# Building a More Efficient Cache Hierarchy by Taking Advantage of Related Instances of Objects

**Ziqi Wang**

CMU-CS-22-154

January 2023

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Thesis Committee:**
Todd C. Mowry (Co-Chair)
Dimitrios Skarlatos (Co-Chair)
Nathan Beckmann (CMU)
Mike Kozuch (CMU and Intel Labs)
Gennady Pekhimenko (University of Toronto)

*Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.*



This research was sponsored by Intel ISTC-CC, Google, Facebook, Samsung, and the National Science Foundation under award numbers CNS-1330596, CNS-1618595, and CNS2107307. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

**Keywords:** Computer Architecture, Cache Hierarchy, Transactional Memory, NVM, Cache Compression, Memory Management

*To all the wise minds that collectively push forward*
*the frontier of Computer Science*

*And also*

*To my family*

# Abstract

As the capacity of the cache hierarchy keeps scaling to match the increasing number of cores and growing working set size, the design of the cache hierarchy has remained relatively static, which has become an obstacle in adapting to new hardware devices and software paradigms. Specifically, we identify two major issues with today's cache hierarchy design. First, multiversioning support is lacking, which prevents efficient implementations of newer hardware paradigms, such as Hardware Transactional Memory (HTM), and blocks efficient usage of Byte-Addressable Non-Volatile Memory (NVM). Second, the hierarchy only provides a rigid load-store interface without the capability of leveraging runtime application-level information. As a result, the hierarchy is unable to react to runtime dynamic software behavior and misses opportunities for optimization based on such information.

In this dissertation, we demonstrate that by taking advantage of multiple instances of related objects, we can address both limitations of the existing hierarchy and improve the performance and usability of the system. To validate this statement, we detail four case studies.In the first two case studies, we present **OverlayTM** and **NVOverlay**. Both designs implement a multiversioned cache hierarchy based on Page Overlays. They support a special "Overlay-on-Write" operation that resembles conventional Copy-on-Write, but creates new cache blocks, which we call "versions", directly in the private cache. OverlayTM is a Hardware Transactional Memory design that enables efficient multi-thread synchronization by allowing concurrent writers to the same address to create their private versions without interfering with each other. Furthermore, concurrent readers and writers will also execute conflict-free by directing readers to the version that constitutes a consistent read snapshot image among several possible versions in the hierarchy. The resulting design greatly reduces transaction abort rates and execution cycles compared with a single-version HTM, showing a 30%–90% reduction on both.

NVOverlay further demonstrates the benefits of multiversioning by extending the multiversioning domain to persistent data on Byte-Addressable Non-Volatile Memory (NVM). NVOverlay implements a memory snapshotting design that captures incremental memory modifications within a time interval. NVOverlay generates incremental memory snapshot data with Overlay-on-Write and gradually writes back snapshot data to the NVM for persistence. Snapshot data is managed on the NVM with a series of mapping tables and can be accessed conveniently for failure recovery or other purposes. Our evaluation shows that NVOverlay minimizes the latency overhead of snapshotting by overlapping most of the operations with execution, considerably reducing NVM write traffic with the multiversioning design compared to logging.

In the third and the fourth case studies, we present **MBC** and **Memento**. Both designs leverage application-level information on instances of related objects to optimize performance. MBC performs inter-block cache compression on the last-level cache (LLC). MBC leverages the insight that blocks of similar contents often exhibit a "stepped" (spatially strided) pattern on a page and compresses these blocks together for a higher compression ratio. To identify the per-page stepped pattern, MBC enables application software to pass a "step size" attribute via virtual memory system calls. The attribute propagates in the cache hierarchy, which the cache controller eventually leverages to group blocks based on the attribute for compression. Compared with similar works on inter-block cache compression, MBC greatly simplifies compression hardware while achieving comparable or even better compression ratio.

Memento optimizes object memory allocation in Serverless Computing by offloading high-level allocation primitives to be executed on the hardware. Memento leverages the semantics of user space and OS kernel memory management functions and implements those that can be executed in parallel with the application on hardware, thus overlapping the latency of memory management with execution. On average, Memento reduces main memory traffic by 22% and speed-up function execution by 14%.

# Acknowledgments

Five years of Ph.D. is a long journey. When I stand near the end of it and look back, I can think of many people that I hold in high esteem and feel grateful for. The first two are my advisor, Professor Todd C. Mowry, and Professor Dimitrios Skarlatos. Todd had always been giving me the freedom to explore research topics I am genuinely excited with, and guided me through the early stages of my Ph.D. with his professional feedback. When paper rejections came in, Todd was always the one that encouraged me to never give up, and assisted me in analyzing the weaknesses exposed by reviewers' comments. I dedicate my fullest possible appreciation to Professor Todd for his kind mentorship, intellectual motivation, and academic training.

Professor Dimitrios joined my thesis committee and became my co-advisor in late 2021. Dimitrios never failed to impress me with his endless supply of novel research ideas, his bottomless knowledge base on operating systems and computer architecture, and energetic discussion with me and other students. I will also never forget the admiration I felt when I sent an email to Dimitrios at 4:00 am, and received the reply at 4:10 am. While Professor Todd motivated me more on high-level concepts and ideas, Dimitrios is always the one who pushed me to perfect every single detail of a hardware design, even the most trivial ones among them. I regret that I will graduate soon and can no longer work with Todd and Dimitrios. Had I been given several more years to spend on the enjoyable trip to research land, I genuinely believe they can train me into a better researcher than I am today.

My thesis committee members—Nathan Beckmann, Mike Kozuch, and Gennady Pekhimenko—have all helped me tremendously by providing feedback on my work and sparing time attending the regular research meeting. Their expertise on a broad range of topics proved incredibly valuable for my research. I will always be grateful for Nathan's suggestions on my writing skill evaluation, Mike's comments on Transactional Memory and NVM, and Gennady's inputs on cache and memory compression.

Before I joined the Ph.D. program, I was under Professor Andrew (Andy) Pavlo's supervision at Carnegie Mellon Database Group, working on the OpenBwTree project. Andy taught me, the then master's student, all the necessary skills to author a research paper and encouraged me to submit the draft to SIGMOD. The project turned out to be a big success, which earned me a ticket to SIGMOD 2016 and opened the door for me to meet with people in academia. Andy amazed me with his energetic working ethic, his humorous (and sometimes backfired) presentations, his profound understanding of main-memory databases, and "street knowledge".

My family—my parents and grandparents—showed unparalleled trust and emotional support during the course of my Ph.D. study. They may not understand even the slightest bit of my ongoing work, but they were always willing to listen when I explained, cheered for me on every single piece of progress I have made, and comforted me when the outcomes of paper submissions did not go as I expected them to. My family is a great treasure that I can take with me in my heart. On cold winter nights when I stayed up alone, they granted me warmth and showed me the path.

Friends—those who are a pleasure to share your thoughts with and who will turn you slightly depressed when they are not around for a while—also constituted an indispensable part of my past five years. It was not about research, not at all, but about everything other than research. We posted news feeds to each other and commented on them. We discussed video games, geopolitics, and celebrities. We forbad certain topics and people in our chat for our shared despite towards

them. We congratulated each other very often, with neither of us being aware of what in particular we were congratulating for. Maybe it is a bond that just kept bonding by saying that word.

I am also particularly grateful to my collaborators and research colleagues. Specifically, I would like to thank (i) Kaiyang Zhao for his unique work on adapting QEMU for full-system simulation; (ii) Vivek Seshadri for introducing me to Page Overlays, as well as his kind support on the related subject; (iii) Lin Ma for his general collaboration on my master's research and a more recent clarification on the graduation process; (iv) Andrew Jacob for his diligent work on transforming DeathStarBench to serverless functions; and (v) everyone else in Andy's Database Group and Dimitrios's CAOS group that I have worked with. Without their help, research would be much harder than it seemed to be.

When Covid-19 hit in early 2020, it was a difficult time for everyone because of the uncertainty of life, the fear for one's personal health, and the lost opportunities. I am glad that I was surrounded by positive (no pun intended) people that always held an optimistic attitude toward life, and they told me things would get better. Indeed, the situation did become better, and everything, including daily life and research work, was back on track shortly without causing too much harm. The pandemic was a tragedy and an unforgettable experience that changed many people's Ph.D. life, including mine. Nevertheless, we were together strong and as a firm unity in Computer Science Department of Carnegie Mellon. I dedicate this paragraph to those unnamed heroes who have committed their efforts or even lives to defend the future of human society, and to those who kept searching for light even during the darkest of times.

Lastly, my special thanks to all anonymous reviewers and their comments on my paper drafts. Harsh or gentle, critical or favorable, reject or accept, their feedback had constantly made myself reexamine my work, better shaped my motivation, and generally helped me improve the quality of my research. We may never know each other due to the double-anonymity rule of major conferences, but those who have helped me from the shadow will be remembered.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Hardware cache is effective in addressing the widening gap between the speed of the processor and the latency of the main memory, a phenomenon known as the memory wall [169, 259]. To better leverage the trade-off between cache capacity and access latency, architectures typically have several levels of cache, naturally forming a *cache hierarchy*, with the smaller and the faster cache placed closer to the processor, and the larger and slower cache closer to the main memory.

Despite great capacity scaling over the last few decades [18, 66, 95, 185, 240], the cache hierarchy has barely seen any change in how it organizes cached data and interacts with application software. More specifically, the current cache hierarchy only exposes a flat address space reflecting the physical address space backed by the underlying main memory and software can only accesses the cache hierarchy via a load/store interface.

While assuring backward compatibility with legacy components in the system, such a simple and overly conservative cache hierarchy design can become an obstacle in adapting to new hardware devices and software paradigms. This statement is especially true given today's rapid advancement of both hardware and software technology. Even on existing systems, the cache hierarchy may force programmers to adopt suboptimal solutions due to ignorance of application-level data organization and runtime information.

In this dissertation, we focus on two properties of the current cache hierarchy design that prevents the software from utilizing it efficiently. First, the current hierarchy only exposes a flat address space abstraction. As a result, applications that benefit from multiversioning must resort to software implementations, which incur significant overhead on version lookup and metadata management. Second, the load/store interface of the hierarchy neglects runtime application-level information, while the software mechanism for passing such information to the hierarchy is non-existent. Consequently, the cache hierarchy is unable to react to runtime software behavior dynamically and can miss precious opportunities for optimization.

## 1.1 Supporting Multiversioning in the Cache Hierarchy

Multiversioning is a common software paradigm that is not supported by the current cache hierarchy, and software pays the price for using it. For example, in transactional memory (which also extends to the transaction engine of Database Management Systems), concurrent transactions

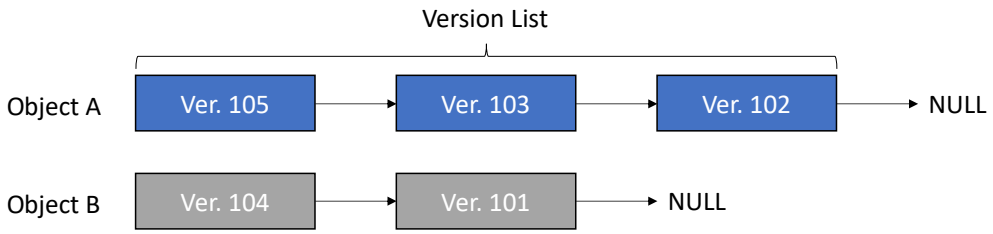



Figure 1.1: **Software Managed Versions Example** – This figure depicts two objects, A and B, and the software managed versions maintained in linked lists. Each version in the list is tagged with the version number. Also, versions are linked together in decreasing order.

that write to the object generate multiple copies of the same memory object to avoid write conflicts. Each object copy is denoted as a "version". Without hardware support, these versions must be explicitly managed by software. Fig. 1.1 presents an example of software-managed versions. Each version is tagged with a software-assigned version number representing its identity. Different versions of the same object are chained together in a linked list.

Software-managed versions incur performance overhead. First, when a memory object is to be accessed, the software must perform a *version lookup* to determine the correct version. Second, versions and their metadata must be explicitly created, stored, and recycled by software. Unfortunately, the cache hierarchy does not support directly performing version lookup and metadata maintenance on hardware due to the flat address space it exposes.

The lack of multiversioning support also hinders the adoption of new hardware. With the introduction of Byte-Addressable Non-Volatile Memory (NVM), taking periodic memory snapshots becomes a feasible approach for crash recovery. Memory snapshots are naturally multiversioned, as multiple copies of memory states on the same address are saved, one for each snapshot. Again, due to the flat address space exposed by the cache hierarchy, the generation and storage of multiversioned snapshot data cannot be performed directly on hardware. Instead, programmers have to implement software libraries as an extra layer of indirection, resulting in suboptimal performance.

Prior works propose various hardware multiversioning designs to support transactional memory (e.g., [23, 97, 156, 177, 209]) and NVM-based memory snapshotting (e.g., [189, 190]), but they fail to provide a satisfactory solution. First, these proposals primarily focus on single problem solving with highly specialized hardware. The resulting design lacks generality and cannot be reused easily for other multiversioning-related tasks. Second, none of these designs implements multiversioning in the *entire* cache hierarchy. As a result, they still suffer version-related performance overhead, such as logging or extra indirection.

## 1.2 Leveraging Runtime Application-Level Information

The current cache hierarchy is also hampered by its inability to leverage runtime application-level information. Such information can be classified into two categories. The first category contains software's runtime behavior, which the hierarchy can leverage to adjust its operation policy dynamically. The second category communicates high-level software primitives to the hardware, such that the cache hierarchy can execute these primitives more efficiently using dedicated functional units.

One particular scenario that can benefit from the first category is identifying redundant

blocks on the address space for inter-block cache compression. Without runtime information on redundant blocks, as is the case with the current hierarchy, hardware needs to perform extensive searches over a potentially large range of blocks, the overhead of which is intolerable. Prior works [19, 83, 84, 242, 245] attempt to address this issue with extra per-block metadata maintained in hardware structures. These solutions, however, incur substantial hardware overhead and have limited scalability due to the fixed-size structures. In reality, redundancy information is often quite obvious and easily obtainable in software. Unfortunately, today's cache hierarchy does not support software passing redundancy information to the hardware, hence missing the opportunity to leverage applications' runtime data patterns to assist compression.

The cache hierarchy can also be extended to leverage the second category to fit better into an emerging software paradigm called Serverless Computing. In Serverless Computing, heap memory allocation constitutes a non-negligible performance bottleneck. However, most of the allocations are short-lived and of small sizes—a common case that can be easily offloaded to the hardware and removed from the execution critical path. The previous work [126] partially achieves this goal by offloading certain low-level primitive operations to the cache. The resulting speedup is somewhat limited because the cache hierarchy does not understand the "big picture" behind these low-level primitives. It is thus helpful to enable the cache hierarchy to leverage high-level software primitives in this case, such that heap memory allocation can be handled entirely and efficiently by the hardware.

## 1.3 Thesis Statement and Insights

The goal of this dissertation is to address the above two limitations of the existing cache hierarchy and hence improve the efficiency of the hierarchy for a wide range of hardware and software paradigms. We strive to accomplish this goal with general-purpose and simple hardware additions that enable the hierarchy to take advantage of multiple instances of related objects. Our work evolves around the following statement:

### Thesis Statement

*By enabling the cache hierarchy to take advantage of multiple instances of related objects, the efficiency of the cache hierarchy can be drastically improved for a variety of important software operations.*

To validate the thesis statement, this dissertation details four case studies that exercise the principle in the statement.

To demonstrate the benefit of creating multiple instances of objects for transactional synchronization, we propose a Hardware Transactional Memory (HTM) design **OverlayTM** that utilizes a fully multiversioned cache hierarchy. OverlayTM implements version lookup and management in hardware, with no software overhead incurred. Since all levels of the cache hierarchy are multiversioned, OverlayTM naturally supports large-than-cache transactions without any added complexity.

We also show that creating multiple instances of blocks on the same physical address can benefit the adoption of new hardware. To this end, we propose an NVM-based memory snap-

shotting framework **NVOverlay** that captures memory snapshots atomically in a multiversioned hierarchy. NVOverlay represents memory states on the same address that belong to different snapshots as different versions, and it flushes these versions into the NVM for persistence. Both version generation and flushing are carried out by the hierarchy autonomously, which overlaps the latency of persistence with normal execution, hence having little to no cycle overhead. NVOverlay also consumes less bandwidth because each version is only persisted once, rather than twice as in logging. The latter is made possible using a technique called shadow mapping, which can be implemented with minimal hardware additions on top of the multiversioned hierarchy.

OverlayTM and NVOverlay are built upon a virtual memory framework, Page Overlays [226]. Not surprisingly, their designs share much in common, especially on the components that provide multiversioning capability. Page Overlays is covered in Section 2.1 as background material.

To demonstrate the benefits of identifying multiple instances of related objects for compression, we propose an inter-block cache compression design, **Multi-Block Cache Compression (MBC)**, that allows application programs to specify the pattern of redundant blocks within a page. In MBC, the redundant block pattern on a single page is represented using a "step size" attribute as these blocks often exhibit a "stepped" (spatially strided) pattern. The patterns result from arrays-of-structs or heap allocators placing same-type objects next to each other on the same page. The "step size" attribute is specified by the application at memory allocation time, stored in the page table, and propagated to the TLB and LLC along with the handling of memory requests.On receiving the request carrying the "step size" attribute, the LLC hardware performs compression across redundant blocks, as indicated by the attribute. By leveraging software-provided runtime redundancy information, MBC eliminates the extra hardware structure storing redundancy metadata, simplifying hardware greatly while achieving similar or better results.

The last case study demonstrates how maintaining instances of objects on a hardware allocator could optimize memory allocation in Serverless Computing. To serve this purpose, we propose **Memento** as a holistic approach to heap memory allocation. Memento offloads both user space heap object allocation and kernel space page management on the allocation path to specialized hardware components. As a result, the latency of object allocation is reduced to only a couple of cycles, and most of the memory allocation work can be moved to the background. Instead of executing low-level primitives that only constitute a fraction of the allocation path, Memento replaces `malloc`, `free`, `mmap`, and `munmap`, in their entirety, with high-level allocation primitives whenever applicable. Being able to see the "big picture" of memory allocation by leveraging high-level primitives helps Memento to achieve higher speedup.

## 1.4 Contributions

This dissertation makes the following key contributions:

### OverlayTM

- We propose Multiversioned Page Overlays (MPO), a multiversioned cache design based on Page Overlays. The MPO tracks versions in the cache hierarchy using a unique hardware structure, i.e., the Version Directory, and implements a Version Access Protocol to support version reads when multiple versions are present.

- Besides multiversioning, OverlayTM also implements a protocol for detecting serialization conflicts between concurrent transactions. OverlayTM guarantees atomic and serializable transaction execution. The protocol includes a commit-time validation protocol and a hardware Commit Queue.
- We quantitatively evaluate OverlayTM against other hardware single-version and multi-version HTM designs on STAMP benchmark and report speedup and abort rate reduction.

### NVOverlay

- We propose Coherent Snapshot Tracking (CST), a multiversioned cache design based on Page Overlays. The CST captures fine-grained incremental memory modifications by creating versions in the cache hierarchy. The CST also coordinates version write-backs to the NVM in a distributed and scalable manner.
- We also propose Multi-snapshot NVM Mapping (MNM) to manage multiple memory snapshots on the NVM using a series of mapping tables. The mapping tables are merged into a master table which maps the most recent consistent snapshot.
- We quantitatively evaluate NVOverlay against other software and NVM-based memory snapshotting designs on STAMP benchmark and report cycle overhead and bandwidth consumption.

### MBC

- We propose MBC, an inter-block cache compression design that leverages software specified per-page "step size" attribute to compress across stepped (spatially strided) redundant blocks.
- To provide a high compression ratio and a matching effective cache size, we also propose a customized dictionary compression algorithm and a novel cache tag organization.
- We quantitatively evaluate MBC against other inter-block cache compression designs on data workload and SPEC benchmark and report speedup and effective cache size.

### Memento

- We propose Memento, a holistic approach for offloading heap memory allocation to hardware. Memento focuses on Serverless Computing, where most objects are small and short-lived.
- To execute memory management on hardware, we propose a Hardware Object Table (HOT) that handles user space object allocation replacing the software object allocation library. We also propose a Hardware Page Manager that performs page-level allocation and handles page walks, which replaces OS kernel components for page management.
- We quantitatively evaluate Memento on Python and C++ serverless workloads and report speedup and memory consumption.

## 1.5 Outline

The remainder of the dissertation is structured as follows. Chapter 2 presents the background material and the motivation of the dissertation. Chapter 3 presents the design of OverlayTM, the

multiversioned Hardware Transaction Memory. Chapter 4 presents the design of NVOverlay, an NVM-based memory snapshotting framework on Byte-Addressable Non-Volatile Memory. Chapter 5 presents the design of Multi-Block Compression, an inter-block compressed cache leveraging software-assigned "step size" attribute to identify redundant blocks on the same page. Chapter 6 presents the design of Memento, a holistic approach to memory management of small and short-lived objects for Serverless Computing. At last, Chapter 7 concludes this dissertation and discusses future research directions.

# Chapter 2

# Background

## 2.1 Page Overlays

### 2.1.1 Design Overview

Page Overlays [226], initially proposed by Seshadri et al., is a novel virtual memory framework that enables a more efficient form of Copy-on-Write (CoW) in the cache hierarchy. CoW is an existing virtual memory technique utilized by OS kernels to perform page copies lazily. When a virtual page is to be copied (e.g., during a `fork`), instead of eagerly copying the underlying physical page, the OS kernel simply creates a new virtual page mapping pointing to the physical page, and sets the permissions of both the new and existing virtual pages to read-only. If neither of the two pages receives any future write, then the OS saves processor cycle and memory bandwidth by not performing the copy. However, when a write operation is about to modify the underlying physical page, the actual copy of data must be conducted, hence the name "Copy-on-Write". The CoW process involves three steps. First, a new physical page frame is allocated as the copy destination. Second, the content of the page is duplicated to the destination. Lastly, the virtual-to-physical mapping of the page receiving the write is updated to point to the destination page, on which the write operation is eventually performed.

Page Overlays was originally motivated by the unnecessarily high overhead of page-level CoW, which always copies an entire 4KB page (or 2MB huge page) regardless of the size of the write. Page Overlays leverages the observation that each write operation only affects a small fraction of data on a page, and hence proposes "Overlay-on-Write", a mechanism that enables both data copying and address mapping to occur at 64-byte cache block granularity. Overlay-on-Write minimizes the latency and bandwidth overhead incurred by data copying because the number of 64-byte blocks that need to be duplicated is typically only one or two, which is a tiny amount compared with a full 4KB page copy that duplicates all 64 cache blocks.

With Overlay-on-Write, the latter half of conventional CoW is changed as follows. First, when a write is about to land, the OS issues a special *overlay write* instruction for every 64-byte block affected by the write. Second, the cache controller executes every overlay write by performing an *in-cache duplication* of the cache block located by regular cache lookup, creating a new *overlay block* directly in the hierarchy, and applies the write to the new block. To avoid address aliasing, the cache controller derives a new address from the write operation's target address and assigns it

to the newly created block. We discuss address assignment in more detail in Section 2.1.2. Lastly, the address mapping is updated by hardware, which is performed in 64-byte block granularity. After the update, future memory accesses on the affected address will be redirected to the newly allocated block, while accesses to the rest of the page would still land on the existing page. We cover fine-grained address mapping in Section 2.1.3.

Page Overlays does not reserve backing storage when new cache blocks are created with Overlay-on-Write. As a result, when the block is written back to the main memory for the first time, the backing storage must be allocated on the write-back path. This step is unique to Page Overlays, and is non-existent in the current system because the current system implicitly direct-maps blocks on the physical address space to storage locations of the main memory. However, blocks created by Overlay-on-Write may have identical physical address tag values, and therefore, cannot be direct-mapped using their physical addresses alone. Page Overlays resolves this problem by introducing an extra level of indirection on the memory controller and allocating backing storage for newly created blocks lazily on the first write-back. We discuss this unique address indirection and backing storage allocation mechanism in Section 2.1.4.

### 2.1.2 Assigning Addresses to Overlay Blocks

As discussed earlier, when a block is duplicated in a cache during Overlay-on-Write, a new address must be assigned to make the new block addressable. Page Overlays hardware, in fact, just copies the physical address tag of the old block and assigns it to the new block. To avoid address aliasing, Page Overlays proposes adding an extra 16-bit "Overlay ID (OID)" field in addition to the address tag for all blocks in the hierarchy. When an Overlay-on-Write is executed, the OID tag of the newly created block is given the Process ID (PID) of the process that performs the Overlay-on-Write operation.

The Overlay ID serves as a disambiguation field for blocks on the same address created by Overlay-on-Write from different processes. When a process intends to read an overlay block created by itself earlier, it must issue an *overlay read* instruction, which carries the reading process's PID as well as the virtual address of the block to be read. The block's virtual address is translated by the TLB to the physical address as usual, after which *both* the physical address and the PID are used to perform cache lookup. [1] An overlay read cache lookup hits a block if and only if (i) the address tag matches the requested address, and (ii) the OID tag matches the PID in the read request.

### 2.1.3 Fine-Grained Address Mapping In Block Granularity

Another challenge that Page Overlays addresses is to update the fine-grained address mapping in block granularity when a new block is created via Overlay-on-Write. This step is essential

[1] In the original Page Overlays paper, cache blocks created by Overlay-on-Write use the virtual address of the block to be written. Correspondingly, the virtual addresses in overlay read and write instructions are directly used for cache lookup, and *not* translated by the TLB. In this dissertation, we override this design decision and force overlay memory operations to always go through the TLB. This change will invalidate some Page Overlays use cases, such as sparse matrix, but it avoids the nasty synonym problem known for virtual address caches [88]. A more comprehensive design may also add a switch that allows users to choose between the two modes.

to ensuring that future reads and writes from the same process will correctly land on the newly created block, rather than on the page underneath. For overlay reads, this challenge is already solved because overlay reads just use the current PID and the requested virtual address to generate the final address for cache lookup.

However, for overlay writes, the write semantics depend on whether an overlay block already exists on the address to be written. If a block exists, then overlay write will be executed as regular write on the overlay block. Otherwise, the overlay write is executed as an Overlay-on-Write described above, which creates a new overlay cache block. The distinction of overlay write semantics is analogous to how conventional CoW handles writes. Before the first write is performed on one of the virtual pages, read operations on either page will land on the same physical address. However, after the first write, both virtual pages will obtain their specific physical page, and all future reads and writes will land on the corresponding physical page.

While conventional CoW relies on the OS kernel to track the CoW status of each page, Page Overlays, on the contrary, tracks per-block overlay status on hardware. To remember whether an overlay block already exists for the currently executing process, Page Overlays proposes adding an "OBitVector" field to all TLB entries. The OBitVector consists of one-bit flags for every cache block on the page to indicate whether an overlay block exists. The size of OBitVector equals the number of 64-byte blocks on the page, adding a negligible one bit overhead to every 64 bytes (0.2% total storage).

Since the OBitVector tracks overlay write status for every page in every process, it is also backed by the main memory on a per-process basis. When a TLB entry is replaced, the OBitVector of the replaced entry will be written back to the backing location. Moreover, the OBitVector of the newly inserted TLB entry will be loaded from the backing location. The original Page Overlays proposal does not specify precisely how OBitVector is maintained in the main memory, but in practice, it can be either stored in vacant bits in the existing page table or a separate per-process structure.

When a process is spawned, the OBitVector fields for all pages are, by default, all-zeros, indicating that no overlay block exists yet for the new process. When an overlay write is performed on an address for the first time in a process, the corresponding bit in the OBitVector is set in the TLB entry, and the entry is marked dirty. The overlay write, in this case, is executed as Overlay-on-Write. Future overlay writes in the same process will see the "1" bit in the OBitVector, and will thus be executed as regular writes to the overlay block. The tag lookup process in the latter case is identical to the one for overlay reads, which takes both the physical address and the PID of the requesting process as inputs.

### 2.1.4 Allocating Main Memory Backing Storage for Overlay Blocks

The main memory storage location of an overlay block is allocated *lazily* when the block is written back from the cache hierarchy for the first time. This design decision is deliberately made such that backing storage allocation is *not* on the memory write critical path. Instead, in Page Overlays, storage allocation for overlay blocks is performed on the eviction path, which is usually not time-critical and can be conducted in the background to overlap with execution.

Due to the possibility of multiple overlay blocks with the same physical address tag but on different OIDs, the challenge of Page Overlays storage allocation is that overlay blocks can no

longer be direct-mapped to the main memory using the physical address alone. Page Overlays addresses this challenge by adding per-OID *Overlay Memory Tables (OMTs)*, the structure of which resembles that of regular page tables, and translating the addresses of overlay blocks to their physical storage locations on the memory controller by walking the OMT. This process takes place in the following steps. First, the OID of the block is checked. If OID equals zero, indicating a non-overlay block, the write-back is handled as usual, and the block is written back to the physical address in the main memory (i.e., direct-mapped). However, if OID is non-zero, necessitating an overlay address translation, then the OMT is located using the OID. Second, the memory controller walks the OMT until reaching the leaf entry (it also extends the OMT if any middle-level node is not present). If the leaf entry is valid, then the write-back operation concludes by writing the block into the address stored in the leaf entry.

Otherwise, if the leaf entry for the address is not valid yet, and the request that triggered the OMT walk is a dirty data write-back from the hierarchy, then Page Overlays performs backing storage allocation. To fulfill this task, the memory controller will first allocate a page from a free page pool and then create a leaf entry in the OMT pointing to the newly allocated page. Storage allocation for overlay blocks written back for the first time completes at this stage, after which the block data is written into the newly allocated page. Future main memory accesses to the overlay block will go through the same address translation process and correctly land on the memory page allocated in this step.

The page pool from which the memory controller requests new pages is managed by a hardware page allocator called the *Overlay Memory Store (OMS)*. The OMS interfaces with the OS kernel to replenish pages when the number of free pages drops below a predetermined threshold. It also returns surplus pages to the OS after they are freed or when demanded by the OS to avoid depleting physical pages in the kernel.

Adding address translation to the memory controller increases overlay memory access latency, which is also, unfortunately, on the main memory read critical path. To overcome this latency issue, the authors proposed adding a TLB-like structure on the memory controller called the *OMT Cache* for caching OMT entries that are frequently accessed. Address translation can thus be performed by fast associative searches on the OMT Cache most of the time rather than by costly OMT walks. In addition, the OMT Cache can be populated by prefetching OMT entries on data TLB misses. Prior works suggest that there is a strong relation between TLB misses and subsequent cache misses that lead to main memory accesses (happens on $> 98\%$ of cases) [32]. This prefetching technique can, therefore, eliminate most OMT walks from the critical path, minimizing the adverse effect of OMT walks on memory access latency.[2]

Page Overlays also optimizes for sparse overlays, where a page only contains a few overlay blocks. Sparse overlays cause storage underutilization, which should be minimized as much as possible. To improve the storage efficiency of sparse overlays, Page Overlays manages physical pages in five different size classes (256B, 512B, 1KB, 2KB, and 4KB). The size class of the page used to back an overlay block is determined based on the fullness of the page. Moreover, the size class will also change dynamically as the fullness of the page changes. The original Page Overlays paper also proposes compact storage formats for every size class, which we do not cover

[2]TLB-triggered prefetching of OMT Cache is not discussed by the original paper. We added this for completeness. In later discussions, we assume that the OMT does not affect memory access latency in any noticeable manner.

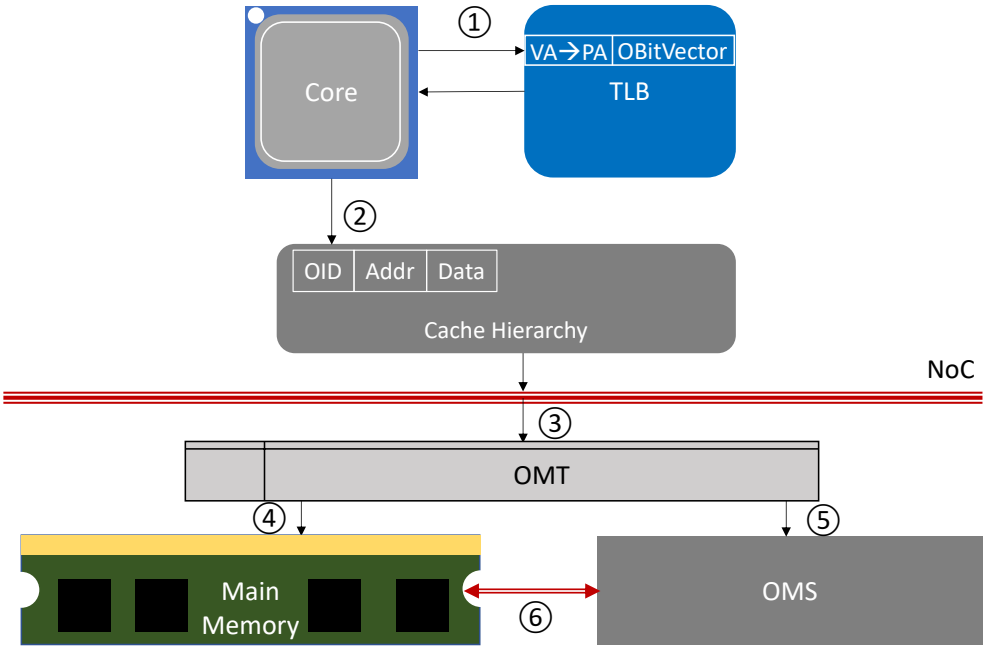


Figure 2.1: **Page Overlays Workflow** – This figure depicts the overall Page Overlays design.

in this dissertation, as this feature of Page Overlays is orthogonal to our discussion. Interested readers may refer to [226] for more information on compact storage formats of overlay pages.

### 2.1.5 Putting It All Together

Fig. 2.1 depicts the overall Page Overlays workflow. The processor initiates an overlay memory access when executing an overlay memory instruction. In the first access step, the MMU generates the overlay address for cache access using both the virtual-to-physical mapping and the OBitVector in the TLB entry (①). Then the memory request is sent to the cache hierarchy for address lookup, which compares both the OID and the address tag of cache blocks in the hierarchy (②). If the access misses the hierarchy, the request is further forwarded to the memory controller (now shown in the picture), which has a built-in OMT that performs address translation (③). If the memory request refers to non-overlay memory (i.e., with OID being zero), then the address is direct-mapped into the main memory (④). Otherwise, if the request refers to overlay memory, then an additional address translation is performed by the OMT, which maps the overlay address to an address in the OMS (⑤). The OMS maintains a collection of pages on the hardware for backing overlay cache blocks written back from the cache hierarchy. The OMS may also acquire or release physical pages from/into the kernel-managed page pool when necessary (⑥).

### 2.1.6 Page Overlays in This Dissertation

Page Overlays also enables many other use cases, despite being proposed initially as a low-overhead alternative to Copy-on-Write. The broad applicability of Page Overlays originates from its ability to create and manage overlay blocks on hardware (the original paper enumerates several scenarios that can benefit from Page Overlays. Interested readers are encouraged to find the

complete list in [226]). In particular, Page Overlays enables the cache hierarchy to (i) create blocks in-cache and address them with an extra OID, and (ii) store overlay blocks that are potentially on the same address but with different OIDs in distinct locations for future access. These two properties, combined, constitute Page Overlays' "hardware-supported multiversioning" capability, which also forms the basis for the two hardware designs, i.e., OverlayTM and NVOverlay, that we will present in the first half of this dissertation. In the rest of this section, we provide a high-level preview of the two hardware designs and show how they can benefit from hardware-supported multiversioning.

**Creating blocks in-cache and accessing them by OIDs.** Page Overlays' "Overlay-on-Write" semantics create a new overlay block by duplicating an existing block in the L1 cache and assigning it the same physical address as the source block and a different OID. This process is lightweight and localized to the L1 cache in which the overlay write is executed. Besides, when multiple overlay blocks on the same address are present in the cache, these blocks can be accessed separately by overlay reads using the address and the respective OIDs.

Being able to create blocks in-cache and access them by OIDs proves to be valuable for designing Hardware Transactional Memory (HTM). Imagine the scenario where several concurrent transactions all write to the same address. A straightforward implementation is to let each writer create an overlay block with different OIDs. The Page Overlays design ensures that these transactions can only see their private modifications using the respective OIDs, without accidentally stumping upon the working set of others—a property called *atomicity*. By contrast, on a regular hierarchy, enforcing multi-writer atomicity is challenging because most coherence protocols follow a "single-writer" rule. As a result, concurrent writes to the same block will expose the block to all writer transactions, therefore breaking atomicity.

Overlay-on-Write also proves to be critical for capturing incremental memory snapshots. At a high level, when a memory snapshot is taken, the cache hierarchy can atomically "freeze" the content of memory by treating all future memory writes as overlay writes. This operation has two implications. On the one hand, existing cache blocks in the hierarchy become immutable, and they represent the in-cache part of the memory state before the snapshot operation. On the other hand, future memory writes will be executed with "Overlay-on-Write" semantics and create new overlay blocks instead of overwriting existing ones in-place, hence protecting the memory state before the snapshot from being altered by memory writes after the snapshot. This process is analogous to conventional memory snapshotting techniques based on page-level Copy-on-Write, but it is much more lightweight, requires no software coordination, and can be conducted extremely fast. Consequently, with Page Overlays, memory snapshots can be taken on hardware at a much higher frequency than existing software techniques while incurring less overhead. The resulting hardware design thus offers many more opportunities of applying memory snapshotting to solve real-world problems than existing techniques.

**Storing overlay blocks that are potentially on the same address but with different OIDs in distinct locations for future access.** Instead of direct-mapping overlay blocks to the main memory using physical addresses alone as in the current system, Page Overlays stores these overlay blocks in the OMS, and indexes them with OMTs, which are per-OID page address mapping tables translating overlay blocks' physical addresses to their storage locations in the OMS. This property simplifies both HTM and memory snapshotting hardware designs. For HTM, when an overlay cache block storing speculative working data of a transaction is evicted from

the hierarchy, the block can be naturally stored in an OMS page that is private to the transaction. The resulting HTM design supports *unbounded transaction* whose working set size exceeds the capacity of the cache hierarchy, without adding significantly more hardware other than Page Overlays itself. By contrast, implementing unbounded transactions on systems without Page Overlays is notoriously difficult, which is evidenced by the limited number of HTM proposals that support this feature (e.g., [16, 211]).

Hardware-managed OMS and OMT also prove to be helpful when designing hardware for memory snapshotting, as the design can simply rely on the OMS as an existing storage manager and treat OMTs as indexes for accessing previously saved snapshot data. The resulting hardware design only brings incremental changes to existing Page Overlays hardware because most of the functionality required for managing snapshot data is already present in Page Overlays.

## 2.2 Hardware-Supported Multiversioning in HTM and NVM-Based Memory Snapshotting

This section covers background material and related work for the first half of the dissertation: hardware-supported multiversioning. We focus on the two case studies discussed in Section 1.1, namely Hardware Transactional Memory (HTM) and NVM-based Memory Snapshotting. For each topic, we discuss the main challenges of system design and familiarize readers with the design space by reviewing related work.

### 2.2.1 Hardware Transactional Memory

Transactions are code sequences that obey two principles: *atomicity* and *serializability* [99]. [3] By observing atomicity, memory operations in a transaction are either executed as an indivisible unit, or not executed at all. In the former case, the transaction *commits*, and no other transaction can access the intermediate states of the execution. Otherwise, the transaction *aborts*, and none of its memory operations takes effect, as if the execution had never occurred. Serializability further defines the logical ordering of transactions. For Hardware Transaction Memory, serializability requires that the memory states observed by individual transactions should be identical to those in a globally agreed, serialized execution. This condition is essential for ensuring that transactional execution can replicate the behavior of a serialized execution, which must be enforced *all the time* for correctness.

Typical usages of transactions include implementing lock-free data structures, executing database queries, enforcing sequential consistency, and Thread-Level Speculation. In lock-free data structures, transactions reduce the overhead of multi-thread synchronization while preserving the consistency of the data structure. Databases also implement what is called "Concurrency Control Protocol" to protect queries from concurrent updates, such that the given *isolation level* is met. The concurrency control protocol bears certain similarities with transactional memory,

[3] In the context of DBMS, transactions have a slightly different definition as obeying the "ACID" (atomicity, consistency, isolation, and durability) principles [235]. In this dissertation, unless otherwise noted, we stick to the definition by Herlihy et al. [99].

but it is more data-centric, hence allowing non-serializable data transformation as long as it is legal under the isolation level. On multicore systems that implement weaker memory consistency models, Sequential Consistency (SC) can be enforced by executing everything as transactions and committing them continuously [42, 97]. Since transactions are globally ordered, all memory operations in a transaction are also globally ordered with memory operations in other transactions, which happens to match the semantics of SC. Thread-Level Speculation [41, 50, 75, 96, 237, 238] breaks down the serial execution of a program into many smaller pieces and dispatches them to multiple cores for parallel execution. Each smaller piece is executed as a transaction. The logical ordering of these transactions is pre-determined to match the program order.

Hardware Transactional Memory (HTM) enables transaction execution on hardware. Two major challenges of designing HTMs are *version management* and *conflict handling* [34]. Both challenges can be addressed with some form of hardware-supported multiversioning.

**Version management.** When a transactional write modifies a block, the block becomes part of the transaction's intermediate state. We denote such a block as a "version". According to the atomicity principle, transactions should never observe other transactions' intermediate states. Therefore, versions should be kept private to the transaction until the transaction commits or aborts.

The problem inevitably arises when multiple transactions concurrently modify a block. On the one hand, each transactional write will generate a version, and every transaction should be capable of accessing its private version without interfering with others. On the other hand, today's cache hierarchy most likely only supports one version per address due to the "single-writer" rule of the coherence protocol [197, 239]. HTM designers hence need to devise unique mechanisms in the hierarchy in order to handle the multi-writer scenario.

The earliest HTM proposal by Herlihy et al. [99] addresses this challenge by limiting the number of concurrent writers per block to one, and enforces the rule by aborting concurrent writers until the number of writers becomes one. This earliest idea is later adopted into commercial HTM products [109, 270] due to its simplicity and non-intrusiveness to the coherence protocol. [4] While preserving the single-writer rule, this design does not allow any write concurrency on the same address. As a result, it demonstrates very limited performance benefit under high concurrency [34]. Other HTM designs that follow this approach include [16, 113, 177, 180, 211, 269].

Later HTM proposals get rid of the single-writer limitation by using the per-core private hierarchy (i.e., L1 and L2 cache) or dedicated hardware structure (i.e., the L0 cache in [209]) as a buffer for storing versions generated by transactions [23, 41, 42, 43, 97, 199, 207, 208, 209, 243]. These versions are, by design, "invisible" to the coherence protocol because the protocol will not register the corresponding cores that perform the transactional writes as writers. While this technique increases write concurrency, it may still violate atomicity when a version is evicted from the private hierarchy to the shared LLC. In this case, the version will be exposed to concurrent transactions, causing the creating transaction of the evicted version to abort for correctness.

Alternatively, HTM designs may also enable the multi-writer scenario by modifying the coherence protocol to track versions generated by different transactions [75, 212]. The modifications to the coherence protocol are typically quite radical, which makes this approach unrealistic due to its high design and verification cost. For example, DATM [212] has eleven stable states (and even

[4] Unfortunately, Intel has decided to disable the HTM support on a broad range of future processors [107].

more transient states), and HMTX [75] has seven states plus two per-block ID fields. Both designs have a substantially larger coherence state machine compared to the standard MESI protocol, where only four stable states are present.

SI-TM [156] differs from all the above approaches by assigning shadow addresses to versions. SI-TM reserves shadow memory in the physical address space for storing transactions' private data. Every transactional write will cause a new shadow location to be allocated for the write address. Concurrent transactional writes will be performed on different shadow addresses instead of interfering with each other via coherence. To manage versions on the same address, SI-TM proposes adding a Multiversion Manager (MVM) between the private hierarchy and the LLC. The MVM translates between the shadow address and a version's canonical address (the address that processors use to issue memory requests). The private hierarchy can thus still access versions using their canonical addresses. When a version is evicted from the private hierarchy, the MVM translates the canonical address to the shadow address. The LLC can thus just treat the evicted version as a regular block.

To summarize: HTM's version management scheme addresses the core challenge of handling concurrent writes. The cache hierarchy's coherence protocol is the most significant factor in designing the version management scheme. HTM designs may either follow the same "single-writer" rule as the coherence protocol does or buffer versions in private structures hiding them from the coherence protocol, or assign shadow addresses to versions and hence "trick" the coherence protocol into treating them as individual cache blocks. All these different schemes trade-off between write concurrency, protocol complexity, and hardware overhead.

**Conflict handling.** Transactions conflict with each other via interleaved reads and writes on the same physical address. [5] Conflicts form data dependencies between transactions and define transactions' serialization order. HTM designs deal with conflicts with a conflict handling protocol that aims to produce serializable executions. The conflict handling protocol is responsible for (i) detecting the formation of cyclic dependencies between transactions, and (ii) resolving the conflict by either aborting or stalling a transaction. These two responsibilities are also sometimes separately introduced as "conflict detection" and "conflict resolution", respectively.

Transactions can conflict in three patterns. First, if two running transactions read and write the same address, respectively, and one of them commits, then it forms a "Read-Write (RW)" conflict, and the reader is logically ordered before the writer. Second, if both transactions write the same address, and one of them commits, then it forms a "Write-Write (WW)" conflict, with the committed transaction ordered before the other. Lastly, if a running transaction reads from the private version of another, then no matter which one commits, it forms a "Write-Read (WR)" conflict, with the writer ordered before the reader. The last case, however, is mainly a theoretical possibility since reading from another transaction's private version would violate atomicity. Transactional reads do not conflict with each other in any case, and they can proceed in arbitrary order.

HTM's conflict handling protocol can be piggybacked on the cache coherence protocol. In this case, the coherence protocol also tracks the *read set* and *write set* of transactions using existing sharer information in the last level of the cache hierarchy. On single-version HTM

[5] Conflict handling in larger or smaller granularity than a block is also possible. The same rules described in this section also apply in those scenarios.

designs, conflicts are detected when a read or write set block receives a downgrade or invalidation coherence message. The conflict is then resolved *eagerly* on the spot by aborting or stalling one of the two conflicting transactions. By contrast, on multi-version HTM designs, conflicts can be detected *lazily*. For example, in TCC [43, 97], transactional writes are only performed by issuing coherence writes at the end of the transaction and acquiring exclusive ownership for every block in its *write set*. This *commit-time validation* procedure invalidates any conflicting readers and writers, preventing both RW and WW conflicts.

HTM designs may also adopt *both* eager and lazy conflict handling. EazyHTM [243] registers transactional write normally and relies on coherence messages to detect conflict eagerly, just like in a single-version design. However, conflict resolution is postponed to a later stage because conflicts can often resolve themselves if no cyclic dependency forms eventually. ForgiveTM [199] follows a similar approach, but it only postpones conflict resolution for a dynamically trained subset of blocks.

Dependency tracking HTMs detect conflicts using the coherence protocol, but instead of resolving conflicts eagerly, in these designs, a dependency graph is built collaboratively by all participating cores [75, 209, 212]. Conflicts are then resolved by attempting to commit transactions following the dependency graph. These designs, however, may allow non-serializable execution due to improper handling of RAW conflicts [63, 212].

Conflict detection can be decoupled from the coherence protocol by using signatures [41, 42, 207, 208, 234, 269] or version numbers [23, 156]. For example, Bulk [41, 42, 207, 208] performs lazy commit-time validation as TCC, but instead of acquiring exclusive ownership for all write set addresses, Bulk summarizes the write set into a signature and broadcasts the signature to all conflicting transactions. SONTM [23] version-tags the entire address space and serializes transactions by deriving the commit version numbers of a transaction from the version number of its read and write set blocks. SI-TM [156] timestamps individual transactions and version data generated by transactions. It detects conflicts by commit-time validation that verifies the transaction's write set against timestamp ordering.

Overall, HTM conflict handling enforces serialized execution by detecting and resolving conflicts. Conflict handling can occur eagerly right on the spot when conflicts happen or lazily by postponing it to a later time, or adopt a hybrid of the two. Many HTM designs piggyback conflict handling on the existing coherence protocol for design simplicity. Nevertheless, conflict handling can also be decoupled from the coherence protocol by using hardware signatures or version numbers. The latter approach introduces more design challenges, such as the implementation of the signature, the maintenance of version numbers, and parameter tuning.

### 2.2.2 NVM-Based Memory Snapshotting

Memory snapshotting captures the content of the address space at a certain time point and saves it for later usage. The snapshot, once taken, can be used for low-latency failure recovery, persistent and durable data structures [67], fine-grained system backup and replication [274], or record-and-play debugging [167, 174, 250]. This dissertation primarily focuses on *full-system* memory snapshotting that records changes over the entire physical address space for failure recovery. Other variants of snapshotting can be implemented on top of this general model with only marginal

changes. [6]

All memory snapshotting designs require some form of multiversioning. When a snapshot is created, the system should potentially maintain at least two versions for every address: one *old version* storing the memory state at the snapshot point, and one *new version* representing the memory state after the snapshot. If multiple snapshots are taken, multiple old versions will be present, and the snapshotting design is supposed to be able to retrieve the memory state of a given address in any of the snapshots.

The most straightforward approach to taking memory snapshots is first to stop processor execution and then dump the memory content, plus the processor register file, into persistent storage as a monolithic snapshot image. This monolithic snapshot can be loaded back to the memory and processor register file later, which restores the state of the system back to the snapshot point. However, in the typical case scenario where most of the memory content remains unmodified between snapshots, this straightforward approach will incur unnecessary data copying and tremendous storage overhead because the same data representing unmodified parts of memory will be replicated multiple times across the snapshot images. To overcome the problem with monolithic snapshot images, early works on memory snapshotting [11, 80, 175, 205] have proposed taking *incremental snapshots* over a time interval called an *epoch*. In incremental memory snapshotting, memory content that is unmodified during an epoch will not be captured as snapshot data belonging to the epoch. Instead, the snapshot only contains memory writes performed within the epoch, hence the term "incremental".

The main design challenge of epoch-based incremental memory snapshotting is efficiently capturing memory modifications during an epoch. Prior works on this subject proposed three different approaches to overcome the design challenge: undo logging, redo logging, and shadow mapping. The first two rely on saving version data in a persistent log such that *both* old and new versions exist in the system at any time during execution. More specifically, with undo logging, the old version of a memory location to be modified is saved in an undo log entry, which is then flushed into persistent storage. The undo log entry will be replayed during failure recovery, which reverts the content of the memory location to the old version stored in the entry. With redo logging, all memory writes during the epoch are redirected to persistent redo log entries as new versions, while the memory content remains unmodified, representing the old version. The redo log is replayed on failure recovery and the completion of the current epoch. The replay process reapplies memory writes contained in the redo log entries to their corresponding memory locations, which brings the system to the most up-to-date state. As we will see shortly, both logging approaches require some form of *write ordering* to be enforced between log writes and dirty data writes for correct failure recovery.

Shadow mapping, on the other hand, redirects write operations on the same memory address to different storage locations, whenever the writes are performed within different epochs. The memory snapshot of an epoch $E$ is hence collectively formed by all memory writes captured in epoch $E$ itself, plus all epochs earlier than $E$. The redirection of write is performed by a mapping table translating the canonical address of the write to the actual location where the written data will be stored. To avoid the mapping table from being corrupted by failures, the table itself must

[6] For complete recovery, certain I/O and external events (e.g., network data from `recv` syscall) may also need to be recorded at the system level. See, e.g., [167, 236, 250] for more details on this matter.

also be continuously snapshotted and saved to persistent storage at the end of every epoch. This last property makes shadow mapping a design that recursively relies on itself because shadow mapping essentially turns a more significant problem (snapshotting the entire address space) into a more minor but similar problem (snapshotting the mapping table).

With the introduction of Byte-Addressable Non-Volatile Memory (NVM) [7], hardware designers face a new set of trade-offs that have motivated reconsidering the design space for memory snapshotting. More specifically, given the unique programming model and the performance characteristics of NVM, we identify that an ideal NVM-based memory snapshotting design should possess the following three properties. First, the design should minimize stalls and keep the processor as busy as possible. In particular, special software primitives such as *persist barriers* are unacceptable because they frequently stall processor execution. Second, the design should also minimize *write amplification*, an inevitable phenomenon where the amount of data written into the NVM is more than the size of snapshot data. Lastly, the design should scale to large systems and working sets. Hardware proposals that rely on centralized control or size-limited hardware resources may find themselves struggling to keep pace with future hardware development. We elaborate on these three properties in the rest of this section.

**Minimizing processor stall.** Processor stall is a result of executing persist barriers, a software primitive for making write data persistent. In the current cache hierarchy, dirty data generated by writes will remain volatile in the hierarchy for a potentially long time and will only become persisted when it is evicted back to the NVM. In order to eagerly persist the modification made by a write, software needs to explicitly flush the address to which the write applies and then issue a store fence to wait for the acknowledgment from the NVM device indicating that the store has been persisted. This combination, i.e., a cache flush (e.g., `clflush` or `clflushopt` on x86 [109]) followed by a store fence (e.g., `sfence` on x86 [109]), is given the name "persist barrier", and as with all software constructs that contain memory fences, the persist barrier will stall the processor and cause performance degradation [123, 137, 138, 230].

Despite the adverse performance impact, persist barrier is crucial for enforcing write ordering that is essential for correctness in specific designs. For example, in undo logging, the persistence of the undo log entry (i.e., the old version) must precede the persistence of the corresponding dirty block (i.e., the new version). This write ordering can be implemented in software by generating the log entry first, executing a persist barrier, and then performing the write. Failure to observe the write ordering will potentially corrupt the snapshot image and prevent system recovery in case of failures. For example, if the system crashes after dirty data on address X is persisted, but before the log entry is, then failure recovery would be unable to revert the memory state of address X to the pre-crash state in the most recent snapshot. In this case, the snapshot memory state is included in the log entry as the old version, which is permanently lost.

Due to the lack of more efficient primitives to enforce write ordering on current commercial hardware, the less efficient persist barrier is prevalent in software-implemented NVM applications. Examples include transactional libraries [40, 44, 55, 82, 86, 90, 98, 102, 103, 111, 118, 136, 140, 159, 160, 165, 170, 176, 193, 231, 256, 263], ad-hoc data structures [22, 51, 60, 105, 146, 147, 158, 184, 194, 225, 249, 251, 267, 278], memory allocators [31, 59, 179, 224], and

[7] In this dissertation, we focus on NVM devices in DIMM form factors, such as Intel Optane persistent memory. Unfortunately, Intel has decided to gradually wind out its Optane storage line as of the time of writing [56].

storage services [21, 48, 69, 124, 135, 186, 260, 261]. Among these software implementations, Romulus [55], a transactional framework that can be adopted for snapshotting, is a typical example that demonstrates how persist barriers affect performance. In Romulus, the next transaction can only start after the working set of the prior one becomes persistent. This process requires issuing cache flushes and persist barriers for the entire working set of the prior transaction, the execution of which is on the critical path. As a result, Romulus suffers significant slowdowns compared with normal execution without persist barriers, exhibiting the general inefficiency of software implementations.

On the other hand, persist barrier overhead can be fully or partially eliminated from the critical path by coordinating persistence on hardware. Prior designs have attempted this approach by adding extra persistence coordination capability to the load/store unit [87, 233], the hyper-threading scheduler [248], the coherence directory [137], the cache controller [33, 58, 65, 116, 122, 189, 254, 275], and the memory controller [7, 37, 92, 117, 121]. Prior works have also investigated software-hardware collaboration supports for decoupling persistence from execution, and proposed to delegate persistence to background hardware threads [255]. These approaches generally suffer less persistence-related overhead than software libraries due to newly added hardware components overlapping persistence with execution.

**Reducing write amplification.** Write amplification is detrimental in two aspects. First, excessive writes will consume memory bandwidth and contend resources, such as memory controller cycles, with concurrently executing processes. For memory snapshotting, this means that the overall system performance will be negatively impacted on memory-intensive applications, which makes the design only suitable for non-memory-heavy scenarios. Second, existing commercial NVM devices can only sustain a limited number of writes before it entirely wears out and becomes unusable [77]. Writing more data than necessary will shorten the lifespan of NVM devices, resulting in more frequent device wear-outs.

There are three sources of write amplification in memory snapshotting designs. The first is extra metadata that accompanies snapshot data, such as log entry header fields (for logging) and mapping table entries (for shadow mapping). This part of write amplification can be reduced with careful engineering but never eliminated. The second source of write amplification is mismatched granularity between memory writes and snapshotting. For example, page-level snapshotting incurs high write amplification if the page is only modified sparsely [80]. By contrast, fine-grained block-level or even word-level snapshotting reduces write amplification greatly by only writing the essential data. The last is protocol overhead. For example, hardware logging [7, 33, 37, 65, 92, 116, 121, 122, 137, 189, 233, 254, 275] doubles the amount of data written into the NVM, because snapshot data is written twice—first as log entries generated during execution, and then as in-cache data flushed back at epoch boundaries (for undo logging) or as data stored in log entries during the replay (for redo logging). Unfortunately, write amplification is a natural consequence of the logging protocol, which is crucial for guaranteeing correct failure recovery. As a result, all hardware logging designs suffer at least $2\times$ write amplification.

On the contrary, in hardware shadow mapping designs [190, 214, 257], snapshot data is only written once, which incurs considerably less write amplification than logging. However, these designs still suffer write amplification by having to update mapping table entries. To further reduce memory writes to the mapping table, these designs all implement some form of the mapping table in hardware on existing hardware components, such as the TLB [190] or the memory

controller [214, 257]. The hardware mapping table can be updated without modifying persistent data, thus reducing the number of NVM writes during execution.

**Scaling to both large systems and large working sets.** Prior memory snapshotting designs have suffered from several scalability challenges. First, because modern multicore processors feature non-inclusive, distributed LLC slices [27, 114, 131, 182, 252], designs with a centralized LLC tag walker [189] or control logic [275] simply will not fit. Second, most previous designs assumed a globally synchronized epoch [189, 214, 257]; achieving this consensus is challenging to scale due to increasing communication costs. Finally, previous designs tend to generate traffic bursts, especially when snapshots frequently occur, due to coordinated, simultaneous write-backs from all components. As systems scale, these bursts of traffic become increasingly likely to hurt performance by saturating the bandwidth of memory buses and NVM devices.

Memory snapshotting designs should also scale to working sets larger than today's system capacity. The reason is twofold. First, the snapshot is taken over the entire address space. As the number of cores increases, the amount of data generated can be overwhelmingly more significant than that with current multicore systems. Second, NVM devices are expected to be denser in data capacity than DRAM. Future systems using NVM as main memory will have larger in-memory working sets than today's DRAM-based systems. Consequently, designs that rely on size-limited and often already heavily contended hardware structures (e.g., the TLB [190]) would find themselves incapable of adapting to increasingly larger working sets in future systems.

## 2.3 Leveraging Runtime Application-Level Information in Cache Compression and Object Allocation

This section discusses background material and reviews related work for the second half of the thesis. We cover inter-block cache compression and object allocation for serverless computing, and we focus on how these two scenarios can benefit from leveraging runtime application-level information.

### 2.3.1 Inter-Block Cache Compression

There has been a proliferation of research on *single-block* last-level cache compression (or merely "single-block compression") [12, 13, 70, 81, 83, 93, 101, 110, 144, 188, 201, 220, 221, 222, 242, 266] and how they can improve performance, conserve memory bandwidth, and reduce energy consumption. In single-block compression, a block is compressed by the hardware before it is inserted into the cache and decompressed to restore its original content when being read out. The compression algorithm only searches for redundancy within the single block without referring to other blocks, hence the phrase "single-block" in its name.

One key limiting factor for single-block compression is its inability to leverage inter-block data redundancy. Prior works have indicated that inter-block data redundancy is a common data phenomenon due to same-type objects, real-world recurring patterns, and multi-dimensional data structures (e.g., graph adjacency arrays) [84, 196, 245]. In these scenarios, different blocks in the physical memory may have similar content, which we call *analogous blocks*. However, the

compression algorithm for single-block compression designs can only search for redundancy within a block. As a result, these designs miss the opportunity for a higher compression ratio by leveraging data redundancy across analogous blocks.

To overcome the limitation of single-block compression, more recent cache compression researches propose *inter-block* compression [19, 83, 84, 196, 242, 245, 272] to leverage inter-block redundancy more effectively. In inter-block compression designs, a block is compressed using another analogous block as the reference. To correctly restore uncompressed value, the same analogous block must also be used when the block is decompressed. This approach is particularly advantageous over single-block compression when data redundancy within a block is lacking. In this case, single-block compression designs would produce poor compression ratio due to not being able to find much redundancy within the block, but inter-block designs can still compress well by finding redundancy in the reference block.

Because analogous blocks are used as the reference for compression, the pivot of *all* inter-block compression designs is to search for analogous blocks efficiently. Exhaustively searching for analogous blocks over the address space or even just within the cache hierarchy is unrealistic due to the vast overhead of comparing block content. In search of optimizing this process, prior works have proposed adding an extra hardware *base cache* alongside the LLC that tracks a small subset of working set data. The base cache is populated by continuously selecting the most representative blocks from the working set, and is expected to contain analogous blocks for the current cache content. The base cache operates in tandem with the LLC to provide analogous blocks during compression and decompression. Analogous block searches are performed only on the base cache with substantially reduced work.

The base cache is commonly seen in inter-block compression designs and can take different forms. For example, Dedup [242] and 2DCC [83] both perform inter-block compression between blocks with exact same content (also known as "cache deduplication"). The hardware compressor maintains a hash table that tracks in-cache blocks and their hash values as the base cache. When a new block is to be inserted, deduplication is performed by first hashing the new block and then using the hash value to fetch the analogous reference block from the hash table. A final comparison is made to check that the block content between the new and the reference block indeed matches. Deduplication succeeds if the check passes.

Thesaurus [84] follows a similar architecture, but instead of only deduplicating between identical blocks, Thesaurus is also capable of compressing between blocks that are *mostly* identical. To this end, Thesaurus adopts a novel *fingerprint hashing* that maps blocks of similar content to the same hash value with a high probability—a property called *clustering*. Compression is performed by taking the bit-wise delta between the new block and the analogous reference block fetched using the hash value from the base cache. Thanks to the clustering property of fingerprint hashing, the bit-wise delta is expected to be minor, thus achieving size reduction.

SC2 [19] trains a hardware dictionary that contains frequent words in the working set, and performs Huffman encoding [104] using the dictionary. Although SC2's dictionary is maintained at word level rather than block level, it essentially functions as a base cache which enables inter-block compression in the same manner as a block-level base cache.

Zippads [245] enables object-level compression by building upon Hotpads [244], which is an object-oriented cache hierarchy. Zippads assumes that objects of the same type are likely analogous due to in-memory data layouts and their semantic relevance. Based on this assumption,

Zippads delta-compresses between cached objects of the same type ID. For each type ID, Zippads maintains a base object as the reference for compression in a base cache, which is added to the baseline cache design [244] to allow fast access to these reference objects.

All the above inter-block compression designs suffer the same issues incurred by the base cache. First, the base cache is a fixed-size hardware structure and is not easily scalable to large working sets. This issue is particularly prominent on a multicore system running a diverse set of workloads. In this scenario, each workload could potentially have its own set of analogous blocks, which can quickly overwhelm the base cache, causing cache contention. Second, the base cache is also hard to scale to large systems with partitioned LLCs because each LLC slice will have to maintain its local base cache for fast access to analogous reference blocks. Finally, the base cache is a complex piece of hardware, which occupies precious real estate, consumes extra power and may increase memory bandwidth (if it misses frequently). All these effects can potentially negate the benefits of cache compression.

On the other hand, DISH [196, 272] abandons the base cache and only performs dictionary inter-block compression across every four consecutive blocks on the physical address space. DISH assumes that consecutive blocks on the address space are likely analogous, and when they are indeed analogous, the dictionary algorithm can exploit inter-block redundancy by extracting common values that frequently appear in these blocks. Unfortunately, this assumption on block analogousness does not always hold, and as a result, DISH performs badly when consecutive blocks do not share many common values.

**Leveraging analogous blocks on the same page with software-provided information.** Prior works on inter-block compression have either gone too far by assuming that analogous blocks can exist anywhere on the address space and therefore need a dedicated base cache for grouping them, or oversimplified the matter by assuming that analogous blocks must be adjacent. As a result, these designs either require complicated hardware additions or fail to accomplish a high compression ratio when the oversimplified assumption does not hold. In reality, the distribution of analogous blocks is between the two extremes. On the one hand, they are often not far away from each other and are likely on the same page, as a result of arrays-of-structs, or memory allocators placing same-type objects next to each other [3, 30, 72, 85]. On the other hand, analogous blocks may not always be adjacent; instead, they can form a *stepped* (spatially strided) pattern. This scenario describes many common cases in data applications, e.g., large objects such as database table rows or index data structure nodes being allocated from a buffer pool by a buffer manager. Once the stepped pattern is known, analogous blocks on the page can be easily identified using a single "step size" attribute that specifies the gap size between two adjacent analogous blocks. Unfortunately, the "step size" of a page is a runtime property, which must be obtained dynamically. Besides, different pages can exhibit different patterns, even when the pages are from the same application. We thus identify the challenges of leveraging runtime stepped pattern as (i) dynamically passing the pattern to hardware and expressing the "step size" attribute as a per-page property, and (ii) leveraging the "step size" attribute in hardware to compress data more effectively.

In Chapter 5 of this dissertation, we present a novel inter-block compression design, Multi-Block Cache Compression (MBC), which compresses across analogous blocks that form a stepped pattern on the same page. MBC addresses the first challenge by enabling application software to pass the runtime "step size" attribute via system call interfaces. The attribute will then be

stored in the system page table and fetched into the TLB during page walk. To address the second challenge, MBC skews the cache controller's index generation function to take the per-page "step size" attribute (which is included in memory access requests) as input. With the runtime "step size" attribute, analogous blocks on the same page will be mapped by the index generation function to the same set, and further grouped into the same cache tag, on which inter-block compression is performed. Compared with prior works, MBC is flexible enough to adapt to various use cases while requiring substantially simpler hardware. We attribute this advantage to MBC's ability to leverage runtime information from the application.

### 2.3.2 Object Allocation for Serverless Computing

Serverless computing has become an increasingly popular cloud paradigm with services like AWS Lambda [14], Azure Functions [171], Google Cloud Functions [89], and IBM Cloud Functions [106]. Built on top of serverless computing, *Functions-as-a-Service* (FaaS) enables a multitude of applications by decomposing them into small code snippets, called *functions*, that are invoked by user-specified trigger events [76, 119, 120, 200, 216]. The benefits of this approach include enabling cloud providers to automatically provision resources while charging users for only their actual resource consumption at a millisecond granularity [15]. Notably, these functions are very short-lived, typically finishing in less than a second [229], which poses new challenges in system design.

Memory management exemplifies the *ephemeral* nature and challenges of short-lived function execution. Specifically, existing OS and hardware layers remain largely oblivious to the short-lived serverless functions. As a result, functions pay the full critical-path costs of memory allocation and deallocation in both user space and the OS without the opportunity to amortize these costs over their short lifetimes. For example, consider the overheads of typical memory management operations. For starters, applications must pay the cost of memory management in user space where each allocation and free typically requires tens of instructions in popular high-level languages for serverless environments (e.g., Python). Furthermore, user space allocators rely on system calls such as `mmap` to request memory from the OS, requiring thousands of instructions in addition to polluting hardware resources due to context switches. However, due to the lazy-by-design nature of modern operating systems, the kernel only reserves a virtual region without physical memory backing (for the sake of hopefully avoiding wasted memory). Upon the first access to a virtual page, a page fault is triggered, forcing the kernel to perform a physical allocation, requiring additional thousands of instructions.

Kanev et al. [125] has highlighted the high percentage of cycles spent on memory management in data center (Warehouse-Scale Computing, WSC) workloads. Unfortunately, while long-running workloads can better amortize their memory management costs over their long lifetimes, this is not the case for short-lived functions. Solutions that work perfectly well for data center workloads may see only little impact for serverless computing.

Prior works on memory allocators have proposed accelerating memory management with specialized hardware. For example, early research on hardware buddy allocators [38, 45, 46] liberates the OS from physical page management. They are, however, primarily designed for small and embedded systems, and they are a mismatch for today's cloud server with tens to hundreds of GBs of physical memory. Li et al. conducted a feasibility study of removing the user space heap

allocator from the critical path but did not present any concrete design to correspond [154]. On a related work [153], Li et al. described a hardware memory allocator. However, the proposed design is overly specific to a particular architecture and no longer functions on modern platforms.

In a more recent proposal, Mallacc [126] identifies that two of the most expensive operations on `tcmalloc`'s critical path are size class computation and linked list insertion/deletion. Mallacc suggests replacing the two expensive operations with the proposed hardware primitives, and then handled the primitives efficiently by specialized hardware in the cache hierarchy. While the Mallacc design improves performance end-to-end, it still leaves a large chunk of work on memory management to be implemented on software. As a result, Mallacc falls short of the ideal speedup that can potentially be achieved by fully eliminating memory management from the critical path.

**Accelerating memory management using high-level primitives.** In Chapter 6 of this dissertation, we present Memento, a holistic approach to memory management that offloads *all* works on the critical path to the cache hierarchy. Memento replaces memory management routines, such as `malloc`, `free`, `mmap`, and `munmap`, in their entirety, with high-level primitives. By leveraging application-level memory management primitives, Memento reduces the latency of memory management operations, i.e., `malloc` and `free`, to only a couple of cycles, which is comparable to single cache access. The majority of the work, including those performed by OS components, is conducted in the background by the cache hierarchy and overlapped with execution. Compared with Mallacc, Memento accomplishes an almost ideal speedup by eliminating memory management from the critical path, demonstrating the benefits of leveraging high-level primitives.

# Chapter 3

# OverlayTM: Enabling Faster Serializable Hardware Transactional Memory

This chapter details OverlayTM, a Hardware Transactional Memory (HTM) design built upon Page Overlays. OverlayTM poses a low-cost solution to implement fine-grained transactional synchronization, where code snippets accessing shared memory are manually marked as transactions, and are dispatched to multiple threads for concurrent execution. For example, in an application where concurrent threads read from and insert into a linked list, the read and insert operations both need to access the pointers to the next list node in shared memory, and each operation can be executed as a transaction. In a regular system, concurrent transactions accessing shared memory must synchronize with each other using software synchronization primitives, such as spin locks, because otherwise, unsynchronized writes may corrupt shared memory, and unsynchronized reads may access inconsistent states. The software synchronization primitives, however, degrade the overall performance of the system because they (i) disallow concurrent accesses to the same address, especially concurrent reads, hence lowering the overall parallelism, and (ii) blocks threads from acquiring an already acquired lock by spin-waiting, which wastes processor cycles.

With OverlayTM, hardware will guarantee the atomicity and serializability of transactions in a multi-threaded environment. To external observers, this would appear that concurrent transactions are executed serially in *some* order as if they were synchronized by a single global lock. The benefits brought by OverlayTM are twofold. First, transactions can execute with higher degrees of parallelism and fewer processor cycles wasted on spin-waiting, especially when contention level is light to moderate, improving overall system performance. Second, the strong serialization guarantee liberates developers from writing and testing ad-hoc multi-threaded synchronization, which is notoriously tricky to make right. OverlayTM enables fast prototyping of multi-threaded algorithms and data structures, resulting in increased productivity.

As Section 2.2.1 points out, the two major challenges of designing HTMs are *version management* and *conflict handling*. OverlayTM addresses the first challenge with *Multiversioned Page Overlays (MPO)*, which extends Page Overlays to support transactional memory operations. With MPO, when a transaction attempts to write an address for the first time, the write will exert the "Overlay-on-Write" semantics, and create a new overlay cache block in the L1 cache representing the transaction's private version on that address. These overlay blocks, unlike the "invisible

versions" in prior works, are handled as regular blocks by the coherence protocol. Evicted overlay blocks from the L2 cache will not force the transaction to abort because overlay blocks from different transactions can co-exist in the hierarchy at the shared level. As a result, OverlayTM can commit much larger transactions than previous works.

The MPO executes transactional reads by redirecting the read operation to the appropriate version, even if multiple versions exist on the same address. A transaction can hence always access a consistent memory snapshot generated by previously committed transactions during its execution, regardless of concurrent transactional writes committing more versions on its read set. This *snapshot read semantics* not only benefits read-heavy transactions but also simplifies conflict detection because a transaction is guaranteed to access consistent data throughout its lifetime. When multiple versions are committed on the same address, the MPO keeps track of the owner of each version in a *Version Directory*. Transactional reads can access the correct version via the *Version Coherence Protocol*, which is just minor addition to regular cache coherence with the notion of versions.

OverlayTM handles conflicts lazily via commit-time validation. The validation process checks the read and write sets of the committing transaction against both previously committed transactions and concurrently executing ones. Compared with previous HTM designs with commit-time validation, such as TCC [43, 97], the commit process of OverlayTM neither lock the coherence directory, nor monopolize the bus, hence scaling better to large systems. Compared with the signature-based validation technique used in Bulk [41, 42, 207, 208], OverlayTM does not require any broadcasting medium for the bulky signature, which consumes less bus bandwidth.

OverlayTM also supports *read-only optimization* and *early release*, which are optimizations that can reduce transaction abort rates in special cases. Thanks to MPO's snapshot read semantics, transactional reads will always land on the versions that constitute a consistent memory snapshot. Read-only transactions can be excluded from conflict detection and always commit successfully. Even for transactions that are partially read-only, i.e., part of the read set can be excluded from conflict detection without harming correctness, the snapshot read semantics still prove to be valuable by treating this subset of reads differently, as we will see in later sections.

## 3.1 Design Overview

This section reviews high-level designs of OverlayTM. We start with the software interface exposed by OverlayTM, and then cover the timestamp ordered transactional model and OverlayTM's extension to Page Overlays. Detailed discussions of hardware designs that implement the model are presented in later sections.

### 3.1.1 Software Interface

OverlayTM adds three new instructions to the ISA for software to control transaction execution:

**`XBEGIN`.** This instruction starts a new transaction. It carries the address of the fall-back path as its only operand. On starting a new transaction, the processor drains its store buffer and the pipeline, and then takes a snapshot of the register file. Every read and write instruction afterward

is executed as transactional read and write, respectively. The fall-back path address is also saved to a special-purpose register for later use.

**XEND.** This instruction attempts to commit a transaction (and raises a fault if not executed in a transaction). The processor first drains the store buffer and the pipeline, and then performs commit-time validation. If validation succeeds, the transaction commits successfully, and the processor discards the register snapshot and the fall-back address, after which it continues executing instructions after `XEND`. Otherwise, the transaction aborts, and `XEND` functions as a branch instruction to the fall-back address. The register snapshot is also restored to the processor context. Overall, transaction abort behaves as though the execution directly branched from `XBEGIN` to the fall-back path. The hardware may also return an abort code to the fall-back path to facilitate diagnosing the cause of the abort. Programmers may implement retry logic in the fall-back path to restart the transaction.

**XABORT.** This instruction unconditionally aborts the current transaction. The same action on the abort path of `XEND` is taken.

OverlayTM supports the Speculative Lock Elision (SLE) [8, 9] paradigm to gracefully allow the cooperation between lock-based synchronization and HTM. In SLE, transactions are first executed by HTM, and if the transaction fails to commit after several retries, it falls back to using locks for serialization. SLE maintains the invariant that at any given time, either many threads are executing transactionally without holding the lock, or only one thread can execute non-transactionally with the lock being held.

To work with SLE, each shared memory region that needs synchronization has an associated spin lock. When a new transaction begins, it first transactionally reads the lock variable and checks its value. If the lock is already acquired, indicating that another transaction is holding the lock, the current transaction aborts using `XABORT`, and retries after some back-off. Otherwise, the transaction proceeds normally. When a transaction aborts and decides to fall back to using the lock, it simply acquires the spin lock, aborting all concurrent transactions, and then the thread executes non-transactionally till the end when it releases the lock.

In OverlayTM, the hardware is designed such that if a transaction has read a block (e.g., the one containing the lock variable), then non-transactional writes on the block (e.g., other threads acquiring the lock) will *immediately* abort the transaction. OverlayTM achieves this by simply registering the transactional reads in the coherence directory as usual and then monitoring incoming coherence messages for regular write invalidations. The transaction is aborted if write permission is requested for a transactionally read block.

### 3.1.2 Timestamp Ordered Transaction Model

Transactions in OverlayTM are ordered by logical *timestamps*. Each transaction possesses a *Commit Timestamp (cts)* indicating the logical time when the transaction commits. When a transaction completes execution, it acquires the cts from a global timestamp dispenser and validates itself using the cts. The global timestamp dispenser serves as a serialization point to all transactions in the system. When a transaction acquires its cts, the transaction essentially serializes itself after all transactions that have successfully committed with a smaller cts value, and before all transactions that have not yet acquired the cts.

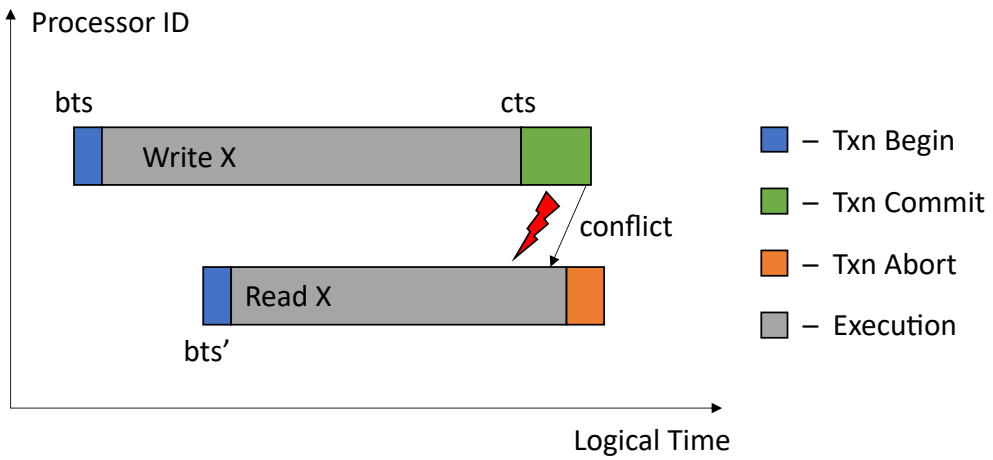



Figure 3.1: **Timestamp Ordered Transactions** – This figure shows two transactions running.

Transactions also acquire a *Begin Timestamp (bts)* from the same global timestamp dispenser when it executes the `XBEGIN` instruction. The transaction then uses the bts to select the version of data it accesses throughout the execution. Intuitively, versions committed with a smaller cts than the bts are accessible to the transaction, and those with a larger cts are invisible. These two rules constitute the *Version Access Rule*, which we cover in detail in Section 3.2.2.

In order to enforce timestamp ordering, OverlayTM validates conflicts between transactions formed by transactional reads and writes, such that these conflicts are consistent with the timestamp ordering. In particular, in OverlayTM, it is illegal for a running transaction to commit, if another transaction with a higher cts than its bts already commits newer versions on the running transaction's read set. The ordering of memory operations contradicts the timestamp ordering in this case. On the one hand, the running transaction is ordered after the committed transaction because the committed transaction acquires its cts earlier. On the other hand, the running transaction forms RW conflict with the committed transaction, and the running transaction is logically ordered before the committed transaction due to having a smaller bts. OverlayTM resolves the conflict by aborting the running transaction to minimize wasted processor cycles.

The above example is also representative of OverlayTM's conflict detection protocol. From a high level, OverlayTM maintains a read set for every transaction and validates a transaction by checking whether its read set has any newer version being committed. If a transaction's read set remains intact during execution, then the transaction can successfully commit. Otherwise, the transaction fails validation and must abort. We cover the full details of conflict detection in Section 3.3.

Fig. 3.1 depicts an example of two timestamp ordered transactions. OverlayTM transactions begin their lifecycle by acquiring a bts, and serialize by acquiring a cts, as shown in the top transaction. Conflicts are detected if one transaction commits on an address while another concurrent transaction reads the address. This is illustrated between the top and the bottom transaction that writes and reads address X, respectively. In this example, the bottom transaction is forced to abort by the commit of a new version on X.

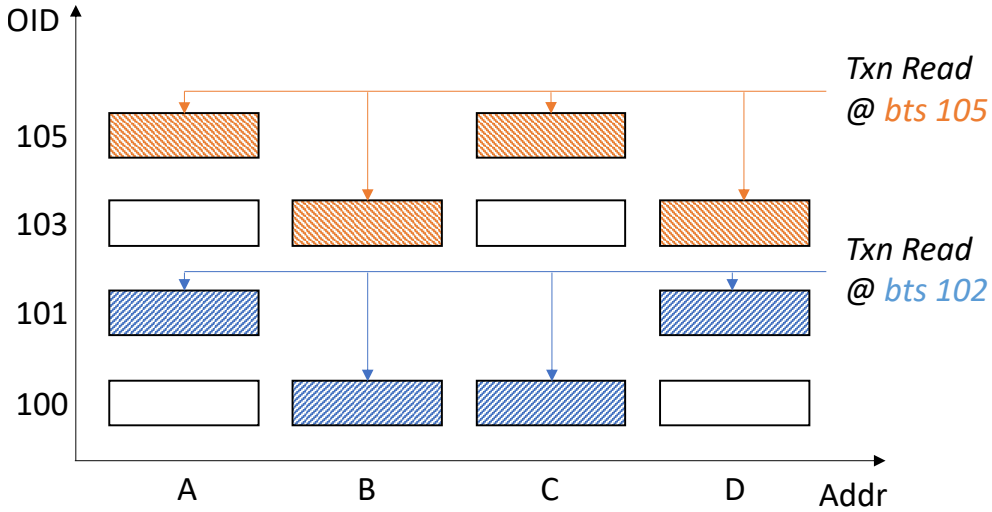



Figure 3.2: **OverlayTM's Snapshot Read Semantics** – This picture illustrates two transactions concurrently reading four versions on the address space. Blocks represent versions that exist in the cache hierarchy. Colored blocks represent the read snapshot that the corresponding reading transaction (marked by the same color) can access.

### 3.1.3 Extending Page Overlays to Support Transactions

As discussed in Section 2.1, Page Overlays enables the cache hierarchy to create new cache blocks using its "Overlay-on-Write" semantics. The newly created block is tagged with the PID of the process that performs the overlay write. In this chapter, we propose Multiversioned Page Overlays (MPO) as an extension to the existing Page Overlays design. Compared with the original Page Overlays, MPO highlights the following changes. First, transactional writes are executed as overlay writes implicitly if the thread is executing a transaction. These transactional writes, if performed for the first time on an address within a transaction, uses the bts value from `XBEGIN` instruction as the OID field. Versions created by transactional writes remain invisible to other transactions by not registering them in the Version Directory, a hardware directory similar to the coherence directory but tracks the owner of versions when multiples of them exist on the same address. Note that this convention does *not* contradict the fact that these versions can still be handled properly by the LLC when they are written back from the private hierarchy—the LLC still treats them as regular cache blocks, and it is just that the existence of these uncommitted versions is kept invisible to other transactions.

Second, transactional reads are executed as overlay reads in the private hierarchy if the OBitVector of the corresponding block is set in the TLB. The OBitVector, in this case, serves as a read set tracker for the private hierarchy. If the corresponding bit in the OBitVector is not set, the read is directly sent to the LLC with the bts of the transaction.

When the LLC handles the request, if only a single version exists on the requested address, then the LLC will return the only version just like a regular read. However, in the case where multiple versions exist, the transactional read should only be able to access the version that constitutes the consistent memory snapshot at logical time bts. The LLC hence performs a version lookup in the Version Directory that tracks all versions on a recently referred set of addresses, and selects the largest version whose OID $\leq$ the requesting transaction's bts. This version selection process implements the Version Access Rule, which is critical for providing

the snapshot read semantics. Fig. 3.2 presents an example of snapshot read semantics. In this example, two transactions, with bts being 102 and 105, respectively, attempt to read four data items on address A, B, C, and D. Multiple versions (represented by blocks) exist on each of the four addresses, and the transactional reads carried out by different transactions are selected based on the transaction's bts. Both transactions can individually read from their own memory snapshot, which is indicated by colored blocks.

Finally, when a transaction successfully commits, all its modifications must be registered to the Version Directory for future access. We propose adding a write-back phase after transaction commit, which registers all versions created by the transaction to the Version Directory. In the write-back phase, the MPO hardware iterates over the write set (which is maintained in a write log as many prior works propose, e.g., [23, 34, 97, 156, 177]), and for each version created by the transaction, registers the current core as an owner of the version to the Version Directory. Future transactional reads can therefore access the version via the Version Coherence Protocol if the Version Access Rule permits. The write-back phase also invalidates copies of other versions on the same address in all private hierarchies of the system. This step is crucial to prevent the address aliasing problem in the private hierarchy, which is not well handled in prior work [156] (we discuss the address aliasing issue in Section 3.7.1).

## 3.2 Multiversioned Page Overlays

This section focuses on OverlayTM's transactional multiversioning support, Multiversioned Page Overlays (MPO). The MPO extends Page Overlays by adding transactional read and write support. To support transactional writes, the MPO leverages the Overlay-on-Write semantics of Page Overlays. To support transactional reads, the MPO adds a Version Directory that enables snapshot read semantics via the Version Access Rule and Version Coherence Protocol. We discuss each of the hardware components in full detail in this section.

### 3.2.1 Transactional Memory Operations in the Private Hierarchy

On transactional writes, if the dirty version does not exist in the L1, the transactional write will first be executed as a transactional read to bring the cache block into the L1 cache. Then an overlay write is performed on the block. There are two cases for the overlay write. First, if the block is dirty, and is from an earlier committed version, then a block duplication is made, and the write is applied to the duplicated block (i.e., Overlay-on-Write). Otherwise, if the block is a clean copy of a version or if the transaction itself generated the block, the write is performed directly on the block. In either case, the block's OID is set to the writing transaction's bts, and the write address is inserted into a per-transaction write log (which can be implemented as an on-chip buffer that overflows to transaction-local memory).

Transactional reads are handled by first checking the OBitVector of the TLB entry in parallel with address translation. If the corresponding bit is set in the OBitVector, then the transactional read is executed as a regular read and is guaranteed to hit the private hierarchy. Otherwise, the read request is sent directly to the LLC together with the transaction's bts. After receiving the response from the LLC, the block is inserted into the private hierarchy, *without* setting the OID

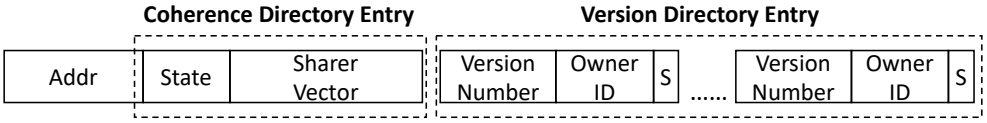



Figure 3.3: **Version Directory Entry** – Each Version Directory entry has four version slots, and each version slot has three fields: the Version Number, Owner ID, and S bit. The Version Directory entry extends the existing coherence directory entry, such that both entries can be retrieved together on directory lookups.

field. Future transactional reads (not necessarily from the current transaction) can hence hit the non-overlay block in the private hierarchy. This technique allows different transactions to share their read snapshots when multiple read snapshots refer to the same block. The corresponding bit is also set in the OBitVector to indicate that the block is now present in the private hierarchy.

To keep the OBitVector and the actual content of the private hierarchy consistent, when an entry is evicted from the private TLB, the OBitVector is lost, causing all blocks on that page to be invalidated or written back from the private hierarchy. This process, however, can be problematic given that on today's cache hierarchy, the private TLB usually does not cover as many addresses as the private cache hierarchy. To prevent the TLB's coverage from becoming a performance bottleneck, MPO may also choose not to rely on OBitVector as a read set tracker. In this case, transactional reads will ignore OBitVector, and always be executed as regular reads in the private hierarchy. When the transactional read misses the private hierarchy, it will be sent to the LLC together with the bts, and the rest remains unchanged.

The MPO also extends every block in the private hierarchy with a "TX" bit indicating whether the block has been accessed transactionally. Note that OverlayTM does need separate bits for marking transactional reads and writes, since these two cases can be distinguished from each other by the existing dirty bit. The "TX" bit in the private hierarchy is flash-cleared when a transaction commits or aborts. Flash-clearing a bit can be implemented in SRAM as a fast bulk operation [180].

One complexity of potentially allowing multiple versions on the same address to co-exist in the private hierarchy is to track the exact number of versions per address in the private hierarchy. The existing coherence directory uses a sharer vector that only allocates one bit per private hierarchy, which cannot track more than one version. To remain compatible with this scheme, the MPO enforces the invariant that only a single version on the same address may exist in the private hierarchy. If a different version is fetched into the private hierarchy, the existing version must be evicted, if any.

### 3.2.2 Handling Transactional Reads in the LLC

**Version Access Rule.** Transactional reads on the LLC must select the appropriate version to access if multiple versions exist on the requested address. OverlayTM addresses this challenge by extending every coherence directory entry with extra metadata fields tracking all committed versions on the address and the owners of these versions. These extended metadata fields are collectively called the Version Directory. The transactional read operation is hence satisfied by selecting the largest version that $\leq$ the bts in the read request. This version selection logic is called the Version Access Rule, and it enables snapshot read semantics by only allowing a transaction to access the memory snapshot generated by all transactions committed before the logical time bts.

**Version Directory.** The extended metadata fields in every coherence directory entry consist of an array of four *version slots*, with each slot tracking a committed version. We chose the maximum number of committed versions per address to be four in order to achieve a balance between parallelism and metadata overhead. Each version slot stores three fields:

- **Version Number:** The OID of the version block.
- **Owner ID:** The current owner of the version block.
- **Speculative bit (S bit):** Whether the version belongs to an uncommitted transaction. If set, the version is excluded from the Version Access Rule for all transactions except the one that creates it.

Fig. 3.3 shows a Version Directory entry and how it extends the regular coherence directory entry. On a 16-core system, five bits are needed for Owner ID field to represent all 16 cores plus the LLC as the version owner. The Version Number has the same width as OID, which needs 16 bits. The total number of additional bits per coherence entry is 85 bits, or 15.1% of total LLC capacity. The storage overhead can be further reduced, given that version numbers likely do not need the full 16 bits in the directory if an address is committed on frequently. In this case, only the lower 4 bits are stored for each slot, and all slots share the higher 12 bits of the OID. This trick reduces metadata overhead to 8.57% of total LLC capacity.

The S bit may seem to contradict the earlier notion that only committed versions are tracked. However, when a version is prematurely evicted from the private hierarchy before the creating transaction commits, the version must also be tracked by the Version Directory. In this case, the version is "pre-committed", but the visibility is limited to its creator only. The Version Access Rule must hence exclude versions tracked by the Version Directory if the S bit is set unless the reading transaction is the creator of the version.

Note that the Version Directory only tracks committed versions. Uncommitted versions created by transactional writes are not tracked by the Version Directory when they stay in the private hierarchy, and hence the number of uncommitted versions on the same address can be significantly larger. To minimize the number of committed versions to track, the MPO also aggressively garbage collects versions (see Section 3.2.4), such that versions that are no longer needed will be quickly removed.

**Version Coherence Protocol.** With the Version Access Rule and Version Directory, transactional reads are handled as follows. First, the LLC controller performs a directory lookup using the requested physical address. The regular coherence states and the version directory entry are fetched in this stage. Then the LLC controller selects the version to read according to the Version Access Rule. In the last step, if the version directory indicates that the a private hierarchy owns the version, then the version is acquired via regular cache coherence, and a clean exclusive copy (in E state) is granted to the requestor. If the LLC itself owns the version, then the LLC will directly respond after fetching block data from its data array without any extra coherence action. In either case, the regular coherence bit is also set to indicate that the requesting core has acquired a cached copy of a version. Tracking version readers using regular coherence bits is crucial for commit-time validation, as we will see in later sections.

The version read process presented above is called the Version Coherence Protocol, and it simply leverages the existing cache coherence mechanisms to deliver versions from the version owner to the requestor. Compared with prior proposals that add new coherence states for tracking

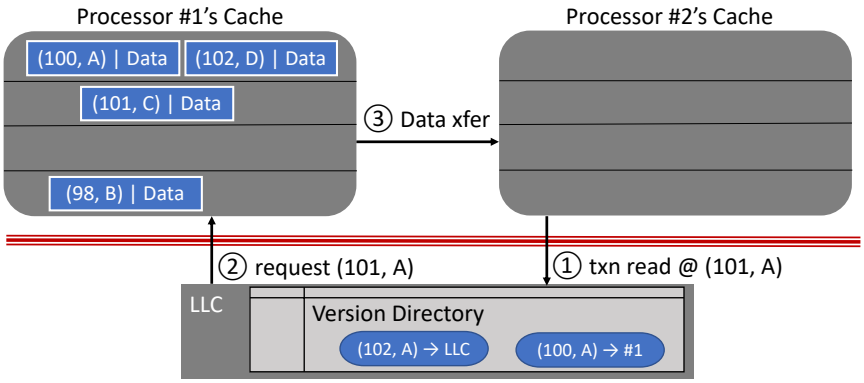



Figure 3.4: **Version Coherence Protocol** – We assume that the coherence protocol supports cache-to-cache data transfer. Only directory entry for address A is shown in the Version Directory.

versions [75, 212], OverlayTM requires substantially less changes to the existing protocol, and as a result, it also takes fewer design and verification efforts.

**Example.** Fig. 3.4 shows an example of version coherence protocol. In this example, processor #2 runs a transaction (bts=101), which issues a transactional read to address A. The transactional read request misses the private hierarchy and is forwarded to the LLC (step ①). On receiving the request, The LLC performs a Version Directory lookup using the requested address A, and figures that there are two committed versions on address A in the system, of version 100 and 102, respectively (bottom part of the figure). The Owner ID field shows that version 100 is currently in the private hierarchy of processor #1, and version 102 is currently in the LLC. The LLC then exerts the Version Access Rule and selects version 100 as the read snapshot version. A coherence request is sent to the owner of version 100, i.e., process #1, to request a read copy of the version it holds (step ②). On receiving the coherence request, the hierarchy of processor #1 replies with version data directly to the requestor via cache-to-cache transfer, fulfilling the transactional read request.

### 3.2.3 Version and Version Directory Evictions

Unlike prior HTM designs where versions must be retained in the private hierarchy, OverlayTM handles evicted versions just like regular cache blocks. When a committed version is evicted from the private hierarchy, the MPO simply updates the Version Owner field in the corresponding entry of the Version Directory to point to the LLC. Similarly, when a committed version is evicted from the LLC to the OMS (recall that Page Overlays remaps overlay blocks on the same address and with different OIDs to distinct locations), the Version Owner is updated to point to the OMS. In the latter case, when this version is accessed in the future, the Version Coherence Protocol will direct the request to the OMS.

When an uncommitted version is evicted, the version is "pre-committed" in the Version Directory by adding it into a vacant version slot and setting the Owner ID to point to the LLC. The S bit is also set such that the version is only accessible to the transaction that creates it. When the pre-committed version is evicted from the LLC to the OMS, the Owner ID is updated to point to the OMS.

A Version Directory entry is evicted when the corresponding coherence directory entry is

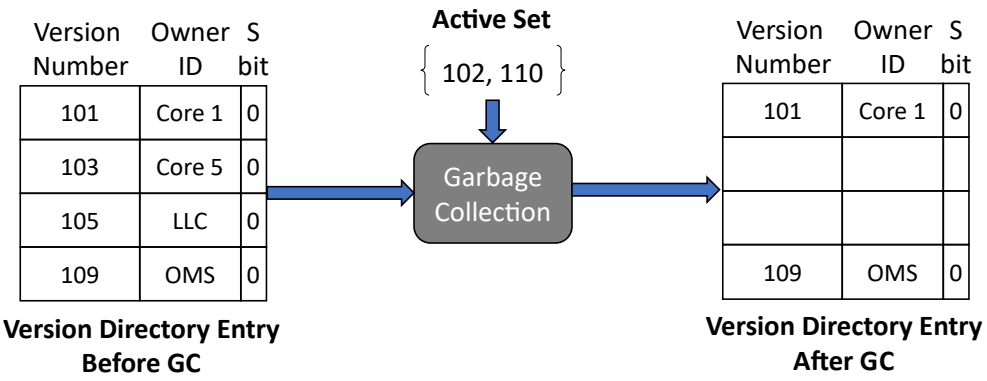



Figure 3.5: **Garbage Collection Example** – The garbage collection logic takes a Version Directory entry and the Active Set as input. Versions that are no longer accessible to existing and future transactions are deleted.

evicted. An evicted Version Directory entry, however, has more complex implications than version evictions because an evicted entry will cause all tracked versions to be lost without special care being taken. OverlayTM can handle Version Directory entry evictions in two ways. In the first and more straightforward approach, all committed versions except the top one (the one with the highest Version Number) will be discarded, and all transactions that can potentially access the discarded versions in accordance with the Version Access Rule, are force aborted. This approach, however, may abort an excessive number of transactions, and it upper-bounds transaction sizes to the maximum coverage of the Version Directory. In the second approach, the Version Directory entry is sent to the OMS, and the OMS backs the entry with physical pages. The entry is fetched back to the Version Directory when the owners of the versions it tracks change, or when a new version is to be added during a version commit.

### 3.2.4 Garbage Collection

MPO regularly performs garbage collection (GC) to remove "dead versions". A version is pronounced "dead" if it is inaccessible to any existing and future transactions. Formally speaking, if two committed versions with OID $X$ and $Y$ exist ($X < Y$), but no running transaction has bts between $X$ and $Y$ (since bts monotonically increases, this condition also holds for future transactions), then version $X$ is dead and can be removed by sending coherence invalidation. For addresses where more than two versions exist, this process is repeated several times (which can be performed in parallel by hardware), on each version. To facilitate GC, the MPO maintains a set of bts that is currently actively running in an *Active Set* implemented as an associative search hardware structure. A bts is added into the Active Set when a new transaction begins and removed from the set when the transaction commits or aborts. The Active Set contains at least as many slots as the number of processors in the system, such that it will unlikely cause resource hazard on insertion (and if resource hazard occurs, one of the existing transactions is aborted to make room for the new transaction).

Fig. 3.5 illustrates the GC process. The GC logic takes a Version Directory entry that contains four valid versions with Version Numbers 101, 103, 105, and 109 as input. The Active Set is also an input, which contains two currently active transactions, with bts 102 and 110, respectively. After the GC, version 103 and 105 are removed because these two versions are no longer accessible

to current and future transactions, as per the Version Access Rule. Version 101 and 109, on the contrary, remain after GC because version 101 can still be potentially accessed by transaction 102. Meanwhile, version 109 can be accessed by transaction 110 and future transactions.

GC can be triggered on several occasions. One possibility is to add a background LLC tag walker that periodically scans Version Directory entries and applies GC to each of them. The other (and potentially better) option is applying GC when a Version Directory is being accessed or updated. In the latter approach, GC can be piggybacked on whatever operation that caused the Version Directory access, with no extra traffic incurred on the Version Directory.

### 3.2.5 Committing Versions

When a transaction commits, it publishes all versions that have been created to the Version Directory, such that all future readers with a higher bts than its cts can access the memory snapshot generated by the transaction. The version commit process iterates over the write log, and for each write address in the log, sends a version commit message to the Version Directory. The version commit message contains the write address and the committing transaction's bts.

On receiving the version commit message, the Version Directory checks if the version to be committed already exists. If negative, then the version is committed by adding a version slot entry, with its Version Number set to the transaction's bts, and the Owner ID set to the core ID the committing transaction is currently running on. Otherwise, the version has already been pre-committed in the Version Directory due to an earlier eviction. In this case, the S bit in the version slot is cleared, unblocking future transactions from reading this version.

During version commit, if a Version Directory entry overflows, i.e., there are already four committed versions tracked by the entry, and none of them could be GC'ed, then the oldest version is forced to be removed. Consequently, all transactions that can still access that version are also forced to abort. This last step is achieved by the Version Directory sending abort messages to the victim transactions. This case, though, is extremely rare in our experiments, as most addresses only have one or two versions.

OverlayTM's version commit process only registers versions and their owners to the Version Directory. Committed version data can still be retained in the private hierarchy without being written back. This feature distinguishes OverlayTM from similar designs such as SI-TM [156], in which version data is written back on transaction commit, incurring a considerable bus bandwidth surge.

## 3.3 Detecting Conflicts with Commit-Time Validation

In OverlayTM, two transactions conflict if one commits new versions on the other's read set. OverlayTM detects conflicts with commit-time validation when a transaction completes execution, and handles conflicts by forcing the transaction whose read set is committed upon to abort. In this section, we focus on OverlayTM's conflict detection mechanism. There are two cases of conflict detection. In the first case, none of the transaction's read and write set versions is evicted from the private hierarchy. This case commonly occurs for most small to medium-sized transactions, and OverlayTM relies on the regular coherence states to detect version commits on the read set.

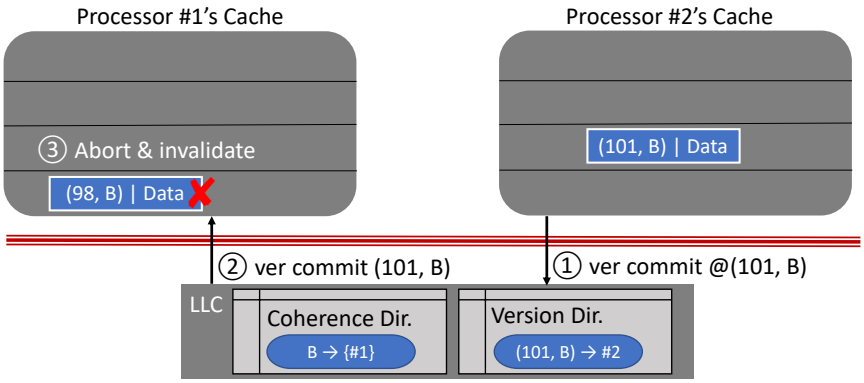



Figure 3.6: **Version Commit Causing Transaction Abort** – We assume that the block B in processor #1's cache has its "TX" bit set. The version commit message forwarded from the Version Directory forces the transaction running on processor #2 to abort because a newer version is committed by processor #2. The block in processor #1's cache is also invalidated by the version commit message.

In the second case, the coherence states could no longer track the transaction's entire read set because part of the read set had been evicted from the private hierarchy. OverlayTM handles this less common case using *both* coherence states (for read set versions that are still in the private hierarchy) and a *Commit Queue* (for evicted read set versions).

### 3.3.1 Detecting Conflicts in the Private Hierarchy

Recall from Section 3.2.1 that transactional reads and writes missing the private hierarchy will be performed as transactional reads on the LLC. This transactional read will set the bit in the regular coherence entry's sharer vector for the requestor, just like a regular read. The per-block "TX" bit will also be set when the block is inserted into the private hierarchy.

The sharer vector in the regular coherence directory approximately tracks the read set of transactions and can be used to detect conflicts. When another transaction performs a version commit, it iterates over the write log and issues version commit messages to the Version Directory. Conflict detection between the committing transaction and transactions with the same address in the read set is performed by forwarding the version commit message to all version readers of the address. Since version readers are tracked per address using the sharer vector of the regular coherence directory, this process can be implemented similarly to write-invalidation in regular coherence.

On receiving the forwarded version commit message, the private hierarchy first checks whether the address exists (in case of silent eviction [73]) and then checks the "TX" bit of the block. If the "TX" bit is set, suggesting that the transaction running on the hierarchy has the block in its read set, then the transaction is forced to abort because a newer version has just been committed on its read set. The version commit message also invalidates existing versions on the same address. This step is crucial to prevent the aliasing problem, where a newer version has been committed on the address, but newly started transactions incorrectly access a stale version cached by the private hierarchy.

Fig. 3.6 gives an example on version commit and conflict detection. In this example, processor #2 commits a version on address B with OID 101. The version commit message is sent to the Version Directory as part of the commit procedure (step ①). On receiving the message, the

Version Directory inserts `(101, B)` $\rightarrow$ `#2` into the Version Directory entry of address B, and in the meantime, it also forwards the version commit message to the sharer of address B, which is processor #1 (step ②). When the forwarded message is being handled, the private hierarchy of processor #1 finds that a read set version `(98, B)` exists (we assume that its "TX" bit is set), and the transaction currently running on processor #1 is forced to abort. In addition, the cached stale version `(98, B)` is also invalidated, such that future transactions started on processor #1 will not read the stale version (step ③).

Also, note that the version commit message may race with transactional reads on the same address. In this special case, a transaction receives the version commit message first and reads the address later. Although this case is not detected in the above protocol, it should be easily recognizable when the transactional read is handled by the Version Directory, as the newer version must have already been inserted into the Version Directory. The reading transaction is therefore aborted when a read request encounters a newer committed version on top of the version that it is supposed to access.

### 3.3.2 Detecting Conflicts for Evicted Versions

Relying on the sharer vector to detect conflicts has one limitation: if a read set version is evicted from the private hierarchy, then the private hierarchy loses track of the version and its "TX" bit. As a result, future version commits may not find the evicted version, hence missing a conflict and causing potential non-serializable execution.

In order to detect conflicts for evicted versions, OverlayTM keeps an approximate set of versions by inserting their addresses into a *evicted read set bloom filter* signature when an eviction occurs and the "TX" bit is set. The resulting signature summarizes read set versions that have been evicted at least once, with no possibility of false-negative, but may incur false-positives on presence tests. Prior works (e.g., [16, 23, 41, 211, 234, 269]) also attempted to perform conflict detection either partially or fully using signatures. OverlayTM differs from many of them by only using bloom filters to track rare cases where read set versions are evicted from the private hierarchy, while the majority case of conflicts is still detected using the sharer vector.

The evicted read set bloom filter is used to validate the transaction on the commit point against previously committed transactions. In order to logically serialize transaction's commit for this purpose, OverlayTM assigns every transaction that requests commit a monotonically increasing cts from the same global timestamp dispenser as it assigns bts. The committing transaction then validates its evicted read set bloom filter with the write sets of committed transactions whose cts is between the committing transactions bts and cts. The validation fails if there is a non-empty overlap, in which case the committing transaction must abort.

**The hardware Commit Queue.** To track the write sets of already committed transactions, OverlayTM adds a hardware Commit Queue at the shared level as shown in Fig. 3.7. The Commit Queue consists of an array of write set entries, with each entry being a bloom filter tagged by the cts of the transaction that generates the write set. The Commit Queue takes the bts of the validating transaction and its evicted read set bloom filter as input. Validation proceeds by (i) comparing the input bts with the cts tags in the queue, and (ii) performing set intersection test by bit-wise AND'ing the input bloom filter and write set bloom filters in Commit Queue entries. The validation succeeds if all write set entries with a cts $>$ input bts yield an empty set for the

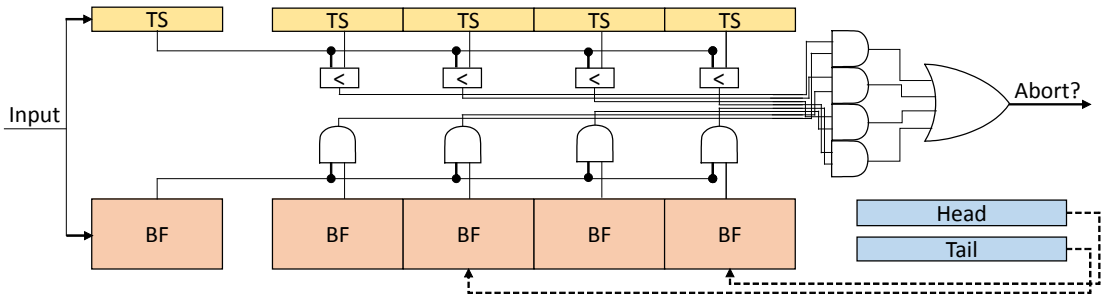



Figure 3.7: **Hardware Commit Queue** – This structure tracks write sets of previously committed transactions. The evicted read set bloom filter is tested against entries in the Commit Queue for validation.

intersection test.

The Commit Queue is populated during the version commit process. When a transaction requests to commit, it first acquires a cts from the global timestamp dispenser, which also causes a write set entry to be allocated from the Commit Queue. If the Commit Queue is full, the oldest entry is removed, and running transactions with bts smaller than the entry's cts are also aborted. During version commit, for every write address being sent to the Version Directory, the address is also inserted into the bloom filter by the Version Directory on behalf of the committing transaction. After version commit, the cts of the transaction is written into the write set entry, "sealing" the entry for future validation.

### 3.3.3 Read-Only Optimization and Early Release

OverlayTM's snapshot read semantics enables read-only transactions to always commit successfully without forced abort for any reason. Read-heavy workloads benefit from OverlayTM by marking transactions that do not modify shared memory as read-only, such that OverlayTM can execute them without validation. Early release is another related technique that excludes certain transactional read operations from conflict detection [100]. Whether and how early release can be applied is application-dependent (e.g., in `labyrinth` from STAMP benchmark, most read operations can be early released). Early release can be taken advantage of in application code by wrapping certain read operations with a software macro which is translated by the compiler to a non-conflicting read operation (e.g., by setting a bit in the read request). The non-conflicting read operation, when executed within a transaction, excludes the address from being added into the read set. The transaction thus suffers fewer conflict aborts, due to having a smaller read set.

OverlayTM supports both read-only optimization and early release by adding an extra "RO" bit to each block in the private hierarchy. Transactional read operations from read-only transactions or marked as early release reads are executed as normal transactional read, except that (i) when the version is inserted into the private hierarchy, the "RO" bit is also set, and (ii) the Version Directory does not abort the transaction even if the read operation sees a newer committed version. Versions with the "RO" bit set also need special treatment, such as (i) when such a version is evicted from the private hierarchy, its address is not added to the evicted read bloom filter; and (ii) when a read-only or early release transaction commits, all blocks with "RO" bit set should be invalidated. This operation prevents the private hierarchy from serving stale versions to future transactions.

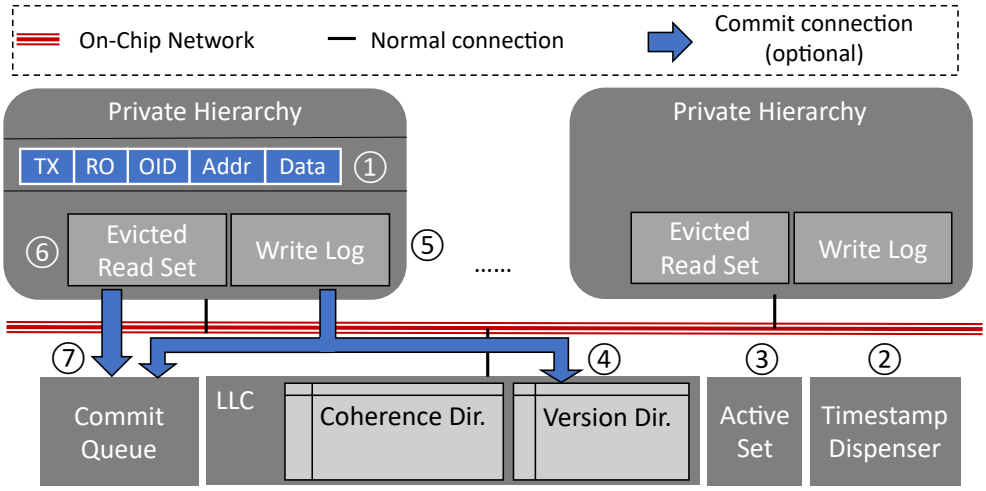



Figure 3.8: **OverlayTM** – Newly added components are marked with circled numbers.

## 3.4 Putting It All Together

Fig. 3.8 depicts OverlayTM's overall system design. At the LLC level, OverlayTM adds four extra hardware components:

- **Timestamp Dispenser (②):** Dispenses begin and commit timestamps to transactions when they begin and commit, respectively. This component can be implemented as a global counter that increments by one after each allocation.
- **Active Set (③):** Maintains the bts of transactions that are still active. This set participates in version garbage collection as described in Section 3.2.4.
- **Version Directory (④):** Tracks committed versions in the hierarchy. Pivot of Version Coherence Protocol, and implements Version Access Rule (Section 3.2.2).
- **Commit Queue (⑦):** Maintains bloom filters representing write sets of committed transactions. Performs conflict detection for the rare case that a transaction's read set version is evicted from the private hierarchy (Section 3.3.2).

OverlayTM also adds two extra components in every private hierarchy:

- **Write Log (⑤):** A set of block addresses that have been written to by the transaction. Provides addresses of dirty versions during version commit (Section 3.2.5).
- **Evicted Read Set (⑥):** A bloom filter summarizing addresses of evicted versions in the read set. This set is validated against write sets of committed transactions during validation.

Finally, OverlayTM extends the per-block tag (①) in the private hierarchy with the following fields:

- **OID:** The OID field from Page Overlays.
- **RO bit:** Whether the block is brought into the private hierarchy by a read-only transaction or an early release read operation (Section 3.3.3).
- **TX bit:** Whether the block belongs to a transactional read or write set. The dirty bit (not shown) further distinguishes between a transactionally read or written block.

Optionally, a dedicated datapath from the private hierarchy to the LLC components (blue arrows in Fig. 3.8) can be added to enable faster commit. In particular, a commit connection from

the write log to the Version Directory and the Commit Queue would help reduce bus bandwidth consumption during version commit. Besides, adding an extra datapath between the evicted read set and the Commit Queue may enable faster validation because the evicted read set bloom filter can then be transferred to the Commit Queue at full speed. These connections, however, are not mandatory because both the version commit messages and the read set bloom filter can be transferred using the regular on-chip network as well.

In the remainder of this section, we summarize OverlayTM's operations by going through a transaction's lifetime.

**Transaction begin:** Transaction begins on executing the `XBEGIN` instruction. The processor first executes a full barrier, and in the meantime, acquires a bts from the Timestamp Dispenser. The bts is stored in a special-purpose register and is included in transactional memory operations and version commit messages throughout the execution. The newly started transaction's bts is also added into the Active Set.

**Transaction execution:** During transaction execution, memory operations implicitly exercise transactional semantics unless indicated otherwise (e.g., for read-only transactions and early release reads). The "TX" bits of transactionally accessed blocks are set, and if the version is accessed in a read-only transaction or via an early release read, then the "RO" bit is also set. For transactional write operations, the write address is also inserted into the write log, if it is performed for the first time.

A running transaction may receive forwarded version commit messages. In this case, the private hierarchy performs conflict detection as described in Section 3.3.1. If a conflict indeed occurs, the transaction will abort.

On eviction of a version, the private hierarchy checks whether the "TX" bit is set and if positive, the block address is inserted into the evicted read set bloom filter.

**Transaction commit:** A transaction commits when it executes the `XEND` instruction. The commit process consists of two stages. In the first stage, the transaction validates by sending its evicted read set bloom filter to the Commit Queue for backward validation (if the bloom filter is empty, this step is simply skipped). If the validation succeeds, then the second stage begins, in which the transaction acquires a cts from the global timestamp dispenser and then performs a version commit for every address in the write log. During the second stage, a write set bloom filter is allocated from the Commit Queue and is populated as the version commit proceeds. This stage also performs forward validation against concurrently executing transactions with forwarded version commit messages. The transaction has not been fully committed yet during both stages and is still vulnerable to conflicts, if it receives version commit messages from concurrently committing transactions. If any of the validation steps fails, the transaction will abort.

The transaction eventually commits after the version commit completes. It performs post-commit cleanup by (i) clearing the "TX" bit in the private hierarchy; (ii) invalidating blocks with "RO" bit set, (iii) clearing the evicted read set bloom filter and the write log, and (iv) removing the transaction's bts from the Active Set. The post-commit cleanup is usually short because bit clearing and block invalidation can be implemented as low-level SRAM operations [180].

**Transaction abort:** A transaction can abort by (i) executing `XABORT`, (ii) version commits that conflict with versions in the private hierarchy, (iii) failing validation, and (iv) (rarely occurring) resource hazards on various hardware structures, such as the Version Directory and the Commit Queue. When a transaction aborts, it initiates a post-abort cleanup process that not only performs

every operation in the post-commit process described above, but also discards all changes by iterating over the write log, and removes all versions created by earlier transactional writes (including those that have been evicted). The post-abort process, however, can occur in the background because post-abort cleanup does not race with transaction execution.

## 3.5 Discussion

**Timestamp wraparound:** The original Page Overlays proposes using 16-bit OID tags in the cache hierarchy. While this narrow OID tag design incurs little storage overhead, it may cause timestamp wraparound in OverlayTM because the global timestamp must monotonically increase. One approach to deal with timestamp wraparound is to extend the OID tag to 64 bits, which will be practically sufficient and never wrap around, at the cost of a non-trivial 12.5% storage overhead in the cache hierarchy solely for OID tags.

A better way of managing timestamps without incurring substantial overhead is to leverage the fact that when the system is quiesced, i.e., when no active transaction is running, the timestamp can be reset, and all versions on the same address can be *consolidated* by discarding all but the top version. We, therefore, propose the following scheme: When the global timestamp is about to wrap around, the global timestamp dispenser blocks transaction begin by stalling begin requests and waits for the existing transactions to commit (and force abort after a timeout). After the system quiesces, the LLC initiates a version consolidation process by scanning every entry in the Version Directory and consolidating the versions. The consolidation logic removes all non-top versions and resets the top version to OID zero.

The above scheme eliminates the extra storage overhead but introduces a gap where no transaction can begin, which can degrade performance, especially on large LLCs. To overcome this problem, we modify the scheme by dividing execution into different phases, identified by a *Phase Variable*, and allowing transactions to begin in the next phase while version consolidation is performed in the previous phase, thus overlapping transaction execution with version consolidation. The modified scheme works as follows. Every 48 blocks in the LLC, called a "phase group", share a 48-bit phase variable indicating the phase that these blocks belong to, adding a mere overhead of one bit per block. The global timestamp dispenser, in addition to allocating timestamps, also manages the current Phase Variable, and distributes it to transactions on transaction begin and commit. The Phase Variable is included in every transactional memory operation. If a transaction accesses a block that is in an earlier phase than the transaction, then the LLC lazily performs version consolidation on the block and all other blocks in the same phase group. After version consolidation, all blocks in the phase group are promoted into the current phase by updating the shared Phase Variable to the accessing transaction's phase. The 48-bit Phase Variable, combined with the 16-bit OID tag, will never wrap around in practice.

**Time-virtualizing transactions:** An HTM design is considered as "time-virtualizing", if transactions could be temporarily suspended by entering the non-transactional mode and migrating to other cores. While OverlayTM efficiently space-virtualizes transactions, i.e., a transaction's working set can become larger than the private hierarchy or even larger than the LLC, extra hardware support is still essential to accomplish time virtualization. First, when a transaction is scheduled out-of-core, all uncommitted versions must be pre-committed into the Version Directory,

such that they can be transactionally accessed from another core. Second, the bts, write log, and evicted read set bloom filter are also saved as part of the transaction's state on a switch-out. Finally, when the transaction is switched in, a validation using the evicted read set bloom filter must be conducted to avoid missing any conflict.

**Race condition between transaction begin and commit:** A race condition might occur between transaction begin and commit, when a new transaction begins while another transaction is in the version commit process, and the new transaction acquires a bts larger than committing transaction's cts. The race condition will emerge when the new transaction reads an address that the committing transaction has also wrote, but the corresponding version has not been committed to the Version Directory. On this occasion, the transactional read operation is unable to access the correct memory snapshot, due to racing with the non-atomic version commit process.

OverlayTM resolves the race condition by letting the new transaction read the *stable* memory snapshot, i.e., the one that is lower than any committing transaction's bts, rather than the most up-to-date but unstable snapshot where new versions have not finished committing. To figure out which snapshot to read, on the first memory operation of a transaction, the Version Directory computes the current *Stable Memory Snapshot Timestamp (smsts)* using the Active Set, and includes that in the response message. The transaction then stores smsts in a special-purpose register and uses it to access versions until the concurrent version commit completes.

## 3.6 Evaluation

| | |
|---|---|
| Processor | 16 cores 4-way Our-of-Order @ 3GHz |
| L1-d/i caches | 32KB, 64B lines, 8-way, 4 cycles |
| L2 cache | 256KB, 64B lines, 8-way, 8 cycles |
| L3 cache | 32MB, 64B lines, 16-way, 30 cycles |
| DRAM | DDR3 1333 MHz, 4 controllers |
| zSim Phase Length | 200 cycles |

Table 3.1: **Simulation Configuration**

### 3.6.1 Simulation Configuration

**Simulation platform:** We extended zSim [219] to simulate OverlayTM. We chose zSim not only for its simulation speed but also because of its *execution-driven* approach is important for correctly modeling execution paths when transactions need to retry. Our simulation parameters are shown in Table 3.1.

The binaries that we simulate are compiled using the Intel TSX Restricted TM (RTM) interface [109]. We instrument three RTM instructions:

- **XBEGIN**: The processor enters transactional mode after taking a snapshot of the current register context. This instruction also takes the address of the fall-back handler. On a transaction abort, the control flow will transfer to the fall-back handler after the context is restored, as if the control flow had just transferred from the instruction to the handler.

- **XEND**: Commits the current transaction and all memory modifications. Control flow resumes normally after the instruction.
- **XABORT**: Aborts the current transaction. This instruction carries a user-defined return code that the fall-back handler can access. The abort code is put into `EAX`, with several hardware-set status bits. Control flow unconditionally jumps to the fall-back handler.

**Simulated HTM designs:** We compare OverlayTM against 2PL, which is a single-version, eager conflict detection HTM that resembles Herlihy et al. [99]. Our simulated 2PL HTM detects conflicts using the regular coherence protocol and adopts a "requestor-win" policy. We compare OverlayTM to 2PL to demonstrate the benefits of OverlayTM's lazy conflict detection. We assume that the 2PL design is unbounded in a way that both transactional read and write sets can be arbitrarily large.

We also compare against an idealized version of TCC [97] that supports unbounded transactions and fully scalable transaction commit. Our simulated TCC allows a transaction's working set to be evicted from the private hierarchy without aborting the transaction. Transaction begin and commit in TCC must be serialized with concurrent commits because TCC is a single-version HTM, and could not support the read snapshot semantics. The simulated performance of the idealized TCC model can be regarded as an upper bound of a more realistic TCC.

Finally, we simulate SI-TM [156] to compare OverlayTM with another multiversioning HTM design. SI-TM adopts snapshot isolation as its conflict detection protocol, but it also supports serializable transactions by validating the read set on commit (although the original paper did not explain how to track the read set). In our evaluation, we simulate the serializable version of SI-TM to ensure fairness of comparison. Compared with OverlayTM, SI-TM adds a Multiversion Manager (MVM) on the LLC access critical path, which increases access latency if a memory operation misses the private hierarchy (more about SI-TM in Section 3.7.1). In our simulation, we assume that the MVM is large enough such that accesses never miss, but it still incurs a fixed lookup latency of a L2 access.

The simulated OverlayTM design assumes background garbage collection and deals with timestamp wraparound lazily as described in Section 3.5. We assume perfect conflict detection between the evicted read set bloom filter and the Commit Queue. Our profiling shows that this simplifying assumption will not distort experimental results because (i) most transactions do not perform backward validation, as their read sets are perfectly contained in the private hierarchy, (ii) even if occasional validation is required, the number of evicted versions in the read set is small, and (iii) a moderately sized (2Kb) bloom filter can already achieve almost-perfect conflict detection. Prior works also suggest similar results [23, 268, 269] (although in [23], the authors may have mistakenly read the bloom filter size as 2KB rather than 2Kb—an 8$\times$ size difference).

All simulated HTM designs detect conflict in 64-bit word granularity. While this assumption is slightly unrealistic (but still possible by manually padding application data structure), it is enforced across all simulated HTM designs, and hence will not create any bias among them. Fine-grained conflict detection helps us better illustrate performance characteristics of each HTM design because it removes false sharing between unrelated words on the same cache block.

In addition, we add a fixed cycle latency for both simulated transaction begin and commit (50 for transaction begin and 100 for transaction commit). This latency covers the overhead of executing the full memory barrier, performing tag walks, and saving the execution context. If the

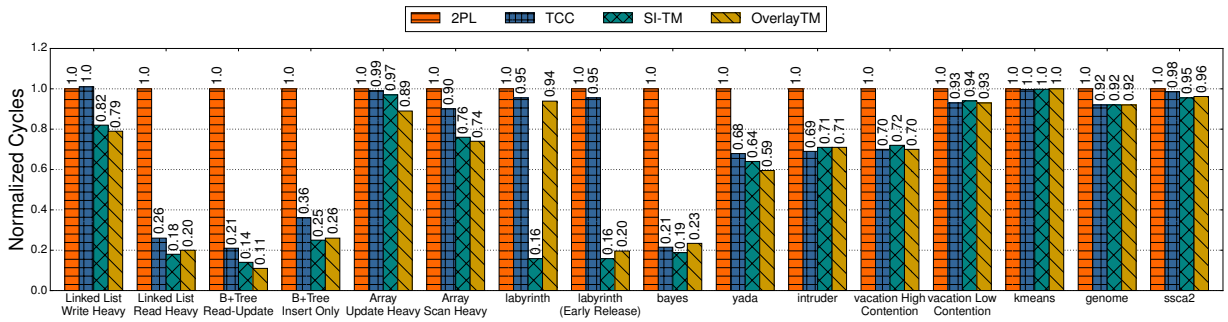



Figure 3.9: **Normalized Cycles** – All numbers are normalized to 2PL.

simulated HTM happens to perform other tasks on transaction begin and commit, these operations can be overlapped with this latency (prior works also adopt this assumption, e.g., [23]).

**Benchmarks:** Our simulation runs STAMP benchmark [173] with recommended parameters. Moreover, to further evaluate the feasibility of OverlayTM on a broad range of workloads, three data structures are used:

- **Linked List**: A singly linked list. Each node has an 8-byte key and 8-byte payload field. Threads first generate a random number $k$ between 0 and the current length minus one and then traverse $k$ nodes before they finally insert, delete or read the node after the current node. This workload models large read-write ratio since threads read a long prefix before they update one pointer.
- **B+Tree**: A standard B+Tree with 4KB nodes. Both key and payload are 8 bytes. Threads insert, update or read on certain "hot spots". This workload models write-heavy workload for insert-only, and moderate read-write ratio for read-update.
- **Array**: An array of integers with $(m+n)$ worker threads. $m$ threads perform linear scan on the array, $n$ threads write randomly chosen array entries. This workload models read-mostly analytic workload.

All random numbers are drawn from the `rand` function in the standard C library. We repeat each test case five times using the same random seed, and then taking the average.

All workloads adopt the lock elision algorithm described in Section 3.1.1. Critical sections are protected by a single state-of-the-art spin lock. Transactions elide the lock by reading the lock variable right after `XBEGIN`, and proactively aborts if the lock is already acquired. We provide a fall-back handler that implements retry logic. In the fall-back handler, the abort status code is checked. The transaction will restart if: (i) the abort is caused by transient conditions such as conflicts, and (ii) the number of retries is fewer than five. Otherwise, the non-lock-elision path will be executed, and the lock will be physically acquired.

All workloads use the default `glibc` allocator. Memory instructions within the transactional region are treated as transactional unless the transaction is explicitly marked as read-only, or the operation is an early release read.

### 3.6.2 Performance Analysis

We present simulation results in Fig. 3.9 (normalized cycles), Fig. 3.10 (aborts per committed transaction) and Fig. 3.11 (normalized aborts). All normalization uses 2PL as the baseline.

Overall, in 11 out of the 16 workloads (all except array update-heavy, `vacation` low,

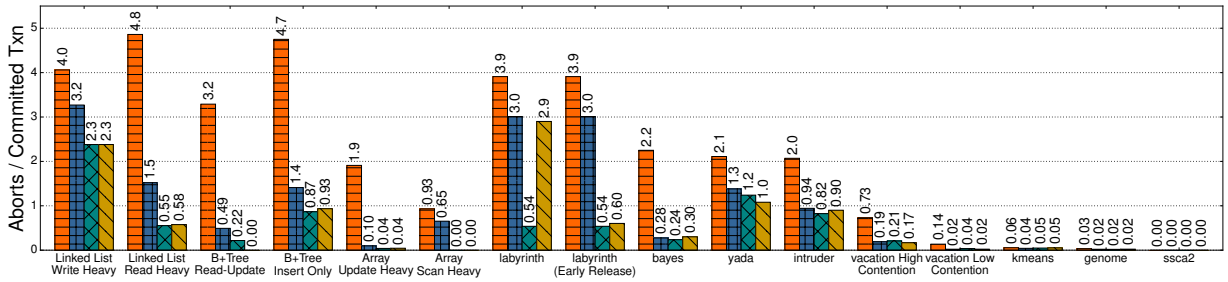



Figure 3.10: **Aborts Per Committed Transaction** – All results are absolute numbers.

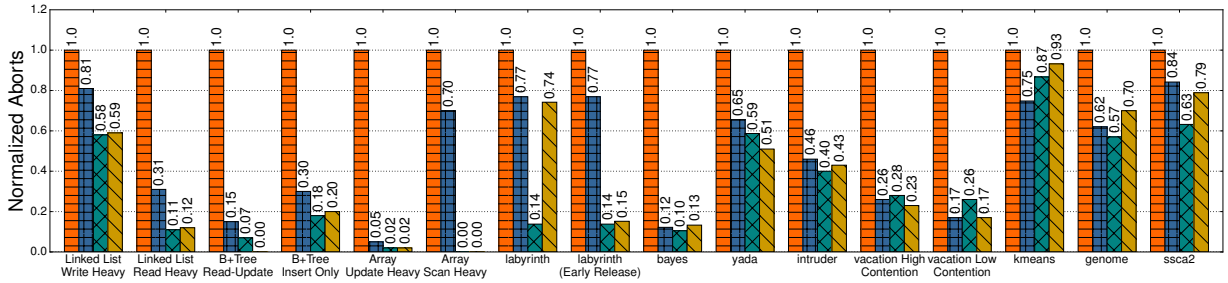



Figure 3.11: **Normalized Aborts** – All numbers are normalized to 2PL.

`genome`, `kmeans`, `ssca2`), OverlayTM outperforms 2PL by more than 20%, which demonstrates a great advantage of lazy conflict detection over eager 2PL. Furthermore, in 8 out of the 16 workloads, OverlayTM outperforms TCC with both fewer cycles and lower abort rates, which shows the effectiveness of multiversioning. Most interestingly, OverlayTM, with its ability to commit read-only transactions regardless of concurrent writes is comparable in most cases with SI-TM (23% more aborts and 11% more cycles, except `labyrinth`), and in some cases (`array`), even outperforms SI-TM, due to OverlayTM's faster version access without MVM. We consider this as the most considerable merit of our design: achieving efficient hardware multiversioning with the support of Page Overlays.

We analyze each benchmark as follows:

**Linked List:** The initial list has 256 nodes. Worker threads either insert, delete, or read a node. Write-heavy workload has 20% read, 40% insert, and 40% delete; Read-heavy workload has 80% read, 10% insert, and 10% delete.

The linked list workload models a broad range of commonly used data structures, including queues, stacks, chaining hash tables, and skip lists. Our configuration features high contention and high read-write ratio since threads read a "prefix" of all nodes in the list before they finally stop. This behavior makes them vulnerable to WAR conflicts incurred by any updating transaction on the prefix.

In Fig. 3.11, we can see that both eager conflict detection (2PL) and forward OCC (TCC) suffer from high abort rates, as well as more wasted cycles. As explained earlier, 2PL and TCC are sensitive to conflicts on the prefix, while multiversioning HTM, i.e., SI-TM and OverlayTM, commits read-only transactions using the snapshot they take at transaction begin. In addition, 2PL has more aborts than TCC because 2PL is an eager HTM design that aborts transactions on coherence requests. Both OverlayTM and SI-TM achieve a 20% speedup with 40% fewer aborts.

**B+Tree:** In the first insert-only stage, 16 worker threads insert 16384 keys into the tree. Then in the second stage, threads either update or query the tree. The first phase models the B+Tree

index of an Online Transactional Processing (OLTP) table, where new rows are concurrently created by assigning monotonically increasing row IDs. The second phase resembles YCSB-A [53] read-update: 50% reads and 50% updates are performed on a "hot spot" that is gradually drifting in the key space. Compared with the linked list workload, the B+Tree workload has fewer contention and lower read-write ratio, since conflict almost only happens on leaf levels, and node split is rare.

For insert-only, all other three HTMs outperform 2PL by 60%–75%, with 70%–80% fewer aborts. The performance issue of 2PL is caused by transactions in 2PL exposing writes eagerly, which is detrimental. On average, half of the node data is moved around when inserted into a B+Tree leaf node, causing uncommitted write-read and write-write conflicts. In addition, SI-TM performs best because SI-TM commits transactions even for committed WAR. For read-update, OverlayTM runs 20%–90% faster than all other HTM designs with negligible aborts, while SI-TM suffers extra latency incurred by the MVM, and TCC suffers excessive aborts due to its inability to perform read-only optimization.

**Array:** The array workload models an Online Analytical Processing (OLAP) table, where worker threads update the table at the front end, and background threads run real-time auditing operations that scan the entire table. This workload features long read-only sequences and short updates, and hence exemplifies the performance advantage of supporting read-only optimization. As expected, both OverlayTM and SI-TM handle this case exceptionally well due to multiversioning: performance improves by 10%–25% with negligible aborts. Although cycle improvement is not as significant as abort rates, we argue that, in this scenario, latency (i.e., number of aborts for auditing transactions) is more critical than aggregated cycles because the background auditing threads might be used to support real-time decision-making systems where the timeliness of data is the uttermost priority.

**STAMP:** OverlayTM, SI-TM and TCC improve performance by approximately 30% on `vacation` high and `intruder` out of the four STAMP workloads, with 57%–77% less aborts. For `bayes` and `yada`, the performance is improved by 80% and 60% respectively, with 90% and 49%–75% less aborts. On `genome`, `kmeans`, `ssca2` and `vacation` low, performance improvement is very limited (less than 10%) for all three HTM designs. This result can be explained by the fact that the absolute abort rate is already low in our implementation of lock elision: Only 3.3% of total 14749 transactions suffer aborts for 2PL in `genome`. Optimizing aborts is meaningless in this case because only a small fraction of execution cycles are wasted.

`labyrinth` stands as an excellent example of how early release can reduce aborts and improve performance. `labyrinth` implements the transactional version of Lee's algorithm [143, 253]. The algorithm runs Breadth-First Search (BFS) between a point pair on a maze after copying the maze to the local storage, and then attempts to establish a connecting path between the two points using backtracking. Without early release, the copy operation will add the entire maze into the transaction's read set, which conflicts with *every* other transaction that updates the maze, resulting in excessive transaction aborts for all simulated designs, as can be observed in our experimental data. However, in this special case, transactions only serialize with each other by writing to the same grid of the maze. The copy operation, contradicting the strict transactional model, can therefore be excluded from conflict detection, as long as a consistent snapshot is read. In this case, a transaction may have used a stale (but still consistent) copy of the maze for path planning. Nevertheless, as long as the actual path it produces does not conflict with other paths,

the transaction can still commit.

Based on the above observations, we implement early release on `labyrinth` by excluding read instructions on the maze during the copy operation from conflict detection. These instructions are marked with a compiler macro. When simulated, they will be recognized as early release reads.

Without early release, all HTM designs except SI-TM perform poorly due to excessive conflicts on the maze read set. SI-TM stands out because of its conflict detection mechanism that can evade conflicts in this case. With early release enabled on OverlayTM, however, only transactions that indeed conflict (i.e., two transactions select the same grid during backtracking) will abort, ignoring version commits on read-only grids, since they are not added to the read set. The overall performance is comparable to that of SI-TM, a significant improvement. Early release, however, cannot be easily implemented on TCC and 2PL because they are single-version designs that track versions using the coherence protocol.

We also noticed that our experimental results on `labyrinth` seem to deviate from what has been published [156], where all HTM designs seem to suffer unanimously high abort rates. The explanation is that the lock elision algorithm we adopt in the evaluation upper bounds the number of retries a transaction may attempt using a retry counter. Once this upper limit is reached, the thread will acquire the global lock and execute non-transactionally. This simple technique avoids one or a few long transactions repeatedly aborting all other transactions and each other in eager HTM designs such as 2PL, reducing aborts significantly as in our experiments.

### 3.6.3 Commit Overhead

This section presents experimental results on OverlayTM's commit overhead. We measure the commit overhead by modeling the contention of the Version Directory during the version commit process. We assume an average contention-free latency of 20 cycles for completing the version commit of one transaction (this figure is derived from results of previous work [97]). The simulator models a sufficiently large buffer in front of the Version Directory that queues incoming transaction version commit requests. We simulate the wait time each version commit request experiences in the buffer as the overhead of commit operations and summarize the result in Table 3.2.

Our analysis shows that, for most workloads that we use, the version commit process does not constitute a bottleneck. 12 out of 16 workloads spend less than 5% of total execution time waiting for version commit to complete. For B+Tree write-heavy, B+Tree insert-only, and `kmeans`, the percentage of wasted cycles is higher but are still less than 7%. `ssca2` represents one extreme case where the version commit overhead constitutes 26% of total execution cycles. After thoroughly investigating the profiling outputs, we figured out that the high overhead of version commit `ssca2` is due to extremely short transactions, the average size of which is only a couple of instructions, which complete in around 20 cycles. In this case, the version commit process has become a significant factor in the transaction's end-to-end latency. Fortunately, since both transaction begin and end in our simulation have a fixed cycle latency, the overhead of frequent version commit is overlapped and, therefore, not reflected in the execution time.

| Workload | Avg. # Pending | % Cycles |
|---|---|---|
| Linked List Write-Heavy | 1.65 | 2.40 |
| Linked List Read-Heavy | 0.79 | 3.33 |
| B+Tree Write-Heavy | 1.42 | 6.39 |
| B+Tree Insert-Only | 5.00 | 5.73 |
| Array Update-Heavy | 0.12 | 0.012 |
| Array Scan-Heavy | 0.39 | 0.43 |
| `genome` | 3.62 | 4.54 |
| `vacation` High Contention | 4.23 | 0.63 |
| `vacation` Low Contention | 4.62 | 1.10 |
| `intruder` | 3.93 | 3.70 |
| `bayes` | 1.66 | 0.059 |
| `kmeans` | 2.46 | 5.75 |
| `labyrinth` | 1.90 | 0 (negligible) |
| `labyrinth (Early Release)` | 1.67 | 0 (negligible) |
| `ssca2` | 3.49 | 26 |
| `yada` | 2.64 | 0.091 |

Table 3.2: **Commit Overhead** – The first column lists the average number of pending commit requests when a transaction requests to commit. The second column shows the percentage of total cycles spent on the commit process.

## 3.7 Additional Related Work

### 3.7.1 Snapshot Isolation HTM

Snapshot Isolation HTM (SI-TM) [156] is another HTM design that implements hardware-supported multiversioning. We have briefly covered SI-TM in Section 2.2.1, and this section discusses some of its design trade-offs in more detail.

As its name implies, SI-TM only enforces *snapshot isolation*, which is a weaker ordering guarantee than serializability [29]. In snapshot isolation, transactions read from a consistent snapshot created by prior committed transactions (i.e., identical to OverlayTM's snapshot read semantics), and transactions only conflict via concurrent writes on the same address. Snapshot isolation is a popular option for DBMS, where more relaxed isolation levels than serializability are common [112]. In these scenarios, transactions only transform data, not control flow. However, for general-purpose HTM, snapshot isolation is barely implemented and often a poor ordering

guarantee. Two reasons contributed to its unpopularity. First, snapshot isolation is not as intuitive as serializability, and the possible interactions between transactions are hard to reason. For example, one of the most notorious non-serializable anomalies, *write skew* [235], will occur on STAMP workloads under snapshot isolation. Second, when applied to control flow, snapshot isolation can produce unpredictable branch conditions or even branch targets, resulting in severe outcomes and security breaches.

SI-TM timestamps transactions with a *bts* and *cts* just like OverlayTM, but cache blocks are not extended with OID tags. To enable hardware multiversioning, SI-TM adds a Multiversion Manager (MVM) between the private levels and the LLC. The MVM functions as an extra level of indirection that translates a transactional access consisting of the *bts* and the canonical address to a shadow address. This process is comparable to the version read operation using the Version Access Rule of OverlayTM. The shadow address serves the same purpose as the overlay address in OverlayTM (to distinguish versions on the same address from different transactions). However, it must be allocated from the MVM when a dirty version is written back from the L2. SI-TM handles conflicts via commit-time validation, but instead of performing both forward and backward validation as OverlayTM, it simply performs backward validation by checking whether another transaction with a higher *cts* has already committed on the committing transaction's write set. Dirty versions are written back to the LLC after successful commits to prevent aliasing.

In addition to the weak ordering guarantee, SI-TM differs from OverlayTM in the following aspects. First, the MVM is on the version access critical path, whose latency cannot be overlapped with LLC operation. As a result, SI-TM increases the access latency for all memory operations that miss the private hierarchy. Second, SI-TM suffers the aliasing problem even for the read set because the private hierarchy solely tags blocks with the canonical address, without any extra identification to indicate which snapshots they belong. Newly started transactions may access a stale version cached in the private hierarchy, even if a newer version has been committed elsewhere. The original SI-TM paper did not address this problem adequately. One possible solution is to flush the private hierarchy to eliminate the potentially incorrect versions when a new transaction starts. This addition to SI-TM will likely cripple its performance because new transactions cannot leverage cached data from the previous execution.

### 3.7.2 Optimizing Existing Software Stack with HTM

Hardware Transaction Memory (HTM) has been gaining popularity in the last decade as a convenient replacement for high-performance, software-based synchronization mechanisms such as spin locks. Since Intel announced its HTM support on the x86 platform, namely, the Transactional Synchronization Extension (TSX) [109], numerous attempts have been made to incorporate this new hardware technology into existing software where synchronization performance is critical (e.g., [8, 9, 127, 149]).

### 3.7.3 Adopting HTM for Byte-Addressable Non-Volatile Memory

Existing HTMs are designed without durability guarantees. As a result, all working data, including those written by committed transactions, will be lost if the system fails. Prior works overcome this limitation by proposing combing HTM with the emerging Byte-Addressable Non-Volatile Memory

(NVM), adding durability support into existing HTM [40, 82, 118, 122, 158, 159, 186]. With so-called "durable transactions", successfully committed data will be persisted to the NVM and can be recovered after a system failure, providing extra durability ("failure atomicity") guarantees in addition to the conventional atomicity guarantee.

Data structures specifically designed for NVM can also benefit from existing HTM implementation. In the existing implementation, the L1 cache temporarily withholds transactionally written blocks until the transaction commits. This property helps NVM libraries to retain dirty data in the cache, protecting the persistent image on the NVM from being polluted by inadvertent write backs [194, 225].

### 3.7.4 Multiversioning in Software Transactional Memory and DBMS

Multiversioning can be implemented in Software Transactional Memory (STM) as a version chain [64, 204] to achieve the same snapshot read semantics as in SI-TM and OverlayTM. Each node in the version chain represents a version of a data item and carries a software-assigned timestamp to reflect the snapshot the object belongs. The software also implements the Version Access Rule by traversing the version chain to locate the correct version to read. Compared with hardware multiversioning, software implementations suffer performance degradation due to the instruction and memory overhead of explicitly maintaining the version chain [39].

OverlayTM's Version Access Rule also resembles the version chain traversal operation in DBMSes that implement Multi-Version Concurrency Control (MVCC) [62, 142, 187, 258]. In MVCC, every database transaction modifying a tuple will generate a version record containing the updated data. The version record is tagged with the writing transaction's timestamp and is typically maintained with version records generated by concurrent and earlier transactions in a linked list. Reading transactions that access the same tuple will have to traverse the linked list and select the correct version to read according to the reader's timestamp.

# Chapter 4

# NVOverlay: Enabling Efficient and Scalable High-Frequency Snapshotting to NVM

This chapter presents NVOverlay, a hardware memory snapshotting design built upon Page Overlays. NVOverlay captures consistent memory snapshots over the entire address space and persists them to Byte-Addressable Non-Volatile Memory (NVM) for later retrieval. One of NVOverlay's most important user scenarios is to perform failure recovery when the system crashes due to power failure or software error. In such an occasion, NVOverlay can restore the system back to a pre-crash consistent state by loading a previously saved snapshot into the main memory, resuming execution from the point where the snapshot is taken. This feature not only improves the system's overall availability when failures are inevitable (e.g., in a data center, or on energy harvesting devices), but also preserves the progress of computation in case of unexpected failures. The latter feature is particularly precious for long-running scientific computing and machine learning applications, where a single crash or software error can easily reset hundreds of hours of computational effort.

Two major challenges are associated with capturing consistent memory snapshots. First, the performance impact on application programs should be minimum. Previously proposed software approaches that focus on persistent transactions are a mismatch because of the high overhead of executing persist barriers for enforcing write ordering. Hardware logging approaches such as PiCL [189] get rid of persist barrier by enforcing write ordering on the cache controller and overlapping the persistence of snapshot data with execution. However, the intrinsic $2\times$ write amplification introduced by logging can still be troublesome due to the extra write bandwidth. Second, the snapshotting design should scale to large multicore systems and working sets. Existing approaches that rely on centralized control or hardware resource will find themselves unable to keep pace with hardware development in both processor count and memory capacity in the near future (which are projected to be hundreds of cores and TBs of storage, respectively). For example, in PiCL, it is assumed that the LLC is an inclusive, monolithic piece of hardware, where the centralized control logic is built. However, even in today's architecture, the LLC is no longer monolithic nor inclusive. Instead, the LLC hardware is distributed across processors as non-inclusive cache slices for better scalability [27, 114, 131, 182, 252]. Moreover, PiCL has

proposed to execute a *global barrier* which halts all processors in the system before a snapshot is taken. This approach severely limits the scalability of the design, despite guaranteeing the consistency of the memory snapshot since the cost of the global barrier increases with the core count. It is, therefore, particularly challenging to design a memory snapshotting mechanism that scales with the size of the system.

NVOverlay addresses the first challenge with hardware-supported multiversioning. First, to enable fine-grained, incremental tracking of memory modifications, NVOverlay extends the "Overlay-on-Write" semantics of Page Overlays in its *Coherence Snapshot Tracking (CST)* component, such that writes to the memory will be performed as overlay writes if the address contains pre-snapshot memory content that has not yet been saved. The overlay write only involves localized L1 operations and is hence extremely lightweight. Second, the cache block containing before-snapshot data produced by the overlay write will then be promptly written back to the NVM, stored in overlay pages, and indexed for future retrieval by a *Multi-snapshot NVM Mapping (MNM)* component. The MNM extends the OMS and OMT from Page Overlays, and it shadow-maps data of different snapshots to different addresses on the NVM. Using shadow mapping considerably reduces write amplification, because it is sufficient to write each block only once to ensure the atomicity of the snapshot.

For scalability, NVOverlay divides the system into smaller autonomous units called *"Versioned Domains (VDs)"*. VDs do not share any hardware resource beyond those that are essential for accessing shared memory and are hence fully scalable on both processor count and memory size. Each VD operates on its own to generate memory snapshots and write them back to the NVM without having to execute the global barrier for synchronization. To ensure the consistency of memory snapshots in the absence of a globally coordinated point where the snapshot is taken, NVOverlay adopts a *relaxed consistency model*, in which the memory snapshot being generated may not represent any valid system state in real-time but is still consistent in a logical sense regarding data dependency. VDs implement this model using the existing coherence protocol to track data dependencies between memory operations from different snapshots and then encode the dependencies by updating a distributed Lamport clock. Write-backs of snapshot data then follow the logical ordering implied by the Lamport clock, such that the memory snapshots already persisted on the NVM are always consistent.

Besides failure recovery, the novel design of NVOverlay also favors the adoption of several software techniques, such as time-travel and record-and-play debugging [167, 174, 250]. These techniques require support for efficient and high-frequency memory snapshotting, and they pose several unique challenges for NVOverlay. First, snapshots tend to occur in bursts, and the interval between two consecutive snapshots are relatively shorter compared with the case for failure recovery. As a result, the snapshotting design ought to be able to create many versions on the same address quickly in a short period without incurring significant overhead. Second, these scenarios are more sensitive to timing changes and unnecessary synchronization than failure recovery. Thus, a feasible design must not rely on the global barrier, which may alter the behavior of the debugged process. As demonstrated later in the evaluation section, NVOverlay satisfies both goals compared to prior works on hardware logging, proving its broad applicability beyond failure recovery.

## 4.1 Design Overview

### 4.1.1 The Failure Model

NVOverlay assumes a rather general failure model, where a system failure will cause all volatile states, including a processor's register context, dirty data in the cache hierarchy, and data stored in volatile DRAM, to be lost entirely. Write operations that have not yet been committed into the persistent NVM will also be lost, including those that are still in the memory controller's Write Pending Queue (WPQ) when the crash happens. Note that some previous works [139, 277] have assumed that the memory controller of NVM device is in the persistence domain, i.e., memory requests that have entered the WPQ of the controller are guaranteed to be persisted using a technique called *Asynchronous DRAM Refresh (ADR)* [181]. As a result, these designs can assume a more lenient failure model where the system receives a "last notification" shortly before the failure, with adequate power for sustaining a few more writes before the system completely dies. NVOverlay, on the contrary, does not require ADR support and only assumes the atomicity of a single 64-byte write instead.

After the system reboots, failure recovery is performed by loading a previously consistent memory snapshot and the register context dump back to the system state, after which execution resumes as if no failure had ever occurred. External events that do not belong to the memory states will be permanently lost. However, previous works have thoroughly studied this problem and proposed a few solutions [167, 236, 250].

### 4.1.2 Epoch-Based Memory Snapshotting

NVOverlay assumes an epoch-based snapshotting model, where the execution is divided into consecutive but disjoint intervals, called "epochs", which are identified by unique numeric notations called epoch numbers. All memory modifications within an epoch are saved to the NVM as an incremental memory snapshot labeled with the epoch number, and snapshot data belonging to an epoch can be randomly accessed once saved.

NVOverlay maintains working data and snapshot data separately. On the one hand, working data on which the system operates is not disrupted by snapshotting, and it can be stored in either DRAM or NVM. On the other hand, snapshot data generated by the hardware will be persisted to the NVM in a side channel, which does not interfere with working data reads and writes. When a snapshot is taken, every processor in the system also takes a dump of the register context and saves it to the NVM. The register dump is loaded back on recovery so that execution can resume from the snapshot.

The clean separation between working data and snapshot data distinguishes NVOverlay from prior works, which often assume that the working data must also be stored on NVM (just as snapshot data) and constitute part of the snapshot. These designs are rather intrusive to regular execution by forcing computation to adopt NVM's programming paradigm and use the NVM as main memory. As a result, they demonstrate worse performance due to NVM's lower bandwidth and longer read latency [264, 265].

### 4.1.3 Extending Page Overlays

NVOverlay leverages the hardware multiversioning support provided by Page Overlays to efficiently track incremental memory modifications within an epoch. NVOverlay consists of two components: a Coherent Snapshot Tracking (CST) component that performs "Overlay-on-Write" when memory writes are conducted on an address for the first time after a snapshot is taken, and a Multi-snapshot NVM Mapping (MNM) component that writes snapshot data back to the NVM and organizes them for future retrieval. As we have discussed in the introduction of the chapter, this combination eliminates *all* persist barriers and minimizes write amplification, hence greatly reducing the cycle and bandwidth overhead of snapshotting memory.

Both the CST and the MNM are designed to be scalable. To avoid using global barriers for generating strictly consistent snapshots, we relax the consistency requirements of snapshots such that snapshots can be taken in a state that not necessarily represents any real-time state that has ever occurred during execution, but yet is still consistent in terms of data dependencies defined by the coherence protocol (we discuss the consistency model shortly in Section 4.1.4). Furthermore, to eliminate centralized control logic for coordinating data write-back, we group private cache hierarchies into autonomous Versioned Domains (VDs), and let each VD maintain its own *local epoch*. VDs update their local epochs when they acquire dirty data from a higher numbered epoch via cache coherence, such that higher numbered epochs may depend on data generated in lower numbered epochs but not vice versa. Snapshot data will then be written back to the NVM in increasing order of epoch numbers. This protocol guarantees the consistency of already persisted snapshots. When an epoch $E$ is persisted, all smaller epochs than $E$, on which snapshot data generated in $E$ depends, must have also been persisted.

NVOverlay tracks snapshot data generated by epochs using Page Overlays' per-block OID tag in the memory hierarchy. When an Overlay-on-Write is performed to create an overlay block, the OID tag of the block is assigned to be the local epoch number of the VD in which the write is executed. The OID tag is transferred in coherence messages and evictions as a means of epoch synchronization between VDs. NVOverlay also extends the main memory to store the OIDs for all blocks in the physical storage to prevent losing track of the OID when a block is evicted out of the cache hierarchy The OID tag is fetched in parallel with data when a memory request accesses the main memory.

### 4.1.4 Relaxed Consistency Model

Ideally, a consistent memory snapshot can be captured when the system is in a quiescent state, in which all processors are halted, and no memory operation is on-the-fly. The memory snapshot captured this way represents an actual state of the system that has existed in real-time during execution, and is therefore strictly consistent. In practice, however, reaching a genuinely quiescent state using a global barrier in a large system is unrealistic, because all processors need to stop and synchronize with each other. The global barrier would severely limit the scalability of the design, and disrupt normal execution by effectively blocking the system from making any progress for an extended period.

Alternatively, processors can create *fuzzy snapshots* by only atomically saving the local states in their caches without any global coordination. While this approach completely removes inter-

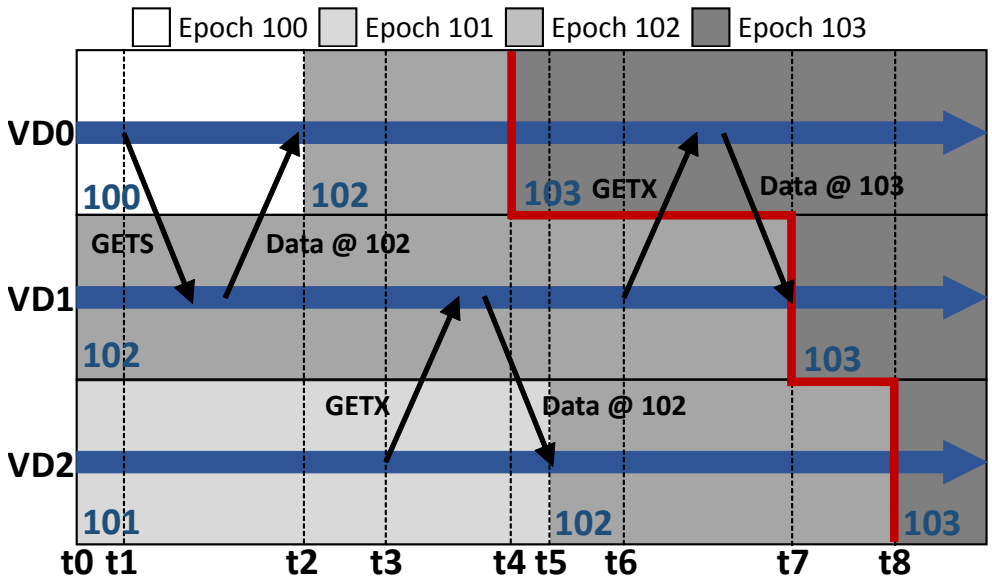



Figure 4.1: **Relaxed Consistency Model** – The horizontal direction reflects the flow of time and the vertical direction represents different VDs. The red line shows the system state captured at epoch 102 (t4, t7, t8 for VD0, VD1 and VD2, respectively).

processor synchronization, the fuzzy snapshot may not represent a valid execution state and hence cannot be used for proper failure recovery. For example, imagine that on a two-core system, processor #1 takes a snapshot and writes address $A$, while processor #2 reads address $A$, multiplies it by 100, and writes the result into address $B$ before it takes a snapshot. In the final snapshot, the value of $B$ is included, but the value of $A$ is not. Imagine that the snapshot is loaded back into the main memory after a crash. In the restored system, processor #2 may proceed to access $A$ and $B$, only to read inconsistent data, because the value of $B$ depends on the value of $A$, but the value of $A$ is missing.

NVOverlay differs from the above two approaches by not using global barriers for capturing strictly consistent snapshots, but avoiding fuzzy snapshots by tracking data dependency between VDs (barriers are still needed within VDs, but they are local barriers and are relatively lightweight). Formally speaking, if a VD running on local epoch $i$ acquires a dirty cache block containing data generated in another epoch $j$, and $j > i$, the VD will immediately upgrade its local epoch to $j$, meaning that every piece of data produced by this VD may potentially depend on epoch $j$, and hence should be persisted no earlier than epoch $j$. This ordering constraint is later on enforced when snapshot data is being written back to the NVM by giving priority to lower number epochs. On recovery, if an epoch $k$ is restored, then all memory states that $k$ might depend on must also have been restored. In the fuzzy snapshot example above, it means that the value of B is included in the restored state only if the value of A is.

The dependency tracking mechanism in NVOverlay is similar to how a Lamport clock captures the ordering of events in a distributed system using the notion of logical time [71, 141]. As a result, the snapshot taken by NVOverlay may not be the exact memory image at any real-time point during execution. An example of NVOverlay's relaxed consistency model is depicted in Fig. 4.1. In this example, the captured snapshot includes the real-time memory state of `VD0`, `VD1`, and `VD2` in logical time `t5`, `t7` and `t8`, respectively (indicated by the red bold line), but it is impossible to

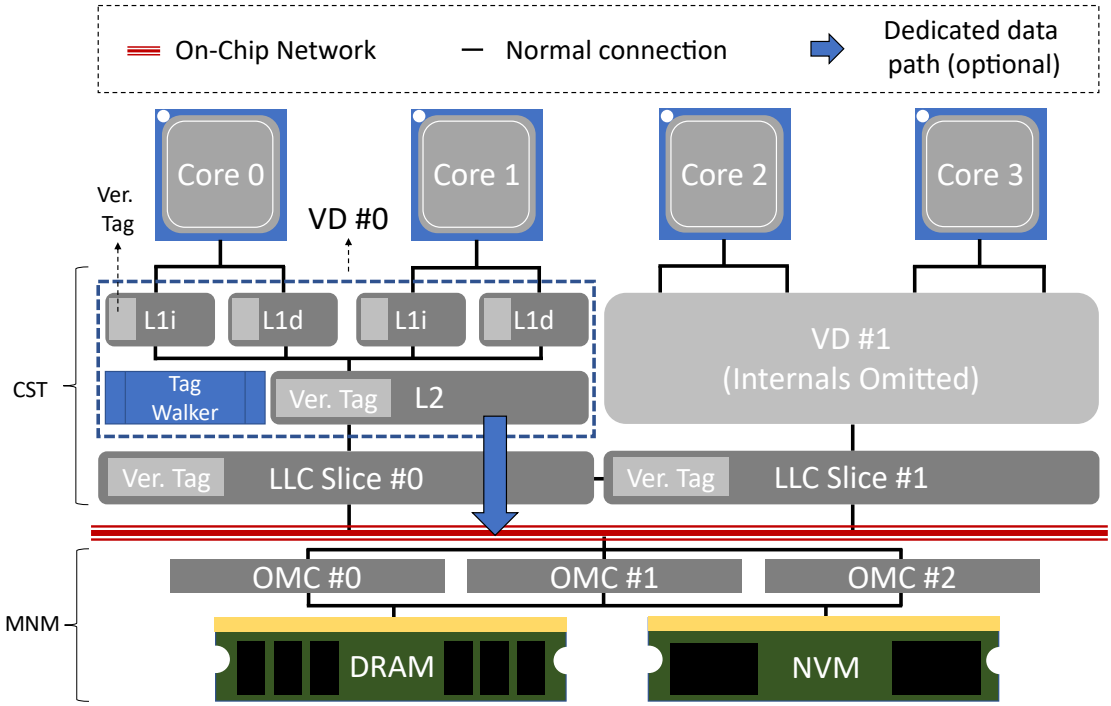



Figure 4.2: **NVOverlay System Architecture** – This figure shows two Versioned Domains (VDs). Each VD consists of a private L2 cache and the two L1 caches that share it.

find a single time point where the snapshot represents the global memory state. Nevertheless, the snapshot still correctly preserves system progress since the image after recovery is consistent in terms of data dependency.

### 4.1.5 Overall Architecture

Fig. 4.2 depicts the overall system architecture. NVOverlay works with any general multicore architecture, even those with partitioned and non-inclusive LLC. The private cache hierarchy can also be shared among a few cores, but we assume that the degree of sharing is small, which is generally the case (e.g., [168]). As with Page Overlays design, all cache tags in the hierarchy are extended with a 16-bit OID field, which we use to store the epoch number in which the block is created.

NVOverlay divides the system into autonomous Versioned Domains (VDs), which we assume are small collections of private caches that are always in the same epoch. In this figure, core 0, core 1, their private L1 caches, and the inclusive L2 cache shared between them form a VD, `VD0`, while the remaining two cores and the private caches form `VD1`. This example, however, is only for demonstration and is by no means the only possibility of forming VDs. In fact, NVOverlay does not restrict how VDs are composed of private caches, except the invariant that all caches in a VD must run the same epoch, and the synchronization overhead of executing local barriers within the VD to enforce this invariant is low. Cache controllers in VDs maintain a `cur-epoch` register for remembering the VD's local epoch and assign it to the OID tags of cache blocks created by overlay writes. Although only four cores and two LLC slices are present in the figure, the actual system can be much larger or even distributed over a network.

Optionally, a dedicated data path can be added between every VD and the memory bus for transferring snapshot data (the blue arrow in the figure). This dedicated data path bypasses the regular data path between the private hierarchy and the memory bus that goes through the LLC. Snapshot data can thus be transferred on the dedicated data path without contending with regular memory requests. This addition, however, is only a performance optimization, and it does not affect how NVOverlay functions. Without the dedicated data path, snapshot data can still be written back to the NVM on the regular data path.

Every L2 cache in the private hierarchy is also equipped with a hardware tag walker (the blue box that juxtaposes the L2 in the figure). The tag walker is a simple hardware addition to the L2 controller, which is crucial to the write-backs of snapshot data.

All VDs in the system plus the dedicated data path constitute NVOverlay's CST component, as shown in the figure.

Snapshot cache blocks evicted from VDs are handled by *Overlay Memory Controllers (OMCs)*, which implements MNM control logic. The OMC maintains a series of mapping tables, which translates the cache block address to the shadow address on the NVM. As shown in the picture, NVOverlay's MNM logic can be distributed over multiple memory controllers for better scalability, each responsible for its assigned address partition. All OMCs and the NVM storage that stores snapshot data constitute the NVOverlay's MNM component.

Lastly, unlike prior works, NVOverlay does not limit the system to be only capable of running on NVM. On the contrary, NVOverlay is compatible with all types of main memory technology—DRAM, NVM, or even a hybrid of the two. This feature gives a great advantage of NVOverlay compared with other related works because applications do not have to be ported to NVM and experience performance drops (and many of them will likely not be ported), while still enjoying the benefits of memory snapshotting.

## 4.2 Capturing Incremental Snapshots with Consistent Snapshot Tracking (CST)

This section presents NVOverlay's Consistent Snapshot Tracking (CST) component. As discussed in earlier sections, the CST component consists of Versioned Domains (VDs) that each runs a local epoch. The CST is responsible for (i) incrementally capturing memory modifications that occur within an epoch; (ii) updating local epochs of VDs on receiving coherence responses from other VDs, and (iii) writing back snapshot data generated by overlay writes, which we call "versions", to the NVM such that they become persisted.

In the following subsections, we discuss the detailed designs of CST that fulfill the above three responsibilities. Our discussion begins with an overview of the multiversioned cache hierarchy and the general design goals of CST. We then elaborate on the protocols that enable the CST to generate snapshot cache blocks and gradually persist them as memory operations are being processed. We next cover coherence handling within and between VDs, and how data dependencies between snapshots are tracked properly with VD's local epochs. We conclude this section with a brief discussion on the tag walker and CST's progress guarantees.

All our discussions assume generic directory-based MESI [197] as the cache coherence


(a) Store (b) L1 Writes Back (c) L2 Writes Back

Figure 4.3: **L1 Store-Eviction** – Only showing block OID. "*" means dirty version. Additions to baseline protocol are marked red (the same applies to the following figures).

protocol. The CST design can also be easily adapted to support snoop-based MESI or its mainstream derivations, such as MOESI [61] or MESIF [241]. We also emphasize that NVOverlay does not modify the coherence protocol. Instead, only a few extra tag checks and evictions are added to existing coherence actions, while states and transitions remain untouched.

### 4.2.1 The Multiversioned Cache Hierarchy

The CST implements a multiversioned cache hierarchy by tagging every block in the hierarchy with an OID, and allowing multiple blocks on the same address but with different OIDs to co-exist. In the multiversioned hierarchy, the term "versions" refer to cache blocks whose contents are produced during epoch execution. A cache block's version number is the value of its OID tag, which is set to the VD's local epoch number when the block is created via "Overlay-on-Write". All memory requests and coherence messages in the system also include a version number, which we denote as *RV (Request/Response Version)*.

A version can be either clean or dirty, depending on its coherence state. For example, in MESI protocol, `M` state blocks are dirty, while `S` and `E` state are clean. NVOverlay maintains the invariant that clean versions are already persistent on NVM. A dirty version from a previous epoch *E'* is, therefore, immutable in epoch *E* since it might be part of the snapshot state of *E'* that has not been persisted.

One of the challenges of designing the multiversioned cache hierarchy is to ensure that only the most up-to-date version is accessed by regular execution, even when multiple versions on the same address with different OIDs co-exist in the hierarchy. The multiversioned hierarchy also needs to guarantee that eventually, all older versions that constitute memory snapshots are written back to the NVM, while the main memory only keeps the most up-to-date working set data. As we will see shortly, the CST addresses this challenge by (i) always storing the most up-to-date version in the L1 cache, while using the L2 cache as a temporary buffer to store snapshot data from an earlier epoch, and (ii) gradually "pushing" versions that belong to older epochs down the hierarchy when new versions are created via overlay writes or when coherence requests cause the most up-to-date version data to leave the VD. The essence of the CST design is that all CST operations can be easily piggybacked on regular cache operations. The CST can thus be implemented elegantly as a simple extension to the existing cache hardware without radical change.

### 4.2.2 L1 Operations

**On receiving a load request** from the processor, the L1 executes it as a regular load operation. In particular, the tag lookup is performed without checking the OID tag, as opposed to an overlay read operation where the tag lookup logic compares both the address tag and the OID tag. If the lookup misses, a `GETS` request is sent to L2, and the load request is inserted into the MSHR for a later retry. Otherwise, the load completes locally and responds with block data.

**On receiving a store request** from the processor, a tag lookup is performed as in loads. The store request's `RV` is always the VD's `cur-epoch`. If tag lookup signals a miss, or if the block is not in a writable state (`E` or `M`), the cache controller will first acquire exclusive permission by sending `GETX` to the lower level with the same `RV`. Otherwise, the block OID is compared with `RV`. The store completes locally if the block is dirty and its OID equals `RV`.

In the case where the block's OID does not equal `RV`, indicating that the block holds data generated in a previous epoch and therefore belongs to that epoch's memory snapshot, the store must be executed as an overlay write such that snapshot data is not overwritten. More specifically, the L1 controller performs the following three steps, which we call "store-eviction". First, it performs an overlay write by duplicating the existing block and setting the new block's OID to `RV`. Second, the controller then evicts the existing block to L2, so the L1 only caches the most up-to-date working set data. If the L2 already contains an older version on the same address, that older version will be evicted from the L2 to the OMC as the L2 processes the eviction from the L1. Finally, the store is applied to the newly created overlay block, and the block is marked as dirty. All future writes on this address within the epoch will land on this overlay block as regular writes.

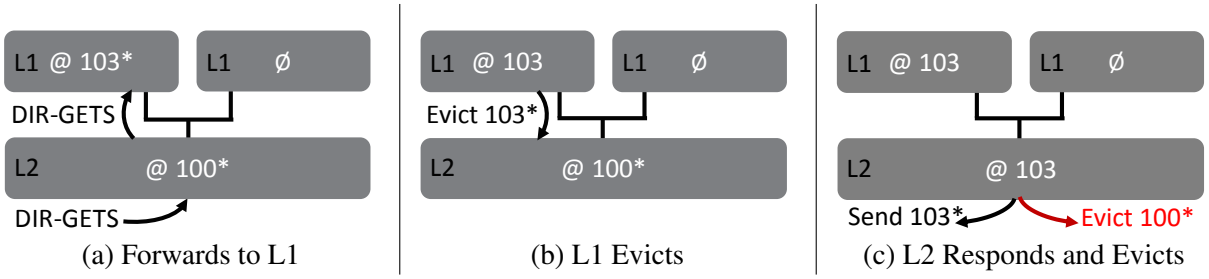
Fig. 4.3 depicts an example of store-eviction. In this example, both the L1 and the L2 already contain dirty blocks on the requested address with OID 103 and 100, respectively. When a store to the same address is executed in epoch 104, the L1 controller will figure out that the store must be executed as an overlay write. The overlay write is executed by first duplicating the block in L1, then evicting the block, and finally performing the write on the newly created block whose OID tag is assigned the write request's *RV*, which is 104. In this particular case, the L2 version 100 will also be "pushed out" when the eviction from the L1 is processed, such that only a single version is cached in both levels.

**On block eviction**, if the block is dirty, a `PUTX` request is scheduled in L1's evict buffer, with `RV` set to block OID. Whether clean evictions are processed is implementation-dependent, and CST works properly with both options. Since cache block evictions are not on the critical path, store-eviction will not affect the L1 cache access latency.

### 4.2.3 L2 Operations

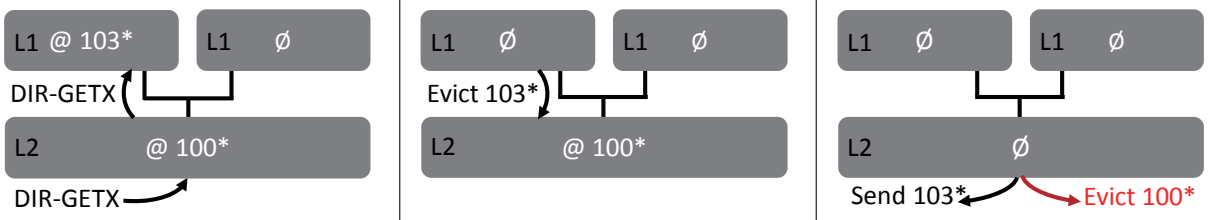
**On receiving a `GETS` or `GETX`** from L1, the L2 performs a tag lookup as in a regular cache and signals a miss if the block is not present or has insufficient permission. Otherwise, the requested block in the L2 is read out and sent to the L1 as the response. The *RV* of the response message is set to the block's OID.

**On receiving a `PUTX`** from L1, the L2 first performs a tag lookup to read out the block OID and coherence state. If the block is dirty, and $OID < RV$, then the L2 will first evict the existing block to the OMC, because the existing block belongs to the memory snapshot of an earlier epoch.




(a) Forwards to L1 (b) L1 Evicts (c) L2 Responds and Evicts

Figure 4.4: **L2 External Downgrade** – The newer version is sent as the response, while the older version is written back to the OMC. After the downgrade, both caches hold a shared copy of the most recent version.




(a) Forwards to L1 (b) L1 Evicts (c) L2 Responds and Evicts

Figure 4.5: **L2 External Invalidation** – The newer version is sent as the response, while the older version is written back to the OMC. After invalidation, none of the caches holds any version.

This cascaded eviction guarantees that at most one version per address is maintained in every cache, which keeps the CST design compatible with the existing coherence protocol. In all cases, the L2 completes the request by inserting evicted data into the cache slot and setting its OID tag to the value of `RV` contained in the eviction message.

**On block eviction**, both the L2 version and the L1 version will be purged from the private hierarchy in order to maintain inclusiveness. If the L1 version is dirty, it will be sent to the LLC because the L1 version is always up-to-date and it constitutes working set data. The L2 version, however, is written back to the OMC for persistence if it is dirty. The `RV` of the write-back message is set to the OID of the block in the L2 cache.

### 4.2.4 External Invalidation and Downgrade

External invalidation and downgrade refer to coherence requests received by a private hierarchy which is sent from the LLC's directory controller. The main challenge here is that the L1 and L2 caches may each cache a dirty version. The CST solves this with extra evictions, as shown below.

**On receiving an external invalidation (`DIR-GETX`) or downgrade (`DIR-GETS`)**, the L2 controller first queries its directory for any L1 sharer (recall that we assume that L2 can be shared by a small number of L1 caches) on the requested address. If there is none, then the L2 already contains the newest version of the address, which is sent back as the response.

However, if there are L1 sharers, newer versions may exist in these sharers. To obtain the newer version, the L2 controller first forwards the request to the L1 sharers of the requested address. The L1 controller will satisfy the request by writing its newer version back to the L2 using `PUTX`. The L2 controller then processes the `PUTX` message as described in Section 4.2.3. In this case, the L2 cache will send the version from the L1 back to the directory as the response, with the message's `RV` set to the OID of the block from the L1. Block state transitions are also

performed just like in a regular cache.

An example of external downgrade is given in Fig. 4.4. In this example, the L1 and L2 cache contain dirty versions on OID 103 and 100, respectively. When the L2 cache receives an external downgrade, it checks the sharers vector on the requested address and finds out that one of the two L1 caches may have a newer version than its locally cached version. As a result, the L2 cache forwards the downgrade message to the L1 sharer, which handles the message by writing back version 103. On receiving the write-back, the L2 cache writes back its locally cached version 100 to the OMC for persistence and sends version 103 from the L1 back as the response. Both versions are "un-dirtied" after handling the coherence downgrade. An example of coherence invalidation is given in Fig. 4.5. Coherence invalidations are processed similarly, except that versions are invalidated, instead of un-dirtied, after the request is handled. As shown by this example, neither the L1 nor the L2 cache holds a copy of the address on any version on the final state.

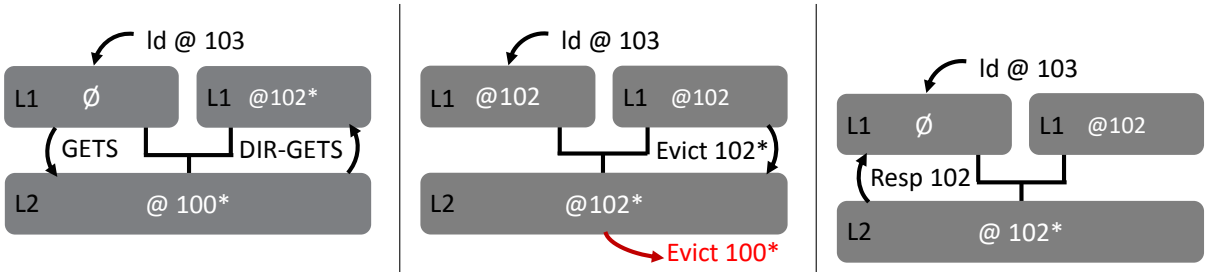
### 4.2.5 LLC and Main Memory Operations

In NVOverlay, both the LLC and the main memory only store the current working set data. Versions that belong to previous epochs are never written back from the private hierarchy into the LLC or the main memory. Instead, these versions will be sent directly from the L2 cache to the OMC.

When a version that belongs to the current working set is written back from the private hierarchy (checked by comparing its OID tag to the `cur-epoch` of the VD), the version will be inserted into the LLC. The OID tag of the block is set to be the `RV` included in the write-back message. When a block is fetched by the private hierarchy using `GETS` or `GETX`, the fetched request is processed normally as in a regular cache, and the `RV` in the response message is set to the OID tag of the block being fetched.

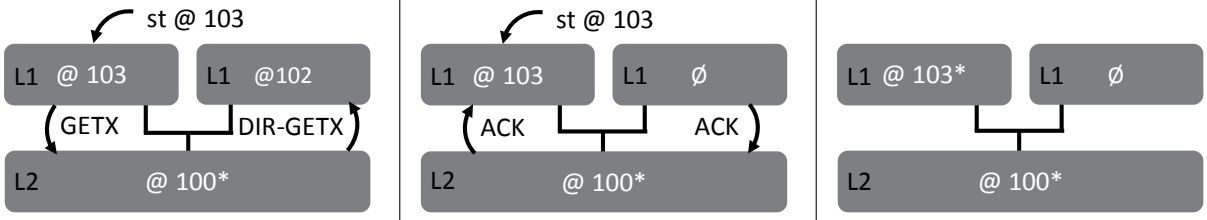
Operations on the main memory resemble those on the LLC, except that the main memory does not natively provision per-block OID tags to track the epoch to which a block belongs. To maintain per-block OID tags, the main memory controller should reserve extra metadata storage that maintain the OID tags for all physical blocks in the main memory. The metadata storage is accessed in parallel with data when a request is processed on the main memory. For writes, the per-block OID tag is updated using the `RV` value included in the write-back message from the cache hierarchy.

NVOverlay does not restrict the implementation of main memory metadata storage for OID tags. In the simplest case, the main memory controller just reserves some storage at system startup time as the metadata storage, which incurs 3.2% storage overhead (16-bit OID per 64-byte block). There has also already been an abundance of prior works that focus on this particular matter. For example, on ECC-enabled main memory modules, the OID tags can just be stored in the ECC banks, which trades off the degree of protection provided by ECC bits with memory snapshotting functionality. Other techniques, such as DRAM compression [228], can also be employed to embed the OID tags within data pages without extra storage overhead.




(a) L2 Sends Downgrade (b) L2 Evicts (c) L2 Forwards Version

Figure 4.6: **Intra-VD Downgrade** – The write-back of a newer version from the L1 cache will cause the older version in the L2 to be written back.




(a) L2 Sends Invalidiation (b) L1 ACK (c) Store Completes

Figure 4.7: **Intra-VD Invalidation** – In this example, no write-back occurs, despite a newer version being cached on the L1 because the newer version is clean.

### 4.2.6 Intra- and Inter-VD Coherence

**Intra-VD coherence.** Recall from Section 4.1.5 that a VD may consist of a few processors sharing the same L2 cache. In this scenario, these processors may generate coherence requests internal to the VD, which we call intra-VD coherence.

When coherence requests occur within a VD, the handling is rather straightforward and does not deviate much from those in a regular hierarchy. An example of coherence downgrade from one processor to another is given in Fig. 4.6. In this example, one of the two L1 caches and the L2 cache contains version 102 and 100, respectively. Then a load request on the other L1 cache causes a coherence downgrade to be sent from the L2 controller to the L1 that caches version 100. The coherence downgrade is handled in three steps. First, the L1 cache writes back version 102 to the L2 cache while keeping a clean copy of version 102. Second, the write-back message is processed by the L2, which further causes the L2's locally cached version, version 100, to be written back to the OMC. Finally, the downgrade response is sent back to the requestor, and the block is inserted into the requestor cache with OID being 102 (i.e., the most up-to-date version number).

Another example that depicts intra-VD coherence invalidation is given in Fig. 4.6, where a processor executes a write on a read-only block and misses, causing the shared L2 cache to issue a coherence invalidation to the other L1 cache. This example differs from the previous one in that the version to be invalidated is clean. Therefore, when the L1 cache handles the coherence invalidation forwarded by the L2, the version is just invalidated without writing back dirty data. On receiving the acknowledgment (ACK) from the L1, the L2 cache simply forwards the ACK message to the requestor. The requestor L1 cache then proceeds to execute the write as a regular write because the block's OID equals the epoch number.

**Inter-VD coherence.** Inter-VD coherence occurs when a request issued by one of the processors in the VD cannot be fulfilled locally (either due to cache misses or insufficient permission). In this case, the VD sends a coherence request to the LLC, which turns the request into an inter-VD coherence request.

Inter-VD coherence requests are no different from normal requests and are handled similarly. The only exception to handling inter-VD coherence request is that data dependency between the current local epoch and a future epoch in the sense of logical time should be checked upon receiving a block via coherence. This check is performed by comparing whether the `RV` in the coherence message is larger than the VD's `cur-epoch`. If data dependency truly forms, it should be observed by advancing the VD's `cur-epoch` to the value of `RV`.

In order to advance the `cur-epoch`, the L2 controller first signals all cores in the VD to execute a local barrier. The L2 cache controller also stops responding to external coherence requests and drains the intra-VD request queue. Next, all cores in the VD dump their non-speculative context to the NVM, tagged with the VD's *cur-epoch*. Finally, the *cur-epoch* registers in all cache controllers are updated to *RV*, after which the barrier is lifted, and execution resumes.

In practice, VDs also advance their local epochs after a fixed number of instructions to avoid significant epoch skews between VDs. As we will see later in Section 4.2.8, significant epoch skews will become problematic when epoch numbers (represented by 16-bit OIDs) wrap around.

### 4.2.7 Cache Tag Walker

The last component of CST is the cache tag walker built into the L2 cache controller logic. The cache tag walker is a hardware state machine that scans cache tags opportunistically (i.e., its request has lower priority than requests from the upper-level cache), in the background, whose operation is overlapped with regular execution. As shown in Fig. 4.2, every VD, which also happens to contain a single L2 cache, has a tag walker, and all tag walkers in the system operate independently from each other.

The responsibility of the cache tag walker is to ensure overall forward snapshotting progress, a critical property that is not guaranteed by the snapshotting protocol alone. Image if an address $A$ is written once during local epoch $E$, creating an overlay block whose OID tag has value $E$ and is never written again after the local epoch has advanced to a later epoch $E'$. In this scenario, the block belongs to the snapshot of epoch $E$ but may potentially never be written back from the VD (e.g., in a pathological case, a block is repeatedly read in the same VD that created it, which keeps refreshing its replacement status and prevents it from eviction). As a result, the memory snapshot of epoch $E$ on the NVM will remain incomplete indefinitely because the block that stores snapshot data on address $A$ is never written back.

The cache tag walker operates as follows. The cache controller periodically brings up the cache tag walker by scanning tag array entries, and for each entry, checking whether it is dirty and contains an overlay block whose OID tag is smaller than the VD's `cur-epoch`. If such a block is found, the cache tag walker will first write back the block content to the OMC, and then un-dirties the block such that it is ignored by future tag walks.

Note that the tag walker only operates on the L2 cache and is hence unable to deal with dirty versions being withheld indefinitely in the L1 cache. To prevent this pathological case from happening, we propose an optional "load-eviction" mechanism, in which the L1 cache controller

will evict a block if the block is dirty and its OID is smaller than the VD's `cur-epoch`. The block is also un-dirtied after the eviction is made. The L1 load-eviction mechanism does not require adding any additional state machine to the L1 cache. Instead, this mechanism simply drives forward the snapshotting progress on the L1 cache using the memory access stream from the processor.

### 4.2.8 Discussion

**Epoch wraparound.** Conceptually, epoch numbers should monotonically increase, since they represent logical time in the system delimiting snapshot boundaries. In reality, epoch numbers are represented by the 16-bit OID and will wrap around to zero eventually. The wraparound will cause NVOverlay to malfunction because it essentially resets the logical time.

The most straightforward approach to deal with wraparounds would be to flush the entire cache hierarchy and then clear all local epochs such that the system starts over from time zero. This approach requires a global barrier and decreases performance considerably due to the cache flush.

The second and more lightweight solution does not require a global reset but limits the maximum skew among VDs' epoch numbers to half the OID space, i.e., 32K. We partition the OID space into two equally sized groups, `L` and `U`, and keep recycling the half of the OID space when all local epochs leave it. To make such recycling possible, the CST enforces an invariant that either all epochs in `U` is logically ahead of all epochs in `L` (shortened as $\texttt{U} > \texttt{L}$), or vice versa ($\texttt{L} > \texttt{U}$). The invariant can be enforced by fast-forwarding epochs that are "lagging behind" to a more recent epoch, and writing back cache blocks in those epochs to the NVM (which the cache tag walker already implements). We also add a persistent `epoch-sense` bit on the OMC, which indicates whether $\texttt{U} > \texttt{L}$ or $\texttt{L} > \texttt{U}$ (in-group ordering of epoch numbers is unchanged). Whenever a VD is about to advance its local epoch from one group to the other, the VD sends a request to the OMC to flip the `epoch-sense` bit and resets its local `cur-epoch` to zero.

The OID comparison logic is also slightly changed, taking the `epoch-sense` bit as an input. If the `epoch-sense` bit indicates $\texttt{L} > \texttt{U}$, then all epochs numbers in group `L` will be regarded as larger than all epoch numbers in `U`. Otherwise, all numbers in group `L` are larger than those in `U`.

**Reducing NVM writes:** One potential problem of writing back snapshot data from the L2 cache to the OMC is that if an address is under heavy read-write or write-write contention, the block may undergo frequent downgrade or invalidation from the private hierarchy. Each of these operations will generate a write-back to the OMC. The frequent write-backs are detrimental to performance because each write-back consumes extra bandwidth by sending a 64-byte block across the bus, and frequent writes can shorten NVM lifetime.

To reduce the performance impact of frequently written back blocks, we propose adding a battery-backed, write-back buffer to the OMC to absorb the write traffic generated by CST. The OMC buffer operates as a persistent LLC for snapshot data. Blocks that are written back from the cache hierarchy will be inserted into the OMC buffer, and blocks evicted from the OMC buffer will be written to the NVM. Blocks that are written back frequently will likely just hit the OMC buffer without being sent to the NVM for persistence, thus saving one NVM write. Note that the OMC buffer only stores snapshot data, and it is by no means on any of the critical paths of normal execution. The OMC buffer can hence be optimized for writes and trades it off with slower

read operation. The persistent OMC buffer will be flushed back to the NVM to avoid losing any snapshot data on a power failure.

## 4.3 Organizing Snapshot Data with Multi-snapshot NVM Mapping (MNM)

The Multi-snapshot NVM Mapping (MNM) component manages snapshot data written back from the cache hierarchy, and organizes them on the NVM such that they can be retrieved later. The MNM achieves this goal with a two-level hierarchy of hardware-managed mapping tables. First, MNM maintains per-epoch tables that index snapshot data generated with that epoch. The per-epoch table is updated when snapshot data is written back from the cache hierarchy. Second, the MNM also maintains a global *master mapping table*, $M_{master}$, that manages the current *recoverable snapshot image*. When an epoch completes globally, the master table is updated to commit snapshot data blocks from that epoch to the recoverable snapshot image.

The MNM is also responsible for computing the *recoverable epoch*, $E_r$, which is the most recently completed epoch whose snapshot data has been entirely written back from the cache hierarchy to the NVM. This task must be performed in a distributed manner because each VD operates under its local epoch, and tag walkers on every L2 cache work independently from each other.

In the rest of this section, we discuss the design of MNM. We begin by elaborating the mapping table hierarchy in detail and proceed to cover how the recoverable epoch is computed. We then cover garbage collection, an essential part of the design that frees up storage when they are no longer referred to by any of the mapping tables. We conclude this section with discussions on how to retrieve snapshot data in a variety of scenarios.

### 4.3.1 Per-Epoch and Master Mapping Tables

For each epoch `E`, the MNM maintains a per-epoch mapping table $M_E$ which contains address-to-data relations for data generated within that epoch. Provided with a physical address, the mapping table will either return the location on the NVM where snapshot data is stored, or `NULL`, indicating that snapshot data does not exist (i.e., the address is not written to during the epoch). The per-epoch mapping table can be maintained in volatile storage (if the system has one) to reduce NVM writes and will be permanently lost on power failures.

When a block belonging to a snapshot is written back from the cache hierarchy, the MNM first persists block data to a *snapshot data store* residing in the NVM, and then inserts a mapping entry into the per-epoch table such that block data can be found later using its address. Recall that write-back messages from the hierarchy carry an `RV` field storing the OID of the block when it was cached in the hierarchy; the per-epoch mapping table is located using the `RV` field to query a table that maps epoch numbers to table roots.

The MNM extends the existing OMT and OMS mechanisms from Page Overlays to implement per-epoch mapping tables and snapshot data store, respectively. In the context of NVOverlay, the OMS manages storage for snapshot data by allocating free pages and using them to store snapshot

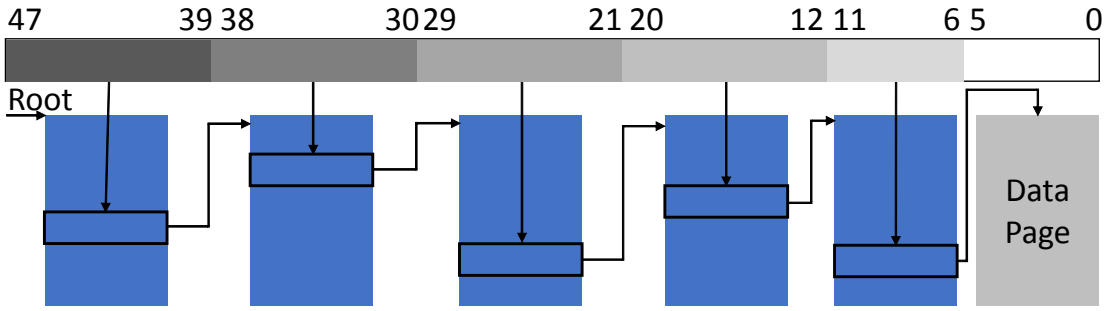


Figure 4.8: **Master Mapping Table** – Shows five-level radix tree structure. Blue and grey represent memory pages allocated to the radix tree and data.

blocks compactly. At the same time, the OMT translates the physical addresses of evicted blocks to their storage locations in the OMS. In the hardware implementation, both the OMS and OMT can be specifically optimized for writes because snapshot data will not be accessed during normal execution.

Besides per-epoch volatile mapping tables, the MNM also maintains a master mapping table, $M_{master}$, which does not belong to any particular epoch. $M_{master}$ maps the current recoverable snapshot image. A recoverable snapshot image refers to the most up-to-date, consistent memory snapshot that can be used for failure recovery, which consists of data blocks from completed epochs. The $M_{master}$ is implemented as a *five-level* radix tree which is similar to the virtual memory page table on newer platforms [108]. The first four levels of $M_{master}$ are identical to today's four-level page table, and the last level is indexed by address bit 6–11 for cache block granularity mapping, as shown in Fig. 4.8.

Similar to per-epoch mapping tables, the $M_{master}$ maps physical addresses to block locations on the snapshot data store. The $M_{master}$ is atomically updated when an epoch is ready to be committed to the recoverable image. The update procedure merges the per-epoch table into $M_{master}$ in a way that resembles Log-Structured Merge (LSM) Trees [195]. The procedure scans the leaf level entries of the per-epoch mapping table, and for each entry that maps a data block, inserts the data block's physical address tag and NVM storage location into $M_{master}$. This process incurs significantly lower write amplification than logging because it only modifies $M_{master}$'s radix tree pages without copying data blocks.

To guarantee the atomicity of $M_{master}$ updates, the MNM may adopt logging or use persistent buffers [92] to batch-commit modifications on the table. Note that logging will increase write amplification on table update operations. However, the updates are marginal compared with the total amount of data because mapping entry update only constitutes a small fraction of all NVM writes the MNM conducts. Alternatively, NVOverlay may just maintain per-epoch tables in the NVM and rebuild $M_{master}$ during crash recovery. The atomicity of updating per-epoch tables can be guarded using a single bit which is set atomically when the epoch completes. This approach trades off the complication of maintaining $M_{master}$ for increased recovery time and, as a result, decreased system availability. However, it may become feasible if high availability is not a concern, e.g., employing failure recovery for preserving computation progress or software debugging.

### 4.3.2 Computing The Recoverable Epoch

In NVOverlay, an epoch `E` becomes a recoverable epoch when it is ready to be committed to the recoverable snapshot image. The name "recoverable epoch" suggests that the epoch is consistent and can hence be used as a restoration point for failure recovery. While many recoverable epochs may exist in a system, NVOverlay always strives to restore system states to the most recent recoverable epoch to minimize lost progress.

Two conditions must be satisfied for `E` to become a recoverable epoch. First, all earlier epochs before `E` must have also become recoverable. Second, all snapshot data generated within `E` must have been written back to the NVM. Both conditions can be checked by reaching a consensus between all VDs regarding the globally minimum completed epoch in a distributed manner.

In order to reach a consensus, all VDs work together to derive the minimum epoch that still has a non-persist block. To serve this purpose, we add a register, `min-ver`, to each tag walker on the L2 cache. The register is initialized to `cur-epoch` when the tag walk begins and updated to the smallest OID tag encountered during the tag walk. The L2 cache controller periodically sends the value of `min-ver` and `cur-epoch` to the OMC in an *epoch consensus* message. The OMC maintains an array of the most recently received epoch consensus for each VD in the system (if a VD consists of more than two tag walkers, then multiple entries are maintained for that VD). On receiving the epoch consensus message, the OMC recomputes the recoverable epoch, $E_r$, by taking the minimum over all `min-ver` and `cur-epoch` values it has received. After $E_r$ is computed, it is sent to the MNM, and, as described earlier in Section 4.3.1, the MNM commits all snapshot data till $E_r$ to $M_{master}$, progressing the recoverable snapshot image to logical time $E_r$.

If multiple OMCs are present, each OMC first computes the per-OMC $E_r$ for VDs that it is responsible for, and then one of them is selected as the coordinator, to which all the remaining OMCs send their own $E_r$s. The coordinator OMC notifies the final result to $M_{master}$ after computing the smallest $E_r$ among all per-OMC $E_r$ it has received.

### 4.3.3 Garbage Collection

When committing a recoverable epoch $E_r$, snapshot data blocks that are unmapped from $M_{master}$ by mapping entry updates will become stale and are left for garbage collection (GC). We cannot, however, immediately reclaim their storage since snapshot data blocks are managed by the OMS and stored in the granularity of OMS pages.

The MNM, therefore, performs GC in page granularity. When a block is unmapped from $M_{master}$, the populace of the containing OMS page is checked (the OMS tracks per-page status using a bitmap and per-page reference counts in volatile memory). If the populace drops to zero, then the OMS page can be freed and added to OMS's free page pool. Otherwise, the OMS only updates the page's metadata to reflect that a block has been removed from the page.

One potential problem with the above GC strategy is that one single snapshot data block from an old epoch is sufficient to prevent the entire OMS page from being GC'ed, which lessens NVOverlay's storage efficiency. Although this occurs relatively rarely and only on addresses that are only written once and never written again, it might eventually exhaust all available NVM storage to the OMS and prevent NVOverlay from making forward progress. To address this problem, the MNM may adopt one of the two following strategies when a few blocks prevent a

page from being GC'ed. First, the MNM may copy these blocks over to a more recent page on the same address and then free the page belonging to the old epoch. This approach resembles data compaction in log-structured storage [103, 217, 218], and may incur slightly higher write amplification due to the data copy. Second, the MNM can reduce the storage overhead by managing storage in smaller granularity. For example, if the OMS allocates pages in 256 bytes, a worse-case $4\times$ storage overhead is expected. In reality, the storage overhead brought by the inability to perform GC is much smaller because of the locality of write operations.

Garbage collection on per-epoch mapping tables occurs instantly after the corresponding epoch has been committed. The pages used by the tables are just freed and retired into a free buffer pool for future allocations.

### 4.3.4 Retrieving Snapshot Data

After a crash, the OS or firmware first restores the system to the initial state when snapshotting was started. The recovery procedure then loads the recoverable snapshot image from the NVM by scanning $M_{master}$ and reading all snapshot data blocks into their corresponding addresses in the main memory. After this completes, the processor contexts that were dumped to the NVM at the end of the recoverable epoch is loaded back to the processors, after which the system resumes execution as if the crash had never happened.

For remote replication, snapshot data is transferred to a remote backup machine via the network. On receiving snapshot data, the remote machine can either replay them as redo logs or archive them into secondary storage for permanent archiving. Previous works [274] have also demonstrated how memory snapshotting can work with remote replication in detail.

NVOverlay's memory snapshots also support live accesses even when the system is still running. This feature is highly compelling as it grants debuggers or memory tracers to "time-travel" back to a previous memory state. To enable such a use case, we extend the ISA of the processor to allow programmers to programmatically start new epochs and access snapshot data on a specific address of an epoch. If an access to address `X` on epoch `E` is made, the MNM must find the largest `E'`, `E'` $\leq$ `E`, where the mapping table of `E'` has an entry for address `X`, and then reads snapshot data from the NVM using a storage location in the mapping entry. This "fall-through" read semantics is similar to how version chains are accessed in an MVCC database [62, 142, 187, 258], since snapshots are saved incrementally.

### 4.3.5 Putting It All Together

Fig. 4.9 depicts an overall architecture of the Multi-snapshot NVM Mapping (MNM). In this depiction, the MNM manages both DRAM volatile storage (left side) and NVM persistent storage (right side). The NVM storage is further divided into those dedicated to OMS pages (bottom right, with OMS pages), and those that store the master mapping table $M_{master}$ (top right, with radix tree nodes). On the volatile DRAM side, MNM maintains a series of per-epoch mapping tables (top left) and two structures (bitmap and reference count array) that assist the OMS in tracking page occupancy (bottom left).

The figure also shows the general data flow on the MNM. When a snapshot data block in epoch `E` is written back from the cache hierarchy, the block is persisted to the NVM on one of the

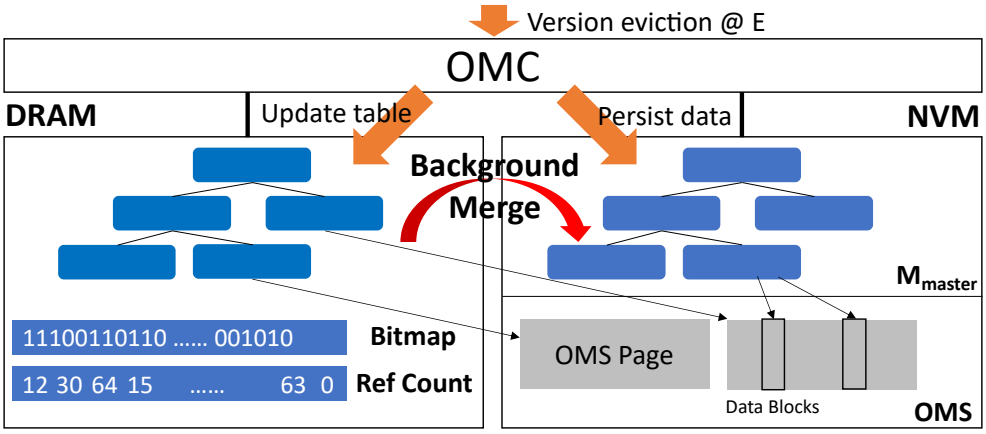



Figure 4.9: **Multi-snapshot NVM Mapping** – Orange arrow represents data flow on L2 write-backs. Red arrow represents background merge after *E* becomes a recoverable epoch. Blue and grey blocks represent metadata and data, respectively.

OMS pages. At the same time, the mapping entry in the per-epoch mapping table is also updated to point to the storage location of the block on the NVM. When an epoch becomes a recoverable epoch, its mapping table is merged into $M_{master}$ by copying over the mapping entries from the leaf level to $M_{master}$. After the merge, the $M_{master}$ entries also point to the NVM storage location of snapshot data blocks, as shown in the figure.

## 4.4 Evaluation

### 4.4.1 Simulation Configuration

**Simulation platform:** Our evaluation uses zsim [219], a Pin-based [164] simulator featuring fast, cycle-accurate multicore simulation. We implemented NVOverlay as a separate module without changing the existing coherence protocol. Table 4.1 presents the configuration of the simulated system.

| | |
|---|---|
| Processor | 16 cores 4-way Our-of-Order @ 3GHz |
| L1-d/i caches | 32KB, 64B lines, 8-way, 4 cycles |
| L2 cache | 256KB, 64B lines, 8-way, 8 cycles |
| Shared LLC | 32MB, 64B lines, 16-way, 30 cycles |
| DRAM | DDR3 1333 MHz, 4 controllers |
| NVM | 16 banks, 133 ns write latency |

Table 4.1: **Simulation Configuration**

**Simulated snapshotting designs:** We compare NVOverlay to five other epoch-based memory snapshotting designs besides the baseline system: (i) Software Undo Logging ("SW Logging"), (ii) Software Shadow Paging ("SW Shadow"), (iii) Hardware Shadow Paging ("HW Shadow"), (iv) PiCL [189], and (v) PiCL running at L2 level (PiCL-L2). We briefly describe these designs as follows.

- **Software Logging:** Software generates and flushes an undo log entry before the first write to a block address since the current snapshot begins. We optimistically assume that the software library tracks the write set with no overhead and flushes them to the NVM at the end of an epoch. All NVM writes use persist barriers, which stall the processor until the write completes.
- **Software Shadow:** Software tracks the write set and flushes dirty blocks in the write set back at the end of each epoch. The software also maintains a persistent mapping table, which is updated at the end of an epoch. All NVM writes use persist barriers.
- **Hardware Shadow:** We model hardware shadow paging using a three-version, cache block granularity shadow scheme similar to ThyNVM [214]. This design works similarly to software shadow, except that hardware can overlap the persistence of the previous epoch with the execution of the next epoch. However, the centralized mapping table is updated synchronously, which stalls the processor at epoch boundaries.
- **PiCL:** Implements hardware undo logging. The design maintains a circular log buffer on the NVM. The cache controller generates an undo log entry and flushes that entry back to the NVM when a block is first time written since the epoch begins. Hardware tracks dirty blocks generated within an epoch with an inclusive LLC extended with per-block version tags that store the current epoch number. A hardware tag walker on the LLC evicts dirty blocks from previous epochs when the epoch completes. Although PiCL requires executing the global barrier to advance the epoch, our simulation ignores the overhead of the global barrier and only focuses on the data path (latency, bandwidth, and write amplification).
- **PiCL-L2:** A hypothetical design that functions the same as PiCL, except that the write set tracking and tag walks are performed at the L2 level instead of the LLC. We use this design to estimate the performance of PiCL-style undo logging on a large multicore system without a monolithic and inclusive LLC.

For fairness of comparison, we assume all simulated designs, including the baseline, are equipped with a write-back DRAM buffer whose size can accommodate the entire working set. Note that this assumption does not give NVOverlay any unique advantage over the other designs because NVOverlay is meant to work on DRAM-NVM hybrid systems where working set data is maintained in the DRAM. For our main experiments, the epoch size is set to 1M store uops. Both PiCL and NVOverlay initiate tag walks immediately (also known as *Asynchronous Cache Scan* in PiCL's terminology) after an epoch completes.

**Benchmarks:** We use benchmarks from the STAMP [173] suite and a set of data structure benchmarks for our evaluation. STAMP consists of memory-intensive applications that stress the data path and its transactional multicore synchronization model stresses the modified coherence protocol.

The data structure benchmarks consist of BTreeOLC [151], ARTOLC [148], red-black tree (`std::map`) and hash table (`std::unordered_map`). They represent workloads with large working sets. We run an insert-only workload with random keys to mimic bulk insertion into a database index.

All benchmarks spawn 16 worker threads, and are compiled with locks except BTreeOLC and ARTOLC. Threads execute until either the end of the program, or 100M instructions per-thread

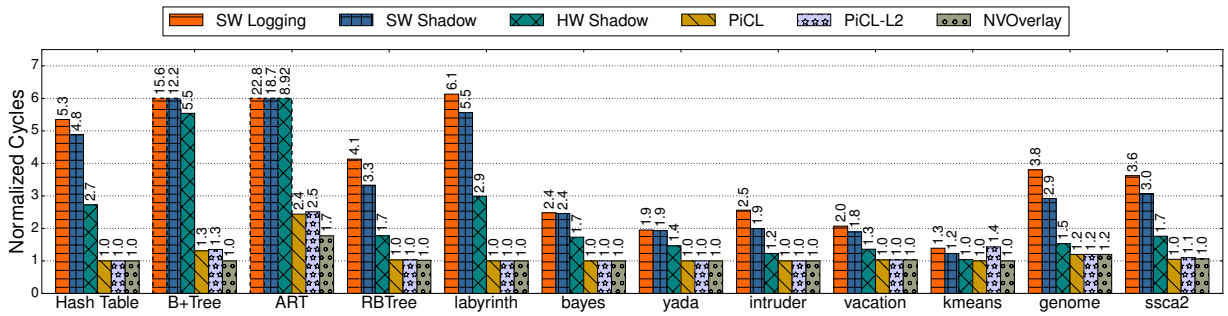



Figure 4.10: **Normalized Cycles** – 16 worker threads. All numbers are normalized to baseline execution without snapshotting.

are reached, totalling 1.6B total instructions.

### 4.4.2 Performance Analysis

**Cycle overhead:** Fig. 4.10 shows the wall-clock cycles for each workload. The results are normalized to a regular system executing the workloads without memory snapshotting. For 9 out of 12 workloads, both NVOverlay and PiCL can fully overlap execution with snapshotting, incurring no cycle overhead. These results are also consistent with the original PiCL paper [189], in which PiCL was shown to perfectly overlap execution with persistence.

PiCL-L2, on the other hand, suffers from slightly slower execution due to the smaller on-chip working set, which results in excessive evictions and log writes from the hierarchy. For `ssca2` and `kmeans`, PiCL-L2 runs 10% and 40% slower compared with NVOverlay and PiCL.

SW Logging and SW Shadow are both considerably slower than NVOverlay. The poor performance is a natural consequence of using persist barriers and having to stall the processor for the write operation to complete.

HW Shadow is moderately slower than NVOverlay. In all but `B+Tree` and `ART`, HW Shadow is at most 3× slower than NVOverlay. We attribute such performance characteristics to its ability to overlap data persistence with execution. It has to, however, synchronously update the mapping table at the end of an epoch to avoid corrupting it in the next epoch, resulting in slowdowns.

Note that, theoretically speaking, SW Logging can only double the number of writes in the worst case compared with OverlayTM, due to having to persist both log entries and data blocks. However, in many cases, the performance gap between NVOverlay and SW Logging is often far larger than 2×. The explanation is that SW Logging tends to generate large bursts of writes when generating log entries and at the end of an epoch, which incurs severe bus contention and results in even lower performance than if those writes were conducted contention-free. By contrast, NVOverlay is far better at handling and distributing bursts of writes because it generates write-backs to the NVM during execution as new snapshot blocks are created via overlay writes, and as coherence messages are being processed. Such performance characteristics benefit workloads that generate a large number of writes, such as B+Trees where insert operations will shift existing elements on a B+Tree node to maintain in-node key order. To confirm it: our evaluation shows that in `B+Tree` workload, out of 11,778,311 total NVM data write requests, 11,503,974 (97.7%) of them are generated by the coherence protocol. NVOverlay evenly distributes these writes across the execution, hence reducing memory bus contention.

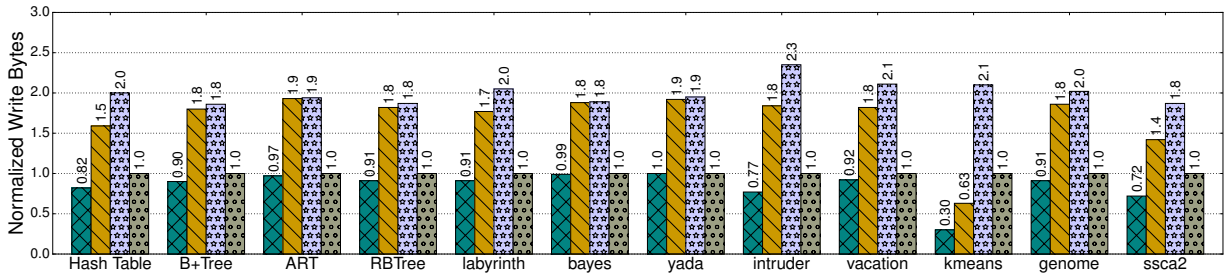



Figure 4.11: **Write Amplification (Bytes of Data)** – 16 worker threads. All numbers are normalized to NVOverlay.

**Write amplification:** Fig. 4.11 shows the results of write amplification in terms of bytes written to the NVM device. We measure both data and metadata writes. For PiCL and PiCL-L2, we assume each log entry takes 72 bytes (64 bytes data + 8 bytes address tag). For HW Shadow and NVOverlay, we measure the number of eight-byte writes performed on the radix tree mapping table as metadata writes. All write amplification numbers are normalized to NVOverlay.

Both HW Shadow and NVOverlay demonstrate lower write amplification than logging. This result is expected since PiCL and PiCL-L2 need to write two whole cache blocks for each cache eviction instead of one. Overall, PiCL writes 1.4×–1.9× more data than NVOverlay. For PiCL-L2, write amplification is higher, ranging between 1.8×–2.3×. This result is caused by the smaller on-chip working set and more frequent block evictions that follows, as we have noted in the previous section.

In 5 out of 12 workloads, NVOverlay incurs more than 10% writes than Shadow Paging. This observation can be explained by the fact that NVOverlay will issue a write-back to the OMC when a snapshot data block is evicted from the L2 cache. The negative impact of such write-backs can become significant, as we see in `kmeans`, where 70% fewer writes are performed on NVM for HW Shadow and 37% fewer for PiCL. The write amplification of PiCL-L2 also proves this point, as PiCL-L2 issues 2× more writes to the NVM compared with NVOverlay.

Further studies of `kmeans` reveal that only 896,837 writes are issued from the LLC when simulating HW Shadow, while 3,087,987 writes (3.4× more) are issued from the L2 when simulating NVOverlay. Among these 3 million writes, 2,413,754 are caused by L2 capacity miss evictions, 668,951 by load-downgrade, 4,994 by store-eviction, and 288 by other events. From these numbers, we conclude that `kmeans` suffers from L2 thrashing by writing a large portion of data it fetched into L2 and later forced to evict them on capacity misses. This explains why `kmeans` favors LLC-based hardware schemes. Fortunately, the larger write amplification does not translate to higher execution time for NVOverlay.

**Persistent metadata overhead:** Fig. 4.12 presents the size comparison between the master mapping table ($M_{master}$) and the total snapshot data set. We define the snapshot data set as the total amount of data mapped by $M_{master}$.

Our experimentation shows that the ratio between snapshot data set size and $M_{master}$ size is relatively stable across different runs. In all workloads except `yada`, the metadata cost is between 12.8%–15.1% of the working set size. This result is consistent with the property of radix trees. In a perfectly populated tree, each 8-byte pointer in a leaf node can map a 64-byte cache block, achieving a theoretical lower bound of 12.5%. Our results show that in most cases, NVOverlay can achieve almost optimal usage of mapping table storage, thanks to the locality of computation.

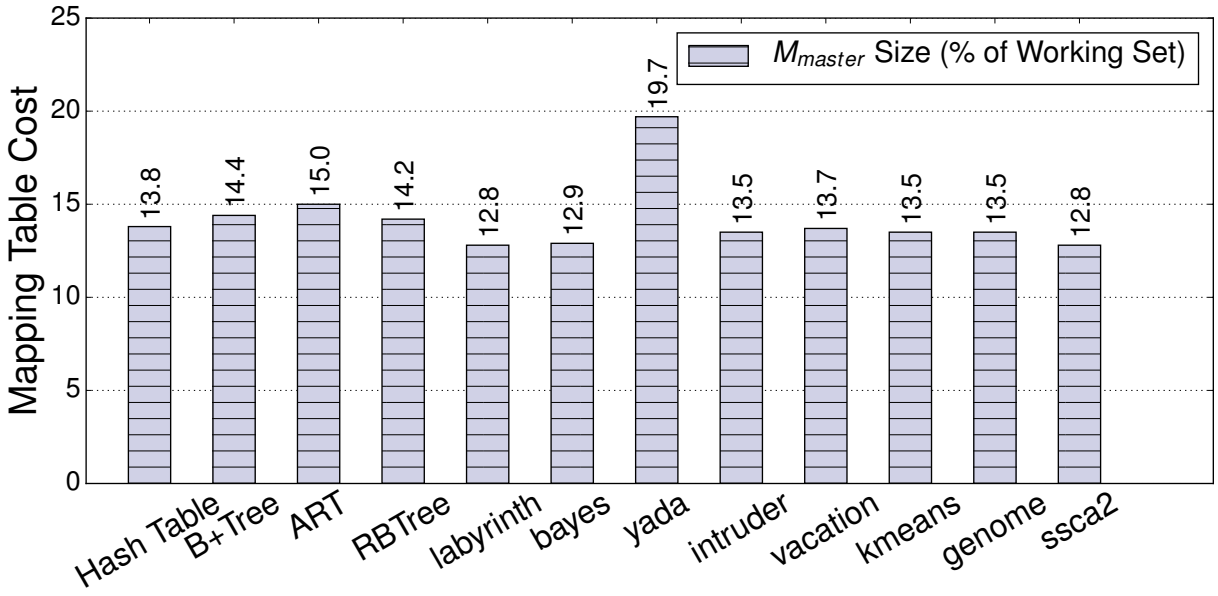
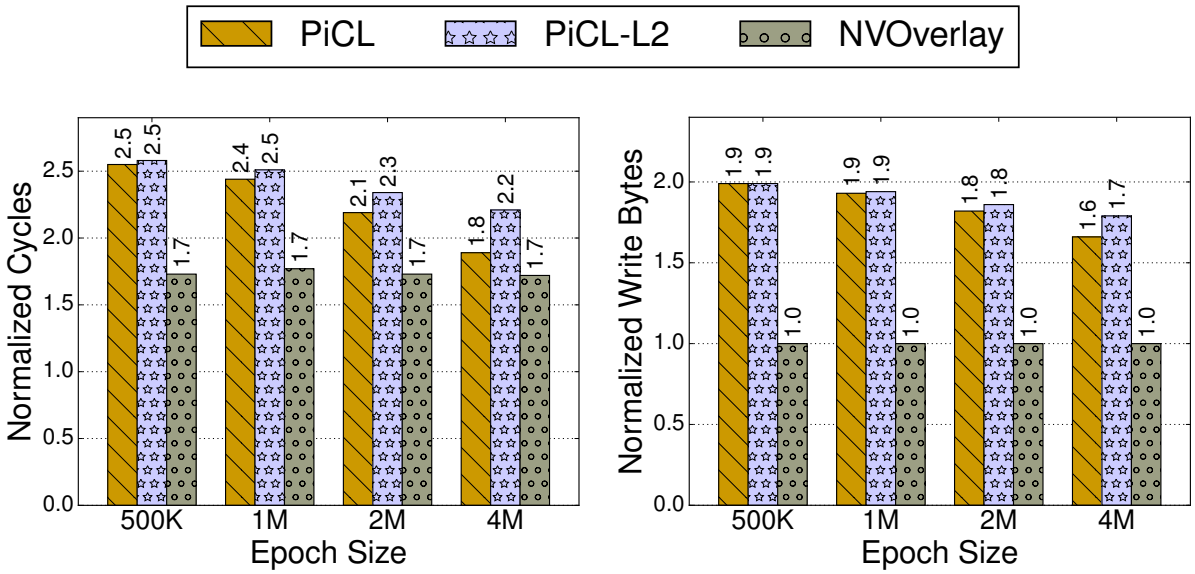



Figure 4.12: **Persistent Mapping Metadata Cost** – All numbers are presented in the percentages of snapshot data set size.

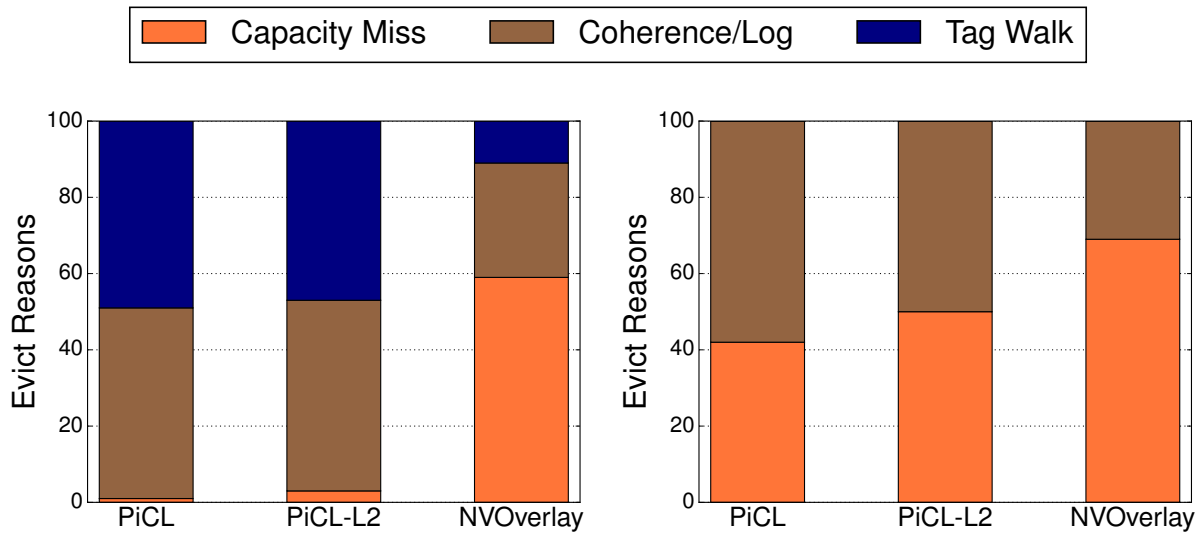



(a) Normalized Cycles

(b) Normalized Writes

Figure 4.13: **Sensitivity to epoch size (ART Benchmark)** – Cycles are normalized to baseline; Writes are normalized to NVOverlay.

As for `yada`, further investigation shows that each inner page in the table only maps 18.14 pages (3.54% of total slots) on average, implying low occupancy of inner pages. In contrast, 93.66% of leaf page slots map a cache block, suggesting that locality in small address ranges are still maintained.




(a) With Tag Walker (1M Epoch)

(b) Without Tag Walker

Figure 4.14: **Evict Reason Breakdown (ART Benchmark)**

### 4.4.3 Sensitivity Study—Epoch Sizes

We study the effect of varying epoch sizes (and hence tag walk frequency) by simulating PiCL, PiCL-L2, and NVOverlay on `ART`, with epoch sizes ranging from 500K to 4M. Results are shown in Fig. 4.13a (cycles) and Fig. 4.13b (write amplification).

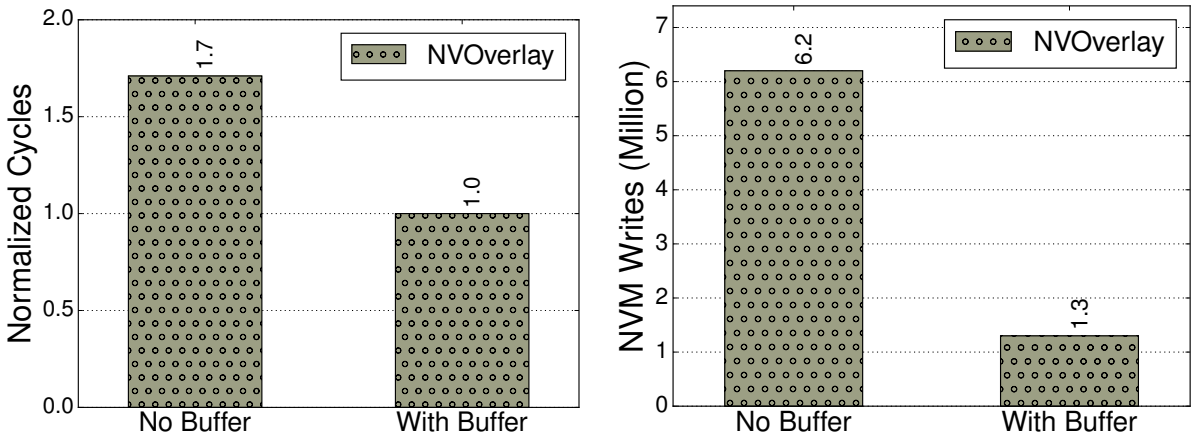
**Cycles:** Both NVOverlay and PiCL-L2 are insensitive to epoch size change. This result could be explained by the fact that most evictions are caused by coherence downgrade (2,105,356, 29.7%) and L2 capacity miss eviction (4,194,012, 59.1%), while tag walks (792,255, 11.2%) only marginally contribute to total write bandwidth. PiCL, on the other hand, performs better under long epochs since around half of the evictions are generated by tag walk evictions (6,086,088, 50%), which its tag walker generates on committing a previous epoch.

**Write Amplification:** As epoch size increases, write amplification of both PiCL and PiCL-L2 steadily drops due to the reduced frequency of tag walks. Besides, since dirty cache blocks can survive longer in the cache without being forced out, fewer log entries are generated. As epoch size increases from 500K to 5M, write amplification drops 11.0% and 15.9% for PiCL and PiCL-L2, respectively.

### 4.4.4 Sensitivity Study—Cache Tag Walker

To evaluate the effect of the cache tag walker on performance, we simulate PiCL, PiCL-L2, and NVOverlay on `ART`, with and without the tag walker. Results are presented in Fig. 4.14 as a breakdown of evict reasons.

Both PiCL and PiCL-L2 are heavily dependent on the tag walker for making progress. As in Fig. 4.14a, more than 47% of total write requests are generated by tag walks, indicating that the tag walker might become a performance bottleneck on larger caches for PiCL and PiCL-L2. NVOverlay, by contrast, writes back dirty blocks mainly by cache coherence and capacity miss eviction, which is distributed evenly during the execution. The tag walker only contributes to




(a) Normalized Cycles (b) NVM Writes

Figure 4.15: **Reducing Writes with OMC Buffer (ART Benchmark)**

around 11% of total evictions. Therefore, the efficiency of the tag walker has a limited effect on NVOverlay, as indicated by Fig. 4.14b.

### 4.4.5 OMC Buffer

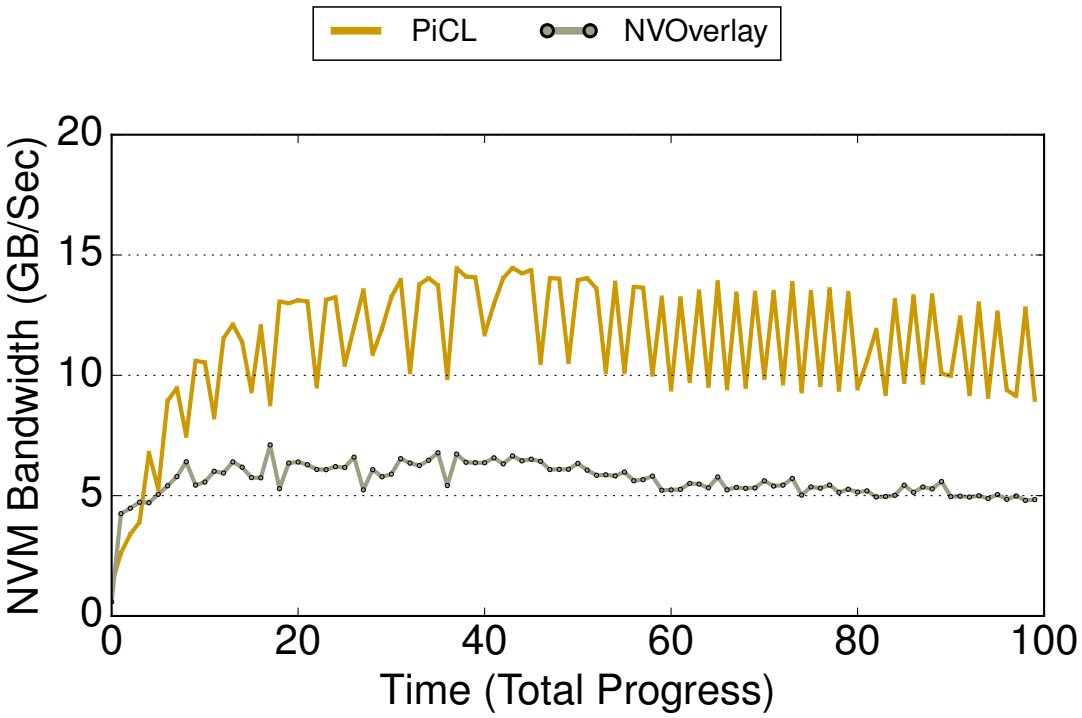
We evaluate the persistent OMC buffer proposed in Section 4.2.8 by simulating NVOverlay on `ART`, with and without the buffer. The evaluation has only one epoch throughout the execution to stress-test the buffer's ability to absorb redundant write-backs (i.e., those generated on the same address and in the same epoch) from the hierarchy. We use a buffer with the same configuration as the simulated LLC, expecting that it would further reduce NVM write traffic as if NVOverlay were built on the LLC. Results are in Fig. 4.15.

As shown in Fig. 4.15a, the OMC buffer improves performance by 41%. Fig. 4.15b further reveals that the performance improvement is a result of reduced writes, proving its effectiveness. Out of 7,136,893 write requests, 5,336,687 hit the buffer, achieving a hit rate of 74.8%.

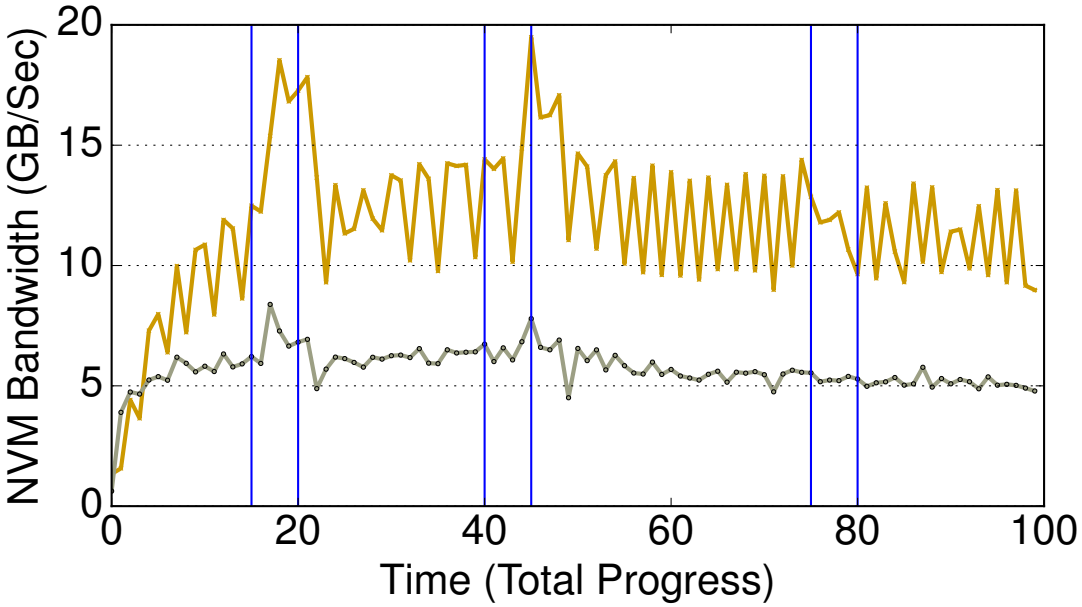
### 4.4.6 Bandwidth

To evaluate bandwidth benefits, we simulate NVOverlay and PiCL on `B+Tree`, and measure NVM write bandwidth as a time series. Results are presented in Fig. 4.16. Fig. 4.16a shows bandwidth over time during the entire simulation using the default epoch size. NVOverlay demonstrates two clear advantages: (i) Average bandwidth consumption is significantly lower than PiCL; (ii) Peak bandwidth and overall fluctuation is also lower, indicating better scalability since more components can be supported on a fixed bandwidth budget. We attribute this to the fact that NVOverlay's version coherence "amortizes" version write-back bandwidth over regular execution while PiCL must evict dirty blocks with tag walk, creating bandwidth surges at epoch boundaries.

Fig. 4.16b shows bandwidth over time when epochs occur in short but localized bursts. We use this to mimic time-travel debugging, where programmers may manually start new epochs around suspicious code regions or on certain events (e.g., when receiving a network packet). Three




(a) 1M Default Epoch




(b) Bursty Epoch

Figure 4.16: **NVM Write Bandwidth Time Series (B+Tree Benchmark)**

"bursty" intervals are present, marked by blue vertical blocks. From left to right, the size of epochs in these bursty intervals are 1K, 10K, and 100K, respectively. The figure demonstrates that both schemes work as intended without introducing a huge amount of bus traffic when the epoch size is moderately small (100K). With extremely small epochs (1K, 10K), however, NVOverlay still

sustains relatively lower bandwidth, while PiCL observes 50% more traffic due to frequent log generation after a new epoch begins, and cache flush when an epoch is being committed.

## 4.5 Additional Related Work

To alleviate write-induced wearing of NVM devices, prior works have explored using entropy data compression that reduces NVM write sizes or bit flips [49, 91, 262]. These designs add extra compression hardware to reduce the size of a block or minimize the number of bit flips when writing the block. They improve the longevity of NVM devices by generally writing less data or avoiding bit flips when the write occurs on the physical storage cells.

Works by Kateja et al. [128, 129, 130] focus on detecting hardware errors induced by firmware bugs on NVM devices. Vilamb [129] performs asynchronous background memory scrubbing on direct-mapped (DAX) NVM storage with background software threads. The background thread computes NVM page checksums and verifies the result with those maintained in the redundant checksum storage. Tvarak [130] further optimizes the latency of the scrubbing operation by offloading most of the work to specialized hardware.

Due to their non-volatility, NVM devices are bound to leak data if malicious attackers acquire the physical device. To secure persistent data from being abused or stolen, prior works proposed counter-mode encryption as a low-cost solution [161, 162, 271, 277, 279]. In counter-mode encryption, data written into the NVM device is encrypted by hardware-generated counter values seeded by a secret key. Users can only access decrypted data after entering the same secret key as in encryption.

# Chapter 5

# Multi-Block Cache Compression: Leveraging Cross-Object Data Compressibility within Pages

This chapter presents Multi-Block Cache Compression (MBC) as a demonstration of how leveraging application-level information on hardware benefits system performance by helping cache compression. MBC performs data compression on the last-level cache (LLC). In MBC, block data is compressed before the block is inserted into the LLC and decompressed before the block leaves the LLC. Since a compressed block generally occupies less space than an uncompressed block, a compressed LLC can store more blocks in compressed form than an uncompressed design. As a result, compressed caches will potentially suffer fewer cache misses and reduce memory bus bandwidth consumption for fetching the block on cache misses.

While conventional cache compression designs operate on single-block granularity, i.e., both compression and decompression only process one block at a time without referring to other blocks, MBC adopts a novel technique called "inter-block compression". With inter-block compression, a data block is compressed using another block that stores similar content (which we call an "analogous block") as reference. Inter-block compression is expected to yield a higher compression ratio than single-block designs because, theoretically speaking, this approach can leverage *both* intra-block and inter-block data redundancy. By contrast, traditional single-block compression techniques can only leverage intra-block redundancy and they may perform poorly when the degree of redundancy within a block is insignificant (which occurs quite often in certain scenarios, e.g., database rows).

Since the compression ratio benefit of inter-block compression originates from data similarity between the analogous reference block and the block to be compressed (which we call a "target block"), the most critical challenge of inter-block compression, therefore, is to find the analogous block when compression is performed. Prior works on inter-block cache compression address this challenge by adding a "base cache" that stores analogous blocks selected from the working set. The base cache operates in tandem with the LLC, and it provides analogous blocks to the corresponding hardware when a block is being compressed or decompressed. These designs may radically differ in the mechanism that they define the representativeness of the selected blocks concerning the entire working set to maximize inter-block redundancy and compression ratio.

Nevertheless, the underlying reasoning for using base caches prevails in all these works.

In this chapter, we present MBC as an inter-block compression design that eliminates the base cache by leveraging cross-object redundancy within a page. As already discussed in Section 2.3.1, the base cache can introduce scalability bottlenecks when the system runs a variety of workloads or when the LLC is partitioned, and the base cache itself is a rather bulky piece of hardware whose power and area consumption defeats the very purpose of cache compression. By eliminating the base cache, MBC scales better to large systems, and the inter-block compression design can also be simplified considerably.

Without a base cache, MBC can still find analogous blocks based on the observation that there is often substantial data similarity within a page. The source of such similarity is usually large arrays-of-structs, or memory allocators placing same-type objects next to each other [3, 30, 72, 85]. In addition, the similarity of blocks also often forms a "stepped" (spatially strided) pattern as a result of large objects or block-aligned allocation. Unfortunately, different applications will have different patterns, and even in the same application, the pattern would differ if it uses assorted object sizes. In other words, the stepped pattern, while seemingly a low-hanging fruit for compression designs to leverage, is heavily application-dependent and would be non-trivial to utilize without application-level information.

Instead of blinding assuming a pattern (as in previous works such as [196]), MBC addresses the difficulty by adding application-level information as input to the hardware, and using the input to guide inter-block compression such that good analogous blocks can be found. More specifically, in MBC, it is the responsibility of the application software to place analogous blocks on a page to form the stepped pattern (which is already the case for many scenarios as discussed above, and hence need not be done intentionally), and then to provide the pattern to hardware using a single per-page "step size" attribute. On seeing the attribute, the compression hardware understands the layout of analogous blocks on a per-page basis, and when a data block is inserted into the LLC, the analogous blocks on the same page can thus be easily found using the "step size" attribute. By leveraging application-level information as redundancy hints, the resulting design of MBC is significantly simpler than those using base caches, but yet is flexible enough to handle many challenging cases that would have otherwise been not possible without the per-page attribute. Based on the above analysis, we regard MBC to be at a sweet point between design complexity and flexibility.

Besides the main contribution of leveraging application-level information to guide inter-block compression, MBC also features two additional design highlights. First, MBC leverages *both* intra- and inter-block redundancy at the same time with one single compression algorithm, thus achieving the theoretical best-case scenario of inter-block compression. The algorithm is based on C-PACK [47], a dictionary compression algorithm that already compresses well within a block and further extends it to extract data redundancy between analogous blocks. Second, MBC combines *super-block tags* with 1.5× tag over-provisioning, accomplishing a maximum effective capacity of 6× compared with an uncompressed cache. The super-block tag organization also enables the cache controller to dynamically map consecutive analogous blocks on the same page to the same super-block tag using a skewed set index generation function according to the page's "step size" attribute. Both the reference analogous block and the target block can therefore be found in the same set with one tag array access.

Compared with prior works on inter-block cache compression, MBC pushes forward the state-

of-the-art in two aspects. First, MBC enables inter-block compression without introducing bulky and complicated hardware, i.e., the base cache, and hence requires less radical hardware changes than designs with a base cache, such as Thesaurus and Zippads [84, 245]. Meanwhile, compared with DISH [196] which only performs inter-block compression across adjacent blocks on the address space and ignores application-level information, MBC is more flexible and supports more scenarios where analogous blocks are not adjacent to each other. Second, with the combination of a high-ratio algorithm capable of leveraging both types of redundancy, and a tag array organization with super-blocks and tag over-provisioning, MBC surpasses prior designs by being able to compress blocks to smaller sizes and tracking more blocks in the tag array. By contrast, prior works either only leverage one type of redundancy, ignoring the other (e.g., using delta compression between analogous blocks ignores intra-block redundancy), or upper bounds effective capacity to $2\times$. The resulting compression ratio is less satisfactory than those of MBC, which can even approach the theoretical $6\times$ upper bound in certain "good cases".

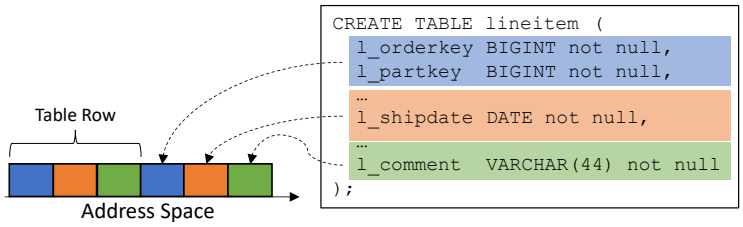
## 5.1 Design Overview

### 5.1.1 A Motivating Example

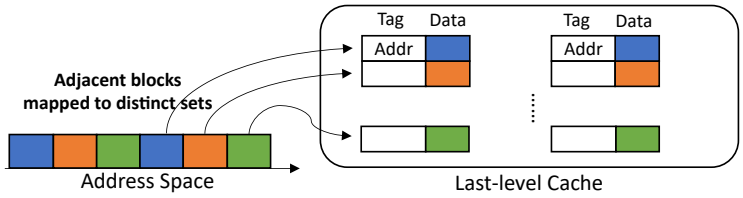
To better illustrate MBC's design motivation, we present a real-world example in Fig. 5.1a. This example shows the memory layout of the `lineitem` table of TPC-H benchmark. We assume that the table is stored in a row format, and each row is stored next to each other on the address space. As the figure illustrates, since a row in the table occupies three consecutive blocks, when multiple rows are laid out on the address space, blocks that contain the same field in different rows form a spatially strided pattern, which we call a "stepped pattern". In this figure, the pattern is marked by blocks of different colors on the left side, with the fields contained in the block marked in the same color to correspond. The takeaway message from this simple example is that those blocks of the same color are likely analogous, because they store the same column of the table. Values from the same column are likely to have similar values because (i) they are of the same semantic meaning, and hence will share the same value domain, which is usually far smaller than the value domain of the type, and (ii) many real-world data sets have value locality [6, 213].

Despite forming an easily recognizable stepped pattern on the address space, these analogous blocks are difficult to leverage for inter-block compression in a conventional cache. Fig. 5.1b illustrates the reason. In a conventional cache, adjacent blocks on the address space are mapped by the set index generation function to different sets to minimize set contention. This phenomenon is shown in the figure as all six blocks are mapped to different sets. Therefore, in a conventional cache, if analogous blocks are to be compressed together with inter-block compression, one more cache access must be made to retrieve the reference block. Similarly, for decompression, the analogous block must also be read out first before the target block is accessed and then decompressed. Inter-block compression is infeasible in this case because the additional access doubles the latency of cache reads, which is on the critical path.

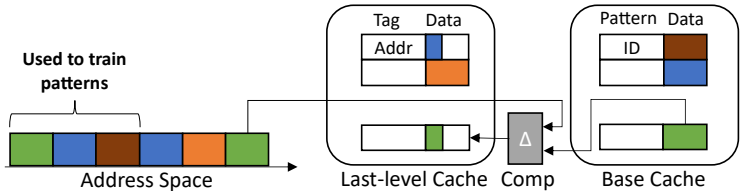
Prior works address this problem by adding a small "base cache" alongside the LLC. The base cache is trained by a selecting process to contain representative cache blocks that are likely to be analogous to the rest of the working set. In our example, the base cache would contain the three



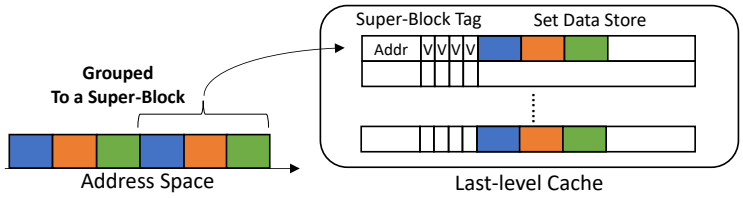

(a) Analogous Blocks in TPC-H `lineitem` Table Row



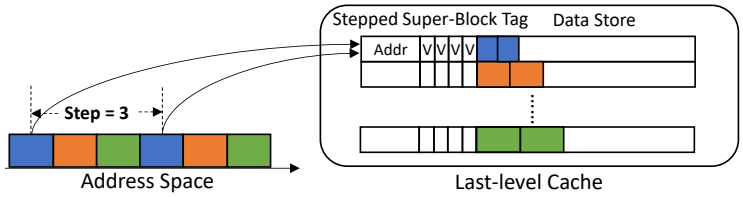

(b) Conventional Cache Block Mapping




(c) Base Cache in Previous Works




(d) Inter-Block Compression on Super-Blocks without Stride Info




(e) Inter-Block Compression on Stepped Super-Block Tags

Figure 5.1: **Design Overview** – *Analogous blocks* are shown with the same color. We use non-power-of-two "step size" to demonstrate the flexibility of our design.

blocks constituting a table row, as shown by the right half of Fig. 5.1c. The base cache operates in tandem with the LLC when a target block is to be inserted or accessed, and it provides the analogous block needed by compression and decompression, respectively. Fig. 5.1c depicts the former case where a new block (marked as green) is to be inserted into the LLC. In this case, the analogous block is retrieved from the base cache (e.g., using the fingerprint hash of the incoming block, which is how Thesaurus works exactly [84]), and then used to compress the newly inserted block with delta compression. Since the base cache provides an analogous block, the resulting

compressed block is much smaller then the original one and occupies fewer bytes in the data array.

Previous inter-block compression designs also attempted not to rely on the base cache to provide analogous blocks. Instead, they assumed that adjacent blocks on the address space must be analogous. As a result, these designs always compress adjacent blocks with the best efforts regardless of the actual pattern [196]. In such a design, every consecutive four blocks on the address space are mapped to the same cache set and then grouped into "super-block tags". Each super-block tag entry can track four consecutive blocks with a single address field (`Addr`) and four copies of per-block status bits, as shown in Fig. 5.1e where adjacent blocks constituting a table row are mapped to the same tag entry. Inter-block compression can be performed across the four blocks in the same super-block tag because only single cache access is needed to retrieve all four blocks. In our TPC-H case, however, this design would work poorly because of the assumption that adjacent blocks must be analogous does not hold. In fact, if every four consecutive blocks are compressed together, then only one of them would be able to leverage inter-block redundancy because analogous blocks form a stepped pattern of size three.

Finally, Fig. 5.1e depicts the optimal way of performing inter-block compression without a base cache for this example, which MBC adopts. In this optimal design, super-block tags are still used to group analogous blocks into the same tag, such that the base cache can be eliminated. However, instead of assuming that adjacent blocks are always analogous and mapping them to the same set, the index generation function takes one more argument, the "step size" attribute, and maps every four blocks that are "step size" blocks away from each other into the same set. In our example, the "step size" value is set to three, which is the length of the strided pattern. As a result, analogous blocks that are three blocks away are mapped to the same set and then grouped into the same *"stepped super-block tags"*, such that inter-block compression can be performed efficiently on them. The figure shows same-color blocks being mapped to the same set and then grouped into the same tag.

### 5.1.2 Multi-Block Cache Compression

The lesson we have learned from the above example is that cache compression hardware can benefit from application-level information, i.e., the "step size" attribute that describes the data pattern of a page. In addition, by grouping every four consecutive analogous blocks in the stepped pattern, the stepped super-block tags enable compression hardware to find analogous blocks in the same tag entry, thus eliminating the need for a base cache.

Multi-Block Cache Compression (MBC) presents an LLC compression design that incorporates these two observations. Instead of relying on the base cache to provide analogous blocks, MBC compresses across stepped analogous blocks on pages, which is guided through a software interface. Specifically, the software provides per-page "step size" attributes that enable applications to pass stepped patterns of analogous blocks to the hardware as software hints. With layout information available, the hardware groups potentially non-contiguous analogous blocks on the stepped pattern into stepped super-block tags. Inter-block compression is performed within the stepped super-block tag by using the first logical block of the super-block as a reference.

The stepped super-block tag derives from DCC [220], but it improves over DCC's regular super-block tag by skewing the cache controller's index generation function with the software-provided "step size" value, such that every four consecutive blocks on the stepped pattern are

mapped to the same set and can thus be grouped into the same tag entry. The reference block can always be retrieved from the same tag entry in single access along with the blocks that are compressed using the reference block, eliminating the need for a base cache.

To maximize compression ratio, MBC adopts dictionary-based C-PACK as the base algorithm and performs inter-block compression/decompression by first running the base algorithm on the reference to generate the dictionary and then using the dictionary to compress/decompress the target. To reduce the latency of running the algorithm twice, we also present two effective optimizations, *parallel decompression* and *zero optimization*, which take advantage of data dependency and the distribution of zero values, respectively. The algorithm design enables MBC to leverage both intra- and inter-block redundancy. To match up to the high compression ratio, MBC features a 1.5× over-provisioned tag array, accomplishing a maximum effective capacity of 6×. The combination of a highly efficient algorithm and large effective capacity in MBC leads to high overall ratio and significant performance improvement.

MBC also demonstrates excellent flexibility by allowing developers to assign different "step size" attributes manually and dynamically to different memory regions through a modified `mmap`, which configures the attribute on a per-page basis. This feature is incredibly precious to cloud applications because it virtualizes cache compression. Cloud users can thus implement their unique compression policies on a per-application or per-user basis while staying isolated. On the other hand, even without a software hinted "step size", applications can still benefit from MBC, in which case MBC assumes a default "step size" of one, compressing across every four consecutive blocks on the physical address space. In this scenario, MBC operates similarly to DISH [196] but still achieves higher compression ratio because of its novel algorithm and tag array design.

The rest of the sections are organized as follows. In Section 5.2, we present the cache components that MBC adds to a baseline LLC, including a stepped super-block tag array, a segmented data array, and a skewed set index generation function. Then in Section 5.3 we present the C-PACK-based inter-block compression algorithm, and the two optimizations for reducing operation latency. Lastly, in Section 5.4, we demonstrate the system-level implication of MBC and the proposed software support.

## 5.2 The Stepped Super-Block Cache Organization

### 5.2.1 Data and Tag Organization

MBC consists of a segmented data store and an over-provisioned super-block tag array, as depicted in Fig. 5.2. We elaborate on the organization of each in detail.

**Data store:** The per-set data store is a unified piece of storage consisting of segments. There is no implicit one-to-one mapping from tag entries to segments. Instead, compressed blocks can be stored in any consecutive interval of segments, but different blocks are not allowed to share a segment (shown in the lower half of Fig. 5.2). The data store is also assumed to be "defragmented" lazily on insertions if external fragmentation of segments prevents contiguous segments from being allocated, which would have otherwise been successful. Previous works have shown that defragmentation happens infrequently, is not on the critical path, and does not incur noticeable overheads [13, 220].

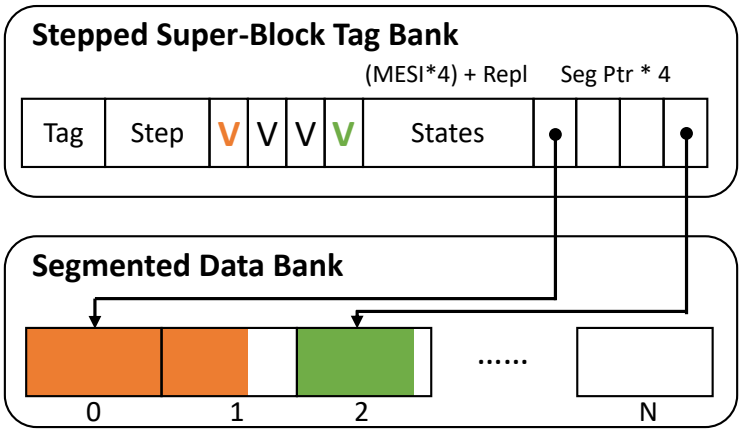


Figure 5.2: **Data and Tag Organization** – Orange and green blocks are the first and last in the stepped super-block tag. The compressed orange block is stored in two consecutive segments.

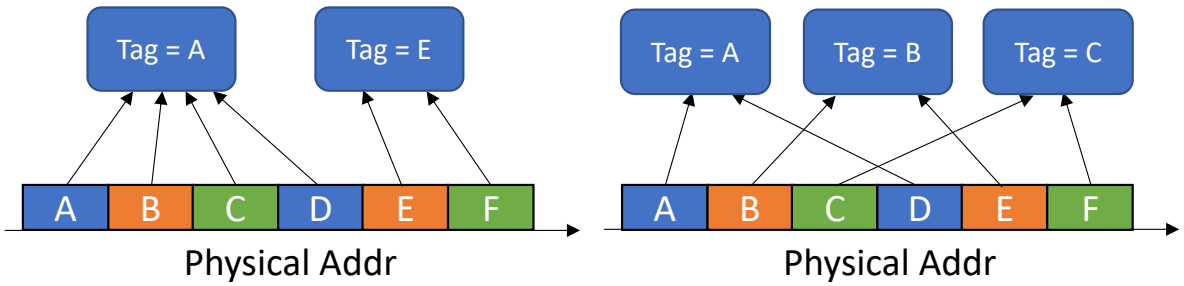
**Tag array:** The tag array is decoupled from the data store by introducing a per-block segment pointer which stores the ID of the first segment that holds compressed block data. Cache lookup needs first to locate the tag entry and then follow the indirection pointer to retrieve one or more consecutive segments from the data store. Each tag entry stores the metadata of a stepped super-block consisting of four consecutive blocks on a step, including a "Tag" field, a "Step" field, and four "valid" bits. Like prior super-block designs, we chose a super-block size of four as a sweet point between compression ratio and metadata cost. Per-block state bits, except replacement, are also duplicated four times, so cache coherence still operates on individual blocks.

To achieve higher effective capacity, MBC further over-provisions the tag array by 1.5×, meaning that the number of tag entries per set is 1.5× more than that of an uncompressed cache. Combining over-provisioning with super-blocks enables a maximum effective capacity of 6× of the uncompressed size while still maintaining a reasonable metadata cost (see analysis in Section 5.5). As shown in Section 5.6, for certain workloads, this tag organization is critical for achieving high effective cache capacity, as conventional 2× tag over-provisioning would exhibit performance that is "tag-bound" despite a high raw compression ratio.

### 5.2.2 Inter-Block Compression on stepped super-blocks

MBC enables inter-block compression within the stepped super-block using a dictionary-based algorithm. The algorithm compresses and decompresses the other three blocks (the target blocks) in the super-block after initializing the dictionary using the content of the first block, which we call the reference block. This novel approach unlocks a higher compression ratio when inter-block redundancy is present because values that would have been incompressible in the target block may now be found in the dictionary generated from the reference block. We present the detailed inter-block compression algorithm in Section 5.3.

As mentioned earlier, the main contribution of MBC is that it eliminates the base cache while efficiently finding the analogous block to be used as the reference on the same page. This process is where software comes into play: MBC expects application software to place analogous blocks on the same page with a stepped pattern and to notify the hardware of the pattern. In practice, stepped patterns can be formed naturally by placing same-type objects next to each other on




(a) Regular (b) Stepped

Figure 5.3: **Comparison Between Regular and Stepped Block Mapping** – Two objects that span three blocks each are depicted, with blocks on the same object offset being analogous, indicated by the same color.

one or more consecutive pages, which is what many commonly used memory allocators (e.g., `tcmalloc` [85], `jemalloc` [72], `hoard` [30], `pymalloc` [3]) already do. Blocks that contain the same field of different object instances are considered analogous since these fields typically share the same value domain and exhibit value locality.

To group analogous blocks on the same step into stepped super-blocks, the cache controller maps every four blocks on a given "step size" to the same cache set, using a skewed index generation function, and then generates the same tag value for these blocks. For example, with a "step size" value of three, memory block 0, 3, 6, and 9 will be mapped to the same set, while block 1, 4, 7, and 10 will all be mapped to a different set. The same tag value is also generated for the four blocks to ensure that they can be grouped into the same super-block tag entry.

**Example.** Fig. 5.3 highlights the difference between regular and stepped super-blocks with the "step size" value being three. In Fig. 5.3a, two objects that span three blocks each are placed next to each other on the same page. Consequently, regular super-block mapping can only group adjacent blocks to the same super-block, namely, block A–D and block E–F. However, since only block A and D are analogous, inter-block compression would not work well with this scheme. By contrast, in Fig. 5.3b, the index generation function is skewed with a "step size" of three, causing block A, D, block B, E, and block C, F to be mapped to different super-blocks (due to space constraints, we only drew two blocks per super-block, but in practice, four blocks can be mapped to the same super-block), where inter-block compression is then performed on analogous blocks, producing a potentially higher ratio.

### 5.2.3 Hardware Design

Fig. 5.4 depicts the overall hardware design of MBC. On memory operations, the TLB translates the virtual address to the physical address and "step size" value (the upper part of Fig. 5.4). The per-page "step size" value is fetched from the page table entry into the TLB by the MMU page table walker, and stored in the TLB entry. The "step size" value is also passed down the hierarchy on cache misses with the memory request until it reaches the LLC.

L1 and L2 lookups proceed unmodified as in a regular cache, and they ignore "step size", as shown by the middle part of Fig. 5.4. At the LLC, the cache controller first generates the set index using the physical address and the "step size":

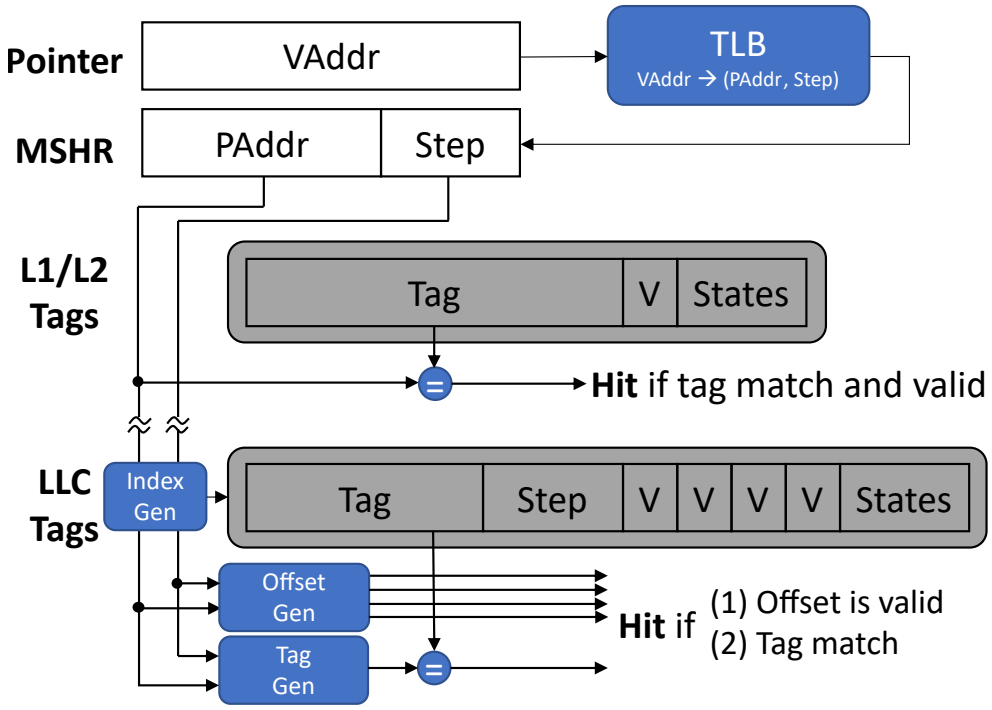



Figure 5.4: **Address Translation and Tag Lookup for MBC**

```
index = hash((PAddr DIV step) RSHIFT 2,
              PAddr MOD step)
```

where `hash` is an arbitrary hash function. The first hash argument yields the same value for every consecutive `step * 4` blocks on the address space, while the second argument yields different values for each of the consecutive `step` blocks. The overall index generation function maps every four blocks on the step to the same set, and maps adjacent blocks belonging to different steps to different sets. Assuming that the hash function distributes the output evenly, this function minimizes the chance of set contention among adjacent blocks on different steps.

Meanwhile, the tag for the associative lookup within the set is generated:

```
tag = (((PAddr DIV step) AND ~0x3) +
       (PAddr MOD step))
```

meaning that every four blocks on the step will use the same tag to perform the associative lookup, which is the address of the super-block's first block. Combined with the set index generation function, it is guaranteed that these four blocks will be grouped into the same stepped super-block, and can thus be compressed together.

If the associativity lookup indicates a tag hit, the offset of the block within the stepped super-block is computed as follows:

```
offset = ((PAddr DIV step) MOD 4))
```

which is then used to retrieve the per-block "valid" and other state bits. A lookup hit is signaled if there is both a tag hit and the valid bit is set.

If the operation is an insert, the "step size" value is also written into the per-tag "step" field. This field will be used to generate the write-back physical address when a block at `offset` of the stepped super-block is evicted:

```
PAddr = tag + (offset MUL step)
```

The index, tag, and offset calculations require two common components, namely, `PAddr DIV step` and `PAddr MOD step`. In practice, this can be done with either a hardware divisor or a ROM lookup table. Since LLC lookup only happens after the L1 and L2 both miss, the latency of the computation can also be overlapped with L1 and L2 lookup, which should take at least a couple of cycles.

### 5.2.4 Operation Details

**Insertion:** On insertion, the block is first compressed. If the block already exists in the cache, the stale data segments are freed before the insertion. In all cases, a consecutive range of segments will be allocated to store the block. Replacement and defragmentation are performed if necessary. The segment pointer, block size, and other state bits are also updated accordingly.

If the block is non-reference, it will be compressed with the inter-block algorithm. The algorithm takes both the reference block, which we assume is always in the cache (we discuss how to guarantee this shortly), and the target block as input, and outputs a compressed target block. Reference blocks, on the other hand, are always compressed with the intra-block algorithm to avoid circular dependencies.

If the insertion operation is from the upper level and hits a reference block, the remaining blocks in the tag must first be decompressed with the old reference and then recompressed. However, this extra step is performed infrequently and is off the critical path.

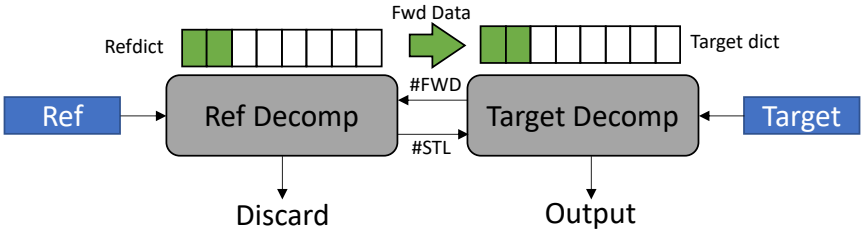
On rare occasions, the compressibility of a block changes significantly after recompression. In this case, these blocks are treated as if they were also freshly inserted into the cache, triggering replacement. In our experiments, this is extremely rare and hence incurs negligible overhead.

**Read:** On read accesses, the cache set index, tag, and offset are calculated as described in Section 5.2.3. A hit is signaled if a matching stepped super-block tag exists, and the "valid" bit on the offset is set.

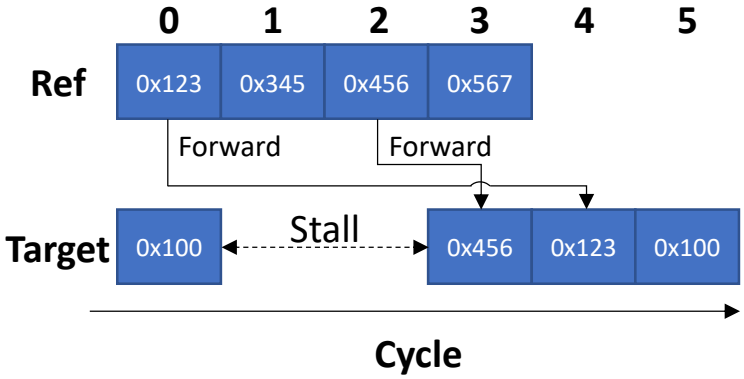
On read access of a non-reference block, it must be decompressed with the inter-block decompression algorithm. The algorithm takes both the reference and the target block in the compressed form and recovers the content of the target block. Reference blocks are always decompressed with intra-block decompression.

**Replacement:** MBC only maintains *per-tag* replacement bits. There are two cases of replacement. In the first case, a tag entry is available to associate with the new block, but the data store does not have sufficient vacant segments. Existing blocks must, therefore, be evicted in order to allocate space. The eviction process first selects the victim tag using per-tag replacement bits. It then evicts valid blocks in the tag with priority given to larger ones until sufficient free segments are available. This process may need to be repeated on several victim tags to free enough space. No per-block replacement bits are needed, as victims are selected by their sizes within the tag. Prior researches [25, 202] also indicate that compressed caches generally benefit from this approach.

If the reference block is selected as the victim, MBC will evict the entire tag to guarantee that the reference block is always present when target blocks are compressed or decompressed. Alternatively, the lowest numbered logical block that is still valid can be re-designated as the




(a) Decompressors




(b) Example – Blue blocks represent uncompressed words; Numbers are word values

Figure 5.5: **Parallel Decompression Architecture** – We assume one word per cycle latency and four-word blocks

reference. When the current reference block is evicted, the rest of the blocks in the tag are decompressed first and then recompressed with the new reference block. This method does not require extra metadata bits while preserving at least part of the content of the stepped super-block.

In the second case, all tags in the cache set are occupied, so eviction begins by selecting a tag to free and then evicting the associated data segments. The victim is selected using per-tag replacement bits as in a regular cache. The new line can be placed after the previous step if it fits in the available data segments. If it does not, the situation now matches the first case, and that process may be used to free sufficient data segments.

## 5.3 The Inter-Block Compression Algorithm

### 5.3.1 The Naive Inter-Block Algorithm

MBC adopts C-PACK [47] as the base algorithm. The base compressor processes the input block in 32-bit words and generates a dictionary (implemented as a FIFO CAM) along the way that consists of words that have already been processed. An input word is compressed by comparing it against dictionary entries. If hardware determines that the input word can be compressed as an exact match or a small delta to an existing dictionary entry, then the compressed code word is generated by outputting the index of the dictionary entry, and optionally the small delta value. Decompression reconstructs the same dictionary as in compression by inserting words that have just been decompressed into the dictionary. The original value of a compressed code word is

recovered by adding the small delta value to the dictionary entry that the code word indexes. The dictionary is generated on-the-fly for both compression and decompression, which is discarded afterward. Experiments show that a 16-entry dictionary works the best.

MBC enables inter-block compression by first running the base algorithm on the reference block, discarding the output, and then using the generated dictionary to compress the target block with the the base algorithm. Target block compression also updates the dictionary using values from the target block itself. When the insertion point reaches the end of the dictionary, it will wrap back, which overwrites earlier entries without affecting correctness. Inter-block decompression works by running the base decompression algorithm on the reference block, and then using the generated dictionary to decompress the target. Since the base algorithm generates the same dictionary on compression and decompression, the target block is guaranteed to be appropriately decompressed.

MBC leverages *both* intra- and inter-block redundancy. On the one hand, by warming up the dictionary using values from the reference block, MBC accomplishes a higher ratio by compressing words in the target block that would otherwise be incompressible using dictionary entries from the reference block. On the other hand, even when inter-block redundancy is lacking (e.g., due to a mismatching "step size"), MBC still extracts intra-block redundancy and should perform at least as well as the base algorithm. By contrast, prior works that adopt simpler delta compression [84, 245] would find it difficult to handle the latter case since they cannot leverage redundancy within a block.

### 5.3.2 Parallelizing the Decompressor

The naive implementation of inter-block decompression described above processes the reference and target blocks serially. With an assumed throughput of two words per cycle [47], inter-block decompression will take 16 cycles to process two blocks. This latency is on the read path and is hence performance critical.

To reduce decompression latency, MBC adopts a parallel decompression architecture in which the decompression of the reference and the target blocks are overlapped, as shown in Fig. 5.5a. In the parallel architecture, two instances of the decompression engine process the two blocks in parallel. Each decompressor has its own instance of the dictionary, and they run the same base decompression algorithm. In addition, the design adds a *forwarding data path* that enables direction entries to be passed from the reference decompressor to the target decompressor.

Whenever the target block decompressor attempts to read an invalid entry in its local dictionary (indicated by a 1-bit flag per entry), it infers that the entry must be from the reference block, and it raises an external signal (`#FWD`) to the reference block decompressor to request the missing entry. On receiving the signal, the reference block decompressor either satisfies the request by forwarding the 32-bit entry data to the target block decompressor, if the entry is valid in the dictionary of the former, or raises another signal (`#STL`) to stall the latter until the requested entry becomes available.

The parallel decompressors effectively reduce the latency of inter-block decompression by only stalling the target block decompressor when there is true data dependency, i.e., a word in the target block is compressed using an entry in the reference dictionary. If the two blocks are perfectly aligned, meaning that all 32-bit words in the target block are compressed by a word

from the reference block on the same offset, then the latency of inter-block decompression should be exactly the latency of the base algorithm. In this case, the two decompressors operate in lock steps without stalling.

**Example:** Fig. 5.5b presents a miniature example of parallel decompression. In this picture, both blocks are decompressed at a throughput of one word per cycle. Each blue block represents a compressed code word, with numbers inside the block to show uncompressed values. In this example, we assume that only exact matches are compressed for the simplicity of discussion.

As shown by the picture, the second word of the target block is compressed using the third word of the reference block as the key, and it requires a dictionary entry forwarding during the decompression, since the dictionary entry it refers to will be from the reference decompressor's dictionary. The entry, however, is not available until the end of cycle 3, in which `0x456` is decompressed from the reference block. The target block decompressor, therefore, must stall two cycles to observe the data dependency. Similarly, `0x123` also needs a dictionary entry forwarding from the reference dictionary. However, this second forwarding will not incur any stall since `0x123` in the target block is decompressed at cycle 4, when the value is already present in the reference dictionary.

In this example, only six cycles are spent on decompression instead of eight (which is required by serial decompression). We present the decompression latency for each workload we evaluated in Section 5.6.3.

### 5.3.3 Zero Optimization

Zero is known to be the most common value in many workloads [70, 110], and it deserves special treatment. Instead of treating zeros as regular input words, MBC encodes all 16-bit zeros in the 64-byte input block with a 16-bit zero-value mask as a separate "fast channel", one bit per input word, and prepends it to the output. During compression, the zero encoder filters out zeros from the input stream and feeds the rest to the dictionary compressor. Decompression works by first initializing the output buffer to all-zeros and then shifting the dictionary decompressor output to the correct offset according to the zero-value mask.

With zero optimization, decompression latency is further reduced and is dependent on zero distribution of decompressed data. In the best case, if the entire line is zero (not uncommon in sparse structures), decompression is instant because the dictionary decompressor does not need to process any code word. Generally speaking, one cycle can be saved from the decompression path for every two zeros (not necessarily consecutive) in the block.

## 5.4 Software Support for MBC

This section discusses MBC's software support. MBC adopts a synergetic hardware-software approach where the runtime software generates per-page "step size" information based on the application's data pattern, and provides it to the hardware via the software-hardware interface. We discuss the software-hardware interface first and then the runtime software implementation.

### 5.4.1 Communicating Per-Page Step Size to Hardware

The "step size" of MBC is a per-page attribute, stored in an extra "step" field in page table and TLB entries. The size of the field is implementation-dependent but six bits should be sufficient for most scenarios, supporting a maximum "step size" of 64 blocks (i.e., 4KB objects). The "step" field will be retrieved from the PTE to the TLB during page walks and passed down the hierarchy alongside the memory request.

For application software to specify the "step size", we propose adding a new flag to `mmap`. The OS stores the flag as an attribute of the mapped region and sets up the "step" field in PTEs when the mapping is materialized. The application may also change the "step size" via `madvise` using the same flag. In this case, the OS needs to flush dirty page data from the LLC (but can retain them in upper levels) before the change takes place because changing the "step size" will also change block mapping in the LLC. We do not consider this case to be a major performance issue, because the flushes can be overlapped with TLB shootdown, which itself is a long latency operation. Besides, cache flushes are not needed on `munmap` (because data on the page is already dead), and they never occur in user space.

MBC is also compatible with huge pages and the OS support such as Transparent Huge Pages (THP).

### 5.4.2 Generating The Step Size Attribute

MBC relies on the heap memory allocator to generate the step size during execution. From a high level, the heap memory allocator places allocated objects of the same type on the same page and then computes the "step size" attribute based on the object size. The "step size" is then communicated to the hardware via the proposed `mmap` interface. This whole process can be performed by the allocator automatically without involving the programmers' efforts.

In order to obtain object type information at allocation time, the compiler may be extended to pass type information to the allocator as an extra hidden parameter. Alternatively, the allocator may treat the return address of the call site as the type which is easily obtainable on most architectures. This solution works well when one allocation call site only allocates same-type objects. However, the correctness of MBC is always guaranteed and does not depend on whether the type information is precise.

In practice, many commonly used memory allocators [3, 30, 72, 85] adopt the so-called arena-based allocation, in which they place same-size objects on the same arena (which consists of consecutive pages in the address space). Porting one of these allocators to support MBC is straightforward. Following the principle, we implemented a prototype allocator based on the glibc allocator with minimum changes. In particular, we added the interfaces to obtain the type information either with an explicit type ID or using the return address on the stack. Objects of the same type are placed next to each other in the same arena, and otherwise, they are placed in different arenas. The step size is computed by dividing the object size by 64. Eventually, when the allocator cannot decide the "step size" (e.g., on allocations bigger than 512 bytes) it will fall back to the default "step size" of one. We use the allocator for the evaluation of MBC in Section 5.6.

| | Uncomp | 2× Tags | DCC | DISH | MBC |
|---|---|---|---|---|---|
| **Over-provisioning** | 1× | 2× | 1× | 1× | 1.5× |
| **Max Ratio** | 1× | 2× | 4× | 4× | 6× |
| **Address Tag** | 42 | 84 | 42 | 42 | 69 |
| **State Bits*** | 9 | 24 | 27 | 27 | 40.5 |
| **Comp Sizes*** | 0 | 6 | 12 | 2 | 18 |
| **Segment Ptrs*** | 0 | 12 | 24 | 0 | 36 |
| **Total**** | 51 | 138 | 105 | 71 | 199.5 |
| **% Data** | 9.96 | 27.0 | 20.5 | 13.9 | 38.9 |
| **Bits/Ratio** | 51 | 69 | 26.25 | 17.8 | 33.25 |
| **Dynamic Power (nJ)** | 0.120 | 0.120 | 0.118 | 0.119 | 0.135 |
| **Static Power (mW)** | 297 | 351 | 334 | 308 | 388 |
| **Area ($mm^2$)** | 1.09 | 1.17 | 1.11 | 1.16 | 1.13 |

*Assuming 8-way, 8-byte segment, 3-bit replacement, 6-bit other
**Computed as metadata bits per 512-bit data.

Table 5.1: **Metadata and Compression Ratio Comparison**

## 5.5 Hardware Cost

Assuming an 8-way set associative cache and 8-byte segments, a 2× tag over-provisioned cache requires 138 metadata bits per 512-bit data with a maximum ratio of 2 (69 bits per logical block, 27.0%). In contrast, DCC with super-blocks requires 105 bits metadata with a maximum ratio of 4 (26.3 bits per logical block, 20.5%). DISH, on the other hand, only requires 71 bits since it is based on the YACC design (17.8 bits per logical block, 13.9%). MBC has 12 super-block tags per-set (1.5× over-provisioning), and one 6-bit "step" field per tag, which needs a total of 199.5 bits and supports a maximum ratio of 6 (33.25 bits per logical block, 38.9%). Our analysis does not include Thesaurus, as it proposes a metadata cache that may scale with factors other than cache size (e.g., the number of concurrently running applications). Table 5.1 presents the detailed analysis.

The compression and decompression logic have roughly the same complexity as C-PACK, since they share the same dictionary-based logic. We conservatively assume that the parallel decompression architecture needs 2.5× (120K `NAND` gates [47, 245]) more area than C-PACK as two decompressors are needed. The hardware cost is still reasonable compared to FPC (290K [245]).

We performed power and area analysis using CACTI 7.0 [183] on 22nm technology node. Results are also presented in Table 5.1. Compared with an uncompressed cache, MBC incurs 12.5% overhead on LLC's dynamic energy, 30.6% higher static power, and 3.67% more area due to the larger tag array and extra state bits. Compared with its baseline design DCC, MBC has 14.4% more dynamic energy, 16.2% more static power, and 1.80% more area.

| | |
|---|---|
| **CPU** | 3.0GHz, 1–8 cores, Our-of-Order, 256 ROB, 64 LSQ |
| **L1** | 32KB, 4-way, 4 cycle, 64-byte block, LRU replacement |
| **L2** | 256KB, 8-way, 11 cycle, 64-byte block, LRU replacement |
| **LLC** | 2MB per-core, 16-way, 30 cycle, 64-byte block, LRU replacement |
| **DRAM** | 16 banks, 300 cycles read/write |

Table 5.2: **Simulation Configuration**

Despite the moderately increased energy numbers on the LLC side, MBC can, in fact, save energy on the main memory side with reduced cache misses and hence fewer DRAM accesses. We used the per-access DRAM energy number from prior publication [84] (32.61nJ on 45nm technology node), and estimated the total energy number using a linear model (DRAM access count $\times$ 32.61 + execution time $\times$ dynamic power + LLC access count $\times$ static power) over all workloads in Section 5.6. Results show that, compared with an uncompressed cache, while MBC consumes 1.12$\times$ more power on the LLC, it only consumes 0.76$\times$ the energy of an uncompressed cache. The overall energy is 0.93$\times$ of the uncompressed, proving its energy efficiency.

## 5.6 Evaluation

**Simulation platform:** We evaluate MBC by running simulations on zsim [219]. Simulation configurations are given in Table 5.2, which is consistent with those in previous works [84, 245] that focus on single-core performance with a partitioned LLC slice. We use the following as comparison points:

- **DCC-BDI:** BDI on super-block DCC design, no tag over-provisioning. 2-cycle decompression;
- **DCC-FPC:** FPC on super-block DCC design, no tag over-provisioning. 5-cycle decompression;
- **DCC-CPACK:** C-PACK on super-block DCC design, no tag over-provisioning. 9-cycle decompression.
- **2x LLC:** Double size LLC with the same associativity.

We also compare against the following state-of-the-art inter-block compression designs:

- **YACC-DISH:** DISH Scheme1 on YACC design, no tag over-provisioning. 2-cycle decompression.
- **Thesaurus:** Thesaurus with an ideal base cache that always hits. 2-cycle decompression.

### 5.6.1 Data Applications

**Experimental setup:** We first evaluate MBC on hard-to-compress data application workloads that focus on processing large amount of data in memory. For the `list` workload, we implemented a doubly linked list data structure with 64-bit integer keys and 40-byte payloads. The key is assigned as a monotonically increasing sequence, and the payload is synthesized with

| | Intra-block Redundancy | Inter-block Redundancy | Zero Percentage |
|---|---|---|---|
| `list1` | × | × | 40 |
| `list2` | × | ±64k | 40 |
| `list3` | ±64k | × | 40 |
| `list4` | ±64k | ±64k | 40 |
| `list5` | ±64k | ±64k | 60 |
| `list6` | ±16k | ±16k | 40 |

Table 5.3: **List Workload Summary** – Redundancy is realized by controlling the maximum difference between a payload word and the same word in the previous node (for inter-block redundancy), or between the word and the previous word in the same node (for intra-block redundancy). This table shows the control values.

varying intra- and inter-block redundancy and zero distribution. Six variants of the list workload are evaluated, which is summarized in Table 5.3. The workload traverses through the linked list in one direction from a random starting node for a random number of steps that is at most 5% the length of the list. We use the synthetic list workloads to evaluate the sensitivity of the compression algorithm with regard to varying levels of data redundancy and zero distribution.

The `ycsb` workload implements an in-memory B+Tree [150] and populates the tree with YCSB keys in Zipfian distribution [52]. The workload performs scan operations on tree nodes as specified in YCSB-E. We use the `ycsb` workload to demonstrate MBC's efficiency on data structures with multi-cache-block nodes.

`taxi` is a real-world dataset consisting of taxi ride records in New York City, which has been used in prior research to evaluate software dictionary compression [157]. `weather` is another real-world dataset on country-wide weather data. The dataset has been used in prior work for supporting decision-making systems [178]. We load both datasets in row format and simulate analytical accesses by performing scans. Both datasets feature large rows that span across several cache lines. We use them to evaluate MBC on compressing stepped pattern.

We also evaluate MBC on three real-world applications: `silo` [246], `feature_reducer (reducer)`, and `redis`. The `silo` workload implements an in-memory representation of the `lineitem` table in TPC-H [2]. The table contents are generated using Microsoft's DBGEN tool [1] with the most skewness. `reducer` is a Python workload from FunctionBench [132] that performs word count, and we use the default input. `redis` is an in-memory object store. We use the official traffic generator with default settings to drive its operation.

We simulate 10B instruction on a single core with the proposed memory allocator as specified in Section 5.4.2. The memory allocator places same-type objects on the same arena whenever it could decide the object type from the call site's return address, hence requiring no explicit support from the programmer. We scaled the workloads to keep the L1 hit ratio lower than 95%. We also monitored the memory footprint of the simulated process and did not observe any significant difference between MBC and other designs under comparison.

**Results:** We present results on data workloads in Fig. 5.6, Fig. 5.7, and Fig. 5.8 for IPC, misses per kilo instructions (MPKI) on the LLC, and effective cache size, respectively. Effective

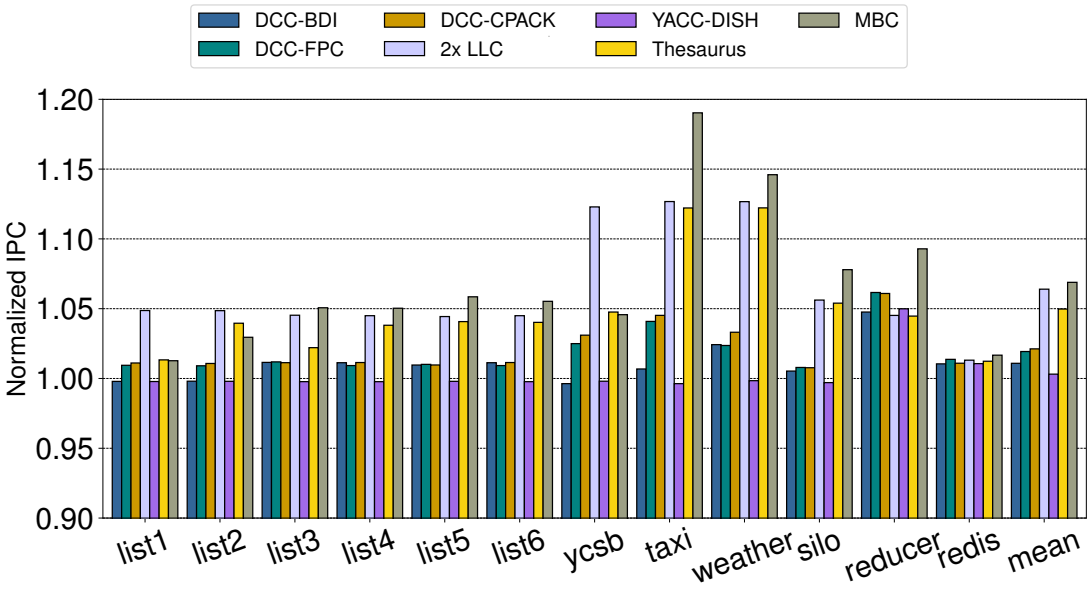



Figure 5.6: **Normalized Instructions Per Cycle (IPC)**

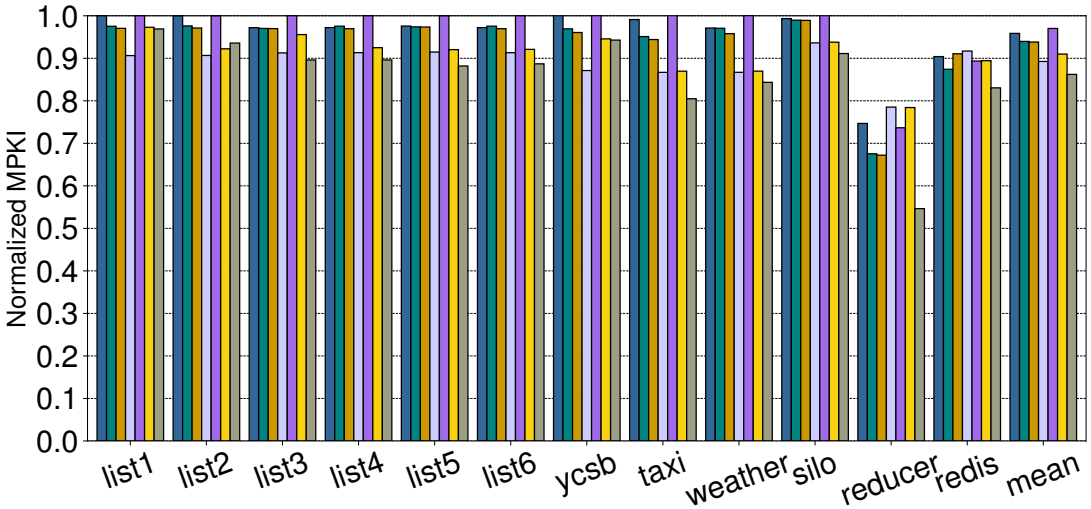



Figure 5.7: **Normalized Misses Per Kilo Instructions (MPKI)**

LLC size is computed by counting the number of logical lines in the LLC from snapshots taken for each 100M instructions.

Overall, MBC provides, over an uncompressed cache, 6.89% higher IPC, 14% less MPKI, and 2.47× larger effective LLC. MBC also slightly outperforms Thesaurus by 1.91% on IPC, 4.78% less MPKI, and 1.35× on effective LLC size (and up to 6.07% and 1.25×), while other designs fail to provide any sensible IPC benefit. For list workloads, it is quite obvious that Thesaurus is entirely insensitive to intra-block redundancy and zero distribution, as evidenced by the stable results between `list2`, `list4`, `list5`, and `list6`. MBC, on the other hand, is sensitive to both intra- and inter-block redundancy, and generally favors a higher percentage of zeros. This demonstrates the benefits of using our zero-optimized, dictionary-based algorithm, as it leverages all three types of redundancy. For `list1`, MBC slightly under-performs Thesaurus. This is

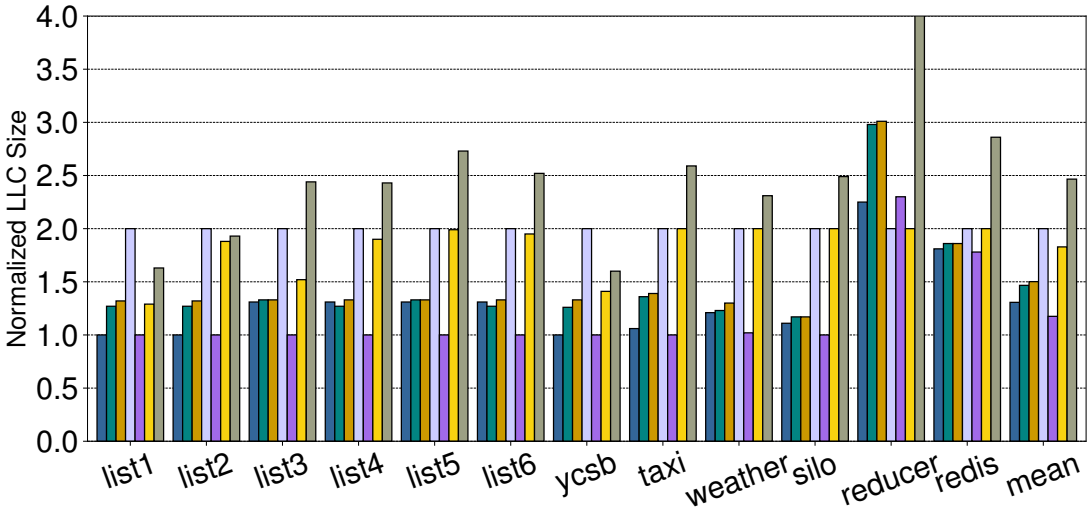



Figure 5.8: **Normalized Effective LLC Size**

caused by the extra overhead of the allocator and the lack of inter-block redundancy.

In `ycsb`, Thesaurus slightly outperforms MBC, despite a lower effective ratio. Further investigation reveals that this is likely caused by tag contention on the super-blocks, which evicted useful data early. The problem, however, is not unique to MBC, as all designs with super-blocks will suffer from tag contention. Prior research also proposed alleviations to this issue [24, 26, 203], which we leave as future work.

`taxi` and `weather` evidence the benefit of compressing over the stepped pattern with the software-provided "step size" for MBC, as MBC outperforms the rest of the designs by a large margin in terms of effective LLC size and MPKI. These two workloads also demonstrate high IPC improvement for MBC (19.0% and 14.6% IPC), due to being data-intensive and moderately compressible (2.59× and 2.31× effective ratio).

The three real-world workloads demonstrate varying behavior under MBC. `silo` is moderately compressible, with an effective LLC size of 2.51× and an MPKI of 90% over an uncompressed cache. As a result, it achieves an IPC improvement of 7.7%, which also outperforms the rest of the designs.

`reducer` is highly compressible with an effective LLC size of 4.10× and reduces the MPKI to only 54% of an uncompressed cache. The resulting IPC improvement is 9.3%, which almost doubles that of Thesaurus, the second highest one under comparison.

`redis`, despite its 2.74× effective LLC size and 18% reduction on MPKI, does not exhibit any significant IPC improvement. Further investigation reveals that `redis` has good access locality on its internal data structure, and hence LLC compression is generally ineffective, as evidenced by the fact that even doubling the LLC would not improve the IPC.

### 5.6.2 SPEC 2006 and 2017

**Experimental setup:** We also simulate all designs on a single core with SPEC 2006 [4] and SPEC 2017 [5]. We only report workloads with an L1 miss ratio no higher than 95%. The rest either have small working sets or high access locality, whose performance is not affected by

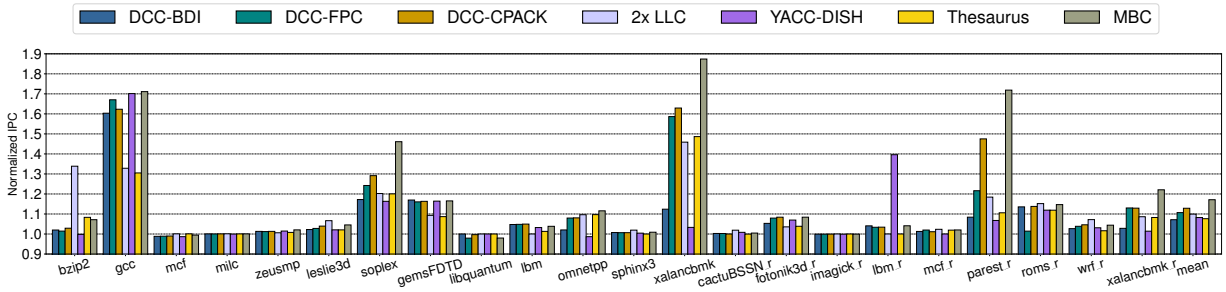



Figure 5.9: **Normalized Instructions Per Cycle (IPC), Single Core**

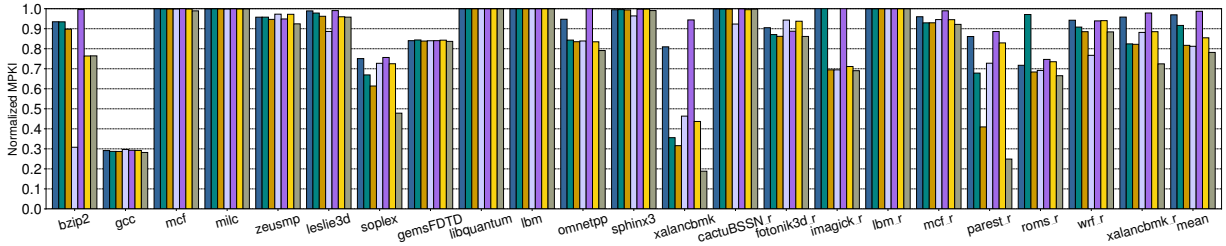



Figure 5.10: **Normalized Misses Per Kilo Instructions (MPKI), Single Core**

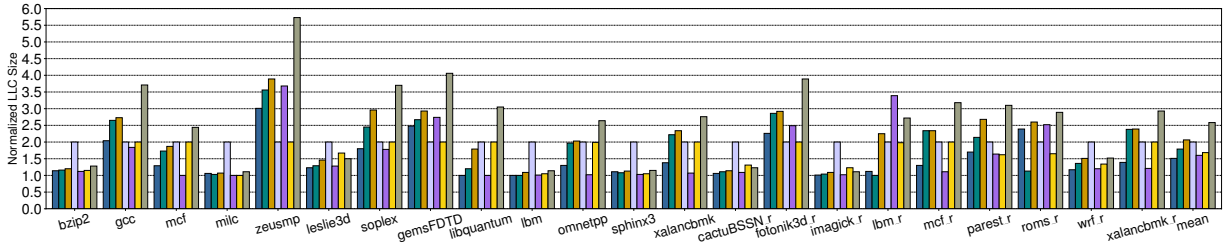



Figure 5.11: **Normalized Effective LLC Size, Single Core**

LLC compression. We evaluate MBC with the proposed memory allocator as in data workload simulation. For each workload, we simulate 10B instructions after skipping the first 100B. We report normalized instruction per cycle (IPC), misses per kilo instructions (MPKI) on the LLC, and effective LLC size in Fig. 5.9, Fig. 5.10, and Fig. 5.11, respectively.

**Results:** Overall, MBC brings an average of 17.1% speedup, outperforming all intra-block designs, reduces MPKI by 22%, and achieves an average effective LLC size of 2.58× the uncompressed. In particular, `gcc`, `zeusmp`, `soplex`, `gemsFDTD`, `libquantum`, `fotonik3d_r`, `mcf_r`, and `parest_r` demonstrate good compressibility with effective LLC size higher than 3.0× of the uncompressed, indicating the effectiveness of adopting over-provisioned, super-block tag design.

In contrast, Thesaurus with 2× tag over-provisioning, despite having an impressive raw compression ratio in some of the workloads, are limited by the number of tags, and therefore only achieved a maximum compression ratio of 2× and an overall IPC benefit of 7.65%. We also noted that, although increasing the over-provisioning of tags in Thesaurus may alleviate the tag pressure, this may introduce new problems, such as the base cache size (which we do *not* model) and high tag metadata cost.

In addition, compared with DISH, an inter-block compressed cache design using dictionary encoding, MBC achieves an average of 1.61× larger effective LLC and 9.12% higher IPC,

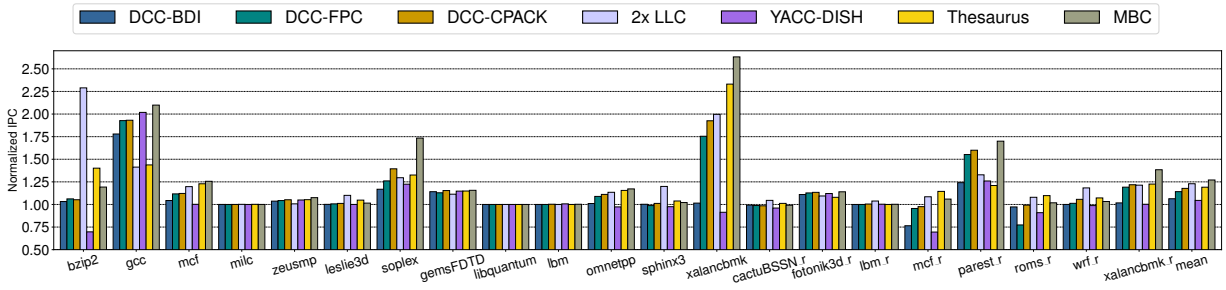



Figure 5.12: **Normalized Instructions Per Cycle (IPC), 8 Cores**

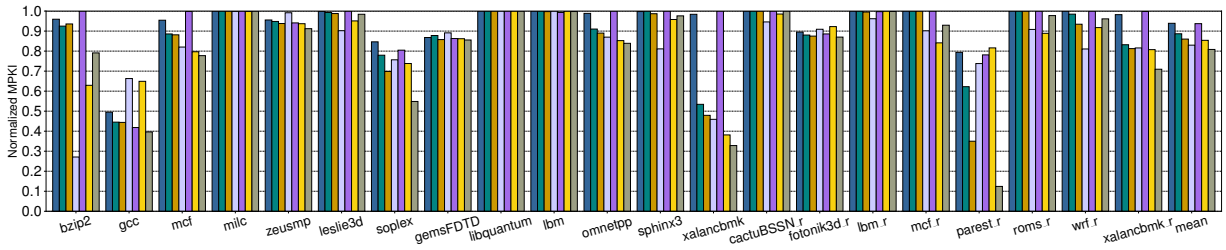



Figure 5.13: **Normalized Misses Per Kilo Instructions (MPKI), 8 Cores**

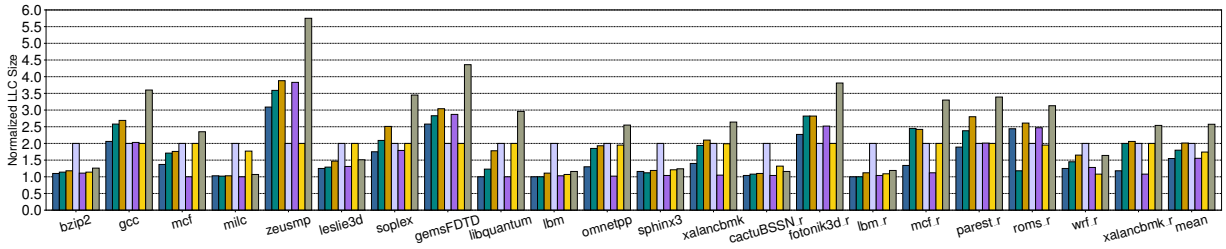



Figure 5.14: **Normalized Effective LLC Size, 8 Cores**

showing the effectiveness of its dictionary compression algorithm. Further investigation into raw compression ratio numbers (not shown) indicate that the eight-entry, explicitly stored dictionary of DISH often cannot encode all four blocks in a super-block, such that two or more tags are used for a single super-block, causing low data utilization.

#### 5.6.2.1 Multicore Results

To evaluate MBC in a multi-process environment, we simulated the design on SPEC with 8 cores each running an instance of the same workload. Each core executes 2B instructions after skipping the first 100B, for a total of 16B simulated instructions. We report normalized IPC, MPKI, and effective LLC size in Fig. 5.12, Fig. 5.13, and Fig. 5.14.

Overall, MBC achieves an average speedup of 27.1% over an uncompressed cache, reduces MPKI by 19.2%, and achieves an effective LLC size of 2.57×, outperforming all other designs under comparison. Compared with Thesaurus, which is the second-best design of all under comparison, MBC has an IPC advantage of 7.98%, incurs 5.8% less MPKI, and achieves 1.48× effective LLC size.

We observed similar patterns of the speedup trends between 8-core multi-process and single-core environments. We regard it as proof that MBC is advantageous over prior works even under multi-core contention.

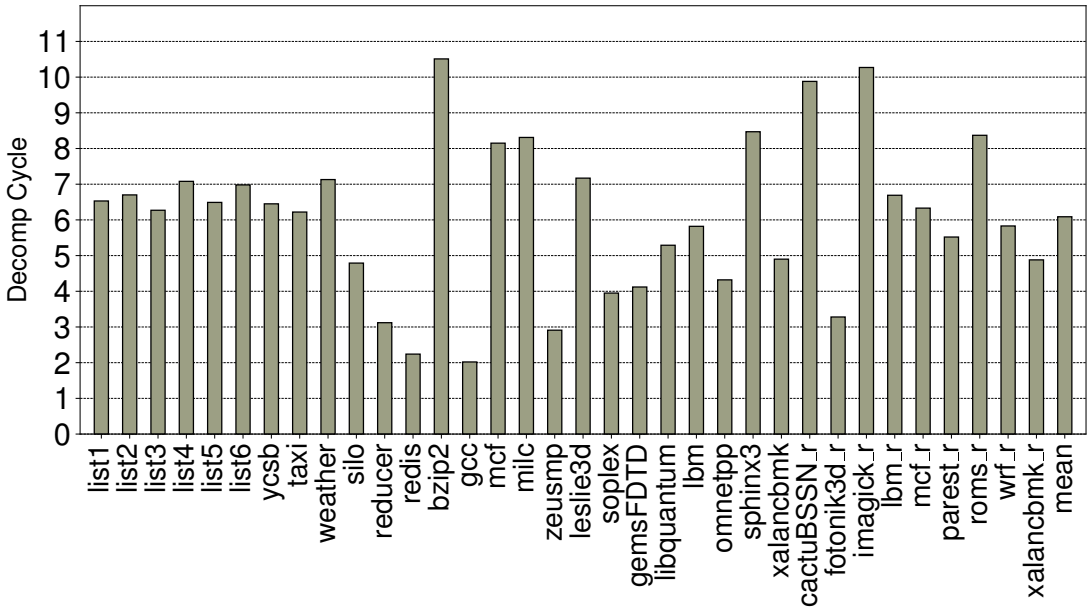



Figure 5.15: **Parallel Decompression Latency**

### 5.6.3 Parallel Decompression Latency

Fig. 5.15 shows the decompression latency of MBC on all single-core simulations. On average, MBC's decompression hardware takes slightly more than 6 cycles per decompression, which is lower than that of intra-block DCC-CPACK (9 cycles), as a result of zero optimization and parallel decompression saving the decoding bandwidth. Data workloads all have low decompression latency, due to the abundance of zeros of the in-cache data, as well as data blocks being perfectly aligned at cache block boundaries in some of the workloads. In the latter case, the two compressors can operate in lock steps without any bubbles. Among SPEC workloads, `bzip2`, `cactuBSSN`, `imagick_r`, and `lbm_r` demonstrate higher latency, as a result of being generally not compressible as well as lacking zeros. Compression may just be turned off in these cases to save energy and prevent the negative performance impact.

## 5.7 Additional Related Work

### 5.7.1 Compression Algorithm

Most single-block cache compression algorithms target compressing integers and are optimized for small blocks and low decompression latency. Many of these algorithms divide the input block into equal units called "words" and leverage repeated patterns (especially on high-order bits) between these words. FPC [12] and BDI [201] fall into this category. Bit-Plane Compression (BPC) [134] pre-processes a block with bit-plane transformation and then compresses words formed by each bit plane (bits from different words on the same position). FP-H [20], on the other hand, is designed to compress floating point numbers. It separately compresses the exponent and high/low mantissa fields of IEEE 754 floating point numbers as different streams because these fields usually have very distinct patterns and value ranges in floating point numbers. Zero compression only detects

zeros in the input block and encodes zeros with a separate zero-value mask. While limited in compression ratio, zero compression only requires a few cycles for compression and singe-cycle decompression, and is hence suitable for L1 cache compression [70, 110].

Some algorithms are also designed or can be leveraged for inter-block compression. C-PACK [47] combines LZ-like dictionary encoding [276] with pattern compression and can generally achieve a high ratio at the cost of longer operation latency. While C-PACK was originally designed as a single-block compression algorithm, inter-block compression is achieved by first generating the dictionary using the reference block, and then use the dictionary to compress the input block. SC2 [19] relies on a software-generated code frequency table as the dictionary, and compresses the input block using Huffman encoding [104]. Deduplication and bit-wise delta compression can also reduce the size of compressed blocks by using another block as a reference. These techniques, however, cannot leverage intra-block redundancy, and their compression ratio is heavily dependent on the similarity between the input block and the reference block.

### 5.7.2 Tag Array Organization

Depending on the compression ratio, a compressed cache can store varying blocks per set, and the total number of logical blocks stored in the cache can be much greater than the physical capacity. The challenge of designing the tag for compressed caches is hence twofold. First, the tag array must support a larger number of blocks than the physical capacity of the data array without incurring significant metadata overhead. Second, the tag array must also be able to support a varying number of ways per set. A poorly designed tag array will become the bottleneck in upper-bounding the maximum effective cache capacity, nullifying the numerous efforts to develop the compression algorithm (as noted in [84]).

The easiest way of organizing tags is to over-provision extra entries per set. Most designs adopt $2\times$ over-provisioning to achieve a maximum of $2\times$ effective cache capacity. Higher over-provisioning is theoretically possible but incurs considerable metadata overhead.

DCC [220] and YACC [222] both propose super-block tags, where a single tag entry encodes four consecutive blocks on the physical address space. Each logical block in the super-block has its own metadata bits, so cache and coherence actions still operate on individual blocks. Since every tag entry encodes four logical blocks, the super-block tag design supports $4\times$ effective cache capacity, with moderate metadata overhead. The super-block tags work optimally when access locality is high. Otherwise, the theoretical effective capacity cannot be achieved due to the under-utilization of tag entries.

Tag over-provisioning and super-blocks can also be combined to achieve even higher effective cache capacity. To the best of the author's knowledge, MBC is the first design that achieves a maximum of $6\times$ effective cache capacity by combining super-blocks with $1.5\times$ tag over-provisioning.

The last technique of tag organization is to allocate tag entries to each set dynamically on-demand. The V-Way cache [210] provides a solid ground where compression designs can be implemented. Other proposals that decouple tags from sets can also potentially serve the purpose [94, 227]. However, these designs radically change the cache's internal structures and introduce some difficulties to block management. As a result, they are rarely used for cache compression.

### 5.7.3 Inter-Block Main Memory Compression

Inter-block compression techniques can also be applied to main memory compression. Like inter-block cache compression, the major challenge of inter-block main memory compression designs is to search for analogous blocks efficiently, and the same "base cache" design has been repeatedly proposed. For example, in BCD Deduplication [198], inter-block redundancy is leveraged in a two-step process. In the first step, blocks with similar content are clustered into a hash table using fingerprint hashing. In the second step, the delta between the block and the hash table entry is generated, which is then deduplicated using a secondary hash table to reduce the compressed size of the delta further. In Global BDI [17] and Entropy-Based Compression (EBC) [133], the memory controller uses a global pattern table storing common patterns in the working set to dictionary compress incoming blocks to be written into the main memory. The pattern table serves as a word-level base cache and can be generated automatically or manually.

While using a similar base cache design as in cache compression, main memory compression faces a different set of design constraints, which makes the base cache design more feasible. First, the goal of main memory compression is usually to reduce DRAM's internal bus bandwidth or to enable higher capacity. As a result, memory compression designs are less sensitive to the extra energy and storage overhead incurred by the base cache. Second, DRAM-based main memory has much higher access latency than SRAM-based base caches. The base cache can, therefore, be made much larger, and scalability is less of a concern in this case.

### 5.7.4 Approximate Cache Compression

Approximate cache compression operates similarly to cache deduplication, but instead of only deduplicating cache blocks with the exact same content, it also deduplicates blocks whose contents are *approximately identical* to each other. Compared with regular cache compression that precisely restores the content of the blocks on decompression, approximate compression allows certain minor errors to be added to decompressed data in exchange for a higher compression ratio and more performance benefits.

Doppelgänger [172] is an approximate compression design that uses fingerprint hashing to cluster similar blocks into a hash table. Blocks that are mapped to the same hash table entry are assumed to contain similar content and are hence "deduplicated" by only storing a reference to the existing hash table entry. Users must manually specify the data type and value range within blocks to be compressed, such that the fingerprint hashing hardware can effectively generate the fingerprint for each block. Doppelgänger is suitable for running machine learning applications, where the imprecision of data can be well tolerated.

# Chapter 6

# Memento: Architectural Support for Ephemeral Memory Management

This chapter of the thesis demonstrates, from a different perspective than the previous chapter, how leveraging application-level information helps optimize one of the most common tasks of an application—memory management. To this purpose, we present Memento, a holistic, hardware-centric approach to memory management that eliminates almost all instruction overhead from the memory management critical path (e.g., `malloc` and `free`) on a special class of cloud workloads named serverless computing.

Serverless computing has become an increasingly popular cloud paradigm with services like AWS Lambda [14], Azure Functions [171], Google Cloud Functions [89], and IBM Cloud Functions [106]. Built on top of serverless computing, *Functions-as-a-Service* (FaaS) enables a multitude of applications by decomposing them into small code snippets (i.e., functions) that are invoked by user-specified trigger events [76, 119, 120, 200, 216]. The benefits of this approach include enabling cloud providers to automatically provision resources while charging users for only their actual resource consumption at a millisecond granularity [15]. Notably, these functions are very short-lived—typically finishing in less than a second [229]—which poses new challenges in system design.

Memory management is responsible for a major chunk of data center cycles as revealed by Google's internal profiling [125], despite significant efforts in optimizing them in software [74, 145]. Unfortunately, while long-running workloads have the opportunity to amortize some of their memory management costs over their long runtimes, this is not the case for short-lived serverless functions. Prior work has focused on reducing the cold-start effects of serverless workloads [10, 36, 68, 78, 191, 229, 247]. However, memory management still relies on expensive allocation and deallocation paths on the critical path of function execution, leading to major overheads in serverless environments.

In particular, functions pay the full cost of memory allocation and deallocation in both user space and the OS on the critical path of their short runtime. For example, consider the overheads of typical memory management operations. For starters, applications must pay the cost of memory management in *user space*, where each allocation and free typically require tens of instructions in popular high-level languages for serverless environments (e.g., Python). Furthermore, the user space allocator relies on the OS to allocate physical memory. However, the allocation path in

the OS, either performed at system call time, such as `mmap` or later on at access time through page faults, is inherently expensive. This is because of the creation of page tables and the actual physical memory allocation and bookkeeping, requiring additional thousands of instructions.

In this chapter, we focus on reducing the performance overhead of memory management for serverless functions. To better understand the characteristics of memory management in serverless functions and to expose opportunities for optimization, we first start with a detailed investigation of function behavior across several functions and three language runtimes that cover a wide spectrum of memory management behavior. Our analysis reveals three key insights. First, the vast majority of allocations are relatively small, no larger than 512 bytes. This characteristic is partly due to the usage of high-level languages, where small objects are used extensively. Second, the allocation lifetime is bimodal based on the language runtime. In particular, objects are either allocated and freed shortly after, usually within less than 16 other allocations of the same size class. Alternatively, allocations are often not freed due to the short runtime of functions and they are instead batch-freed by the OS when the function exits. Finally, memory management spends a significant amount of cycles in user space and interactions with the OS.

Based on these insights we propose *Memento*, a holistic hardware-centric design that optimizes memory management by moving most of the work on the software critical path to specialized hardware. Memento introduces two key mechanisms that operate in tandem. The first is a *hardware object allocator* that tracks objects using *arenas* and performs memory allocations and frees in a per-core metadata cache. Due to the small sizes of allocations, a small number of size classes can be efficiently cached. Furthermore, for allocations that are allocated and quickly freed allocation metadata have strong temporal locality. Therefore, the hardware object allocator can satisfy most requests entirely in the metadata cache within only a few cycles.

The second mechanism in Memento is a *hardware page allocator* that manages a small pool of free virtual and physical pages. The hardware page allocator serves the dual purpose of (i) replenishing the hardware object allocator with free virtual pages that constitute arenas; and (ii) backing virtual pages with physical memory on-demand when a virtual page is accessed for the first time. The hardware page allocator removes the kernel path of memory management from the critical path of function execution, thereby avoiding invoking expensive system calls and page fault handlers. Furthermore, for allocations that are batch-freed at the end of function execution, the hardware page allocator can deallocate their memory with low latency.

Overall, these two mechanisms work together in a complementary fashion that resembles the user space and kernel software path in the current software stack. Memento naturally integrates with the existing software stack by adding two new instructions to the ISA for the allocation and deallocation of memory. The hardware design and interface of Memento enable seamless integration with both compiled and interpreted languages, including those with and without garbage collection.

## 6.1 Background and Opportunities on Memory Management

This section first provides a brief background on memory management across user space and the OS. Then we showcase the memory behavior of serverless workloads.

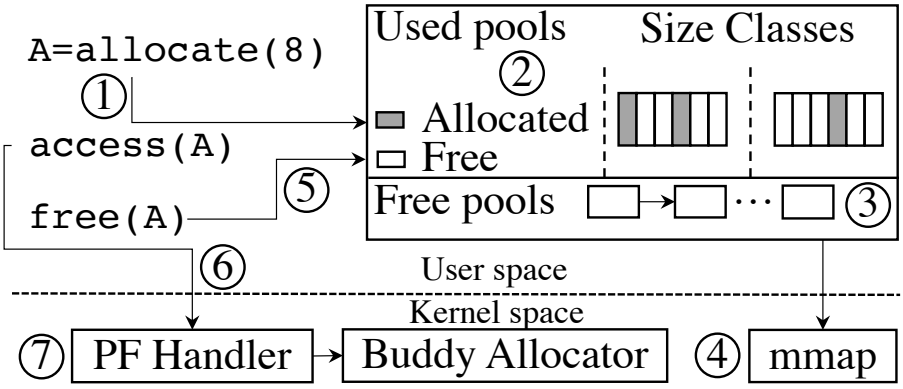



Figure 6.1: Memory Management in User Space and Kernel

### 6.1.1 A Day in the Life of a Memory Allocation

Memory management of an application is split between user space and OS operations. Typically, variable byte-sized allocations are performed in user space while the OS manages memory resources at the granularity of pages. Referencing Fig. 6.1, we describe the steps and effects of memory allocation and free, based on a typical example of user space memory allocation, and then shed light on the steps performed by the OS.

**User space operations:** We use the CPython implementation of the heap allocator, `pymalloc` [3], as an example to describe the allocator procedure. Allocators in other high-level language runtimes also follow similar steps. The allocator requests memory from the OS at the granularity of 256KB *arenas* and then splits them into smaller 4KB *pools*. Free objects within a pool are organized as linked lists, and each pool serves allocation requests of a particular size class.

On an allocation request, the size class is computed by aligning the requested size up to the nearest 8-byte boundary (Step ① in Fig. 6.1). Then, the allocator checks the per-size class free pool list, and if a pool with free objects is present, the head entry of the free list is returned to the caller (Step ②). However, if no free objects can be found, the allocator attempts to grab a new free pool from the free pool list (Step ③) and tries again. If there are no free pools, `mmap` in the kernel is called to allocate more arenas (Step ④). Allocation requests that are larger than 512 bytes by default are directly serviced by `malloc` in glibc, which eventually calls `mmap` as well.

On a free operation (Step ⑤), the pool header is first obtained by aligning the address down to the nearest 4KB boundary. Then the object is returned to the pool by linking it to the head of the free list. If the free operation turns the pool entirely free, then the pool is returned to the free pool list. Finally, if all pools in an arena become free, the allocator returns its memory by calling `munmap`.

**Kernel space operations:** User space allocators request memory allocations from the kernel through system calls. Continuing the above example, when the `mmap` [166] system call is invoked (Step ④), it performs the following. First, `mmap` finds an unused region of addresses in the virtual address space of the process and then sets up mapping metadata describing the allocation. Note that no physical storage backs the virtual address region allocated in this stage. Instead, the system call only returns the region's start address to the user space without setting up any virtual-to-physical mapping.

As a result, when the software accesses a newly allocated virtual memory page for the first

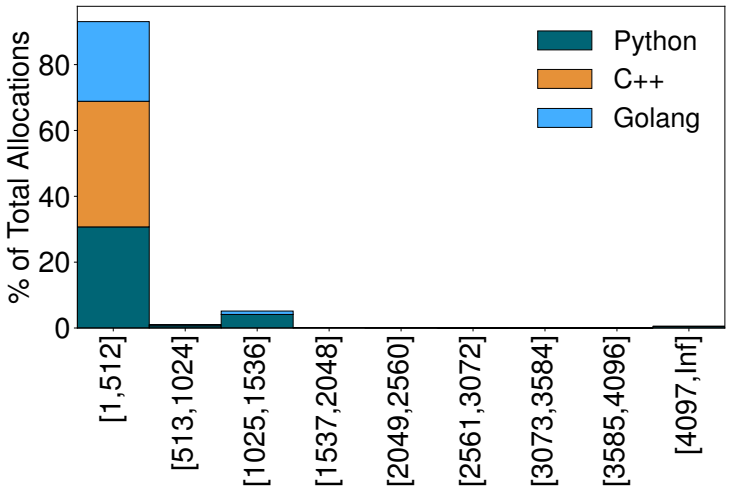



Figure 6.2: Allocation Size Distribution (Bytes)

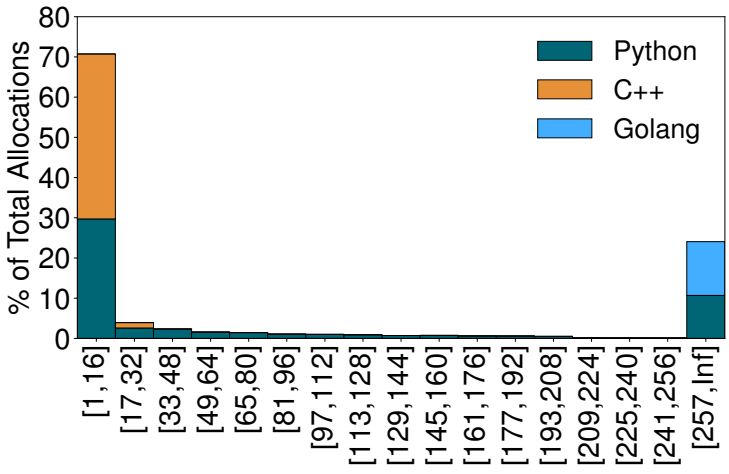



Figure 6.3: Allocation Lifetime Distribution (Malloc-Free Distance)

time, a page fault will be raised (Step ⑥) by hardware due to lacking a valid address mapping. The page fault is serviced by a kernel handler, which finds the mapping metadata set up earlier by `mmap` on the faulting address. In this stage, the handler performs physical memory allocation by requesting a free physical page from the kernel's physical page allocator and then setting the corresponding page table entry to point to the physical page (Step ⑦). Eventually, the faulting memory access is retried and will land on the newly allocated physical page.

Applications deallocate memory through the `munmap` [166] system call that performs the opposite steps. First, `munmap` tears down the mapping metadata. Then it walks the page table entries describing the mapping, clears page table entries, and returns physical pages to the kernel as needed. Finally, if relevant page tables become empty, they are also freed.

### 6.1.2 Memory Management Behavior of Serverless Functions

Short-lived functions pose new challenges to memory management. In this section, we study the memory behavior of function execution across serverless workloads based on four suites [54,

79, 132, 192] and three languages, Python, C++, and Golang. We further discuss our workloads in detail in Section 6.4.1. To characterize the allocation sizes and lifetimes of functions we instrumented the allocators of each language and collected allocation traces. We normalize the number of allocations of each function, then we aggregate across functions, and show the per language breakdown.

Fig. 6.2 shows the object size distribution in 512-byte increments. The results show that allocations are small. Specifically, 93% of allocations are smaller than 512 bytes. For several workloads, small allocations can account for even more than 98% of all allocations, making large allocations a rare occurrence. Size distributions within 512 bytes are heavily workload-dependent and we did not observe any consistent patterns for small allocations across the workloads. Finally, small allocations are preferred across Python, C++, and Golang.

To characterize object lifetimes, we define a lifetime metric based on the malloc-free distance. In particular, we compute the number of allocations of the same size class before the object is freed. This is a good predictor of allocation metadata locality. Fig. 6.3 shows the average lifetime distribution of object allocations. As we can see from the results, serverless function allocation lifetimes exhibit a bimodal behavior. Specifically, 71% of allocations are short-lived as they are freed within only 16 allocations of the same size class. On the other hand, 27% of allocations are long-lived and rely on OS deallocation when the function exits. For object lifetime we see that different languages exhibit different behavior. Specifically, for C++ the majority of allocations are short-lived while for Python they are primarily short-lived allocations with a few long-lived ones. Finally, Golang allocations are long-lived. This is because garbage collection is not invoked due to the short runtime of functions. As a result, allocations are batch-freed at the end of function execution.

In Table 6.1 we further show the joint distribution of allocation size and lifetime. On average, 61% of allocations are *both* small and short-lived. These objects are small in size and freed quickly after allocation. For several workloads, short-lived and small allocations account more than 80% and up to 96% of all allocations. Furthermore, 32% of allocations are long-lived and small. Finally, larger allocations account for 7% of total allocations with the majority being short-lived.

| | **Small** | **Large** |
|---|---|---|
| **Short-lived** | 61% | 6.55% |
| **Long-lived** | 32% | 0.45% |

Table 6.1: Combined Distribution of Size and Lifetime

**Insights:** The above observations illustrate the two most prominent patterns in the memory allocation behavior of serverless workloads. First, most objects are small and under 512 bytes. In addition, workloads exhibit varying patterns in object size distribution under 512 bytes. Therefore, the allocator should optimize for small and varying size class distributions under 512 bytes. Second, most objects are freed shortly after allocation, indicating that they are short-lived. The allocator should prioritize short-lived object handling by taking advantage of the strong allocation metadata locality. Furthermore, long-lived allocations should be handled efficiently by the OS allocator instead.

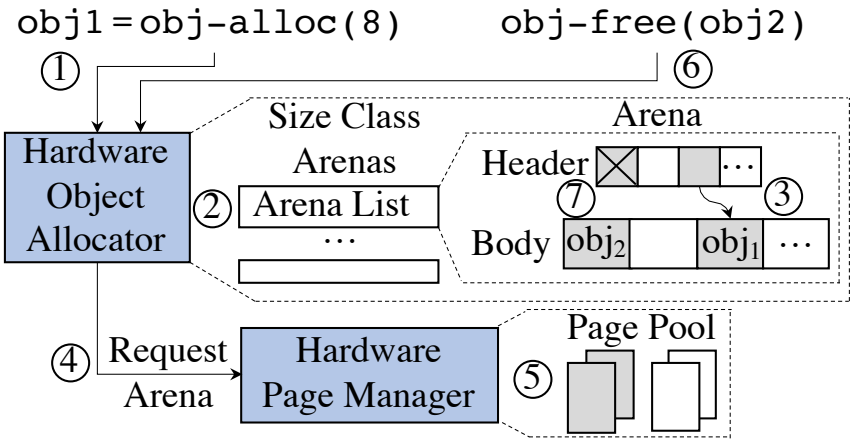



Figure 6.4: Workflow of memory management with Memento

Based on these insights, we design Memento, a holistic hardware-centric approach to eliminating the overheads of serverless memory management. To handle varying patterns of object sizes under 512 bytes, Memento maintains multiple size classes. Allocations in each size class are satisfied by the dedicated metadata maintained for that particular size class. Furthermore, Memento leverages the strong temporal locality of allocation metadata by adding a hardware object allocator that caches allocation metadata near the processor, fulfilling most allocation requests in only a few cycles. Finally, Memento's hardware page allocator facilities the needs of the object allocator and efficiently recycles pages that contain long-lived objects at the end of function execution.

## 6.2 Memento Design

The design of Memento consists of two mechanisms that operate in tandem to handle user space and kernel memory management operations. The first mechanism is a *hardware object allocator* (located close to the core) responsible for arena-based object allocation and deallocation. The hardware object allocator interfaces with the application and provides similar abstractions as memory management functions in user space (e.g., `malloc` and `free`). In addition, Memento also introduces a *hardware page allocator* at the memory controller managing physical pages. The hardware page allocator interfaces with the hardware object allocator similarly to kernel memory management system calls by replenishing the object allocator with physical memory on-demand. Overall, Memento offloads *both* user space and kernel execution paths for memory management to their hardware counterparts and overlaps these paths with processor execution. Consequently, Memento is capable of satisfying most memory management requests with a latency equivalent to single roundtrip to the L1 cache, speeding up function execution by reducing the amount of work the processor needs to perform.

Fig. 6.4 depicts a high-level overview of Memento's workflow and how the two mechanisms cooperate to perform memory management. First, upon an allocation (step ①), the hardware object allocator will identify the size class of the request. In step ②, the corresponding arena from the arena list will be selected. Next, in step ③, the hardware object allocator (which caches arena headers) will identify and mark a free location within the arena's header where allocation

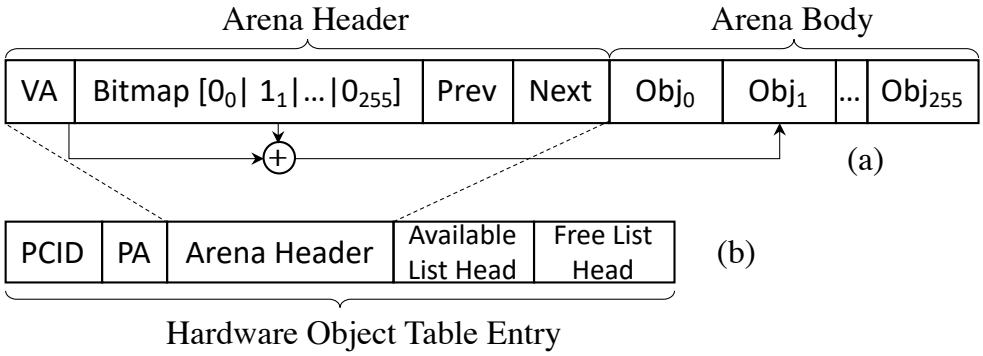



Figure 6.5: (a) Layout of arena header and body, and (b) Layout of hardware object table entries

metadata is stored. The arena metadata can be used to identify the location of the body of the arena storing the allocated objects. Finally, the allocated virtual address is returned to the caller.

In the scenario that an arena containing free objects is unavailable, the hardware object allocator requests an arena from the hardware page allocator, as shown in step ④. The hardware page allocator then allocates the appropriate number of pages from the page pool (shown in step ⑤) and returns the new arena to the hardware object allocator.

Finally, upon a free operation (step ⑥), the hardware object allocator performs the reverse operations that undo the object allocation. Specifically, the allocator will clear the appropriate metadata entry of the arena header, as shown in step ⑦. Similarly, the hardware object allocator notifies the hardware page allocator to free arenas and then returns pages to the free page pool.

### 6.2.1 Hardware Object Allocator

Conceptually, the hardware object allocator performs user space-level memory management operations. It provides the abstraction of an object allocator through an interface similar to the conventional `malloc` and `free`. To realize this interface, we extend the ISA with two new instructions for allocating and freeing objects: `obj-alloc` and `obj-free`. The `obj-alloc` instruction carries the requested allocation size as an operand. When executed, it returns the virtual address pointing to a memory block that is available for use and satisfies the requested size. The `obj-free` instruction has the virtual address to be freed as its operand, and it deallocates the block so that future allocations can reuse the block.

We now discuss the two main components of the hardware object allocator: (i) the in-memory *arenas* that track object-level allocations, and (ii) the *hardware object table* that facilitates arena management in hardware.

**Arenas:** The object allocator tracks the allocation status of memory addresses by maintaining bookkeeping information in the unit of *arenas*. An arena is a consecutive range of virtual addresses. An arena serves allocation and free requests of only one specific size class during its lifetime. Fig. 6.5(a) shows the arena layout, which includes the header and the body.

An arena header includes three parts: (i) a virtual address (VA) field, (ii) an allocation bitmap, and (iii) two pointers for forming doubly-linked lists of same-size-class arenas. The VA field stores the base virtual address of the arena in the address space to which the arena belongs. The bitmap is used for tracking the allocation status of objects. A "1" bit in the bitmap indicates that

the object at the corresponding offset has been allocated. The *arena body* is an array of objects of the same size. The combination of the VA field with the offset of a bitmap entry points to the allocated object in the arena body. Each arena contains a fixed number of objects. In our experiments, we set this parameter to 256 objects per arena, balancing metadata cost and internal fragmentation and working well in practice.

Arenas are organized into *arena lists* for each size class. Specifically, two lists are maintained per size class. The first is an *available* list tracking arenas with at least one free object, and the second is a *full* list tracking arenas without any available objects. An arena can be in at most one of the two lists at any time. In addition, arenas are connected to the appropriate list through the *prev* and *next* pointers. Fig. 6.6 shows an example of the two arena lists.

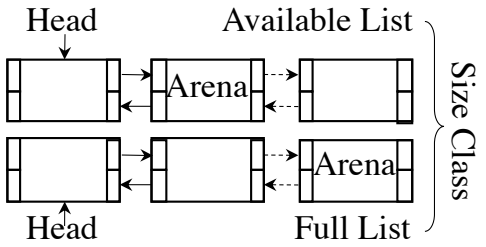



Figure 6.6: Per-size class *available* and *free* arena lists

**Hardware object table:** The core of the hardware object allocator is the *Hardware Object Table (HOT)*, which is an array of HOT entries indexed by size class index. The HOT holds the most recently used arena header for each size class. Based on our analysis of the size distribution of allocations, we opt to support allocations up to 512 bytes in 8-byte increments, resulting in a total of 64 size classes. Fig. 6.5(b) shows the HOT entry layout. Each entry stores a cached copy of the arena's header fields, which are loaded from memory. Additionally, to support operations on the available and full list, HOT entries also contain the following: (i) the *PA* field storing the physical address of the arena, and (ii) two pointers, the *available list head* and *full list head* pointers, which store the physical addresses of the first arenas in the available and full list of the size class, respectively. Finally, each entry is tagged with a process context identifier (PCID) that distinguishes arenas of different address spaces. This field can facilitate resource sharing on HOT for simultaneous multithreading (SMT).

The hardware object allocator manipulates the hardware object table for initialization, allocation, and free operations following the steps in Fig. 6.7.

**Initialization:** On initialization (①), an arena is allocated for each size class by requesting free pages from the page allocator (②). We further describe the design of the hardware page allocator in Section 6.2.2. Individual arenas are initialized by preparing the arena header (③) via clearing the bitmap and the linked list prev and next pointers. The arena header's VA field is set to the virtual address assigned to the arena. The PA of the HOT entry is set to the header's physical address returned from the page allocator. The available and full list head pointers are set to indicate that both lists are empty. Finally, the initialized arena header is loaded into the corresponding HOT entry of the allocator (④).

**Allocation:** On an allocation request (⑤), the size class is computed by rounding the requested size up to the nearest 8-byte boundary. Then the HOT entry is located swiftly using the

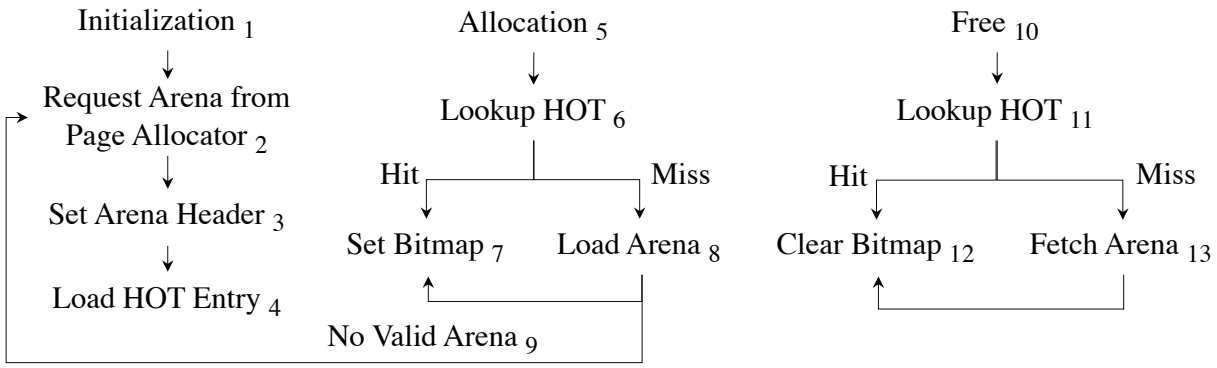



Figure 6.7: Hardware object allocator steps for initialization, allocation, and free operations.

size class as an index without any associative search (⑥). Next, the bitmap is scanned. Three cases might occur in this step which we discuss as follows.

In the most common case, a zero bit is found, and the allocation completes quickly by setting the appropriate bit (⑦). The allocator computes the address of the allocated object based on the entry's VA field and the position of the zero bit in the bitmap before returning the computed address to the processor. We refer to this scenario as an allocation "HOT hit".

In the rare scenario that a zero bit cannot be found, indicating that the arena is currently full, the hardware will then use the available list pointer to load the next arena header into the HOT entry from the memory hierarchy (⑧). Meanwhile, the hardware replaces the existing entry by writing it back to the corresponding memory location using the *PA* field of the replaced entry. The hardware also performs two additional list operations. First, the full arena being replaced is inserted into the full list as the head. Furthermore, the newly loaded arena is removed from the head of the available list.

Finally, if the available list field indicates that no valid arena exists in the list, a new arena is allocated and loaded into the HOT entry by requesting more pages from the page allocator (⑨). The new arena is initialized following the initialization procedure described earlier. Finally, the current full arena is inserted into the full list. We refer to the last two scenarios as an allocation "HOT miss".

As an optimization, the hardware allocator may *eagerly* load an available list arena or requests a new arena at the moment the last valid object of the current HOT entry is allocated. The hardware allocator can hide the potential latency of HOT misses in this manner.

**Free:** On a free request (⑩), the hardware identifies the appropriate arena by calculating the base virtual address and size class of the arena to which the object belongs. This step is performed using the operand of `obj-free`, which is the virtual address of the object to be freed. The calculation only requires simple bit operations, as we will discuss shortly in Section 6.2.2.

After obtaining the size class, the arena's base address is compared to the VA field of the corresponding HOT entry (⑪). In the common case, the VA matches, and the free "hits" in the HOT. The appropriate bit in the bitmap is cleared, and the free operation completes (⑫). If the VA does not match, the free "misses" in the HOT, and the following steps are performed (⑬). First, the hardware allocator translates the arena's base address to the physical address by requesting a translation from the TLB. Next, the hardware allocator fetches the header from the memory hierarchy using the physical address from the previous step. Finally, the hardware clears the

appropriate bit in the header's bitmap and writes the header back.

In the case of a HOT miss, if the arena is in the full list (having all-ones in its bitmap) before the free operation, then the arena will be moved to the available list. The hardware removes the arena from the full list and inserts it into the head of the available list by updating the appropriate pointers.

### 6.2.2 Hardware Page Management

The page allocator is a hardware component located on the memory controller, and it interfaces with the object allocator for page-level operations. The two primary responsibilities of the page allocator are: (i) allocating arenas to the object allocator in the virtual address space, and (ii) managing a small pool of physical pages and using them to allocate backing storage for arenas. These two responsibilities roughly correspond to `mmap` and the page fault handler in the OS kernel as we have seen in Section 6.1.1.

**Managing arena virtual addresses:** On the creation of each process that uses Memento, the OS reserves a region of virtual addresses from the process's address space. The OS exposes the begin and end addresses of the region to the hardware through special *region control registers* on the core and the memory controller, which we refer to as *Memento Region Start (MRS)* and *Memento Region End (MRE)*. These two registers are managed in a multicore system at a per-address space level, i.e., each executing process maintains its private pair of registers backed by per-process memory locations. The OS is responsible for spilling and loading the region control registers into and from memory on context switches.

Memento divides the reserved virtual address region evenly into 64 size classes. This design decision enables the hardware object allocator to calculate the size class and the arena's base address by simple bit operations, answering the pending question in Section 6.2.1. Given an object's address, the hardware calculates the size class by dividing the address offset in the reserved region by 64. The arena base address is calculated by further rounding the offset within the size class down to the arena size of that particular size class. Note that the second operation can be implemented on hardware efficiently because the arena sizes are known in advance.

The first responsibility of the page allocator is to allocate virtual addresses to arenas. This scenario occurs when the object allocator runs out of arenas and requests a new one from the page allocator. In Memento, an arena can consist of single or multiple pages depending on the particular size class of the arena. Therefore, the page allocator maintains a per-size-class pointer as the starting address of the next allocation. On receiving an allocation request for a new arena, the page allocator bumps the pointer of the requested size class forward by the arena size of that size class. In addition, a physical page is also eagerly allocated to back the first page of the arena containing the header. Both the virtual and physical addresses are sent back to the object allocator, which then uses them to initialize the VA field of the arena header and the PA field of the HOT entry, respectively.

On a multiprocessor system, the page allocator maintains per-size-class pointers for each core in a reserved memory block, and enables fast access to frequently used entries with a small on-chip cache called the *Arena Allocation Cache (AAC)*. The AAC is direct-mapped using core IDs as indexes, and each entry stores the pointers for a few frequently used size classes. In practice, a

small number of size classes per workload is sufficient to cover most of its memory allocation activities.

**Assigning physical pages to arenas:** The second responsibility of the page allocator is to assign physical pages to the arenas handed out to the object allocator. For this purpose, the page allocator implements two more components. The first is a simple physical page pool consisting of free physical pages replenished by the OS on-demand. The second is a hardware-managed page table that tracks the virtual-to-physical address mapping for arenas. We next describe how Memento's hardware page allocator maps arenas to the physical address space.

As mentioned earlier, when the hardware page allocator gives out new arenas, the page allocator only physically backs their first page. This design decision is deliberately made to simplify the operation of the object allocator, as the object allocator will initialize the metadata located on the first page of the arena right after an arena is allocated. However, the page allocator does not back the rest of the virtual addresses in the arena. Instead, physical pages are assigned to these virtual addresses only on the first access, reducing potential memory waste.

When arena memory is accessed for the first time, the MMU attempts to translate the accessed virtual address, which will incur a TLB miss and a subsequent page walk. Then the MMU will first check whether the virtual address lies in the process's reserved Memento address region by comparing the requested virtual address against the pair of region control registers. Supposing that the address lies in the Memento address region, the MMU then conducts the page walk from a different page table root address, specified by a *Memento Page Table Root (MPTR)* register (instead of the regular one, e.g., `CR3`). The MMU issues page walk requests marked with a special flag that will be identified by the page allocator.

The hardware page allocator manages a Memento page table for each process similarly to the kernel. The only exception is that the page allocator constructs the Memento page table on page walk requests and automatically expands the table when encountering invalid entries.

On receiving the page walk request, the page allocator fetches and returns the Memento page table entry using the physical address in the walk request, if the entry is valid. If, however, the valid bit of the entry is cleared, meaning the virtual address is currently not mapped, then one of the following happens before the page allocator returns the entry. In the first case, the walk is on the leaf level. The page allocator will proceed to allocate a new physical page and populate the leaf entry with the page's address. Otherwise, if the entry is higher up in the page table tree (e.g., it could be a PGD, PUD, or PMD entry on x86), the next level of the Memento page table is allocated from the pool and zeroed out (i.e., all entries are invalid). Eventually, all levels on the page walk path will be populated, after which the page walk concludes.

The TLB inserts the mapping after the page walk is complete. Future memory access hitting the entry will proceed as any other memory access. Moreover, future page walks on the arena address will not cause new allocation since both the Memento page table and the address are backed by physical memory already.

Memento hard codes the permissions of all arena pages to be readable, writeable, and non-executable. The page allocator sets this permission combination in the mapping entry returned to the page walker. While other permission combinations are also generally valid, in Memento, we choose only to support the sole case of heap memory allocation, where this combination alone is sufficient. This design decision dramatically simplifies Memento by avoiding complex use cases such as file-backed allocations.

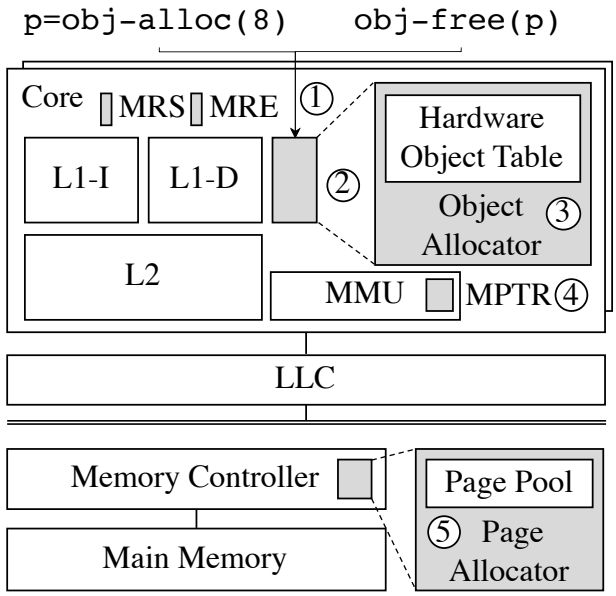



Figure 6.8: Memento Design Overview

The page allocator also handles arena frees, which occur when the object allocator has freed the last live object within an arena. On receiving an arena free, the page allocator walks the Memento page table and reclaims physical pages backing the arena. The corresponding page table entries are also invalidated, with page table pages freed if the last valid entry on that page is invalidated.

Similar to `munmap`, when freeing an arena, TLB shootdowns are also sent to cores that have issued page walk requests on the address space in the past. To identify which cores should receive the shootdown request, the hardware page allocator tracks per-process shootdown information using a hardware bit vector, the length of which equals the number of cores in the system. Each bit in the vector represents whether the corresponding core has issued any page walks on the address space. On arena frees, the page allocator scans the bit vector and sends TLB invalidations to the core whose bit is set. Note that typical serverless workloads are single-process and single-threaded, and as a result, this procedure's overall cost is negligible.

Overall, the hardware page allocator in Memento enables applications to acquire memory from the Memento address region without incurring any OS costs, saving on both context switch and kernel code execution.

### 6.2.3 Putting It All Together

The overall design of Memento is shown in Fig. 6.8. The main components of Memento are depicted in gray. The OS exposes the Memento memory region to the processor through control registers *MRS* and *MRE* (①). Applications communicate with Memento through the ISA extensions `obj-alloc` and `obj-free` that allocate and free objects (②). The hardware object allocator of Memento (③) is responsible for managing the memory arenas and the corresponding arena lists. The MMU further maintains an additional control register, the *MPTR* (④), that enables page walks to traverse the page tables that belong to the Memento memory region. To allocate

new arenas, the hardware object allocator requests free pages from the hardware page allocator that lives in the memory controller (⑤).

## 6.3 Discussion

**Integrating with software memory allocators:** Memento focuses primarily on small object allocations within 512 bytes. Larger allocations, which are rare in serverless workloads, are handled by software. We propose two approaches that enable Memento to collaborate with software allocators on this matter. The first approach is to let `malloc` check the allocation size. If the size is within 512, the software `malloc` will use Memento to fulfill the allocation. Similarly, `free` will check whether the object pointer to be deallocated lies in the Memento memory region and, if positive, use Memento to execute the free. An alternative approach is to expose the address of the software allocation and free function calls and let Memento redirect control accordingly. For simplicity, our design opts for the first approach so that the existing `malloc`/`free` interfaces exposed to the application remain unchanged.

For C and C++ applications, the memory allocator is mostly likely dynamically linked at load time. Therefore, deploying Memento does not require rewriting existing applications, and instead, Memento can be deployed as lightweight software libraries that override the default glibc allocator. For language interpreters and runtimes, developers can choose to release Memento as an opt-in feature to provide a better performance, which can be enabled or disabled on users' requests. Memento itself can also be integrated easily with various memory allocators such as malloc in glibc [155], jemalloc [115], and TCMalloc [85]. We demonstrate three examples of integrating Memento into existing memory allocators in our evaluation, namely garbage-collected Python memory allocator pymalloc [3], Google's jemalloc [115] for C++, and garbage-collected Golang runtime allocator.

**Multi-Process and multi-threaded support:** Current serverless workloads are single-process but multiple independent functions can be collocated on the same server. To support multi-tenancy the OS flushes the HOT table before a process is context switched. As we will see in Section 6.4.7, this operation is inexpensive as the HOT table is small.

Moreover, function workloads are single-threaded. Nevertheless, Memento supports multi-threaded applications. In particular, each thread manages its own arenas in the object allocator as part of its context. The page allocator shares the virtual address space and page address mapping among threads from the same process. On context switches, the OS populates the control registers of Memento such that threads from the same address space will use the same *MPTR* for page walks and the same sets of arena virtual address pointers for arena allocations.

In the rare case that an object is allocated by one thread and deallocated by a different thread, Memento must be able to tell that the object is managed in a different thread's arena. in Memento, this can be achieved by piggybacking on regular cache coherence. Specifically, when a HOT misses and reads an arena header from the memory, it issues the request to the cache hierarchy. Then, the cache coherence will locate the arena header's current owner and forward the request appropriately. On receiving the coherence message, the HOT controller will first read the size class index using the physical address, and then invalidate the corresponding entry before responding with the line.

| | |
|---|---|
| **CPU** | 4-issue OOO, 3 GHz, 256-Entry ROB, 64-Entry LSQ |
| **TLB** | L1 64-Entry, 4-Way; L2 2048-Entry, 12-Way |
| **L1d** | 32KB, 8-Way, 2 Cycle, LRU Replacement |
| **L1i** | 32KB, 8-Way, 2 Cycle, LRU Replacement |
| **HOT** | 3.4KB, Direct-Mapped, 2 Cycle |
| **L2** | 256KB, 8-Way, 14 Cycle, LRU Replacement |
| **LLC** | 2MB Slice, 16-Way, 40 Cycle, LRU Replacement |
| **AAC** | 32-Entry, Direct-Mapped, 1 Cycle |
| **DRAM** | 64GB, DDR4 3200, 16 Banks |
| **OS** | Ubuntu 20.04 |

Table 6.2: Simulation Configuration

**Protecting arena headers:** Memento protects arena headers from being corrupted by strayed writes or malicious attacks by blocking writes to the arena header. For every store operation, if the virtual address is within the region delimited by the *MRS* and *MRE* registers, the MMU calculates the offset of the address within the arena. If the store attempts to modify an arena header, which always resides at the beginning of the arena, an exception is raised to the processor to indicate an access violation. Since the size classes and arena header size are known in advance, the calculation can be performed with simple bit operations that also overlap with TLB lookup.

By protecting arena headers against store operations from the software, Memento also defends against a common type of memory bug that overruns an allocated object and corrupts allocation metadata, hence offering more robust memory safety guarantees than software allocators. Unlike software allocators, with Memento, applications can know immediately if a store operation is conducted illegally instead of suffering undefined behavior.

## 6.4 Evaluation

### 6.4.1 Simulation Configuration

**Simulation platform:** We evaluate Memento with full-system simulation using QEMU [28] integrated with the SST [215] framework and DRAMSim3 [152]. The simulated architecture is presented in Table 6.2. We use Ubuntu 20.04 for the OS with the 5.18 kernel. We further show the two main hardware structures of Memento, the hardware object table (HOT) modeled as a 3.4KB direct-mapped cache and the arena allocation cache (AAC) of the hardware page allocator as a 32-entry direct-mapped cache.

We instrumented both user space and kernel functions to capture memory management-related routines. For Python workloads, we instrumented CPython 3.8 [206] heap allocation functions. For C/C++ workloads, we link them against an instrumented version of `jemalloc` from the official repository [72, 115]. For Golang workloads, we instrumented heap memory allocation and garbage collection functions in go-1.13 runtime library and linked it against Golang binaries. To capture page faults, we instrumented the page fault handler function. We also instrumented memory system calls to `mmap` and `munmap`. The simulator replaces calls to memory allocation and free functions with the corresponding ISA extensions of Memento. Allocations of size greater

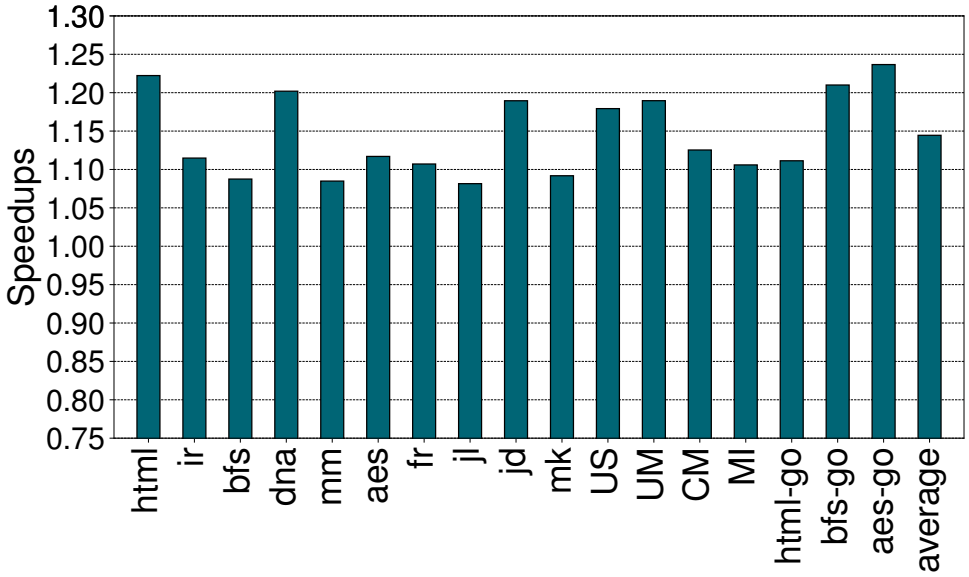



Figure 6.9: **Normalized Speedup** – Numbers are normalized to execution using software library and normal OS kernel

than 512 bytes and deallocations to addresses outside of the reserved virtual address region are handled by software without Memento's intervention.

**Benchmarks:** To evaluate Memento, we use a total of fourteen benchmarks. We focus our evaluation on functions that exhibit at least 0.5 MallocPKI (malloc per kilo instructions) on average. We use function workloads from the serverless cloud function benchmark suite, FunctionBench [132], and the serverless benchmark suite SeBS [54]. `dynamic-html (dh)`, `image-recognition (ir)`, `graph-bfs (bfs)`, `dna-visualisation (dna)` are from SeBS, and `matmul (mm)`, `pyaes (aes)`, `feature_reducer (fr)` are from FunctionBench. We further perform experiments with Python's official benchmarking suite, pyperformance [192] using memory management related benchmarks `json_loads (jl)`, `json_dumps (jd)`, and `mako (mk)`. In addition, we adapted `UrlShorten (US)`, `UserMentions (UM)`, `ComposeMedia (CM)` and `MovieID (MI)` from DeathStarBench [79], written in C++, into function-like units [223]. Lastly, we ported the Python `dynamic-html`, `graph-bfs`, and `pyaes` functions to Golang (`dh-go`, `bfs-go`, and `aes-go`). All workloads use the recommended input whenever possible. Workload execution time ranges from sub-second level to a few seconds, and memory consumption numbers are from low MB to tens of MBs.

Before simulation begins, we perform a system-level only warm-up run and then we simulate functions workloads from the beginning to completion. Each function executes within a `crun` [57] container.

### 6.4.2 Speedup

Fig. 6.9 shows the execution time reduction achieved by Memento as the speedup over the baseline system. Memento achieves substantial speedups between 8–24% and 15% on average. These are substantial gains achieved by Memento for serverless functions.

To further study the source of speedups, we present in Fig. 6.10 a breakdown of the perfor-

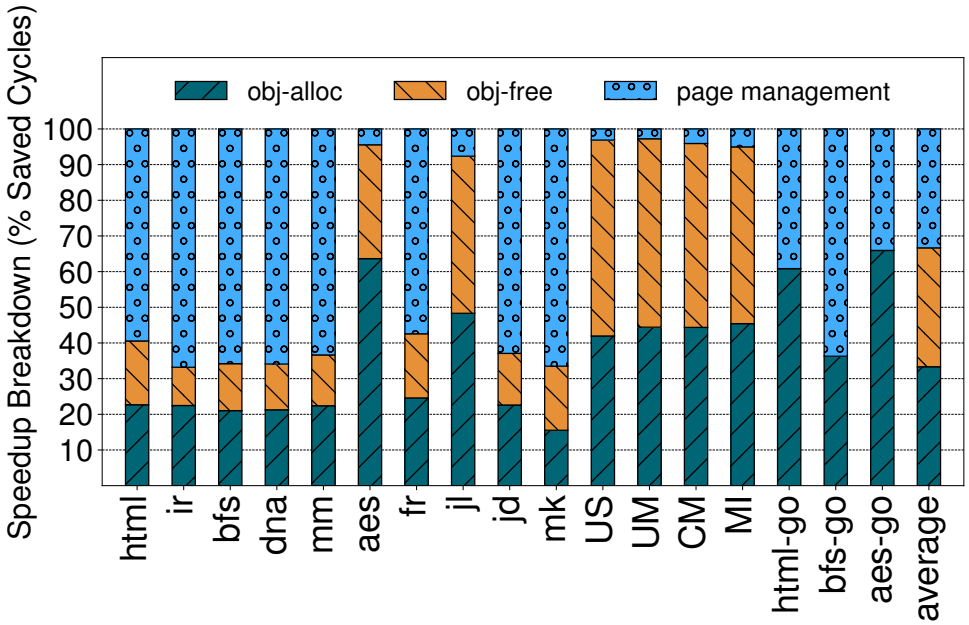



Figure 6.10: **Performance Gains Breakdown**

mance gains of Memento. The figure shows three main sources of gains that can be achieved by Memento: (i) hardware object allocation (*obj-alloc*), (ii) hardware object free (*obj-free*), and (iii) hardware page management operations (*page-management*). Memento can satisfy serverless memory management operations with a latency equivalent to single cache access in most cases, and hence it significantly reduces the execution time of functions. On average, 32% of the gains are attributed to hardware object allocation and 34% to free. The hardware page management component is responsible for 34%.

Individual workloads demonstrate different gains from either or both sources, further highlighting the need for both hardware object and page management to be performed by Memento. Eight of the ten Python workloads and all Golang workloads get at least 40% of the gains from Memento's hardware page management. This observation is explained by the larger heap size, which causes a higher number of page faults that increases proportionally to the number of pages in the heap memory. The majority of the page management gains come from dynamic page allocations in hardware that eliminate costly OS page fault handling. In the case of `aes` and `jl`, more than 90% of the gains come from hardware object management. This result is because their working sets are much smaller, shifting the critical path from page allocation to object allocation.

The DeathStarBench workloads show significant speedups of 14% on average. The majority of the gains come from hardware object management. The page management gains are smaller because jemalloc pre-maps and pre-faults a small pool of memory during library initialization. This mechanism instead turns object allocation and free operations into a performance bottleneck that Memento addresses.

Overall, the results highlight the importance of both of Memento's mechanisms, hardware object allocation and hardware page management.

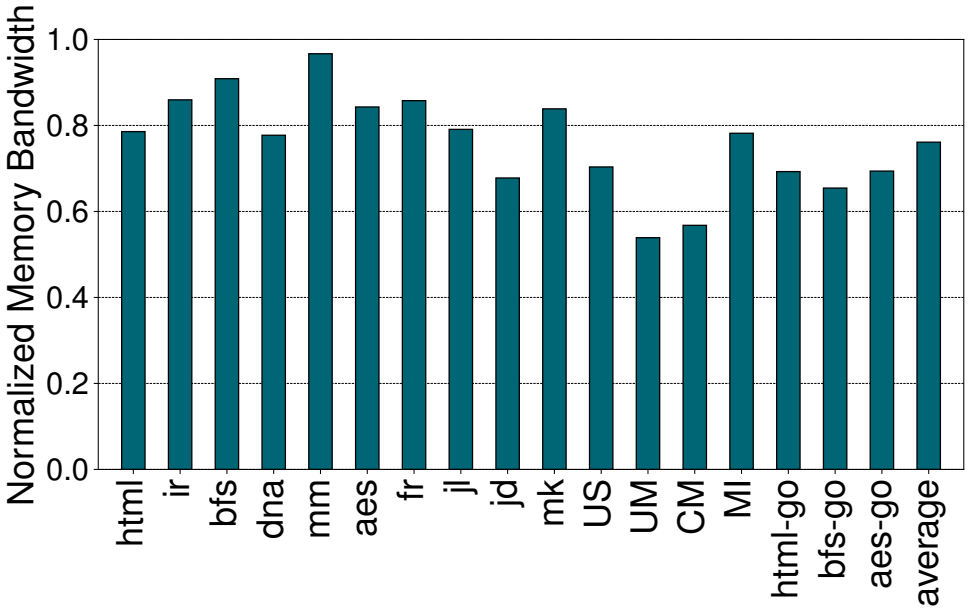



Figure 6.11: **Normalized Memory Bandwidth Usage** – Numbers are normalized to execution using software library and normal OS kernel

### 6.4.3 Memory Bandwidth Savings

To study the effect of Memento on main memory traffic, we present results on normalized memory bandwidth savings in Fig. 6.11. Overall, Memento reduces memory bandwidth usage by 22%. For two workloads, `UM` and `CM`, Memento achieves a 40% reduction of DRAM traffic. Memento's bandwidth savings are attributed to two sources. First, the hardware object table (HOT) absorbs allocation traffic to memory as objects can be instantiated in the cache without adding unnecessary traffic to the main memory. Furthermore, Memento eliminates the instruction and data movement caused by memory management in user space and the kernel.

### 6.4.4 Aggregate Main Memory Usage

Fig. 6.12 presents normalized aggregated memory usage results. We measure aggregated memory usage as the total number of physical pages allocated during simulated execution. We present normalized user and kernel space memory separately in addition to the total memory usage. Memento achieves an overall 15% reduction in aggregate memory usage. When we look at the user space and kernel usage, we see that user space reduction is about 10% and kernel is 28%.

However, compared to the baseline, Memento increases user space memory usage for Python and Golang workloads. A primary reason is that the software allocators for these two languages share free pages among size classes, which reduces external fragmentation. We choose not to include this potential optimization and trade-off user space memory instead for a less complicated hardware design. Nonetheless, Memento reduces kernel memory usage for these workloads by 29%. In workloads like `dh`, Memento can achieve major kernel memory usage reductions, more than 60%, due to the reduction of kernel metadata needed to manage memory regions.

For C++ workloads from DeathStarBench [79], Memento achieves 41% user space memory

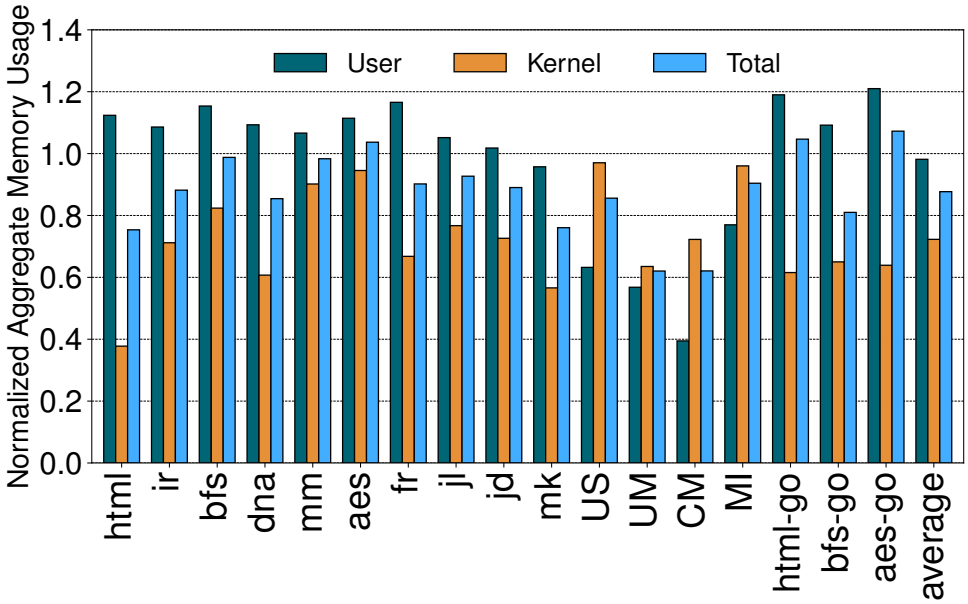



Figure 6.12: **Normalized Aggregate Memory Usage** – Numbers are normalized to execution using software library and normal OS kernel

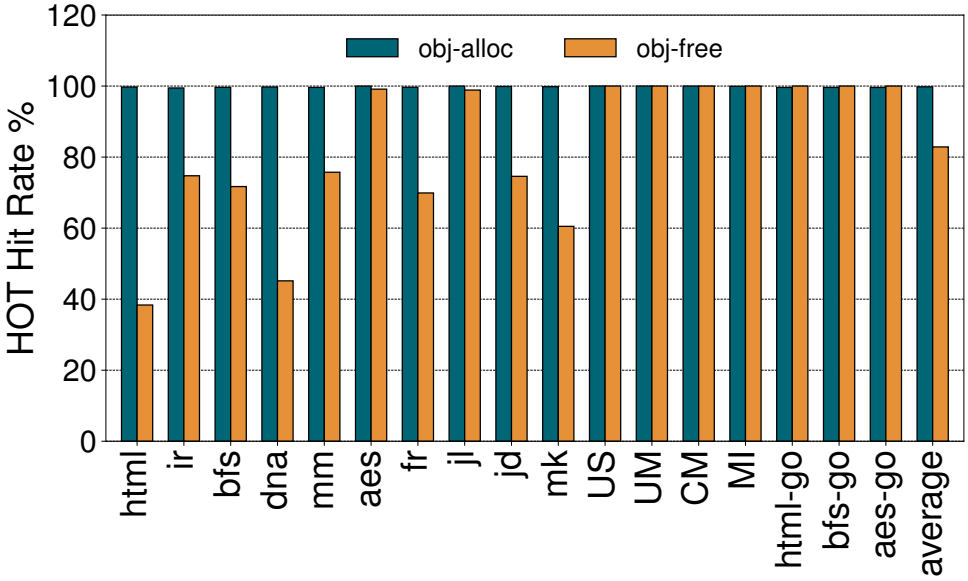



Figure 6.13: **Hardware Object Table Hit Rate**

savings. We attribute this to the low utilization of the memory pool of jemalloc, which ends up wasting user space memory resources. Memento instead can dynamically respond to the memory usage of each function. Finally, Memento achieves significant kernel savings for the C++ workloads from DeathStarBench, with an average of 25%.

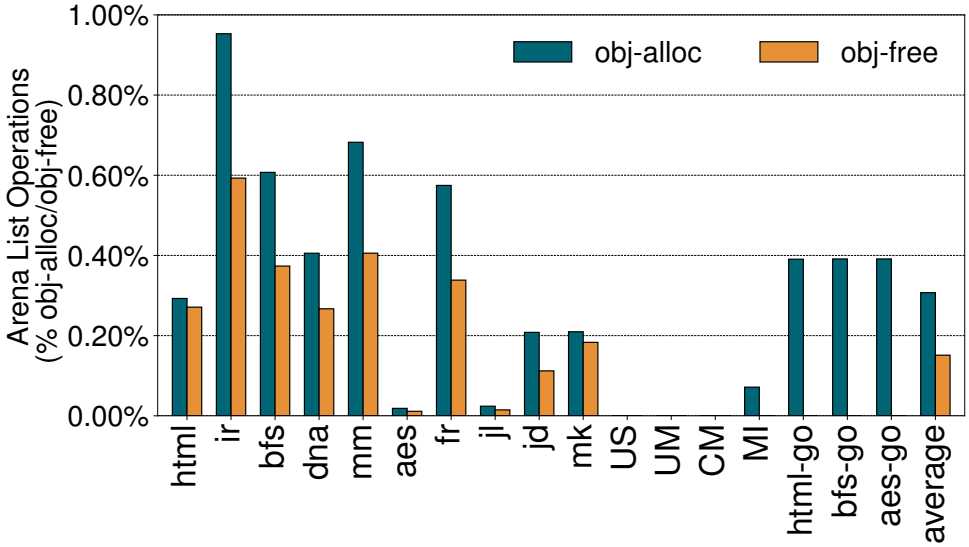



Figure 6.14: **Arena List Operation Frequency** – Measured as the percentage of `obj-alloc` and `obj-free` that include arena list operations

### 6.4.5 Characterizing Memento

**Hardware object table hit rate:** We present the hit rate of the hardware object table (HOT) in Fig. 6.13. On allocation, a HOT hit happens when a request can find an available entry in the cached header bitmap. Similarly, on free operations, a HOT hit happens when the cached header can fulfill the free request without incurring additional memory operations. A miss in the HOT causes a cache request to be issued to load the appropriate entry from memory. Hits in the HOT are completed in two cycles without issuing memory requests.

Overall, allocations in Memento enjoy a high hit rate of 99.8%. Furthermore, they exhibit a uniform behavior across workloads. Free operations show an average hit rate of 76%. We observe that Python workloads generally exhibit lower free hit rate, while Golang and C++ workloads show a very high free hit rate. Further investigation reveals that C++ workloads operate on tight loops with objects being allocated at the beginning of the loop and freed at the end. Golang workloads, on the other hand, rely on garbage collection to batch-free objects. Therefore, they never need to call free on individual objects. For Python workloads, some of the allocations are made by the interpreter to store global information that tends to live much longer than other local objects (e.g., those created in tight loops). These long-lived objects cause low HOT hit rates. Nevertheless, Memento handles them correctly by performing the free operation out of the execution critical path. As a result, Memento is still able to eliminate their overheads successfully, despite low HOT hit rates.

The results further corroborate the insights from Section 6.1.2 as function workloads are short-lived in nature with short malloc-free distances. When an object is allocated from the HOT and freed shortly afterward, the free request will very likely result in a hit. Consequently, Memento achieves a high HOT hit rate even with a small direct-mapped structure.

While not shown in the figure, Memento's arena allocation cache (AAC) enjoys uniformly high hit rates as only a few size classes are utilized in each of the workloads.

**Arena list operations:** Fig. 6.14 presents the frequency of linked list operations for arena

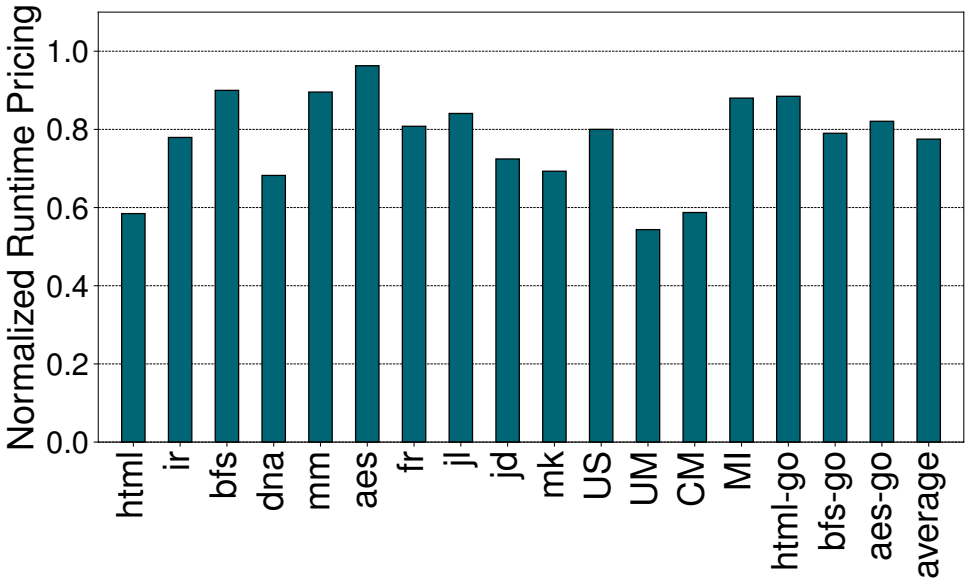



Figure 6.15: **Normalized Function Runtime Pricing** – Numbers are normalized to execution using software library and normal OS kernel

management during allocation and free. Arena linked list operations occur when the object allocator puts arenas on/off the full/available arena lists as described in Section 6.2.1. We characterize such operations by calculating the percentage of allocation or free operations that include arena list operations.

For all workloads, less than 1% of allocations and less than 0.6% of frees need to operate on the linked lists. Workloads with smaller working sets require fewer linked list operations because there are fewer arenas to manage, and the locality of allocation and free is high. Compared with the total number of instructions, the linked list operations only constitute a tiny fraction of total memory accesses ($\leq$ 0.01%). We also evaluated an ideal environment without the linked list operations and observed minimal changes in the result. Overall, the performance implications of arena list operations are negligible.

### 6.4.6 Function Pricing

We compute the cost of running functions on both the baseline system and Memento according to AWS pricing [15]. Fig. 6.15 shows the runtime cost based on execution time and memory usage. The current AWS pricing policy computes the cost of functions in the granularity of milliseconds for runtime and MB for consumed memory. On average, we can see from the results that Memento can achieve a significant runtime cost saving of 24%.

Function providers add a fixed per-invocation cost that accounts for the infrastructure management and deployment costs outside the function costs. While this cost is outside the scope of Memento, when included for end-to-end pricing cost, Memento is able to achieve cost savings up to 31%, and 11% on average.

### 6.4.7 Sensitivity Studies

**Populating pages on mmap:** `mmap` provides a flag, `MAP_POPULATE` that will force the OS to eagerly populate virtual pages with physical memory, hence reducing the page fault overhead. However, eagerly populating pages may cause an increased physical memory footprint. To evaluate the effect, we instrumented the software allocators under comparison to pass the flag to `mmap` and evaluated Memento on this setting. For Golang workloads, we observed that on average function execution increased by 3% but the physical memory footprint increased by 8.6×, due to the large `mmap` size of Golang allocator. For Python and C++ workloads, we did not observe any significant change in speedup, with physical memory increasing by an average of 9.6%. We conclude that eagerly populating allocations is not cost-efficient for serverless functions based on the AWS pricing model in Section 6.4.6 due to the increased physical memory footprint.

**Multi-process environments:** To evaluate Memento in multi-process environments, where a single core is over-subscribed to several time-sharing function instances, we ran the simulation by starting four randomly selected function instances and pinning them to the same core. We repeated the experiment ten times, with different workloads each time. The main potential overhead of Memento is induced due to the flushing of the HOT table. We find that the overall effect on the speedup is negligible. We attribute the results to the fact that context switches are relatively infrequent.

**Tuning software allocators:** We manually tuned the available configuration knobs of software allocators to study their effects on Memento. Our results show that while changing most of the knobs does not affect Memento's speedup, enlarging the arena size of software allocators causes a noticeable but less than 1% speedup. Further investigation revealed that using larger arenas reduces the frequency of `mmap` at the potential cost of fragmentation. Physical memory footprint is unaffected as `mmap` reserves physical pages lazily.

**Warm-start versus cold-start:** Our earlier experiments warm-started functions, i.e., functions are executed by an existing container process and skip the container set-up stage. However, in practice, functions may also experience cold starts where the latency of setting up containers is added to their execution latency. To study the effect of cold-started functions on Memento, we ran experiments where containers are set up before executing the function. Our results show that cold-start slightly reduces the benefits of Memento between 0.1-1.4% depending on the function runtime.

| Name | Read/Write | Static | Area |
|---|---|---|---|
| Memento HOT | 0.005/0.006nJ | 1.32mW | 0.0084mm$^2$ |
| Memento AAC | 0.002/0.002nJ | 0.43mW | 0.0023mm$^2$ |

Table 6.3: Hardware Cost of Memento

### 6.4.8 Hardware Cost

We evaluate hardware energy and area cost of Memento using CACTI 6.5 [183] under 22nm technology node. We model the Memento's hardware object table (HOT) as a 64-entry direct-mapped cache structure with the entry format as described in Section 6.2.1. We model the extra

storage on the page allocator for arena pointers and the shootdown bit vector as 32-entry direct-mapped cache. Each entry stores four most recently used arena header pointers and a bit vector for storing shootdown information. Results are presented in Table 6.3. Overall, the hardware cost of Memento is minimal while providing substantial performance improvements.

### 6.4.9 Comparison with Related Work

The closest related work is Mallacc [126] which integrates with TCMalloc and offers hardware support for a subset of the operations of malloc targeting only C++ workloads. We further discuss Mallacc in Section 6.5. We simulate an ideal version of Mallacc, in which the Mallacc cache has zero latency and always hits. Since Mallacc only supports C++ workloads, we only compare it to Memento on DeathStarBench. On average, this ideal Mallacc configuration achieves a speedup between 5-10% and an average of 8%. Instead, Memento achieves much higher speedups of 10-17%, and an average of 14%. Overall, Memento significantly outperforms Mallacc. The reason is twofold. First, Mallacc relies on software for many allocation operations. Second, Mallacc does not consider the OS allocation path. This feature is crucial for high-level languages such as Python, as we showed in Section 6.4.2, that Mallacc does not support. Instead, Memento provides a holistic hardware-centric solution that alleviates the overheads of serverless memory management in both user space and the OS while being able to integrate with existing software stacks easily.

## 6.5 Additional Related Work on Reducing Cold-Start Latency

Cold-start latency reflects the overhead of bringing up serverless functions to a state that is ready for execution. The overhead may include the time required to (i) start the virtualization platform, (2) set up the execution environment, and (3) resolve software or library dependencies [191]. Cold-start latency is particularly difficult to amortize over the rest of the execution for serverless functions due to their extremely short lifetime.

Prior works have attempted to minimize cold-start latency by individually or collectively reducing the latency of performing each of the tasks during function start-up. For example, unikernels can reduce the amount of unnecessary work during OS bootloading, allowing function instances to fast boot into the OS [10, 36]. In addition, by eliminating non-essential services from the virtualization platform, prior works present the design of "lean containers" [191] or "micro-VMs" [10] that minimize the overhead incurred by virtualization. Process snapshots or "zygote processes" shrink process creation latency by enabling function instances to be started from an already initialized snapshot [36, 68, 191, 247], rather than performing expensive process initialization. Keep-Alive Caching prevents the destruction of function instances after they have completed execution, in the hope that future invocations can be handled using the warm instance, hence skipping all unnecessary work for initialization [78, 229]. Prefetching can help eliminate expensive page faults [247] or instruction cache misses [223] on all stages of execution, therefore also reducing cold-start latency. Lastly, prior works also resort to compiler-enforced isolation to reduce the cost of enforcing isolation between function instances in the runtime [35, 232].

As a result, these instances can be spawned without strict runtime isolation guarantees and the associated overhead while remaining functionally isolated from each other.

# Chapter 7

# Conclusions and Future Work

## 7.1 Conclusions

This dissertation addresses two major limitations with the existing cache hierarchy design that prevent software and hardware from utilizing it efficiently. The first limitation is the lack of multiversioning support, which leads to suboptimal solutions for transaction memory and NVM-based memory snapshotting. The second limitation is not being able to leverage application-level information, which prevents the cache hierarchy from (i) achieving a higher compression ratio on LLC compression, and (ii) optimizing object allocation from a high level that leverages the allocation semantics. This dissertation addresses both limitations by taking advantage of multiple instances of related objects.

In the first half of the dissertation, we presented **OverlayTM** (Chapter 3), a Hardware Transactional Memory (HTM) design that enables threads to synchronize via hardware executed transactions, and **NVOverlay** (Chapter 4), an NVM-based memory snapshotting design that periodically checkpoints consistent memory states and saves them to the NVM device. Both designs share a base mechanism implementing hardware multiversioning, i.e., Page Overlays (Section 2.1), and utilize its "Overlay-on-Write" semantics to create cache blocks on the same address but with different OIDs. The creation of overlay blocks is performed directly in the cache hierarchy, without allocating backing storage, and hence incurs minimum overhead. The newly created overlay cache blocks, also known as "versions", are managed by hardware. In this dissertation, we proposed several mechanisms for managing these versions to satisfy different design requirements. We hence conclude that the main contribution of the first half of the dissertation is the demonstration of practical hardware-supported multiversioning and applying it to real-world problems.

In OverlayTM, overlay cache blocks created by overlay writes represent the private write working set of uncommitted transactions. Multiple transactions can write the same address concurrently without interfering with each other, each creating its overlay block in the private hierarchy. To manage these overlay blocks belonging to different transactions, we propose Multiversioned Page Overlay (MPO), which consists of a Version Directory that tracks committed overlay blocks and their owners, and a Version Access Protocol that directs transactional reads to the correct version. The versioning support on OverlayTM implements what is called the snapshot

read semantics, which guarantees the consistency of data accessed by transactional read operations. To detect conflicts between concurrent transactions, we also proposed a conflict detection protocol based on commit-time validation, and a backward protocol that relies on a hardware Commit Queue. By combining the version management and conflict detection capabilities, OverlayTM can execute transactions with a greater degree of parallelism than prior works and strong serialization guarantees. Read-only transactions and early-release reads also benefit from the multiversioning support and can execute with fewer conflicts. Experiments using STAMP and data structure workloads on a 16-core simulated platform show that OverlayTM greatly reduces transaction abort rates and total execution cycles compared with an eager conflict detection design (both show 30%–90% reduction on OverlayTM). When compared with TCC and serializable SI-TM, OverlayTM also demonstrate its advantage on specific workloads due to read-only and early release support and its strong isolation guarantee.

NVOverlay treats overlay cache blocks as incrementally captured memory snapshot data and manages these blocks in the L2 level of the private hierarchy. The overlay blocks will be gradually written back to the NVM for persistence either alongside regular cache coherence or by the cache tag walker. In addition to leveraging overlay blocks as the in-cache representation of snapshot data, NVOverlay also extends the main memory components of Page Overlays, i.e., the OMS and the OMT, to organize persistent snapshot data efficiently using a series of mapping tables managed similarly to Log-Structured Merge (LSM) trees. In order to maintain the consistency of memory snapshots without executing global barriers, NVOverlay adopts a more relaxed consistency model and guarantees data consistency with a distributed Lamport clock tracking data dependencies. We evaluated NVOverlay on STAMP and data structure workloads. Results show that NVOverlay can hide the latency overhead of persistence in most cases (and only incur a relatively marginal overhead for the rest), and efficiently reduces write amplification by more than 80% compared with hardware logging. When evaluating against PiCL, a hardware undo logging design, NVOverlay also demonstrates lower peak memory bandwidth, and generally less bandwidth consumption when capturing snapshots in small but localized bursts.

In the second half of this dissertation, we demonstrated how leveraging application-level information in the cache hierarchy can benefit performance. Specifically, in Chapter 5, we presented **Multi-Block Cache Compression (MBC)** as an exhibition of leveraging software-provided per-page data layout to assist inter-block cache compression. In Chapter 6, we further looked into one of the most common and mundane tasks of an application—memory management, and proposed **Memento** as a holistic and hardware-centric approach for optimizing heap memory allocation. We conclude that the main contribution of the second half of the dissertation is the demonstration of practical hardware-software collaboration on leveraging application-level information.

MBC is designed based on the insight that analogous blocks, i.e., blocks that contain similar content and can be compressed together, often form a stepped (spatially strided) pattern on one or more pages. To leverage this data pattern, MBC enables the application to pass the per-page "step size" attribute using existing memory management system calls, which propagates the attribute from the page table to the TLB, and finally to the LLC (alongside memory requests). Inter-block redundancy can thus be utilized by grouping analogous blocks on the same page into stepped super-block cache tags, based on the "step size" attribute of the page, and then using the first block of every super-block tag as the reference of compression. In order to accomplish high

compression ratio, we also proposed a dictionary-based algorithm derived from C-PACK that is capable of leveraging both intra- and inter-block redundancy, and a 1.5× over-provisioned tag array organization which achieves a maximum effective cache capacity of 6×. We evaluated MBC on data application workloads (list, B+Tree, TPC-H, and real-world analytical dataset), with application-assigned "step size" attributes, and SPEC, with the default "step size" of one. Results show that MBC outperforms an uncompressed cache by 6.99% and 12.5% higher IPC, and 2.55× effective cache capacity, on data application workloads and SPEC, respectively. Furthermore, when compared to a prior state-of-the-art inter-block compression design, MBC can achieve up to 6.07% higher IPC and 1.25× effective cache capacity on data workloads, and 8.37% higher IPC and 1.61× effective cache capacity on SPEC. We attribute the performance speedup and high compression ratio of MBC to an optimal combination of the compression algorithm, tag array organization, and effectively leveraging application-level information.

Memento optimizes memory management by offloading most of the memory management work to dedicated hardware components, hence reducing the amount of work on the execution critical path. The design of Memento is based on the insight that user space memory management functions (`malloc` and `free`) are not semantically dependent on execution, and therefore, can be executed in the background without changing their semantics. Besides, Memento also leverages the OS-level observation that only a small subset of OS kernel memory management functions are involved in order to manage physical pages properly. Following this observation, Memento only implements the small subset of kernel functions as a separate hardware page management module, thus enabling even greater speedup by performing page management directly on hardware. We recognized that Memento is particularly beneficial to Serverless Computing due to the short lifetime and the object-oriented nature of serverless functions (the basic execution unit of Serverless Computing applications). To evaluate the benefits, we conducted experiments with Memento on real-world serverless functions implemented in Python and C++. Our evaluation shows that Memento reduces main memory traffic by 22%, speeds up function execution by 8–22% (14% on average), and, as a result, reduces the end-to-end cost of running functions on Serverless Computing platforms by 24%. When compared with a prior proposal that does *not* leverage high-level application information (but instead executes low-level memory allocation operations on hardware), Memento exhibits 1.75× higher speedup. We regard these results as strong proof that Memento benefits greatly from leveraging application-level information of memory management functions in both user space and the kernel.

## 7.2 Future Works on Addressing Interactions Between Proposed Designs

This dissertation has primarily been an exploration of different opportunities presented by taking advantage of object relationships within the cache hierarchy, and we have presented several concrete proposals for exploiting those relationships: OverlayTM, NVOverlay, MBC, and Memento. In this section, we collect some of the considerations one might encounter when simultaneously supporting multiple of these proposals in a single system. We leave a more detailed analysis for future work.

**Contending for the Overlay ID (OID) tags:** Both OverlayTM and NVOverlay use the per-block OID tag to represent version numbers. The per-block OID tag may hence become the source of structural hazard between the two designs when a cache block is modified. To resolve this structural hazard, we propose to divide the existing 16-bit OID field into two smaller sub-fields, the "OverlayTM-OID" and the "NVOverlay-OID", such that each of them can be accessed and updated by individual circuits. In this proposed design, OverlayTM transactions only update the "OverlayTM-OID" field, while NVOverlay only updates the "NVOverlay-OID" field. The two designs can then operate without affecting each other when both of them are enabled. However, when only one design is enabled, it can still monopolize the whole 16 bits of the OID tag.

Alternatively, hardware designers may also choose to add an extra OID tag bank such that each design operates on its designated OID bank. This design option trades off the performance impact of having narrower OID tags (i.e., more frequent OID wraparound) for more significant hardware costs. For example, if we add one extra OID bank of 16 bits per block, the total storage overhead of the OID bits will increase to 6.25% of the total data bank size.

When both OverlayTM and NVOverlay are enabled in the system, uncommitted transactions may be persisted as part of the durable memory snapshot, incurring extra complications during crash recovery as these uncommitted transactions need to be rolled back. We postpone the discussion of a potential solution that enables *durable transactions* to Section 7.3.

**Maintaining persistent allocation metadata on NVM:** Memento is designed to work with volatile main memory where a system crash will wipe out the content of the main memory, including the allocation metadata maintained by Memento. However, with NVOverlay enabled, the system operates under full-address space snapshotting mode where the working memory is periodically checkpointed into non-volatile storage automatically in the background. In this scenario, the allocation metadata of Memento may not be checkpointed correctly as it is maintained in two separate caches called the Hardware Object Table (HOT) and the Arena Allocation Cache (AAC), respectively. As a result, the system snapshot captured by NVOverlay's protocol is incomplete, as the snapshot does not include the allocation metadata of Memento.

To address this problem, on systems that implement both NVOverlay and Memento, we propose to extend every HOT and AAC entry with an OID tag just like regular cache blocks. When an allocation or deallocation request is made, the current epoch of the requesting processor is included in the request message. On receiving the message, Memento compares the epoch number with the OID tag of the HOT entry. If they differ, then Memento updates the OID tag of the HOT entry to the one included in the message and sends the content of the entry to the MNM backend. The MNM backend will then update the current snapshot using the physical address of the arena (which is included in the HOT entry). AAC entries are handled similarly except that the epoch number is included in the arena allocation message sent by the HOT.

**Guaranteeing the atomicity of memory allocation:** Memento also interacts with OverlayTM as the contents of HOT and AAC are not updated atomically, causing the allocation metadata to be corrupted when a transaction aborts and when multiple transactions attempt to deallocate the same memory object. To ensure atomic updates of both caches, we propose to extend HOT and AAC entries with an extra OID tag as in the previous solution (the resource hazard on the OID tags between OverlayTM and NVOverlay can be dealt with similarly as presented in earlier sections). When a transaction attempts to allocate or deallocate memory, it

includes its *bts* in the request message. On receiving the message, if the *bts* and the entry's OID differ, the HOT performs in-cache CoW just like in the data cache and writes the old entry back to the cache hierarchy (tagged with the old OID). Deallocation of objects maintained by a remote HOT occurs normally using the protocol described in Section 6.3.

**Obtaining "step size" information via the hardware allocator:** The last possible interaction that we investigate is between Memento and MBC. Memento performs memory allocation on hardware without involving any software allocator, which contradicts the design philosophy of MBC that software should provide hints to aid compression. However, the software support for MBC is merely a simple extension of the existing arena-based object allocation technique (see Section 5.4.2). This software technique can be easily ported to Memento which we briefly describe as follows. First, memory allocation requests to Memento should allow an optional operand, i.e., the type of the object. Moreover, we also extend the HOT entries and arena header format to store the type of the arena. On receiving the allocation request, Memento performs HOT lookup by checking whether the requested type matches the arena type cached in the HOT. If they do not match, the Memento hardware searches for an existing arena of the same class and type. If no such arena can be found, a new arena is requested from the page allocator with the type information and the "step size" attribute calculated by hardware using the requested size. Finally, when the hardware page allocator processes the request, it sets up the "step size" field of the PTE for the first page of the arena. The "step size" attribute will then be fetched by the MMU into the TLB and propagated to the LLC, following the usual path as in the current MBC design.

**Compressing Speculative Data:** MBC can compress speculative cache blocks generated by OverlayTM if analogous blocks with the same OID tags (i.e., they are generated by the same transaction) are mapped to the same stepped super-block tag. In this case, the LLC index generation circuit takes *both* the OID tag and the "step size" attribute when calculating the set index. When a transaction aborts, all of its speculative blocks are invalidated, which can then be implemented by invalidating the super-block tag.

By compressing speculative blocks generated by uncommitted transactions, OverlayTM enjoys the extra benefit of fewer LLC write backs of speculative data, which results in faster transactional execution in general.

## 7.3 Future Works on Hardware-Supported Multiversioning

**Supporting thread-level speculation:** The transactional execution capability of OverlayTM and its timestamp-ordered transaction model can be extended to support thread-level speculation (TLS) in future works. In TLS, loops or similar software constructs are unrolled into iterations and then dispatched to multiple threads for parallel execution. TLS assumes that iterations are mostly independent of each other but does not exclude the possibility of occasional data dependency, hence necessitating transactional execution of iterations. Compared with transactional memory, where transactions are only serialized in *some* order, TLS requires a pre-determined ordering (as defined by program semantics) between loop iterations to be enforced. The design challenge of supporting TLS hence concentrates on enforcing this external serialization order, in addition to providing atomicity and serializability.

Prior works on TLS [75, 237, 238] encode the external serialization order in the coherence

states, resulting in a very complicated design where the coherence state machine is tightly coupled with TLS support. With OverlayTM, since transactions are already timestamp ordered, the external serialization order can just be represented using transaction's *Begin Timestamp (bts)*, and enforced by the existing conflict detection protocol. Only moderate hardware modifications are needed to transform OverlayTM to support TLS. First, the design must expose a software interface that enables the application to start speculation and specify the ordering, which OverlayTM lacks. Second, conflict detection should be modified to enforce serialization ordering based on *bts* rather than *cts*. One potential way is to track the read set of each speculation and then validate the read set by re-reading every address in it to confirm that the external serialization order is observed. More efficient solutions are also possible, which we leave as future work.

**Memory snapshotting for intermittent computing:** Future works on NVOverlay may extend it to support memory snapshotting for intermittent computing. Intermittent computing refers to a new computing paradigm where the device no longer has a stable power supply and hence must endure constant power interrupts as a norm rather than as an exception [163]. This scenario is mainly associated with Internet-of-Things (IoT) devices that operate with limited battery capacity, charged by unstable power sources such as solar panels or other energy harvesting methods. Because of this unique property, one of the top priorities for intermittent computing devices is to preserve the progress of computation before imminent power interrupts, such that execution could resume from a recent checkpoint instead of restarting from the initial state.

One approach for preserving computation progress is to take memory snapshots and restore them after the power supply re-stabilizes. The challenge for this particular use case is that intermittent computing has a slightly different failure model than the one we assumed for NVOverlay in Section 4.1.1. More specifically, intermittent computing devices can be configured to receive a notification in advance before the battery completely drains, which allows the device to sustain a certain number of NVM operations before it entirely shuts down. The design focus of the memory snapshotting mechanism, therefore, is shifted from ensuring the consistency of persistent data due to unexpected failures, to designing a protocol that schedules ordered shutdowns before imminent power interrupts, in which all data that has not yet been persisted will be written back. In this situation, NVOverlay can be simplified by not having to guarantee the atomicity of updating the master mapping table ($M_{master}$). Prior works [33] that adopt a similar approach to NVOverlay also leverage this unique failure model to simplify the design.

Another challenge of intermittent computing is to preserve power as much as possible. A viable design must, therefore, minimize the extra power overhead incurred by memory snapshotting. NVOverlay in its current form is a poor fit due to the 16-bit OID tag in the hierarchy and the extra DRAM operations for maintaining the tags in the main memory (Section 4.2.5). In future works extending NVOverlay to intermittent computing devices, the width of the OID tags in both the cache hierarchy and the main memory should be shrunk, which also reduces the number of concurrent snapshots it supports (in the most extreme case, only 1-bit OID per block is left, which supports two concurrent snapshots). Fortunately, NVOverlay does not impose any restriction on the width of the OID tags and is hence not intrinsically power-hungry.

**Enabling durable transactions:** In future systems that employ NVM as main memory, it might be feasible to combine OverlayTM and NVOverlay to create a Durable Hardware Transactional Memory (DHTM) design that executes durable transactions. In the DHTM design, OverlayTM provides a transactional model which guarantees the atomicity and serializability

of transactional execution in memory consistency order (i.e., the order of memory operations perceived by processors via data dependency), while NVOverlay writes back dirty data generated by committed transactions to match the order of persistence with the serialization order of transactions. In the proposed DHTM design, the per-block OID tags serve dual purposes. On the one hand, they indicate the serialization order of transactions in the logical time scale (as described in Section 3.1.2). On the other hand, they also encode data dependencies between dirty blocks, which dictate the proper ordering these blocks should be persisted to the NVM. The main challenge of materializing the DHTM design is to handle NVM-residing persistent data in a more read-efficient manner, because NVOverlay assumes that NVM data will be write-mostly, and hence optimizes it for writing rather than reading.

## 7.4 Future Works on Leveraging Runtime Application-Level Information

**Automatic discovery of per-page stepped pattern:** While MBC is able to leverage per-page stepped pattern for inter-block compression, in Chapter 5, we assumed that the "step size" attribute is explicitly provided by the application whenever the attribute is known to users or software libraries. In future works, this mechanism can be automated either statically by compilers or dynamically by profilers. In the former setting, the compiler may perform simple static analyses on performance-critical code that allocates large chunks of memory. For example, when an array-of-structs is to be allocated, the compiler can compute the "step size" attribute statically (since the struct size is known at compilation time) and insert function calls to pass the attribute as a compression hint to the hardware. In the dynamic setting, a software profiler is scheduled in the background, whose task is to analyze the memory layout of frequently accessed pages. Whenever the profiler identifies a larger-than-one stepped pattern on one or more pages, it will notify the OS using the system call interface proposed in Section 5.4.1. Furthermore, dynamic profiling can be conducted either online or offline. Previous works such as Zhang et al. [273] can be used as a reference for offline profiling.

**Main memory inter-block compression:** MBC's inter-block cache compression scheme can be extended to main memory compression, which reduces the storage and bandwidth overhead within the main memory. The main memory compression scheme also performs inter-block compression across analogous blocks. However, instead of using stepped super-block tags for grouping analogous blocks, the scheme may compress analogous blocks on the same DRAM row (the size of which is typically a few KBs) using the same algorithm as in MBC. The compressed blocks are then stored compactly on the DRAM row. Memory accesses to compressed rows need to be redirected to the correct "segment" which is the subunit that compressed analogous blocks are stored within the row. As with similar works on main memory compression, the extra mapping information must be maintained either in-line within the physical row storage or in a separate memory region for storing mapping metadata.

Furthermore, if both the LLC and the main memory are compressed, then on LLC misses, compressed blocks can be fetched from the main memory and transferred on the memory bus without decompression. The synergic cache and main memory compression design can harvest

even more memory bandwidth benefits compared with an uncompressed design because fewer bytes are transmitted on the memory bus for handling LLC misses.

**Allocation-driven cache prefetching:** Future works on Memento may focus on reducing the access latency of hardware-allocated objects by prefetching cache blocks containing the object into the private hierarchy before accesses to the object are made. One reasonable design option is to let allocation requests drive prefetching decisions, which would at least work well for the workloads we have evaluated in Chapter 6. In these workloads, an object will be initialized shortly after allocation returns the object's address. In addition, we also observe that the first memory operation to a newly allocated object is almost always a write because, according to the semantics of the user space memory allocator (i.e., `malloc`), newly allocated objects contain undefined data. Memory reads, therefore, are extremely unlikely on newly allocated objects before the objects are properly initialized.

We sketch the design of a hardware object prefetcher that collaborates with Memento as follows. When an allocation takes place, Memento sends a prefetching request containing the object's address and size to the prefetcher. On receiving the request, the prefetcher decides whether or not to perform prefetching based on (i) whether the object has already been fetched into the private hierarchy recently, (ii) whether the hierarchy can sustain additional memory requests without incurring contention, and (iii) the history of object usage pattern of the requested size class (which can be collected from a per-size-class profiler). If the prefetcher decides to execute prefetching, then it issues a coherence message to the LLC to acquire exclusive ownership of the object's address (or addresses if the object spans multiple blocks). Future access patterns on the prefetched object are also recorded by a hardware profiler, which will then be used to provide feedback for fine-tuning the prefetching policy.

Future works may also explore page-level prefetching, which is driven by Memento's physical page allocator, and is performed when the initial tag walk request on a newly allocated page is received. In page-level prefetching, it is even possible to directly create cache blocks in the hierarchy, in a way that resembles "Overlay-on-Write" without actually fetching data from the main memory. In such a design, the hardware prefetcher will instruct the LLC controller to insert dirty cache blocks with address tags on the newly allocated page, as if these blocks were fetched from the main memory. Zeroing of the allocated page (which many OS kernels will do) can also be piggybacked on block insertion by simply zeroing the content of the blocks after they are inserted.

# Bibliography


[1] Dgben download page: https://www.microsoft.com/en-us/download/details.aspx?id=52430. 5.6.1

[2] Tpc-h, home page: http://www.tpc.org/tpch. 5.6.1

[3] Python memory management and pymalloc, home page: https://docs.python.org/3/c-api/memory.html. 2.3.1, 5, 5.2.2, 5.4.2, 6.1.1, 6.3

[4] Spec cpu2006, home page: https://www.spec.org/cpu2006, . 5.6.2

[5] Spec cpu2017, home page: https://www.spec.org/cpu2017, . 5.6.2

[6] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682, 2006. 5.1.1

[7] Ahmed Abulila, Izzat El Hajj, Myoungsoo Jung, and Nam Sung Kim. Asap: architecture support for asynchronous persistence. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 306–319, 2022. 2.2.2

[8] Yehuda Afek, Amir Levy, and Adam Morrison. Programming with hardware lock elision. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 295–296, 2013. 3.1.1, 3.7.2

[9] Yehuda Afek, Amir Levy, and Adam Morrison. Software-improved hardware lock elision. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 212–221, 2014. 3.1.1, 3.7.2

[10] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020. 6, 6.5

[11] Rishi Agarwal, Pranav Garg, and Josep Torrellas. Rebound: scalable checkpointing for coherent shared memory. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 153–164, 2011. 2.2.2

[12] Alaa Alameldeen and David Wood. Frequent pattern compression: A significance-based compression scheme for l2 caches. Technical Report, 2004. URL `http://digital.library.wisc.edu/1793/60388`. 2.3.1, 5.7.1

[13] Alaa R. Alameldeen and David A. Wood. Adaptive cache compression for high-performance processors. In *Proceedings of the 31st Annual International Symposium*

*on Computer Architecture*, ISCA '04, page 212, USA, 2004. IEEE Computer Society. ISBN 0769521436. 2.3.1, 5.2.1

[14] Amazon. Aws lambda, . https://aws.amazon.com/lambda/. 2.3.2, 6

[15] Amazon. Aws lambda pricing, . https://aws.amazon.com/lambda/pricing/. 2.3.2, 6, 6.4.6

[16] C Scott Ananian, Krste Asanovic, Bradley C Kuszmaul, Charles E Leiserson, and Sean Lie. Unbounded transactional memory. In *11th International Symposium on High-Performance Computer Architecture*, pages 316–327. IEEE, 2005. 2.1.6, 2.2.1, 3.3.2

[17] Alexandra Angerd, Angelos Arelakis, Vasilis Spiliopoulos, Erik Sintorn, and Per Stenström. Gbdi: Going beyond base-delta-immediate compression with global bases. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1115–1127. IEEE, 2022. 5.7.3

[18] Mohamed Arafa, Bahaa Fahim, Sailesh Kottapalli, Akhilesh Kumar, Lily P Looi, Sreenivas Mandava, Andy Rudoff, Ian M Steiner, Bob Valentine, Geetha Vedaraman, et al. Cascade lake: Next generation intel xeon scalable processor. *IEEE Micro*, 39(2):29–36, 2019. 1

[19] Angelos Arelakis and Per Stenstrom. Sc2: A statistical compression cache scheme. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 145–156. IEEE Press, 2014. ISBN 9781479943944. 1.2, 2.3.1, 5.7.1

[20] Angelos Arelakis, Fredrik Dahlgren, and Per Stenstrom. Hycomp: A hybrid cache compression method for selection of data-type-specific compression methods. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 38–49, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340342. doi: 10.1145/2830772.2830823. URL https://doi.org/10.1145/2830772.2830823. 5.7.1

[21] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind logging. *VLDB*, 2016. 2.2.2

[22] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *VLDB*, January 2018. 2.2.2

[23] Utku Aydonat and Tarek S Abdelrahman. Hardware support for relaxed concurrency control in transactional memory. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 15–26. IEEE, 2010. 1.1, 2.2.1, 3.1.3, 3.3.2, 3.6.1

[24] Seungcheol Baek, Hyung Gyu Lee, Chrysostomos Nicopoulos, Junghee Lee, and Jongman Kim. Ecm: Effective capacity maximizer for high-performance compressed caching. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 131–142. IEEE, 2013. 5.6.1

[25] Seungcheol Baek, Hyung Gyu Lee, Chrysostomos Nicopoulos, Junghee Lee, and Jongman Kim. Ecm: Effective capacity maximizer for high-performance compressed caching. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 131–142. IEEE, 2013. 5.2.4

[26] Seungcheol Baek, Hyung Gyu Lee, Chrysostomos Nicopoulos, Junghee Lee, and Jong-

man Kim. Size-aware cache management for compressed cache architectures. *IEEE Transactions on Computers*, 64(8):2337–2352, 2014. 5.6.1

[27] J-L Baer and W-H Wang. On the inclusion properties for multi-level cache hierarchies. *Computer Architecture News*, 1988. 2.2.2, 4

[28] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. Califor-nia, USA, 2005. 6.4.1

[29] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995. 3.7.1

[30] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11): 117–128, 2000. 2.3.1, 5, 5.2.2, 5.4.2

[31] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. *OOPSLA*, 2016. 2.2.2

[32] Abhishek Bhattacharjee. Translation-triggered prefetching. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–76, 2017. 2.1.4

[33] Abhishek Bhattacharyya, Abhijith Somashekhar, and Joshua San Miguel. Nvmr: non-volatile memory renaming for intermittent computing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 1–13, 2022. 2.2.2, 7.3

[34] Jayaram Bobba, Kevin E Moore, Haris Volos, Luke Yen, Mark D Hill, Michael M Swift, and David A Wood. Performance pathologies in hardware transactional memory. *ACM SIGARCH Computer Architecture News*, 35(2):81–91, 2007. 2.2.1, 3.1.3

[35] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Putting the" micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 645–650, 2018. 6.5

[36] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020. 6, 6.5

[37] Miao Cai, Chance C Coats, and Jian 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)*, pages 584–596. IEEE, 2020. 2.2.2

[38] Hasan Cam, Mostafa Abd-El-Barr, and Sadiq M Sait. A high-performance hardware-efficient memory allocation technique and design. In *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No. 99CB37040)*, pages 274–276. IEEE, 1999. 2.3.2

[39] Calin Cascaval, Colin Blundell, Maged Michael, Harold W Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Communications of the ACM*, 51(11):40–46, 2008. 3.7.4

[40] Daniel Castro, Alexandro Baldassin, João Barreto, and Paolo Romano. {SPHT}: Scalable persistent hardware transactions. In *19th USENIX Conference on File and Storage*

*Technologies (FAST 21)*, pages 155–169, 2021. 2.2.2, 3.7.3

[41] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. *ACM SIGARCH Computer Architecture News*, 34 (2):227–238, 2006. 2.2.1, 3, 3.3.2

[42] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. Bulksc: Bulk enforcement of sequential consistency. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 278–289, 2007. 2.2.1, 3

[43] Hassan Chafi, Jared Casper, Brian D Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun. A scalable, non-blocking approach to transactional memory. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 97–108. IEEE, 2007. 2.2.1, 3

[44] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. *OOPSLA*, 2014. 2.2.2

[45] J. Morris Chang and Edward F. Gehringer. A high performance memory allocator for object-oriented systems. *IEEE Transactions on Computers*, 45(3):357–366, 1996. 2.3.2

[46] J Morris Chang, Witawas Srisa-An, and C-TD Lo. Architectural support for dynamic memory management. In *Proceedings 2000 International Conference on Computer Design*, pages 99–104. IEEE, 2000. 2.3.2

[47] Xi Chen, Lei Yang, Robert P. Dick, Li Shang, and Haris Lekatsas. C-pack: A high-performance microprocessor cache compression algorithm. *IEEE Trans. Very Large Scale Integr. Syst.*, 18(8):1196–1208, August 2010. ISSN 1063-8210. doi: 10.1109/TVLSI.2009.2020989. URL `https://doi.org/10.1109/TVLSI.2009.2020989`. 5, 5.3.1, 5.3.2, 5.5, 5.7.1

[48] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1077–1091, 2020. 2.2.2

[49] Sangyeun Cho and Hyunjin Lee. Flip-n-write: A simple deterministic technique to improve pram write performance, energy and endurance. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 347–357, 2009. 4.5

[50] Marcelo Cintra, José F Martínez, and Josep Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 13–24, 2000. 2.2.1

[51] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. Fine-grain checkpointing with in-cache-line logging. *ASPLOS*, 2019. 2.2.2

[52] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010. 5.6.1

[53] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium*

*on Cloud computing*, pages 143–154. ACM, 2010. 3.6.2

[54] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, pages 64–78, 2021. 6.1.2, 6.4.1

[55] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. *SPAA*, 2018. 2.2.2

[56] Tom Coughlin. Intel winding down its optane memory business. https://www.forbes.com/sites/tomcoughlin/2022/07/28/intel-winding-down-its-optane-memory-business/, 2022. 7

[57] crun. crun container official github repo. https://github.com/containers/crun/. 6.4.1

[58] Mahesh Dananjaya, Vasilis Gavrielatos, Arpit Joshi, and Vijay Nagarajan. Lazy release persistency. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1173–1186, 2020. 2.2.2

[59] Zheng Dang, Shuibing He, Peiyi Hong, Zhenxin Li, Xuechen Zhang, Xian-He Sun, and Gang Chen. Nvalloc: rethinking heap metadata management in persistent memory allocators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–127, 2022. 2.2.2

[60] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. *ASPLOS*, 2015. 2.2.2

[61] A Micro Devices. Amd64 architecture programmer's manual volume 2: System programming, 2006. 4.2

[62] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. *SIGMOD*, 2013. 3.7.4, 4.3.4

[63] Dave Dice, Timothy L Harris, Alex Kogan, Yossi Lev, and Mark Moir. Hardware extensions to make lazy subscription safe. *arXiv preprint arXiv:1407.6968*, 2014. 2.2.1

[64] Nuno Diegues and Paolo Romano. Time-warp: lightweight abort minimization in transactional memory. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178, 2014. 3.7.4

[65] K. Doshi, E. Giles, and P. Varman. Atomic persistence for scm with a non-intrusive backend controller. *HPCA*, 2016. 2.2.2

[66] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro*, 37(2):52–62, 2017. 1

[67] J R Driscoll, N Sarnak, D D Sleator, and R E Tarjan. Making data structures persistent. *STOC*, 1986. 2.2.2

[68] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan

Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, 2020. 6, 6.5

[69] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–15, 2014. 2.2.2

[70] Julien Dusser, Thomas Piquet, and André Seznec. Zero-content augmented caches. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, page 46–55, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584980. doi: 10.1145/1542275.1542288. URL `https://doi.org/10.1145/1542275.1542288`. 2.3.1, 5.3.3, 5.7.1

[71] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 2002. 4.1.4

[72] Jason Evans. A scalable concurrent malloc implementation for freebsd. In *Proc. of the bsdcan conference, ottawa, canada*, 2006. 2.3.1, 5, 5.2.2, 5.4.2, 6.4.1

[73] Ricardo Fernández-Pascual, Alberto Ros, and Manuel E Acacio. To be silent or not: On the impact of evictions of clean data in cache-coherent multicores. *The Journal of Supercomputing*, 73(10):4428–4443, 2017. 3.3.1

[74] Tais B. Ferreira, Rivalino Matias, Autran Macedo, and Lucio B. Araujo. An experimental study on memory allocators in multicore and multithreaded applications. In *2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 92–98, 2011. doi: 10.1109/PDCAT.2011.18. 6

[75] Jordan Fix, Nayana P Nagendra, Sotiris Apostolakis, Hansen Zhang, Sophie Qiu, and David I August. Hardware multithreaded transactions. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 15–29, 2018. 2.2.1, 3.2.2, 7.3

[76] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9. URL `https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi`. 2.3.2, 6

[77] Richard F Freitas and Winfried W Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 2008. 2.2.2

[78] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 386–400, 2021. 6, 6.5

[79] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019. 6.1.2, 6.4.1, 6.4.4

[80] Shen Gao, Bingsheng He, and Jianliang Xu. Real-time in-memory checkpointing for future hybrid memory systems. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 263–272, 2015. 2.2.2

[81] Jayesh Gaur, Alaa R. Alameldeen, and Sreenivas Subramoney. Base-victim compression: An opportunistic cache compression architecture. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 317–328. IEEE Press, 2016. ISBN 9781467389471. doi: 10.1109/ISCA.2016.36. URL https://doi.org/10.1109/ISCA.2016.36. 2.3.1

[82] Kaan Genç, Michael D Bond, and Guoqing Harry Xu. Crafty: Efficient, htm-compatible persistent transactions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–74, 2020. 2.2.2, 3.7.3

[83] Amin Ghasemazar, Mohammad Ewais, Prashant Nair, and Mieszko Lis. 2dcc: Cache compression in two dimensions. In *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020*, pages 756–761. IEEE, 2020. doi: 10.23919/DATE48585.2020.9116279. URL https://doi.org/10.23919/DATE48585.2020.9116279. 1.2, 2.3.1

[84] Amin Ghasemazar, Prashant Nair, and Mieszko Lis. Thesaurus: Efficient cache compression via dynamic clustering. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 527–540, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378518. URL https://doi.org/10.1145/3373376.3378518. 1.2, 2.3.1, 5, 5.1.1, 5.3.1, 5.5, 5.6, 5.7.2

[85] Sanjay Ghemawat and Paul Menage. Tcmalloc: Thread-caching malloc, 2009. https://google.github.io/tcmalloc/design.html. 2.3.1, 5, 5.2.2, 5.4.2, 6.3

[86] Ellis R Giles, Kshitij Doshi, and Peter Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2015. 2.2.2

[87] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. Relaxed persist ordering using strand persistency. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 652–665. IEEE, 2020. 2.2.2

[88] James R Goodman. Coherency for multiprocessor virtual address caches. *ACM SIGARCH Computer Architecture News*, 15(5):72–81, 1987. 1

[89] Google. Google cloud functions. https://cloud.google.com/functions/. 2.3.2, 6

[90] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A scalable and efficient persistent transactional memory. *ATC*, 2019. 2.2.2

[91] Yuncheng Guo, Yu Hua, and Pengfei Zuo. Dfpc: A dynamic frequent pattern compression scheme in nvm-based main memory. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1622–1627. IEEE, 2018. 4.5

[92] Siddharth Gupta, Alexandros Daglis, and Babak Falsafi. Distributed logless atomic durability with persistent memory. *MICRO*, 2019. 2.2.2, 4.3.1

[93] E. G. Hallnor and S. K. Reinhardt. A unified compressed memory hierarchy. In *11th International Symposium on High-Performance Computer Architecture*, pages 201–212, 2005. 2.3.1

[94] Erik G. Hallnor and Steven K. Reinhardt. A fully associative software-managed cache design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, page 107–116, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132328. doi: 10.1145/339647.339660. URL `https://doi.org/10.1145/339647.339660`. 5.7.2

[95] Per Hammarlund, Alberto J Martinez, Atiq A Bajwa, David L Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, et al. Haswell: The fourth-generation intel core processor. *IEEE micro*, 34(2):6–20, 2014. 1

[96] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. *ACM SIGOPS Operating Systems Review*, 32(5):58–69, 1998. 2.2.1

[97] Lance Hammond, Vicky Wong, Mike Chen, Brian D Carlstrom, John D Davis, Ben Hertzberg, Manohar K Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. *ACM SIGARCH Computer Architecture News*, 32(2):102, 2004. 1.1, 2.2.1, 3, 3.1.3, 3.6.1, 3.6.3

[98] Swapnil Haria, Mark D Hill, and Michael 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*, pages 775–788, 2020. 2.2.2

[99] Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, 1993. 2.2.1, 3, 3.6.1

[100] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, 2003. 3.3.3

[101] Seokin Hong, Bulent Abali, Alper Buyuktosunoglu, Michael B. Healy, and Prashant J. Nair. Touché: Towards ideal and efficient cache compression by mitigating tag area overheads. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 453–465, New York, NY, USA, 2019. Association

for Computing Machinery. ISBN 9781450369381. doi: 10.1145/3352460.3358281. URL https://doi.org/10.1145/3352460.3358281. 2.3.1

[102] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. *EuroSys*, 2017. 2.2.2

[103] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-structured non-volatile main memory. *ATC*, 2017. 2.2.2, 4.3.3

[104] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. 2.3.1, 5.7.1

[105] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. *FAST*, 2018. 2.2.2

[106] IBM. Ibm cloud functions. https://www.ibm.com/cloud/functions/. 2.3.2, 6

[107] Intel. Intel® transactional synchronization extension (intel® tsx) disable update for selected processors. 4

[108] Intel. 5-level paging and 5-level ept, 2017. 4.3.1

[109] Intel. Intel 64 and ia-32 architectures software developer's manual. 1(64):64, 64. 2.2.1, 2.2.2, 3.6.1, 3.7.2

[110] M. M. Islam and P. Stenstrom. Zero-value caches: Cancelling loads that return zero. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 237–245, 2009. 2.3.1, 5.3.3, 5.7.1

[111] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. *ASPLOS*, 2016. 2.2.2

[112] Ken Jacobs, R Bamford, G Doherty, K Haas, M Holt, F Putzolu, and B Quigley. Concurrency control, transaction isolation and serializability in sql92 and oracle7. *Oracle White Paper, Part*, (A33745), 1995. 3.7.1

[113] Syed Ali Raza Jafri, Gwendolyn Voskuilen, and TN Vijaykumar. Wait-n-gotm: improving htm performance by serializing cyclic dependencies. *ACM SIGPLAN Notices*, 48(4): 521–534, 2013. 2.2.1

[114] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies. *MICRO*, 2010. 2.2.2, 4

[115] jemalloc. jemalloc official github repo. https://github.com/jemalloc/jemalloc/. 6.3, 6.4.1

[116] J. Jeong, C. H. Park, J. Huh, and S. Maeng. Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory. *MICRO*, 2018. 2.2.2

[117] Jungi Jeong and Changhee Jung. Pmem-spec: Persistent memory speculation. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021. 2.2.2

[118] Jungi Jeong, Jaewan Hong, Seungryoul Maeng, Changhee Jung, and Youngjin Kwon. Unbounded hardware transactional memory for a hybrid dram/nvm memory system. In

*2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 525–538. IEEE, 2020. 2.2.2, 3.7.3

[119] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, New York, NY, USA, 2021. Association for Computing Machinery. doi: 10.1145/3445814.3446701. URL https://doi.org/10.1145/3445814.3446701. 2.3.2, 6

[120] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, 2021. doi: 10.1145/3448016.3459240. URL https://doi.org/10.1145/3448016.3459240. 2.3.2, 6

[121] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. Atom: Atomic durability in non-volatile memory through hardware logging. *HPCA*, 2017. 2.2.2

[122] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. Dhtm: Durable hardware transactional memory. *ISCA*, 2018. 2.2.2, 3.7.3

[123] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Efficient persist barriers for multicores. *MICRO*, 2015. 2.2.2

[124] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. *SOSP*, 2019. 2.2.2

[125] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 158–169, 2015. 2.3.2, 6

[126] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. Mallacc: Accelerating memory allocation. *ACM SIGPLAN Notices*, 52(4):33–45, 2017. 1.2, 2.3.2, 6.4.9

[127] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. Improving in-memory database index performance with intel® transactional synchronization extensions. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 476–487. IEEE, 2014. 3.7.2

[128] Rajat Kateja. *Reducing Performance Overhead of Direct Access NVM Storage Redundancy*. PhD thesis, Carnegie Mellon University, 2020. 4.5

[129] Rajat Kateja, Andy Pavlo, and Gregory Ganger. Vilamb: Low overhead asynchronous redundancy for direct access nvm. *Carnegie Mellon University Parallel Data Lab*, 2019. technical report CMU-PDL-20-101. 4.5

[130] Rajat Kateja, Nathan Beckmann, and Gregory Ganger. Tvarak: software-managed hardware offload for redundancy in direct-access nvm storage. *ISCA*, 2020. 4.5

[131] Changkyu Kim, Doug Burger, and Stephen W Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *ASPLOS*, 2002. 2.2.2, 4

[132] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. IEEE, 2019. 5.6.1, 6.1.2, 6.4.1

[133] Jinkwon Kim, Mincheol Kang, Jeongkyu Hong, and Soontae Kim. Exploiting inter-block entropy to enhance the compressibility of blocks with diverse data. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1100–1114. IEEE, 2022. 5.7.3

[134] Jungrae Kim, Michael Sullivan, Esha Choukse, and Mattan Erez. Bit-plane compression: Transforming data for better compression in many-core architectures. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 329–340. IEEE Press, 2016. ISBN 9781467389471. doi: 10.1109/ISCA.2016.37. URL https://doi.org/10.1109/ISCA.2016.37. 5.7.1

[135] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. Nvwal: Exploiting nvram in write-ahead logging. *ASPLOS*, 2016. 2.2.2

[136] Apostolos Kokolis, Thomas Shull, Jian Huang, and Josep Torrellas. P-inspect: Architectural support for programmable non-volatile memory frameworks. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 509–524. IEEE, 2020. 2.2.2

[137] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. Delegated persist ordering. *MICRO*, 2016. 2.2.2

[138] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. *ASPLOS*, 2016. 2.2.2

[139] Kunal Korgaonkar, Joseph Izraelevitz, Jishen Zhao, and Steven Swanson. Vorpal: Vector clock ordering for large persistent memory systems. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 435–444, 2019. 4.1.1

[140] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable transactional memory can scale with timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 335–349, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378483. URL https://doi.org/10.1145/3373376.3378483. 2.2.2

[141] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 1978. 4.1.4

[142] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *VLDB*, 2011. 3.7.4, 4.3.4

[143] Chin Yang Lee. An algorithm for path connections and its applications. *IRE transactions*

*on electronic computers*, (3):346–365, 1961. 3.6.2

[144] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim. An on-chip cache compression technique to reduce decompression overhead and design complexity. *Journal of systems Architecture*, 46(15):1365–1382, 2000. 2.3.1

[145] Sangho Lee, Teresa Johnson, and Easwaran Raman. Feedback directed optimization of tcmalloc. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450329170. doi: 10.1145/2618128.2618131. URL `https://doi.org/10.1145/2618128.2618131`. 6

[146] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: write optimal radix tree for persistent memory storage systems. *FAST*, 2017. 2.2.2

[147] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. *SOSP*, 2019. 2.2.2

[148] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. *ICDE*, 2013. 4.4.1

[149] Viktor Leis, Alfons Kemper, and Thomas Neumann. Scaling htm-supported database transactions to many cores. *IEEE Transactions on Knowledge and Data Engineering*, 28 (2):297–310, 2015. 3.7.2

[150] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The art of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, pages 1–8, 2016. 5.6.1

[151] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The art of practical synchronization. *DaMoN*, 2016. 4.4.1

[152] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. Dramsim3: a cycle-accurate, thermal-capable dram simulator. *IEEE Computer Architecture Letters*, 19 (2):106–109, 2020. 6.4.1

[153] Wentong Li, Saraju Mohanty, and Krishna Kavi. A page-based hybrid (software-hardware) dynamic memory allocator. *IEEE Computer Architecture Letters*, 5(2):13–13, 2006. 2.3.2

[154] Wentong Li, Mehran Rezaei, Krishna Kavi, Afrin Naz, and Philip Sweany. Feasibility of decoupling memory management from the execution pipeline. *Journal of Systems Architecture*, 53(12):927–936, 2007. 2.3.2

[155] The GNU C Library. The gnu allocator. https://www.gnu.org/software/libc/manual/html_node/The-GNU-Allocator.html. 6.3

[156] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P Stevenson. Si-tm: Reducing transactional memory abort rates through snapshot isolation. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 383–398, 2014. 1.1, 2.2.1, 3.1.3, 3.2.5, 3.6.1, 3.6.2, 3.7.1

[157] Chunwei Liu, McKade Umbenhower, Hao Jiang, Pranav Subramaniam, Jihong Ma, and Aaron J Elmore. Mostly order preserving dictionaries. In *2019 IEEE 35th International*

*Conference on Data Engineering (ICDE)*, pages 1214–1225. IEEE, 2019. 5.6.1

[158] Jihang Liu, Shimin Chen, and Lujun Wang. Lb+trees: Optimizing persistent index performance on 3dxpoint memory. *Proceedings of the VLDB Endowment*, 13(7):1078–1090, 2020. 2.2.2, 3.7.3

[159] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. *ASPLOS*, 2017. 2.2.2, 3.7.3

[160] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. *MICRO*, 2018. 2.2.2

[161] S. Liu, A. Kolli, J. Ren, and S. Khan. Crash consistency in encrypted non-volatile main memory systems. *HPCA*, 2018. 4.5

[162] Sihang Liu, Korakit Seemakhupt, Gennady Pekhimenko, Aasheesh Kolli, and Samira Khan. Janus: Optimizing memory and storage support for non-volatile memory systems. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 143–156. IEEE, 2019. 4.5

[163] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. Intermittent computing: Challenges and opportunities. *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, 2017. 7.3

[164] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *PLDI*, 2005. 4.4.1

[165] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai 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*, pages 757–773, 2020. 2.2.2

[166] Linux manual. mmap(2) - linux manual page. https://man7.org/linux/man-pages/man2/mmap.2.html. 6.1.1

[167] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards practical default-on multi-core record/replay. *ASPLOS*, 2017. 2.2.2, 6, 4, 4.1.1

[168] Hugh McIntyre, Srikanth Arekapudi, Eric Busta, Timothy Fischer, Michael Golden, Aaron Horiuchi, Tom Meneghini, Samuel Naffziger, and James Vinh. Design of the two-core x86-64 amd "bulldozer" module in 32 nm soi cmos. *IEEE journal of solid-state circuits*, 2011. 4.1.5

[169] Sally A McKee. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, page 162, 2004. 1

[170] Amirsaman Memaripour, Joseph Izraelevitz, and Steven 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*, pages 789–806, 2020. 2.2.2

[171] Microsoft. Azure functions. https://azure.microsoft.com/en-us/services/functions/. 2.3.2, 6

[172] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. Doppelgänger: a cache for approximate computing. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 50–61, 2015. 5.7.4

[173] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. *2008 IEEE International Symposium on Workload Characterization*, 2008. 3.6.1, 4.4.1

[174] A. Miraglia, D. Vogt, H. Bos, A. Tanenbaum, and C. Giuffrida. Peeking into the past: Efficient checkpoint-assisted time-traveling debugging. *ISSRE*, 2016. 2.2.2, 4

[175] A. Mirhosseini, A. Agrawal, and J. Torrellas. Survive: Pointer-based in-dram incremental checkpointing for low-cost data persistence and rollback-recovery. *IEEE Computer Architecture Letters*, 2017. 2.2.2

[176] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 1992. 2.2.2

[177] Kevin E Moore, Jayaram Bobba, Michelle J Moravan, Mark D Hill, and David A Wood. Logtm: Log-based transactional memory. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pages 254–265. IEEE, 2006. 1.1, 2.2.1, 3.1.3

[178] Sobhan Moosavi, Mohammad Hossein Samavatian, Arnab Nandi, Srinivasan Parthasarathy, and Rajiv Ramnath. Short and long-term pattern discovery over large-scale geospatiotemporal data. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2905–2913, 2019. 5.6.1

[179] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. *TRIOS*, 2013. 2.2.2

[180] Michelle J Moravan, Jayaram Bobba, Kevin E Moore, Luke Yen, Mark D Hill, Ben Liblit, Michael M Swift, and David A Wood. Supporting nested transactional memory in logtm. *ACM SIGARCH Computer Architecture News*, 34(5):359–370, 2006. 2.2.1, 3.2.1, 3.4

[181] David Mulnix. Intel xeon processor d product family technical overview. *Retrieved April*, 12:2020, 2015. 4.1.1

[182] David Mulnix. Intel xeon processor scalable family technical overview, 2017. 2.2.2, 4

[183] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, 27:28, 2009. 5.5, 6.4.8

[184] Moohyeon Nam, Hokeun Cha, Young-Ri Choi, Sam H. Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. *FAST*, 2019. 2.2.2

[185] Nevine Nassif, Ashley O Munch, Carleton L Molnar, Gerald Pasdast, Sitaraman V Lyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, et al. Sapphire rapids: The next-generation intel xeon scalable processor. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 44–46. IEEE,

2022. 1

[186] Ian Neal, Gefei Zuo, Eric Shiple, Tanvir Ahmed Khan, Youngjin Kwon, Simon Peter, and Baris Kasikci. Rethinking file mapping for persistent memory. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 97–111, 2021. 2.2.2, 3.7.3

[187] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. *SIGMOD*, 2015. 3.7.4, 4.3.4

[188] Tri M. Nguyen and David Wentzlaff. Morc: A manycore-oriented compressed cache. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 76–88, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340342. doi: 10.1145/2830772.2830828. URL https://doi.org/10.1145/2830772.2830828. 2.3.1

[189] Tri Minh Nguyen and David Wentzlaff. Picl: A software-transparent, persistent cache log for nonvolatile main memory. *MICRO*, 2018. 1.1, 2.2.2, 4, 4.4.1, 4.4.2

[190] Yuanjiang Ni, Jishen Zhao, Heiner Litz, Daniel Bittman, and Ethan L. Miller. Ssp: Eliminating redundant writes in failure-atomic nvrams via shadow sub-paging. *MICRO*, 2019. 1.1, 2.2.2

[191] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. {SOCK}: Rapid task provisioning with {Serverless-Optimized} containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, 2018. 6, 6.5

[192] CPython official repo. The python benchmark suite. https://github.com/python/pyperformance/. 6.1.2, 6.4.1

[193] M. A. Ogleari, E. L. Miller, and J. Zhao. Steal but no force: Efficient hardware undo+redo logging for persistent memory systems. *HPCA*, 2018. 2.2.2

[194] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. *SIGMOD*, 2016. 2.2.2, 3.7.3

[195] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996. 4.3.1

[196] B. Panda and A. Seznec. Dictionary sharing: An efficient cache compression scheme for compressed caches. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016. 2.3.1, 5, 5.1.1, 5.1.2

[197] Mark S Papamarcos and Janak H Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th annual international symposium on Computer architecture*, pages 348–354, 1984. 2.2.1, 4.2

[198] Sungbo Park, Ingab Kang, Yaebin Moon, Jung Ho Ahn, and G Edward Suh. Bcd deduplication: Effective memory compression using partial cache-line deduplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 52–64, 2021. 5.7.3

[199] Sunjae Park, Christopher J Hughes, and Milos Prvulovic. Forgive-tm: Supporting lazy

conflict detection in eager hardware transactional memory. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 192–204. IEEE, 2019. 2.2.1

[200] Liam Patterson, David Pigorovsky, Brian Dempsey, Nikita Lazarev, Aditya Shah, Clara Steinhoff, Ariana Bruno, Justin Hu, and Christina Delimitrou. Hivemind: a hardware-software system stack for serverless edge swarms. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 800–816, 2022. 2.3.2, 6

[201] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, page 377–388, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311823. doi: 10.1145/2370816.2370870. URL `https://doi.org/10.1145/2370816.2370870`. 2.3.1, 5.7.1

[202] Gennady Pekhimenko, Tyler Huberty, Rui Cai, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. Exploiting compressed block size as an indicator of future reuse. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 51–63. IEEE, 2015. 5.2.4

[203] Gennady Pekhimenko, Tyler Huberty, Rui Cai, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. Exploiting compressed block size as an indicator of future reuse. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 51–63. IEEE, 2015. 5.6.1

[204] Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 16–25, 2010. 3.7.4

[205] Milos Prvulovic, Zheng Zhang, and Josep Torrellas. Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. *Computer Architecture News*, 2002. 2.2.2

[206] Python. Cpython official github repo. https://github.com/python/cpython/. 6.4.1

[207] Xuehai Qian, Wonsun Ahn, and Josep Torrellas. Scalablebulk: Scalable cache coherence for atomic blocks in a lazy environment. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 447–458. IEEE, 2010. 2.2.1, 3

[208] Xuehai Qian, Josep Torrellas, Benjamin Sahelices, and Depei Qian. Bulkcommit: scalable and fast commit of atomic blocks in a lazy multiprocessor environment. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 371–382, 2013. 2.2.1, 3

[209] Xuehai Qian, Benjamin Sahelices, and Josep Torrellas. Omniorder: Directory-based conflict serialization of transactions. *ACM SIGARCH Computer Architecture News*, 42(3): 421–432, 2014. 1.1, 2.2.1

[210] Moinuddin K. Qureshi, David Thompson, and Yale N. Patt. The v-way cache: De-

mand based associativity via global replacement. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, page 544–555, USA, 2005. IEEE Computer Society. ISBN 076952270X. doi: 10.1109/ISCA.2005.52. URL https://doi.org/10.1109/ISCA.2005.52. 5.7.2

[211] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 494–505. IEEE, 2005. 2.1.6, 2.2.1, 3.3.2

[212] Hany E Ramadan, Christopher J Rossbach, and Emmett Witchel. Dependence-aware transactional memory for increased concurrency. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 246–257. IEEE, 2008. 2.2.1, 3.2.2

[213] Vijayshankar Raman and Garret Swart. How to wring a table dry: Entropy compression of relations and querying of compressed relations. In *Proceedings of the 32nd international conference on Very large data bases*, pages 858–869, 2006. 5.1.1

[214] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. Thynvm: Enabling software-transparent crash consistency in persistent memory systems. *MICRO*, 2015. 2.2.2, 4.4.1

[215] Arun F Rodrigues, Gwendolyn Renae Voskuilen, Simon David Hammond, and Karl Scott Hemmert. Structural simulation toolkit (sst). Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2016. 6.4.1

[216] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 1–17, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386388. doi: 10.1145/3472883.3486972. URL https://doi.org/10.1145/3472883.3486972. 2.3.2, 6

[217] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM TOCS*, 1992. 4.3.3

[218] Stephen M Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for dram-based storage. *FAST*, 2014. 4.3.3

[219] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. *ISCA*, 2013. 3.6.1, 4.4.1, 5.6

[220] Somayeh Sardashti and David A. Wood. Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, page 62–73, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450326384. doi: 10.1145/2540708.2540715. URL https://doi.org/10.1145/2540708.2540715. 2.3.1, 5.1.2, 5.2.1, 5.7.2

[221] Somayeh Sardashti, André Seznec, and David A. Wood. Skewed compressed caches. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, page 331–342, USA, 2014. IEEE Computer Society. ISBN 9781479969982.

doi: 10.1109/MICRO.2014.41. URL https://doi.org/10.1109/MICRO.2014.41. 2.3.1

[222] Somayeh Sardashti, Andre Seznec, and David A. Wood. Yet another compressed cache: A low-cost yet effective compressed cache. *ACM Trans. Archit. Code Optim.*, 13(3), September 2016. ISSN 1544-3566. doi: 10.1145/2976740. URL https://doi.org/10.1145/2976740. 2.3.1, 5.7.2

[223] David Schall, Artemiy Margaritov, Dmitrii Ustiugov, Andreas Sandberg, and Boris Grot. Lukewarm serverless functions: characterization and optimization. In *ISCA*, pages 757–770, 2022. 6.4.1, 6.5

[224] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. nvm malloc: Memory allocation for NVRAM. *ADMS*, 2015. 2.2.2

[225] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam H. Noh. Failure-atomic slotted paging for persistent memory. *ASPLOS*, 2017. 2.2.2, 3.7.3

[226] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, Todd C. Mowry, and Trishul Chilimbi. Page overlays: An enhanced virtual memory framework to enable fine-grained memory management. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 79–91, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334020. 1.3, 2.1.1, 2.1.4, 2.1.6

[227] A. Seznec. Decoupled sectored caches: conciliating low tag implementation cost and low miss ratio. In *Proceedings of 21 International Symposium on Computer Architecture*, pages 384–393, 1994. 5.7.2

[228] Ali Shafiee, Meysam Taassori, Rajeev Balasubramonian, and Al Davis. Memzip: Exploring unconventional benefits from memory compression. *HPCA*, 2014. 4.2.5

[229] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020. 2.3.2, 6, 6.5

[230] Sara Mahdizadeh Shahri, Seyed Armin Vakil Ghahani, and Aasheesh Kolli. (almost) fence-less persist ordering. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 539–554. IEEE, 2020. 2.2.2

[231] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 323–337, 2017. 2.2.2

[232] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433, 2020. 6.5

[233] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. Proteus: A flexible and fast software supported hardware logging approach for nvm. *MICRO*, 2017. 2.2.2

[234] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L Scott. Flexible decoupled transactional memory support. In *2008 International Symposium on Computer Architecture*, pages 139–150. IEEE, 2008. 2.2.1, 3.3.2

[235] Abraham Silberschatz, Henry F Korth, Shashank Sudarshan, et al. *Database system concepts*, volume 5. McGraw-Hill New York, 2002. 3, 3.7.1

[236] Sudarshan M Srinivasan, Srikanth Kandula, Christopher R Andrews, Yuanyuan Zhou, et al. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. *ATC*, 2004. 6, 4.1.1

[237] J Greggory Steffan, Christopher B Colohan, Antonia Zhai, and Todd C Mowry. A scalable approach to thread-level speculation. *ACM SIGARCH Computer Architecture News*, 28(2): 1–12, 2000. 2.2.1, 7.3

[238] J Gregory Steffan and Todd C Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings 1998 Fourth International Symposium on High-Performance Computer Architecture*, pages 2–13. IEEE, 1998. 2.2.1, 7.3

[239] Paul Sweazey and Alan Jay Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. *ACM SIGARCH Computer Architecture News*, 14(2): 414–423, 1986. 2.2.1

[240] Simon M Tam, Harry Muljono, Min Huang, Sitaraman Iyer, Kalapi Royneogi, Nagmohan Satti, Rizwan Qureshi, Wei Chen, Tom Wang, Hubert Hsieh, et al. Skylake-sp: A 14nm 28-core xeon® processor. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 34–36. IEEE, 2018. 1

[241] Michael E Thomadakis. The architecture of the nehalem processor and nehalem-ep smp platforms. *Texas A&M Tech Report*, 2011. 4.2

[242] Yingying Tian, Samira M. Khan, Daniel A. Jiménez, and Gabriel H. Loh. Last-level cache deduplication. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, page 53–62, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326421. doi: 10.1145/2597652.2597655. URL `https://doi.org/10.1145/2597652.2597655`. 1.2, 2.3.1

[243] Saša Tomić, Cristian Perfumo, Chinmay Kulkarni, Adria Armejach, Adrián Cristal, Osman Unsal, Tim Harris, and Mateo Valero. Eazyhtm: Eager-lazy hardware transactional memory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 145–155, 2009. 2.2.1

[244] P. Tsai, Y. L. Gan, and D. Sanchez. Rethinking the memory hierarchy for modern languages. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 203–216, 2018. 2.3.1

[245] Po-An Tsai and Daniel Sanchez. Compress objects, not cache lines: An object-based compressed memory hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 229–242, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304006. URL

https://doi.org/10.1145/3297858.3304006. 1.2, 2.3.1, 5, 5.3.1, 5.5, 5.6

[246] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013. 5.6.1

[247] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 559–572, 2021. 6, 6.5

[248] Marina Vemmou and Alexandros Daglis. Cosplay: Leveraging task-level parallelism for high-throughput synchronous persistence. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 86–99, 2021. 2.2.2

[249] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. *FAST*, 2011. 2.2.2

[250] Nicolas Viennot, Siddharth Nair, and Jason Nieh. Transparent mutable replay for multicore debugging and patch validation. *ASPLOS*, 2013. 2.2.2, 6, 4, 4.1.1

[251] Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. Easy lock-free indexing in non-volatile memory. *ICDE*, 2018. 2.2.2

[252] Craig Warner, Dan Robinson, John Wastlick, Michael Schroeder, and Jeffrey Moy. Non-inclusive cache systems and methods, 2014. US Patent 8,661,208. 2.2.2, 4

[253] Ian Watson, Chris Kirkham, and Mikel Luján. A study of a transactional parallel routing algorithm. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 388–398. IEEE Computer Society, 2007. 3.6.2

[254] Xueliang Wei, Dan Feng, Wei Tong, Jingning Liu, and Liuqing Ye. Morlog: Morphable hardware logging for atomic persistence in non-volatile main memory. *ISCA*, 2020. 2.2.2

[255] Kai Wu, Ivy Peng, Jie Ren, and Dong Li. Ribbon: High performance cache line flushing for persistent memory. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 427–439, 2020. 2.2.2

[256] Kai Wu, Jie Ren, Ivy Peng, and Dong Li. {ArchTM}:{Architecture-Aware}, high performance transaction for persistent memory. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 141–153, 2021. 2.2.2

[257] Song Wu, Fang Zhou, Xiang Gao, Hai Jin, and Jinglei Ren. Dual-page checkpointing: An architectural approach to efficient data persistence for in-memory applications. *ACM Trans. Archit. Code Optim.*, 2019. 2.2.2

[258] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *VLDB*, 2017. 3.7.4, 4.3.4

[259] Wm A Wulf and Sally A McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995. 1

[260] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-

volatile main memories. *FAST*, 2016. 2.2.2

[261] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. *ASPLOS*, 2019. 2.2.2

[262] Jie Xu, Dan Feng, Yu Hua, Wei Tong, Jingning Liu, and Chunyan Li. Extending the lifetime of nvms with compression. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1604–1609. IEEE, 2018. 4.5

[263] Yi Xu, Joseph Izraelevitz, and Steven Swanson. Clobber-nvm: log less, re-execute more. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 346–359, 2021. 2.2.2

[264] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory, 2019. 4.1.2

[265] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, 2020. 4.1.2

[266] Jun Yang and Rajiv Gupta. Energy efficient frequent value data cache design. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, page 197–207, Washington, DC, USA, 2002. IEEE Computer Society Press. ISBN 0769518591. 2.3.1

[267] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. *FAST*, 2015. 2.2.2

[268] Luke Yen. *Signatures in transactional memory systems*. PhD thesis, University of Wisconsin–Madison, 2009. 3.6.1

[269] Luke Yen, Jayaram Bobba, Michael R Marty, Kevin E Moore, Haris Volos, Mark D Hill, Michael M Swift, and David A Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 261–272. IEEE, 2007. 2.2.1, 3.3.2, 3.6.1

[270] Richard M Yoo, Christopher J Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2013. 2.2.1

[271] Vinson Young, Prashant J. Nair, and Moinuddin K. Qureshi. Deuce: Write-efficient encryption for non-volatile memories. *ASPLOS*, 2015. 4.5

[272] Qi Zeng, Rakesh Jha, Shigang Chen, and Jih-Kwon Peir. Data locality exploitation in cache compression. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 347–354. IEEE, 2018. 2.3.1

[273] Jialiang Zhang, Michael Swift, and Jing Li. Software-defined address mapping: a case on 3d memory. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 70–83, 2022. 7.4

[274] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A

reliable and highly-available non-volatile memory system. *ASPLOS*, 2015. 2.2.2, 4.3.4

[275] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. *MICRO*, 2013. 2.2.2

[276] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theor.*, 24(5):530–536, September 2006. ISSN 0018-9448. doi: 10.1109/TIT.1978.1055934. URL https://doi.org/10.1109/TIT.1978.1055934. 5.7.1

[277] P. Zuo, Y. Hua, M. Zhao, W. Zhou, and Y. Guo. Improving the performance and endurance of encrypted non-volatile main memory through deduplicating writes. *MICRO*, 2018. 4.1.1, 4.5

[278] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. *OSDI*, 2018. 2.2.2

[279] Pengfei Zuo, Yu Hua, and Yuan Xie. Supermem: Enabling application-transparent secure persistent memory with low overheads. *MICRO*, 2019. 4.5