence the performance of memory persistence mechanisms. Among the persistent process states, the memory state is a crucial component in size and importance. Thus, the performance of the memory state persistence mechanism decides the overall performance of process persistence. The memory layout of a process consists of heap and stack areas, and they exhibit distinct usage patterns. To analyze the stack usage of different applications, we create an efficient stack tracing framework, *SniP*, for multi-threaded applications. *SniP*, with its targeted stack tracing capability, generates a trace file containing only stack access of an application, reducing the time taken to create a stack trace compared to tracing the entire memory area of a program using a state-of-the-art program tracing tool like Intel Pin and separating stack accesses from it. *SniP* results in $\sim$75$\times$ reduction in trace file size for the TinyDBM key-value store application and up to ∼24× reduction in tracing time for the Python3 HTTP server compared to Intel Pin.

We further investigate unique properties of the program stack, such as the grow and shrink pattern of usage, presence of activation record, and indirect usage, with the help of *SniP*. We identify that these unique stack properties necessitate nontrivial adaptations to state-of-the-art memory persistence mechanisms for NVM to achieve program stack persistence efficiently. Even with adaptations, state-of-the-art memory persistence mechanisms such as undo and redo logging incur more than 35× slowdown in persisting stack compared to no persistence (i.e., stack in DRAM). Thus, to address inbuilt inefficiencies in state-of-the-art memory persistence approaches while using them for the stack, we propose *Prosper*, a periodic checkpointing-based hardware-software co-designed approach that handles unique stack properties. *Prosper* hardware tracks stack modifications at sub-page granularity, resulting in ∼4× reduction (on average) in checkpoint size compared to the state-of-the-art scheme based on the memory management unit for tracking memory updates (dirty bit scheme). *Prosper* hardware introduces performance overhead of less than 1% on average, providing up to 3.6× reduction in stack persistence overhead compared to the state-of-the-art NVM memory persistence schemes. Integration of Prosper with existing state-of-the-art memory persistence mechanisms for heap also benefits achieving persistence for the entire memory area, providing 2.6× improvement over solely using the state-of-the-art mechanism for the entire memory area persistence.

This thesis contributes infrastructure and mechanisms for process persistence in a hybrid memory system with NVM and DRAM. The thesis initially provides insights into the performance overhead of primitive and advanced memory persistence mechanisms for NVM and later proposes a hybrid memory simulation framework with NVM and DRAM. The framework incorporates process persistence semantics using a periodic checkpoint-based scheme to maintain the execution context and memory state of a process consistently. Notably, we create an efficient stack tracing framework and show the need to specialize memory persistence schemes based on the memory area (heap or stack) under consideration by bringing out stack-specific properties using the framework. Finally, we propose a checkpoint-based memory persistence scheme for the stack that handles unique stack properties and provides efficient stack persistence.

Acknowledgements

I extend my heartfelt gratitude to all faculty members and technical and non-technical staff of the Computer Science and Engineering Department at IIT Kanpur for their unwavering support throughout my PhD journey. I am grateful to CDOS research group members, KD-222 lab members, and fellow PhD students for the invaluable guidance and knowledge they shared during discussions. I thank Dr. Biswabandan Panda, who moved to IIT Bombay during COVID-19, for making this PhD journey possible by offering me a position at IIT Kanpur. I am fortunate to have worked with Dr. Debadatta Mishra as my primary supervisor and received his technical wisdom and constant support, which made my PhD journey smooth and intellectually satisfying.

I am immensely thankful to my wife, parents, family, and friends for their comfort and support throughout this journey, enabling me to focus on and complete my PhD work. I thank all my teachers, starting from school days, who believed in me unconditionally.

Finally, I am indebted to all citizens of India for the financial support I received through my PhD fellowship and to every known and unknown individual whose blood, sweat, and tears made each day of this journey possible.

Contents

List of Figures

List of Tables

Abbreviations

NVM	Non-Volatile Memory
ISA	Instruction Set Architecture
HDD	Hard Disk Drive
SSD	Solid State Drive
PCM	Phase Change Memory
FASE	Failure Atomicity Section
ADR	Asynchronous DRAM Refresh
SP	Stack Pointer
ROB	Reorder Buffer
HWM	High Water Mark
LWM	Low Water Mark
WPQ	Write Pending Queue
MC	Memory Controller
WAL	Write Ahead Logging
CoW	Copy on Write
MPKI	Misses Per Kilo Instructions

LLC	Last Level Cache
DCPMM	Data Center Persistent Memory Module
PTBR	Page Table Base Register
TLB	Transaction Lookaside Buffer
PTE	Page Table Entry
API	Application Programming Interface

Dedicated to Universe.

Chapter 1

Introduction

Process persistence enables applications to resume execution from a consistent state after restart or system crash. Process persistence requires periodically saving the state of a process consisting of CPU registers, memory, and OS meta-data in a persistent device [1]. The time needed to save state in a persistent device depends upon the access latency of the device and the method used to persist data. For example, using file system layers [2, 3] to persist data on secondary storage devices (HDDs and SSDs) incurs serialization and other related software overheads. Therefore, an efficient process persistence approach requires a device with better access latency. The device should also provide access without additional software layers. Byte addressable Non-Volatile Memory (NVM) meets this requirement by providing access through the CPU load/store interface and persists data with access latency comparable to volatile DRAM [4]. NVM, connected to a system's memory interface, facilitates direct access by eliminating complex storage management middle-wares such as file systems [5]. Thus, NVM provides an opportunity to build efficient infrastructure and mechanisms to achieve process persistence with minimum performance overhead.

However, using NVM for persistence presents the following challenges related to *crash consistency* and *system efficiency*. *Crash consistency* challenges arise due to direct access from the CPU through the load/store interface. The *system efficiency* challenge occurs if NVM is used as the sole memory in the system by replacing conventional DRAM memory.

1. **Crash consistency:** Data path from CPU to NVM consists of intermediate volatile components such as hardware caches. Ensuring consistency of memory updates in the presence of these volatile components requires specialized techniques to provide a

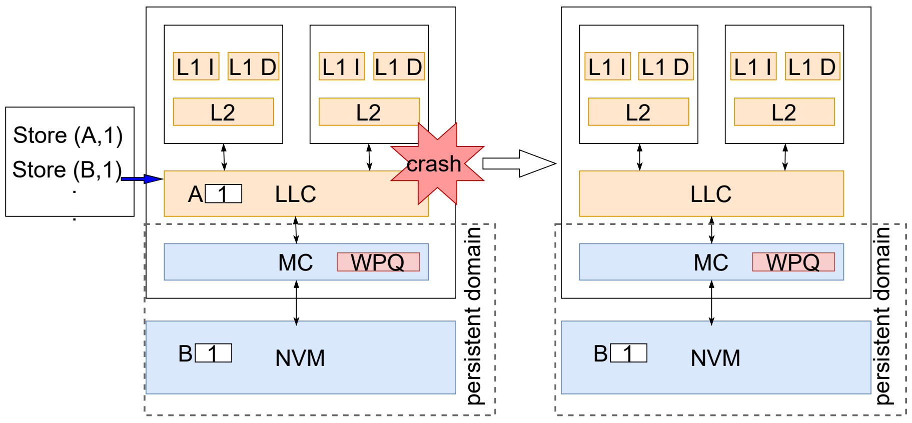


FIGURE 1.1: Illustration of the need to order stores to NVM in the presence of volatile caches.

consistent memory state across system restarts. In-flight data in these intermediate volatile components are lost in case of a power failure or abrupt restart. Moreover, these components can change the order of writes to NVM based on exerted data eviction policies, causing deviation in the actual memory state from the expected program state. These intricacies of processor-memory data path organization require non-trivial adaptation of traditional file systems or database techniques (e.g., journaling and transactions [6]) to use for NVM [7–10].

2. **System Efficiency:** NVM has higher read and write latency than DRAM; for example, phase change memory (PCM) is 1.6× slower than DRAM [11]. Therefore, using NVM as a drop-in replacement for DRAM leads to poor application performance.

1.1 Crash consistency challenge

The crash consistency challenge is due to the presence of volatile intermediate components in the data path to NVM. Therefore, addressing this challenge requires mechanisms to determine whether writes reach the persistent domain and verify the order in which writes reach the persistent domain. Persistent domain encompasses system components that can retain data across reboots. Thus, mechanisms to address this challenge, i.e., memory persistence mechanisms, should ensure the order of writes to the persistent domain. The mechanisms should also provide detectability, i.e., assurance on whether a write is complete or not from the objective of recovery after a crash [12–15]. For example, Figure 1.1

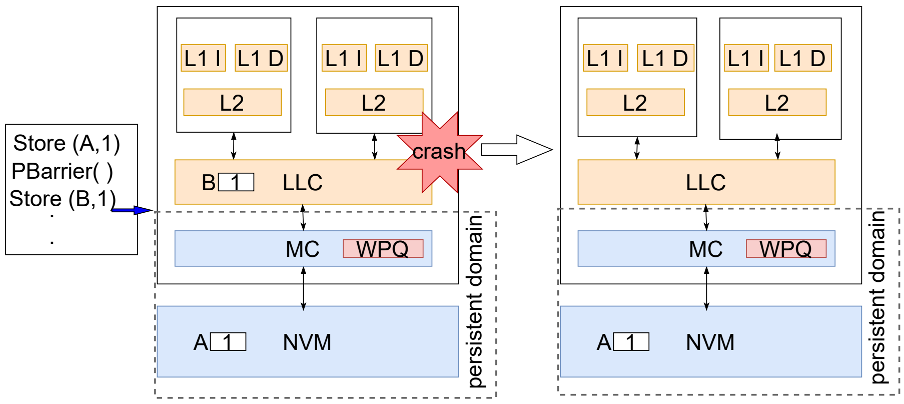


FIGURE 1.2: Illustration of persistent barrier to order stores to NVM in the presence of volatile caches.

demonstrates the need to order writes to the persistent domain in the presence of volatile caches.

The persistent domain in Figure 1.1 comprises of write pending queue (WPQ) and NVM. The program order expects store to memory location `B` persists after `A`, but stores to memory locations `A` and `B` may reach the persistent domain out of program order due to cache eviction policy, which is independent of the program order. In Figure 1.1, store to `B` reaches the persistent domain, but store to `A` remains in the volatile cache. In this case, only the value at memory location `B` survives after an unexpected restart, creating an inconsistent memory state after reboot. The issue highlighted in Figure 1.1 can be fixed by ensuring store to `A` reaches the persistent domain before store to `B`. The schemes that impose such an order of writes to NVM, i.e., memory persistency [16], define the order of NVM writes with respect to failures.

We can broadly categorize memory persistency as *strict* or *relaxed* based on its association with the memory consistency model. Under the strict category, the order in which writes persist is the same as in which writes become visible outside a core, as defined by the memory consistency models. A store's visibility is associated with persistence under *strict* persistency. Some of the approaches for *strict* persistence are,
(i) store instruction completes after updating value in NVM, which requires access latency of NVM.
(ii) use an intermediate persistent buffer in the data path to NVM for pooling stores.

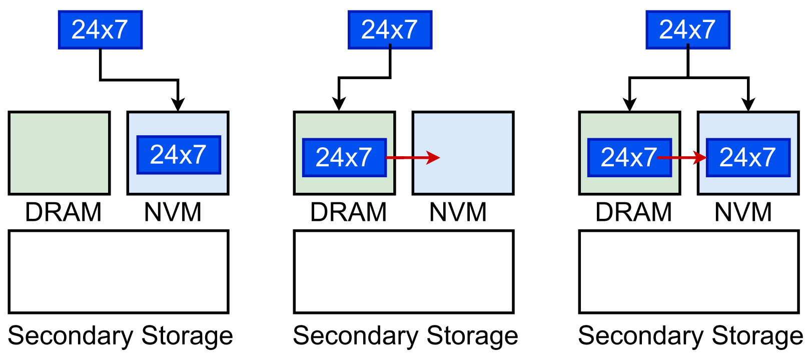
Relaxed persistence disassociates the order of NVM writes from memory consistency semantics, requiring persistence barriers [17] to enforce the required persistence order guarantees. Figure 1.2 shows the usage of a persistent barrier to enforce the order of NVM writes. Pelley, S et al. [16] propose epoch persistency, an example of relaxed persistence that allows multiple stores inside an epoch, demarcated with persistent barriers. Epoch persistence allows reordering persistent stores inside an epoch but not across persistent barriers demarcating epoch boundaries.

We can implement strict and relaxed persistence methods in software and/or hardware. An example of a software-based approach for strict persistence is to flush modified cache lines to the persistent domain, and a hardware-based approach is to make caches part of the persistent domain by providing enough power backup to writeback changes to NVM in the event of power failure. Relaxed persistence provides more flexibility and can provide a failure atomicity section (FASE) by ensuring the durability of stores inside a section demarcated by persistence barriers. FASE [18] guarantees that a set of operations is completed as a whole or none.

FASE implementation approaches include write-ahead-logging (WAL), dual-copy, Copy-on-Write (CoW), checkpointing [18]. WAL has variants that log old values before modification (undo log [9, 19–21]), new value of modification (redo log [22–24]), or memory operations [8]. CoW or shadow paging makes a copy of the memory page before modification to retain the old value if required, with optimizations allowing copy at sub-page granularity [25]. The dual copy mechanism keeps two versions of a data structure: a working version and a consistent version. The working version is modified in place, and the consistent version is synced with changes at the end of consistency intervals [26]. A periodic checkpointing-based scheme keeps track of modifications and persists changes at the checkpoint interval end [1]. These Memory persistency schemes ensure that NVM is in a consistent memory state by handling challenges associated with the order of writes and intermediate volatile components.

1.2 System efficiency challenge

The system efficiency challenge is due to the higher read/write latency of NVM compared to DRAM. Therefore, an approach to address this challenge is to compensate for the access latency of NVM by using a hybrid memory organization with DRAM [27]. A hybrid memory setup can provide better application performance than a system with only NVM by taking advantage of both DRAM and NVM. A hybrid memory system allows data to be placed in NVM and/or DRAM, enabling OS memory managers to coordinate allocations for high memory capacity with low access latency [28]. For example, a typical OS policy



(a) Use NVM (b) Use DRAM (c) Use DRAM and NVM

FIGURE 1.3: Schematic diagram for different ways to persist data in hybrid memory systems.

can place frequently accessed hot memory pages in DRAM and migrate cold pages to NVM for better performance. Applications can also employ different performance, capacity, and persistence approaches by keeping data in NVM or DRAM in a hybrid memory system.

Figure 1.3 illustrates different configurations to maintain data, such as process state in a hybrid memory system for persistence. The process state can be maintained entirely in NVM with state changes wrapped inside one of the persistence mechanisms to provide failure atomic section (FASE) [18] for state changes (Figure 1.3(a)), or the process state can be maintained in DRAM and periodically persisted to NVM by copying from DRAM to NVM (Figure 1.3(b)). In this approach, copying modifications from DRAM to NVM should be performed in a failure atomic manner to maintain a consistent state in NVM. Finally, we can have a mixed approach by maintaining some states in DRAM and some in NVM, and copying states in DRAM to NVM in a failure atomic manner (Figure 1.3(c)). Maintaining the process state completely in NVM has the advantage of immediate persistence but encounters the overhead due to higher read/write latency of NVM. Retaining the process state in DRAM has the advantage of better read/write access but provides delayed persistence due to the copy operation from DRAM to NVM. Thus, using hybrid memory systems, we can design process persistence schemes that address both *crash consistency* and *system efficiency* challenges of NVM.

1.3 Process persistence in a hybrid NVM system

Process persistence enables applications to continue execution from a consistent state after an abrupt system restart. Process persistence in a hybrid memory system with NVM allows saving the process state in NVM by employing memory persistence mechanisms. Therefore, in the context of process persistence, choosing a memory persistence mechanism that allows coordination with OS activities is essential to retrieve the process state. It is crucial to communicate OS operations such as context switches to memory persistence mechanisms for maintaining a per-process consistent state. Memory persistence mechanisms at hardware [9, 20, 24, 29] and user space [22, 30–34] generally guarantee memory persistence for a section of code in the application. Hence, these mechanisms require non-trivial adaptation for process persistence, especially for identifying the boundaries of a process memory state that spans across the user and OS layers. For example, a hardware-based undo logging mechanism such as ATOM [9] provides consistency for memory updates from sections of code demarcated between `atomic_begin` and `atomic_end`, thus limiting the scope of ATOM to the user space of an application. Thus, generic memory persistence mechanisms have challenges in coordinating with OS activities to capture the process state. Therefore, OS-level checkpoint solutions are more practical for process persistence.

OS-level checkpoint mechanism can capture per-process state as it has visibility into process boundaries. OS-level checkpoint procedures can leverage the additional hardware support for memory persistence to simplify the complexities associated with periodic memory checkpointing. This thesis uses a periodic checkpoint-based technique [35–37] in a hybrid memory system with NVM and DRAM to achieve process persistence.

1.4 Contributions

This thesis makes four contributions to enable process persistence in hybrid memory systems with NVM. First, we quantify the performance implications of persistent barriers, which are essential for enforcing the order of stores to NVM. The second contribution is an open-source hybrid memory framework incorporating process persistence, allowing quick prototyping of designs and ideas for a hybrid memory system. Our third contribution is an open-source stack tracing framework providing insights into stack usage of multi-threaded applications. Finally, the fourth contribution provides a hardware-software (OS) co-designed checkpoint-based approach for program stack persistence. The details about each contribution are mentioned in the following subsections.

Figure 1.4 shows a high-level representation of contributions in this thesis. In the following sections, we outline details on the contributions of this thesis (§ 1.4) and its organization (§ 1.5).

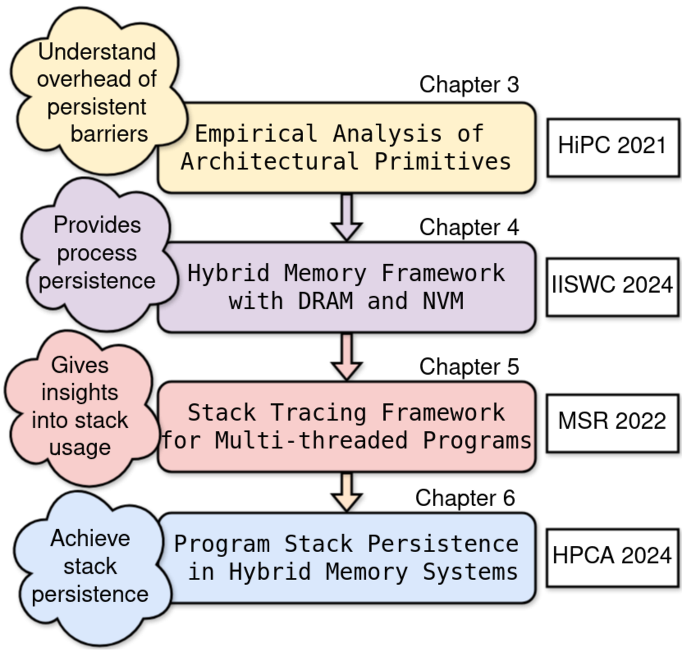

Understand overhead of persistent barriers
Chapter 3
Empirical Analysis of Architectural Primitives
HiPC 2021
Provides process persistence
Chapter 4
Hybrid Memory Framework with DRAM and NVM
IISWC 2024
Gives insights into stack usage
Chapter 5
Stack Tracing Framework for Multi-threaded Programs
MSR 2022
Achieve stack persistence
Chapter 6
Program Stack Persistence in Hybrid Memory Systems
HPCA 2024

FIGURE 1.4: Contributions in this thesis.

1.4.1 Empirical analysis of architectural primitives

We study the overhead of primitive architectural artifacts, such as cache line flush and memory fences, typically used as persistent barriers in NVM systems [38]. The study also provides insights into the overhead of high-level software-based memory persistence mechanisms such as redo and undo logging since they use persistent barriers to order log writes to NVM. Persistent barriers also play a crucial role in the data structures, memory allocators, and memory management schemes proposed specifically for NVM [39–46]. Most instruction set architectures (ISAs) provide a set of instructions (e.g., `clflush` and `mfence` in x86-64) along with the required extension of the persistent domain (e.g., Intel ADR [47]) to propagate changes to NVM consistently. We empirically analyze the performance overhead of different data consistency methods provided by Intel x86-64 (`clflush`, `clflushopt`, `clwb`) and Arm64 (`civac`, `cvac`) ISAs [48][49]. Understanding the performance implications of persistence primitives can guide application/middleware developers in choosing the right technique and accounting for the resource overheads during capacity planning. Moreover, the analysis presented in this empirical study can provide directions to improve the efficiency of architectural primitives for NVM consistency. We observe that there is no *one-size-fits-all* approach to choosing persistency primitives, as the performance overhead of persistency primitives depends upon the nature of the workload. Moreover, the influence of memory footprint and memory access pattern on performance overhead of data persistency primitives mostly depends on the nature of workloads. Insights from this empirical study provide information on the expected overhead to consistently maintain process and memory states using different mechanisms.

1.4.2 Hybrid memory framework with DRAM and NVM

Kindle is an open-source hybrid memory framework based on gem5 [50, 51] and gemOS [52]. We created *Kindle* because the nature of problems and proposed solutions for hybrid memory systems require an infrastructure allowing *extension, validation, and evaluation* of the complete system stack. *Kindle* serves as a platform to explore and prototype research ideas in hybrid memory systems crossing hardware-software boundaries. *Kindle* uses gemOS as the operating system component since using gem5-Linux full-system simulation setup to explore ideas in hybrid memory systems has the following shortcomings,

(i) While gem5 models NVM controller and Linux kernel can detect NVM on real hardware (such as Intel DCPMM) [53, 54], their integration is non-trivial (in gem5), especially considering constantly evolving hardware and OS design.

(ii) Linux kernel is heavy with features whereby the OS functions and services can consume a significant part of a simulation, which may not be desirable for quick prototyping of design ideas.

(iii) Designing proof of concepts in Linux requires considerable understanding and changes in the Linux memory management subsystem, which has non-trivial complexities.

Kindle provides process persistence in hybrid memory systems while facilitating the analysis of different design challenges and alternatives. We showcase two design choices to consistently maintain the page table containing virtual to physical address translation information as part of achieving process persistence using *Kindle*,

- Hosting the page tables directly on NVM.
- Hosting the page tables in DRAM while performing periodic checkpoints into NVM.

Kindle aided as a platform to perform our study on process persistence and designing a memory persistence scheme specialized for program stack (*Prosper*).

1.4.3 Stack tracing framework for multi-threaded programs

SniP is an open-source stack tracing framework for multi-threaded applications. Stack captures a program's dynamic run time state and can provide insights [55] into program behavior. Stack analysis requires dynamic intervention as run-time stack usage can not be foreseen, which makes static analysis techniques ineffective. Therefore, we depend on dynamic run-time techniques for stack analysis. Using debugging tools (e.g., GDB [56]) can provide insights into the stack usage but requires manual intervention and does not scale for long-running applications. Dynamic binary instrumentation (DBI) tools are very convenient as they require no preparation and can trace the entire program [57, 58].

One of the approaches for run-time stack analysis is to trace all memory accesses and filter stack-specific accesses during the offline analysis phases. However, stack analysis through a trace-driven approach by tracing the program results in large trace files and higher tracing time. In addition, the offline analysis approach requires dynamic stack virtual address ranges to filter the trace, and offline analysis can use memory layout information provided by the operating system for this purpose. However, capturing the stack range information for different threads in multi-threaded applications is challenging. The stack areas for threads can be allocated anywhere in the program address space depending on the state of the address space and OS support for dynamic allocation. For example, stack areas for the threads created using the POSIX thread library in the Linux OS are allocated at run time in the non-contiguous areas in the address space. *SniP* handles the challenge of identifying stack areas of threads in a multi-threaded application, making dynamic tracing possible. *SniP* is built around Intel's binary instrumentation tool Pin [58]. It provides a framework for efficient run-time tracing of stack areas of multi-threaded applications by identifying the stack areas dynamically. The targeted tracing capability of *SniP* reduces trace file size by up to 75× and time to trace by up to 24×, showing its efficacy in reducing the trace size and time to trace. Using *SniP*, we analyze the potential benefit of incorporating stack-specific properties into memory persistence mechanisms for program stack.

1.4.4 Checkpoint based scheme for program stack persistence

Prosper is a hardware-software (OS) co-designed checkpoint-based approach for providing program stack persistence in a hybrid memory system. For process persistence, one of the crucial components of the execution state of any process is its memory state, which consists of mutable stack and heap segments. The properties that differentiate the program stack from the heap are its unique grow/shrink usage pattern and activation record write characteristics. Moreover, the stack is used in a programmer-agnostic manner; the compiler uses the support provided by the underlying ISA to use the stack, and the OS manages the memory used by the stack region in an on-demand fashion. Additionally, even though the stack size is smaller than the heap, *the number of operations* on the stack can be significant for some applications. *Prosper* considers these stack properties and tracks stack changes at sub-page byte granularity in hardware, allowing symbiosis with OS to realize efficient checkpointing of the stack region. *Prosper* significantly reduces (on average $\sim 4\times$) the amount of data copied during checkpoint and improves the overall checkpoint time with minimum overhead (less than 1% on average). Integration of *Prosper* with existing state-of-the-art memory persistence mechanisms (such as SSP [25]) for heap provides $2.6\times$ improvement over solely using the state-of-the-art mechanism for the entire memory area persistence.

We summarize our contributions in the following.

- Empirical evaluation of performance overhead of architectural primitives used for persistent barriers in Intel x86-64 and Arm64 ISAs.
- Open-source hybrid memory framework with built-in process persistence, *Kindle*, for enabling research and prototyping ideas crossing hardware-software layers.
- Open-source stack tracing framework, *SniP*, for efficiently capturing stack accesses in multi-threaded applications and enabling dynamic stack analysis.
- Periodic checkpoint-based mechanism, *Prosper*, with hardware-software co-designed approach for persisting program stack.

1.5 Thesis Organization

Chapter 2 introduces the background material on the need for NVM data persistence mechanisms and details existing primitive and advanced data persistence mechanisms at the software and hardware layers. Chapters 3, 4, and 5 present an empirical analysis of architecture primitives for data persistence to understand the persistence overhead and factors influencing data persistence overhead, a hybrid memory framework with process persistence capability to prototype and steer process persistence research, and a framework for multi-thread program stack analysis to expose unique properties of the stack, respectively. Chapter 6 presents a memory persistence scheme for the program stack that handles properties specific to the stack to enable overall process memory persistence. Chapter 7 concludes the thesis with a discussion of possible future research directions of the thesis.

Chapter 2

Background

Using NVM for process persistence has associated challenges, and we have introduced these challenges in Chapter 1. Understanding existing approaches and mechanisms to address these challenges requires details on the context of NVM usage in a hybrid memory system with DRAM. In this chapter, we discuss the need to maintain a consistent state of data in NVM and a set of available support at ISAs such as Intel x86-64 and Arm64 for building mechanisms at primitive and advanced levels to ensure a consistent memory state in NVM. We further provide details on existing process persistence approaches, which aim to address the ephemeral nature of the process and the challenges in achieving process persistence in traditional operating systems due to the design approach of associating the lifetime of data with the process.

2.1 Importance of Data Consistency in NVM

Modern systems have multiple levels of caches, with the last-level cache (LLC) shared across cores. CPU caches offer significant performance improvements by leveraging both temporal and spatial locality of memory accesses and help to bridge the memory wall problem [59] to a large extent. However, the volatile nature of caches poses challenges when NVM is accessed for data persistence using LOAD and STORE instructions; caches lose data during a power failure or a crash. The challenge here is a possible inconsistency between the execution state (program counter, register values, etc.) and the memory state after a system crash.

```
0x1101 STORE(A, 1)
0x1102 STORE(B, 1)
0x1103 X = A+B
0x1104 if( X == 2 )
0x1105   JUMP OUT
0x1106 else
0x1107   JUMP ERR
```

Listing 2.1: Persistent Stores to NVM

Pseudo code in Listing 2.1 shows an example with stores to NVM addresses `A`, `B`, and a branch to output (`OUT`) or error (`ERR`) section based on the value of variable `X`, calculated as the sum of `A` and `B`. There can be two consistency issues if the system crashes before the cache line containing the updated value of `A` is written back to the non-volatile memory and the program counter (PC) points to the instruction following `Store B` (instruction address 0x1103).

1. At the time of application restart, assuming the PC restores to a point after the `Store A` instruction, the application *will not* see the updated value of `A` as it was in the volatile cache.
2. If the cache line corresponding to `B` is written back before `A` due to cache eviction reordering, at the time of restart, the application will find that `B` is updated while `A` is not, causing an incorrect calculation of `X = A+B` and taking error (`ERR`) branch in code listing 2.1.

```
0x1201 fn_PB(Addr){
0x1202   FENCE
0x1203   FLUSH Addr
0x1204   FENCE
0x1205 }
0x1206 STORE(A, 1)
0x1207 fn_PB(A)
0x1208 STORE(B, 1)
0x1209 fn_PB(B)
```

Listing 2.2: Usage of Persistent Barrier

The second one is a more serious issue as it makes any smart recovery process non-deterministic; the recovery process can not determine where to resume by analyzing the memory content at the time of application restart. Using a *persistent barrier* as shown in code Listing 2.2, the cache line corresponding to store address `A` is written back to the persistent domain, guaranteeing data availability in NVM in case of a system crash after store to `A`.

Implementation of a persistent barrier depends on the Instruction Set Architecture (ISA). In the next section, we look into the architecture primitives available in Intel x86-64 and Arm64 ISAs for constructing persistent barriers.

2.2 Architecture primitives for Data Consistency

A persistent barrier uses `flush` and `fence` instructions, as shown in the code listing 2.2. Wu, Kai, et al. [60] observed that the overhead of flushing a cache line depends upon whether the cache line is in a clean or dirty state. The authors also observed overhead in the concurrency of flushing cache lines, as concurrent flushes may create contention in the internal buffers of NVM device. Existing approaches for cache line flushing include flushing the cache line immediately after modification [16], flushing asynchronously using a thread, flushing multiple cache lines at the end of an interval, relying on natural cache line eviction [61], or bypassing cache [33]. Ribbon [60] improves cache line flush performance by controlling concurrency and proactive flushing of cache lines. Intel x86-64 and Arm64 provide various `flush` and `fence` instructions.

2.2.1 Intel x86-64

Intel x86-64 provides three cache-line flush instructions, `clflush`, `clflushopt`, and `clwb`, with subtle differences in their implementation. `clflush` invalidates the cache line containing the linear address from all cache hierarchy levels and performs write back of the dirty cache lines to the memory if required. Similar to the `clflush` instruction, `clflushopt` also invalidates the cache line containing the linear address from all levels of the cache hierarchy in the coherence domain and writes back data to memory if the cache line contains modified data at any level of the cache hierarchy [48]. `clwb` differs from these two instructions in that it retains the cache line in a clean state after writing back the modified cache line to memory. Therefore, `clwb` is beneficial for cases where future access to the data is expected. `clflush`, `clflushopt` and `clwb` are ordered by store-fence (`mfence` and `sfence`) instructions. `clflushopt` is an optimized variant of `clflush` as it is not ordered with respect to execution of other `clflushopt`, `clflush` and `clwb`. `clflushopt` is also not ordered with respect to younger writes to the same cache line being invalidated.

For example, while flushing two addresses `A` and `B` using `clflushopt` as `clflushopt(A)`, `clflushopt(B)` then flushing of address `B` could happen before `A`. So `clflushopt` allows concurrent flushing of multiple cache lines from a thread [47]. The persistent barrier in code listing 2.2 can be optimized in Intel x86-64 systems by removing `fence` before `flush` because `clflush` instruction orders with respect to writes and other `clflush` instructions. Similarly, the `clflushopt` and `clwb` instructions order with respect to older writes to the cache-line being flushed [48].

2.2.2 Arm64

Arm Cortex-A provides mechanisms to *invalidate* or *clean* a cache line. `civac` and `cvac` are two such operations provided by Arm64 to perform *invalidate* or *clean* on data cache. The `civac` instruction performs both *clean* and *invalidate* of a virtual address to the point of coherency, and `cvac` performs *clean* of a virtual address to the point of coherency; more information about Point of Coherency (PoC) is given below.

```
0x1201 fn_PB(Addr){
0x1202    dsb ish
0x1203    dc civac, Addr
0x1204    dsb ish
0x1205 }
0x1206 STORE(A, 1)
0x1207 fn_PB(A)
0x1208 STORE(B, 1)
0x1209 fn_PB(B)
```

Listing 2.3: Persistent Barrier in Arm64

Invalidation of a cache line clears the valid bit; *cleaning* writes the content of cache line to the next cache level or the main memory if the cache line is dirty. *Cleaning* clears the dirty bit and makes the content of a cache line consistent with the next level of cache or memory. The *invalidate* or *clean* operation takes either the virtual address or the set-and-way of the cache line. If we use a virtual address, Arm64 allows invalidation and clean operations at two points,

(i) Point of Coherency (PoC)

(ii) Point of Unification (PoU)

PoC is the point at which all observers see the same copy of a memory location, and PoU for a core is the point at which all operations of the core see the same copy of a memory location. Both `civac` and `cvac` operates at PoC [49].

For implementing persistent barrier corresponding to code listing 2.2 on Arm64, we can use data memory barrier (`DSB ISH`) as fence [49] along with `civac` and `cvac` operations. Code listing 2.3 shows persistent barrier on Arm64 using `civac`.

2.3 Advanced Techniques for Data Consistency

A persistent barrier using `flush` and `fence` combination ensures consistent NVM update for a memory address. However, ensuring consistency for a code section performing multiple NVM writes, such as adding a new element into a linked list or rotating a red-black tree for height balancing requires advanced memory persistence techniques. Failure atomicity (FASE [18]) for memory writes in code listing 2.2 can be ensured by using a transaction scheme with `begin_tx` and `end_tx` semantics as in code listing 2.4. The FASE ensures that either NVM locations `A` and `B` are updated with the latest value or none of them are updated.

```
begin_tx
STORE(A, 1)
STORE(B, 1)
end_tx
```

Listing 2.4: Usage of FASE

We can broadly partition a failure atomic transaction into four phases, as shown in listing 2.5. The *begin_tx* performs operations that prepare for a new transaction based on the underlying memory persistence scheme, assuming a scheme based on dirty bit, these operations involve clearing dirty bits in page table entries. The *body* contains loads and stores within a transaction and associated operations, such as setting dirty bits in the page table entries for modified pages. The *end_tx* performs operations that consume the metadata populated during the transaction and associated operations to ensure consistency, such as reading dirty bits in page table entries and copying modified pages from DRAM to NVM in a hybrid memory setup. The *recover_tx* phase only comes if a transaction aborts due to system failure. Listing 2.5 shows the total performance overhead of a transaction if it succeeds or fails.

TABLE 2.1: Memory persistence mechanisms for NVM

Scheme	**Usage Granularity**	**Existing Works**
	Software Approach	
Logging	Data Structure	AsymNVM [62], PMAlloc [63], InCLL [21], NV-Heaps [30]
	Transaction	Rewind [64], HPTPM [65], Mnemosyne [22]
Shadowing	Data structure	MOD [66], SoftPM [67], CDDS [68]
	Transaction	DudeTM [69]
Checkpoint	Application	NVM-checkpoints [70], Aurora [1], Libcrpm [71], ResPCT [72],MemSnap [73]
	System wide	Exascale [74]
Cache line flush/bypass	Data Structure	NoveLSM [40], WAlloc [43]
Dual Copy	Transaction	Romulus [26]
Recompute	Data structure	CLBC [75]
	Hardware-Software Approach	
Logging	Transaction	NV-HTM [76], SpecPMT [77], Proteus [20]
Shadowing	Transaction	SSP [25]
	Hardware Approach	
Logging	Transaction	Silo [78], Atom [9], LOC [79], HOOP [29], ASAP [80]
	Whole system	Capri [81]
Checkpoint	Application	ThyNVM [82]
	System wide	Dual-page [83]

```
failure_atomic_tx_phases()
  begin_tx → operations to prepare for new tx, incurs overhead W
  body → loads and stores inside tx, incurs overhead X
  end_tx → operations to finish current tx, incurs overhead Y
  recover_tx → recovery operations for current tx, incurs overhead Z

failure_atomic_tx_overhead()
  if completes:
    Total_overhead →W + X + Y
  if aborts:
    a →% of loads and stores completed before failure
    Total_overhead →W + a*X + Z
```

LISTING 2.5: Components and overhead of failure atomic memory section

Maintaining a consistent NVM state for a group of NVM updates requires advanced techniques built around architectural primitives for cache line flushing and memory store ordering. These advanced techniques can be employed at different levels of usage granularity, such as data structure [12, 14, 30, 40, 41, 62, 64, 66–68, 84–90], memory allocator [42, 43, 63], file system [91–95], or transaction [65, 69, 76, 96].

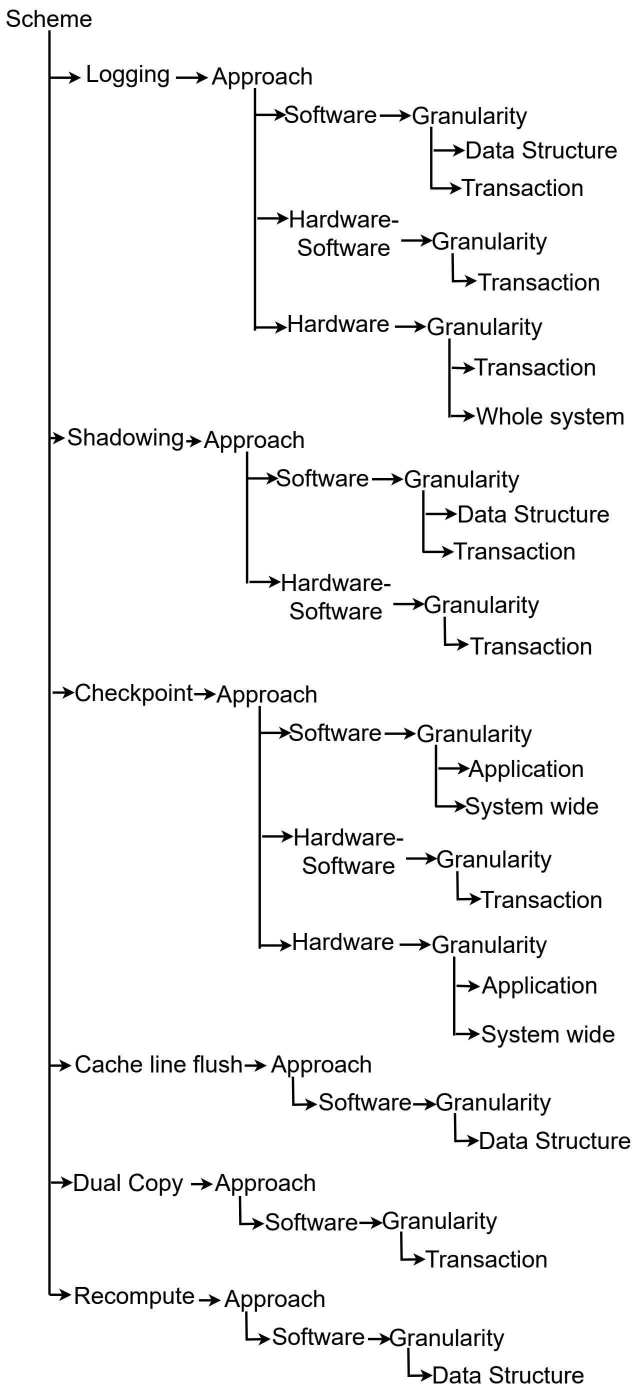


FIGURE 2.1: Broad classification of Memory persistence mechanisms for NVM

Figure 2.1 shows the broad classification of memory persistence mechanisms based on schemes such as logging, shadowing, and approaches at hardware, software, and hardware-software layers. Figure 2.1 also shows the granularity at which persistence mechanisms provide memory persistence guarantees, such as data structure, application, and system-wide. Table 2.1 expands on classifications in Figure 2.1 with a list of state-of-the-art memory persistence strategies under each classification in Figure 2.1. For example, NV-Heaps [30] ensures memory persistence at the data structure level using a logging scheme.

In summary, the advanced memory persistence mechanisms can be classified based on the granularity of usage, such as data structure or transaction, based on the underlying scheme for achieving consistent memory updates, such as logging or shadowing, based on the approach that is used to monitor updates to memory locations, i.e., dirty data monitoring.

2.3.1 Classification Based on Usage Granularity

We can employ memory persistence mechanisms at different levels of usage granularity, such as data structure, application, transaction, or system-wide. Mechanisms at data structure granularity ensure consistency for operations that modify a data structure by inserting, updating, or deleting elements. These mechanisms provide consistency for a specific data structure. Similarly, mechanisms at the application granularity ensure a consistent memory state for the application that may contain multiple data structure updates. The system-wide granularity persistence mechanism ensures a consistent memory state for all applications on the system. Mechanisms at the transaction granularity provide consistency for memory operation in a section of code demarcated between begin and end of the transaction.

Data Structure Granularity

Mechanisms at data structure granularity ensure consistency for different operations such as insert, update, and delete on a specific data structure. For example, Guerra, Jorge, et al. [67] proposed SoftPM, a lightweight container abstraction for data structure persistence within conventional systems. The application developer creates a container and points the root structure of the container to the data structure, which needs to be maintained consistently. Kannan, Sudarsun, et al. [40] created a persistent key-value store, NoveLSM, based on log-structured merge tree (LSM). NoveLSM maintains mutable persistent *memtable* in NVM and allows parallel reads to multiple levels of LSM. The benefits of using NVM in NoveLSM are, (i) *memtable* in NVM reduces serialization and deserialization cost of data compared to keeping it in secondary storage, (ii) enables direct updates to persistent *memtable* as NVM is accessible through the CPU load/store interface, (iii) reduces application stalls due to compaction, and committing changes to *memtable* directly avoids the need to maintain logs. Venkataraman, Shivaram, et al. [68] presented consistent and durable data structure (CDDS) that ensures consistent memory updates for data structures using versioning. CDDS creates a version for every update and tracks the latest version to return recent values on read. CDDS garbage collects old versions based on reference count. Intel released development kit for persistent memory, PMDK [47]. PMDK provides a set of libraries and tools for persistent memory programming.

Application Granularity

Memory persistence mechanisms at application granularity handle memory persistence for modifications in an application consisting of multiple data structures. Tsalapatis, Emil, et al. [1] proposed Aurora, memory persistence at application granularity using a single-level store [97], which eliminates the distinction between the transient and persistent state of an application. As the state of an application spans both user and kernel space, creating problems in capturing the application state, Aurora incorporates persistence as a service in OS to handle difficulties in capturing the application state. Libcrpm [71] is a library that improves the checkpoint performance of applications in NVM. It uses static instrumentation to capture memory changes at fine granularity. Libcrpm reduces the size of checkpoint data and the number of required fence instructions. Ren, Jinglei, et al. [82] proposed a software transparent memory persistence mechanism, ThyNVM, for hybrid memory systems with DRAM and NVM. ThyNVM uses a hardware-assisted periodic checkpoint mechanism. ThyNVM is a dual-scheme checkpointing mechanism, which checkpoints data at two granularities, cache line for sparse and page for dense writes, to reduce metadata overhead associated with checkpoints.

Transaction Granularity

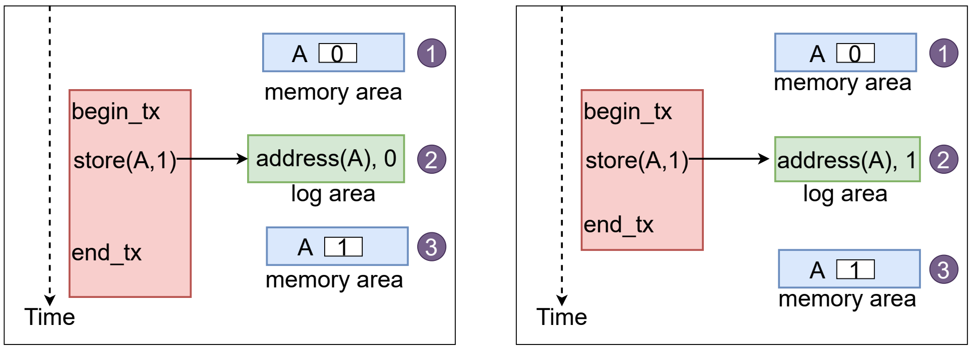
Memory persistence mechanisms at transaction granularity provide consistency for memory updates from a section of code demarcated as a transaction. Chatzistergiou, Andreas, et al. [64] proposed Rewind, a user library for managing updates in a transaction using logging. Mnemosyne [22] provides a durable transaction mechanism by converting regular code to transactions with compiler support. Mnemosyne uses a per-thread redo log to ensure consistent updates. Liu, Mengxing, et al. [69] proposed DudeTM, a shadow-memory-based approach for providing durable transaction updates. DudeTM captures transaction modifications in a shadow memory that acts as a redo log. It then flushes log entries from shadow memory to persistent memory and finally modifies the original data using the persistent redo log. DudeTM uses a redo log to transfer updates from shadow volatile memory to persistent memory. Atom [9] uses hardware-based undo logging to provide consistency for stores in a region marked between `atomic_begin` and `atomic_end`. Atom modifies the memory controller to ensure ordering between log and data write. Cai, Miao, et al. [29] designed a hardware-assisted mechanism for out-of-place update, HOOP, using redo logging. HOOP stores modified data in a dedicated NVM location and applies these changes to the actual memory location using a garbage collection scheme; it also uses an address redirection table to get updated values from the dedicated NVM region. Proteus [20] uses a hardware-software approach for durable transactions. Proteus introduced new instructions for logging, and the compiler augments each store inside a transaction with these log instructions. ASAP [80] uses a hardware undo logging scheme that allows atomic regions to commit asynchronously by tracking control and data dependencies between atomic regions in hardware. It maintains a dependency list in the memory controller that tracks active atomic regions and dependency between atomic regions. All dependencies of an atomic region are resolved before freeing undo log.

File-system Granularity

Memory persistence mechanisms at the file-system level expose NVM through the file-system layers, albeit using optimization to minimize software layer overheads. File-system approaches include techniques such as NOVA [94] and SplitFS [98]. NOVA is a log-structured file system for hybrid memory systems with NVM. NOVA maintains per inode log as a linked list of 4 KB pages and atomically updates the tail pointer on log append. NOVA uses a radix tree in DRAM to perform search operations and keeps log and file data in NVM. The log contains meta-data about changes and uses copy-on-write for file data. SplitFS [98] is built upon ext4 DAX [99], bypassing page cache and using memory mapping to access persistent memory. SplitFS consists of a user-space library file system for data operations and a kernel persistent memory file system (ext4 DAX) for handling metadata operations. SplitFS provides applications with transparency and offers a variety of crash consistency guarantees.

Whole System Granularity

Mechanisms at whole system granularity ensure persistence for all data updates in the system. Jeong, Jungi, et al. [81] proposed a compiler and architecture co-design approach, Capri, for providing persistence for the whole system at the region level. Capri compiler



(a) Undo Logging (b) Redo Logging

FIGURE 2.2: Write-Ahead logging mechanisms

defines regions in a program and inserts instructions for checkpointing registers. Capri uses a non-volatile proxy buffer in hardware to maintain data updates from in-progress regions. It combines undo and redo logging, allowing dirty cache line writeback to NVM. Dual-Page Checkpointing [83] is a hardware-based approach for whole system persistence by tracking memory changes at cache line granularity in the memory controller. Each virtual page is associated with two physical pages. It uses two-bit vector per cache line to track whether a cache line is dirtied and the location of dirty cache lines in these physical pages.

2.3.2 Classification Based on Scheme

Memory persistence mechanisms can use different underlying schemes such as write-ahead logging (WAL), copy-on-write (CoW), shadow paging [25, 100, 101], or dual copy [26] to ensure consistent memory state.

Write-Ahead Logging

Write-ahead logging (WAL) creates a log entry before modifying the memory location. The values saved in a WAL depend upon the logging variation; it can be either the old value (*undo logging*) before modification or the new value after modification (*redo logging*). Figure 2.2 shows the working of a failure-atomic transaction (marked between `begin_tx` and `end_tx`) using a WAL-based mechanism.

The undo log mechanism stores the old value (in a log area in the NVM) prior to the modification and discards the log if operations in the atomic section are completed successfully (Figure 2.2(a)). A memory location is modified in place after persisting the log record. In case of a failure, the recovery mechanism uses values in the undo log to revert changes. Read operations inside an atomic section access values directly from the corresponding memory location as the memory contains updated values. In the example shown in Figure 2.2(a), `store(A,1)` creates an undo log entry containing the address of `A` and the value of `A` prior to store (step ②). The application is allowed to modify the value of `A` inside the atomic section after persisting the undo log entry (using a persistent barrier), as shown in Figure 2.2(a). In redo logging, updated values are stored in the log during a transaction. Read requests for modified values from an application are served from the log area (Figure 2.2(b)).

The redo log entries persist during the transaction commit, and modifications are applied to the memory location from the log either synchronously during the commit or asynchronously after committing the transaction. In case of a failure, the recovery mechanism re-performs operations in the redo log for committed but not applied transactions. The recovery mechanism discards logs for an uncommitted transaction since the memory locations still have old, consistent values. In the redo logging example shown in Figure 2.2(b), `store(A,1)` creates a redo log entry containing the address of `A` and updated value of `A` (step ②). Transaction mechanism updates the value of `A` during the commit operation after persisting the redo log, as shown in Figure 2.2(b) (step ③).

Undo and redo logging mechanisms present different trade-offs; transaction commit is faster with undo logging since the memory locations are modified in place, whereas redo logging requires applying changes during commit. With undo log, overhead referred to as `X` in components of a transaction listing 2.5 is expected to be higher than redo since each write requires persisting the log before the memory location is modified, whereas overhead referred to as `Y` in listing 2.5 is expected to be lower for undo compared to redo. Failed transaction recovery is faster for redo logging since the memory location is not modified until the transaction is committed, whereas undo logging requires reverting modifications using log entries during recovery, indicating a lower overhead referred to as `Z` in listing 2.5 for redo. There is also a difference in the number of persistent barriers required in undo and redo logging techniques; undo logging requires a persistent barrier after each log write, while redo logging requires only one persistent barrier for all log entries during the commit [18, 102]. Undo logging performs better for workloads with more reads and is more sensitive to the read-write ratio, as observed by Hu et al. [102].

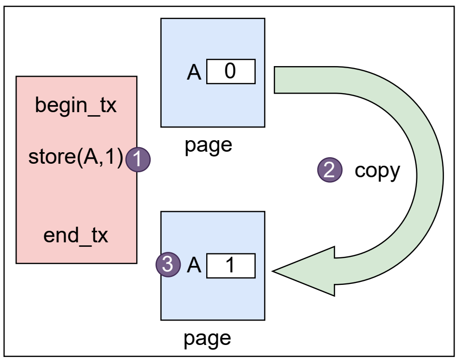


FIGURE 2.3: Copy-on-write mechanism

The redo logging incurs the additional overhead of redirecting reads to the log area to get the updated value, contributing to overhead referred to as `X` in listing 2.5. Both redo and undo logging mechanisms cause write amplification and lead to cache pollution because of additional memory operations.

Shadowing

Shadowing-based schemes create a consistent copy of data before modification. Shadowing can be at different levels based on the granularity of copying, such as page, object, or cache line. Compared with write-ahead logging schemes, shadowing or copy-on-write (CoW) at page granularity (shadow paging) makes a copy of the page on the first write to the page for preserving old content, and subsequent writes in the failure atomic section have no additional overhead. Figure 2.3 shows that `store(A,1)` (step ①) creates a copy of the page containing `A` (step ②), and the value of `A` is updated in the new page (step ③). The copy operation is performed only on the first write, and future writes to the page containing `A` in the transaction are directed to the new page.

One general approach to implement shadow paging in conventional systems is by marking pages as read-only to raise page fault on first write and then copy page and update page table mapping to point to newly copied page [103, 104]. These newly copied pages remain the final valid page if the failure atomic section commits, i.e., on a successful transaction, we can discard pages from which copies are created (old pages). On failure, page table mappings are reverted to old pages and new pages are reclaimed.

The CoW-based mechanism avoids extra operations required to create and manage logs in a logging scheme but with a drawback of write amplification if the modification size is less than half of the page size [105], indicating a higher overhead referred to as `X` in listing 2.5.

In addition to ensuring a consistent memory state in a transaction, the copy-on-write mechanism at page granularity is also used for fork system calls, data deduplication, and snapshotting [103].

Checkpointing

Creating failure atomic section using checkpointing [74, 83, 106] requires saving the state of an application into a persistent device for fault tolerance. We can perform checkpointing at the full-system level or specific to an application. Multi-level checkpoint methods combine frequent local checkpoints with infrequent remote checkpoints for better checkpointing performance [70]. As an alternative to checkpointing, a recompute-based method recovers from a failure by recomputing parts that are not complete, avoiding the need to restore a copy from the checkpoint or log [75]. Checkpointing may generate a large volume of data, and to reduce the volume of data, checkpointing can be performed incrementally [70, 74]. Incremental checkpointing involves monitoring state changes and copying changes to a persistent device, contributing to overheads referred to as `X` and `Y` in listing 2.5.

For checkpointing memory, conventional systems allow monitoring memory state changes at page granularity using virtual address translation infrastructure [104]. Zhang et al. [107] enable sub-page granularity tracking by placing objects on different virtual pages and using the page-protection mechanism to identify modified objects in a checkpoint interval. In their approach, objects in different virtual pages are allocated on the same physical page to avoid physical memory wastage. Libcrpm [71] is a software approach for incremental checkpointing that divides the NVM area into main and backup regions to save the *working* and *checkpoint* states of objects. The main and backup regions are further divided into segments (2MB each) and blocks (256B each) for meta-data maintenance. Libcrpm uses static instrumentation to track memory changes for recording memory dirty information and initiating copy-on-write on the first write to the main region segment in an interval. In Libcrpm, first write to a segment triggers the copying of dirty blocks from the main to the associated segment in the backup region and also adds the segment to the list of dirty segments. At the end of a checkpoint interval, modified blocks in each segment from the list of dirty segments are flushed to memory using a persistent barrier consisting of `clwb` and `sfence` constructs. Wu, Song, et al. [83] provide hardware-based checkpointing at cache line granularity using two physical pages for a virtual page. These two physical pages store checkpoint and working data and use two-bit vector per cache line to denote the page with working/checkpoint data and dirty/clean status.

2.3.3 Classification Based on Memory Update Monitoring

We can further categorize these existing memory persistence mechanisms for failure atomicity as *tracking* and *non-tracking* based on their method for monitoring and capturing memory updates.

Mechanisms with Memory Access Tracking

In tracking-based mechanisms, memory modifications are tracked in the hardware and/or software layers for periodic checkpointing. Aurora [1] tracks memory changes using the per-page dirty bit [48] to provide checkpoint-based process persistence in the OS layer by incrementally saving various subsystem states associated with a process, including memory. Kona [108] proposes tracking memory modifications at cache line granularity using a memory exposed through FPGA. Improvements in memory dirty tracking techniques using software and hardware enhancements ([104, 108–110]) attempt to reduce the dirty tracking overhead or support dirty tracking at a finer granularity (e.g., 64 bytes). OS-driven checkpoint solutions require tracking at a finer granularity for efficiency. Moreover, OS-level checkpointing requires flexibility in terms of programming and orchestrating additional hardware support from the OS layer.

Mechanisms without Memory Access Tracking

Non-tracking techniques employ two main approaches, logging [22, 23, 72, 77, 78, 102] and shadow paging [25, 100]. SSP[25] is a shadow paging-based mechanism at cache line granularity that redirects modifications to two different physical pages using hardware-assisted cache line remapping and consolidates these pages using a background OS thread. ATOM [9] uses hardware-based undo logging and manages log allocation, ordering, and truncation in the hardware. InCLL [21] is based on in-cacheline undo logging to provide fine-grained checkpointing. Compiler-assisted techniques can add log instructions for each store inside a transaction and use special log registers to improve the performance of log-based checkpoint solutions. Shin et al. [20] use hardware support to order log write and data update operations to realize an efficient compiler-assisted logging solution. LOC [79] uses logging by extending the CPU load/store interface and cache, ensuring memory persistence through a relaxed order of writes within and between persistent memory transactions (PTM). Capri [81] modifies the compiler to assist the hardware in maintaining undo+redo logs in a targeted fashion. HOOP [29] uses hardware-based redo logging, while JUSTDO

[31] is a software logging approach that minimizes log size by storing only the most recent store instruction executed within an atomic session. ThyNVM [82] provides a hardware-assisted dual checkpointing scheme for DRAM+NVM hybrid memory that dynamically decides the checkpoint size to reduce overhead. These software, hardware, or software-hardware combined approaches under non-tracking require non-trivial operation during a persistence interval.

2.4 Achieving Process Persistence

We can use FASE [18] mechanisms to maintain a consistent memory state for process components to achieve process persistence, allowing resumption from its last saved consistent state. The state of a process includes multiple components such as CPU registers, memory, execution context in the OS, etc. Traditional OSes use the classical process/thread execution model. In these OSes, data and pointer references to the data last till the lifespan of a process [111]. This process-centric design requires special mechanisms and support from OS to persist the state of a process in a crash-consistent manner.

Existing approaches to provide process persistence aim to tackle the transient nature of a process by proposing new abstractions for a process in NVM or using a data-centric approach in which data is treated as a first-class citizen by having its lifespan beyond the process that creates it. NV-Process [112] follows the former approach to provide a fault-tolerant process abstraction using NVM by decoupling the notion of processes from OS. In conventional OS, the process state is intermingled with the OS state, making segregation of the process state from OS difficult. This entanglement destroys the state of a process and associated meta-data on OS reboot. Therefore, NV-process maintains the process state independently from the OS state. Twizzler [113] follows the latter approach and proposes a data-centric OS for NVM, extending the design principles of Grasshopper[97], a single-level store OS. Twizzler aims to minimize OS intervention in persisting data and disassociates persistent data from ephemeral virtual memory context. For this, Twizzler divides NVM into persistent objects within a global object space and uses a library OS to map objects into address space. The user and kernel space share the address space layout (view object), and the user-space maps objects into this address space at a specific location, kernel then reads this view object to define the required virtual address layout. Twizzler also enables persistent pointers that remain valid across reboots; persistent pointers are represented using offsets inside objects. The persistent object model of Twizzler is motivated by the object model in NV-heap [30]. NV-heap implements persistent objects (Non-volatile

Memory Heap) and handles persistent pointers by ensuring that there are no pointers from non-volatile heap to volatile data and from one non-volatile heap to another non-volatile heap. NV-heap maintains a root; all objects reachable from the root are persistent. As Twizzler and NV-heap aim to provide persistent data objects, adapting them for process persistence is non-trivial.

This thesis proposes a checkpoint-based scheme for process persistence in a hybrid memory system with DRAM and NVM. As process persistence requires consistently maintaining process states, we show the overhead of using architecture primitives through persistence barriers in achieving data persistence for common data structures associated with process states in OS. We create a hybrid memory framework with process persistence capability to explore research directions in hybrid memory systems about persistence and capacity. We showcase that persistence schemes for maintaining a process's memory state require techniques specialized for the memory area under consideration by exposing peculiar characteristics of the stack area through an efficient stack tracking framework. We then propose a checkpoint-based memory persistence scheme for the stack that handles unique properties of the program stack and provides lower overhead than using other advanced memory persistence techniques for the program stack.

In the next chapter, we study the performance overhead of different hardware primitives and logging-based mechanisms to achieve NVM consistency on Intel x86-64 and Arm64 systems.

Chapter 3

Empirical Analysis of Architectural Primitives

Byte addressable Non-Volatile Memory provides persistent memory semantics and large memory capacity with access latencies comparable to volatile memory (DRAM). The persistent semantics offers an attractive alternative to meet data persistence requirements of applications' without relying heavily on off-the-chip persistent storage devices (e.g., HDDs and SSDs) and complex storage management middleware such as file systems [5]. The traditional application programmer interfaces need a revisit to leverage the benefits of NVM by storing the data directly in NVM, avoiding any serialization requirements [114]. However, the transition from volatile memory to non-volatile memory presents some unique challenges. One of the major research challenges is to design specialized techniques to provide *consistent* memory state for applications using NVM across system restarts as mentioned in § 1.1. Considering the intricacies of processor-memory data path organization, traditional file system or database techniques (e.g., journaling and transactions) may require non-trivial adaptation. To provide *consistency* and *atomicity* guarantees for the applications using NVM, several techniques are proposed [7–10].

To leverage the full benefits of the NVM systems, applications can store and access data in the form of in-memory data structures residing in the NVM. The root cause of inconsistency arises in the event of a system crash where hardware storage elements such as caches and other volatile micro-architectural components lose in-flight memory modifications, as mentioned in § 2.1. There can be two major repercussions in the above scenario,

(i) The system is left in a corrupt memory state

(ii) The expected program execution state is different from the NVM state.

One of the approaches used to address this inconsistency is to allow the application developer achieve a guaranteed temporal order for moving data into the persistent domain (i.e., NVM) [38]. With this approach, the application developer is guaranteed to see the updated memory content even after an abrupt system reboot caused by software/hardware failure. *Persistent barrier* is one of the fundamental techniques to achieve NVM consistency in an efficient manner. The application programmer (or the NVM support library) inserts persistent barriers at appropriate places to ensure data propagation to the NVM and achieve crash consistency as mentioned in § 2.1. In this chapter, *we analyze the performance implications of different architectural primitives* for NVM consistency. This study aims to guide application, support library, and middleware developers in choosing the appropriate architecture primitives based on their performance objectives and memory access patterns to achieve NVM consistency. We also show other techniques like controlling the data cache behavior using page table attributes (e.g., PAT in x86) for NVM consistency.

The primitive architectural artifacts such as cache flush and memory fences are used to realize persistent barriers in NVM systems [38]. Logging based techniques (similar to file system journaling) such as redo or undo logging [9, 19, 21, 31, 115] also depend heavily on the architectural implementation of persistent barriers as mentioned in § 2.3. Persistent barriers also play a key role in the data structures, memory allocators and memory management schemes proposed specifically for NVM [39–46]. Most instruction set architectures (ISAs) provide a set of instructions (e.g., `clflush` and `mfence` in x86 ISA) along with required extension of the persistent domain (e.g., battery powered memory controller a.k.a. Intel ADR [47]) to propagate changes to NVM in a consistent manner. We empirically analyze the performance overhead of different data consistency methods provided by Intel x86-64 (`clflush`, `clflushopt`, `clwb`) and Arm64 (`civac`, `cvac`) ISAs [48][49]. Through this empirical study, we answer following questions,

(i) What is the performance overhead of different consistency primitives for different ISAs (x86-64 and Arm64) with single and multi-threaded applications?

(ii) How does the memory footprint and access characteristics (read-to-write ratio) of different applications influence the performance overhead of different data consistency methods?

(iii) Transaction-based consistency mechanisms like redo and undo logging use the underlying architectural persistence barriers to achieve atomic update semantics. In this

TABLE 3.1: Microbenchmarks used in the experiments

Benchmarks	Description
BST	Binary Search Tree
BST_P	Parallel Binary Search Tree
CUH	Cuckoo Hashing Table [116]
QUE	Linear Queue
RBT	Red-black Tree [117]

context, an interesting question is: What is the comparative performance overheads for redo and undo logging schemes with different data consistency methods?

Answering the above questions can provide guidelines for application/middleware developers to choose the right technique and account for the resource overheads during capacity planning. Moreover, the analysis can provide directions to improve the efficiency of architectural primitives for NVM consistency. We have used a set of workloads designed to operate on well known data structures and executed them on gem5 architecture simulator [50, 51] to analyze the performance implication by providing justifications through different architecture layer metrics.

3.1 Setup and Methodology

3.1.1 Benchmarks and Parameters

We have used micro-benchmarks implementing well-known data structures (Table 3.1) to study the performance overhead of different data consistency methods. We choose these data structures as they are commonly used in applications and operating systems to maintain different states. For example, Queues are extensively used to manage tasks, I/O requests, and network packets in operating systems. Similarly Red-black trees or other tree data structures are used to manage processes' virtual memory areas.

BST benchmark performs insert, delete and search operations on a binary search tree. One of the data consistency methods (passed as a parameter to the benchmark) is used after each insert and delete operation to push changes into the persistence domain. BST_P is a parallel implementation of binary search tree using `POSIX` threads to parallelize search and update operations where a read-write lock is used for synchronization across the three operations.

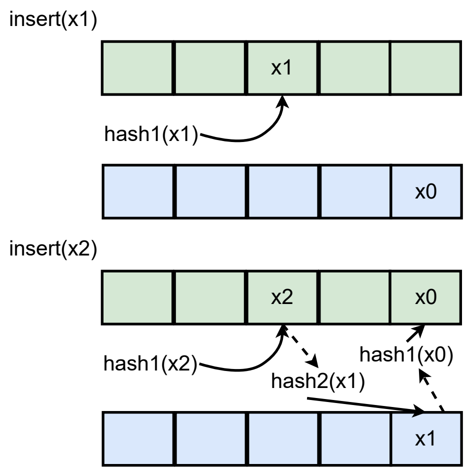


FIGURE 3.1: Insert operations on Cuckoo Hashing data structure.

CUH benchmark uses cuckoo hashing [116], a dictionary data structure with constant worst case lookup time. Cuckoo hashing uses two hash tables, with two hash functions `hash1` and `hash2`, respectively. Every key `x` is stored in the cell `hash1(x)` of the 1st table or the cell `hash2(x)` of the 2nd table. During insertion of any key `x`, if the cell `hash1(x)` is free, then key is inserted there. Otherwise, the previous occupant of the cell `hash1(x)` becomes "nestless" after `x` is inserted in that position. The nestless key is inserted into the 2nd table by following the same procedure. This process continues until the nestless key finds a free slot or reaches "MaxLoop" count. Reaching "MaxLoop" count results in resizing of the hash table. Note that, an insertion may cause multiple keys to become "nestless". Figure 3.1 shows an example of inserting keys x1, x2 in a cuckoo hash table. Insertion of x1 finds an empty cell in 1st table, whereas inserting x2 moves the previous occupant of the cell in 1st table to 2nd table which necessitates moving x0 to 1st table. We use data consistency methods for each cell update in the hash tables.

QUE benchmark is based on linear queue with a head and a tail pointer providing enqueue and dequeue operations. Enqueue operation adds a new element at the rear end and updates the tail pointer. The enqueue operation ensures that both tail pointer and newly added item reach persistence domain by using one of the data consistency methods. Dequeue operation removes an item from the front and updates the head pointer while ensuring that the head pointer updation reach the persistent domain.

RBT benchmark performs different operations on a height balanced binary search tree (red-black tree) with *red-black properties*—(i) each node is either red or black, (ii) both children of a red node are black, (iii) path from a node to descendant leaves have same number of black nodes, and, (iv) root and leave nodes are black. An insert or delete

TABLE 3.2: Working set sizes used for experiments on gem5

Type	Size Description	Inserts	Deletes	Searches*
Tiny	0.90 x L1-D Size	460	400	600
Small	0.90 x L2 Size	7370	4000	6000
Medium	0.90 x LLC Size	29490	4000	6000
Large	4 x LLC Size	131070	4000	6000

*Queue does not support search operations

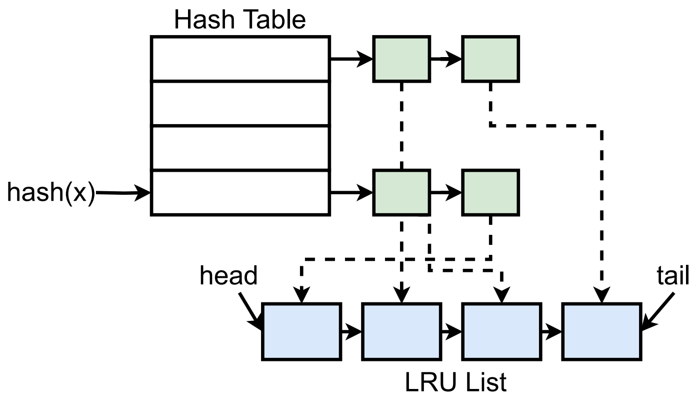


FIGURE 3.2: Schematic diagram of LRU micro-benchmark.

operation may violate the red-black property, thus requiring change of color for multiple nodes or rotation operations to restore the red-black properties. RBT uses one of the data consistency methods when the tree is updated to ensure data persistence.

To study the performance implications with different cache working set size, we have used four variants— *tiny*, *small*, *medium* and *large* (Table 3.2), as parameters for the experiments. Table 3.2 shows the number of insert, delete and search operations performed under each working set size variant; insert and delete operations maintain the working set size (mentioned under the size description) of the micro-benchmarks. For example, in case of *tiny* working set, size of micro-benchmarks always fits into L1 data cache (L1-D) while performing insert and delete operations.

3.1.2 Redo and Undo Logging

As discussed in § 2.3, the undo-logging technique records old values in the log area and discards them on successful transactions. In contrast, the redo-logging technique records new values in the log area and applies them to memory locations on successful transactions. To study the influence of different data consistency primitives with redo and undo logging schemes, we use a micro-benchmark as shown in Figure 3.2 that maintains the LRU order

Table 3.3: Gem5 configuration

CPU	Out-of-order CPU
L1-D/I	32 KiB/core (8 way)
L2	512KiB/core (16 way)
L3	2 MiB/core shared (16 way)
MSHRs	16, 32, 32/core at L1-D, L2, L3
Cache data access latency	2, 9, 15 cycles at L1-D, L2, L3
Cache line size	64 B in L1, L2, L3
Replacement policy	LRU
L1 prefetcher	StridePrefetcher with degree=4
Memory controller	Nvmain*[118]
Memory	PCM with configuration based on [119]
Memory capacity	10 GB (20GB for § 3.3.1)

*gem5 NVM Interface [51] used for results in § 3.3.1

of objects in the presence of different access patterns. The LRU micro-benchmark consists of a linked list of objects where the recently accessed object is maintained at the head of the list. The implementation consists of a hash table to speed up the access where an entry in the hash table index points to an object in the LRU linked list. The hash table resolves collision at an index using chaining. Each object in the LRU list also contains a fixed-size data field. For accessing an object, the hash table entry is used to get the object's index in the LRU linked list, and the object is moved to the head position of the list after access. To ensure failure atomicity, undo or redo logging is implemented as a separate linked list of entries, with each entry consisting of a <address, value> pair, where the *value* is a pointer to a memory address of configurable size to support different log values. In our redo implementation, read access is implemented by looking up the item in the LRU data structure. Our study differs from Wan et al. [102] as we have compared the performance with different persistent barrier primitives while Wan et al. [102] used redo and undo logging with Intel PMDK framework.

For the experimental evaluation, we have configured gem5 [50] with parameters in Table 3.3.

3.1.3 Why do we focus on flush-based data consistency methods?

Even though cache line flushing is the most common approach to ensure data consistency in NVM, we can design other alternative mechanisms using cache bypassing, write-through caches, or non-temporal stores [120]. To understand the behavior of these alternatives, we compared the performance overhead of uncacheable (UC) and write-through (WT) [48]

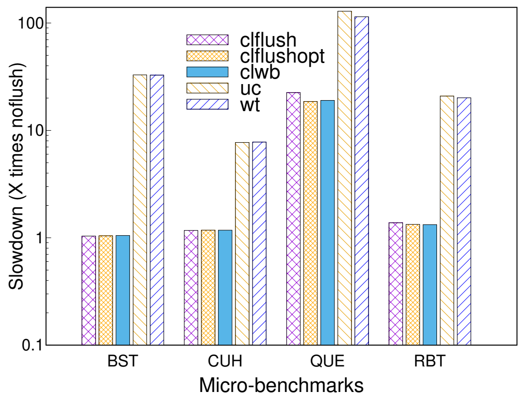


FIGURE 3.3: Performance overhead of data consistency methods. Y-axis values are slowdown with respect to noflush (not using any data consistency method).

memory against different cache line flush-based data consistency methods. We used a set of micro-benchmarks (Table 3.1), with LLC thrashing working set size, on an Intel Xeon(R) 3.20 GHz system with L1-D/I 32KB (8 way), L2 1MB (16 way) and L3 8MB (11 way), with Ubuntu 18.04.3 LTS (kernel 4.19.13).

We have used Page Attribute Table (PAT) of x86-64 systems to set the memory type as UC or WT. We augmented the `mmap` system call in the Linux kernel (v4.19.13) to introduce a special flag (`MAP_SENSITIVE`) to allow the user space to control the caching behavior. Depending on the value of the flag passed from the user space, any given virtual address range is mapped as UC or WT by configuring the page table entry with appropriate PAT value [48]. Figure 3.3 shows the performance overhead of different data consistency methods normalized to `noflush`. The performance overheads with UC and WT is significantly higher (between 5$\times$ - 31$\times$) compared to the cache line flush based data consistency methods. The results clearly demonstrate the benefits of cache flushing-based techniques, and therefore, we focus on studying different cache line flushing-based data consistency methods in this chapter.

3.2 Evaluation of data consistency primitives

In this section, we study the performance overhead of different data consistency methods with Intel x86-64 and Arm64 systems using the micro-benchmarks in Table 3.1. To analyze the performance implication of working set size with different levels of cache occupancy, we vary the working set size by configuring the benchmark parameters as mentioned in Table 3.2.

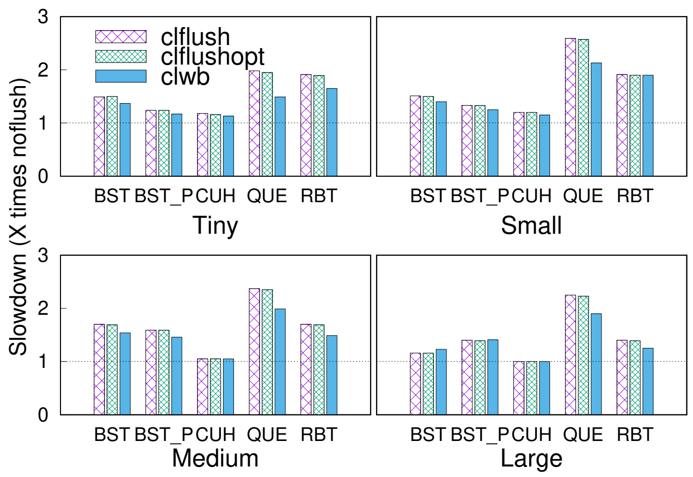


FIGURE 3.4: Performance slowdown with different data consistency methods in gem5 simulation of x86-64 system. Y-axis values are slowdown normalized to `noflush` (lower the better).

3.2.1 Performance with Intel x86-64

We analyze the impact of persistent barrier methods on the performance of data structures on a simulated gem5 system with configuration in Table 3.3 and on a real system with Intel Optane persistent memory module [5] using micro-benchmarks in Table 3.1.

(A) Performance on simulated gem5 system:

Figure 3.4 shows the performance slowdown of different micro-benchmarks with different data consistency methods. The slowdown is the ratio of completion time with different data consistency methods to `noflush` (i.e., not using any data consistency method). Memory allocation for `noflush` happens from NVM in the simulated system. The result shows that `clflush` and `clflushopt` have similar slowdown across all working set sizes with all workloads. This similarity in slowdown is primarily due to the inevitable requirement of ordering the flushes to ensure NVM consistency, which negates the optimizations (i.e., concurrency of flush operations) provided by `clflushopt` (Refer § 2.3 for details). With `clwb`, the performance overheads are lower (by 1× - 1.3×) compared to both `clflush` and `clflushopt` depending on the benchmark and working set size. The performance overheads of different flush methods vary between (1× - 2.5×) compared to `noflush`, where QUE benchmark results in the highest performance overhead for all working set sizes and CUH has the lowest for all working set sizes.

For QUE micro-benchmark, with `clflush` and `clflushopt`, each insert results in two additional cache misses

(i) cache miss while updating the next pointer of the previously inserted element.

(ii) cache miss while updating tail pointer.

The delete operation also results in additional cache misses while updating the head pointer. By taking large working set size as an example, we study the cache misses at L1-D for QUE with `clflush` and `clflushopt` compared to other benchmarks (Table 3.4). QUE also results in the highest MPKI (misses per kilo instruction) based on demand misses for CPU data at LLC for `clflush` and `clflushopt` as shown in Table 3.4. It is interesting to note that, QUE with `clwb` results in a significant slowdown (Figure 3.4) even though the MPKI at LLC is lower compared to other techniques (Table 3.4). This slowdown in QUE with `clwb` relates to the delay in committing instructions at the reorder buffer (ROB) head. Table 3.4 reports number of times committing an instruction that reaches the ROB head has to stall because the instruction has not finished execution. The number of such stalls is high for `clwb` and all other data consistency methods in comparison with `noflush` as shown in Table 3.4. Therefore, the presence of *flushes and memory fences to order flushes* contributes significantly towards the overall performance overhead. The highest number of commit stalls at ROB head is for `clflush` and decreases by 33% to 50% with `clflushopt` or `clwb` for all micro-benchmarks except for BST_P. BST_P experiences high number of commit stalls at the ROB head even with `noflush` due to the usage of locks.

The minimum performance overhead of CUH with different data consistency methods can be correlated with its similar miss rate at L1-D and similar MPKI values at LLC (Table 3.4). The similar cache miss behavior can be attributed to the number of keys becoming "nestless" during inserts, which remains same across all techniques. The L1-D miss rate and LLC MPKI indicates that, CUH benchmark do not exhibit high temporal locality, and therefore, the performance impacts due to cache line flushing is negligible. RBT has a higher performance slowdown than BST with different flush methods, as inserting or deleting elements in the tree may create multiple changes to balance the tree's height. RBT has 1.4× higher LLC MPKI for `clflush` and `clflushopt` than `noflush`.

There is a decrease in slowdown (with all flush methods) between medium and large working set sizes, as shown in Figure 3.4. An increase in time taken for `noflush` can result in a comparatively lower slowdown because we calculate the slowdown in a relative manner (w.r.t. the `noflush` performance). As the medium working set benchmark fits into

Table 3.4: Cache miss and ROB stall behavior with large working set

Methods	BST	CUH	QUE	RBT	BST_P
LLC MPKI					
noflush	1.74	1.68	5.23	2.56	0.73
clflush	1.79	1.68	20.42	3.79	1.16
clflushopt	1.79	1.67	19.27	3.77	1.15
clwb	1.69	1.66	0.69	2.49	0.71
L1-D miss rate (in %)					
noflush	6.23	0.65	2.63	6.39	12.56
clflush	5.97	0.63	7.89	6.52	12.33
clflushopt	5.96	0.63	7.55	6.51	12.65
clwb	6.03	0.63	3.11	6.08	12.73
#commit stalls at ROB					
noflush	6	34	6	161	1091245
clflush	798,423	3,348,775	536,286	1614611	1489288
clflushopt	532,284	2,232,528	270,146	779187	1356583
clwb	532,284	2,232,528	270,146	779187	1351100

the LLC while the large working set does not, `noflush` results in better performance with medium working set by maximizing the benefits of temporal locality of memory accesses. Therefore, there is a decrease in the relative slowdown with large working set size compared to that of medium working set size. We can confirm the change in `noflush` performance under medium and large working set sizes by comparing the MPKI at LLC with `noflush` and `clflush` methods by taking BST benchmark as an example. The MPKI values at LLC for medium working set size are—0.22 for `noflush`, 1.04 for `clflush`, 1.04 for `clflushopt` and 0.22 for `clwb`, as shown in Table 3.5. All benchmarks show higher LLC MPKI with `clflush` and `clflushopt` compared to `noflush` for medium and small working set types. The MPKI at LLC for `clflush` compared to `noflush` is 4.7× and 0.02× for medium and large working sets, respectively, resulting in higher relative performance overhead for medium working set in Figure 3.4. The cache-friendly nature of benchmarks decides the comparative performance slowdown for different persistent barrier mechanisms.

(B) Performance on real system:

We examined the impact of persistent barrier primitives on a real x86-64 system with NVM using micro-benchmarks in Table 3.1. As the baseline in this study, micro-benchmarks allocate memory from volatile DRAM instead of NVM and do not flush cache lines after modification. With this baseline, the results mainly signify the performance overhead after migrating applications from volatile DRAM to NVM for persistence. Figure 3.5 shows the performance overhead of different data consistency methods on a two-socket

Table 3.5: LLC MPKI with Medium and Small working set

Methods	BST	CUH	QUE	RBT	BST_P
Medium					
noflush	0.22	0.9	4.76	0.92	0.18
clflush	1.04	1.06	19.97	2.13	0.77
clflushopt	1.04	1.04	18.87	2.11	0.8
clwb	0.22	0.88	0.54	0.5	0.17
Small					
noflush	0.11	0.19	3.86	0.14	0.1
clflush	0.55	0.39	19.03	0.92	0.36
clflushopt	0.52	0.39	18.03	0.91	0.36
clwb	0.11	0.19	0.5	0.91	0.09

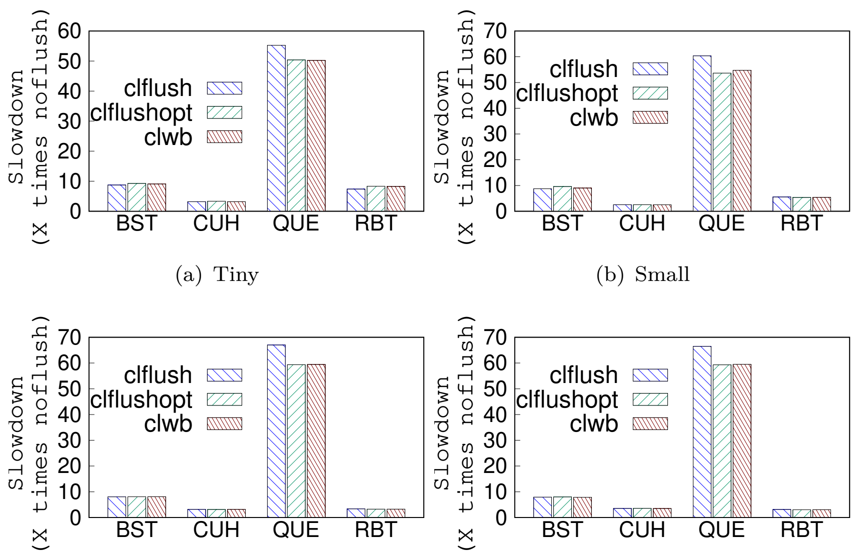
Table 3.6: Working set sizes used for real system experiments

Type	Size Description	Inserts	Deletes	Searches*
Tiny	0.90 x L1-d Size	460	90	138
Small	0.90 x L2 Size	14746	2947	4423
Medium	0.90 x LLC Size	324403	64880	97320
Large	2 x LLC Size	720896	144177	216268

*Queue does not support search operations

Intel(R) Xeon(R) Gold 6226R system with private L1d (32 KB, 8 ways) and L2 (1 MB, 16 ways) and shared L3 (22 MB, 11 ways) cache, running Ubuntu 20.04.5 operating system. This system contains an Intel Optane persistent memory module of capacity ∼512 GB [5], exposed to applications through the direct access (DAX) [99] feature of Linux ext4 file system. Figure 3.5 shows four working set size variants, *tiny*, *small*, *medium*, and *large*, performing a number of insert, delete, and search operations as shown in Table 3.6. In this experiment, micro-benchmarks allocate memory from NVM using `mmap` system call by mapping the DAX file. Figure 3.5 shows the slowdown in execution time while using one of the cache line flush methods with respect to the case when not flushing the cache line, i.e., *noflush*. The micro-benchmarks use `sfence` instruction to order cache line flush instructions in all flush variants in Figure 3.5, and as mentioned earlier, micro-benchmarks allocate memory from volatile DRAM instead of NVM and do not flush cache line after modification in the *noflush* setup.

The cuckoo hashing table (CUH) exhibits the lowest slowdown across *tiny*, *small*, and *medium* working set sizes. CUH shows similar slowdown across all cache line flush variants, with a maximum change in slowdown of 3.20× to 3.35× from `clflush` to `clflushopt` for *tiny* working set size and 2.57× to 2.53× change in slowdown from `clflush` to `clwb` for *small* working set size. It shows that CUH gets minimum benefit by retaining the cache



(a) Tiny (b) Small (c) Medium (d) Large

FIGURE 3.5: Performance slowdown under data consistency methods in real x86-64 system. Y-axis values are slowdown with respect to `noflush` (lower the better).

line in a clean state by using `clwb`. Linear queue (QUE) has the highest slowdown across all working set sizes, with `clflush` showing a higher slowdown than `clflushopt` and `clwb`. QUE gets benefit by keeping the cache line in a clean state using `clwb`, showing a maximum reduction in slowdown, 67× to 59.45×, from `clflush` to `clwb` for *medium* working set size. Binary search tree (BST) gets benefits when cache line is invalidated using `clflush` as it shows lower slowdown than `clwb` for cache fitting working set sizes *tiny*, *small*, and *medium*. Whereas, the Red-black tree (RBT) gets the benefit of keeping the cache line in a clean state using `clwb` by providing a maximum reduction in a slowdown, 3.38× to 3.27× from `clflush` to `clwb` for *medium* working set size.

In summary, the choice of cache line flush variant in persistent barrier primitives should depend upon the nature of the workload and working set size. The performance overhead of different persistent barriers depends upon the cache-friendly nature of benchmarks. For example, CUH shows minimum slowdown across all working set types in simulated and real x86-64 systems. Similarly, QUE has the highest slowdown in simulated and real systems across all working set types. The difference in the extent of slowdown between simulated and real system is because of the difference in configurations of simulated and real system, and the difference in memory allocation for the `noflush` case. The `noflush` in simulated system allocates memory from PCM memory attached to gem5 albeit without performing flushes and `noflush` in real system allocates memory from volatile DRAM. Thus, results

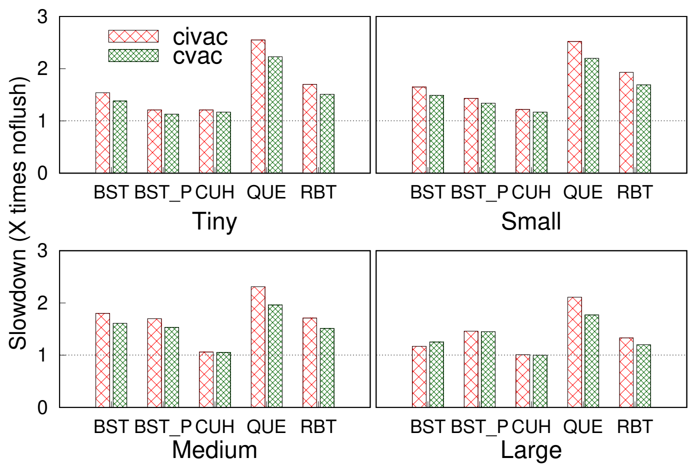


FIGURE 3.6: Performance slowdown with different data consistency methods in gem5 simulation of Arm64 system. Y-axis values are slowdown normalized to noflush (lower the better).

from the simulated system show the overhead of persistent primitives, whereas real system results include overhead due to differences in memory technologies and persistent primitives.

3.2.2 Performance with Arm64

We characterized the performance of different benchmarks (Table 3.1) with `civac` and `cvac` consistency primitives for ARM (refer §2.2 for details) on a simulated gem5 system with configuration in Table 3.3. As Figure 3.6 shows, `cvac` performed better and resulted in a lower slowdown compared to `civac` across all micro-benchmarks and working set sizes. We can functionally equate `cvac` with `clwb` as both of these mechanisms retain cache lines in a clean state and show the same performance trend across different working set sizes. Like in the case of x86-64, the QUE benchmark results in the highest performance overhead for all working set sizes, and CUH has the lowest for all working set sizes. The performance overhead of QUE is associated with the cost of ordering *clean* operations because MPKI at LLC for QUE is lower for `cvac` than for `noflush` and total number of times commit has to stall is higher for `cvac` than `noflush` due to usage of memory fence to order writes to NVM.

We experimented using the read:write ratio as a parameter to study the impact of application memory access behavior. For this experiment, we used three variants for each benchmark: **(i)** *read-light* with read:write ratio of 10:90 **(ii)** *read-balanced* with read:write ratio of 50:50; and **(iii)** *read-heavy* with read:write ratio of 90:10. Note that, the write

Table 3.7: Influence of read-to-write ratio on performance slowdown in gem5 simulation of Arm64 system

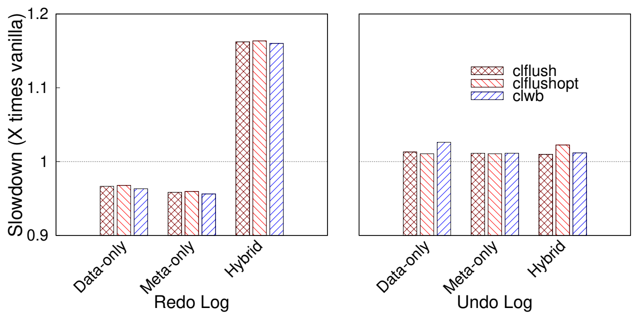
Benchmark	read-light	read-balanced	read-heavy
		civac	
BST	1.67	1.78	1.89
BST_P	1.54	1.64	1.97
QUE	2.47	2.34	2.22
RBT	1.66	1.71	1.82
		cvac	
BST	1.53	1.63	1.83
BST_P	1.45	1.48	1.88
QUE	2.08	1.98	1.88
RBT	1.48	1.51	1.61

operation comprises both insert and delete operations. All benchmarks with different working set sizes result in similar slowdown across the three variants (i.e., read-heavy, read-balance, and read-light) except for the medium working set (shown in Table 3.7). With an increase in the percentage of read operations, the performance overhead increases for all benchmarks except for QUE. With *read-heavy*, BST_P results in 1.27× performance degradation compared to *read-light*. Interestingly, QUE results in comparatively less overhead with increasing number of read operations, with *read-heavy* QUE results in ∽10% less overhead compared to *read-light*; the L1-D miss rate is reduced by ∽5% with *read-heavy* as compared with *read-light* for QUE.

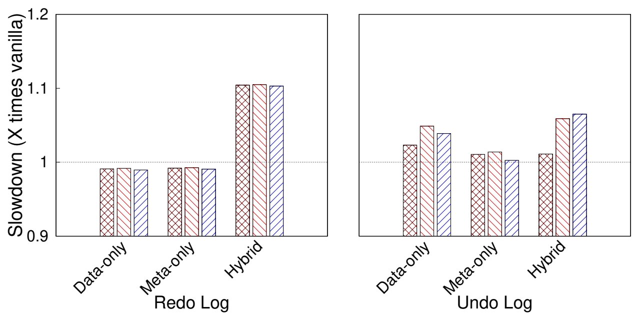
3.3 Performance of Redo and Undo Logging

3.3.1 Performance with x86-64

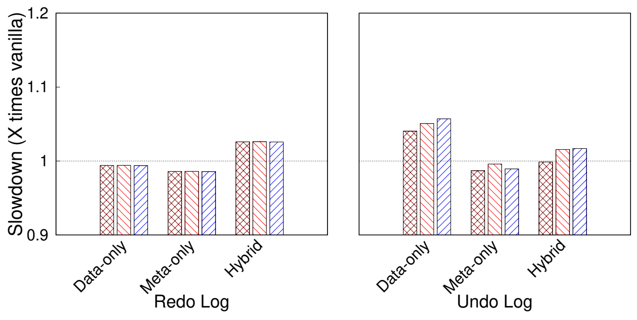
We studied the performance overhead of redo and undo logging with different data consistency methods to ensure consistency of the LRU benchmark (§ 3.1.2). The characterization uses the nature of modifications (Data-only, Meta-only, Hybrid) and logging requirements (Log Heavy, Log Medium, Log Light) as parameters for a transaction with a total of 1000 operations. In the LRU benchmark, *Data-only* corresponds to a *write* access operation to an object in the LRU list where both the LRU structure and data field of the object are modified. *Meta-only* refers to a *read* operation to an object in the LRU list where the LRU structure is modified. *Hybrid* comprises of 75% of *Data-only* and 25% of *Meta-only* accesses. The logging requirement denotes the percentage of total operations requiring logging. For example, *Log Heavy* requires logging for 90%, *Log Medium* requires logging



(a) Log Heavy (No Logging:Logging ratio 10:90)



(b) Log Medium (No Logging:Logging ratio 50:50)



(c) Log Light (No Logging:Logging ratio 90:10)

FIGURE 3.7: Influence of data consistency methods based on nature of update operations for different logging requirements in gem5 simulation of x86-64 system. Y-axis values are slowdown w.r.t. vanilla case of no logging (lower the better).

for 50% and *Log Light* requires logging for 10% of total operations performed in the LRU benchmark. All operations are performed under a single transaction demarcated between *tx_begin* and *tx_end*.

Figure 3.7 shows that using undo log causes a slight performance slowdown for all modification categories and all types of logging requirements, whereas the redo log performs better compared to the vanilla case of *no logging* for all modification categories except for *Hybrid*. Figure 3.7 also shows that `clflush` or `clflushopt` performs better than `clwb` for some cases with undo log, which suggests that `clwb` is not always the best.

Table 3.8: L1-D cache miss and replacements with logging using clwb in gem5 simulation of x86-64 system

Methods	Data-only	Meta-only	Hybrid
L1-D miss rate (in %) with logging			
Log Heavy			
vanilla	20.46	20.43	20.27
redo	18.44	18.52	17.76
undo	20.23	20.27	20.17
Log Medium			
vanilla	15.60	15.59	15.51
redo	14.65	14.81	14.58
undo	14.75	15.55	14.41
Log Light			
vanilla	11.27	11.33	11.26
redo	11.08	11.20	11.19
undo	10.59	11.28	11.16
% of change in L1-D replacements with redo			
Log Heavy	<0.1% ↓	<0.1% ↓	21.89% ↑
Log Medium	<0.4% ↑	<0.4% ↑	12.62% ↑
Log Light	<0.3% ↑	<0.3% ↑	3.18% ↑
↑ increase ↓ decrease w.r.t. vanilla			

With redo log, `clwb` performs marginally better than `clflush` and `clflushopt`. We can interpret the benefit of redo log by analyzing the reduction in percentage of L1-D cache misses in Table 3.8 for the redo log with `clwb`. We have also noticed that L1-D cache write-backs are reduced (between 31% to 5%) in comparison with `vanilla` (no memory persistence) while using redo with `clwb` (because of log access locality).

Additionally, Table 3.9 shows L1-D cache misses for redo and undo logging under different cache line flush mechanisms. With *Log Heavy* logging requirement, redo log reduces L1-D cache misses for *Data-only* and *Meta-only* modification categories, signifying the benefit of locality in accessing redo log; for example, modifying the data structure may result in accesses to different memory locations, whereas with redo, accesses happen to the log area, providing locality. The redo log also reduced L1-D cache misses under *Log Medium* and *Log Light* logging requirements, showing a lower slowdown in Figure 3.7. However, the *Hybrid* modification category did not provide any cache benefits for the redo log mechanism, as the L1-D cache misses are higher than the vanilla in Table 3.9. In addition to higher L1-D cache misses with Hybrid for redo, one more possible reason for comparatively higher performance slowdown for *Hybrid* with redo logging can be the L1-D cache pollution since L1-D replacements are increased with *Hybrid* as shown in Table 3.8 for `clwb` with respect to `vanilla`. This cache pollution may be created by the hardware prefetcher at L1-D

Table 3.9: L1-D cache misses with redo and undo logging in gem5 simulation of x86-64 system

Methods	Scheme	Data-only	Meta-only	Hybrid
		Log Heavy		
vanilla		13258270	13220313	13220555
redo	clflush	12152891	12092364	13531899
	clflushopt	12148182	12089823	13529220
	clwb	12140265	12081518	13521065
undo	clflush	13274812	13255702	13292138
	clflushopt	13264238	13249313	13303524
	clwb	13292468	13257919	13281894
		Log Medium		
vanilla		10084048	10066492	10075016
redo	clflush	9593982	9629593	10360906
	clflushopt	9591458	9628106	10359366
	clwb	9587053	9623506	10354846
undo	clflush	10072095	10105160	10139698
	clflushopt	10060922	10103187	10096180
	clwb	10077741	10100200	10097703
		Log Light		
vanilla		7295364	7321849	7321750
redo	clflush	7224271	7260415	7404615
	clflushopt	7223820	7260180	7404331
	clwb	7222878	7259267	7403427
undo	clflush	7287481	7301643	7349731
	clflushopt	7283329	7305540	7350459
	clwb	7293082	7302328	7352758

since we noticed a 37% increase in the number of prefetch requests for *Hybrid* compared to `vanilla`.

The undo log scheme creates more L1-D misses than `vanilla` for *Log Heavy*, showing a slowdown with respect to `vanilla` in Figure 3.7. For undo log, the slight performance slowdown for all modification categories across different cache line flush schemes in Figure 3.7 is due to higher L1-D misses for undo in Table 3.9. However, the undo log shows a slight (∼0.1%) decrease in L1-D misses for *Data-only* modification category with *Log Medium* and *Log Light* even though Figure 3.7 shows up to 1.04× to 1.05× slowdown for undo under *Data-only* modification category with *Log Medium* and *Log Light*. We can correlate this slowdown with the block in the rename stage of CPU pipeline because of ROB becoming full, as shown in Table 3.10. The rename pipeline stage is blocked more times with undo logging than `vanilla` since the undo log creates and flushes a log entry before each data modification to preserve the old value. In contrast, the redo log flushes log

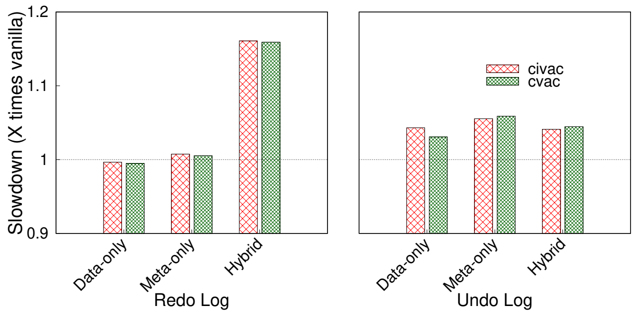
Table 3.10: Number of times rename stage is blocked due to no space in ROB in gem5 simulation of x86-64 system

Methods	Scheme	Data-only	Meta-only	Hybrid
Log Heavy				
vanilla		1296	1336	1511
redo	clflush	1450	1493	1755
	clflushopt	1453	1506	1757
	clwb	1455	1493	1752
undo	clflush	2554	1532	1533
	clflushopt	5699	4430	4350
	clwb	3959	2909	3820
Log Medium				
vanilla		1597	1583	1598
redo	clflush	1572	1666	1760
	clflushopt	1572	1671	1770
	clwb	1572	1669	1759
undo	clflush	2088	1665	1869
	clflushopt	6203	3488	4430
	clwb	3780	3056	4152
Log Light				
vanilla		1649	1617	1685
redo	clflush	1571	1454	1583
	clflushopt	1574	1460	1594
	clwb	1571	1457	1585
undo	clflush	1712	1583	1931
	clflushopt	2938	2086	2907
	clwb	2733	1989	2725

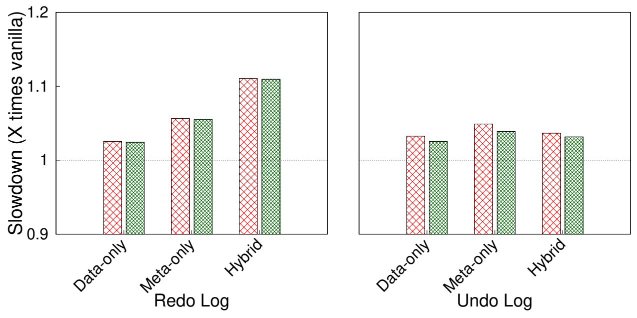
entries on transaction commit, reducing the number of flush operations compared to undo logging. The undo log blocks rename stage by 2× on average across all cache line flush schemes and modification categories with a maximum of 4.39× for *Data-only* category with `clflushopt` scheme. The redo log experiences less number of blocks in the rename stage than undo logging, with an average of 1.03× across all cache-line flush schemes and modification categories.

3.3.2 Performance with Arm64

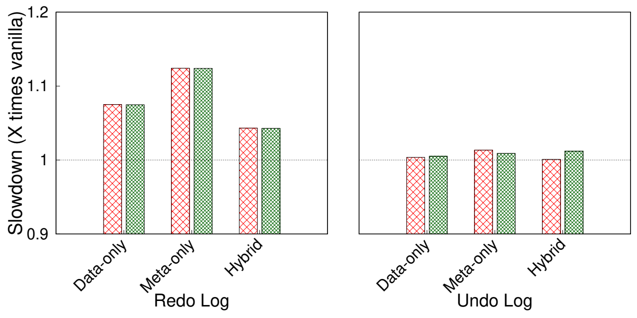
We repeated the undo and redo logging experiments (with the same workload scenario as in § 3.3.1) for ARM using the gem5 simulator. Figure 3.8 shows that redo and undo logging have slowdown with all types of LRU access patterns and logging requirements. Similar



(a) Log Heavy



(b) Log Medium



(c) Log Light

Figure 3.8: Influence of data consistency methods based on nature of update operations for different logging requirements in gem5 simulation of Arm64 system. Y-axis values are slowdown w.r.t. vanilla case of no logging (lower the better).

to the previous experiment, the slowdown in the Figure 3.8 is normalized to `vanilla` (no memory persistence).

The slowdown with respect to `vanilla` in Figure 3.8 can be correlated with the number of times instruction commits stalls for `civac` and `cvac` with redo and undo logging in Table 3.11. Redo logging experiences an average number of commit stalls of 1.10× with respect to `vanilla` across all cache line flush schemes and logging requirements. The undo logging experiences 1.15× average number of commit stalls with respect to `vanilla` across all cache line flush schemes and logging requirements. `Cvac` performs better than

Table 3.11: Number of times instruction commit has stalled in gem5 simulation of Arm64 system

Methods	Scheme	Data-only	Meta-only	Hybrid
Log Heavy				
vanilla		61470	61470	66304
redo	civac	75876	72275	73176
	cvac	75876	72275	73176
undo	civac	83072	77671	79021
	cvac	83072	77671	79021
Log Medium				
vanilla		58032	58032	59430
redo	civac	66037	64037	64537
	cvac	66037	64037	64537
undo	civac	70033	67033	67783
	cvac	70033	67033	67783
Log Light				
vanilla		56848	56848	57058
redo	civac	58452	58052	58152
	cvac	58452	58052	58152
undo	civac	59248	58648	58798
	cvac	59248	58648	58798

Table 3.12: L1-D miss rate (in %) with logging using cvac in gem5 simulation of Arm64 system

Methods	Data-only	Meta-only	Hybrid
Log Heavy			
vanilla	21.41	21.47	21.39
redo	19.15	18.99	18.43
undo	21.16	21.22	21.12
Log Medium			
vanilla	16.64	16.68	16.61
redo	14.58	14.12	15.02
undo	16.57	16.58	16.54
Log Light			
vanilla	12.01	12.04	12.04
redo	10.34	9.69	11.42
undo	12.06	12.03	12.03

`civac` with redo as `cvac` is functionally equivalent to `clwb`. Further, `cvac` also performs better with undo for most of the cases, especially with medium logging requirement. The benefit of `cvac` with redo logging can be correlated with better cache performance since we observe a lower percentage of L1-D misses while using redo with `cvac` in comparison to `vanilla` as shown in Table 3.12.

TABLE 3.13: Number of L1-D replacements in gem5 simulation of Arm64 system

Methods	Scheme	Data-only	Meta-only	Hybrid
Log Heavy				
vanilla		12763128	12751614	12757110
redo	civac	12755605	12749959	15555022
	cvac	12755604	12749956	15555053
undo	civac	12771195	12773142	12771654
	cvac	12776485	12777142	12778770
Log Medium				
vanilla		12709850	12704835	12704950
redo	civac	12747279	12744681	14302353
	cvac	12747277	12744675	14302358
undo	civac	12717631	12713516	12715445
	cvac	12714569	12716420	12716895
Log Light				
vanilla		12728623	12724426	12721995
redo	civac	12759929	12769325	13129054
	cvac	12759929	12769320	13129064
undo	civac	12727151	12725661	12724371
	cvac	12725427	12731740	12727743

Similar to x86-64 setup, the higher slowdown under *Hybrid* set of modifications with redo can be correlated with an increase in L1-D replacements under redo compared to `vanilla` as shown in Table 3.13.

There is an increase in performance slowdown while using redo for meta-only modification with light logging requirements; this is because the completion time of LRU benchmark using redo with `cvac` for meta-only modifications increases by 1.18% (Log Medium), 3.4% (Log Light) as compared with data-only modifications, whereas `vanilla` case decreases by 1.16% (Log Medium), 1.11% (Log Light) as compared with data-only modifications, showing comparatively higher performance slowdown for meta-only modification with respect to data-only for light logging requirement.

3.4 Influence on co-running applications

Data consistency operations (flush+fence) may influence the performance of co-running applications due to the presence of shared resources such as LLC, internal buffers, and interconnects. We studied the impact of data consistency methods of co-running applications on the performance of standard applications. We use an x86-64, 4-core gem5 setup where *cpu0* executes SPEC CPU 2017 [121], and the remaining three cpus execute a mix

Table 3.14: Performance of SPEC while co-running with micro-benchmarks in gem5 simulation of x86-64 system

cpu0	cpu1	cpu2	cpu3	Scheme	CPI(cpu0)
605.mcf_s	BST	CUH	QUE	noflush	30.11
				clflush	29.99
				clflushopt	30.12
				clwb	30.10
	BST	CUH	RBT	noflush	30.07
				clflush	30.14
				clflushopt	30.09
				clwb	30.11
	BST	QUE	RBT	noflush	30.02
				clflush	30.07
				clflushopt	30.07
				clwb	30.07
	RBT	CUH	QUE	noflush	31.51
				clflush	31.45
				clflushopt	31.48
				clwb	31.58
544.nab_r	BST	CUH	QUE	noflush	2.25
				clflush	2.20
				clflushopt	2.22
				clwb	2.21
	BST	CUH	RBT	noflush	2.10
				clflush	2.13
				clflushopt	2.07
				clwb	2.05
	BST	QUE	RBT	noflush	2.20
				clflush	2.21
				clflushopt	2.21
				clwb	2.21
	RBT	CUH	QUE	noflush	2.28
				clflush	2.23
				clflushopt	2.24
				clwb	2.27

of different benchmarks from the set of micro-benchmarks (Table 3.1). We configure these micro-benchmarks with large working set size with 131070 inserts, 10 million operations split as 50% deletes, and 50% searches. Table 3.14 shows the performance of different SPEC benchmarks co-running with a mixture of micro-benchmarks with different flush methods against the case when micro-benchmarks do not use flush (`noflush`). Table 3.14 shows that, among the SPEC benchmarks, 605.mcf_s experiences less than ∼1% slow-down across different micro-benchmark combinations and flush schemes, and 544.nab_r experiences ∼1.25% slowdown while co-running with BST, CUH, and RBT performing

TABLE 3.15: LLC MPKI of SPEC while co-running with micro-benchmarks in gem5 simulation of x86-64 system

cpu0	cpu1	cpu2	cpu3	Scheme	LLC MPKI (cpu0 data)	LLC Avg Miss Latency in Cycles (cpu0 data)
605.mcf_s	BST	CUH	QUE	noflush	5.3453	3861061.01
				clflush	5.3415	3851644.19
				clflushopt	5.3563	3874173.41
				clwb	5.3544	3856103.79
	BST	CUH	RBT	noflush	5.3464	3835715.25
				clflush	5.355	3834648.06
				clflushopt	5.348	3839118.09
				clwb	5.3495	3832508.45
	BST	QUE	RBT	noflush	5.3524	3869823.12
				clflush	5.3658	3866151.66
				clflushopt	5.3658	3866151.66
				clwb	5.3658	3866151.66
	RBT	CUH	QUE	noflush	5.4922	4029302.46
				clflush	5.4768	4025681.34
				clflushopt	5.4721	4025792.41
				clwb	5.4841	4034405.5
544.nab_r	BST	CUH	QUE	noflush	0.1209	4342599.48
				clflush	0.1208	4230257.61
				clflushopt	0.1167	4292193.18
				clwb	0.1156	4324631.82
	BST	CUH	RBT	noflush	0.1211	3822136.77
				clflush	0.1193	3987475.03
				clflushopt	0.1227	3776774.4
				clwb	0.1207	3614881.83
	BST	QUE	RBT	noflush	0.1179	4199030.09
				clflush	0.1194	4130111.8
				clflushopt	0.1194	4130111.8
				clwb	0.1194	4130111.8
	RBT	CUH	QUE	noflush	0.1226	4400267.69
				clflush	0.121	4188637.3
				clflushopt	0.1213	4268630.32
				clwb	0.1218	4377243

`clflush`.

Table 3.15 shows LLC MPKI and average LLC miss latency of SPEC benchmarks corresponding to their performance in Table 3.14. 605.mcf_s has less than a 0.5% difference in LLC MPKI while using different cache-line flush schemes across all micro-benchmark combinations compared to `noflush`. 544.nab_r has a maximum of 1.32% difference in LLC MPKI while co-running with BST, CUH, and RBT performing `clflushopt` scheme. 605.mcf_s and 544.nab_r have a reduction in average LLC miss latency for most flush schemes across micro-benchmark combinations compared to `noflush`. 605.mcf_s has a maximum reduction of ∼0.25% while co-running with BST, CUH, and QUE performing `clflush`, and 544.nab_r has a maximum reduction of ∼5.4% while co-running with BST, CUH, and RBT performing `clwb`. Even though there is a reduction in LLC MPKI for 544.nab_r while co-running with BST, CUH, and RBT performing `clflush`, the overhead in CPI for this case in Table 3.14 is primarily due to the increase in LLC miss latency of ∼4.3% with respect to `noflush`.

We also observe that even with an increase in LLC MPKI, there is a chance for a decrease in LLC average miss latency and vice versa; for example, 544.nab_r co-running with BST, CUH, and RBT performing `clwb` has an increase in LLC MPKI but a decrease in LLC miss latency, indicating the influence of flushing on shared elements between LLC and memory such as memory controller queues and others. Thus, it will be interesting to study memory and cache-congested scenarios further to analyze the extent of the performance implications.

3.5 Summary

Application state recovery in a consistent manner with NVM requires data consistency mechanisms supported by underlying architectural primitives like `flush` and `fence`. In this chapter, we empirically analyzed the performance overhead of data consistency methods on Intel x86-64 and Arm64. Table 3.16 lists key points and observations from this empirical analysis.

In this chapter, we studied data structures that are commonly employed in applications and operating systems. OS majorly uses these data structures to maintain different states of a process, i.e., the program in execution. Process persistence requires saving all state elements of a process. Thus, understanding the performance overhead associated with consistently maintaining these popular data structures provides insights into the overhead for process persistence. Process persistence also requires the persistence of other state elements, such as execution context and memory state. Therefore, we need a framework to study the end-to-end performance overhead of process persistence, especially one that supports a hybrid memory system with DRAM and NVM. Such a framework aims to aid in exploring alternative design approaches for process persistence.

TABLE 3.16: Key takeaways from the empirical analysis

Key points	Observations
Performance overhead of data consistency methods.	Depends upon the nature of workload, proportion of read-to-write operations, and working set size.
	Depends upon the usage of serialization operations such as `sfence` to enforce order of writes to NVM.
Usage of `clwb` and its equivalents for data consistency methods.	Benefits applications with cache locality, for example, `clwb` benefits in redo log schemes.
	Using `clwb` and its equivalents over other methods, such as `clflushopt`, may not always be advantageous for better performance.
Performance of `clflush` or `clflushopt` for data consistency methods.	Better than `clwb` in some cases with undo log.
Selection of data consistency methods for persistence in NVM.	Should be decided on a case-by-case basis depending on the workload characteristics.

Keeping this motivation, in the next chapter, we introduce an open-source framework, Kindle, based on gem5 [49] and gemOS [51], to explore and prototype research ideas in hybrid memory systems crossing the hardware-software boundaries and perform comprehensive empirical evaluation.

Chapter 4

Kindle: A Hybrid Memory Framework

Non-volatile memory (NVM) provides high memory capacity with reduced energy cost compared to volatile memory (DRAM). However, higher read/write latency of NVM [11] raises performance concerns when used as a drop-in replacement for DRAM. Therefore, a more attractive memory organization is a hybrid memory system with DRAM and NVM [27], designed to complement each other to reduce energy costs, achieve persistence, and improve performance. A hybrid memory system allows placing data in NVM and/or DRAM, enabling OS memory managers to coordinate allocations for high memory capacity with low access latency. For example, a typical OS policy can place frequently accessed hot memory pages in DRAM and migrate cold pages to NVM to increase the overall system performance. Several existing works on hybrid memory systems propose efficient access for large workloads by intelligently placing data across the NVM and DRAM [122–125], reducing energy consumption by migrating data across the DRAM and NVM tiers [126–129]. Another usage of NVM is as an alternative to external storage hardware (HDD or SSD) for data persistence. In this usage scenario, application developers or file system designers must incorporate consistency and durability semantics into their design. Existing research works attempt to address the memory consistency issues by designing solutions such as persistent object stores [130], lock-based failure atomicity for multi-threaded programs [19, 34, 131], and hardware and/or software memory consistency mechanisms [17, 21, 24–26, 132].

Hybrid memory provides extensive opportunities in system design, Figure 4.1 shows the number of publications on hybrid memory systems with NVM between 2018 and 2023

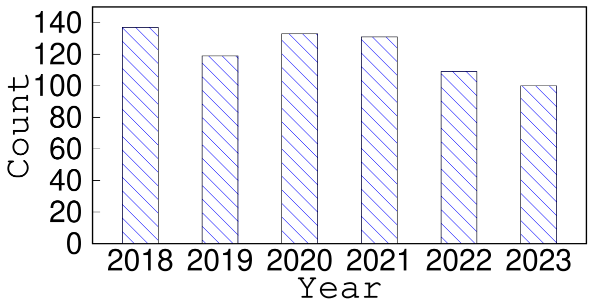


Figure 4.1: Number of publications on hybrid memory systems with NVM, listed in Google Scholar.

based on data extracted from Google Scholar [133]. The number of publications shows an average of 120 research papers annually, demonstrating the wide range of research opportunities in hybrid memory systems. The nature of problems and proposed solutions for hybrid memory systems require an infrastructure allowing *extension, validation, and evaluation* of the complete system stack.

Architecture simulators capable of full-system simulation, such as gem5 [51], provide platforms for prototyping ideas crossing the hardware-software boundaries. However, using gem5-Linux full-system simulation setup to explore new ideas in hybrid memory systems has the following shortcomings. *First*, while gem5 models the NVM controller and the Linux kernel can detect NVM on real hardware (such as Intel DCPMM) [53, 54], their integration is non-trivial (in gem5), especially considering constantly evolving hardware and OS design. *Second*, the Linux kernel is heavy with features whereby the OS functions and services can consume a significant part of a simulation, which may not be desirable for quick prototyping design ideas. *Third*, designing a proof of concepts (PoCs) in Linux requires significant understanding and changes in the Linux memory management subsystem, which has non-trivial complexity.

This chapter proposes *Kindle*, a comprehensive infrastructure for quick prototyping and evaluating novel mechanisms and policies *crossing architecture and operating system boundaries* in hybrid memory systems. In the core of *Kindle*, support for full process persistence is provided, whereby a process can restart consistently after a system crash. Several design alternatives, challenges, and performance insights in achieving process persistence in hybrid/persistent memory systems. As one specific illustration, we compare two design choices for consistently maintaining the page table across system restarts—*(i)* hosting the page tables directly on NVM and *(ii)* hosting the page tables in DRAM while performing periodic checkpoints into NVM. Next, we present a complete evaluation framework for hybrid memory systems by combining the process persistence support with other tracing and simulation techniques where a user can perform end-to-end modeling of real-world application benchmarks such as SPEC [121], GAP [134], and YCSB [135] in a generic manner (§ 4.1). Finally, we showcase the utility of *Kindle* in gaining new insights by using prototype implementations of two well-established research ideas proposing optimizations for hybrid memory systems in OS and/or hardware layers (§ 4.2).

We design and implement *Kindle* with support for full process persistence by modifying a lightweight OS (i.e., *gemOS* [52]). We modify the gemOS system call APIs to allow user applications to allocate memory in DRAM and/or NVM using memory allocation API, enabling exploration of memory usage and access behavior of standard applications on hybrid memory systems. At a high level, *Kindle* consists of two components.

1. A preparation component for transforming and extracting required information from the application (and its interaction with the OS) to prepare the stage for the simulation run.
2. A simulation component for running the applications in full persistence mode with the configurations provided by the user.

Kindle aids in quickly realizing complex design goals and providing new insights into existing schemes, as we show through the prototype implementations of two state-of-the-art hardware-software hybrid memory schemes—Shadow Sub-Paging (SSP) [25] and Hardware/Software Cooperative Caching (HSCC) [27]. SSP handles the memory consistency requirement of NVM by using shadow sub-paging. HSCC provides memory capacity by arranging DRAM and NVM in a flat address space and managing DRAM as a cache to NVM using a hardware/software cooperative caching mechanism. These prototype implementations show the capability of *Kindle* to quickly validate ideas in hybrid memory systems and gather new insights. Through the SSP prototype, Kindle provides new insights into the impact of consistency interval on the performance overhead of SSP, showing that a wider consistency interval reduces the overhead. HSCC prototype with Kindle provides insights into the migration overhead due to the OS activities, which authors did not shown in the original work as the evaluation framework did not have an OS component.

4.1 Design and Implementation

Kindle provides a hybrid memory system, allowing applications to allocate memory from NVM and DRAM. Figure 4.2 shows the interactions between gem5 [50] and gemOS [52]

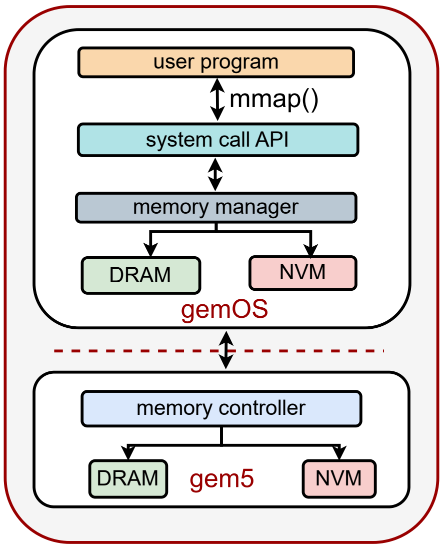


FIGURE 4.2: Interaction between gem5 and gemOS in Kindle

TABLE 4.1: gem5 Memory Configuration

Parameter	**Used Setting**
DRAM interface	DDR4-2400 16x4
NVM interface	PCM ‡
NVM Write buffer size	48
NVM Read buffer size	64
Memory capacity	3GB DRAM + 2GB NVM
‡PCM timing parameters based on [119]	

in *Kindle* to provision memory based on the application requirements. *Kindle* partitions the physical memory address range between NVM and DRAM and inserts corresponding entries in the gem5 BIOS implementation of `e280` (BIOS memory map). We configure the NVM memory controller interface in gem5 with the specifications mentioned in Table 4.1. *Kindle* provides a hybrid memory system in flat address mode, allowing the OS to expose DRAM and NVM to applications.

```
// mmap() does not follow POSIX mmap semantics
int main(){
   char* ptr1= (char*)mmap(NULL,4096,PROT_WRITE,MAP_NVM); //allocation in NVM
   char* ptr2= (char*)mmap(NULL,4096,PROT_WRITE,0); //allocation in DRAM
   ptr1[0] = 'A'; //store to NVM
   ptr2[0] = 'B'; //store to DRAM
   //munmap allocations
   return 0;
}
```

LISTING 4.1: Sample mmap() code to allocate in NVM

The gemOS in *Kindle* exposes DRAM and NVM to user-space applications through an extended `mmap` system call API. An application differentiates memory requests for NVM from DRAM by passing an additional flag `MAP_NVM` in `mmap()` system call as shown in the sample code Listing 4.1.

Additionally, the virtual memory area (VMA) layout in gemOS is modified to tag each VMA as either DRAM or NVM depending on the value of `MAP_NVM` flag passed in the `mmap()` system call invocations from the user space. The physical memory allocation from the NVM or DRAM page frames is performed based on this memory type tag.

4.1.1 Process Persistence

Process persistence requires saving the state of a process that consists of CPU registers and memory state encompassing the code, heap and stack areas. Process persistence also requires saving the OS meta-data (e.g., address space layout, process relations) to resume execution from a consistent state after a system crash.

We employ a process persistence scheme based on periodic checkpointing of process contexts in the modified gemOS. We maintain a per-process *saved state* in NVM, containing two copies of the execution context—one as a consistent copy and another as a working copy. We use a redo log (stored in NVM) to capture all modifications to the OS-level process meta-data. As part of the saved state, we also maintain a list of virtual page to NVM physical page frame mappings. At the end of each checkpoint interval, we first log the CPU state and update the working copy using all logged entries in that interval. We consider only the consistency of the CPU state and OS-level meta-data and assume that all heap/stack data pages are consistently maintained in NVM using some existing memory consistency techniques [18, 21, 24, 29, 83, 132]. The list containing the virtual to NVM physical page mapping is maintained to preserve the association from any virtual to a physical page, as it is useful for scenarios where we need to rebuild the virtual memory mapping of a process in the page table after reboot. We also modify the physical page allocation mechanism in gemOS to persist the page allocation meta-data to ensure correctness after crash and reboot scenarios.

At the end of a checkpoint interval, the *saved state* of a context is updated to reflect changes in the interval, which includes modifying the virtual to NVM physical page mapping list by traversing the page table of the process, getting the working copy of context and applying changes in the redo log. After applying all changes in the redo log, mark the updated context as the latest consistent copy of the context in the *saved state*. The

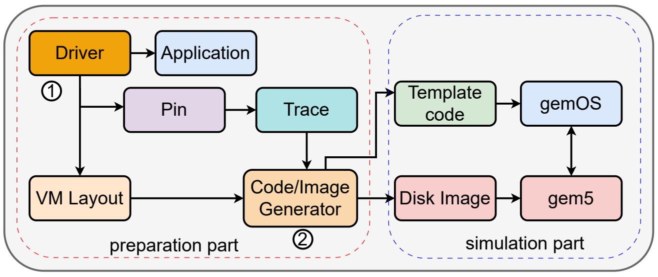


FIGURE 4.3: Schematic diagram of Kindle

resume procedure reconstructs the execution context using the latest consistent copy of the context in the *saved state* after a crash and reboot. The recovery procedure scans through the list of *saved states* and creates a new execution context for each saved state. We copy the latest consistent copy of the context for each process and recreate the virtual memory layout as part of the recovery procedure. Finally, the recovery process sets up the page table mapping for the virtual address space and marks the process state as ready for execution.

4.1.2 Kindle Framework

Figure 4.3 shows the complete *Kindle* framework and interactions between its different components. *Kindle* consists of two major components—a simulation sub-system and a preparation sub-system. The simulation part holds cycle-accurate architecture simulator gem5 [50, 51] and the extended lightweight operating system gemOS [52], specialized for running on gem5.

The preparation component creates the disk image for gem5 and the benchmark template code for gemOS. The vanilla gemOS can not run standard workloads as it is primarily designed for OS education and has limited user-space libraries. We use gemOS as the operating system component of *Kindle* since the primary aim is to provide a framework for quick prototyping, and gemOS reduces simulation time compared to Linux. GemOS, while providing most of the POSIX-compliant APIs, does not include most of the background processes present in production OSes as they interfere with the application under study and hide its actual behavior in the statistics collected. Thus, gemOS benefits in providing cleaner statistics. As gemOS currently has limited support for user space libraries, the preparation component overcomes this limitation by setting up an environment for tracing and replaying the memory operations on gemOS for running standard applications. The preparation sub-system consists of a driver program (①) to trace the instructions executed by the application of interest using Intel's dynamic binary instrumentation tool Pin [58]. The driver program (using fork and exec) coordinates an application's execution and memory access tracing with Pin while saving the virtual memory layout by reading the `/proc/pid/maps` pseudo file exposed by the Linux kernel. In case of multi-threaded applications, *Kindle* can use SniP [136] along with the *maps* file to capture address layout of application. SniP is a framework capable of capturing the stack area of threads. The trace generated by Pin contains the type of memory access (read/write), address and size of memory access, and time of access.

The code/image generator component (②) makes the disk image required by gem5 in the simulation part of *Kindle*. It processes the trace file to generate a tuple containing *(period, offset, operation, size, area)* for each memory access. The *period* shows the time of memory access, *offset* shows the address of memory access within a heap/stack area, *operation* indicates the type of memory access, whether it is a read/write, and *size* denotes size of memory read/write and *area* denotes the name of the memory area, i.e., which heap and stack area is read/written. The *image generator* labels each memory area in the virtual memory layout information captured using the *maps* pseudo-file and then associates memory accesses in trace to an *area* name by checking whether access lies within the address range of that area. The prepared disk image contains *(period, offset, operation, size, area)* information of all memory accesses in the trace. The *code generator* prepares a template gemOS code containing heap and stack allocations matching the number and size of allocations in the application. The generated code also contains routines to access *(period, offset, operation, size, area)* tuple from the disk image for mimicking the memory access in the application. Users of the Kindle framework can update this template code to include additional functionality before launching the `init` process (the first user process in gemOS) with required arguments [52]. For example, we added a loop counter to compensate for the time interval between two memory accesses in the generated code as an approximation for computation between memory accesses before running experiments.

Validation of Kindle

We have validated the process persistence feature of *Kindle* by crashing and restarting the application multiple times. Validating the working of *Kindle* can be associated with the fidelity of individual components such as Intel Pin and gem5. Additionally, *Kindle* does not impact the simulation time of gem5.

Table 4.2: Benchmark Details

Benchmark	**Total Ops**	**read %**	**write %**
Gapbs_pr [134]	10,000,000	77	23
G500_sssp [137]	10,000,000	68	32
Ycsb_mem [135]	10,000,000	71	29

Availability of Kindle

Kindle is open-sourced and available at `https://github.com/arunkp1986/Kindle`. Users can explore new hardware-software designs in hybrid memory systems by changing the simulation part, implementing software level changes in gemOS and hardware level changes in gem5, and then running applications of interest by following the README documentation. *Kindle* earned all three badges in the artifact evaluation, *Code Available*, *Code Reviewed*, and *Code Reproducible*.

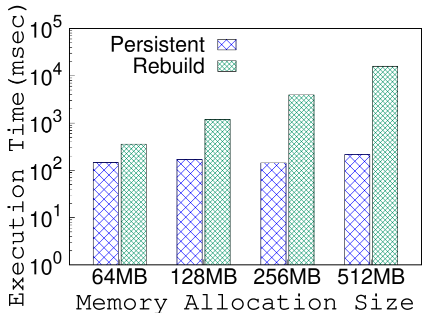
4.2 Evaluation

In this section, we show a consistency scheme based on checkpointing for execution context to achieve process persistence. We study trade-offs in checkpoint performance while using two different approaches to keep the page table consistent. We also demonstrate the capability of *Kindle* in performing an initial evaluation of existing or new ideas on a hybrid memory system. To demonstrate the utility of *Kindle* in hybrid memory research, we implemented two research ideas,

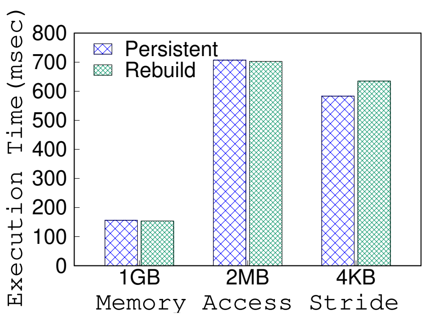
(i) Shadow sub-paging (SSP) [25] to ensure consistent memory state of an application in NVM by employing shadow paging [25, 91] at sub-page cache line granularity.

(ii) A hardware-software co-operative caching mechanism, HSCC [27], for managing DRAM as cache to NVM.

In these two studies, we used standard applications in Table 4.2 and configured gem5 with Intel 64-bit in-order CPU at 3GHz with 32KB L1, 512KB L2 and 2MB/core LLC.



(a) sequential access (log scale)



(b) stride access

FIGURE 4.4: Influence of memory access size and stride length on execution time with periodic checkpointing for context while using different page table consistency schemes.

4.2.1 Process Persistence

As the end-to-end performance of checkpointing the execution context depends upon virtual address space management, we study the performance trade-offs in maintaining the page table using two approaches,

(i) Rebuild the page table from virtual to NVM page using the mapping maintained in the saved state (i.e., *rebuild scheme*) on reboot.

(ii) Maintain the complete page table in NVM and wrap page table modifications inside the NVM consistency mechanism [17]; this only requires setting the PTBR (Page Table Base Register) to point to the first level of page table after a reboot (i.e., *persistent scheme*).

While the *rebuild* scheme allows hosting page tables in DRAM, it may suffer from checkpoint overheads due to additional maintenance of virtual to physical mapping information. On the other hand, while the *persistent* scheme simplifies the checkpoint process, it adds additional overheads to host and maintain the page table in NVM in a consistent manner.

We looked into the impact of *address space size* and *page table size* on the end-to-end performance of periodic checkpointing of execution context while using *rebuild* and *persistent* schemes for page table consistency. Figure 4.4 shows the end-to-end execution time (in msec) for consistently maintaining context using periodic checkpointing under *rebuild* and *persistent* page table maintenance schemes. The periodic checkpoint interval is fixed to 10 msec (based on Aurora [1]).

In the sequential memory allocation and access experiment (Figure 4.4(a)) using a microbenchmark, we allocate virtual memory of different sizes using *mmap* system call with

MAP_NVM flag and sequentially access all pages in the allocated space. Sample code in Listing 4.2 shows the sequential micro-benchmark for memory allocation and access of 64 MB.

```
// Removed error checks for brevity.
#define PAGE_SIZE 4096
#define SIZE (1UL<<24)
    checkpoint_start();
    char* ptr[4];
    ptr[0] = (char*)mmap(NULL,SIZE,PROT_WRITE|PROT_READ,MAP_NVM);
    for(int i=0; i < SIZE; i+=PAGE_SIZE){
        ptr[0][i] = i;
    }
    ptr[1] = (char*)mmap(NULL,SIZE,PROT_WRITE|PROT_READ,MAP_NVM);
    for(int i=0; i < SIZE; i+=PAGE_SIZE){
        ptr[1][i] = i;
    }
    ptr[2] = (char*)mmap(NULL,SIZE,PROT_WRITE|PROT_READ,MAP_NVM);
    for(int i=0; i < SIZE; i+=PAGE_SIZE){
        ptr[2][i] = i;
    }
    ptr[3] = (char*)mmap(NULL,SIZE,PROT_WRITE|PROT_READ,MAP_NVM);
    for(int i=0; i < SIZE; i+=PAGE_SIZE){
        ptr[3][i] = i;
    }
    checkpoint_end();
```

Listing 4.2: Sequential memory allocation and access of 64MB.

In the stride access experiment (Figure 4.4(b)), the micro-benchmark performs a fixed number of 4KB page allocations with a predefined gap (1GB, 2MB, or 4KB) in the virtual address space to ensure different page table levels are populated to result in larger page table size. For example, with a 1GB gap, the micro-benchmark allocates ten 4KB pages at a gap of 1GB to make entries at page-directory-pointer-table (Level-3), page-directory-table (Level-2), and page-table (Level-1) in Intel x86-64 system page table [48], the micro-benchmark allocates 256 4KB pages at a gap of 2MB in the 2MB case and 512 4KB pages at a gap of 4KB in the 4KB case. Sample code in Listing 4.3 shows the stride micro-benchmark with a 1GB gap. The outer loop (j) iterates through 25 times in 2MB and 4KB cases.

In the sequential access experiment (Figure 4.4(a)), the *rebuild* scheme results in higher execution time for all allocation sizes with overhead ranging from ∼2.4× (64MB) to ∼74.2× (512MB) compared to the *persistent* scheme. The overhead in the *rebuild* scheme comes from the need to maintain a list containing virtual to NVM physical page mapping for reconstructing the page table after reboot, and the overhead to maintain this list increases with an increase in mapped virtual memory area size, ∼44× increase in execution time from 64MB allocation size to 512MB. On the other hand, for the stride access experiment (Figure 4.4(b)), the *persistent* scheme results in slightly more execution time compared to *rebuild* for 1GB and 2MB stride access scenarios, as more page table levels are updated for 1GB and 2MB.

```
// Removed error checks for brevity.
#define PAGE_SIZE 4096
#define SIZE (1UL<<12)
  checkpoint_start();
  char* ptr[10];
  int j = 0;
  while(j<100){
     char * addr = (char*)mmap(NULL,SIZE,PROT_WRITE|PROT_READ,MAP_NVM);
     munmap(addr,SIZE);
     for(int i=0; i<10; i++){
        ptr[i] = (char*)mmap(addr,SIZE,PROT_WRITE|PROT_READ,MAP_FIXED|MAP_NVM);
        ptr[i][0] = i;
        addr = ptr[i]+(1UL<<30);
     }
     for(int i=0; i<10; i++){
        munmap(ptr[i],SIZE);
     }
     j++;
  }
  checkpoint_end();
```

Listing 4.3: Stride experiment with 1GB gap.

Page table modifications are minimal for the 4KB case, and the *persistent* scheme performs better than the *rebuild* scheme. The overhead in the *rebuild* scheme can be attributed to maintaining virtual to NVM physical page mapping, similar to memory allocation size experiment. In short, the *persistent* scheme results in more overhead if the virtual address space is sparsely populated. However, the *persistent* scheme performs better than the *rebuild* for scenarios with minimum page table modifications, as shown in the memory allocation size experiment where the page table is updated only on the first access.

Next, we looked into a *scenario with more page table updates* using a sequence of *mmap* and *munmap* operations. Sample code in Listing 4.4 shows the *mmap* and *munmap* micro-benchmark.

The micro-benchmark allocates a virtual address space of 512MB and writes to all pages in 512MB to create valid page table entries. The benchmark then frees a fixed size memory (i.e., 256MB, 128MB, and 64MB) from the start of 512MB space by calling *munmap* and reallocated the same fixed size memory by calling *mmap*. Similarly, *munmap* and *mmap* of

Table 4.3: Execution time with periodic checkpointing of execution context for different VMA modification size

Alloc/Free Size	Persistent (msec)	Rebuild (msec)
64MB	325	19377
128MB	389	23438
256MB	517	29376

the same fixed size are performed on 512MB space for one more time. The newly allocated fixed size regions are then accessed for reading, and finally, the entire area is unallocated by called *munmap*. Sample code in Listing 4.4 uses fixed memory size as 64 MB.

```c
// Removed error checks for brevity.
#define PAGE_SIZE 4096
#define SIZE (1UL<<29)
   checkpoint_start();
   char * ptr = (char*)mmap(NULL, SIZE,PROT_WRITE|PROT_READ,MAP_NVM);
   for(int i=0; i<SIZE; i+=PAGE_SIZE){
       ptr[i] = i%10;
   }
   munmap(ptr,SIZE>>3);
   char * ptr1 = (char*)mmap(NULL, SIZE>>3,PROT_WRITE|PROT_READ,MAP_NVM);
   for(int i=0; i<(SIZE>>3); i+=PAGE_SIZE){
       ptr1[i] = i%10;
   }
   munmap((ptr+(SIZE>>3)),SIZE>>3);
   char * ptr2 = (char*)mmap(NULL, SIZE>>3,PROT_WRITE|PROT_READ,MAP_NVM);
   for(int i=0; i<(SIZE>>3); i+=PAGE_SIZE){
       ptr2[i] = i%10;
   }
   int sum = 0;
   for(int i=0; i<(SIZE>>3); i+=PAGE_SIZE){
       sum += ptr1[i];
   }
   munmap(ptr1,SIZE>>3);
   for(int i=0; i<(SIZE>>3); i+=PAGE_SIZE){
       sum += ptr2[i];
   }
   munmap(ptr2,SIZE>>3);
   munmap(ptr+(SIZE>>2),(SIZE>>1)+(SIZE>>2));
   checkpoint_end();
```

Listing 4.4: Micro-benchmark with sequence of mmap and munmap operations.

Table 4.3 shows end-to-end execution time with execution context checkpointing and maintaining page table using *persistent* or *rebuild* method. The table shows execution time while performing *munmap* and *mmap* sequences of different allocation/free fixed sizes. The *persistent* scheme overhead increases with an increase in allocation/free size as more page table changes are required with increased *munmap* and *mmap* size, showing ~1.6× increase in execution time from 64MB to 256MB. The execution overhead increases for *rebuild* scheme as well since the virtual to NVM physical page mapping maintained for page table rebuilding requires constant change due to changes in virtual memory area, showing ~1.5× increase in execution time from 64MB to 256MB.

```
// Removed error checks for brevity.
#define PAGE_SIZE 4096
#define SIZE (1UL<<29)
#define TARCK_SIZE 64
#define ITERATION 3
   checkpoint_start();
   char * ptr = (char*)mmap(NULL, SIZE,PROT_WRITE|PROT_READ,MAP_NVM);
   int j = 0;
   while(j<ITERATION){
       for(int i=0; i<SIZE; i+=PAGE_SIZE){
          ptr[i] = i%10;
       }
       j += 1;
   }
   munmap(ptr,SIZE>>3);
   char * ptr1 = (char*)mmap(NULL, SIZE>>3,PROT_WRITE|PROT_READ,MAP_NVM);
   j = 0;
   while(j<ITERATION){
       for(int i=0; i<(SIZE>>3); i+=PAGE_SIZE){
           ptr1[i] = i%10;
       }
       j += 1;
   }
   munmap((ptr+(SIZE>>3)),SIZE>>3);
   char * ptr2 = (char*)mmap(NULL, SIZE>>3,PROT_WRITE|PROT_READ,MAP_NVM);
   j = 0;
   while(j<ITERATION){
       for(int i=0; i<(SIZE>>3); i+=PAGE_SIZE){
           ptr2[i] = i%10;
       }
       j += 1;
   }
   int sum = 0;
   for(int i=0; i<(SIZE>>3); i+=PAGE_SIZE){
       sum += ptr1[i];
   }
   munmap(ptr1,SIZE>>3);
   for(int i=0; i<(SIZE>>3); i+=PAGE_SIZE){
       sum += ptr2[i];
   }
   munmap(ptr2,SIZE>>3);
   munmap(ptr+(SIZE>>2),(SIZE>>1)+(SIZE>>2));
   checkpoint_end();
```

LISTING 4.5: Micro-benchmark for pagetable maintenance with 64MB alloc/free size.

Table 4.4: Influence of checkpoint interval on execution time for periodic checkpointing of execution context

Alloc/Free Size	Interval	Persistent (msec)	Rebuild (msec)
64MB	10 msec	445	55270
	100 msec	445	10580
	1 sec	444	430
128MB	10 msec	534	68078
	100 msec	532	13103
	1 sec	532	515
256MB	10 msec	710	88193
	100 msec	708	15775
	1 sec	708	685

Finally, we look into the scenario which shows the benefit of keeping the page table in DRAM. The page table is accessed in cases to update mapping entries as a result of virtual memory area changes due to *mmap*, *munmap*, *mremap*, *mprotect* calls or to translate/populate mapping corresponding to a virtual address after missing in TLBs and other intermediate caches. Keeping the page table in DRAM benefits from increased page table access as DRAM provides better read and write latency than NVM. We use a micro-benchmark similar to the previous case to study the benefit of keeping the page table in DRAM.

Listing 4.5 shows the micro-benchmark sample code for page table maintenance. Micro-benchmark allocates a virtual address space of 512MB and accesses pages to make page table entries and then frees a fixed size memory (i.e., 256MB, 128MB, and 64MB) from the start of 512MB space by calling *munmap* and reallocated the same fixed size memory by calling *mmap*. After allocation, all pages in the virtual memory area are accessed multiple times to cause TLB misses. The benchmark does one more round of deallocation and allocation of the same fixed size and multiple rounds of accesses to allocated space. Finally, the entire area is unallocated by calling *munmap*. Code in Listing 4.5 uses 64 MB as the fixed freeing size from 512 MB.

Table 4.4 shows the end-to-end execution time to checkpoint process context with different checkpoint intervals while using *persistent* and *rebuild* schemes for page table maintenance. The execution time for the *persistent* scheme remains similar across different checkpoint intervals for a virtual memory area size of alloc/free operations. This similarity in execution time for the persistent scheme is because the overhead comes from the NVM consistency mechanism, and the number of page table modifications remains the same for a modification size, irrespective of checkpoint intervals. Meanwhile, the execution time for the *rebuild* scheme increases with the frequency of checkpoints. The increase in execution time is because of the overhead of checking and updating virtual to physical address mapping during each checkpoint. Table 4.4 shows that increasing the checkpoint interval from 10 to 100 milliseconds reduces execution time by ∼5× on average for the *rebuild* scheme across all virtual memory area sizes of alloc/free operations. When we further increase the interval to one second, beyond the execution time of the benchmark, the benefit of keeping the page table in DRAM appears for the *rebuild* scheme as the execution time is lower than the *persistent* scheme, highlighting the reason for performance overhead in *rebuild* scheme as maintenance of virtual to physical address mapping. In summary, keeping the page table in NVM with the *persistent* scheme benefits applications with minimal virtual address modifications. Access to page table entries for address translation gets the benefit of multiple levels of TLBs and intermediate caches, thus hiding NVM read latency while accessing page table entries for address translation in the *persistent* scheme.

In the following two subsections, we show the prototype implementation of two research ideas to demonstrate the benefit of using Kindle in exploring ideas and providing the initial results below.

4.2.2 SSP using Kindle

Shadow Sub-Paging (SSP) [25] provides memory consistency in NVM. It ensures consistency of memory modifications by maintaining a copy of unmodified data at cache line granularity. SSP allocates two physical pages for each virtual page and uses a remapping scheme at the cache controller hardware to route modifications at cache line granularity to alternate physical pages. SSP also extends the TLB by adding extra fields per entry to capture the supplementary physical page mapping and bitmaps (updated, current) to track the page containing the latest modification. SSP proposes a background OS thread to consolidate two physical pages but leaves out the detailed implementation and evaluation of *the consolidation* aspects in the paper.

In *Kindle*, we allocate the additional physical page in the page allocation routine call in gemOS. The original and extra page addresses and the bitmap values (commit, current) are recorded in a metadata area (i.e., SSP cache). We extend the page table walker hardware in gem5 to fill fields in the TLB during an address translation on TLB miss. TLB may contain translations for DRAM and NVM pages in a hybrid memory system, and the memory consistency requirement applies only to NVM pages. Therefore, in the prototype design, we use Model Specific Registers (MSRs) to communicate the virtual

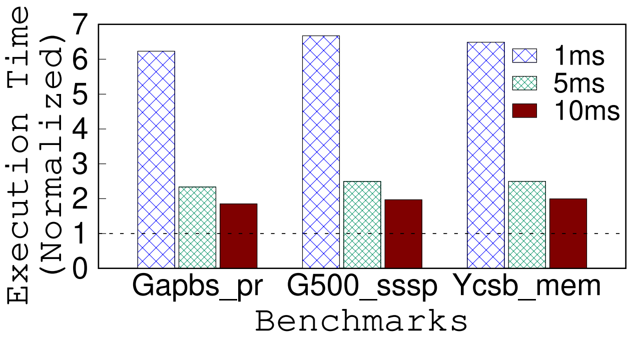


FIGURE 4.5: Influence of memory consistency interval on performance. Y-axis shows normalized execution time with no memory consistency.

address range corresponding to NVM allocation to hardware. We also use MSR to pass the base address of SSP cache to translation hardware in gem5. The address translation hardware checks the address range and sets the corresponding bit in the *updated* bitmap in TLB if a write happens to the NVM address range. The translation hardware generates a memory request to modify metadata in SSP cache when a consistency interval ends or a TLB entry eviction happens. We mark the entry as *TLB evicted* in the SSP cache.

We use a programming model in which the user demarcates the failure atomic section (FASE) in code using `checkpoint_start` and `checkpoint_end` calls. Usage of `checkpoint_start` enables custom hardware components in the address translation and cache controller hardware in gem5. A consistency interval of choice is set in gemOS. For example, setting consistency intervals as 5 msec ensures that at every 5 msec interval ends, and activities associated with `checkpoint_end` are performed, i.e., gemOS kernel instructs the address translation hardware to initiate a memory request to send all modified bitmap in TLBs to the metadata region. The gemOS kernel then calls `clwb` write-back instructions to flush all data and metadata updates in hardware caches to NVM. Physical page consolidation happens asynchronously; a thread periodically calls page consolidation routine to merge pages corresponding to evicted TLB entries by inspecting the SSP cache entries in gemOS.

Figure 4.5 shows the overhead introduced by SSP in making the memory state of applications consistent. This study used consistency intervals of 1, 5, and 10 msec. The page consolidation thread interval is fixed to 1 msec, as a lower interval would result in higher consolidation overhead. Figure 4.5 shows the execution time of applications normalized to the execution time with no memory consistency applied. Having a wider consistency interval (10 msec) reduces the consistency overhead as the number of metadata inspections and `clwb` calls to write-back cache lines reduce with a wider consistency interval. All applications in Figure 4.5 show an average $\sim 3\times$ reduction in memory consistency overhead with 10 msec compared to a 1 msec consistency interval.

Kindle provides an easy way to extend studies such as the influence of consistency interval on the application performance, and it also allows carrying out additional studies on the influence of page consolidation thread invocation frequency on an application by varying the thread time interval, which is not explored in original SSP proposal.

4.2.3 HSCC using Kindle

Hardware/Software Cooperative Caching (HSCC) [27] aims to utilize high NVM capacity in a hybrid memory system for improved system performance. HSCC maintains NVM and DRAM in a flat address space and uses DRAM as a cache managed by OS in a hardware/software cooperative manner. HSCC tracks the access count of NVM pages to select candidate pages for migration to DRAM and maintains an NVM-to-DRAM page mapping after migration. HSCC extends the page table and TLB to handle NVM to DRAM remapping and track the access count of NVM pages. NVM pages with an access count exceeding a specific fetch threshold value in a migration interval are selected for migrating to DRAM.

HSCC extends page table entry (PTE) to record DRAM and NVM page frame numbers, using 96 bits (12 bytes) for PTE as opposed to 64 bits in conventional x86-64 systems. In this case, the last level page table in HSCC can only map 341 pages (i.e., 4KB/12B), leaving 171 pages unmapped in a 2MB address region. In our implementation, we have designed NVM to DRAM mapping in a lookup table to avoid the previously mentioned PTE size issue. The mapping table entries can be looked up using both DRAM and NVM page frame numbers as an offset. We also maintain a pool of DRAM pages (512 pages), categorized as lists of free, clean, and dirty pages, updated at the start of each migration interval of 31.25 msec (equivalent to 10^8 cycles mentioned in the HSCC paper). The migration activity inspects the *page access count* maintained in PTEs corresponding to NVM pages and migrates the pages to the DRAM cache if the count exceeds the fetch threshold. The page access count is also maintained in TLB and is incremented if data access is missing in the LLC. The access count in TLB is written out to PTE on TLB eviction or once during the translation in a migration interval. We have not incorporated dynamic fetch threshold adjustment in our implementation but have fixed the threshold to

Table 4.5: Number of Pages Migrated

Benchmark	**Th-5**	**Th-25**	**Th-50**
Gapbs_pr	354	273	132
G500_sssp	4489	1475	1346
Ycsb_mem	23093	1661	221

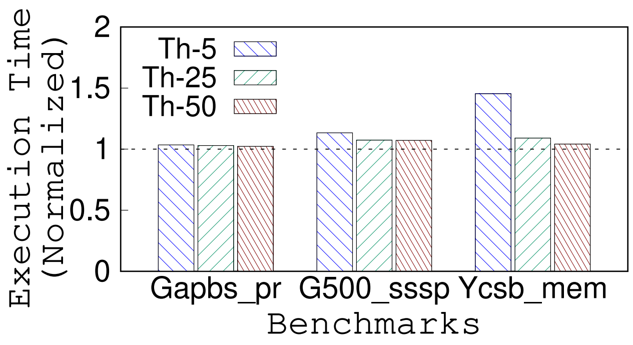


Figure 4.6: Influence of OS migration activities on application performance under DRAM fetch threshold 5, 25, and 50.

static values. The page access count in PTEs is reset in each migration interval to ensure that NVM pages from the most recent interval are considered for migration.

We investigated the migration-related processing overheads in the OS-mode and its impact on the execution time of applications. The candidate pages for migration are identified by inspecting the *page access count* maintained in the PTEs (by performing a software page table walk) corresponding to the pages mapped to NVM. Migrating a page to DRAM consists of two steps,

(i) *page selection*, selecting the destination DRAM page.

(ii) *page copy*, copying the page from NVM to DRAM.

Page selection includes allocating the destination DRAM page from the free, clean, or dirty list of DRAM pages. If any page is selected from the dirty list, then we copy back the page from DRAM to NVM before use. The page copy step includes flushing cache lines corresponding to the NVM page under migration before copying data from NVM to DRAM and then copying data to DRAM. The corresponding PTE entry is updated with DRAM page address, the access count in PTE is reset, and the corresponding TLB entry is invalidated. The page access count in all PTEs is reset, and corresponding TLB entries are invalidated in a migration activity to ensure that page accesses for the most recent interval are considered for migrations.

Table 4.6: Percentage of time spent for page selection and page copy in OS migration activity.

Benchmark	**Fetch Threshold**	**Page Selection (%)**	**Page Copy (%)**
Gapbs_pr	Th-5	1.74	98.26
	Th-25	1.92	98.08
	Th-50	2.06	97.94
G500_sssp	Th-5	37.35	62.65
	Th-25	1.37	98.63
	Th-50	1.39	98.61
Ycsb_mem	Th-5	21.71	78.29
	Th-25	19.16	80.84
	Th-50	1.86	98.14

Figure 4.6 shows the overhead of migration activities performed by OS. The figure shows the execution time of applications with migration (i.e., performing hardware and OS migration activities) normalized to the execution time without OS migration activities (i.e., performing only hardware migration activities) under different fetch thresholds; the fetch threshold decides the number of candidate NVM pages for migration. Two important factors influencing the execution time of an application with migration are—the overhead of activities performed by OS as part of the migration and the benefit in memory access time after migrating pages to DRAM. All applications in Figure 4.6 show migration overhead due to OS activities, and a higher value indicates that the overhead of activities performed by OS as part of the migration overshadows the benefit in memory access time after migrating pages to DRAM. Gapbs_pr shows the minimum overhead, indicating that Gapbs_pr has the maximum benefit in memory access time after migrating pages to DRAM. For all applications, the migration overhead reduces with an increase in the fetch threshold as the number of candidate pages migrated reduces with an increase in the threshold, as shown in Table 4.5; hence, the overhead of OS activity reduces. For example, Ycsb_mem showed $\sim$13$\times$ and $\sim$101$\times$ reductions in the number of pages migrated for Th-25 and Th-50 compared to Th-5 respectively.

Table 4.6 shows the percentage of time spent for activities associated with selecting a destination DRAM page (referred to as Page Selection) and copying the page from NVM to DRAM (referred to as Page Copy) as part of the total time required for the OS migration activities, a higher percentage value for a particular activity under a specific migration threshold signifies that the activity contributes predominantly to the total time required for OS migration activities under that migration threshold. Two factors contributing to page selection time are—*(i)* the number of pages migrated, and *(ii)* the availability of pages in the free and clean list of pages. The second component is relevant because if the page is unavailable in the free list, selecting a dirty page requires copying back data to NVM before using that page. Table 4.6 shows which is the major contributing factor among page selection and page copy in migration overhead. In the case of Gapbs_pr, page copy is the significant contributing factor in OS migration activities, and page selection time is less than ∼2% across all DRAM fetch thresholds as the number of pages migrated for Gapbs_pr (Table 4.5) is lower than the total number of pages in the DRAM pool (512 pages). Thus, most of requests for pages are satisfied from the free or clean list of pages, requiring no copying from DRAM to NVM before use. In absolute numbers, page selection takes 2924 nsec with Th-5, 1753 nsec with Th-25, and 1450 nsec with Th-50. G500_sssp with fetch threshold five spends ∼37% of time in for page selection, owing to large number of pages migrated with fetch threshold five (Table 4.5), similar is the case with Ycsb_mem with fetch threshold five. Even when relatively less number of pages migrated, page selection can consume significant portion of time in OS migration activities due to lack of free or clean pages. For example, Ycsb_mem with fetch threshold 25 takes ∼19% of time in page selection even with 1661 pages migrated (refer Table 4.5). Across all benchmarks and fetch thresholds, page copy occupies a significant portion of time in OS migration activities.

HSCC, in its original work, used ZSim [138], a user-level simulator that replays traces collected using Pin [58], and Zsim does not support OS-level simulation [27]. In HSCC, authors accounted for performance overhead by adding delays in the simulator for stages in NVM page caching such as flushing on-chip caches, DRAM page allocation, page migration, and execution of dynamic threshold adjustment algorithm. As *Kindle* provides a full-system simulation, it allows studying the actual effect of copying pages from NVM to DRAM and associated OS activities on page migration, for example, accessing page-table to find candidate pages for migration. *Kindle* also provides insights into the influence of other OS activities, such as context switches, and the effect of cache pollution due to OS activities on migration. On the contrary, user-level simulators like ZSim miss out on such insights about OS interactions in hybrid memory systems. Kindle also allows for separately investigating performance overhead due to hardware and OS activities, as shown in Figure 4.6 for OS migration activities.

Kindle allows researchers to quickly evaluate ideas crossing hardware-software layers on hybrid memory systems, as shown in the two prototype implementations.

4.3 Summary

The hybrid memory system provides the benefit of both DRAM and NVM technology. NVM offers high capacity and data persistence, and DRAM delivers lower read-/write access latency. The existing infrastructure for hybrid memory exploration crossing architecture-OS boundary using simulators such as gem5 is limited by the complexity of integrating NVM support in Linux for gem5 and the simulation overhead of Linux due to OS services and functions running. Incorporating NVM support in Linux requires modifying the memory management system in Linux to expose NVM to applications through memory allocation APIs such as `mmap` and allocate physical pages from NVM. The changes in Linux also require modifying physical memory allocation routines such as buddy allocator and persisting changes to associated data structures to consistently maintain memory page allocation metadata for NVM pages to retain it across reboots.

This chapter introduced an open-source framework, *Kindle*, based on gem5 and gemOS for hybrid memory exploration in architecture and operating systems. *Kindle* enables hybrid memory with NVM and DRAM in a flat addressing mode, allowing users to study the memory behavior of applications with required hardware-software changes. Using *Kindle*, we study end-to-end overhead in maintaining execution context using periodic checkpointing to achieve process persistence under two schemes to keep the page table in a consistent manner. *Kindle* also provides a quick way to study and prototype solutions for hybrid memory systems. We show prototype implementation of state-of-the-art hardware-software hybrid memory schemes, SSP [25] and HSCC [27], using *Kindle* to demonstrate its efficacy in realizing complex design goals and analyzing new insights. While *Kindle* can provide process persistence, it also has limitations originating from the trace-based approach used in its design to run applications, similar to any other trace-based simulators such as ChampSim [139], as the trace file only captures the non-speculative path of application and loses possible thread interleaving in multi-threaded applications, etc. We target memory system study using *Kindle* and hence focus on tracing memory operations. *Kindle* also enables studying NVM memory technologies beyond Phase-Change Memory (PCM). We configure the NVM interface in gem5 with PCM configuration (a widely used NVM technology) to showcase the utility of *Kindle* and the process persistence feature. However, we can use *Kindle* to study other NVM technologies by changing NVM interface parameters in gem5. The scope for such studies increases the value of *Kindle* in hybrid memory research.

Achieving process persistence requires consistently maintaining the memory state, and with *Kindle*, we can analyze overheads associated with memory state persistence. Designing schemes for memory state persistence requires special consideration since the memory area consists of different types, such as heap and stack, having distinct usage characteristics. Thus, the overhead of consistently maintaining the memory state may differ based on its nature and usage pattern. Therefore, we need tools to study and understand the unique characteristics of these memory segments and design efficient approaches for memory persistence. In the next chapter, we introduce an open-source tool named *SniP* to examine the unique characteristics of the stack. *SniP* is a framework for efficient run-time tracing of the stack for multi-threaded programs. We later propose a hardware-assisted periodic checkpointing mechanism for the stack based on the insights provided by *SniP*.

Chapter 5

Framework for Multi-threaded Program Stack Tracing

Stack is an important entity in software programs as it helps to efficiently implement language constructs such as subroutine call and return, allocation and freeing of local variables. Stack also plays a key role in program analysis as programs leave their footprint in the stack throughout their execution. Programmers can gain insights into the behavior and security loopholes, such as buffer overflow, by analyzing the stack usage. However, programmers face some unique challenges in performing analysis related to run-time stack usage. Unlike other program memory areas where the programmer explicitly controls the usage (and can profile), the stack areas are hidden from the programmer. The usage of the execution stack is transparent to programmers as the compiler inserts instructions to manage the stack for the correct implementation of program logic, like function calls.

These unique properties of the stack make it a fascinating element in the domain of program analysis. It is easy to observe that run-time stack usage can not be foreseen, which makes static analysis techniques ineffective. Therefore, we depend on dynamic run-time techniques for stack analysis. Debugging tools (e.g., GDB [56]) can provide a lot of insights into stack usage but require manual intervention and, therefore, do not scale for long-running applications. On the other hand, run-time memory access tracing techniques provide flexibility to perform automated tracing and analysis. Dynamic binary instrumentation (DBI) tools such as Intel Pin [58] are widely used to trace the program at run-time and perform offline analysis using variety of techniques [140–142].

DBI tools are very convenient as they require no preparation and can trace the entire program [57, 58]. One of the approaches adopted for run-time stack analysis is to perform full program tracing and filter stack specific accesses during the offline analysis phases. However, stack analysis through a trace-driven approach by tracing the program results in large trace files and higher tracing time (§ 5.2.1). In addition, the offline analysis process requires the stack virtual address ranges to filter stack accesses in the trace. For this purpose, we can use memory layout information provided by the operating system. However, capturing the stack range information for different threads in multi-threaded applications is challenging. The stack areas for threads can be allocated anywhere in the program address space depending on the state of the address space and the OS support for dynamic allocation. For example, stack areas for the threads created using the POSIX thread library in the Linux OS are allocated at run-time in the noncontiguous regions of the address space. So one possible simple approach to capture the stack range of a thread is to inspect its virtual address space layout to identify stack. In Linux, we can inspect the virtual address space layout of a process in the *maps* file of the *proc* pseudo filesystem. The *maps* file has *[stack]* field to show the stack range, but the *[stack]* field shows the stack range of main thread in the threads' *maps* file. POSIX thread library allocates stack areas of threads using *mmap* system call, and in current versions of Linux (5.11.0-49), it is not evident which *mmap* area is the stack area of a thread by inspecting its *maps* file.

In this chapter, we propose a framework that takes a different approach for tracing stack. We filter the stack accesses during tracing to avoid the additional overheads (trace size and tracing time) and remove the requirement of identifying the stack virtual address ranges during offline analysis. We propose techniques to identify and manage an information base for the stack address ranges of different threads, which is consumed by the DBI tool to apply stack address range filters at the time of tracing. `SniP`, a **S**tack tracing framework for multi-threaded applications, is built on top of Intel's Pin [58] (**niP**). `SniP` captures and manages the information regarding the stack of different threads using an OS-level extension where the operations relating to the stack areas (starting from the creation) are monitored. Further, the up-to-date information related to the stack areas is shared with the user-space tracing process to enable target tracing.

To the best of our knowledge, `SniP` is the first framework for tracing stack in multi-threaded programs; showcasing reduced trace sizes (75× less trace file size with key-value store TinyDBM [143]) and tracing time (24× less tracing time with Python3 HTTP server) compared to tracing the programs in entirety. Furthermore, with the targeted tracing

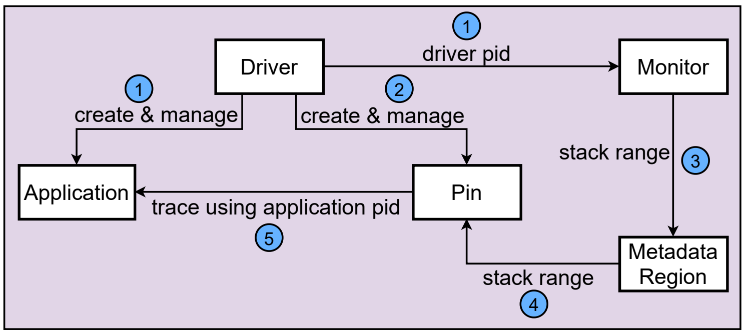


FIGURE 5.1: Schematic diagram of `SniP`

capability of `SniP`, the offline analysis process becomes simpler for multi-threaded applications, which enables seamless integration with existing trace based analysis tools. `SniP` provides strength to the multi-threaded application stack analysis spectrum as multiple tools and use-cases can be built around `SniP`. We show two such sample use-cases using `SniP` (§ 5.2.2) to demonstrate its capabilities.

Availability of SniP

`SniP` is open-sourced and available at https://github.com/arunkp1986/SniP.git

5.1 Design and Implementation

The high-level design of `SniP` is shown in Figure 5.1. The major components of `SniP` are the user space program (Driver) and the OS module (Monitor). The user space program is responsible for executing the application and Intel's dynamic binary instrumentation tool Pin [58]. The monitor module captures the application threads' stack range at the point of thread creation and stores this information in the metadata region (Figure 5.1). This design enables the monitor module to record stack ranges of newly created threads throughout the application's lifetime. When tracing starts, Pin consumes the stack range information stored in the metadata region and generates traces only for access to the stack.

We implement `SniP` in Linux (kernel version 4.19.83). As Figure 5.1 shows, the user space driver program creates two child processes, one for executing the application (to be traced) and the other for Pin. The driver program also passes its process ID (step ①) to the monitor module (implemented as a kernel module) through a character device.

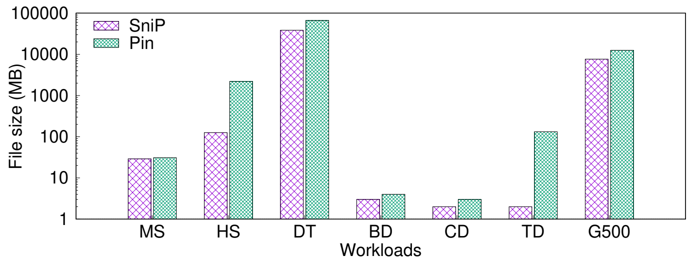
The process ID enables the monitor module to identify the application using the parent-child relationship between the driver and the application process. The monitor module intercepts thread creation of the application process by hooking the `wake_up_new_task` function in `clone` system call handler of the Linux kernel and saves the stack range (step ③) of each application thread in the metadata region. The metadata region is exposed from the kernel module using the kernel `sysfs` API which the Pin process can access using the file API. The driver passes the application process ID to Pin (step ②) for tracing. Pin starts reading stack ranges from metadata area (step ④) and generate output by tracing the application (step ⑤).

In the current implementation, we need to turn off Address Space Layout Randomization (ASLR) to prevent the `exec` system call (in create & manage) from changing the stack range of the application's main thread (after its fork from the driver process). We intend to address this limitation by hooking the `exec` system call handler in the future. All the configurations for using the Pin tool remain unchanged in the proposed system, and therefore, `SniP` does not hamper the vast feature set provided by Pin.

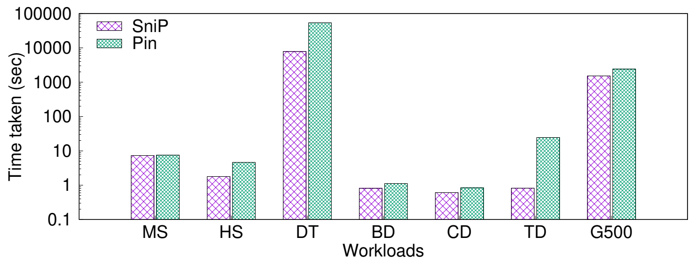
5.2 Evaluation

5.2.1 Stack tracing with `SniP`

To analyze the trace size and tracing time, we used the following workloads: merge-sort (MS) with four threads, each sorting 250 numbers, Python3 default HTTP-server (HS) used to download 4 files with each 200+ MB size, decision tree classifier (DT) from Python scikit-learn library used with 77280 training and 19315 testing samples, in-memory key-value stores BabyDBM (BD), CacheDBM (CD), TinyDBM (TD) from Tkrzw [143] performing 50 set and 50 get operations, Graph500 (G500) benchmark [137] BFS kernel run with scale parameter as 10. Figure 5.2 shows the benefits of using `SniP` for multi-threaded program stack tracing where we compare trace file size and tracing time between full program tracing using Intel Pin [58] and stack tracing using `SniP`. For long-running applications such as Python HTTP-server, machine learning algorithms, and in-memory key-value store applications, the difference between Pin and Snip is significant. For example, a reduction of 17× in trace file size and 2.5× in trace time is observed for the HTTP server workload. The benefit of `SniP` is marginal for short-running applications with heavy stack usage, such as merge-sort which extensively uses the stack for recursive calls.



(a) Trace file size comparison (the lower the better)



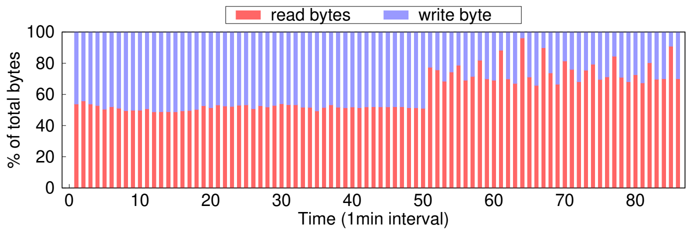
(b) Tracing time comparison (the lower the better)

FIGURE 5.2: Comparison of size and time for `SniP` stack tracing w.r.t Pin full tracing [MS: Merge-Sort, HS: Python3 Http Server, DT: Decision Tree Classifier, BD: BabyDBM, CD: CacheDBM, TD: TinyDBM, G500: Graph500 BFS]

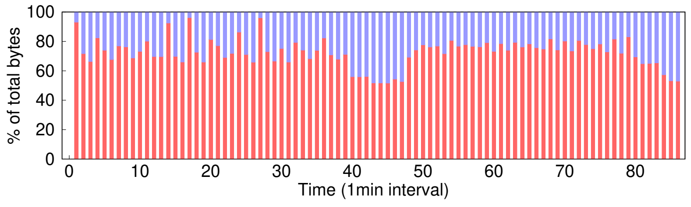
TABLE 5.1: Comparison of size and time for SniP stack tracing w.r.t Pin full tracing

workload	Method	Size	Time
Memcached-YCSB (workloada load)	Pin	830 MB	127664 ms
	SniP	119 MB	31619 ms
Memcached-YCSB (workloada run)	Pin	233 MB	53337 ms
	SniP	74 MB	12822 ms
MySQL-Wikipedia (load)	Pin	495 GB	214 Hrs
	SniP	6.8 GB	10 Hrs
MySQL-Wikipedia (execute)	Pin	345 GB	78 Hrs
	SniP	6.6 GB	6 Hrs

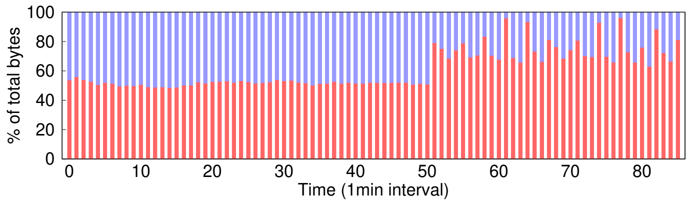
To confirm the robustness, we also used `SniP` to trace long-running applications such as MySQL with Wikipedia [144] and Memcached with YCSB workloads[135]. Table 5.1 shows $\sim 6.9\times$ reduction in trace file size with `SniP` for Memcached with YCSB workload performing *workloada* load and $\sim 4.15\times$ reduction in trace time for *workloada* run. `SniP` reduced trace file size by $\sim 72.7\times$ and trace time by $\sim 21\times$ for MySQL (version 8.0.42) with Wikipedia load. We used *batchsize* as 64 for Wikipedia (execute) and 128 for Wikipedia (load) in the benchmark configuration file.



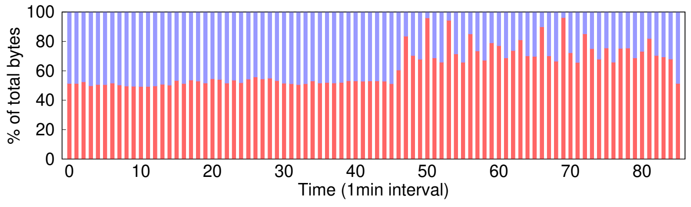
(a) Decision Tree Classifier



(b) Extra Trees Classifier



(c) Gradient Boosting Classifier



(d) Gaussian Naive Bayes

FIGURE 5.3: Percentage of read and write access to stack of Machine Learning classification algorithms.

5.2.2 Use cases

In this subsection, we show different usage scenarios of `SniP` for tracing the stack. We show that `SniP` can be easily extended to build use-cases around stack analysis.

5.2.2.1 Tracing ML Classification Algorithms

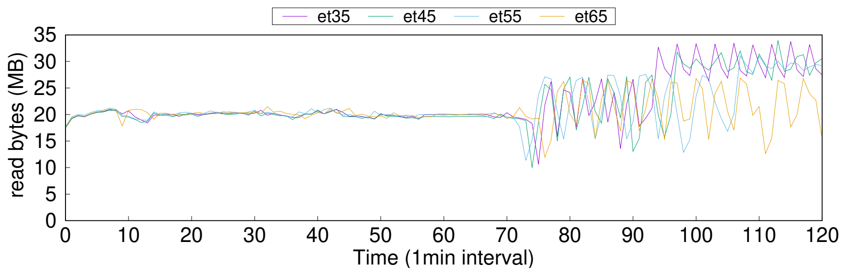
The popularity of machine learning in a wide range of domains is driving software and hardware changes in computer science. ASICs and accelerators are designed to meet specific performance demands [145]. Understanding the memory access patterns of machine learning algorithms helps designers perform optimizations [146] at software and hardware levels. Moreover, understanding the memory access patterns is important for systems with non-volatile memory due to its internal micro-architecture difference with DRAM and asymmetric read, write access time [147]. First commercially available persistent memory, Intel's Optane DC persistent memory performance depends upon access size, access type (read vs. write), and pattern [148, 149].

Given the popularity of machine learning algorithms and the availability of non-volatile memory, it will be of great benefit to programmers to know the expected performance of machine learning algorithms executing on systems with non-volatile memory. In this context, we show a use-case for `SniP` by collecting the stack read and write access patterns of popular machine learning classification algorithms such as decision trees, extra trees, gradient boosting, and Gaussian naive bayes in Figure 5.3. These classification algorithms used 75 features from DeepDetect [150] and used 77280 samples for training and 19315 for testing. During this tracing, Decision Tree Classifier created 22, Extra Trees Classifier 14, Gradient Boosting Classifier 22, and Gaussian Naive Bayes 21 threads.

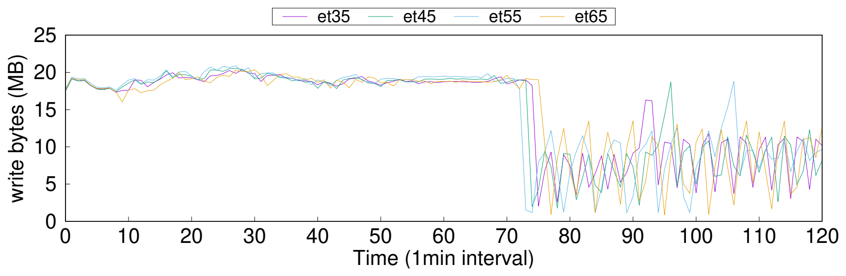
Figure 5.3 shows fraction of read and write operations to stack area in 1 minute intervals. We observed that stack accesses in these algorithms are dominated by reads. Using `SniP`, we also performed a further study on the influence of feature set size on stack usage for ML classification algorithms by taking the extra-tree classifier as an example. Figure 5.4 shows that the feature set size did not influence stack read/write usage in the initial stage of the extra trees classifier algorithm, but impacted in the later stage of the algorithm. Figure 5.4 also shows that there is an increase in bytes read from and decrease in bytes written to stack in the later stage of extra tree classification. We suspected it to be due to data access pattern differences in training and test phases of the classification algorithm but confirmed that the pattern is present even after separating out training and test phases of extra tree classification, which implies the access pattern is a property of the algorithm in this case.

5.2.2.2 Detecting uninitialized memory usage in stack

Memory corruption bugs such as buffer overflow, Use-After-Free (UAF), and uninitialized memory use happen due to incorrect programming practices or mistakes [151]. These are common memory bugs in a program, and static code analysis tools are effective in catching bugs based on a set of predefined rules, but they have a high false positive rate [152].



(a) Bytes read from stack area in MB



(b) Bytes written into stack area in MB

FIGURE 5.4: Read and write behaviour of Extra Trees Classifier with varying feature set size [et35, et45, et55, et65 represents 35, 45, 55, 65 feature sizes]

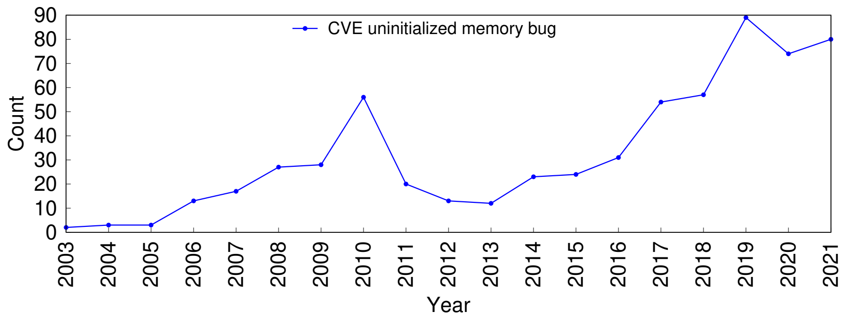
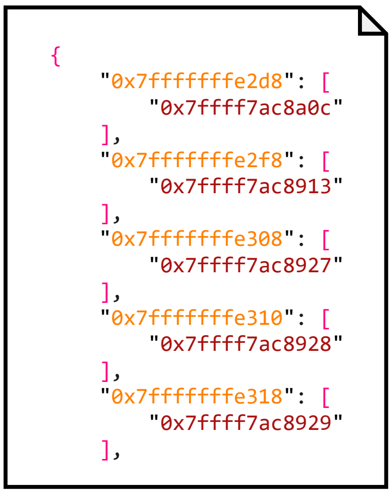


FIGURE 5.5: CVE uninitialized memory bugs reported over years

In these common bugs, the uninitialized memory usage can expose kernel data to user programs, thus breaking the security guarantee [153].

```
int main(){
  int data[20];
  int i;
  for(i=0;i<20;i++)
    data[i] += 1;
  for(i=0;i<20;i++)
    printf("Value at data[%d] = %d\n",i,data[i]);
  return 0;
}
```

LISTING 5.1: Prototype of uninitialized memory bug.



```
{
    "0x7fffffffe2d8": [
        "0x7ffff7ac8a0c"
    ],
    "0x7fffffffe2f8": [
        "0x7ffff7ac8913"
    ],
    "0x7fffffffe308": [
        "0x7ffff7ac8927"
    ],
    "0x7fffffffe310": [
        "0x7ffff7ac8928"
    ],
    "0x7fffffffe318": [
        "0x7ffff7ac8929"
    ],
```

Figure 5.6: JSON file format, Python parser output of uninitialized memory bug

The number of uninitialized memory bugs reported at CVE over the years in Figure 5.5 indicates the importance of detecting it. We show that `SniP` can detect uninitialized memory bugs, such as shown in code listing 5.1.

In this use-case, we developed a prototype to identify uninitialized memory bugs by parsing the stack trace generated by `SniP`. The parser tagged instances where read to any stack location happened before write, thus indicating an uninitialized memory usage bug in the code. `SniP`'s trace contained details such as access type (read/write), instruction address, and memory access virtual address. The parser generated a JSON file containing uninitialized memory bug virtual addresses and instruction addresses as shown in Figure 5.6. Even though we parsed the `SniP` trace in an offline mode to detect bugs, this can be done to detect uninitialized memory bugs at run-time by running the parser alongside the traced program.

5.3 Related Work

Static and dynamic analysis are two well known techniques for program analysis. Static analysis parses the code or uses abstract models, whereas dynamic analysis executes the program and observes the runtime behavior; hence, no abstractions or approximations are required [154]. Dynamic analysis requires inclusion of analysis routines within the program, which is called binary instrumentation. Binary instrumentation can be done statically (binary is modified before the program runs) or dynamically (modification occurs at run-time) [155].

Dynamic binary instrumentation (DBI) is very convenient for the users to trace and analyze programs as it requires no preparation and can be applied in a flexible manner. Valgrind [57] is a DBI framework that uses shadow values, which maintains a copy of the program state containing register values and user-mode address space. Dynamo performs run-time performance optimization. It generates an optimized version of the hot code sequence in the program to a code cache by interpreting the instructions [156]. Intel Pin [58] uses dynamic compilation to instrument programs while they are executing. Users place analysis routines at points of interest in binary using instrumentation routines. Pin also allows attaching and detaching to a running process, and we use this feature of Pin in `SniP` for tracing. Chabbi et. al. integrated a call path collection library with Pin that collected call path context for each executed instruction using a shadow stack [157].

`SniP` generates trace for program stack accesses; stack trace holds important pieces of information in debugging and program analysis, as shown by Schroter et. al. [158] in their empirical study on the usefulness of stack trace. Experienced users can write their own tools for analyzing trace generated by `SniP` or use existing tools such as STAT [140], which is used in debugging thousands of processes by sampling stack trace to form process equivalence class, then performing root cause analysis on the representative processes from equivalence class. Stack trace can also be used to compute similarities between bugs while reporting [159] or to identify dependency conflicts in projects using an automated approach [141]. Stack trace plays an important role in system security as well, as highlighted by Feng et al., by using stack trace for anomaly detection, they extracted return address information from the stack for anomaly detection [142].

5.4 Summary

Stack holds important pieces of information to gain insights into the program behaviour which can help with debugging, security and performance analysis. In this chapter, we discussed the challenges of tracing the stack accesses in multi-threaded applications using existing tools like Intel Pin. We introduced `SniP`, an efficient stack tracing framework for run-time tracing of application stack using techniques that combine tools like Pin and intelligent extensions to the OS. We implemented `SniP` in the Linux OS and demonstrated its efficacy in terms of its light tracing footprint and flexibility in terms of applicability. Our experiments with a set of multi-threaded applications show that `SniP` not only outperforms Intel Pin in terms of resource usage but also makes the offline analysis of stack access traces comparatively simpler. Furthermore, we demonstrated the utility of `SniP` to perform stack analysis with contemporary application use cases by performing offline analysis of the stack access traces collected using `SniP`.

Based on this stack analysis, in the next chapter, we show the benefit of a checkpoint-based mechanism for stack persistence and the inefficiency of adapting existing generic memory persistence mechanisms for the stack region. We propose *Prosper*, a hardware-software (OS) co-designed checkpoint approach for stack persistence.

Chapter 6

Program Stack Persistence in Hybrid Memory Systems

Process persistence using checkpoint techniques [35–37] has gained popularity with the emergence of hybrid memory systems consisting of traditional volatile random access memory (DRAM) and byte-addressable non-volatile memory (NVM).

To achieve process persistence through checkpoints, it is required to persist the process state periodically. The process state, which consists of the CPU register state, memory state, and other associated states, should be checkpointed so that the process can resume from the last execution point across system restarts [1]. Capturing periodic snapshots of the memory state of processes consisting of different mutable memory segments (e.g., heap and stack) presents non-trivial challenges regarding checkpoint complexity and checkpoint size [160]. Therefore, many research contributions treat the general problem of consistently persisting the memory state in isolation by employing techniques at both the software layers [8, 26, 30–32, 161] and at the hardware layer [25, 29, 80, 162].

In the context of process persistence, the OS-level checkpoint solutions are more practical than the generic memory persistence solutions, considering the semantic proximity of the OS to the notion of processes. For example, it is non-trivial for a memory persistence technique at hardware or user space to demarcate the boundaries for the memory state of a process spanning across the user and OS layers. On the other hand, OS-level checkpoint procedures can potentially leverage the additional hardware support for memory persistence (with some adaptation) to simplify the complexities associated with periodic memory checkpoints. In this chapter, we demonstrate that the state-of-the-art solutions

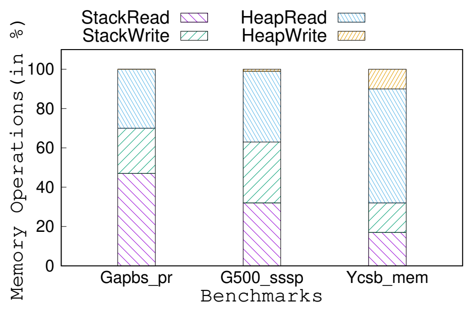


FIGURE 6.1: Memory operations to the stack and heap regions (in %) demonstrating the significance of stack operations.

are not suitable for all memory segments, specifically for memory segments with unique characteristics such as *the program stack*. Furthermore, we propose *specialized hardware extensions* to efficiently checkpoint the stack region and show that it can improve the overall efficiency of OS-level checkpoint solutions when combined with tailored generic memory persistence techniques.

Typically, the stack segment size is smaller than the heap segment, but *the number of operations* on the stack can be significant for some applications. Figure 6.1 shows the fraction of memory operations in the stack region for three representative benchmarks from the graph and cloud workloads—Gapbs_pr [134], G500_sssp [137], and YCSB [135]. We traced these benchmarks for stack and heap operations using Intel Pin [58] on a four-core Intel(R) Xeon(R) W-2104 system for the highest weighted interval identified by SimPoint [163]. For Gapbs_pr, 70% of operations (reads and writes) are performed to the stack regions. We highlight some of the important usage characteristics of the stack segment (used to implement function calls and store program objects in local scope) that differ from other memory segments before discussing the applicability of state-of-the-art techniques.

Usage pattern: The stack exhibits a grow and shrink pattern, i.e., back-and-forth movement of the stack pointer (SP) during the lifetime of processes (and threads), which differs from the allocate-use-free pattern of other memory segments such as heap.

Write characteristics: The stack region is not only write-intensive (Figure 6.1), but also maintains activation records across function invocations and returns, resulting in a significant number of writes to a cluster of memory addresses.

Indirect usage: Unlike heap, where the application layer uses the region through explicit allocation and de-allocation, for a stack, the compiler or run-time system introduces the required stack operations that are hidden from the application layer. The role of the OS is a little different for the stack as it handles the growth and shrinkage of a stack in an on-demand fashion [6].

For stack persistence, periodic commit-based techniques *operating in tandem* with application execution [29, 32, 162] may give rise to inefficiencies because they can not adequately address the subtleties of stack usage. *First*, the stack usage pattern may lead to unnecessary operations in periodic commit-based techniques, which operate in cooperation with application execution because the SP may grow and shrink multiple times during an interval. For example, the stack grows from address A to B, then from B to C, then shrinks from C to B during a checkpoint interval. The stack pointer at the commit point is address B, so the active stack region for the checkpoint interval is between addresses A and B. Accesses between B and C are beyond the active stack region for the checkpoint interval. We show that having future knowledge regarding the value of SP at commit points (referred to as *SP awareness*) significantly improves the efficiency of existing techniques (§ 6.1.1). We designate a memory persistence mechanism to be *SP aware* if the overhead incurred to persist the stack region is predominantly determined by the active stack region at the commit point. *Second*, considering the write-intensive nature of the stack region, maintaining the stack in NVM leads to performance and endurance issues [147, 164]. Approaches that do not employ periodic checkpoints have to maintain the stack in NVM along with the meta-data required to achieve consistency. *Third*, many existing approaches, specifically the logging-based approaches—redo, undo [17, 102], and its variations [31])—require invocation of special APIs from the application layer for different events such as load, store, and commit. Considering the indirect usage of the stack region, non-trivial extensions are required for the compiler to insert calls appropriately for different operations in the stack region.

Checkpoint-based solutions allow allocation and usage of the stack in DRAM while achieving persistence by copying the dirtied stack memory addresses into NVM at the end of every checkpoint interval. An OS-level periodic checkpoint solution for the stack region can address the previously mentioned challenges for the following reasons. *First*, the checkpoint mechanism is *SP aware* as the activity performed by OS at the checkpoint time (i.e., copying the dirtied stack memory into NVM) depends upon the *active* stack region(s). *Second*, hosting the stack region in DRAM alleviates the problem of excessive writes to NVM. Moreover, periodic checkpoints allow higher levels of write coalescing, addressing the inefficiency concerns due to the *write characteristics*. *Third*, an OS-layer checkpoint solution for stack regions can be used in a generic manner without requiring any special support from the compiler/run-time, addressing the challenges arising due to the *indirect*

Table 6.1: Comparison of existing memory persistence mechanisms.

Property	**SSP [25]**	**JUSTDO [31]**	**SoftWrAP [32]**	**Timestone [8]**	**Romulus [26]**	**HOOP [29]**	**Prosper (our work)**
Achieves process persistence	✗	✗	✗	✗	✗	✗	✓
Works without compiler support	✓	✗	✗	✗	✗	✓	✓
Stack pointer awareness	✗	✗	✗	✗	✗	✗	✓
Allows stack in DRAM	✗	✗	✓	✗	✗	✗	✓

usage of the stack memory. One of the challenges in capturing the snapshot for stack region is the amplification of checkpoint size due to limited hardware support for efficient dirty tracking. For example, as we show in § 6.1.2, dirty tracking of the stack region at the OS page granularity (e.g., SoftDirty [165], LDT [104]) results in significantly large checkpoint sizes compared to dirty tracking at the sub-page granularity.

Observations. Without OS-level adaptations, existing memory persistence techniques *in their current form* are inadequate to achieve efficient process persistence. Even with OS-layer adaptations, there can be inefficiencies when existing techniques are used for stack, considering the usage and access pattern of the stack region. A summary of existing techniques and their applicability is presented in Table 6.1. While the OS-layer checkpoint approach for the stack can be seen as an extension to checkpointing other non-memory states of the process (e.g., the register state), checkpointing overhead for the stack should be minimized. Moreover, generic hardware-layer solutions for dirty tracking at sub-page granularity [108, 109] require special hardware support and are used to address specific usage scenarios such as disaggregated memory and capturing VM snapshots. The flexibility required by the OS to manage and consume dirty tracking information in a generic manner is not trivial to achieve using these hardware extensions.

Design Approach. We propose *Prosper*, a hardware-assisted checkpointing mechanism for the stack region, to achieve efficient process persistence. The hardware assistance provided by *Prosper* can track stack modifications at a finer granularity with very little overhead, reducing data copy overheads compared with dirty tracking at page granularity. To provide greater flexibility to the OS, we propose a hardware-software co-design approach where the OS can control and take advantage of *Prosper* to checkpoint the stack region efficiently. Further, the design of *Prosper* allows the OS to combine existing hardware-layer solutions for memory persistence with *Prosper* for different memory regions in the process address space.

Key results. Our experiments show that the performance overhead introduced by the *Prosper* hardware extension is, on average less than 1% (maximum ∼3%). Leveraging dirty tracking at sub-page granularity, *Prosper* significantly reduces (on average ∼4×) the amount of data copied during checkpoint and improves the overall checkpoint time.

For a workload performing sparse writes to the stack region, *Prosper* reduces checkpoint size by 99% compared to page granularity dirty tracking, resulting in ∼22× improvement in the time taken to checkpoint the stack region. *Prosper* performs better than state-of-the-art NVM memory persistence schemes such as Romulus [26] and SSP [25] for providing stack persistence. *Prosper* provided up to 3.6× reduction in stack persistence overhead with respect to SSP and a maximum of 1.27× reduction with respect to the page-level Dirtybit mechanism. A process persistence solution combining *Prosper* and SSP results in up to 2.6× improvement in achieving memory state persistence compared to a scenario when only SSP is used for the entire memory.

6.1 Motivation

For the experiments presented in this section, we traced the stack usage of some memory-intensive application benchmarks (Figure 6.1) using SniP [136], an open-source stack tracing framework, on a four-core Intel(R) Xeon(R) W-2104 system. For Gapbs_pr, the input parameters are: `kronecker` graph with 2^{27} vertices, 1000 iterations, $1e^{-4}$ tolerance, and 16 trials. G500_sssp uses scale as 16 and edge factor as 64. For Ycsb_mem, we traced Memcached while performing YCSB `workload-A load` followed by `workload-B run`. We traced for the highest weighted interval identified by SimPoint [163].

6.1.1 Inefficiency due to Stack Pointer Unawareness

Existing techniques without SP awareness perform non-trivial operations (e.g., create a log entry) throughout the interval to maintain the persistence state of the stack. The overhead of such operations depends upon the specific persistence mechanism under consideration. For example, a log-based scheme may create log entries for each write to the stack region, resulting in overall performance overhead proportional to the number of writes and the

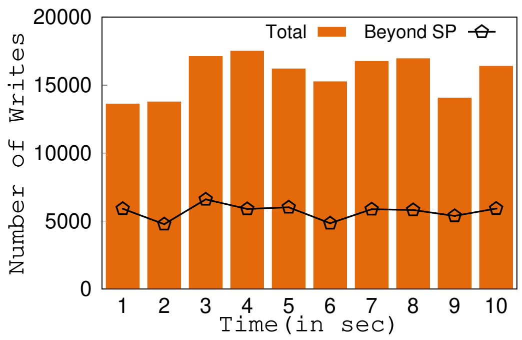
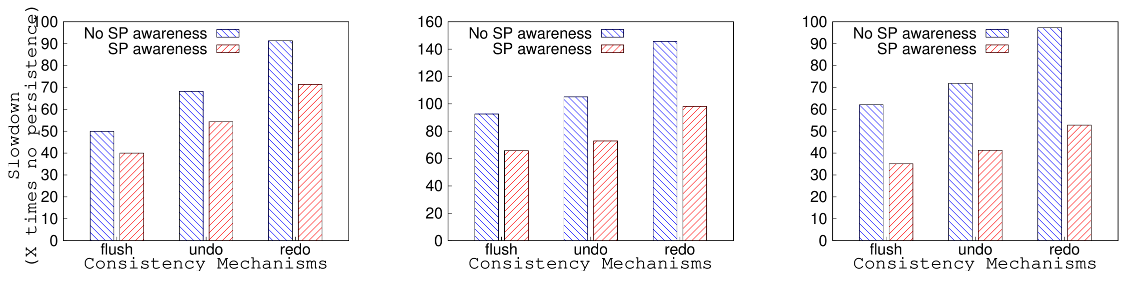


FIGURE 6.2: Number of total stack writes and writes beyond final stack pointer (SP) aggregated at 100 intervals with each interval of 10ms for `Ycsb_mem` benchmark.

cost of creating and serializing log entries. In this case, the overhead incurred to persist stack is not determined by the active stack region at the end of an interval; therefore, a log-based scheme is not SP aware. Towards capturing the overhead quantitatively, we calculate the number of stack modifications during an interval that is beyond the active stack region (i.e., beyond the value of SP) at the end of the interval using the access traces. Figure 6.2 shows the total number of stack writes and writes beyond the final SP, aggregated for 100 consistency intervals with each interval of 10ms duration (used for process persistence in Aurora [1]) for the Ycsb_mem benchmark. On average, more than 36% of the stack modifications are beyond the final SP for Ycsb_mem, and the behavior is similar for other benchmarks such as Gapbs_pr and G500_sssp (not shown in Figure 6.2). The impact of SP unawareness on a persistence mechanism can be significant, considering the non-negligible proportion of operations turning out to be wasteful.

Next, we analyze the benefit of incorporating SP awareness into common memory persistence mechanisms such as *flush*, *undo*, and *redo* to understand the extent of performance penalty these mechanisms suffer due to *SP unawareness*. We replayed the read/write accesses in the stack memory traces using a custom program on an Intel(R) Xeon(R) Gold 6226R system with NVM (Optane DCPM [147]). The custom program performed an equivalent number of reads/writes in the trace with the configured memory persistence methods (i.e., *flush*, *undo*, or *redo*) in *"No SP awareness"* scenario, whereas it applied the persistence method only to the active stack region in *"SP awareness"* scenario. The flush technique used a `clwb` instruction after every store operation. Note that, inherently, the above memory persistence mechanisms can not have SP awareness as they have to intervene and perform operations for every write to stack. The trace-driven replay allows us to incorporate SP awareness in these techniques for analysis purposes.



(a) Gapbs_pr (b) G500_sssp (c) Ycsb_mem

FIGURE 6.3: Slowdown of primitive memory persistence mechanisms with SP awareness and No SP awareness with respect to no persistence (DRAM). SP awareness: stack pointer awareness.

Figure 6.3 shows the potential benefit of incorporating SP awareness in *flush*, *undo*, and *redo* techniques to achieve stack persistence. The results show the execution time for different mechanisms with and without SP awareness (in NVM) normalized to execution time when no persistence mechanism is used, i.e., stack region is allocated in the DRAM. We observe two important performance trends from this experiment. *First*, as shown in Figure 6.3, all persistence mechanisms benefit from having *"SP awareness"*; the average performance improvement compared to *"No SP awareness"* scenario across all workloads is observed to be 30%, 31%, and 33% for *flush*, *undo*, and *redo*, respectively. For example, the execution time for Gapbs_pr while using *flush* with SP awareness is 8.5 seconds, 10.6 seconds without SP awareness, and 0.2 seconds with no persistence. Even though Ycsb_mem has comparatively fewer stack modifications (~15% in Figure 6.1), it has more number of stack modifications beyond the active stack region compared to Gapbs_pr and G500_sssp, thus benefiting more due to *"SP awareness."*

Second, even with SP awareness, the overhead is significant—more than 35× slowdown across all benchmarks. Techniques requiring the maintenance of the stack in NVM and lacking the capability to merge the consistency-preserving operations incur significant overhead, considering the write-intensive nature of stack operations. For example, for *flush*, every store to the stack region would result in a write-back to NVM. A checkpoint solution, apart from allowing the stack allocation in DRAM, provides enough opportunities for coalescing as the checkpoint is performed only at the end of a checkpoint interval. Moreover, checkpoint techniques are required to perform limited operations during an interval (i.e., dirty tracking) and hence, the amount of wasted work can be minimized with efficient dirty tracking.

6.1.2 Inefficiency of Page-level Dirty Tracking

The primary sources of overhead in checkpointing any memory region are,

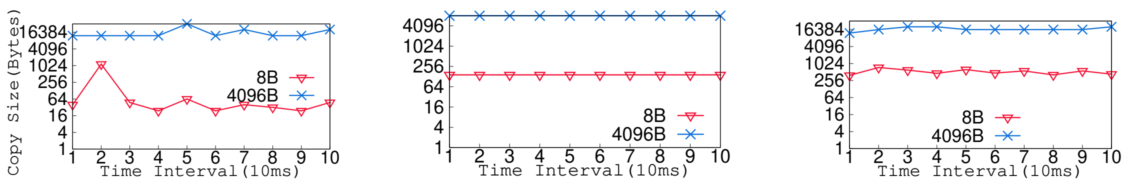
1. Dirty tracking overhead, i.e., overhead associated with designating dirty status of memory chunks.
2. Data copy overhead, i.e., time taken to copy modified data from DRAM to NVM.

Dirty tracking techniques in contemporary systems depend upon the information gathered during the virtual to physical address translation. There are two standard techniques for dirty tracking at page-level granularity.

- Using the dirty bit indicator set by the address translation hardware (e.g., the dirty bit in the page table [104]). We refer to this as the Dirtybit approach.
- Disabling the write access by write-protecting the pages in the page table [165, 166]. We refer to this approach as a write-protection based approach.

The write-protection bit-based scheme forces page faults on write access to a page during a dirty tracking interval. This scheme removes the write permission bit from the page table entries (PTEs) for all physically mapped addresses at the start of a tracking interval. Therefore, the first write to such pages in an interval would generate a page fault where the system software (OS) may record the page as dirty, which can be used at the end of the tracking period. In the Dirtybit approach, the dirty bit in the page table entries (PTEs) is reset at the start of a tracking interval. The hardware page-table walker (PTW) sets the dirty bit in PTE if there is any writes to the pages corresponding to the PTEs. At the end of the tracking interval, the OS can examine the PTEs to determine the dirty pages.

Both page granularity dirty tracking techniques require the OS to walk the page table to collect dirty page information and prepare the PTEs for the next interval. However, the write-protection-based approach incurs additional overhead due to page faults and may lead to significant overheads, as shown by Singh et al. [104]. On the other hand, the Dirtybit approach is nimble and is supported by default in most hardware architectures. LDT [104], a technique leveraging dirty bit support of x86-64 systems, shows that LDT can reduce the dirty tracking overhead in the Linux OS compared to the write-protection-based technique [165]. In this chapter, we use LDT [104] as the reference implementation to design Dirtybit-based approach for comparative analysis. For the stack region, dirty



(a) Gapbs_pr (b) G500_sssp (c) Ycsb_mem

FIGURE 6.4: Comparison of checkpoint size in page and byte level dirty tracking of stack modifications. Y axis shows copy size in page (4096 Byte) and sub-page (8 Byte) granularity tracking for benchmarks in Figure 6.1.

tracking overhead should be minimized to reduce the wasteful work during the tracking interval. However, the granularity of tracking memory modifications is limited by the address translation unit, typically an OS page [104, 166]. This can be a bottleneck in terms of *increased checkpoint size*, resulting in higher copy overheads.

Ideally, a dirty tracking approach should track modifications at sub-page (or byte-level) granularity and copy only modified bytes at the end of a checkpoint interval. This conventional wisdom of tracking modifications at lower granularity [21, 25] is much more crucial for the stack than other memory areas since the stack modifications majorly happen at lower granularity due to procedure calls or local variable writes. To understand the extent of reduction in checkpoint copy size with dirty tracking at byte-level (sub-page) granularity for stack modifications, we compared data copy size in byte-level dirty tracking with conventional page-level granularity.

We post-processed the traces of benchmarks in Figure 6.1 to calculate the data copy size with *page* and *8-byte* granularity dirty tracking at 10ms intervals for the stack regions. Figure 6.4 compares the data copy size for the page (4KB) and 8-byte granularity dirty tracking for the stack regions. Dirty tracking at sub-page byte granularity for stack reduces the checkpoint size by a factor of 300× for Gapbs_pr, 56× for G500_sssp, and 33× for Ycsb_mem.

Summary. Observations presented in this section form the basis of *Prosper*, where we make a case for tracking stack modifications at a finer (byte) granularity to reduce the checkpoint size. Apart from dirty tracking at a finer granularity, the proposed system by virtue of its design, should allow stack allocation in DRAM, better symbiosis with the OS-layer process persistence mechanisms, support efficient software implementation to capture stack checkpoints, and limit the penalties of SP unawareness by efficient dirty tracking.

6.2 Dirty Tracking with Prosper

To achieve process persistence through OS-level periodic checkpointing, the memory state of the process needs to be persisted in a crash-consistent manner along with other process states. Hardware-only approaches face non-trivial challenges for stack persistence due to SP unawareness and integration difficulties with the OS layer checkpoint procedures. As summarized previously, a periodic checkpoint approach for stack persistence has many advantages. However, tracking stack modifications at sub-page granularity to reduce the checkpoint size requires additional hardware support. *A desirable solution should provide low-overhead dirty tracking of the stack region while organically supporting the OS-layer checkpointing.* One possible design choice can be *an OS-hardware co-design* where the following non-trivial design challenges are addressed.

(i) OS should notify hardware about stack address range and start/stop tracking at the beginning/end of any checkpoint interval. Thus, separation of responsibilities along with an efficient communication protocol between the hardware and software (OS) components is necessary. For correctness, the co-design approach must confirm synchronization between the OS and hardware to ensure the quiescence of dirty information before consuming it from the OS.

(ii) Hardware tracker monitors stack modification and should not stall load/store requests from the processor to the stack memory. Tracking must be done out of the critical path of demand requests from the processor. Hardware tracker should also generate minimum memory requests as part of tracking to reduce its footprint in the memory hierarchy.

(iii) The OS and hardware components should coordinate information sharing regarding the tracking granularity, address ranges of stacks used by different execution entities (such as threads), and their corresponding meta-data regions to record tracking information efficiently, which the OS later consumes. Across different events, such as checkpoints, context switches, and others, correctness and efficiency should be ensured.

Prosper uses a hardware-software (OS) co-design approach in which the OS records stack address range, and the hardware component tracks stack modifications. Even though *Prosper* is proposed for tracking stack modifications, its generic design can be leveraged

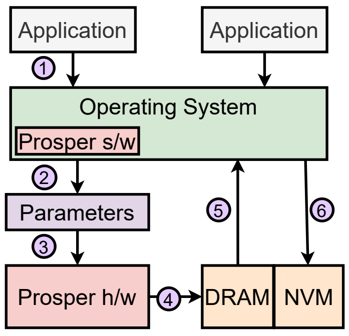


FIGURE 6.5: Schematic diagram of Prosper.

to track modifications to any virtual address range. For example, we can use *Prosper* to track modifications to dynamically allocated virtual address ranges in the heap.

Figure 6.5 shows the division of responsibilities and handshakes between the hardware and the OS components in *Prosper*.

6.2.1 Prosper Software

The software (OS) component assists the hardware component by providing required information through a set of parameters, addressing the first and third communication challenges between software and hardware. *Prosper*'s OS component records the stack address range of an application thread (① in Figure 6.5) and passes it along with other information, such as tracking granularity and address of memory area to record metadata about stack modifications through parameters (②). *Prosper*'s h/w component uses these parameters (③) to track the application's stack modifications. *Prosper* saves the tracked dirty information in memory (④) using a bitmap, addressing the second challenge regarding metadata storage, and OS utilizes it (⑤) to decide which stack areas are modified in the current checkpoint interval. A bit in the dirty bitmap corresponds to a stack address range based on the tracking granularity. OS finally initiates a copy of data (⑥) in memory after ensuring all dirty tracking information is in a consistent state. Before performing bitmap inspection, the OS ensures quiescence of the bitmap area using a two-step process,

(i) Instruct the *Prosper* h/w to flush all tracked dirty information.

(ii) Ensures completion of flush related activities by checking a hardware indicator.

The OS may perform other activities (e.g., preparing for copy) between the two steps to reduce overheads due to stalling in the second step. The OS clears the recorded dirty

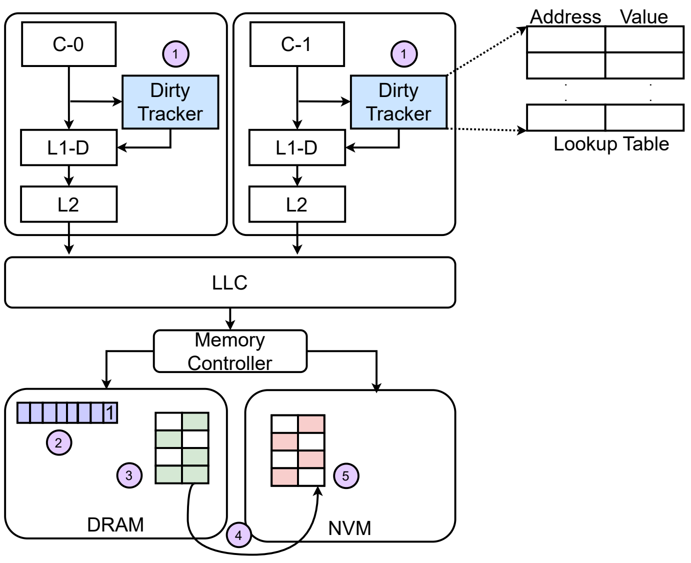


Figure 6.6: Schematic diagram of Prosper hardware.

bits before starting the next checkpoint interval to ensure the correct recording of dirty information in the next interval.

The efficiency of OS processing depends on performing targeted processing of the stack region where the OS examines the dirty meta-data and performs copy operations only for the active stack region. To avoid walking the entire meta-data area to clear the bits set in the last iteration, the OS should know the maximum active stack region during the checkpoint interval. The *Prosper* hardware tracks this information and shares it with the OS at the end of the checkpoint interval. The OS component also handles events such as context switches and process/thread migration, which are not shown for simplicity.

6.2.2 Prosper Hardware

The nucleus of *Prosper*'s hardware component is a dirty tracker hardware, as shown in Figure 6.6. The tracker (shown as ① in Figure 6.6) is active during a checkpoint interval (i.e., between a checkpoint start and end). The tracker monitors memory store operations and filters the ones to the stack region without interfering with the progress of the store operation, addressing the second challenge (mentioned previously). The primary task of the tracker is to set bit(s) in the bitmap area (shown as ②) corresponding to the addresses of the filtered store(s). Each bit in the bitmap is associated with an address range in the stack based on the tracking granularity. The tracker can be configured with granularity as multiples of 8-byte.

With *Prosper*, the bitmap and volatile state of an application reside in DRAM (shown as ③ in Figure 6.6). A per-thread persistent stack is maintained in NVM (not shown in figure), which is consistently updated in two steps. In the first step, after inspecting the bits in the bitmap area, the OS copies (④) data to a temporary buffer in NVM (shown as ⑤ in Figure 6.6) at the checkpoint end. In the second step, the per-thread persistent stack in NVM is updated using the data copied to the temporary buffer in the first step. To reduce the overheads due to bitmap inspection and copy operations, the OS looks for coalescing opportunities within every eight bytes of the bitmap.

We build different design elements through the following series of questions related to the maintenance of the bitmap area during the tracking interval.

(A) When should the tracker issue bitmap store?

A straw-man approach could be to issue bitmap-store requests as and when a bit needs to be set in the bitmap due to stack modification. However, the straw-man strategy could interfere with the demand stores from the core because of the additional bitmap-store requests it generates. Therefore, we use a *lookup table* as a small cache within the tracker to coalesce the bitmap store requests for a given stack range. Bitmap store requests are generated due to,

(i) Eviction of an entry from the lookup table due to lack of space in the table.

(ii) The entry has reached the coalescing threshold as explained below.

(iii) At the end of a checkpoint interval.

Each entry in the lookup table is a tuple of <*bitmap location address (64bits), bitmap value (32bits)*> (Figure 6.7). The lookup table has parallel search capability using the bitmap location address as the key. The target address for each bitmap store is searched in the table where a hit results in an update of bitmap value and a miss results in creating a new entry in the table, evicting an existing entry if required. We considered two design choices while creating a new bitmap entry in the table.

1) Accumulate and Apply: Tracker creates an empty entry in the lookup table without loading the old bitmap value from memory. Bitmap value changes are accumulated in this entry until a bitmap store request is generated for this entry. The store request is converted into a load request for the old bitmap value; then, the accumulated bitmap value is merged with this old value and stored back if required. Loading the old bitmap value is delayed until a bitmap store request is initiated. The advantage of this approach is that the table entry is allocated instantaneously without waiting for the load operation of the old bitmap value from memory to be completed.

2) Load and Update: Tracker issues a load request for the old bitmap value from memory into the table and updates the bitmap value in the table. The table contains the latest bitmap value when a bitmap store request is generated for this entry. The advantage of this approach is that no additional load apart from the initial load is required when the exact bitmap location is modified after evicting from the table multiple times in an interval. The drawback is the need to delay bitmap entry allocation until the load for the old value is completed.

We use the first option, *Accumulate and Apply* in *Prosper* for creating a new bitmap entry in the table as it allows quick allocation of lookup table entries. This avoids complications of reserving an entry marked as "not ready" in the table for the duration of load and queuing of stores corresponding to the same entry.

(B) What are the coalescing thresholds?

As bits corresponding to stack modifications have coalesced in the lookup table entry, the tracker should decide on an appropriate event to issue bitmap store requests. The tracker should not be too eager or lazy; the former may increase the interference in the memory hierarchy, while the latter can result in more evictions to accommodate new bitmap store requests. In the current design, the tracker issues a bitmap write request when the number of bits set in any lookup table entry reaches a high-water-mark (HWM) threshold. An optimal HWM attempts to strike a balance between memory bandwidth usage for bitmap store requests and the number of evictions.

(C) What is the eviction strategy for the look-up table?

As the lookup table has a limited size, the tracker should employ an eviction policy to accommodate new bitmap store requests. Under the current eviction policy, the tracker selects victims based on the number of bits set in the *bitmap value* of all table entries. The tracker evicts the entries for which the number of bits set in the lookup table is less than a low-water-mark (LWM) threshold. The tracker may evict a random entry if no suitable entries adhering to the LWM criteria are found. The rationale for this simple LWM-based design is to give priority to table entries corresponding to frequently modified stack areas. Moreover, function *calls* and *returns* may touch stack areas momentarily without a lot of reuse, which should be evicted from the table with higher priority.

HWM threshold value decides the lifetime of a lookup table entry before generating a bitmap store request. A high HWM threshold value allows for longer retention of lookup table entries and enables setting more bits before issuing a bitmap store request. Thus, applications with spatial locality of reference to program stack benefit from high HWM and also reduce the number of bitmap loads and stores generated by such applications compared to a low HWM threshold value. LWM threshold value decides the number of vacant entries created in the lookup table during eviction. A high LWM creates more vacant entries in the lookup table during eviction, and applications without spatial locality in stack access require more entries in the lookup table. These applications benefit from high LWM value.

6.2.3 Multi-threading Support

As each software thread has its own stack, the stack can be tracked on the logical CPU on which the thread is scheduled. *Prosper*'s per hardware thread dirty tracker can track the stack modifications of software threads and set bit(s) in the dedicated bitmap areas. During the process (and thread) context switch, the OS is required to save and restore the dirty tracker state (i.e., configuration and bitmap information), similar to other architectural states. During a context switch, quiescence of the bitmap area for outgoing context is ensured using the two-step process mentioned in § 6.2.1. Specifically, as soon as the OS decides the incoming context, it instructs the *Prosper* h/w to flush entries from the lookup table to update the bitmap area of the outgoing context. Next, the OS ensures the completion of flushing before resuming the incoming context by loading the saved *Prosper* hardware state.

One of the challenges in multi-threaded scenarios is to handle inter-thread stack modifications, i.e., when one thread of a process accesses (writes to) the stack region of another thread of the same process. Inter-thread stack modification is possible because threads share the address space and can access each other's stack region. However, we observed that such inter-thread stack modifications are rare in applications such as the ones used in § 6.1. Nonetheless, the issue can be addressed by combining *Prosper* with existing privilege separation mechanisms in multi-threaded applications [167] where inter-thread stack accesses can be tracked by forcing OS interventions such as raising page faults. For *Propsper*, this can be achieved by maintaining separate page table entries for stack address ranges of different threads. The permission bits in page tables are set such that a thread has write access to its own stack but has read-only access to the stacks of other threads. On a write fault to any stack address, OS allows the write to proceed after setting the

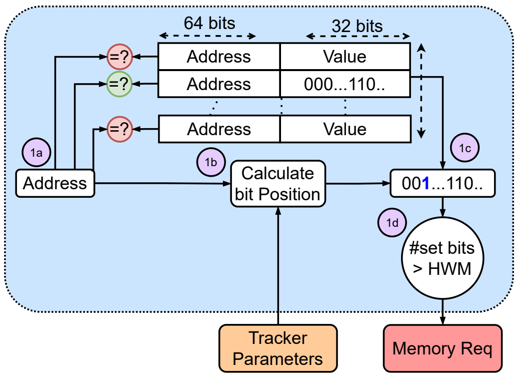


FIGURE 6.7: Working of the Prosper hardware tracker.

required bits in the bitmap area. Similar to the design of Wang et al. [167], changes to the stack page table of any thread need to be propagated to the page table entries of other threads in this design.

6.2.4 Implementation

The dirty tracker hardware employs mechanisms to identify and filter stores of interest (SOI). The demand store requests from the processor are inspected by the *Prosper* dirty tracker while being forwarded to the L1D. To filter the SOIs, a comparator circuit compares the store addresses against the address range set by the OS using two custom per-core model-specific registers (MSRs). The hardware dirty tracker identifies and filters required information from SOIs without impacting their progress. The comparator circuit is placed near L1D to track accesses as early as possible before they are translated or merged. In comparison to employing tracking further down in the memory hierarchy (e.g., at the memory controller), this approach has two primary benefits,

(i) h/w filtering logic to identify SOIs becomes simple as the virtual address range for the stack is contiguous which need not be the case when filtering is based on physical addresses.

(ii) the tracker has immediate visibility of all stack modifications, which may not be possible at further levels in the memory hierarchy because the accesses can be served from the upper-level cache(s).

Figure 6.7 shows the working of per-core dirty tracker hardware after filtering the SOIs. The tracker uses the tracking granularity and bitmap base address values passed through two additional MSRs to calculate the corresponding bitmap address and the bit position in the *bitmap value* ($\textcircled{1b}$). The lookup table coalesces bitmap store operations to reduce the number of bitmap store requests generated by the tracker. The tracker performs concurrent address comparison to search the bitmap location address in the lookup table $\textcircled{1a}$, by comparing it against addresses stored in the lookup table. Next, if an entry exists, the tracker sets the appropriate bit in the *bitmap value* of the existing entry in the lookup table. Otherwise, it creates a new entry where only the corresponding bit for the SOI address is set in the *bitmap value* (*Accumulate and Apply* approach). The tracker finally issues a bitmap store request $\textcircled{1c}$ when the number of bits set in the lookup table value is higher than a high-water-mark (HWM) $\textcircled{1d}$. The bitmap store request for a given entry is performed in two steps—first, the old *bitmap value* is loaded by generating a load request to the address of the bitmap entry, and second, the old value is merged with the value in the lookup table and stored back, if necessary. The eviction operation also follows the same path, albeit the entry is marked free.

Entries in the lookup table are flushed by performing eviction of all entries when the OS indicates the end of a checkpoint interval. In this case, the OS polls the tracking hardware to ensure all tracker-generated operations (load and store) are completed before proceeding further. The tracker maintains outstanding load and store request counters to ensure the completion of all in-flight operations and coordination with OS. We implement *Prosper* hardware component on gem5 [50, 51] (version 21.2.1.1).

End-to-end Checkpoint Solution

A typical process checkpoint mechanism for hybrid memory systems consists of an OS layer to capture the process state periodically. Thus, the OS should support hybrid memory (DRAM + NVM) and baseline checkpoint operations for different process states. The OS on a system with *Prosper* must also incorporate additional support for *Prosper* software component. While gem5 [50, 51] supports Linux in full system mode, Linux does not support the baseline features mentioned above for a hybrid memory. Therefore, to design an end-to-end checkpoint solution using *Prosper*, we create an application checkpoint-restore infrastructure on *Kindle*, the hybrid memory framework with NVM and DRAM.

The memory management subsystem in *Kindle* supports hybrid memory where the process uses DRAM and stores checkpoints in the NVM (refer to chapter 4). *Kindle* also enables periodic checkpoint operations for a process by passing the checkpoint interval as a parameter.

The baseline checkpoint mechanism in GemOS, which is part of *Kindle*, captures all process states (including the stack) in an incremental manner and stores them in the NVM. The

Table 6.2: gem5 Configuration

Parameter	**Used Setting**	**Setup**
CPU	3GHz	I&II
L1-D/I	32 KiB/core (8 way, 3 cycles)	I&II
L2	512KiB/core (16 way, 12 cycles)	I&II
L3	2 MiB/core (shared) (16 way, 20 cycles)	I&II
MSHRs	16, 32, 32/core L1-D, L2, L3	I&II
Cache line size	64 B in L1, L2, L3	I&II
DRAM interface	DDR4-2400 16x4	I&II
NVM interface	PCM ‡	I
NVM Write buffer	48	I
NVM Read buffer	64	I
Memory capacity	3GB DRAM + 2GB NVM	I
Memory capacity	32GB DRAM	II
‡PCM timing parameters based on [119]		

memory modifications for the process are tracked at a page granularity using Dirtybit approach (§ 6.1.2). For byte granularity checkpointing with *Prosper*, we incorporate the *Prosper* software component into the GemOS component of *Kindle* to perform different handshake operations with the underlying *Prosper* hardware component using custom MSRs. To test the correctness, we emulate abrupt system crashes by killing the gem5 component of *Kindle* on the host while an application process is active within *Kindle* framework. After the crash, we restarted the gem5 component of *Kindle* and observed that the process within *Kindle* framework restarted from the last checkpoint successfully.

6.3 Experimental Setup

We performed two sets of experiments using two setups (referred to as Setup-I and Setup-II). The first set of experiments demonstrates the end-to-end improvement of checkpoint performance with *Prosper*, while the second set of experiments analyzes the hardware dirty tracking overhead introduced by *Prosper*. We used gem5 (version 21.2.1.1) [50] with configurations mentioned in Table 6.2 for the experiments. Table 6.2 lists the NVM parameters that are different from the default NVM interface (i.e., NVM-2400 1x64) in gem5. The parameters not mentioned in Table 6.2 are set to the default settings of the gem5 simulator. Unless explicitly mentioned, we use the lookup table size as 16, HWM as 24, LWM as 8, tracking granularity as 8 bytes, and checkpoint interval as 10 ms for all experiments (refer to § 6.2.4).

6.3.1 Checkpoint Performance

We used Setup-I to demonstrate the efficacy of *Prosper* through the following experiments.

1) Performance of *Prosper* to persist the stack in a consistent manner vis-a-vis other memory persistence mechanisms such as Romulus [26], SSP [25], and page-granularity checkpoint using hardware dirty bit support [104] (referred to as Dirtybit).

2) Comparatively analyze the performance of achieving process memory state persistence by combining different stack persistence techniques with SSP.

3) Performance of *Prosper* for different stack usage patterns using micro-benchmarks (Table 6.3).

We modified GemOS [52] for this set of experiments, running on gem5 with DRAM + NVM hybrid memory. Further, we implemented Romulus and SSP in GemOS.

Romulus [26] provides memory persistence by maintaining twin copies of data in NVM, with one copy considered backup and the other as main. The authors have proposed Romulus as a user-space library; however, since the compiler manages the stack operations, we have implemented Romulus as a hardware-software co-design to interpose stack modifications. The hardware component logs the address and size of stack modifications. The software component copies modifications from main to backup by inspecting the log entries created by the hardware.

SSP [25] ensures memory persistence at cache line granularity using a shadow paging scheme. SSP maintains two physical pages for each virtual page and distributes modifications across these two pages at cache line granularity. Using an OS thread, SSP consolidates two physical pages associated with an inactive virtual page. We have varied the OS page consolidation thread invocation frequency as 10 μs, 100 μs, and 1ms in the experiments (OS thread invocation frequency is not mentioned in the paper). At the end of each consistency interval, SSP writebacks modified cache lines using *clwb*, sends updated bitmap in extended TLB to the SSP cache, and applies it on the commit bitmap maintained in NVM.

To study the performance of *Prosper* with different stack usage scenarios, we compare it against the page-level dirty-bit mechanism (Dirtybit) applied for the stack. The micro-benchmarks in Table 6.3 capture different stack access categories by operating on an array allocated in the function scope. The *Sparse* micro-benchmark dirties four bytes of each memory page used for stack across recursive invocation of a function. The *Random* micro-benchmark writes to a fixed number of random words, while the *Stream* micro-benchmark writes to the entire stack region. The *Sparse*, *Random*, and *Stream* micro-benchmarks are designed to explore the best, average, and worst case performance of *Prosper*, respectively.

The *Quicksort* and *Recursive* micro-benchmarks capture the stack access pattern with repeated function calls and returns. Finally, we use *Normal* and *Poisson* micro-benchmarks to study the performance when the number of stack accesses between two computation code fragments follows a probability distribution. To introduce stochastic behavior in terms of the number of accesses between two compute code blocks, the number of stack writes is chosen from a normal distribution (with $\mu = 63$ and $\sigma = 20$) for the *Normal* workload. For *Poison* workload, the number writes to stack between two compute code blocks are chosen from a Poison distribution with $\lambda = 63$. The compute code block in these workloads increments a register value one thousand times.

6.3.2 Tracking Overhead Experiments

We used gem5 with Setup-II configurations and Linux (kernel version 5.2.3) to measure the dirty tracking overhead of the *Prosper* hardware. We modified the Linux kernel to incorporate the system software component of *Prosper*. A kernel thread coordinates with the *Prosper* hardware to control and collect dirty information for the stack region(s) in every 10ms interval. At the start of an interval, the kernel thread communicates tracking parameters and stack address range to *Prosper* hardware using custom MSRs. At the end of the interval, the thread synchronizes with *Prosper* hardware to ensure the completion of tracking activity before examining the dirty tracking meta-data.

We used SSSP from Graph500 [137], PR from GAPBS [134], SPEC CPU 2017 (SPEC-speed) benchmarks [168], and micro-benchmark Stream (Table 6.3) for this study and allowed the benchmark application to run for one minute (as warm-up time) before starting the incremental checkpoint. The kernel thread performed a total of 6000 checkpoints at 10ms intervals.

6.4 Evaluation

In the first set of experiments, we evaluate the performance benefits of *Propser* in providing process persistence by comparing it against state-of-the-art memory persistence

Table 6.3: Microbenchmarks with different stack usage scenarios

Category	**Name**	**Description**
Access Pattern	Random	Write to random elements of an array allocated in the stack
	Stream	Write to all elements of an array allocated on stack sequentially
	Sparse	Write to 4KB spaced elements of an array allocated on the stack
Function Invocation	Quicksort	Sorting elements of an array allocated in the heap
	Recursive	Recursive function invocation with parameterized call depth
Access Intensity	Normal	Normally distributed stack writes between computation operations
	Poisson	Poisson distributed stack writes between computation operations

mechanisms using Setup-I. Furthermore, we investigate the benefit of integrating *Prosper* with state-of-the-art memory persistence mechanisms for achieving memory state persistence (heap and stack combined). We analyze the performance of *Prosper* for different stack usage patterns. In the second set of experiments, we evaluate *Prosper*'s hardware overhead, the dirty tracker's sensitivity to HWM and LWM thresholds using Setup-II, and the energy requirements of *Prosper*'s hardware.

Performance of Prosper: Figure 6.8 shows the performance comparison of *Prosper* with existing NVM memory persistence mechanisms—Romulus, SSP, and page-level Dirtybit scheme, when used to achieve persistence of the stack region of different applications.

Figure 6.8 shows the execution time of different applications with one of the memory persistence mechanisms applied for the stack normalized to execution time without memory persistence. For SSP, the invocation interval for the OS page consolidation thread is varied from $10\mu s$ to $1ms$ (referred to as SSP-$10\mu s$, SSP-$100\mu s$ and SSP-$1ms$).

Propser performs better than Romulus, and SSP for all workloads and performs better than page-level Dirtybit for all except Random and Stream. SSP and Romulus require the memory area to be allocated in NVM. In contrast, Dirtybit and *Prosper* allow allocating stack in DRAM, which leads to improved performance due to the differences in access latency between DRAM and NVM. For SSP, the page consolidation OS thread also contributes to the performance overhead, as merging pages may interfere with the application execution.

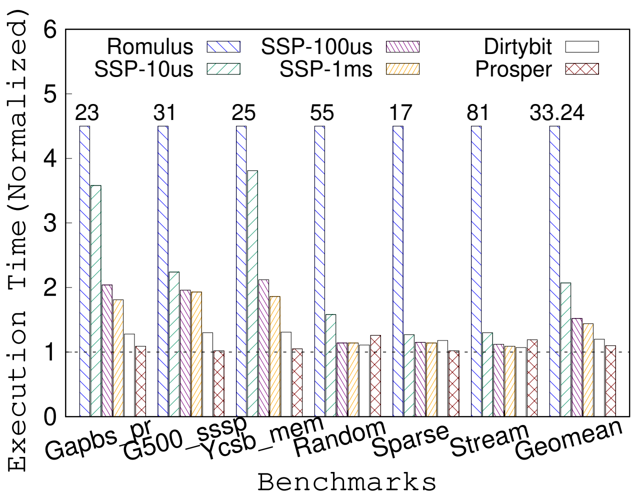
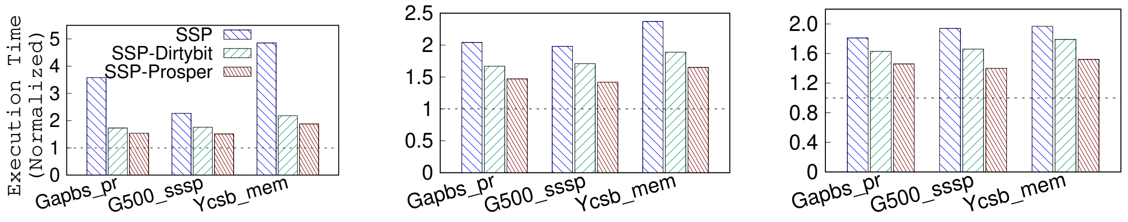


FIGURE 6.8: Application performance comparison with different memory persistence mechanisms applied to stack. Y-axis shows application execution time with memory persistence normalized to no persistence, the lower the better.

In Dirtybit and *Propser*, metadata inspection and data copy happen at the end of the checkpoint interval. Romulus results in significant performance overheads across all workloads as the hardware generates redo log entries for all stack modifications, and the software may copy overlapping addresses from the primary memory location to the backup memory location (in the absence of coalescing as is the case in our implementation). On the other hand, Dirtybit and *Propser* coalesce bitmap updates for the same location and avoid redundant copying at the interval end. The performance benefit of *Prosper* compared to Dirtybit results from the reduction in copy size due to the sub-page granularity dirty-tracking support of *Prosper*, as well as the efficient inspection/preparation of the dirty information metadata (§ 6.2). *Prosper* results in an average of 2.1× (maximum of 3.6× for Ycsb_mem) reduction in stack persistence overhead compared to SSP-10μs and a maximum of 1.27× reduction in stack persistence overhead for G500_sssp with respect to Dirtybit. The stack persistence overhead for SSP decreases with an increase in page consolidation OS thread invocation interval from 10 μs to 1ms. For example, ∼2× reduction for Gapbs_pr from 10μs to 1ms is observed, but SSP incurs higher overheads compared to *Prosper* even with 1ms setting. *Prosper* efficiently provides stack persistence compared to other existing memory persistence mechanisms applied for stack persistence. *Prosper* design allows changing tracking granularity based on the dirty behavior of an application or turning it off to use a page-level Dirtybit scheme.

Process memory state persistence: The stack and the heap regions are two primary mutable regions in a process. To analyze the performance overhead of achieving memory



(a) 10 μs (b) 100 μs (c) 1ms

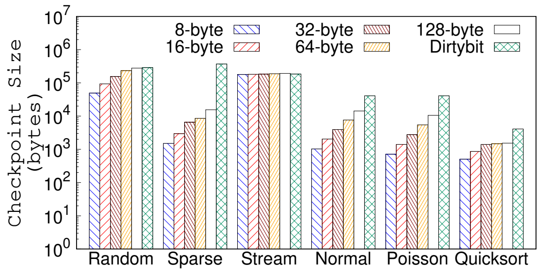
FIGURE 6.9: Performance comparison for heap+stack area persistence, with SSP with different intervals.Y-axis shows the execution time normalized to no persistence, lower the better.

state persistence in different applications, we used different combinations of SSP, Dirtybit, and *Prosper* for the heap and stack segments. The combinations used for this experiment are—*(i)* SSP for both stack and heap, *(ii)* SSP for heap and Dirtybit for stack, and *(iii)* SSP for heap and *Prosper* for stack. The design of *Prosper* is inclusive enough to integrate with other memory persistence schemes, such as logging for heap area.

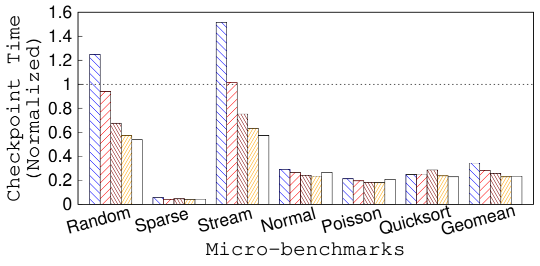
Figure 6.9 shows the execution time of different applications with one of the memory persistence mechanisms applied for heap and stack normalized to execution time without memory persistence. Figure 6.9 clearly demonstrates the benefit of combining *Prosper* or Dirtybit with SSP to achieve memory persistence compared to using SSP for the entire memory area under all three OS thread invocation intervals. SSP-Prosper performed better than SSP-Dirtybit and SSP across all three SSP page consolidation thread invocation interval scenarios. SSP-Prosper provided an average 2× (maximum of 2.6× for Ycsb_mem) reduction in memory persistence overhead compared to SSP with 10 μs thread invocation interval and an average of ~1.4× and ~1.3× reduction in memory persistence overhead compared to SSP with 100μs and 1ms, respectively. An increase in SSP OS thread invocation interval benefits all three stack memory persistence mechanisms, with SSP showing a 2.4× reduction in memory persistence overhead for 1ms compared to 10μs for Ycsb_mem.

A combination of *Prosper* with other existing memory persistence mechanisms can provide persistence for the entire memory area with minimum overhead.

Prosper with different stack usage scenarios: We study the performance impact of different stack usage patterns with *Prosper* using micro-benchmarks in Table 6.3. For this experiment, five different tracking granularities (8 byte, 16 byte, 32 byte, 64 byte, and 128 byte) are used with a fixed checkpoint interval of 10 ms. Figure 6.10 shows the performance of *Prosper* for different workloads vis-a-vis the baseline, i.e., the Dirtybit (page-granularity) scheme.



(a) Checkpoint size comparison with different tracking granularity and with page granularity dirty tracking. Y-axis is in log scale.



(b) Checkpoint performance improvement with `Prosper`. Y-axis shows checkpoint time with *Prosper* normalized to checkpoint time with page granularity dirty tracking.

FIGURE 6.10: Stack checkpoint performance for different micro-benchmarks with *Prosper*.

Figure 6.10(a) shows the checkpoint size for the stack averaged over all checkpoint intervals. Figure 6.10(b) shows the time taken to complete the checkpoint with *Prosper* normalized to the time taken for the page-level Dirtybit scheme. The checkpoint time consists of the time taken to inspect the dirty tracking bitmap, clear bits in the bitmap, and copy the modifications from DRAM to NVM. While inspecting the bitmap, contiguous bits set in the bitmap are coalesced, allowing faster bitmap processing. Thus, the time required to inspect a dirty tracking bitmap depends upon the bitmap area size (based on tracking granularity) and the pattern of bits set in the bitmap (based on the stack access pattern of the application).

Prosper benefits the most in reducing checkpoint size when the stack modification in a checkpoint interval is localized to a range of stack region of size equal to or less than the tracking granularity. For example, compared to the Dirtybit scenario, checkpoint size is reduced by $\sim$200$\times$ for *Sparse* with 8-byte tracking granularity. Due to the significant reduction in checkpoint size for *Sparse*, maximum checkpoint time reduction is observed with all tracking granularity (average 22$\times$ compared to the baseline).

The checkpoint performance of *Prosper* is negatively impacted when the stack modification in a checkpoint interval is to a contiguous range of memory pages in the stack region (as in the case of the *Stream* workload). In such a case, checkpoint sizes for both byte-level and page-level dirty tracking are similar, and there is no reduction in the data copy overhead with *Prosper*. Apart from having no benefits of byte-granularity dirty tracking, the *Stream* benchmark incurred high checkpoint time due to the additional dirty bitmap processing operations at the end of the checkpoint and resulted in a slowdown of $\sim$1.5$\times$ compared to the baseline with 8-byte granularity. As the size of the dirty meta-data (i.e., dirty bitmap area in *Prosper*) decreases with increased tracking granularity, the checkpoint time

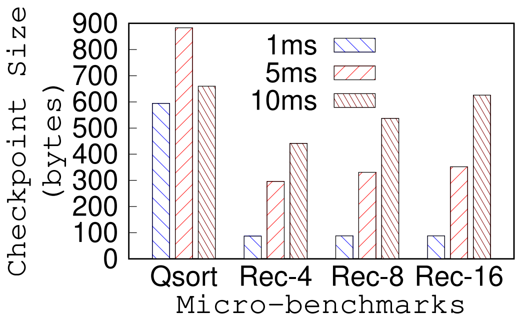


FIGURE 6.11: Influence of checkpoint interval on checkpoint size with dirty stack tracking using Prosper.

overhead for *Stream* reduces with an increase in tracking granularity, showing the lowest for 128-byte granularity.

The *Random* micro-benchmark resulted in the second to lowest reduction in checkpoint size compared to the page granularity checkpoint scenario. Even though 8-byte helped in reducing checkpoint size, the checkpoint time overhead of ∼1.2× in Figure 6.10(b) is due to the overhead in dirty bitmap processing. The random access pattern limits the coalescing opportunities in dirty bitmap processing. For *Random*, tracking at higher granularity helped, showing a ∼1.8× reduction in checkpoint time with 128-byte, which benefited from improvement in dirty bitmap processing. A lower normalized checkpoint time in Figure 6.10(b) is contributed by two components: a reduction in checkpoint size with respect to the page-level Dirtybit scheme and a decrease in bitmap inspection time (decided by the bitmap area size and coalescing opportunities in bitmap). The tracking granularity plays an essential role in balancing the size of checkpoint and bitmap area size, as a higher tracking granularity reduces bitmap area size but may increase checkpoint size.

Prosper performed better with all tracking granularity for *Normal* and *Poisson* micro-benchmarks. The *Quicksort* benchmark sorted elements in a heap to ensure that the stack usage is only due to function calls. Compared to the baseline, *Quicksort* performs better with *Prosper*, showing maximum checkpoint time reduction with 128-byte tracking granularity.

While *Prosper* reduces checkpoint size and checkpoint time for most stack access patterns, the granularity setting should be dynamically adjusted (from the OS layer) to reduce the overhead for workloads like *Stream*.

Prosper with different checkpoint Intervals: The checkpoint interval influences the stack checkpoint size as the stack grows and shrinks multiple times during a checkpoint interval. A large checkpoint interval also allows the coalescing of multiple modifications to the same stack location in a checkpoint interval.

We studied the influence of checkpoint interval on the stack checkpoint size using function call benchmarks *Quicksort* and *Recursive* (Table 6.3) with call depths four (Rec-4), eight (Rec-8), and sixteen (Rec-16). We used eight bytes as the tracking granularity for this experiment. Figure 6.11 shows the checkpoint size (averaged over all checkpoint intervals) for 1 ms, 5 ms, and 10 ms checkpoint intervals. For *Recursive*, checkpoint size increases with an increase in checkpoint interval, denoting that the stack access pattern of *Recursive* does not provide coalescing opportunity, and the stack does not shrink within an increased checkpoint interval. Whereas, the stack access pattern of *Quicksort* provides benefits with an increase in checkpoint interval as the checkpoint size for *Quicksort* reduces with 10 ms interval.

We also observed that even though the checkpoint size is minimum with 1 ms interval for *Recursive*, per byte checkpoint time (i.e., time to checkpoint a byte, measured as checkpoint time to size ratio) is the highest for *Recursive* benchmark with 1 ms interval; 22 ns with 1 ms interval in comparison to 11 ns with 10 ms interval for *Rec-4*. This is because 1 ms interval results in several checkpoints with no stack modifications (i.e., checkpoint size is 0) and incurs only dirty bitmap inspection overhead without any data copying. Therefore, short checkpoint intervals may be counterproductive because of unnecessary bitmap inspections.

The benefit of a longer checkpoint interval depends on the stack access pattern, and having a shorter interval may be counterproductive, resulting in high checkpoint overheads.

Context switch overhead of Prosper: We use the Setup-I (Table 6.2) to study the context switch overheads, i.e., GemOS with *Prosper* modifications executing on gem5 with *Prosper* hardware. In GemOS, while handling the timer interrupt, the OS instructs the *Prosper* hardware to flush tracked information in the lookup table to memory if the outgoing process is persistent. The scheduling logic in GemOS continues with other activities related to context switch, such as selecting and preparing the new context. Before scheduling the incoming context, OS ensures quiescence of the dirty tracker state for the outgoing process by checking a counter maintained in the *Prosper* hardware (§ 6.2.3). Depending on the persistence requirement of the incoming process, OS loads the required *Prosper* parameters of the incoming context (by setting the MSRs) to notify the *Prosper* hardware.

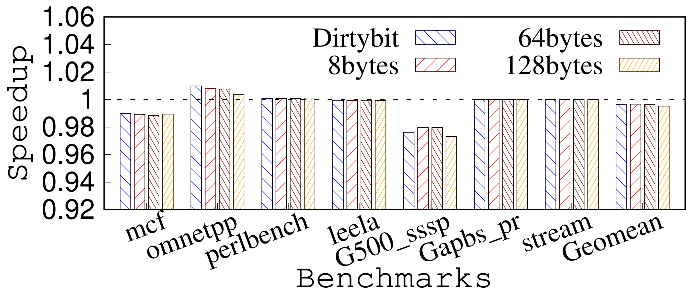


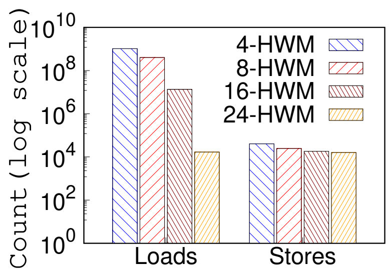
FIGURE 6.12: Tracking overhead for different SPEC workloads. Y-axis shows an application's performance under dirty tracking with respect to no dirty tracking.

We used a multi-threaded micro-benchmark with two threads to evaluate the context switch overhead introduced by *Prosper*. Each thread performs a fixed number of random writes to its stack, and the main thread waits for completion. The stack area of each thread other than the main thread is persistently maintained using *Prosper*. To measure the overhead introduced by *Prosper* during a context switch, we capture the time taken to flush the outgoing context's dirty tracker state and set the incoming context's parameters. The additional overhead introduced by the save-restore of the tracker state was ∼870 cycles on average.

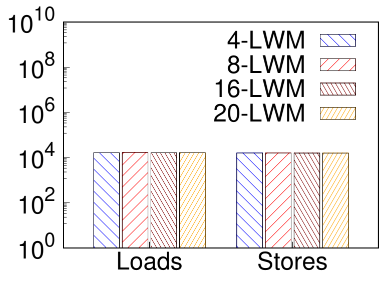
Dirty Tracking Overhead of Prosper: To analyze the overheads introduced by *Prosper* due to hardware tracking of stack modifications, we performed experiments with selected benchmarks 605.mcf_s, 620.omnetpp_s, 600.perlbench_s, 641.leela_s from the SPEC CPU 2017, SSSP from Graph500, PR from GAPBS and micro-benchmark Stream (Table 6.3).

For this experiment, we use the gem5 configuration for Setup-II (Table 6.2) and the modified Linux kernel (refer to § 6.3). Each benchmark application was executed initially for one minute without dirty tracking (for warm-up), and then the kernel thread performed 6000 checkpoints, each at a 10 ms interval. We used three tracking granularity—8bytes, 64bytes, and 128bytes for *Prosper*. At the end of an interval, an inspection of the bitmap area corresponding to the active stack region is performed for *Prosper*, an inspection of dirty-bit in page table entries for the stack address range is performed for Dirtybit.

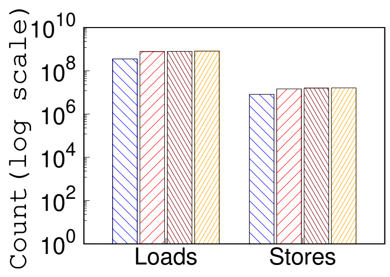
Figure 6.12 shows the application's performance with dirty tracking for a fixed interval of 6000 checkpoints, calculated with respect to its performance with no dirty tracking. To isolate the performance of the benchmark application from kernel interference, we captured the number of instructions and number of cycles spent only in the user space, and the speedup in Figure 6.12 is based on IPC in the user space. *Prosper* resulted in



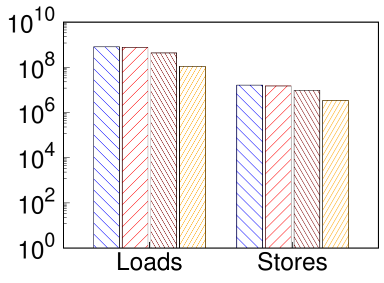
(a) G500_sssp, HWM.



(b) G500_sssp, LWM.



(c) mcf, HWM



(d) mcf, LWM

FIGURE 6.13: Sensitivity of HWM and LWM for bitmap load and store operations. For HWM study LWM is fixed at 4 and for LWM study HWM is fixed at 24.

minimum overhead (on an average less than 1%, a maximum of ∼3% for G500_sssp) across all applications for all tracking granularity. Note that, even the IPC values for user space can be impacted by the execution of OS background services (e.g., due to cache pollution), and therefore, the results should be interpreted considering the inherent variations [169, 170].

Dirty Tracker Sensitivity to HWM and LWM Parameters: The HWM and LWM parameters influence the state of the lookup table and impact the amount of memory load and store operations to maintain the dirty bitmap area (§ 6.2.2).

We analyzed the influence of HWM and LWM values on the number of bitmap loads and stores generated with mcf from SPEC CPU2017 and SSSP from Graph500 using the gem5 configuration for Setup-II (Table 6.2) and the modified Linux kernel (refer to § 6.3). Figure 6.13 shows the number of bitmap loads and stores issued by *Prosper* with varying HWM and LWM thresholds. We have fixed the LWM threshold value to 4 while varying the HWM (Figure 6.13(a) and Figure 6.13(c)) and the HWM value to 24 while varying the LWM (Figure 6.13(b) and Figure 6.13(d)) for this study. For SSSP, the number of bitmap loads and stores decreases with an increase in HWM, indicating spatial locality in its stack access. At the same time, LWM variation marginally influences the loads and stores, indicating that creating more vacancies in the lookup table has no added benefits. On the other hand, for mcf, the trend is reversed where the number of bitmap loads and stores increases with increased HWM, indicating the lack of spatial locality. Further, we observe a decrease in number of load and store operations with an increase in LWM, implying more evictions can be helpful. The influence of HWM and LWM on bitmap loads and stores depends on the stack access pattern. While we have used a fixed setting (LWM = 8, HWM = 24) in the previous experiments, a dynamic scheme based on the access pattern is left as a future direction.

Energy and area overhead: We obtain the dynamic energy consumption for read and write operations on the lookup table (§ 6.2.2) configured with two read ports and one write port, using CACTI-P [171] for 7nm FINFET technology. The total dynamic read energy per access is 0.0000773194 nJ, the write energy per access is 0.000128375 nJ, and the leakage power of a bank is 0.01067596 mW. The lookup table with 16 entries occupies a cache area of 0.000704786 mm^2.

6.5 Summary

Process persistence requires persisting its memory state consisting of mutable stack and heap segments. In this chapter, we present *Prosper*, a sub-page byte granularity checkpoint-based persistence mechanism for process stack that handles unique stack properties, providing an average of 2.1× (maximum of 3.6×) reduction in stack persistence overhead with respect to state-of-the-art memory persistence mechanism (SSP). We showed that *Propser* complements well with existing memory persistence mechanisms for persisting the entire memory area of a process for process persistence; *Prosper* with SSP provided an average 2× (maximum of 2.6×) reduction in memory persistence overhead for persisting the entire memory area of a process. Our evaluation using SPEC CPU 2017, SSSP from Graph500, and PR from GAPBS showed that *Prosper* causes negligible tracking overhead compared to baseline (on an average of less than 1%). *Prosper* addresses the unique stack properties for achieving stack persistence efficiently that can complement different varieties of existing application checkpoint mechanisms in hybrid memory systems.

Source code of *Prosper* is available at https://github.com/arunkp1986/Prosper.git and *Prosper* earned all 3 badges in artifact evaluation, *Code Available*, *Code Reviewed*, and *Code Reproducible*.

Chapter 7

Conclusions and Future Directions

This thesis contributes to the domain of process persistence in hybrid memory systems with NVM and DRAM. This chapter summarizes the four contributions of this thesis and draws conclusions (§ 7.1). Next, we present two future directions for which this thesis paves way in the domain of process persistence (§ 7.2).

7.1 Summary and Conclusions

Process persistence requires saving the state of a process in a persistent device such as NVM. We first presented the challenges and mechanisms for achieving memory persistence in a system with NVM for process persistence. Memory persistence mechanisms ensure that writes to NVM reach the persistent domain in the order the application developer expects. Persistent process system designers can implement memory persistence mechanisms in software and/or hardware to guarantee a consistent NVM memory state. These mechanisms utilize cache line flush and memory fence primitives in the underlying architecture to enforce the required order of writes and persistence guarantee. We studied the performance overhead of these architectural primitives as they form the basic building blocks of existing memory persistence mechanisms. This empirical study included flush and fence variants in two popular architectures, Intel x86-64 and Arm64. We examined the influence of working set size and memory access characteristics of applications on the performance of these data consistency primitives. We also studied the impact of data consistency primitives on the performance of advanced memory persistence techniques based on undo/redo logging. The study shows that the performance overhead depends upon the nature of the workload and the proportion of the read-to-write ratio in memory access. The study also reaffirms that serialization operations such as `sfence` are the major contributing factors in performance overhead. We also observed that it is not always advantageous to stick to a particular architecture primitive (for example, `clwb`) for all memory persistence mechanisms (for example, redo and undo logging), and the selection of architecture primitives should be decided on a case-to-case basis depending on the workload characteristics.

Using NVM as the only system memory for process persistence degrades application performance because of the higher read/write access latency of NVM compared to volatile DRAM. So, a better approach is to use a hybrid memory system with NVM and DRAM that compensates for the high access latency of NVM by combining the access latency benefits of DRAM, providing overall good system performance. We created a hybrid memory simulation framework, *Kindle*, to study process persistence. *Kindle* enables analyzing performance trade-offs in persisting individual process states such as execution context, CPU register states, virtual to physical memory translation table, and memory state comprising stack and heap areas across different design choices for achieving process persistence. *Kindle* supports process persistence and provides an end-to-end framework for quick prototyping of mechanisms and policies related to process persistence. Using *Kindle*, we studied the overhead in maintaining the execution context of a process consistently under two schemes for page table consistency. *Kindle* helps realize research ideas crossing hardware-software layers in hybrid memory systems, as we showcased a prototype implementation of two state-of-the-art hybrid memory schemes using *Kindle*.

Memory is a significant component in the state of a persistent process, and the performance of the persistence scheme used for the memory state of a process determines the overall performance of process persistence. The memory layout of a process consists of heap and stack areas, each exhibiting different usage characteristics. The stack is crucial in memory layout as it holds important information on program behavior and running state. As existing state-of-the-art memory persistence schemes for NVM are majorly designed for heap area and do not consider memory usage characteristics of memory area in persistence scheme, we created a stack tracing framework, *SniP*, to analyze the program stack of multi-threaded applications to understand stack usage characteristics such as grow-and-shrink pattern of usage, activation record of usage and indirect usage and its influence on memory persistence. *SniP* efficiently handles challenges in identifying stack areas of threads for tracing the stack of multi-threaded applications. Stack analysis using *SniP* reveals that persisting the program stack requires special consideration due to its unique usage characteristics. The state-of-the-art memory persistence mechanisms for NVM require non-trivial adaptation to achieve stack persistence efficiently.

To address this gap in state-of-the-art memory persistence mechanisms for stack persistence usage, we created *Prosper*, a hardware-software co-designed checkpoint approach for stack persistence. *Prosper* tracks changes at sub-page byte granularity and handles stack-specific usage patterns such as grow and shrink patterns, indirect usage, and write characteristics. *Prosper* provides an average of 2.1× (maximum of 3.6×) reduction in stack persistence overhead with respect to the state-of-the-art memory persistence scheme (SSP). We also show that using *Prosper* with existing memory persistence schemes for the entire memory area is beneficial. *Prosper* with SSP provided an average of 2× (maximum of 2.6×) reduction in memory persistence overhead for persisting the entire memory area. *Prosper* had negligible tracking overhead compared to baseline (on average, less than 1%).

This thesis facilitates full process persistence in a hybrid memory system with NVM by combining *Kindle*, which provides persistence for execution context and virtual address translation state of a process with memory state persistence mechanisms, *Prosper* for stack combined with a state-of-art scheme such as SSP for heap, to persist entire memory area of a process.

7.2 Future Directions

We discuss two possible directions to extend the different problems addressed in this thesis. First, exploring the influence of NVM device characteristics on persistent barrier performance in multiple ISAs. In the second direction, study the overhead of cache line flush and memory fence architectural primitives in the context of persistent devices attached to a high-bandwidth PCIe interface with Compute Express Link (CXL) capability.

7.2.1 Empirical Analysis on the Influence of NVM Device on Persistence

The read/write access latency of persistent memory varies based on the underlying technology, such as PCM, STTRAM, MRAM, and FeRAM. The device access latency then impacts the expected performance overhead of primitive and advanced memory persistence schemes, as the time taken for cache line writebacks varies from one device to another, influencing the latency to flush cache lines to ensure NVM memory state consistency. The aim is to study the performance of primitive and advanced memory persistence schemes for different persistent memory technologies, such as PCM, STTRAM, MRAM, and FeRAM, to understand the influence of device characteristics on memory persistence schemes. The high-level questions that can be part of this analysis are,

1. What is the extent to which the underlying persistent memory device technology influences the latency of cache line writebacks?
2. Understand the performance of cache-line flush instructions in popular ISAs (Intel x86-64, Arm64, RISC-V, PowerPC).
3. What is the deviation in cache-line flush performance from one memory technology to another?
4. How much does cache-line flush performance based on devices impact the performance of software-based advanced memory persistence techniques such as logging, shadow-paging, and checkpointing?
5. How do the application characteristics, such as working set size and memory access patterns, influence observed performance overhead?

This study gives insights into the benefit of opting for one memory technology over another for various memory persistence schemes, and for which memory access patterns and working set sizes.

7.2.2 Memory Persistence using CXL attached persistent memory

Compute Express Link (CXL) [172, 173] defines a family of interconnect protocols between CPUs and devices, allowing devices to cache host memory and mapping device memory to system cacheable memory space [174]. CXL allows persistent memory devices to be attached to the CPU through PCIe interconnect, enabling both memory expansion and persistence. We can use byte-addressable storage class memory (SCM) on PCIe with CXL for process persistence.

In the second research direction, the aim is to study the influence of emerging interconnect technologies such as CXL on the performance of memory persistence mechanisms, as the performance of cache line flush and memory fence architectural primitives varies while targeting persistent devices attached to high bandwidth PCIe interface with Compute Express Link (CXL) capability. The high-level questions are

1. How much do interconnect characteristics influence existing memory persistence schemes, as existing mechanisms are mainly designed for persistent memory attached to the memory bus?

2. Which CXL interconnect characteristics influence memory persistence schemes for CXL-attached persistent memory/storage?
3. What is the performance of existing primitive and advanced memory persistence, and adapt/design memory persistence for memory attached using CXL

The goal is to propose a memory persistence scheme for CXL-attached persistent memory because naively using a memory persistence scheme such as undo/redo logging may create fabric congestion and affect overall system throughput. A CXL-attached persistent device also enables process persistence for heterogeneous shared memory applications running on CPU and accelerator, application scheduling based on IO characteristics, and composable system design.

List of Publications

1. Arun K.P., and Debadatta Mishra. ”Kindle: A Comprehensive Framework for Exploring OS-Architecture Interplay in Hybrid Memory Systems.” In 2024 IEEE International Symposium on Workload Characterization (IISWC), pp. 262-272. IEEE, 2024. Kindle received all three badges, **Code Available**, **Code Reviewed** and **Code Reproducible** in Artifact Evaluation.

2. Arun K.P., Debadatta Mishra, and Biswabandan Panda. ”Prosper: Program Stack Persistence in Hybrid Memory Systems.” In 2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp. 1168-1183. IEEE, 2024. Prosper received all three badges, **Code Available**, **Code Reviewed** and **Code Reproducible** in Artifact Evaluation.

3. Arun K.P., Saurabh Kumar, Debadatta Mishra, and Biswabandan Panda. ”Snip: an efficient stack tracing framework for multi-threaded programs.” In Proceedings of the 19th International Conference on Mining Software Repositories (MSR), pp. 408-412. 2022.

4. Arun K.P., Debadatta Mishra, and Biswabandan Panda. ”Empirical analysis of architectural primitives for nvram consistency.” In 2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC), pp. 172-181. IEEE, 2021.

Others

1. Arun K.P., Rohit Singh, and Debadatta Mishra. ”LENS: Experiencing Multi-level Page Tables at Close Quarters.” In Proceedings of the ACM Conference on Global Computing Education Vol 1, pp. 105-111. 2023.

2. Singh, Rohit, Arun K.P., and Debadatta Mishra. ”LDT: Lightweight dirty tracking of memory pages for x86 systems.” In 2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC), pp. 85-94. IEEE, 2022.

Bibliography

[1] E. Tsalapatis, R. Hancock, T. Barnes, and A. J. Mashtizadeh, "The aurora single level store operating system," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 788–803, 2021.

[2] Y. Jiao, S. Bertron, S. Patel, L. Zeller, R. Bennett, N. Mukherjee, M. A. Bender, M. Condict, A. Conway, M. Farach-Colton, *et al.*, "Betrfs: a compleat file system for commodity ssds," in *Proceedings of the Seventeenth European Conference on Computer Systems*, pp. 610–627, 2022.

[3] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, pp. 1–32, 2013.

[4] S. Gugnani, A. Kashyap, and X. Lu, "Understanding the idiosyncrasies of real persistent memory," *Proceedings of the VLDB Endowment*, vol. 14, no. 4, pp. 626–639, 2020.

[5] "The intel® optane™ persistent memory website." `https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html`.

[6] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating systems: Three easy pieces.* Arpaci-Dusseau Books LLC, 2018.

[7] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Dhtm: durable hardware transactional memory," in *2018 ACM/IEEE 45th ISCA*, pp. 452–465, IEEE, 2018.

[8] R. M. Krishnan, J. Kim, A. Mathew, X. Fu, A. Demeri, C. Min, and S. Kannan, "Durable transactional memory can scale with timestone," in *Proceedings of ASPLOS*, pp. 335–349, 2020.

[9] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *2017 IEEE International Symposium on HPCA*, pp. 361–372, IEEE, 2017.

[10] "The persistent memory development kit website." `https://pmem.io/pmdk/`, 2020.

[11] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of ISCA*, pp. 2–13, 2009.

[12] P. Fatourou, N. D. Kallimanis, and E. Kosmas, "The performance power of software combining in persistence," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 337–352, 2022.

[13] H. Attiya, O. Ben-Baruch, and D. Hendler, "Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory," in *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pp. 7–16, 2018.

[14] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank, "A persistent lock-free queue for non-volatile memory," *ACM SIGPLAN Notices*, vol. 53, no. 1, pp. 28–40, 2018.

[15] N. Li and W. Golab, "Brief announcement: Detectable sequential specifications for recoverable shared objects," in *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pp. 557–560, 2021.

[16] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ACM SIGARCH Computer Architecture News*, vol. 42, pp. 265–276, IEEE Press, 2014.

[17] K. Arun, D. Mishra, and B. Panda, "Empirical analysis of architectural primitives for nvram consistency," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp. 172–181, IEEE, 2021.

[18] A. Baldassin, J. Barreto, D. Castro, and P. Romano, "Persistent memory: A survey of programming support and implementations," *ACM Computing Surveys (CSUR)*, vol. 54, no. 7, pp. 1–37, 2021.

[19] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 433–452, 2014.

[20] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 178–190, 2017.

[21] N. Cohen, D. T. Aksun, H. Avni, and J. R. Larus, "Fine-grain checkpointing with in-cache-line logging," in *Proceedings of ASPLOS*, pp. 441–454, ACM, 2019.

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

[23] J. Gu, Q. Yu, X. Wang, Z. Wang, B. Zang, H. Guan, and H. Chen, "Pisces: A scalable and efficient persistent transactional memory," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 913–928, 2019.

[24] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 520–532, IEEE, 2018.

[25] Y. Ni, J. Zhao, H. Litz, D. Bittman, and E. L. Miller, "Ssp: Eliminating redundant writes in failure-atomic nvrams via shadow sub-paging," in *52nd Annual IEEE/ACM MICRO*, pp. 836–848, 2019.

[26] A. Correia, P. Felber, and P. Ramalhete, "Romulus: Efficient algorithms for persistent transactional memory," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pp. 271–282, 2018.

[27] H. Liu, Y. Chen, X. Liao, H. Jin, B. He, L. Zheng, and R. Guo, "Hardware/software cooperative caching for hybrid dram/nvm memory architectures," in *Proceedings of the International Conference on Supercomputing*, pp. 1–10, 2017.

[28] L. Liu, S. Yang, L. Peng, and X. Li, "Hierarchical hybrid memory management in os for tiered memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2223–2236, 2019.

[29] M. Cai, C. C. Coats, and J. Huang, "Hoop: Efficient hardware-assisted out-of-place update for non-volatile memory," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 584–596, IEEE, 2020.

[30] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 105–118, 2011.

[31] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 427–442, 2016.

[32] E. R. Giles, K. Doshi, and P. Varman, "Softwrap: A lightweight framework for transactional support of storage class memory," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–14, IEEE, 2015.

[33] L. Zhang and S. Swanson, "Pangolin: A {Fault-Tolerant} persistent memory programming library," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 897–912, 2019.

[34] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "Nvthreads: Practical persistence for multi-threaded applications," in *Proceedings of EuroSys*, pp. 468–482, 2017.

[35] W. R. Dieter and J. E. Lumpp Jr, "User-level checkpointing for linuxthreads programs.," in *USENIX Annual Technical Conference, FREENIX Track*, pp. 81–92, 2001.

[36] G. J. Janakiraman, J. R. Santos, D. Subhraveti, and Y. Turner, "Cruz: Application-transparent distributed checkpoint-restart on standard operating systems," in *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pp. 260–269, IEEE, 2005.

[37] O. Laadan and J. Nieh, "Transparent checkpoint-restart of multiple processes on commodity operating systems.," in *USENIX Annual Technical Conference*, pp. 323–336, 2007.

[38] "Persistent memory programming website." `https://pmem.io/book/`, 2020.

[39] R. Xiao, D. Feng, Y. Hu, F. Wang, X. Wei, X. Zou, and M. Lei, "Write-optimized and consistent skiplists for non-volatile memory," *IEEE Access*, 2021.

[40] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning lsms for nonvolatile memory with novelsm," in *USENIX Annual Technical Conference*, pp. 993–1005, 2018.

[41] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *USENIX OSDI*, pp. 461–476, 2018.

[42] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner, "nvm malloc: Memory allocation for nvram.," *ADMS@ VLDB*, vol. 15, pp. 61–72, 2015.

[43] S. Yu, N. Xiao, M. Deng, F. Liu, and W. Chen, "Redesign the memory allocator for non-volatile main memory," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, pp. 1–26, 2017.

[44] S. Kannan, A. Gavrilovska, and K. Schwan, "pvm: Persistent virtual memory for efficient capacity scaling and object storage," in *Proceedings of the EuroSys*, pp. 1–16, 2016.

[45] M. Cai and H. Huang, "A survey of operating system support for persistent memory," *Frontiers of Computer Science*, vol. 15, no. 4, pp. 1–20, 2021.

[46] I. El Hajj, A. Merritt, G. Zellweger, D. Milojicic, R. Achermann, P. Faraboschi, W.-m. Hwu, T. Roscoe, and K. Schwan, "Spacejmp: programming with multiple virtual address spaces," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 353–368, 2016.

[47] A. Rudoff, "Persistent memory programming," *Login: The Usenix Magazine*, vol. 42, pp. 34–40, 2017.

[48] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, vol. 2, p. 5, 2011.

[49] "Arm cortex-a series programmer's guide for armv8-a website." `https://developer.arm.com/documentation/den0024/a/preface`, 2020.

[50] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[51] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[52] D. Mishra, "Gemos: Bridging the gap between architecture and operating system in computer system education," in *Proceedings of the Workshop on Computer Architecture Education*, pp. 1–8, 2019.

[53] M. Weiland, H. Brunst, T. Quintino, N. Johnson, O. Iffrig, S. Smart, C. Herold, A. Bonanni, A. Jackson, and M. Parsons, "An early evaluation of intel's optane dc persistent memory module and its impact on high-performance scientific applications," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pp. 1–19, 2019.

[54] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[55] C. Linn, S. Debray, G. Andrews, and B. Schwarz, "Stack analysis of x86 executables," *Manuscript. April*, 2004.

[56] R. Stallman, R. Pesch, S. Shebs, *et al.*, "Debugging with gdb," *Free Software Foundation*, vol. 675, 1988.

[57] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[58] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[59] S. A. McKee, "Reflections on the memory wall," in *Proceedings of the 1st conference on Computing frontiers*, p. 162, 2004.

[60] K. Wu, I. Peng, J. Ren, and D. Li, "Ribbon: High performance cache line flushing for persistent memory," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pp. 427–439, 2020.

[61] M. Alshboul, J. Tuck, and Y. Solihin, "Lazy persistency: A high-performing and write-efficient software persistency technique," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 439–451, IEEE, 2018.

[62] T. Ma, M. Zhang, K. Chen, Z. Song, Y. Wu, and X. Qian, "Asymnvm: An efficient framework for implementing persistent data structures on asymmetric nvm architecture," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 757–773, 2020.

[63] Z. Dang, S. He, X. Zhang, P. Hong, Z. Li, X. Chen, H. Song, X.-H. Sun, and G. Chen, "Pmalloc: A holistic approach to improving persistent memory allocation," *ACM Transactions on Computer Systems*, 2024.

[64] A. Chatzistergiou, M. Cintra, and S. D. Viglas, "Rewind: Recovery write-ahead system for in-memory non-volatile data-structures," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 5, pp. 497–508, 2015.

[65] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, “High-performance transactions for persistent memories,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 399–411, 2016.

[66] S. Haria, M. D. Hill, and M. M. Swift, “Mod: Minimally ordered durable datastructures for persistent memory,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 775–788, 2020.

[67] J. Guerra, L. Marmol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei, “Software persistent memory,” in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pp. 319–331, 2012.

[68] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell, *et al.*, “Consistent and durable data structures for non-volatile byte-addressable memory.,” in *FAST*, vol. 11, pp. 61–75, 2011.

[69] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, “Dudetm: Building durable transactions with decoupling for persistent memory,” *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 329–343, 2017.

[70] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, “Optimizing checkpoints using nvm as virtual memory,” in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 29–40, IEEE, 2013.

[71] F. Ren, K. Chen, and Y. Wu, “libcrpm: Improving the checkpoint performance of nvm,” in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pp. 811–816, 2022.

[72] A. Khorguani, T. Ropars, and N. De Palma, “Respct: fast checkpointing in non-volatile memory for multi-threaded applications,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, pp. 525–540, 2022.

[73] E. Tsalapatis, R. Hancock, R. Hossain, and A. J. Mashtizadeh, “Memsnap μcheckpoints: A data single level store for fearless persistence,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 622–638, 2024.

[74] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, “Hybrid checkpointing using emerging nonvolatile memories for future exascale systems,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 2, pp. 1–29, 2011.

[75] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin, "Efficient checkpointing of loop-based codes for non-volatile main memory," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 318–329, IEEE, 2017.

[76] D. Castro, P. Romano, and J. Barreto, "Hardware transactional memory meets memory persistency," *Journal of Parallel and Distributed Computing*, vol. 130, pp. 63–79, 2019.

[77] C. Ye, Y. Xu, X. Shen, Y. Sha, X. Liao, H. Jin, and Y. Solihin, "Specpmt: Speculative logging for resolving crash consistency overhead of persistent memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 762–777, 2023.

[78] M. Zhang and Y. Hua, "Silo: Speculative hardware logging for atomic durability in persistent memory," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 651–663, IEEE, 2023.

[79] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pp. 216–223, IEEE, 2014.

[80] A. Abulila, I. E. Hajj, M. Jung, and N. S. Kim, "Asap: architecture support for asynchronous persistence," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pp. 306–319, 2022.

[81] J. Jeong, J. Zeng, and C. Jung, "Capri: Compiler and architecture support for whole-system persistence," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, pp. 71–83, 2022.

[82] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 672–685, IEEE, 2015.

[83] S. Wu, F. Zhou, X. Gao, H. Jin, and J. Ren, "Dual-page checkpointing: An architectural approach to efficient data persistence for in-memory applications," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 4, pp. 1–27, 2019.

[84] Y. Xu, W. Xu, K. Keeton, and D. E. Culler, "Softpm: Software persistent memory," in *13th Non-Volatile Memories Workshop (NVMW)*, 2022.

[85] T. David, A. Dragojevic, R. Guerraoui, and I. Zablotchi, "{Log-Free} concurrent data structures," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 373–386, 2018.

[86] B. Lu, X. Hao, T. Wang, and E. Lo, "Dash: scalable hashing on persistent memory," *Proc. VLDB Endow.*, vol. 13, p. 1147–1161, apr 2020.

[87] A. Memaripour, J. Izraelevitz, and S. Swanson, "Pronto: Easy and fast persistence for volatile data structures," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 789–806, 2020.

[88] D. Schwalb, M. Dreseler, M. Uflacker, and H. Plattner, "Nvc-hashmap: A persistent and concurrent hashmap for non-volatile memories," in *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*, pp. 1–8, 2015.

[89] T. Wang, J. Levandoski, and P.-A. Larson, "Easy lock-free indexing in non-volatile memory," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 461–472, IEEE, 2018.

[90] Y. Zuriel, M. Friedman, G. Sheffi, N. Cohen, and E. Petrank, "Efficient lock-free durable sets," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–26, 2019.

[91] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 133–146, 2009.

[92] Z. Weiss, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "{DenseFS}: a {Cache-Compact} filesystem," in *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[93] G. O. Puglia, A. F. Zorzo, C. A. De Rose, T. D. Perez, and D. Milojicic, "Non-volatile memory file systems: A survey," *IEEE Access*, vol. 7, pp. 25836–25871, 2019.

[94] J. Xu and S. Swanson, "{NOVA}: A log-structured file system for hybrid volatile/non-volatile main memories," in *Conference on File and Storage Technologies (FAST)*, pp. 323–338, 2016.

[95] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, pp. 1–15, 2014.

[96] H. Shu, H. Chen, H. Liu, Y. Lu, Q. Hu, and J. Shu, "Empirical study of transactional management for persistent memory," in *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pp. 61–66, IEEE, 2018.

[97] A. Dearle, R. Di Bona, J. Farrow, F. Henskens, A. Lindström, J. Rosenberg, F. Vaughan, *et al.*, "Grasshopper: An orthogonally persistent operating system," *Computing Systems*, vol. 7, no. 3, pp. 289–312, 1994.

[98] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "Splitfs: Reducing software overhead in file systems for persistent memory," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 494–508, 2019.

[99] DAX, "https://docs.kernel.org/filesystems/dax.html."

[100] Y. Ni, J. Zhao, D. Bittman, and E. Miller, "Reducing {NVM} writes with optimized shadow paging," in *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[101] H. Du, Q. Li, R. Pan, T.-W. Kuo, and C. J. Xue, "Multi-granularity shadow paging with nvm write optimization for crash-consistent memory-mapped i/o," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 108–121, IEEE, 2023.

[102] H. Wan, Y. Lu, Y. Xu, and J. Shu, "Empirical study of redo and undo logging in persistent memory," in *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pp. 1–6, IEEE, 2016.

[103] J. Zhou, A. Awad, and J. Wang, "Lelantus: Fine-granularity copy-on-write operations for secure non-volatile memories," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 597–609, IEEE, 2020.

[104] R. Singh, K. Arun, and D. Mishra, "Ldt: Lightweight dirty tracking of memory pages for x86 systems," in *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp. 85–94, IEEE, 2022.

[105] T. Hwang and Y. Won, "Copy-on-write with adaptive differential logging for persistent memory," *IEICE TRANSACTIONS on Information and Systems*, vol. 102, no. 12, pp. 2451–2460, 2019.

[106] A. Singh and S. R. Sarangi, "Jass: A tunable checkpointing system for nvm-based systems," in *2023 IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp. 164–173, IEEE, 2023.

[107] W.-z. Zhang, K. Lu, M. Luján, X.-p. Wang, and X. Zhou, “Fine-grained checkpoint based on non-volatile memory,” *Frontiers of Information Technology & Electronic Engineering*, vol. 18, no. 2, pp. 220–234, 2017.

[108] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, “Rethinking software runtimes for disaggregated memory,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 79–92, 2021.

[109] D. Waddington, M. Hershcovitch, S. Sundararaman, and C. Dickey, “A case for using cache line deltas for high frequency vm snapshotting,” in *Proceedings of the 13th Symposium on Cloud Computing*, pp. 526–539, 2022.

[110] Y. Ozawa and T. Shinagawa, “Exploiting sub-page write protection for vm live migration,” in *International Conference on Cloud Computing (CLOUD)*, pp. 484–490, IEEE, 2021.

[111] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska, “Sharing and protection in a single-address-space operating system,” *ACM Transactions on Computer Systems (TOCS)*, vol. 12, no. 4, pp. 271–307, 1994.

[112] X. Li, K. Lu, X. Wang, and X. Zhou, “Nv-process: a fault-tolerance process model based on non-volatile memory,” in *Proceedings of the Asia-Pacific Workshop on Systems*, pp. 1–6, 2012.

[113] D. Bittman, P. Alvaro, P. Mehra, D. D. Long, and E. L. Miller, “Twizzler: a data-centric {OS} for non-volatile memory,” in *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pp. 65–80, 2020.

[114] D. Bittman, P. Alvaro, P. Mehra, D. D. E. Long, and E. L. Miller, “Twizzler: a data-centric OS for non-volatile memory,” in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 65–80, USENIX Association, 2020.

[115] T. Nguyen and D. Wentzlaff, “Picl: A software-transparent, persistent cache log for nonvolatile main memory,” in *51st Annual IEEE/ACM MICRO*, pp. 507–519, IEEE, 2018.

[116] R. Pagh and F. F. Rodler, “Cuckoo hashing,” *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[117] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[118] M. Poremba, T. Zhang, and Y. Xie, “Nvmain 2.0: A user-friendly memory simulator to model (non-) volatile memory systems,” *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 140–143, 2015.

[119] S. Song, A. Das, O. Mutlu, and N. Kandasamy, “Improving phase change memory performance with data content aware access,” *arXiv preprint arXiv:2005.04753*, 2020.

[120] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, “Implications of cpu caching on byte-addressable non-volatile memory programming,” *Hewlett-Packard, Tech. Rep. HPL-2012-236*, 2012.

[121] J. Bucek, K.-D. Lange, and J. v. Kistowski, “Spec cpu2017: Next-generation compute benchmark,” in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE ’18, (New York, NY, USA), p. 41–42, Association for Computing Machinery, 2018.

[122] B. Wang, J. Tang, R. Zhang, W. Ding, S. Liu, and D. Qi, “Energy-efficient data caching framework for spark in hybrid dram/nvm memory architectures,” in *Proceedings of International Conference on High Performance Computing and Communications*, pp. 305–312, IEEE, 2019.

[123] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen, “Exploiting program semantics to place data in hybrid memory,” in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 163–173, IEEE, 2015.

[124] H. Liu, R. Liu, X. Liao, H. Jin, B. He, and Y. Zhang, “Object-level memory allocation and migration in hybrid memory systems,” *IEEE Transactions on Computers*, vol. 69, no. 9, pp. 1401–1413, 2020.

[125] X. Wang, H. Liu, X. Liao, J. Chen, H. Jin, Y. Zhang, L. Zheng, B. He, and S. Jiang, “Supporting superpages and lightweight page migration in hybrid memory systems,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 2, pp. 1–26, 2019.

[126] Y. Xie, “Modeling, architecture, and applications for emerging memory technologies,” *IEEE design & test of computers*, vol. 28, no. 1, pp. 44–51, 2011.

[127] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller, “Energy management for commercial servers,” *Computer*, vol. 36, no. 12, pp. 39–48, 2003.

[128] H. Park, S. Yoo, and S. Lee, "Power management of hybrid dram/pram-based main memory," in *Proceedings of the 48th Design Automation Conference*, pp. 59–64, 2011.

[129] B. Peng, Y. Dong, J. Yao, F. Wu, and H. Guan, "Flexhm: A practical system for heterogeneous memory with flexible and efficient performance optimizations," *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 1, pp. 1–26, 2022.

[130] T. Hwang, J. Jung, and Y. Won, "Heapo: Heap-based persistent object store," *ACM Transactions on Storage (TOS)*, vol. 11, no. 1, pp. 1–21, 2014.

[131] Z. Wu, K. Lu, A. Nisbet, W. Zhang, and M. Luján, "Pmthreads: Persistent memory threads harnessing versioned shadow copies," in *Proceedings of PLDI*, pp. 623–637, 2020.

[132] A. KP, D. Mishra, and B. Panda, "Prosper: Program stack persistence in hybrid memory systems," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1168–1183, 2024.

[133] "Google scholar." https://scholar.google.com/. Accessed: 2024-05-30.

[134] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[135] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, 2010.

[136] A. KP, S. Kumar, D. Mishra, and B. Panda, "Snip: an efficient stack tracing framework for multi-threaded programs," in *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 408–412, 2022.

[137] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[138] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," *ACM SIGARCH Computer architecture news*, vol. 41, no. 3, pp. 475–486, 2013.

[139] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, "The championship simulator: Architectural simulation for education and competition," *arXiv preprint arXiv:2210.14324*, 2022.

[140] D. C. Arnold, D. H. Ahn, B. R. De Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Stack trace analysis for large scale debugging," in *2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 1–10, IEEE, 2007.

[141] Y. Wang, M. Wen, R. Wu, Z. Liu, S. H. Tan, Z. Zhu, H. Yu, and S.-C. Cheung, "Could i have a stack trace to examine the dependency conflict issue?," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 572–583, IEEE, 2019.

[142] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *2003 Symposium on Security and Privacy, 2003.*, pp. 62–75, IEEE, 2003.

[143] Tkzzw, "Tkrzw: a set of implementations of dbm." `https://dbmx.net/tkrzw/`, 2021.

[144] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *PVLDB*, vol. 7, no. 4, pp. 277–288, 2013.

[145] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey of machine learning accelerators," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–12, IEEE, 2020.

[146] S. Mittal, P. Rajput, and S. Subramoney, "A survey of deep learning on cpus: opportunities and co-optimizations," *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[147] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and modeling non-volatile memory systems," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 496–508, IEEE, 2020.

[148] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pp. 169–182, 2020.

[149] T. Mason, T. D. Doudali, M. Seltzer, and A. Gavrilovska, "Unexpected performance of intel® optane™ dc persistent memory," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 55–58, 2020.

[150] S. Kumar, D. Mishra, B. Panda, and S. K. Shukla, "Deepdetect: A practical on-device android malware detector," in *21st IEEE International Conference on Software Quality, Reliability and Security*, IEEE (in press), 2021.

[151] G. George, J. Kotey, M. Ripley, K. Z. Sultana, and Z. Codabux, "A preliminary study on common programming mistakes that lead to buffer overflow vulnerability," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 1375–1380, IEEE, 2021.

[152] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 1–8, 2007.

[153] M. Jurczyk, "Detecting kernel memory disclosure with x86 emulation and taint tracking," 2018.

[154] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," 2003.

[155] N. Nethercote, "Dynamic binary analysis and instrumentation," tech. rep., University of Cambridge, Computer Laboratory, 2004.

[156] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pp. 1–12, 2000.

[157] M. Chabbi, X. Liu, and J. Mellor-Crummey, "Call paths for pin tools," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 76–86, 2014.

[158] A. Schroter, A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pp. 118–121, IEEE, 2010.

[159] R. Vasiliev, D. Koznov, G. Chernishev, A. Khvorov, D. Luciv, and N. Povarov, "Tracesim: a method for calculating stack trace similarity," in *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*, pp. 25–30, 2020.

[160] R. Stoyanov and M. J. Kollingbaum, "Efficient live migration of linux containers," in *International Conference on High Performance Computing*, pp. 184–193, Springer, 2018.

[161] D. Li, B. Reidys, J. Sun, T. Shull, J. Torrellas, and J. Huang, "Uniheap: managing persistent objects across managed runtimes for non-volatile memory," in *Proceedings of the 14th ACM International Conference on Systems and Storage*, pp. 1–12, 2021.

[162] S. Yadalam, N. Shah, X. Yu, and M. Swift, "Asap: A speculative approach to persistence," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 892–907, IEEE, 2022.

[163] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[164] C. J. Xue, Y. Zhang, Y. Chen, G. Sun, J. J. Yang, and H. Li, "Emerging non-volatile memories: Opportunities and challenges," in *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 325–334, 2011.

[165] P. Emelyanov, "Soft-dirty ptes." `https://www.kernel.org/doc/html/v5.4/admin-guide/mm/soft-dirty.html`. (accessed December 8, 2024).

[166] M. Vasavada, F. Mueller, P. H. Hargrove, and E. Roman, "Comparing different approaches for incremental checkpointing: The showdown," in *Linux Symposium*, vol. 69, 2011.

[167] J. Wang, X. Xiong, and P. Liu, "Between mutual trust and mutual distrust: Practical fine-grained privilege separation in multithreaded applications," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pp. 361–373, 2015.

[168] S. P. E. Corporation, "Spec cpu 2017." `https://www.spec.org/cpu2017/`.

[169] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver, "Sources of error in full-system simulation," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 13–22, 2014.

[170] A. Alameldeen and D. Wood, "Ipc considered harmful for multiprocessor workloads," *IEEE Micro*, vol. 26, no. 4, pp. 8–17, 2006.

[171] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 694–701, IEEE, 2011.

[172] D. D. Sharma, “Compute express link®: An open industry-standard interconnect enabling heterogeneous data-centric computing,” in *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*, pp. 5–12, IEEE, 2022.

[173] M. Jung, “Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd),” in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pp. 45–51, 2022.

[174] D. Das Sharma, R. Blankenship, and D. Berger, “An introduction to the compute express link (cxl) interconnect,” *ACM Computing Surveys*, vol. 56, no. 11, pp. 1–37, 2024.

[175] P. Shanthraj, P. Eisenlohr, M. Diehl, and F. Roters, “Numerically robust spectral methods for crystal plasticity simulations of heterogeneous materials,” *International Journal of Plasticity*, vol. 66, pp. 31–45, 2015.

[176] U. R. Kiran, A. Panchal, M. Sankaranarayana, G. N. Rao, and T. Nandy, “Effect of alloying addition and microstructural parameters on mechanical properties of 93% tungsten heavy alloys,” *Materials Science and Engineering: A*, vol. 640, pp. 82–90, 2015.

[177] A. Jahan, K. L. Edwards, and M. Bahraminasab, *Multi-criteria decision analysis for supporting the selection of engineering materials in product design.* Butterworth-Heinemann, 2016.

[178] Y. Choi, I. Song, M.-H. Park, H. Chung, S. Chang, B. Cho, J. Kim, Y. Oh, D. Kwon, J. Sunwoo, *et al.*, “A 20nm 1.8 v 8gb pram with 40mb/s program bandwidth,” in *2012 IEEE International Solid-State Circuits Conference*, pp. 46–48, IEEE, 2012.

[179] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert, “Consistent, durable, and safe memory management for byte-addressable non volatile main memory,” in *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, p. 1, ACM, 2013.

[180] A. Appleby, “Murmurhash3 64-bit finalizer,” tech. rep., Version 19/02/15. https://-code.google.com/p/smhasher/wiki/MurmurHash3, 2011.

[181] A. Snavely and D. M. Tullsen, “Symbiotic jobscheduling for a simultaneous mutlithreading processor,” *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 234–244, 2000.

[182] P. Michaud, “Best-offset hardware prefetching,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 469–480, IEEE, 2016.

[183] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, *et al.*, “Spin-transfer torque magnetic random access memory (stt-mram),” *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 9, no. 2, pp. 1–35, 2013.

[184] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, *et al.*, “Phase change memory technology,” *Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena*, vol. 28, no. 2, pp. 223–262, 2010.

[185] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, “The missing memristor found,” *nature*, vol. 453, no. 7191, pp. 80–83, 2008.

[186] P. Ramalhete, A. Correia, P. Felber, and N. Cohen, “Onefile: A wait-free persistent transactional memory,” in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 151–163, IEEE, 2019.

[187] K. Genç, M. D. Bond, and G. H. Xu, “Crafty: efficient, htm-compatible persistent transactions,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 59–74, 2020.

[188] S. Singh and M. Awasthi, “Memory centric characterization and analysis of spec cpu2017 suite,” in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pp. 285–292, 2019.

[189] V. Gogte, W. Wang, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, “Relaxed persist ordering using strand persistency,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 652–665, IEEE, 2020.

[190] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, “Kiln: Closing the performance gap between systems with and without persistence support,” in *46th Annual IEEE/ACM MICRO*, pp. 421–432, 2013.

[191] K. Matsumoto, T. Ugawa, and H. Iwasaki, “Replication-based object persistence by reachability,” in *Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management*, pp. 43–56, 2022.

[192] R. Elkhouly, M. Alshboul, A. Hayashi, Y. Solihin, and K. Kimura, “Compiler-support for critical data persistence in nvm,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 4, pp. 1–25, 2019.

[193] N. Cohen, D. T. Aksun, and J. R. Larus, “Object-oriented recovery for non-volatile memory,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–22, 2018.

[194] A. D. Breslow, D. P. Zhang, J. L. Greathouse, N. Jayasena, and D. M. Tullsen, “Horton tables: Fast hash tables for {In-Memory}{Data-Intensive} computing,” in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pp. 281–294, 2016.