Co-designing Reliability and Performance for Datacenter Memory

Adarsh Patil

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
The University of Edinburgh

2023

To my beloved Kumudha

Acknowledgments

Undertaking a PhD has been a truly humbling experience and this thesis is the epitome of human emotion and bonding. I have many to thank in this journey.

Think carefully

My advisor Vijay Nagarajan has been a great mentor. He has steered me towards interesting problems while also providing me the necessary freedom to explore and pursue ideas of my own. I am inspired by Vijay's approach to research - the wholistic understanding of a topic from first principles. Our brainstorming discussions were always insightful and thought provoking. His patience and encouragement during the difficult phase of my thesis was truly admirable. He was instrumental in setting up my collaborations with Rajeev Balasubramonian (University of Utah) and Nikos Nikoleris (ARM). I have enjoyed working with them and learnt a great deal over the course of the projects.

Being a better person

I wouldn't be where I am today without my wife, Kumudha. You have played an extraordinarily important role in the making of this thesis and I can only say that this is your dissertation, as much as it is mine. You have always lent me a patient ear and provided the best advice possible, at any and all times regardless of your work commitments. There are no words that can satisfactorily express my gratitude for the several sacrifices you've made to enable me to pursue this PhD. Your unyielding love and unwavering support has been my source of strength. You have stood by me through the highs and lows and always been there for me as my best friend, my confidant, my sounding board, my moral compass, my north star and to you, I dedicate this thesis.

If you want to go far, go together

I am grateful to have interacted with and received feedback from Prof. Murray Cole, Dr. Sam Ainsworth, Prof. Boris Grot, Prof. Michio Honda, Prof. Antonio Barbalace. Learning from the very best in the field has been a privilege, and has made a far reaching impact on my thought process. I thank the many anonymous reviewers from the Computer Architecture community who have vetted my work and helped improve their scientific value and resulting impact.

In the development of this thesis I have used a number of open source projects.

I thank the developers and contributors of gem5, SynchroTrace, Valgrind, FunctionBench, SeBS, numpywren, Kubernetes and so many more.

Rousseau's amour de soi (love thyself)

The highest improvements always lie at the edge of one's ability and the PhD process has rightly pushed me above and beyond mine. My sincere thanks to the following people for teaching me self-belief and showing me that my limits are only in my mind. My theraphist Artis Orbis, for guiding me to manage my debilitating panic attacks and to reconcile with my overly self-critical inner self. All the Pleasance gym instructors - Julie Paton-Monk, Fabrizio Di Battista, Linzi Drew, Max Freyne to name a few, who provided me the physical strength to endure and encouraged me to push harder. Their constant positivity and energy helped me cope with stress and never cave to pressure. The beautiful outdoors in the country of Scotland - Fife, Loch Lomond, Isle of Arran, amongst several others that helped me to recoup and rediscover myself and humbly embrace my limits, fully and without fear.

the community and a sense of belonging

I am grateful for all my friends Nicolai, Vasilis, Antonis, Andres, Sukarn, Karim, Dmitrii, Artemiy, Shyam for their valuable company, both technically and socially. I thank the faculty members and staff of the School of Informatics for all their support. Thanks are also due for the funding sources ARM Center of Excellence, ACM travel grant for supporting me financially and providing for my sustenance and living expenses.

my circle of strength

I express my profound gratitude to my parents for their endless love and belief in me. Everything good in me stems from them. Thank you both for providing me every opportunity to reach for the stars and chase my dreams.
Thanks are also due to my extended family for all their support. My grandfather Dr. M R Gorbal for his constant encouragement. My cousins Harsha Imrapur and Kanchana Imrapur and their family who provided me with a homely environment to celebrate festivals every year, away from home. My parents-in-law for their constant prayers and well wishes.

I sincerely thank you all and the many I have missed, who have contributed in one way or another over the 4 years it has taken to write this thesis.

Lay Summary

Today's world is increasingly reliant by cloud-based always-online applications for everything from banking, social media to online shopping and food delivery. These applications run on computers within large warehouses, colloquially known as datacenters. The datacenter computers are high-end systems called servers which are required to be highly-reliable and also provide high-performance. For instance, a banking application must be available to customers at any time and provide fast access to accounts, cards, etc.

One key component of the server that affects both reliability and performance is memory, which holds data. Similar to memory in humans, memory in servers is volatile and can sometimes be lost. The memory loss can occur due to failures in several sources like worn out or faulty memory cells, damaged wires that carry data, external factors like temperature or electrical disturbances. Such memory failures can cause applications to crash. This thesis explores techniques to strengthen memory reliability while simultaneously ensuring the improved performance of memory for applications.

To keep memory reliable, whenever data is written to memory, a full data replica is kept as far apart and disjoint in another memory system within the server. This comprehensively allows tolerating memory loss arising from failures in any source. Beneficially for performance, the full data replica can also be read by routing memory requests to the replica that is nearest to the requestor. All this behavior is made possible by specifying a protocol to perform precise actions for each read and write.

Servers in the datacenter are also connected to a common shared memory. This allows applications running on the servers to share data efficiently. For example, the bank application running on server A can send money to the food delivery application running on server B through the shared memory. However, just like memory, the independent servers in the datacenter can fail unexpectedly. Therefore, actions in the protocol for memory reads and writes must not be reliant on other servers being alive. In our example, if the food delivery order on server B fails, the money must not be lost. This thesis proposes to harden the protocol, transforming an unreliable protocol into a fault-tolerant one.

Abstract

Memory is one of the key components that affects reliability and performance of datacenter servers. Memory in today's servers is organized and shared in several ways to provide the most performant and efficient access to data. For example, cache hierarchy in multi-core chips to reduce access latency, non-uniform memory access (NUMA) in multi-socket servers to improve scalability, disaggregation to increase memory capacity. In all these organizations, hardware coherence protocols are used to maintain memory consistency of this shared memory and implicitly move data to the requesting cores.

This thesis aims to provide fault-tolerance against newer models of failure in the organization of memory in datacenter servers. While designing for improved reliability, this thesis explores solutions that can also enhance performance of applications. The solutions build over modern coherence protocols to achieve these properties.

First, we observe that DRAM memory system failure rates have increased, demanding stronger forms of memory reliability. To combat this, the thesis proposes Dvé, a hardware driven replication mechanism where data blocks are replicated across two different memory controllers in a cache-coherent NUMA system. Data blocks are accompanied by a code with strong error detection capabilities so that when an error is detected, correction is performed using the replica. Dvé's organization offers two independent points of access to data which enables: (a) strong error correction that can recover from a range of faults affecting any of the components in the memory and (b) higher performance by providing another nearer point of memory access. Dvé's *coherent replication* keeps the replicas in sync for reliability and also provides coherent access to read replicas during fault-free operation for improved performance. Dvé can flexibly provide these benefits on-demand at runtime.

Next, we observe that the coherence protocol itself requires to be hardened against failures. Memory in datacenter servers is being disaggregated from the compute servers into dedicated memory servers, driven by standards like CXL. CXL specifies the coherence protocol semantics for compute servers to access and cache data from a shared region in the disaggregated memory. However, the CXL specification lacks the requisite level of fault-tolerance necessary to operate at an inter-server scale within the datacenter. Compute servers can fail or be unresponsive in the datacenter and therefore, it is important that the coherence protocol remain available in the presence of such failures.

The thesis proposes Āpta, a CXL-based, shared disaggregated memory system for keeping the cached data consistent without compromising availability in the face of compute server failures. Āpta architects a high-performance fault-tolerant object-granular memory server that significantly improves performance for stateless function-as-a-service (FaaS) datacenter applications.

Keywords: memory, DRAM, reliability, performance, coherence protocols, memory disaggregation, fault-tolerance, function-as-a-service

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. This thesis incorporates and extends work that first appeared in the following papers:

[146] Adarsh Patil, Vijay Nagarajan, Rajeev Balasubramonian, Nicolai Oswald
Dvé: Improving DRAM Reliability and Performance On-Demand via Coherent Replication
Appears in proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA), 2021

[147] Adarsh Patil, Vijay Nagarajan, Nikos Nikoleris, Nicolai Oswald
Āpta: Fault-tolerant object-granular CXL disaggregated memory for accelerating FaaS
Appears in proceedings of 53nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2023

(Adarsh Patil)

Contents

List of Figures

List of Tables

1 Introduction

Memory is a vital component of today's datacenter computing systems and has evolved along with the rest of the system. Memory is actually a complex subsystem which is a combination of various technologies, each with their own unique trade-offs (latency, bandwidth, capacity, cost) as shown in Fig. 1.1. For datacenter servers, the objective is to architect an intelligent combination of these to achieve high performance, efficiently.

To satisfy this goal, system designers have relentlessly pursued innovations in the memory subsystem. These innovations are enabled by continual advancements in manufacturing processes and tighter circuit designs. For instance, increase in memory density is achieved through shrinking DRAM process technology [13], increase in memory bandwidth by way of physically distributing memory across sockets and/or chiplets within a server [187], higher memory capacity by means of attaching memory over newer generations of PCIe with an alternate protocol like CXL [33, 55, 143].

However, memory designs are faced with increased failure rates and newer models of failure [17, 122, 170, 171]. The higher density DRAM has exacerbated various unwanted circuit-level phenomena leading to increased memory errors. Despite several redundancies incorporated to mitigate these errors, at the detriment of performance, memory errors still remain a serious cause of concern [16, 71, 122, 162, 166, 170, 171]. Similarly, using a loosely coupled memory over a high-speed PCIe interconnects introduces partial system failure model [14, 39, 85]. The lack of fault-tolerance to such failures limits the use of this high-performance memory system for modern datacenter applications.

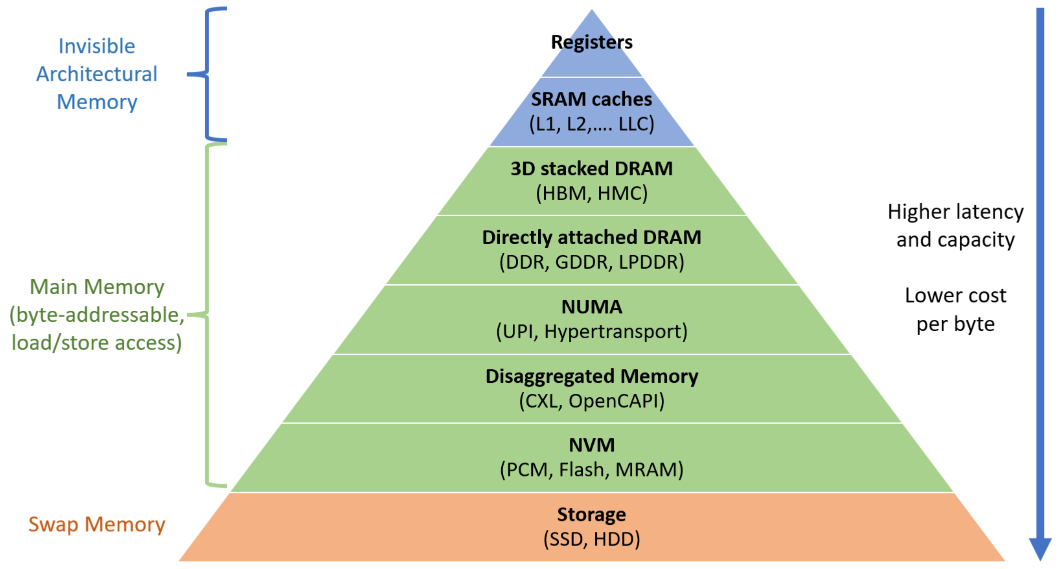
Invisible Architectural Memory
Main Memory (byte-addressable, load/store access)
Swap Memory
Registers
SRAM caches (L1, L2,.... LLC)
3D stacked DRAM (HBM, HMC)
Directly attached DRAM (DDR, GDDR, LPDDR)
NUMA (UPI, Hypertransport)
Disaggregated Memory (CXL, OpenCAPI)
NVM (PCM, Flash, MRAM)
Storage (SSD, HDD)
Higher latency and capacity
Lower cost per byte

Figure 1.1 *The memory pyramid showing the plethora of options available today to design the memory hierarchy for datacenter systems. Each level is a type/class of memory and in brackets, the corresponding technologies used to implement it. The challenge today is in selecting the right mix of technologies to architect a well-balanced memory (capacity/cost/performance/reliability) for datacenter servers.*

Therefore it is vital that in modern memory systems, system designers must incorporate further error mitigation mechanisms to prevent such hardware errors from causing software-visible failures. Care must be taken to ensure that these reliability mechanisms must not stymie performance for applications.

The pervasiveness of coherence protocols: Modern servers combine all these memory technologies and present an abstraction of a single atomic memory. To achieve this illusion of a singular memory that is shared between all the cores, hardware coherence protocols are employed at various levels. As seen in Fig. 1.2, atleast 3 levels of protocols exist in a next generation datacenter rack, namely intra-processor, inter-processor and inter-server coherence. These coherence protocols are designed to enforce memory consistency and provide better cache performance. Fault tolerance is generally not a consideration. However, as memory designs evolve and more closely resemble traditional distributed systems, building reliability into these protocols must be a first order design concern.

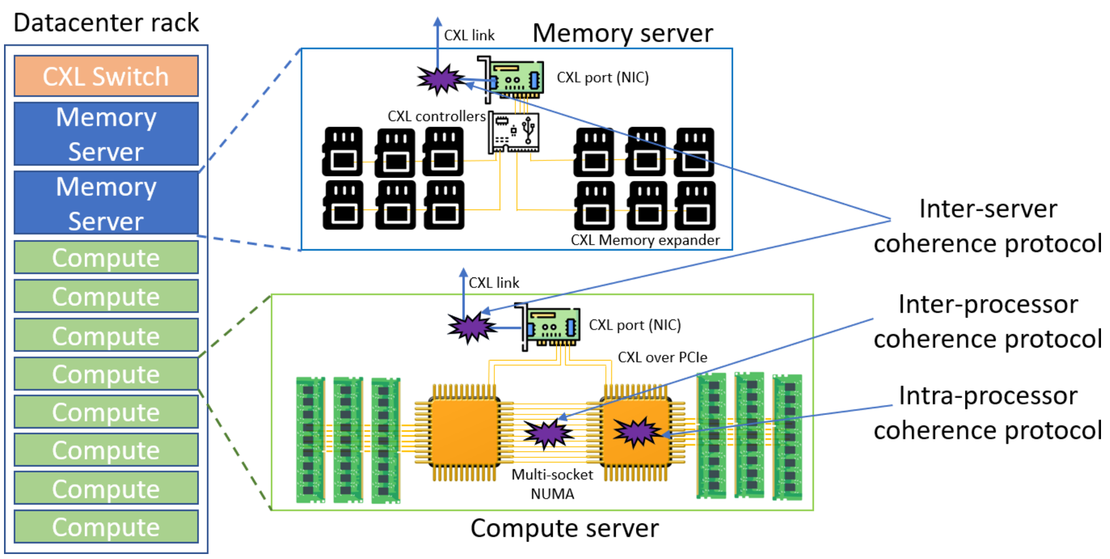

Figure 1.2 *Physical organization of memory in datacenters. The figure shows a datacenter rack/cabinet with 2-socket NUMA-based compute servers, sharing memory from a pooled disaggregated memory server over CXL. A top-of-rack CXL switch connects the compute servers and memory servers within a rack.*

Thesis insight: These ubiquitous coherence protocols can be employed to jointly improve the reliability and performance of the datacenter memory system. We are also cognizant that coherence protocols themselves can be a problem for improved reliability and performance. Therefore, in this thesis, we aim to both reinforce the system and harden the coherence protocols against common modes of failures. Further, to design sound and efficient system solutions, we aim use a holistic approach to system design with an end-to-end understanding of all the components of the system - from the application software to the hardware architecture.

1.1 Problem discussion

We now identify and describe two key reliability problems in modern datacenter memory system.

Insufficient DRAM memory reliability: As technologies continue to shrink, DRAM memory system failure rates have increased, demanding support for stronger forms of reliability [16, 71, 122, 162, 166, 170, 171]. Errors have been

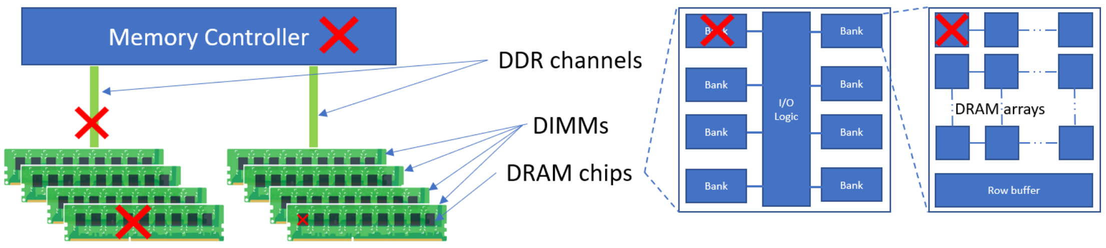

Figure 1.3 *Risk of errors in various DRAM components, denoted by ×. Several field studies have characterized these errors and observed the increasing variety and granularity of DRAM errors. Growing DRAM error rates are a constant cause of concern.*

observed in almost every component of DRAM memory as illustrated in Fig. 1.3. DRAM architects have reacted by incrementally applying an evolution of the same technique used to mitigate DRAM cell failures – forward error correction. Forward error correction scheme pads data with error correcting code (ECC) and distributes the resulting code word on a set of components such that, on a partial failure, data can be recovered from the remaining fault-free data and the padded error correction code. DRAM memory now has this technique built-in at various points in the hierarchy (e.g., cell, rank, chip, channel) to detect and correct errors.

We observe that in the face of increasing errors, to maintain memory reliability, the capacity overhead incurred using the current techniques is rapidly growing (e.g., additional on-die ECC, doubling of ECC bits, row/column sparing). Further, these mitigation techniques are inadequate for the nature of errors seen in field studies (e.g., non-cell chip-internal circuitry errors) and for handling newer failure models (e.g., temperature induced, row-hammer, disturbance faults).

Additionally, there are more problems with such designs (a) typically, error detection/correction imposes a performance overhead, (b) the reliability achieveable is strictly fixed at design time and cannot be increased on-demand for say, critical applications or aged DRAMs. Therefore, there is a need for a new holistic approach to error correction that can correct errors of any granularity.

Non fault-tolerant disaggregated memory: Hardware disaggregated memory is the capability for compute servers to access a physically distributed remote memory with the traditional load/store semantics. The CXL specification [33], proposed by a consortium of hardware vendors, defines a standardized interface to create such an architecture over the PCIe physical interconnect. As illustrated in Fig. 1.4, CXL provides low-latency, high-bandwidth access to remote memory from compute servers, just like in traditional NUMA architectures.

The recent CXL 3.0 specification allows memory regions in the disaggregated memory to be shared between multiple compute servers. The compute server can transparently cache data from the shared disaggregated memory. CXL defines a coherence protocol to keep these caches coherent.

However, because individual compute servers can fail or become unresponsive in the datacenter [14, 39], it is important that the coherence protocol remains available in the presence of such failures: i.e., the protocol should not block indefinitely if any of the servers fail. Alas, while the CXL protocol enforces strong consistency of the data in the caches, it fundamentally blocks in the presence of server failures: if a server caching a shared cache line fails, a write to that shared cache line from any other server could indefinitely block waiting for an acknowledgment from the failed sharer.

This naive application of a traditional multi-processor non fault-tolerant coherence protocol for distributed disaggregated memory leaves CXL systems vulnerable to system crashes. Such availability guarantee is of paramount importance for modern cloud computing applications that have already embraced compute servers failures via a new programming paradigm - Function-as-a-service (FaaS) [4, 60, 126]. Today, all major cloud providers support FaaS deployment, also called serverless computing (Fig. 1.4).

1.2 Solution direction

To reiterate, in this thesis, we seek to develop solutions that enhance memory system reliability while also aiming to improve performance for datacenter applications. We achieve this design with broad and holistic understanding of the multiple components of production cloud datacenters viz.,
$\succ$*the memory subsystem*: from current caches, coherence, DRAM architectures to upcoming technologies like CXL disaggregated memory,

≻*the compute workloads*: from classical shared memory programs to prospective function-as-a-service applications.

More specifically,

(i) to improve DRAM reliability, we explore a new design which replicates data and flexibly trades-off reduced capacity for improved reliability and performance.

(ii) to provide fault-tolerant disaggregated memory, we harden the CXL coherence protocol and specialize the memory to accelerate modern function-as-a-service datacenter applications.

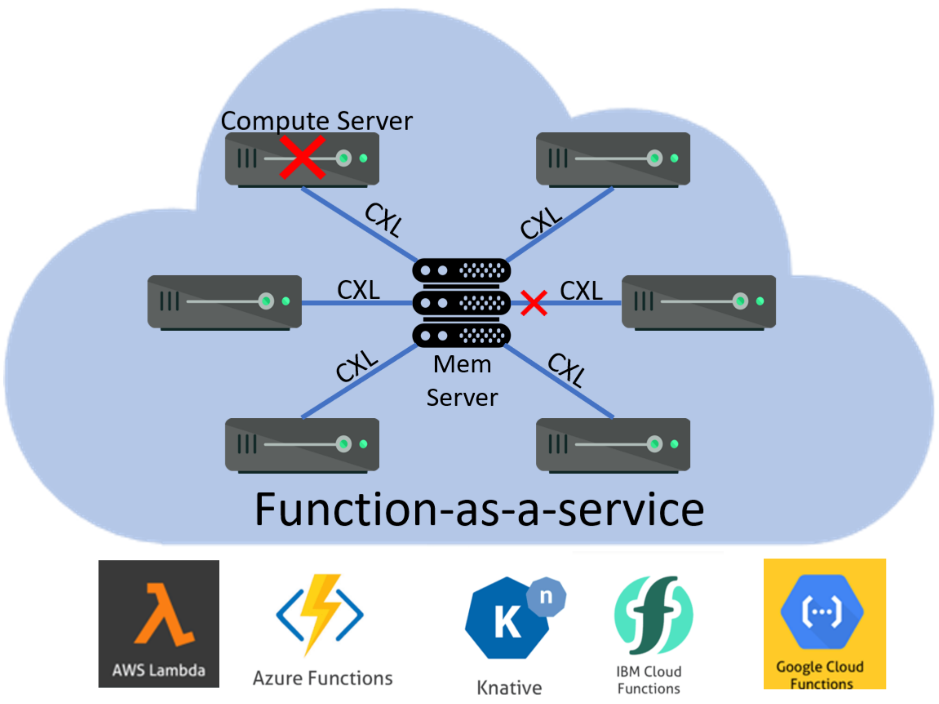

Figure 1.4 *Risk of errors in the disaggregated memory configuration in the datacenter, denoted by* ×*. Because of this risk, function-as-a-service (FaaS) cloud computing paradigm uses applications programmed to be fault-tolerant to such errors. Below: popular cloud providers' FaaS offerings.*

1.3 Our Approach

We have thus far described the problem space and the direction we seek to address the challenge of providing higher reliability memory. We now briefly preview our approach and the mechanisms we employ, for building fault-tolerance in the remainder of this section.

1.3.1 Coherent memory replication

To address the need for improved DRAM reliability, we take inspiration from the two-tier approach that decouples DRAM error correction from detection and explore a novel extrapolation. We propose Dvé, a hardware-driven replication mechanism where data blocks are replicated in 2 different sockets across a cache-coherent NUMA system. Each data block is also accompanied by a code with strong error detection capabilities so that when an error is detected, correction is performed using the replica. Such an organization has the advantage of offering two independent points of access to data which enables:

1. strong error correction that can recover from a range of faults affecting any component in the memory, upto and including the memory controller.
2. higher performance by providing another nearer point of memory access.

Dvé realizes both of these benefits via Coherent Replication, a technique that builds on top of existing cache coherence protocols for not only keeping the replicas in sync for reliability, but also to provide coherent access to the replicas during fault-free operation for performance. Dvé can flexibly provide these benefits on-demand by simply using the provisioned memory capacity which, as reported in recent studies, is often underutilized in today's systems. Thus, Dvé introduces a unique design point that offers higher reliability and performance on-demand for workloads that do not require the entire memory capacity.

1.3.2 Lazy invalidation with coherence-aware scheduling

As datacenter applications increasingly adopt the fault-tolerant Function-as-a-Service (FaaS) model, demand for improved performance has increased. Alas, the performance of FaaS applications is heavily bottlenecked by the remote object store in which FaaS objects are maintained. We identify that the upcoming CXL-based cache-coherent disaggregated memory is a promising technology for maintaining FaaS objects. Our analysis indicates that CXL's low-latency, high-bandwidth access characteristics coupled with compute-side caching of objects, provides significant performance potential over an in-memory RDMA-based object store.

However, CXL lacks the requisite level of fault-tolerance necessary to operate at an inter-server scale within the datacenter. Furthermore, its cache-line granular accesses impose inefficiencies for object-granular data store accesses.

We propose Āpta, a CXL-based object-granular memory interface for maintaining FaaS objects. Āpta's key innovation is a novel fault-tolerant coherence protocol for keeping the cached objects consistent without compromising availability in the face of compute server failures. Our evaluation of Āpta using 6 full FaaS application workflows (totaling 26 functions) indicates that it outperforms a state-of-the-art fault-tolerant object caching protocol on an RDMA-based system by 21–90% and an uncached CXL-based system by 15–42%.

1.4 Summary

To recap, reliability is a key design facet of modern memory systems. Designing memory with higher reliability must not compromise on performance.

In this thesis, we observe two reliability deficiencies in the next generation of datacenter memory systems. Firstly, newer generations of DRAM memory are faced with higher error rates. Existing mitigation techniques are insufficient and are detrimental to performance. Secondly, emerging hardware disaggregated memory has the potential to accelerate new paradigms of datacenter applications. To realize this potential, disaggregated memory systems need to provide fault-tolerant operations without performance penalties.

Our aim is to architect techniques to solve the challenge of improving both reliability and performance of modern datacenter memory. We elaborate on the issues discussed above in the rest of this thesis.

Chapter 2 covers the necessary background about the various organizations of memory in datacenter systems and the coherence protocols employed therein. We also describe the fundamental workings of and examine the fault-tolerance mechanisms in the DRAM memory system and in the function-as-service programming model.

Chapter 3 **Dvé**, delves into how memory replication is employed to guarantee higher DRAM reliability, provide memory performance gains and the ability to avail higher on-demand reliability.

Chapter 4 **Āpta**, describes the design of a fault-tolerant CXL disaggregated memory with an object-granular interface to accelerate FaaS applications in the datacenter.

Chapter 5 concludes with a summary of contributions and key results. We critically review our designs and provide perspectives on future directions.

2 Background

This chapter provides the background necessary to understand the contributions of this thesis. Sec. 2.1 describes the various hardware organizations of memory in the datacenter (NUMA, disaggregated memory). Sec. 2.2 discusses the coherence protocols employed in each of the memory organizations to keep the memory consistent. Sec. 2.3 explains the DRAM memory structure, its operation, the failure models observed and the various mitigation techniques. Finally, Sec. 2.4 details the function-as-a-service cloud computing model, its execution model and its approach to fault-tolerance.

In the rest of this thesis, we use the terms:
CPU or *core* to refer to the logic circuitry that processes instructions and includes components like the ALU, pipelines, registers.
Processor or *chip* to refer to the full integrated circuit containing multiple cores, caches, memory controllers and other architectural controllers that are co-located within a single silicon die. The processor is seated into a socket on the motherboard within a server chassis.

2.1 Memory organization

This section provides a background of the main memory organization in modern servers. Fig. 2.1 illustrates the baseline processor and memory model we use throughout this thesis. It consists of a processor chip with multiple cores which communicate over an underlying on-chip interconnection network. There is an off-chip main memory connected to the processor by a separate, parallel memory interconnects also known as a channels. We elaborate on the logical hierarchy of the various memory components below.

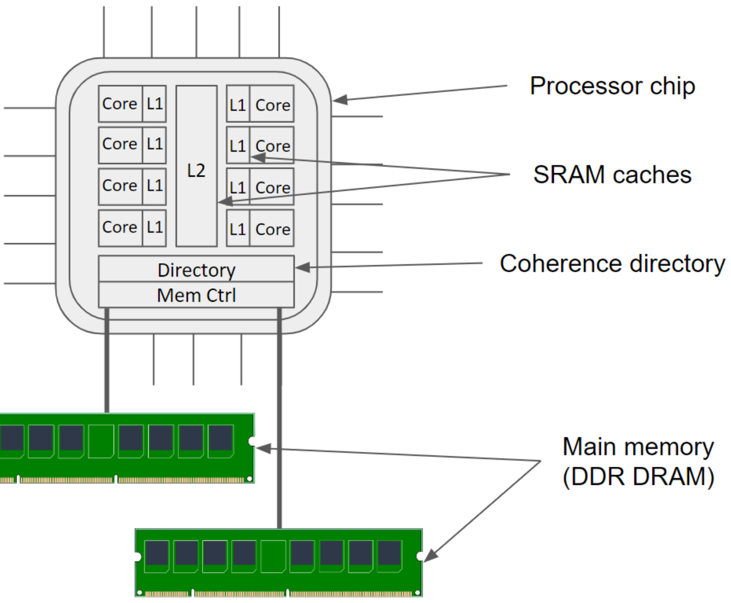

Figure 2.1 *Baseline processor and memory model used throughout this thesis*

SRAM caches are the highest level of the memory hierarchy, closest to the core within the processor. To take advantage of the principle of locality at several stages, multiple levels of caches exist. Some levels of caches are private to each core (e.g., L1 caches), while some may be shared between cores (e.g., last-level cache). Caches closer to the cores have a lower latency than the ones further away. A cache controller, located beside the cache, defines the specific action to be performed for each request from the core. A request may be satisfied at a certain cache level (cache hit), or the request is sent through to subsequent cache level (cache miss). If the data is not available in any cache levels, it is retrieved from the main memory. The data retrieved is stored in the caches for future reuse, thereby exploiting locality. The caches are accessed with the memory physical address and organized in any of the several well-known ways.

Main memory is the byte addressable, physical memory exposed to the core. Main memory is commonly built with Dynamic Random Access Memory (DRAM) technology and is located off the processor chip (off-chip). The total main memory capacity in the system can be a combination of directly attached *local*

DRAM memory (e.g, DDR3/DDR4, HBM) and over the interconnect *remote* DRAM memory (NUMA, disaggregated memory) as shown in Fig. 1.2. A DDR4 local memory can provide an access latency between 30-50ns while NUMA-based and CXL-based remote memories add further 50ns [69] and 100ns [108] interconnect latencies, respectively, to the access time.

Main memory in modern computer systems is normally shared between multiple cores and even multiple processors, as seen in Fig. 1.2. Computer systems provide the ability to share this memory using hardware implementations. This is referred to as *hardware shared memory* or simply **shared memory**. In a shared memory system, each of the cores may read and write data to the same region of memory concurrently (also termed shared address space). A memory consistency model defines the correct and precise shared memory behavior for the reads and writes. In a shared memory system with caches, the data held in the caches can potentially become out-of-date (or *incoherent*) when one of the other core updates the data. A hardware **coherence mechanism** seeks to make the caches of a shared memory system invisible by propagating a core's write to other caches. We explain further details of hardware coherence mechanism in Sec. 2.2.

2.1.1 Non-uniform memory access (NUMA)

To scale-up a shared memory system, multiple processor chips are seated into sockets within a server. Fig. 2.2 shows an example of a 2-socket compute servers in the NUMA organization. The multi-core chips are connected by a high-bandwidth, low-latency, point-to-point, serial interconnect like Intel's QPI/UPI, AMD's Hypertransport/InfinityFabric or similar. The memory in these multi-socket systems is physically distributed i.e., some memory capacity is located adjoining each socket. However, the entire memory capacity is exposed as a single, flat address space to all the cores in the system. Therefore, in this architecture, some main memory accesses are much faster than others as it depends on the location of the data. Accessing data locations mapped to a remote sockets memory experiences a higher latency, as it requires traversing the socket interconnect links, compared to accessing locations on the same socket.

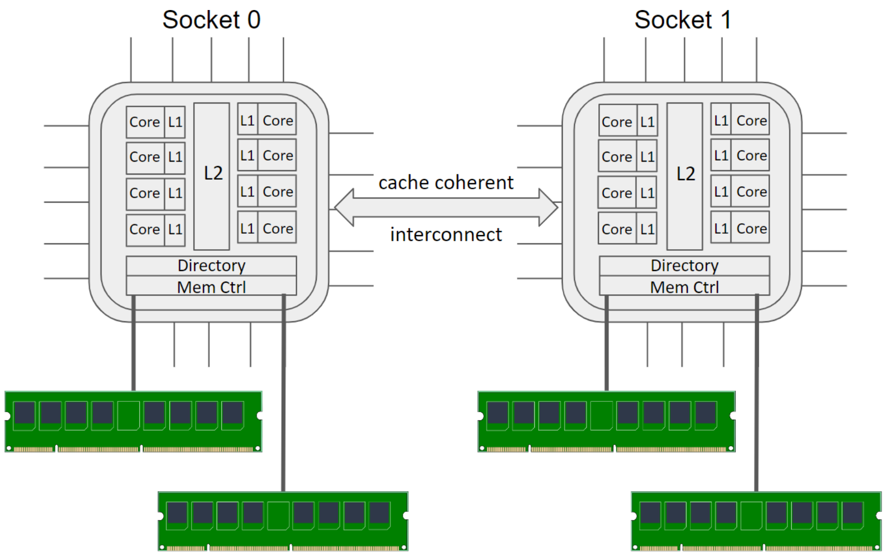

Figure 2.2 *Scale-up system: Two processor system in a NUMA organization*

NUMA-aware software: Modern operating systems (OS) can automatically employ policies to ensure that data accessed by each core is mapped to the socket adjacent, local memory. This best effort mapping techniques can help reduce the NUMA effects for applications running on such architectures. However, these policies cannot completely eliminate NUMA effects in all scenarios. For example, when an applications have a memory working set size larger than the size of local memory, some access requests will inevitably experience higher access latency. Alternatively, applications can also use system calls to explicitly specify allocation policy for each request at a fine-grain.

Cache-coherent NUMA: In NUMA architectures, caches within a processor chip are allowed to cache memory from both local and remote memory regions. An inter-processor hardware cache-coherence protocol is employed to keep these caches consistent. This protocol guarantees that when a memory location is read the most up-to-date value of memory (also known as the most recently written value) is returned to the cores, irrespective of where the data is located. Therefore, in NUMA architectures the placement of data in memory can only influence performance and not the correctness of the code. We discuss more about the NUMA coherence protocols in Sec. 2.2.1.

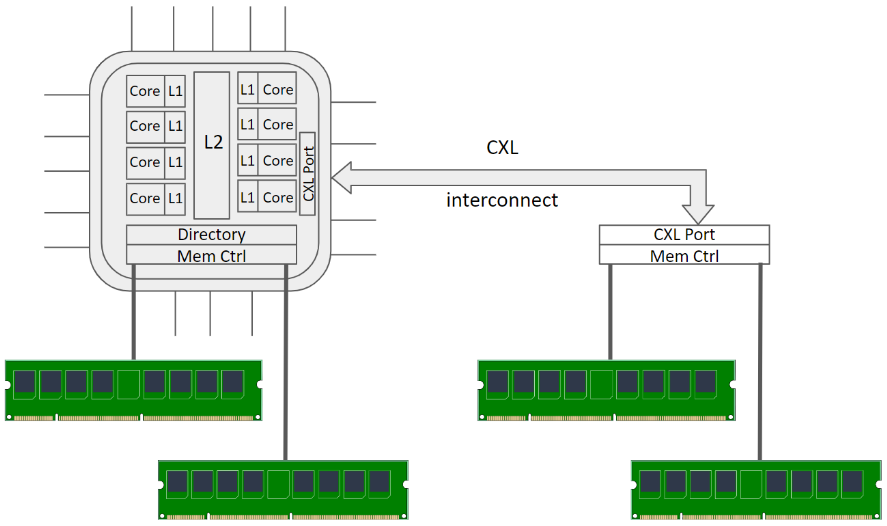

Figure 2.3 *Hardware disaggregated memory system: A processor connected to memory over a CXL interconnect*

2.1.2 Hardware disaggregated memory (DM)

To expand memory capacity and reduce memory over-provisioning / under-utilization, it has been proposed to pool and consolidate some of the main memory into dedicated memory servers. CPUs on compute servers can access this physically disaggregated memory via the regular load/store (ld/st) accesses, similar to remote NUMA memory. The data from memory can also be cached in CPUs local caches and kept coherent via an inter-server coherence protocol. Several competing standards were initially proposed to build such a DM architecture - GenZ[55], OpenCAPI[143], CXL[33], CCIX[24]. Over time, all standards have now coalesced under the CXL umbrella due to their synergistic goals. Fig. 2.3 illustrates such a DM architecture where a processor accesses memory over the CXL interconnect.

≻ **Key benefit:** The DM architecture enables the transition from a fixed memory system into a flexible one with desired capacity, bandwidth, and cost-per-GB based on the workloads requirements.

Compute Express Link (CXL) [33] is an industry-consortium driven, interconnect standard for providing high-bandwidth, low-latency connectivity between processors and memory. CXL runs over next gen PCI-Express (PCIe) 5.0 and 6.0 physical layer (PHY) but with custom link and transaction layers to achieve lower, nanosecond-order latency. Typically, any memory attached over PCIe is mapped into the system as a memory-mapped I/O (MMIO) region, which is uncacheable. CXL aims to provide coherency and memory semantics on top of non-coherent PCIe interconnect. CXL defines 3 protocols - CXL.io, CXL.cache, CXL.mem - for various use cases. These protocols run on specialized hardware controllers on the CPU and CXL memory device.

The CXL.mem protocol enables the above described DM architecture i.e., cache-able load/store accesses from CPUs to pooled physically disaggregated memory. (For the scope of this thesis, we focus only on the CXL.mem protocol.) CXL.mem allows mapping remote memory into the system address space. A last-level cache (LLC) miss to a CXL memory address is translated into requests on a CXL port whose responses bring in the missing cachelines, as shown in Fig. 2.3.

$\succ$ **Key benefit:** CXL's open standard enables the design of high-performance memory systems, specialized for the needs of modern datacenter applications [8, 70, 108, 114, 149].

Shared disaggregated memory: CXL 3.0 specification [34] introduced the concept of coherent sharing of DM - called "shared FAM" (fabric attached memory). This provides the ability for CXL-attached memory to be coherently shared across compute servers using hardware coherency. In other words, a given region of CXL memory can be simultaneously accessed by more than one compute server and with a guarantee that every compute server sees the most up-to-date data at that location, without the need for software managed coordination. This leads a to more efficient and performant way to use memory. For example, in a scenario where many compute servers are accessing the same data set (like in FaaS applications), using this CXL shared DM architecture can provide huge performance improvements. We discuss more about the CXL.mem shared DM coherence protocol in Sec. 2.2.2.

$\succ$ **Key benefit:** Shared, coherent DM allows system designers to build cluster of machines that can be employed to solve compute problems larger than a single compute server through familiar, well-understood shared memory constructs.

2.2 Coherence protocols

As evidenced in NUMA and shared DM architectures, a region of memory, can be shared by multiple processors. This essentially provides communication among the processors through reads and writes to the shared data. The shared data can also be held in caches in the processors memory hierarchy and hence may be replicated in multiple caches. Because the view of memory held by two different processors is through their individual caches, this introduces the problem of keeping the data up to date in these caches. Coherence protocols are algorithms, implemented in hardware, to allow caches to return the latest value of memory for a request, thus making the caches invisible.

In this section, we delve into the design of these coherence protocols. Most protocols in use today descend from the four-state protocol proposed in 1983 by J. R. Goodman [58]. The 4 stable states are commonly called: Modified (M), Exclusive (E), Shared (S) and Invalid (I), with meaning of each state remaining the same as defined in the original work. Coherence protocols also contain a large number of transient coherence states, in addition to these handful stable states [134]. These transient states arise since transitions from one stable state to another is not atomic. At each step in a coherence transition, the coherence controller usually changes the state of the block to a different transient state that reflects that step in the transition.

Coherence invariants: To enforce memory consistency, coherence protocols satisfy two key invariants [134]. The Single-Write-Multiple-Reader (*SWMR*) invariant i.e., for any given memory location, at any given time, there is either a single core that may write to it or some number of cores that may read from it. The Data-Value invariant i.e., a read returns the value of the latest write to that location. Coherence protocols that enforce these invariants make the caches invisible and provide an atomic memory system that guarantees per-memory location linearizability i.e., a write takes effect at some real time between its invocation and response.

To summarize, the *write-invalidate coherence protocols* implemented in today's systems ensure a single writer by invalidating the copies of a cache line in all other caches.

Directory-based coherence: The key to implementing a cache coherence protocol is tracking the sharing state of cache blocks. The status of every cache block of physical memory that is currently cached is kept in one location - the *directory*. In modern systems, the directory is physically distributed, just like memory (different requests go to different memories). In a straightforward manner, the directory is distributed along with the memory i.e., each directory has entries for all the memory addresses behind it. The coherence protocol directs requests to go to different directories based on the memory address. Thus, the coherence protocol always knows where to find the directory information for any block of memory. The place where memory location and directory entry of an address reside is known as the *home node*.

2.2.1 NUMA coherence

Recall that in a cache-coherent NUMA architecture, there are 2 levels of coherence protocols: an intra-processor protocol (responsible for local coherence within a chip) and an inter-processor protocol (responsible for global coherence of the entire system) as shown in Fig. 1.2. Requests that can be satisfied by the intra-processor protocol do not interact with the inter-processor protocol; only when a request cannot be satisfied within the processor chip, the request gets promoted to the inter-processor protocol. The inter-chip coherence traffic is carried over low-latency, high-bandwidth, point-to-point interconnects like Intel QPI/UPI or AMD HyperTransport/InfinityFabric.

To summarize, in a multi-processor (multi-socket) NUMA systems, an inter-processor coherence protocol is layered hierarchically over and above the intra-processor coherence protocol [134]. Such a hierarchical organization enables efficient scaling of the system.

2.2.2 DM coherence - CXL.mem

Recall, the CXL 3.0 specification defines the inter-server protocol for a hardware coherent, shared DM architecture as shown in Fig. 1.2. Specifically, the CXL.mem protocol specifies the interface between the processor and the memory to read and write data to DM. CXL.mem is a memory technology agnostic protocol and can support any memory technology - HBM, DDR, NVM etc. We only provide the background pertinent to enable a coherent shared DM in this thesis. We refer the reader to the full CXL specification [34] for further details.

CXL.mem allows the coherence directory to be located within the memory server / device (referred to as Device coherence engine (Dcoh) in the CXL specification). To support coherent shared DM, the CXL.mem protocol was enhanced in CXL 3.0 to provide new inter-node coherence semantics - called HDM-DB. This included new message types called Back-Invalidation Snoop (BISnp) and Back Invalidation Response (BIRsp) and new dedicated channels S2M BISNP and M2S BIRSP to send these messages. These new requests allow sending invalidations from the directory on the memory server to compute server caches which cache data in shared state and receive invalidation-acknowledgment responses back from the compute server caches.

2.3 DRAM overview

Dynamic random access memory (DRAM), introduced by Robert Dennard at IBM in the late 1960s has been the foundation of main memory as it provides relatively large, fast and cost effective storage capacity. In this section, we describe the DRAM organization, operation and more importantly the DRAM error models and state-of-the-art error mitigation mechanisms.

2.3.1 DRAM fundamentals

DRAM structure: DRAM memory systems are organized hierarchically as shown in Fig. 2.4. DRAMs store each bit of data in a smallest unit called a cell. The cell comprises a storage capacitor and an access transistor. The capacitor encodes the binary data value using the charge level of the capacitor. The transistor is used to access the capacitors charge and read or modify the stored charge.

A DRAM array organizes DRAM cells into a two-dimensional grid of rows and columns (typically 512-1024 cells per dimension). A single DRAM chip contains multiple banks of DRAM arrays which operate in parallel. Banks are further organized into ranks which share command buses but have separate data buses. A chip select signal is used to select a bank to issue a command. Multiple banks in a rank operate in parallel and serve data independently to match the data bus width aka channel. A typical memory system would consist of multiple channels with completely independent DRAM devices, each having separate data and address buses.

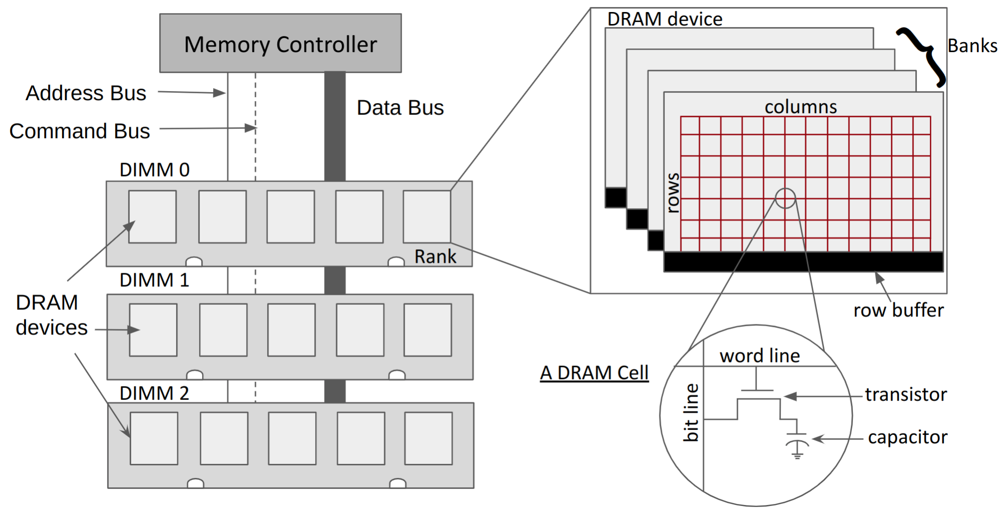

Figure 2.4 *Detailed diagram showing the internal components and organization of DRAM memory*

Memory controller DRAMs are typically a passive device with no logic or controllers to access the stored data. A memory controller, located within the processor chip, interacts with the memory device over a memory bus. The memory controller issues read and write operations to retrieve and store data into the DRAM. The memory controller is also responsible for orchestrating all other DRAM maintenance operations (e.g, initialization, calibration, refresh operations to restore data into the leaky capacitors within the DRAM cells). These operations are invoked through the use of DRAM commands, and DRAM manufacturers consortium (JEDEC standards) clearly specify the timing constraints surrounding their usage.

For high reliability memory systems required in datacenter servers, the memory controller also performs error detection and correction of data retrieved from the DRAM. These reliability mechanisms are discussed in Sec. 2.3.2.

DRAM operation: First, we examine the basic circuitry structure within DRAMs. Starting from the cell, the access transistor's gate is manipulated by a control signal known as the wordline. A common wordline connects the gates of all access transistors in a row of the array. When the wordline is enabled, the storage capacitor is connected to the bitline, so the charge stored in the storage capacitor equalizes with that in the bitline. This process "reads" the data out of the capacitor through the coupled transistors. A row of sense amplifiers (called row buffers) is used to sense this charge and hold it temporarily for purpose of transferring the data over the data bus. Subsequent requests to columns in this activated row are served from the row buffer. Note that each bank has only one global row buffer and hence only one row may be read from any given bank at a time. A column access command transfers a number of columns (consecutive bytes) to/from the row buffer. Since the read from the DRAM array is destructive i.e., the charge in the capacitor of the cells is lost, a precharge operation writes back contents from the sense amplifiers back to the corresponding row.

Now, we describe the commands issued by the memory controller to perform the above DRAM operations. Row activation (ACT) command activates a row within a bank by asserting the row's wordline and allowing the sense amplifiers to read the voltage. A row decoder circuitry decodes the issued row address and activates the transistors of the corresponding row. Column access RD and WR commands read and write to a given column within the open row. Bank precharge (PRE) command precharges the currently open row within a bank by de-asserting the wordline and resetting all bitlines.

Several other timings are specified for DRAM operation by the JECDEC standards like the required delay between issuing two consecutive commands (tRCD, tRAS, tRP etc.) We refer the reader to the JEDEC DRAM specifications [83] for a more detailed overview of standardized DRAM commands and timing.

2.3.2 DRAM errors and mitigation mechanisms

Having understood the components and operation of DRAMs, we now discuss the observed DRAM failure models and the corresponding mitigation mechanisms to handle these failures.

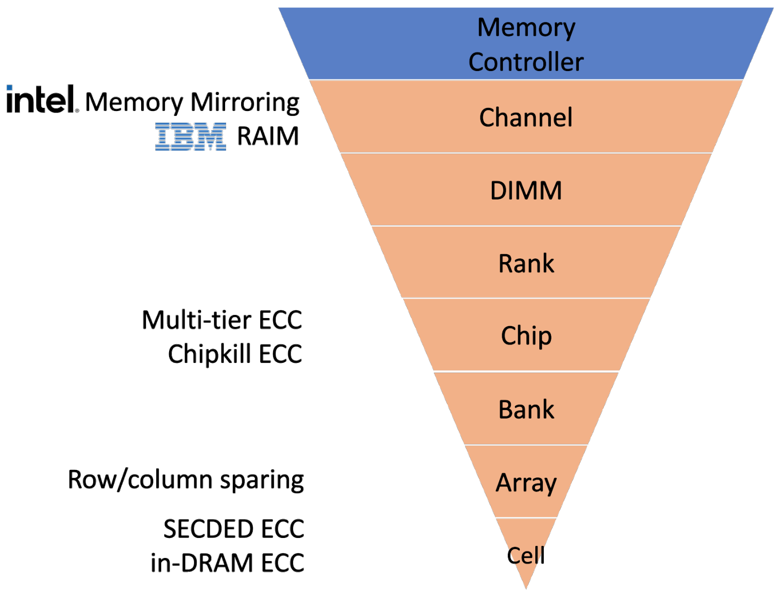

Figure 2.5 *Error mitigation mechanisms in the DRAM hierarchy (on the left). The inverted pyramid showing the increasing data that is lost if errors occur at a level.*

2.3.2.1 DRAM failure models:

DRAMs suffer from a broad range of failure mechanisms that can lead to different types of errors. Fig. 2.5 shows the increasing data that is lost if errors occur at a level. At the very basic level, DRAM devices are complicated by the "dynamic" nature of their storage cells i.e., the capacitors storing the charge in the DRAM are not perfect and leak charge. This naturally results in data loss (data-retention errors) if the charge is not periodically restored i.e., refreshed. Further, DRAM cells can develop faults that are prevalent among semiconductor devices like aging/wearout effects and random external events (e.g., particle strikes, temperature variations). Cells can also experience faults due to manufacturing defects like RowHammer [97], variable time data retention [130, 153], stuck-at faults [132], etc. These errors are expected to become worse with continued process technology scaling leading to increased errors.

2.3.2.2 DRAM error mitigating mechanisms

Several error detection and correction mechanisms have been designed into DRAM systems over the decades. Fig. 2.5, left shows the mitigation mechanisms introduced to correct errors at each level of the DRAM hierarchy.

≻ **In-DRAM error mitigation:** The following mechanisms are designed to operate within DRAM chip and are invisible to other external components.

(i) **On-die ECC:** To correct single-bit error within a DRAM die, DRAM manufacturers employ on-die ECC as a solution. On-die ECC is kept as simple and efficient as possible, since this is implemented within a DRAM die that is not optimized for logic operations. DRAM manufacturers today use extremely basic single-error correcting Hamming codes.

(ii) **Row-column sparing:** DRAM manufacturers provision extra rows and columns within storage arrays in order to provide replacements in the event that some are defective. By employing sparing, DRAM manufacturers are able to tolerate imperfections during manufacturing, thereby improving manufacturing yield at the expense of some area overhead.

≻ **Rank-level ECC:** System designers integrate an ECC mechanisms within the memory controller at a rank level. These ECC mechanisms use data blocks at the granularity of the cache lines to provide stronger error correction. ECC memory modules have additional DRAM chips and wider data buses on the memory channels to store and retrieve the ECC bits from the DRAM. DDR4 ECC memory are provisioned with 12.5% redundancy while newer DDR5 ECC memory increases this to 25% redundancy.

(i) **SECDED ECC:** Single error correcting, double error detecting (SECDED) ECC is the simplest rank-level ECC mechanism that is capable of correcting one error and detecting two errors within each cache line. SECDED ECC stores 8 additional bits in an ECC DRAM, per 64 bits of data using a (72,64) Hamming code.SECDED ECC is employed to address higher error rates than on-die ECC and it can correct errors caused by other components failures in the access path to DRAM (such as burst errors).

(ii) **Chipkill ECC:** Chipkill is a rank level ECC mechanism and a generic term for a solution that guarantees recovery from failure of an entire DRAM chip within a rank. Several approaches exist to implement Chipkill ECC. Current commercial Chipkill schemes utilize single tiered ECC mechanisms to operate with conventional ECC memory modules i.e., the data retrieved in a single access has enough information to both detect and correct a specified number of errors. This is achieved by using interleaving SECDED codes, using Reed-Solomon codes with symbol sizes that align to entire DRAM chips [6, 7]. There have also been proposals to achieve Chipkill with multi-tier ECC codes [87, 89, 118, 135, 177, 191]. In these schemes a first tier performs error detection and a second tier performs error correction. Due to the increase in complexity incurred for such designs, they have not seen wider industry adoption.

≻ **Channel-level ECC:** To provide even higher DRAM reliability and protect against channel level failures (or any errors occurring to any components within a channel), sophisticated newer techniques are being employed at the memory controller. These techniques resemble well-known reliability schemes employed in storage media - Redundant Array of Independent Disks (RAID). In the RAID parlance, these implemented scheme may be loosely associated with various RAID levels, where a channel takes the role of the disk (although the technologies and system considerations used to implement each are different).

(i) **Intel Memory Mirroring [74]:** This technique uses 2 channels operated by a single memory controller in a RAID-1 layout i.e., memory is replicated on DRAMs in two channels that are operated by a single controller. The primary channel is used to read memory. The secondary channel's copy is used as a backup and is read only on the failure of the primary. The memory region that is replicated is fixed at boot time. The OS, particularly Linux, uses the mirrored memory address range for kernel memory allocations only. The OS does not make the mirrored memory available to applications.
Architecturally, each Intel Xeon processor supports up to two mirror ranges, one mirror range per memory controller. Each mirror range size can be defined using 64MB granularity intervals. The OS discovers the mirrored address ranges using a firmware-OS interface. An existing UEFI call GetMemoryMap() returns to the OS all the address ranges presented by the platform. In the returned memory map, the mirrored memory

range is indicated with EFI_MEMORY_MORE_RELIABLE attribute in the EFI_MEMORY_DESCRIPTOR field.

(ii) **IBM RAIM [119]:** This technique uses 5 ganged channels in a RAID-3 layout. The memory system is organized so that five memory channels are used in any given read or write requests. The fifth channel stores a XOR of the corresponding data from the first four channels. After the data from the five channels is formatted into frames, CRC is added to each channel independently. The CRC is used to detect whether there are any errors within a channel. If there is a CRC mismatch when the data is read from the DRAMs, the XOR redundant information will be used to detect and correct errors. However, IBM RAIM forces 256 byte reads and write from memory (4 channels each storing a cache line). This can negatively impact performance due to overfetch.

2.4 FaaS overview

Function as a service (FaaS), also known as *serverless computing*, is a programming model for deploying applications in the cloud. FaaS enables developers to execute code in a cloud-provider's datacenter without server management. Almost all major cloud-providers today offer the FaaS deployment model such as Amazon's AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions. A panoply of applications from popular domains like multimedia processing (video/image), machine learning, scientific computing, web applications, HPC, etc., have been ported to this model for its several benefits [35, 44, 96, 192].

2.4.1 FaaS fundamentals

In FaaS, a monolithic application is decomposed into a set of independent segments called functions. A FaaS orchestration service such as AWS Step Functions [4], Azure Durable Functions [126], or Google Workflows [60], allows developers to define state machine workflows that compose multiple individual functions ("function chains"). State machine workflows allow defining all major

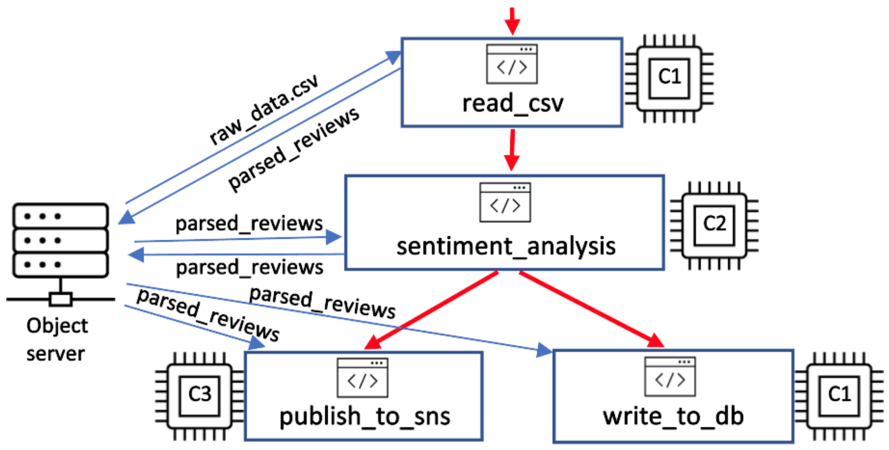

Figure 2.6 *A FaaS application state machine workflow, showing the interaction of the component functions with the object store and an instance of a schedule as executed on a pool of compute servers - C1, C2, C3.*

function communication patterns like producer-consumer, map-reduce, pipeline, broadcast-gather and scatter-gather [44]. On an invocation request / event, the workflow of functions is triggered to execute the application. Scheduling the workflow and managing compute resources for the application execution is the responsibility of the cloud provider.

2.4.1.1 Execution of FaaS applications

An example of a real world use case of the FaaS model [100] is show in Fig. 2.6. In this use case of sentiment analysis / opinion mining, the FaaS application evaluates the sentiment of the customer reviews for the many products and services of a company. We walk through its execution as an illustrative example to explain the working of FaaS applications.

Invocation: The workflow is triggered when the collated raw reviews file (csv) is uploaded to the object store. Upon receiving an invocation trigger, a runtime scheduler like Kubernetes [101] collects the requested inputs and invokes the functions in the workflow, on compute servers, as per the state machine. To invoke a function, its code and its dependent libraries are instantiated and initialized to run inside a microVM or container [182]. In our example, the first function *read_csv* is scheduled on compute server C1, the subsequent pipelined function *sentiment_analysis* is scheduled on compute server C2 and the next set of broadcast functions *publish_to_sns* and *write_to_db* are scheduled on compute servers C3 and C1 respectively.

Execution: Each function running on the compute servers executes in three phases. First, the function executes a basic operation called `get` to read the input from an external server which stores all data as blobs/objects. In our example, the `get` operation in *read_csv* downloads the *raw_data.csv* from the remote S3 server into the compute server memory. Second, the computation phase processes the input data by solely reading data from the local memory. Finally, post-computation, the function executes another operation called `put` which upload the function output to the external object store. In our example, the `put` operation in *read_csv* uploads *parsed_reviews* object from the compute server into a remote S3 server.

Note: This example demonstrates the execution using today's production object stores that use disk storage and general purpose interconnect like ethernet to explain the working of `get` and `put` operations. These operations can also be implemented using other high-performance techniques, depending on the available underlying technology. We examine this in more detail in Chapter 4.

Completion: When the function invocation finishes executing, a Kubernetes agent on the compute server (called kubelet) notifies the runtime scheduler, which then processes the subsequent steps. If the function did not complete execution or exited with a non-zero status, the function is re-scheduled and invoked again with the same input.

2.4.1.2 Features and benefits of the FaaS model

The main features and benefits of the FaaS model are:

Stateless functions: The functions do not retain state across invocations and each invocation is completely independent from each other. Functions employ external object store services such as Amazon S3 [3], Azure Blob Storage [128] or Google Cloud Object Storage [59] to store state that can be used across requests.

On-demand execution: In contrast to other computing models, a FaaS function instance is created only when the function is called and is terminated after processing a request. Some providers also allow developers to provision pre-initialized function instances to reduce delays in processing requests [181]. However there is no guarantee that requests would be directed to the same instance of the functions and therefore any data written to server local storage would be lost.

Automatic Scaling: FaaS functions are inherently scalable as additional instances can be created to respond to increased invocation load. Developers don't have to worry about creating contingencies or provisioning for high traffic or heavy use. The FaaS provider directly handles all of the scaling concerns.

Efficiency for the cloud provider: Cloud providers are able to co-locate thousands of independent function instances on a single physical server, thus achieving high server utilization. A high degree of co-location is possible because most functions are invoked relatively infrequently and execute for a very short amount of time. The cloud provider dynamically schedules a function invocation on a fleet of compute server with great flexibility. This is achieved by employing frameworks like Kubernetes [101] that correctly and efficiently schedule functions on any available compute servers.

Pay for what you use: Developers pay only for the resources used during the function execution (in time units of hundreds of milliseconds) and not for any idle time. This fine-grained billing model reduces costs to deploy to the cloud and avoids any upfront cost of renting large units such as servers or VMs.

Separation of concerns: Developers focus on writing efficient, performant and correct function code with high level abstraction API of underlying infrastructure. The cloud provider is responsible for the operational concerns such as provisioning, configuring, monitoring and managing the compute servers, memory, network, storage.

2.4.2 Fault tolerance in FaaS

Fault-tolerance is a key tenet of the FaaS model. The use of stateless functions already makes the functions resilient to failures of compute servers. FaaS providers also provide error handling so that applications don't just silently crash without a way to react to failures. If a functions fails partway during execution, the *automatic retry* mechanism is initiated by the provider to re-execute the same function [11, 32, 52]. This retry model simplifies the recovery mechanism and can handle any kind of failures encountered - software crashes, intermittent hardware errors, transient network glitches, permanent compute server failures etc. To avoid side effects with the retry mechanism, FaaS providers ask developers to write idempotent functions i.e., the functions give the same output even when called multiple times with the same input.

Semantically speaking, the retry mechanism ensures that functions are executed at-least once. The idempotence property logically ensures at-most once execution. Therefore, combining retry-based model with idempotent functions guarantees exactly-once execution. Several frameworks and mechanisms exist to make functions idempotent or to directly ensure functions are executed exactly-once [111, 168, 194].

In short, the FaaS model is designed to be resilient to all common models of failures in the datacenter. It handles all failures comprehensively by simply retrying the execution of the idempotent functions.

2.5 Summary

In summary, memory in modern datacenter systems is shared between multiple processor chips. The shared memory itself is also physically distributed as seen in NUMA and DM organizations. Hardware coherence protocols are employed at each level (intra-processor, inter-processor, inter-server) to keep the caches up-to-date and ensure coherent access to data in memory from any processor.

The main main memory itself is composed of volatile DRAMs. These are increasingly vulnerable to failures at any level in the memory subsystem. Current mechanisms have been incrementally protecting against failures rising up in the memory hierarchy and thereby protecting larger granularity of failures. Each of these mechanisms is adding additional capacity overheads and negatively impacting performance.

Datacenter providers and application developers have widely adopted the fault-tolerant FaaS cloud deployment to make applications scalable and resilient. FaaS applications are compositions of stateless functions. This model inevitably forces a split of the state which is maintained in a separate remote object store. The functions are orchestrated and scheduled dynamically by the cloud provider on a fleet of compute servers. The functions when executing on the compute servers read/write the state from/to the object store.

3

Dvé: Improving DRAM Reliability and Performance On-Demand via Coherent Replication

In this chapter, we present Dvé - a unique design point in the DRAM memory reliability design space. It provides 3 key benefits: (i) guarantees improved DRAM reliability (ii) provides memory performance gains (iii) the ability to avail higher on-demand reliability. We now delve into how Dvé achieves each of these in detail.

3.1 Overview

For servers or systems with high-value data like in finance, automotive, and healthcare, system reliability is of utmost importance. Improving memory reliability has been key to improving overall system reliability. Field studies of memory in datacenters [122, 162, 166] and supercomputers [16, 71, 170, 171, 172] have reported patterns of larger granularity memory errors/failures and unexpected DRAM failure modes [97], corroborating the need for improved memory reliability.

There have been several techniques and proposals for improving the fault tolerance of DRAM memory. These works have largely focused on incrementally increasing the efficiency and scope of error control mechanisms in memory subsystems. DRAM memory schemes initially only targeted cell failures and progressively evolved to handle chip, DIMM, and channel failures. Primarily, these schemes pad data with error correcting code (ECC) and distribute the resulting codeword on a set of components such that, on a partial failure, data can be recovered from the remaining fault-free data and the padded error correction code.

One notable step in this progression is the recent body of work that has advocated the decoupling of error detection from correction by breaking down DRAM fault tolerance into two tiers [87, 89, 118, 135, 177, 191]. These works add a check code per codeword for detecting errors in the first stage and an error correction mechanism at a larger granularity in the second stage. Doing so allows for deploying more powerful ECC codes to recover from a larger class of errors. This decoupling also allows storing ECC bits elsewhere in memory and thus does not impose restrictions on the configuration and operation of DRAM DIMMs.

In this work, we take inspiration from the two-tier approach and explore a novel extrapolation. We propose Dvé[1], a hardware-driven replication mechanism with the following important features.

1. Dvé leverages ECC codewords and other existing mechanisms for error detection but provides recovery from a detected error by reading from a replica of the data, instead of reconstructing data using the ECC bits.

2. It significantly improves memory reliability by keeping replicas as far apart and disjoint as possible – replicating data across 2 different sockets on the same system, thereby tolerating errors anywhere in the entire memory path (controllers, channels, DIMMs, and DRAM chips).

3. Dvé introduces *Coherent Replication*, a technique that builds on top of existing cache coherence protocols for not only keeping the data and replica in sync, but also providing coherent access to the replicas during fault-free operation. In doing so, Dvé alleviates some of the NUMA latency overheads as data can be accessed at the nearest replica memory. It also provides improved memory access bandwidth by providing two endpoints to access data.

[1] Dvé (Sanskrit) translates to *the two*, referring here to the dual benefits of replication.

4. Dvé can be employed on-demand by taking advantage of the memory that is often underutilized in large installations ([82, 113, 145, 151]), thus allowing flexibility between capacity and reliability.

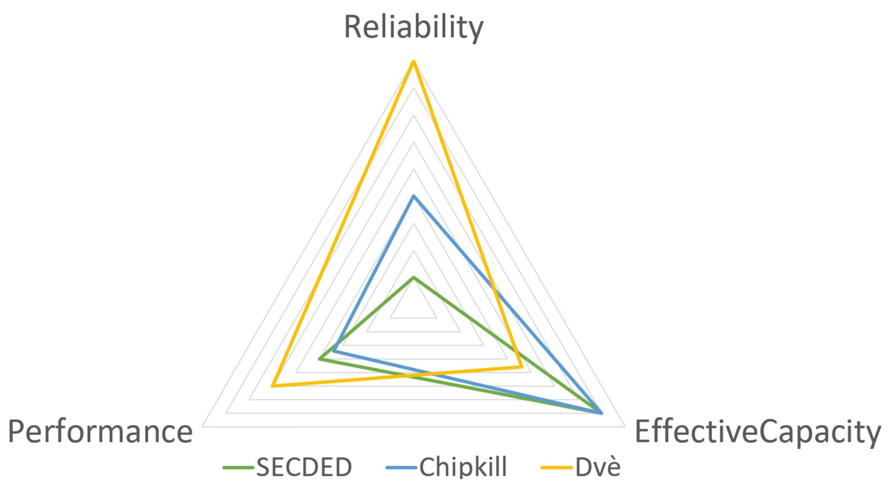

Figure 3.1 *Comparison of DRAM reliability designs on the 3 goodness metrics.*

A New Tradeoff. Fig. 3.1 compares DRAM RAS mechanisms: SEC-DED (bit level error protection), Chipkill (chip level error protection), and Dvé across the goodness metrics of reliability, performance and effective capacity (inverse of capacity overheads).

Dvé achieves higher reliability (at least $4\times$ lower uncorrected error rate than Chipkill) as its design is more robust to failures and can detect/correct a larger granularity of errors. The only Achilles heel for Dvé is in the case of simultaneous failure in exactly the same location on a pair of completely independent replicated memory components, the occurrence of which is lower in probability than 2 DRAM devices failing in the same memory rank as in the case of Chipkill. Thus, Dvé provides stronger protection against memory errors.

Typically, error detection/correction imposes a performance overhead. Many manufacturers concede that Chipkill ECC DRAM will be roughly 2-3% slower than non-ECC DRAM [160]. Although Dvé still requires error detection, by providing two independent points of access to the data, it provides performance improvement in a multi-socket NUMA organization.

Dvé uses simple data replication with higher capacity overheads (lowering effective capacity to 43.75% compared to 85% for Chipkill). While the capacity overheads for Chipkill are strictly fixed at design time, Dvé overheads are applicable only when employed on-demand at runtime (for example, when memory is underutilized).

Contributions. We introduce Dvé, a hardware-driven replication mechanism which provides the dual benefit of improved memory reliability and performance. Specifically:

1. We explore a unique design point which trades off reduced memory capacity for higher reliability and performance by replicating data blocks across two sockets of a cache-coherent NUMA system
2. We analytically quantify Dvé's reliability benefits and show that it provides lower uncorrectable and undetectable error rate over Chipkill ECC and thus provides higher memory reliability. Similarly, Dvé in conjunction with Chipkill ECC provides 2 orders of magnitude higher reliability over IBM RAIM [119]. Further, Dvé's thermal risk aware mapping lowers DUE by at least 11% over Intel memory mirroring [74].
3. We propose Coherent Replication, a technique that builds on top of existing protocols to not only maintain the replicas in sync (required for reliability), but also provide coherent access to both of the replicas during common-case fault-free operation (for performance).
4. To allow for flexibility between capacity and reliability, we propose a hardware-software co-design approach to enable/disable replication when desired at runtime.
5. Our experiments indicate that Dvé provides performance improvements of between 5%-117% across 20 workloads over a dual-socket NUMA system, and between 3%-107% over an improved (hypothetical) version of Intel's memory mirroring scheme.

3.2 Motivation

We first motivate the need for improved DRAM reliability to combat: (a) projected increase in errors caused by DRAM design trends; and (b) DRAM failures caused by *any* part of the DRAM subsystem. Secondly, we motivate the need (and opportunity) for providing improved memory reliability *on demand*. Finally, we summarize by identifying the limitations of existing approaches.

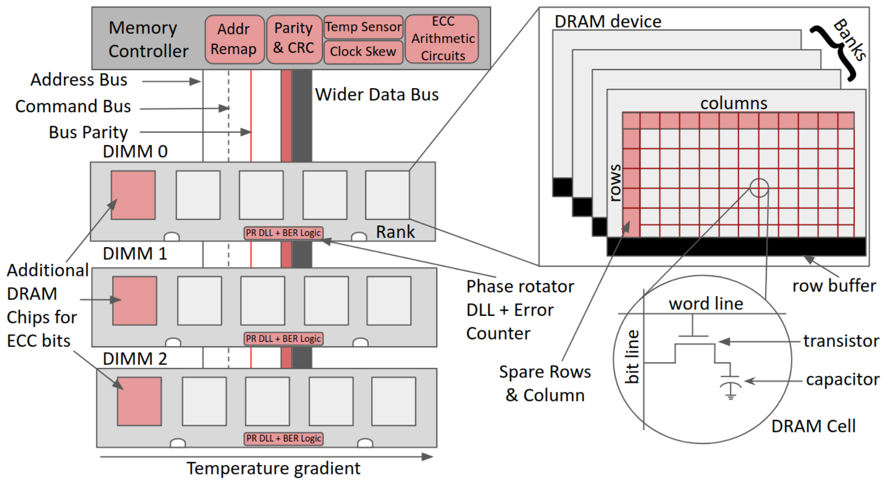

Figure 3.2 *Anatomy of RAS features in memory*

3.2.1 Growing DRAM error rates

The memory subsystem has several modules built-in at various points in the hierarchy (e.g., cell, rank, bank, chip, memory controller) to improve reliability of the DRAM as highlighted in red in Fig. 3.2. We analyze these current DRAM specs/chips and state-of-the-art error protection mechanisms and observe that these would be inadequate for handling the nature of faults/errors seen in field studies.

Cell errors and their increasingly costly mitigation. DDR5 DRAM chips are expected to have 4 times the memory capacity per DRAM chip, in line with the trend of miniaturization and higher density of DRAM. To combat the increase in cell fault rates due to smaller cell geometry, increased variability of manufacturing, and additional refresh pressure, DDR5 includes simple in-DRAM (on-die) ECC [124]. DDR5 DIMMs have also doubled the number of error correcting bits compared to DDR4 DIMMs, i.e., from 8-bit to 16-bit ECC for 64-bit data (25%) [124]. Several mechanisms already propose row/column sparing and selective replication [25], [137] to tolerate higher number of faulty cells caused during manufacturing. The capacity overheads (provisioned invisible redundant capacity) required to maintain reliability is growing.

Non-cell errors are becoming important. Studies have shown that unpredictable large multi-bit DRAM failures can occur due to faults in the chip-internal circuitry [170] that can affect multiple banks, rows, etc. within a DRAM chip. To handle such DRAM chip errors, several variants of Chipkill [2] solutions have been developed. Further studies have observed shared board-level circuitry failures that cause cascading errors, rendering multiple DRAM chips (that share circuitry within a DIMM) erroneous or inaccessible [86]. To cope with such failures Bamboo ECC [94] and Virtualized ECC [191] were proposed to handle multi-pin and multi-chip failures. In addition, studies have shown that faults outside the DIMM such as faults in memory controller logic that interacts with the external DRAM subsystem, errors in the channel, or electrical disturbances also affect the reliability of DRAM memory [95, 122, 166]. These studies suggest that a wide variety of memory failures can occur and existing mechanisms are insufficient to handle these errors because they co-locate data and correction codes on the same channel.

To reduce channel errors, DDR5 memory uses a host of bus reliability mechanisms like command/address parity checks, bus CRC, gear down mode. DDR5 chips are to feature delay-locked loop and forward feedback equalization to handle channel errors that occur because of the higher DDR frequencies [171]. These bus error checks only detect errors and perform transaction retry; they cannot tolerate hard channel errors.

Stronger ECC codes with longer codewords were introduced to detect and correct channel errors in [87], [95]. Increasing the codeword length is also problematic, as the decoder complexity increases more than linearly with the codeword length [20]. Further, sophisticated techniques use 2 channels in a RAID-1 layout (Intel Memory Mirroring [74]) or 5 “ganged” channels in RAID-3 layout (IBM RAIM [119]) to tolerate complete channel failures. However, each of these techniques limits reliability and performance benefits. RAIM’s ganged channel mode forces 256 byte memory reads and writes which negatively impacts performance [196] and leaves it susceptible to any errors in the single RAIM controller. Although Intel’s mirroring scheme replicates memory between

[2]Chipkill is a generic term for a solution that guarantees recovery from failure of an entire DRAM chip.

channels within a controller, the secondary channel's copy is used as a backup and is read only on the failure of the primary – thus, providing no performance benefits despite the existence of data replicas. Further, Intel's approach localizes replicas to a single socket and a single memory controller leaving it susceptible to any faults in the controller or its subsystem. Additionally, the memory that is replicated is fixed at boot time and limited to the OS kernel memory.

DRAM reliability is also impacted by external factors like temperature, requiring sufficient timing slack margins while operating DRAMs [104] or throttling of requests to avoid thermal emergencies [112]. DRAM disturbance faults or row-hammer faults [97] demonstrate that new and unexpected multi-bit failures may occur while in operation. Mitigations include more frequent memory refresh for frequently accessed rows which could cause performance degradation.

3.2.2 Need for on-demand memory reliability

The mitigation techniques thus far have fixed area and logic overheads (at design time) for providing memory reliability. We observe the possibility of using idle memory capacity to opportunistically improve memory reliability. To accomplish this, we motivate the need for such a reliability service to be flexible and allocatable on-demand.

Large scale memory underutilization is prevalent. Several studies point to the phenomenon of memory underutilization in HPC systems [82], [145] and in cloud datacenters (e.g., Alibaba-[29], [113], Google-[151]). These works report that at least 50% of the memory is idle 90% of the time. There exists a large gap between a node's maximum/worst-case memory utilization and the common-case memory utilization; i.e., a few workloads have high utilization while most other workloads have significantly lower utilization. However, memory resources are often over-provisioned due to peak estimation. Datacenter operators procure systems with a view to keeping the machines homogeneous with respect to the workloads or to improve the system's capability to solve large problem sizes, which is an important figure of merit in HPC systems [145].

The bulk of applications, that are not memory capacity intensive, would benefit from increased memory reliability and improved memory access latency[145]. For example, capacity-agnostic long running HPC applications would greatly benefit from increased fault tolerance to memory errors; stateless cloud applications can provide high availability selectively for just the data regions of their memory. Providing this flexibility between capacity and reliability allows deploying large numbers of commodity DRAMs or lower reliability DRAMs for high capacity while still being able to turn on/off higher reliability on demand.

A central observation in our work is that servers under-utilize their memory capacity, and applications can exploit this idle memory to boost performance and reliability. Industry products like Intel Memory Mirroring [74], that relinquish half their memory capacity for high reliability, confirm that this observation is valid, but has only been partially exploited.

Error Rates increase as DRAMs age. Another need for on-demand reliability is to combat the higher error rates observed as DRAMs age and suffer from wear-out faults[47]. This is due to degradation of retention time and increased sensing delays. Memory systems today do not allow for flexibly boosting reliability, requiring periodic memory replacement.

Summary. State-of-the-art DRAM error protection mechanisms suffer from the following limitations.

1. Existing mechanisms jeopardize the correction capability by putting correction mechanisms in the same “basket” as data. Given that failures can occur at any level in the memory subsystem, current mechanisms are therefore vulnerable to failures beyond a channel, including errors in the memory controller.

2. They trade off reduced error detection capability for some amount of correction capability which limits their effectiveness to detect more errors (e.g., designing for Double Symbol Correction before Triple Symbol Detection).

3. Existing mechanisms typically impact performance negatively. Using Chipkill ECC DRAM reduces performance by 2-3% [160] over non-ECC DRAMs. Even contemporary reliability techniques like Virtualized ECC [191] and Bamboo ECC [94] reduce performance further by 3-9% and

2-10% respectively. Further, alleviating temperature induced effects and row hammer mitigations tend to hurt performance.

4. They lack flexibility to provide memory reliability on demand by adapting to workloads requirements.

3.3 Dvé

3.3.1 Design

The variety and the granularity of DRAM errors are increasing. Correcting all of these errors demands a robust mechanism. We argue for a broadsword approach to error correction that is decoupled from error detection, and can correct errors of *any* granularity. We advocate for changing the perspective of DRAM protection from an incremental, short-sighted view to a holistic approach leveraging the time-tested end-to-end argument. We argue for protecting memory at the highest end point of memory (i.e., at the memory controller level), thereby subsuming all other types of errors.

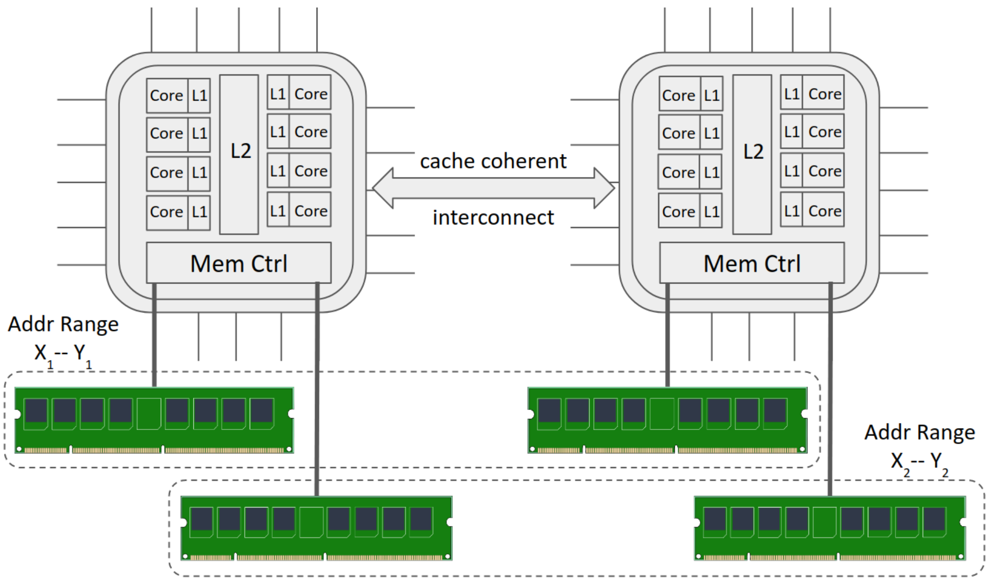

Figure 3.3 *Dvé replication schematic. Data is replicated across DRAM memory on two-sockets of a cache-coherent NUMA system i.e., data replicas are kept as far apart and disjoint as possible, within a server.*

Our solution, Dvé is a hardware-driven replication scheme for achieving not only reliability but also performance. Dvé performs memory replication on two memory controllers located on different cache-coherent NUMA nodes (as shown in Fig. 3.3); when a dirty block is written back to its home memory node, it is also written to its replica. Using a different "basket" for recovery allows us to recover from a wide variety and granularity of failures. Indeed, Dvé can tolerate large multi-bit errors due to memory controller logic failure, bus failures as well as any failures in shared components in the hierarchy.

Because Dvé relies on a replica for correction it needs to store only error detection codes. Therefore, it requires only error detection circuitry that is simpler to build as it involves computation of just the error-locator polynomial (error correction also involves computing the extra error-evaluator polynomial for symbol based codes [19]). The extra code space available as a result of forgoing the correction code can be used to store stronger detection codes for detecting larger number and/or larger granularity of errors. Along with ECC based detection, Dvé can rely on any new and/or existing fault detection techniques, such as CRC or parity present in the DDR4 spec [158], and additional hardware and firmware diagnostic capabilities like temperature sensors, clock skew detection to mark failed components (Fig. 3.2).

Dvé's replication proves advantageous in several other scenarios as well. Mapping replicated data onto DIMMs with different thermal properties – e.g., data on a hot DIMM/chip replicated on a relatively cooler DIMM/chip on the other socket – ensures reduced temperature induced failures. Row hammer errors can be mitigated by load balancing requests between the independent replicas. Sec. 3.3.2 quantifies the reliability benefits of Dvé using failure rates from field studies.

While performance penalties are a problem for existing schemes, in the case of Dvé, the presence of the replica in another NUMA node provides an opportunity to boost performance. Note that in order to ensure strict recovery semantics, the data and its replica needs to be kept consistent at all times. Happily, a replica that is kept strongly consistent allows for the replica to be accessed by reads even during fault-free operation. In other words, it allows for a read request to be potentially serviced from the nearest replica to mitigate some of the NUMA latency overheads and also improve memory bandwidth. In Dvé, we realize this via Coherent Replication (Sec. 3.3.5), a technique that extends existing coherence protocols to keep the data and the replica consistent, while providing coherent access to both the data and the replica during fault-free operation.

In Dvé, each physical address is mapped to a replica physical address. This can either be a fixed function mapping or a flexible table-based mapping. A flexible mapping allows for providing memory reliability on demand and requires the OS to map each allocated physical page to a replica physical page. Using the OS memory allocator's understanding of the system's memory topology, replica page pairs are made such that they exist on memory adjoining different sockets. A single system-wide OS managed *replica map table* (RMT) maps a physical page to its corresponding replica page. If an entry does not exist in the RMT, Dvé seamlessly falls back to using a single copy. On the other hand, a fixed function mapping[3] benefits from fast translation to replica address and works well if the entire memory space is being replicated en masse.

Unless stated otherwise, we assume that all memory is replicated using a fixed function mapping and that there is one replica for every data item (i.e., two copies). The core workings of Dvé are unchanged even with a table based mapping, which would require an additional lookup into the RMT to locate the replica address. We discuss the details of such a flexible mapping system in Sec. 3.4.1.

3.3.2 Quantifying the reliability of Dvé

Dvé's robust design can recover from a large breadth of memory related errors that can be detected. This is because Dvé can simply adopt differing bits from the replica, re-compute the code, and confirm it matches. This provides

[3] a fixed mapping is a static direct-mapped function of the form $\mathtt{f} : \mathtt{p}(\mathtt{S}, \mathtt{Ro}, \mathtt{Ra}, \mathtt{Ba}, \mathtt{Co}, \mathtt{Ch}) \rightarrow \mathtt{p_r}(\mathtt{S'}, \mathtt{Ro'}, \mathtt{Ra'}, \mathtt{Ba'}, \mathtt{Co'}, \mathtt{Ch'}),\ \forall \mathtt{p}, \mathtt{p_r} \in \mathtt{P}$ where $\mathtt{p}, \mathtt{p_r}$ are a physical addresses in P and $\mathtt{S}, \mathtt{Ro}, \mathtt{Ra}, \mathtt{Ba}, \mathtt{Co}, \mathtt{Ch}$ correspond to socket number, row, rank, bank, column, channel respectively. An example for such a function which we use in this work, is given by $\mathtt{f(p)} = \frac{\mathtt{p}}{\mathtt{L}} + 1 - (2 * \mathtt{S})$ where $\mathtt{L}$ is page size. The function considers consecutive physical pages interleaved between sockets and maps to a replica page on the other socket but retains the same DRAM internal mapping.

Table 3.1 *DUE and SDC rates (per billion hours of operation) and improvement. † symbol shows temperature scaled FIT rate*

Scheme	DUE		SDC	
	Rate (lower is better)	Impr.	Rate (lower is better)	Impr.
Chipkill	10^{-2}	–	3.1×10^{-10}	–
Dvé+DSD	2.5×10^{-3}	4×	6.3×10^{-10}	0.49×
Dvé+TSD	2.5×10^{-3}	4×	2.5×10^{-16}	~10^6×
IBM RAIM	1.5×10^{-14}	–	4.0×10^{-10}	–
Dvé+Chipkill	8.7×10^{-17}	172×	6.3×10^{-10}	0.63×
Chipkill†	2.2×10^{-2}	–	1.0×10^{-9}	–
Intel+TSD†	5.9×10^{-3}	3.72×	1.1×10^{-15}	~10^6×
Dvé+TSD†	5.3×10^{-3}	4.15×	1.1×10^{-15}	~10^6×

asymptotically better reliability than any ECC based correction scheme. We now quantify the reliability improvements.

DRAM reliability mechanisms use Forward Error Correction (like ECC) which add redundant information so that data can be recovered when errors are encountered. Block codes that work on fixed-size blocks or "symbols" are used to allow encoding/decoding in polynomial time. Various classical block codes such as Hamming codes, Reed-Solomon codes, BCH codes apply the algebraic properties of Finite (Galois) Field Arithmetic to correct and detect errors in DRAM. These error control systems can have one of the following outcomes: (a) corrected error (CE), (b) detected but uncorrected (DUE) error, or (c) suffer Silent Data Corruptions (SDC).

Comparative Case Studies: For this, we analytically model and quantify reliability improvements of Dvé using DUE, SDC rates with a uniform DRAM device FIT rate of 66.1. This rate was reported in the empirical field study of DRAM failure is the Jaguar HPC cluster at Oak Ridge National Laboratories [170] and is widely used is DRAM system failure modeling. For a fair comparison, although Dvé can use *any* detection code, we equip Dvé with a similar detection capability as the scheme being compared against.

The analytical modeling computes overall system reliability, given the reliability of the individual DRAM device components, and the configuration that makes up the whole memory system (also known as structural properties). This reliabilitywise configurations of the devices can be modeled in several ways, depending on the arrangement i.e., series, parallel, combined complex configuration, k-out-n parallel etc. [106]. In this modeling the DRAM devices are assumed to be statistically independent. This DRAM system modeling and failure rate calculation is similar to prior work [174].

3.3.2.1 Comparison to Chipkill ECC

Consider a system with 32 single rank ECC DIMMs, each DIMM containing 9 DRAM chips. The baseline Chipkill ECC can tolerate one failed chip per rank[4]. For Dvé the 9th chip is modeled in 2 ways:
(i) equipped with detection code similar to the baseline (DSD)
(ii) equipped with stronger detection code (TSD)[5]; using the extra capacity obtained by relinquishing the correction code present in the baseline

DUE rate: A Chipkill system fails to correct an error if 2 chips fail simultaneously within a single DIMM[6] inside a scrub interval which is given by $[9 \times 66.1 \times 8 \times 66.1 \times 10^{-9}] \times 32$ ($\approx 10^{-2}$) in every billion hours of operation.

Each model imposes certain constraints on the chip failures that are uncorrectable; more constraints lead to a lower uncorrectable rate. In Dvé, the system fails to correct if 2 corresponding chips *in the same position* on two DIMMs in the corresponding rank fail together inside a scrub interval. This is because in Dvé the original data can still be recovered, even if both the replicas independently suffer a DUE, by using a combination of both the replica symbols. This process of error decoding and recovery is explained in Sec. 3.3.2.4. The DUE rate of this is given by $[9 \times 66.1 \times 1 \times 66.1 \times 10^{-9}] \times 32 \times 2$ ($\approx 2.5 \times 10^{-3}$) per billion hours of operation.

Thus, Dvé provides $4\times$ lower DUE rate than a Chipkill system. It is worth noting that this number is irrespective of the detection code and is only a factor of the number of replicas.

[4]Assuming 8-bit symbol based RS(18,16,8) code with SSC-DSD (Single Symbol Correct-Double Symbol Detect), organized as in Virtualized ECC [191]

[5]Triple Symbol Detect (TSD) provided using 16-bit Reed-Solomon code as in Multi-ECC [87]

[6]Chipkill ECC is per rank. This being a single rank DIMM, failure of the rank implies failure of the DIMM.

SDC rate: A Chipkill system potentially fails to detect an error if three or more chips fail simultaneously within a DIMM inside a scrub interval. A simultaneous three device failure probability is given by $[9 \times 66.1 \times 8 \times 66.1 \times 10^{-9} \times 7 \times 66.1 \times 10^{-9}] \times 32$ ($\approx 4.6 \times 10^{-9}$). The probability of DSD code failing to detect three chip failure is 6.9% [189]. Thus, overall SDC is alteast $(4.6 \times 10^{-9} \times 0.069)$ per billion hours of operation.

For Dvé the SDC rate using a DSD code is twice that of Chipkill since we use double the number of DIMMs for replication and a SDC error can occur in either replica. However, with a TSD code this number reduces drastically as the detection potentially fails only if four or more chips fail simultaneously within a single DIMM.

For even better detection options, low-overhead highly-efficient codes [19] which come closer to reaching the theoretical Shannon limit can be employed. Alternatively, incremental multi-set log hashes [28] can also be used to detect errors. We leave such options for future work.

3.3.2.2 Comparison to IBM RAIM

While Chipkill ECC was not designed to tolerate channel failures, a high reliability system such as IBM RAIM provides a more outright design point for comparison. Recall RAIM uses Chipkill ECC DIMMs with RAID-3 organization across 5 channels; striping four cache lines across four channels and adding redundant diff-MDS ECC code [103] in the fifth channel to correct upto 1 entire channel failure.

We assume 5 RAIM channels each with 8 Chipkill ECC DIMMs and Dvé equipped with 2 replicated channels with 32 Chipkill ECC DIMMs each on different NUMA nodes.

DUE rate: RAIM fails to correct an error if 2 two corresponding Chipkill DIMMs on 2 channels (out of the 5 channels) fail together. Thus, the DUE is calculated as $[(1^{st}\ \mathtt{Chipkill\ DUE} \times 8) \times (4) \times (2^{nd}\ \mathtt{Chipkill\ DUE} \times 1)] \times 5 (\approx 1.5 \times 10^{-14})$ per billion hours of operation.

Dvé+Chipkill fails to correct an error only if 2 pairs of chips in the same position on two DIMMs fail together which is given by $[9 \times 66.1 \times 8 \times 66.1 \times 10^{-9} \times 1 \times 66.1 \times 10^{-9} \times 1 \times 66.1 \times 10^{-9}] \times 32 \times 2$ ($\approx 8.79 \times 10^{-17}$) per billion hours of operation. Hence, Dvé+Chipkill provides $172.4\times$ *(2 orders of magnitude)* lower DUE than RAIM.

SDC rate: Both systems suffer a SDC when Chipkill ECC fails to detect an error. In addition, RAIM also potentially suffers an SDC when 3 channels fail simultaneously. (Probability of this is significantly lower and hence both are limited by Chipkill ECC SDC). Since the total number of DIMMs in Dvé is higher Dvé+Chipkill (64 DIMMs, compared to 40 in RAIM) it has a marginally higher SDC compared to RAIM.

3.3.2.3 Thermal effects on reliability

To factor in temperature effects on reliability we scale the FIT rate using Arrhenius Equation [125]. There exists a 10°C temperature gradient between the DRAM chip closest and farthest to the fan [112], leading to non-uniform FIT rates for the 9 chips within a DIMM scaled as [66.1, 74.3, 82.5, 90.7, 98.9, 107.1, 115.3, 123.5, 131.7]. Using a similar calculation as above, we see that although overall DUE and SDC increases for baseline Chipkill system, Dvé+TSD is able to lower DUE by $4.15\times$ and provide significant reduction in SDC compared to the temperature-factored Chipkill baseline by using a risk inverse mapping (data in chips that have higher FIT rate are mapped to chips in the replica that have lower FIT rate).

When compared to an Intel mirroring-like scheme (employing TSD for a fair comparison), Dvé+TSD is able to lower DUE by 11% using the thermal risk inverse mapping while the Intel mirroring scheme, despite the presence of replicas, does not. While the analysis above exploits a non-uniform thermal distribution across chips in a rank, some boards may exhibit a non-uniform thermal profile across ranks, e.g., ranks closer to the processor may exhibit higher temperatures than ranks further from the processor. Memory controller policies can be designed to place the two copies of data in ranks that are not both at high risk of failure, thus achieving higher overall reliability than thermal-unaware policies – we leave such explorations for future work.

3.3.2.4 Error detection, decoding and recovery process in Dvé

After reading the codewords (data + ECC symbols) stored in the DRAM, the data is checked for errors - commonly termed as error detection or syndrome computation. In this phase, the decoder first recomputes the ECC symbols from the symbols in the received codeword. The difference (in Galois arithmetic) between the symbols in received codeword and the recomputed one is termed as syndrome. The syndrome can be represented as a polynomial equation of the received symbols. If the syndrome is zero, the data is error free. If the syndrome is non-zero, then the received codeword is corrupted and the magnitude and location of the error must be calculated. These additional steps are organized as 3 steps.

First step, the decoder computes two polynomials: error-locator and error-evaluator polynomials. The current ECC logic decoder circuits for this computation are designed based on the Berlekamp–Massey algorithm. Alternatively, other efficient algorithms like Peterson-Gorenstein–Zierler or Galois field Fourier transforms can also be employed for decoding. Second step, to find the erroneous locations, the roots of error locator polynomial are obtained by a Chien search [30]. Third step, the magnitude of the error is calculated on the error-evaluator polynomial using Forney's algorithm [49]

If the local ECC is equipped with detect and correct code, the exact corrected codeword can be obtained by the above steps and the correction can be performed locally. In some cases, the decoding will fail when the number of errors exceeds the capability of the coding scheme or if the local ECC is equipped with a detect-only code [19]. This decoding failure leads to a local DUE. In such cases, a recovery action is initiated using the replica. If the replica also has a DUE failure in a different symbol, the original data can still be retrieved from a combination of the original and replica symbols. For this, the decoding process uses the erroneous symbols from the replica. After reconstruction, the syndrome can again be calculated to ensure the data is error free.

3.3.2.5 Reliability summary

Table 3.1 summarizes the DUE & SDC rates. A key reason why Dvé provides significantly higher reliability over Chipkill and RAIM is because Dvé relies on full replication, while other schemes are all based on ECC (which is a “k-out-of-n” system). More precisely, our competitors rely on `(n - k)` out of `n` hardware entities operating correctly – where hardware entities can be chips, channels etc., and `k` = 1 or 2 and `n` is between 5 and 9, typically. In contrast, Dvé only relies on the exact same hardware entity in the other independent replica not failing. Therefore there are more ways for our competitors to fail compared to us. Finally, it is worth noting that our analytical model does not account for other memory subsystem failures like those in address/data bus, memory controllers etc., due to absence of field data for these. Because Dvé is the only scheme that can tolerate such failures, our analyses serve as lower bound for the actual DUE, SDC rates. Compared to DDR5, which incurs 25% capacity overheads for error detection and correction, Dvé can provide even higher reliability by employing its novel organization i.e., strong detection-only code coupled with replica for correction (12.5% + 50% overheads respectively). The overheads for the detection-only code can be reduced with higher-efficiency optimized codes, which come closer to reaching the Shannon limit [19]

3.3.3 System model

The sections hereinafter describe how Dvé achieves both reliability and performance via replication. We first outline the baseline system model and then describe how Dvé is built over it.

We assume a typical modern system consisting of multiple multi-core chips connected via a cache-coherent, high-bandwidth, low-latency point-to-point interconnect like Intel QPI, UPI or AMD Hyper-transport. Each multi-core chip seated within a socket has a DRAM memory array co-located with it. Each chip has multiple levels of SRAM caches including a last-level cache (LLC) that is shared by the cores within that socket, but globally the LLC is private to each socket as shown in Fig. 3.4(a). On an LLC miss, the request is routed to the “home directory” adjoining the physical location of the home memory

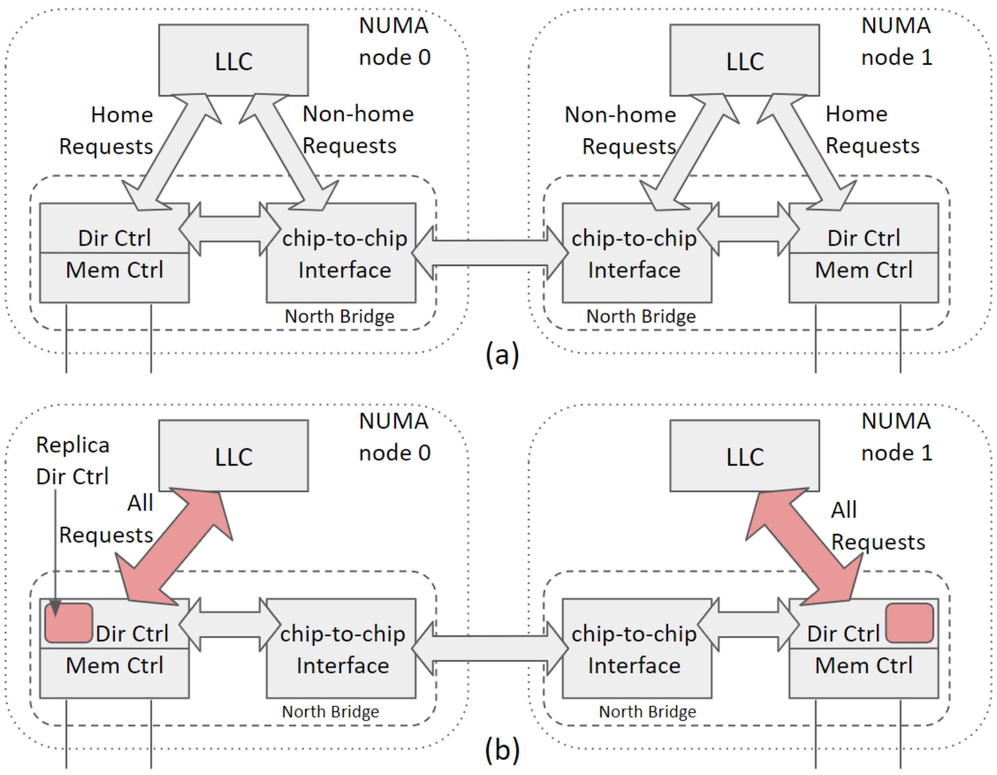

Figure 3.4 *Coherence operation in (a) NUMA (above) (b) Dvé (below)*

controller. (The home directory for an address is determined based on a static hash function of the address.) Thus, the memory access latency depends on where the request originated and where the memory is located. Accessing locations mapped to a remote socket experiences a higher latency, as they require traversing one or more socket interconnect links compared to accessing locations on the same socket. A hierarchical cache coherence protocol handles permissions for each write request and enforces write serialization. Coherence is enforced by looking up the logically centralized (but physically distributed) global directory. We assume a full directory with the recently accessed entries cached on-chip [134].

Dvé uses either a statically reserved portion of the entire memory space for replication with a fixed function mapping or employs a flexible table based mapping populated by the OS as explained in Sec. 3.3.1. For either case, blocks are always inserted into the caches using the original physical address and only the directory controller is responsible for maintaining consistency between the replicas. Note that there are no requests from caches for addresses in the replica pages since these are unused/unallocated by the OS.

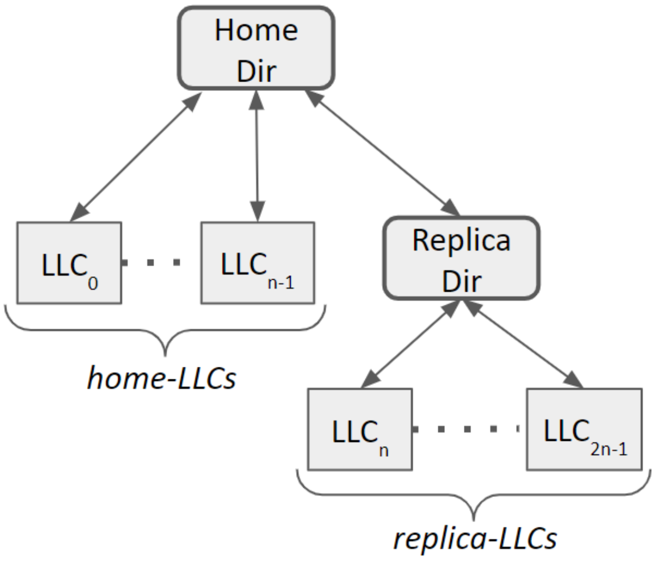

Figure 3.5 *Logical view of coherence in Dvé*

3.3.4 Consistency and recovery semantics

We now precisely specify the consistency and recovery semantics of Dvé.

3.3.4.1 Consistency

Dvé extends the coherence protocol to: (a) keep the replica strongly consistent with the data; (b) ensure that the replica is accessible during fault-free operation.

To maintain a strongly consistent replica, when a dirty cache block is replaced from the LLC, the block is written back to the home node as well as the replica, synchronously. (By “synchronously”, we mean that the request completes only after both home node and replica are written to.) A strongly consistent replica is a necessary but not sufficient condition to ensure that the replica can be accessed during fault-free operation. The data in memory, and hence the replica, could be stale when some cache in the system holds the location in writable state. Therefore, we augment the coherence protocol with additional metadata and logic (in the form of a *replica directory*) to ensure that the replica is accessed only when it is not stale. With these extensions, Dvé ensures that a read request can be serviced from nearest replica.

3.3.4.2 Recovery

Dvé's strong consistency guarantee makes recovery straightforward. When a memory read fails in one of the replicas, i.e., the local ECC check (if equipped with DSD/TSD) or local ECC check+repair (if equipped with Chipkill) at the memory controller fails after a DRAM read, the home/replica directory diverts the request to the other memory controller for recovery. (If the other copy's read also fails, the data is lost (DUE) and a machine check exception is logged and signaled.) If the copy is good, data is returned to the requester and the system logs a Corrected Error (CE). The initial memory controller attempts to fix its copy by updating (writing) it with the correct data and then re-reading the DRAM. If the error was temporary, this read will succeed else the system is placed in a *degraded state* with only one working copy.

3.3.5 Coherent replication

This section describes the details of the coherence protocol extensions for realizing the above consistency and recovery semantics.

Logically speaking, Dvé introduces a new replica directory and so, each location now has a home directory as well as a replica directory. In physical terms, Dvé augments each directory controller with metadata (as shown in Fig. 3.4(b)) to allow for the replica values held in that socket to be safely accessed. Normally each physical directory maintains state for only locations mapped to that node. In Dvé each directory also maintains state about replica locations mapped to that node.

For each location, all LLCs in the system can be classified into two classes: (a) *home-LLCs*: LLCs that send their request to the home directory – the home directory being nearer to them; and (b) *replica-LLCs*: LLCs that send their request to the replica directory – the replica directory being nearer. (Note that LLCs can still cache any block in system memory, only that now requests need not go to the home directory alone.) There is a hierarchical relationship between home directory and replica directory as shown in Fig. 3.5. The replica-LLCs view the replica directory as a cache of the home directory. Transactions originating from the replica-LLCs check the replica directory first before going to the home directory. The home directory on the other hand views the replica directory as one of its "sharers"; it forwards requests to the replica directory which in turn consults its own sharer vector and forwards the requests to one or more of the replica-LLCs.

We propose two protocol families – allow-based and deny-based – based on how access permissions are acquired for replicas. In the former, the replica directory pulls "allow permissions": replica can be accessed only if a replica directory entry for that location explicitly says the location can be accessed; the absence of an entry means "no". In the latter, the home directory pushes "deny permissions" to the replica directory: replica can be accessed unless a directory entry explicitly forbids its access; absence of an entry means "yes".

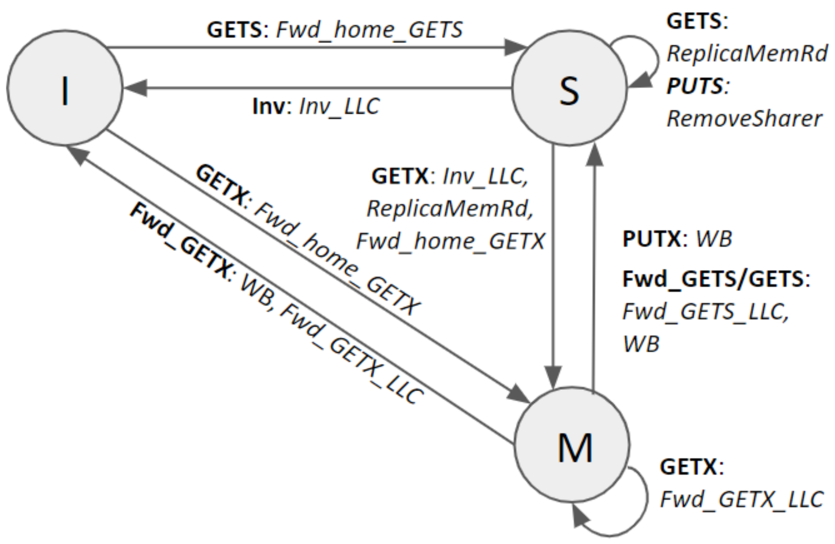

Figure 3.6 *Replica directory controller protocol: stable states and transitions for the allow-based protocol*

3.3.5.1 Allow-based Protocol

The replica directory maintains entries like in a conventional director – including state, sharer vector, and owner. Without loss of generality we assume the MSI states. "Invalid" means that the location is not cached in any of the replica-LLCs. "Shared" means that the location is cached in readable state in one or more of the replica-LLCs. "Modified" means that the location is cached in writable state in one of the replica-LLCs.

Suppose that there is an LLC read miss in a socket that is sent to the replica directory – the socket being closer to the replica directory. The request can safely read from the replica if (and only if) that location is in shared state in the replica directory. If it is in modified state, it has to be routed to the owner in one of the replica-LLCs.

Importantly, if the entry for the location does not exist (i.e., location is in invalid state), the replica cannot be safely accessed because it is possible that one of the home-LLCs may currently hold the block in modified state. In such a case, the request is forwarded to the home directory. The home directory responds to the request with the value and adds the replica directory as one of its “sharers”. Upon receiving the response, the replica directory goes into shared state and sets the sharer vector to point to the LLC that initiated the request.

The complete state transition diagram of the replica controller is illustrated in Fig. 3.6. The states and transitions resemble a conventional MSI directory controller but with one crucial difference. When one of replica-LLCs evict a dirty block in modified state, the replica directory not only writes back the block to the replica memory but also the home memory. In a similar vein, the home directory also writes back a dirty block to both the home memory and replica memory.

In summary, the allow-based family of protocols lazily pull read permissions for the replica upon access. This reactive approach works well on workloads with significant private writes; it makes sense to avoid pushing permissions to the replica directory via the inter-socket link when other threads are not likely to read those lines.

3.3.5.2 Deny-based Protocol

In contrast to allow-based, in the deny-based family of protocols, the home directory eagerly pushes deny permissions (i.e., knowledge about writable locations in their LLCs) to the replica directory. In doing so, the replica memory can be accessed even if there is no directory entry corresponding to that loca-

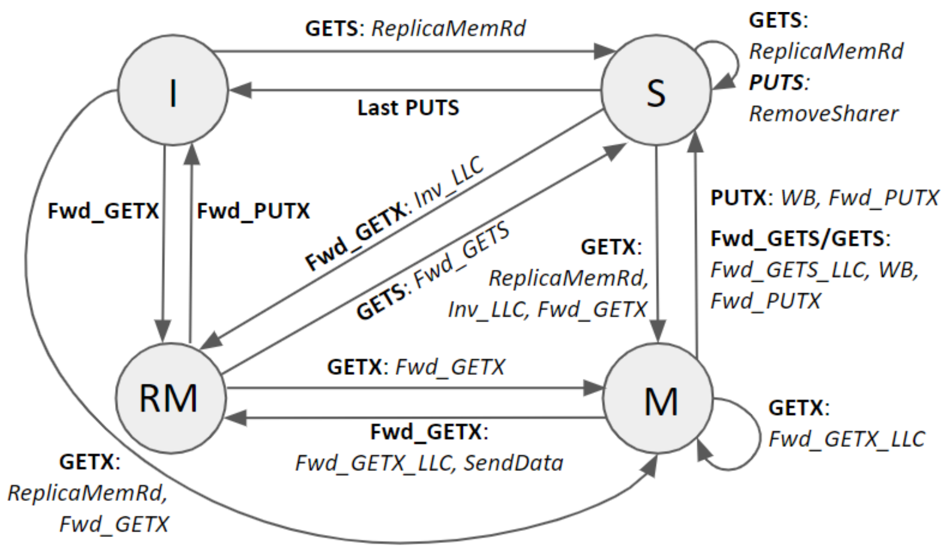

Figure 3.7 *Replica directory controller protocol: stable states and transitions for the deny-based protocol*

tion. The proactive approach is suited to read-only (or mostly-read) workloads since directories can read the nearest replica without the need for requesting permissions.

Like in the allow-based protocol, the replica directory again maintains entries like in a conventional protocol including: state, sharer vector, and owner. States include the conventional MSI states, but additionally a new *remote modified* (RM) state. A location in RM implies that one of the home LLCs have the block in modified state; this implies that the replica is stale and hence cannot be accessed directly. “Invalid”,“Modified” and “Shared” states mean the same as in the allow-based protocol.

Suppose that there is an LLC miss that is sent to the replica directory. The request can be safely read from the replica as long as the location is not in RM state. Note that the replica can be read even if there is no entry in the replica directory. Indeed, the absence of an entry implies that there are no remote writers and hence means that the replica is not stale.

Finally, as in the allow-based protocol, an evicted dirty LLC block is written to both the home memory and the replica memory. The state diagram of the replica directory controller is illustrated in Fig. 3.7.

3.3.5.3 Handling Recovery

The recovery process of Dvé is simple and does not involve any stop-the-world state update: when a local memory controller returns a read failure, the directory simply forwards the request to the replica memory controller. In other words, the coherence protocol is agnostic to the recovery action. Any concurrent request from a cache or an I/O operation is serialized and coalesced at the directory in the MSHR/intermediate state as in the baseline coherence protocol. This invariant ensures correctness for all cases.

3.3.5.4 Complete protocol and Verification

Until now we have discussed only stable states and transitions, implicitly assuming that state transitions happen atomically. In reality, each transition involves a number of steps in modern systems. For this reason, transient states and actions are necessary to enforce logical atomicity. We have fully fleshed out complete protocol specifications including transient states and actions for both protocol variants. Further, we have modeled the complete protocol in the Murϕ model checker [40] and exhaustively verified the protocol for deadlock-freedom and safety, i.e., they enforce the Single-Write-Multiple-Reader invariant [134]. The detailed state transition table for the replica controller and the Murϕ model are available online[7].

3.3.5.5 Protocol Optimizations.

We describe 3 protocol optimizations for improving performance.

(i) Speculative replica access. Consider an LLC miss that is sent to the replica directory in the allow-based protocol; further let us assume that an entry for the location is not present in the replica directory. This can either mean that the location is in writable state in one of home-LLCs or the directory entry has been evicted. The only way to find out for sure is to ask the home directory. But in the meantime the local replica can be speculatively accessed to overlap memory

[7] https://github.com/adarshpatil/dve

latency with home directory access. A similar optimization can be employed in the deny-based protocol as well: in case of a replica directory miss, the local replica can be speculatively accessed while waiting for the entry to be retrieved from DRAM.

(ii) Coarse-grained replica directory. Until now we had assumed that the replica directories operate at cache line granularity. We draw inspiration from prior works [23],[131] to amortize overheads by using coarse-grain replica directories. We use a contiguous, aligned block of memory to be covered by one entry when possible i.e. a full memory block is entered into the replica directory if no cacheline within it is currently in writable state.

(iii) Sampling based dynamic protocol. The performance of allow-based vs disallow-based protocols is dependent on the workload. A sampling based dynamic protocol can be used to achieve the best of both. We apply both approaches to a region of memory for a few epochs and monitor their effectiveness (similar in spirit to set dueling). Such an implementation requires a few additional saturating counters in the profiling phase. The scheme that performs better is then applied to the rest of the memory. When switching between the protocols (based on a register being set in the CPU by compiler or OS), we enter a drain phase to clear the replica directory and stop reading from the replica memory. We then switch state machines, followed by a warmup phase to bring the metadata entries *au courant*.

3.4 Discussion

We discuss how a flexible replica region is organized for on-demand replication.

3.4.1 OS support for memory replication

We now discuss the OS support needed to enable memory replication using the flexible table based mapping (relaxing the restriction of fixed function mapping assumed so far). There are 3 fundamental questions that need to be addressed for this: (i) How does the OS carve and manage space required for replication?

(ii) How does the OS map replica page pairs? (iii) When does the OS enable or disable replication?

(i) Carving and managing space required for replication: The OS already uses heuristics to estimate unused memory to transparently cache accessed files (called as disk buffering or file caching). Such approaches to estimate maximum DRAM resident set size can be reused to opportunistically steal system visible memory for replication. Further, balloon drivers [185] in the OS can be used to create memory pressure, forcing it to select pages to swap to disk, thereby carving memory space for replication. If memory pages are required to be swapped out to disk, the OS can monitor page fault rate to ensure that excessive swapping does not cause performance degradation beyond a pre-defined threshold. Note that Dvé only requires pairs of pages in different NUMA nodes and not a large contiguous address space, thus avoiding the need to perform memory compaction. In the absolute worst-case, when additional memory capacity is essential (either during burst periods or diurnal workloads), server management infrastructure can notify the OS to disable Dvé replication. The memory relinquished can be hot-plugged back to system visible capacity (free memory pool). Dvé's modular design of building over ECC enabled DIMMs with Chipkill allows the system to provide the baseline reliability when Dvé is disabled.

(ii) Mapping replica page pairs: As explained in Sec. 3.3.1, replica page pairs are stored in an in-memory data structure called RMT. As the OS is already aware of the memory layout on boot via EFI it can create replica pairs such that they exist on memory adjoining different sockets. The RMT can be cached at the directory controller for quick lookups and the controllers can lookup/walk the RMT in hardware when needed (similar to a page table walker). Replication can even be performed at coarse granularity, allowing the RMT to be organized as a simple linear table or a 2-level radix-tree (similar to the page table) to perform fast end-to-end translations. A mapping entry can remain in the RMT despite the page being deallocated. This reduces the number of times the RMT cache would need to be shot down or quiesced. Further, RMT changes are infrequent since it only needs modifications in the rare event of a capacity crunch i.e., free list runs out, and the OS reclaims replica pages for use as addressable memory. Lastly, RMT entries can also be apriori populated by the OS heuristically in anticipation of replication requests for fast turn-around at memory allocation.

The RMT is an in-memory data structure and therefore it is conceivable that the DRAM locations housing the RMT might encounter an error. To enable recovery from such cases, the RMT can also be replicated. The RMT replicas are housed at a fixed location which is computed with a fixed function mapping (as explained in Sec. 3.3.1). If a RMT memory read encounters an error, the fixed function mapping is used to compute the location of the RMT replica. Correspondingly, an RMT write must also be replicated to both the fixed locations.

(iii) When should replication be enabled or disabled?: The onus is on the workload placement and server management infrastructure (aka Control Plane) to define critical workloads and notify the OS when such replication costs are justified. The workload placement infrastructure manages the capacity vs reliability memory mode decision as a soft setting on a fleet of commodity high-capacity machines without having to procure specialized hardware for high reliability memory. This allows greater flexibility as datacenters and HPC installations can provision single homogeneous iso-config hardware to enable easier cost-efficient management while using Dvé to run mission critical workloads where reliability is a non-negotiable first order concern.

Dvé's replicated reliable memory can be flexibly deployed: (i) by the hypervisor at per-VM granularity, (ii) per-container or per-process granularity for serverless FaaS (Function-as-a-Service) deployments, (iii) for kernel allocations where system stability is of utmost importance or in workloads like file servers which use large amounts of OS memory to perform basic kernel operations, (iv) by mirroring entire address space to protect aging machines errors. With this knowledge, the OS adds a flag to the process control block (PCB) at creation process time to always allocate replicated memory.

Alternatively, to allow an application to explicitly provide high reliability to certain memory regions (say for a stateless application to allocate failure resilient data segments), a variant of the malloc/calloc call can be provided to request the OS to allocate a replicated physical memory.

Finally, note that Dvé guarantees higher reliability and improved performance only for the replicated region.

Future Work: Dvé's above design of the OS functionality for flexible memory replication i.e., the carving space for replication, mapping replicas and the mapping service, is based on the Linux virtual memory model. We leave the actual implementation of the design to future work.

3.4.2 Performance caveats of Dvé

Dvé aims to maximize the performance obtained by using the Coherent Replication scheme. Recall that if there are any issues with one of the replicas, for e.g. due to hard errors or thermal throttling or preventing a row-hammer access pattern, the system is placed in a degraded state where there is only one working copy. This negates the performance benefits for that memory location. Note that by marking the locations which are in a degraded state and funneling their requests to the single functional location, Dvé will provide performance comparable to baseline NUMA. The performance benefits may also be nullified or marginally adversely affected with Dvé if all compute and memory accessed is localized to a single NUMA node. This is because all memory writes have to be replicated to both NUMA nodes (not in critical path).

3.5 Evaluating Dvé

3.5.1 Evaluation goals

The goals of our evaluation are as follows. First, to evaluate the performance benefits of the coherent replication protocols over baseline NUMA architecture. Second, to explain the performance benefits using the key metric of coherence traffic between sockets. Third, evaluate the optimizations presented in Sec. 3.3.5.5. Fourth, evaluate the robustness of performance gains of Dvé schemes to varied inter-socket latencies. Finally, study the impact of replication on DRAM and system energy.

Table 3.2 *Configuration of the Dvé simulated system*

Processors	16-core, 2 socket, (8-core/socket), 3.0 GHz
L1 I/D Cache	64KB, 8-way, private per core, 1 cycle, writeback
L2 Cache (LLC)	8MB, 16-way, shared per socket, 20 cycles, writeback
Local Directory	embedded in L2, fine-grain (cores) sharing vector
Global Directory	20-cycle, coarse-grain (sockets) sharing vector
Baseline Memory	$2 \times$ 8GB DDR4-2400 Mhz, 8 devices, 8-bit interface tCL-tRCD-tRP-tRAS=14.16ns-14.16ns-14.16ns-32ns 1KB row buffer, 16 banks/rank, 1 channel/socket
Replicated Memory	$4 \times$ 8GB DDR4-2400 Mhz, 2 channel/socket
Intra-socket interconnect	$2{\times}4$ Mesh, SSSP routing, 1 cycle per hop
Inter-socket interconnect	point-to-point, 50ns

3.5.2 Evaluation methodology

We now set out the parameters and configuration of the multi-socket NUMA architecture used in our experimental evaluation of the coherent replication protocols proposed.

3.5.2.1 System Configuration

We model a two-socket, Intel Skylake-like processor configuration with mesh topology within the socket and a point-to-point QPI/UPI-like interconnect between the sockets. All links are ordered and have a fixed latency. A table-based static routing is enforced with a shortest path route with minimum number of link traversals. Each multi-core chip has per-core private caches, a shared LLC, a directory and a memory controller. The system is kept coherent using a hierarchical MOESI/MOSI protocol (full config details in Table 3.2). Memory is allocated using an interleave policy whereby adjacent pages are interleaved across memory controllers in a round-robin fashion. We use an inter-socket latency of 50ns per hop. This is in-line with the difference between local and remote memory on a dual-socket Intel SandyBridge machine [69]. We also study the performance sensitivity to inter-socket latency.

3.5.2.2 Memory Configuration

Since our primary aim is to demonstrate the benefits of coherent replication in conditions where workloads memory needs are already satisfied, both the baseline and Dvé have the same system visible memory capacity. We assume the entire system memory is replicated using a fixed function mapping as explained in Sec 3.3.1. In our evaluation, to accommodate the additional capacity required to store the replica, we add DIMMs on another channel on both the NUMA nodes as shown in Fig. 3.3. While several other lower performance configurations can be used to increase the capacity required to house the replica (like using a dual rank DIMM or higher capacity chips in a DIMM), this does not affect the overall reliability of the system as the replicas are anyway stored on different NUMA nodes.

3.5.2.3 Simulator setup

Simulating large core count systems with high-capacity DRAMs and large application working set sizes is challenging with existing publicly available tools. To circumvent this, we generate traces of benchmarks using the Prism framework [140] which uses Valgrind to generates traces of compute, memory, multi-threading APIs like create/join, mutexes, barriers, conditional wait/signaling/broadcasting, spin locks/unlocks and producer-consumer thread communication events. The tool captures synchronization and dependency-aware, architecture-agnostic multi-threaded traces.

We replay traces in a modified gem5 simulator [157]. Integer/floating point computations and thread API events have fixed latency of 1 cycle and 100 cycles respectively while all memory operations are simulated in detail. The replay mechanism respects synchronizations, barriers and mutexes. We skip the serial sections and/or initialization parts of the programs and instrument only the region of interest (ROI) in the benchmarks. Since the simulator is driven by program execution traces, only the entire user-space level data is replicated. The kernel-space/OS data and events are not collected and thus not captured in the simulated replication scheme.

Table 3.3 *Multi-threaded workloads evaluated*

Suite	Benchmark
HPC (assorted)	comd[9], xsbench[176], graph500[63], rsbench[175]
PARSEC [18]	canneal, freqmine, streamcluster
SPLASH-2x [179]	barnes, fft, ocean_cp
Rodinia [26]	backprop, bfs, nw
NAS PB [138]	mg, bt, sp, lu
Parboil [90]	stencil, histo
SPEC 2017 [173]	lbm

3.5.2.4 Workloads

We use OpenMP and Pthreads based multi-threaded workloads from 7 benchmark suites. Memory intensive applications were chosen from these suites (Table 3.3). We use the largest input dataset available for these benchmarks. Traces of the ROI are used to warm-up the caches and structures for the first 1 billion operations i.e. computation, memory or communication events (which correspond to between 0.5-1.15 billion instructions) and then simulated in detail for 20 billion operations (8-19.3 billion instructions). We order the workloads in descending order of L2 MPKI and report the geometric mean of speedup as an aggregate statistic for the top-10 (high MPKI), top-15 and all 20 benchmarks.

3.5.2.5 Protocol Configuration

We use a fully associative 2K entry structure for the replica directory. We assume same access latency for the replica directory as the home directory for both protocols. We employ speculative replica memory access optimization as part of the default configuration. While this does trade-off bandwidth for squashed replies, we find that in our simulations the latency benefits outweigh the bandwidth loss. For the sampling based dynamic scheme optimization, we run a profile phase for the workload for 100 million instructions every 1 billion instructions for each scheme. We then apply the scheme that performs better for the rest of the phase.

3.5.3 Evaluation results

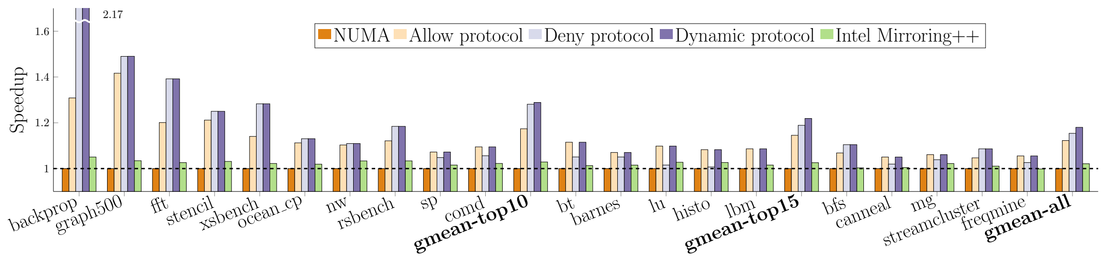

Figure 3.8 *Performance comparison of all configurations (normalized to NUMA)*

3.5.3.1 Performance Benefits of Coherent Replication

Fig. 3.8 shows the performance of allow, deny and dynamic protocol normalized to a baseline NUMA system without memory replication. The deny protocol achieves an average speedup of 28%, 18% and 15% for the top-10, top-15 and all benchmarks respectively, while the allow protocol achieves an average speedup of 17%, 14% and 12% respectively over the baseline. Although deny protocol performs better on average, only 10 benchmarks (backprop, graph, fft, stencil, xsbench, ocean_cp, nw, rsbench, bfs, streamcluster) show better performance with a deny protocol while the other workloads perform better with the allow protocol. The dynamic protocol is always able to detect the better of 2 protocols. Therefore the dynamic protocol achieves the best average speedup of 29%, 22% and 18% over the baseline. A key point to note is that all benchmarks for all schemes, perform equal to or better than the baseline which shows that the overheads of coherent replication does not cause any adverse performance penalties.

To compare against the best possible performance that can be gained by reading data from 2 mirrored channels, we implement an improved (hypothetical) version of Intel's memory mirroring scheme by actively load balancing reads between them. (Recall the default Intel memory mirroring scheme does not read the secondary copy, unless the primary fails.) Dvé's schemes provide a geomean speedup of 9% and 13% better than the Intel-mirroring++ scheme for allow and deny respectively, as they are able to avoid the inter-socket latency by providing node local reads while the Intel-mirroring++ scheme can only provide higher read memory bandwidth by allowing reads from both channels.

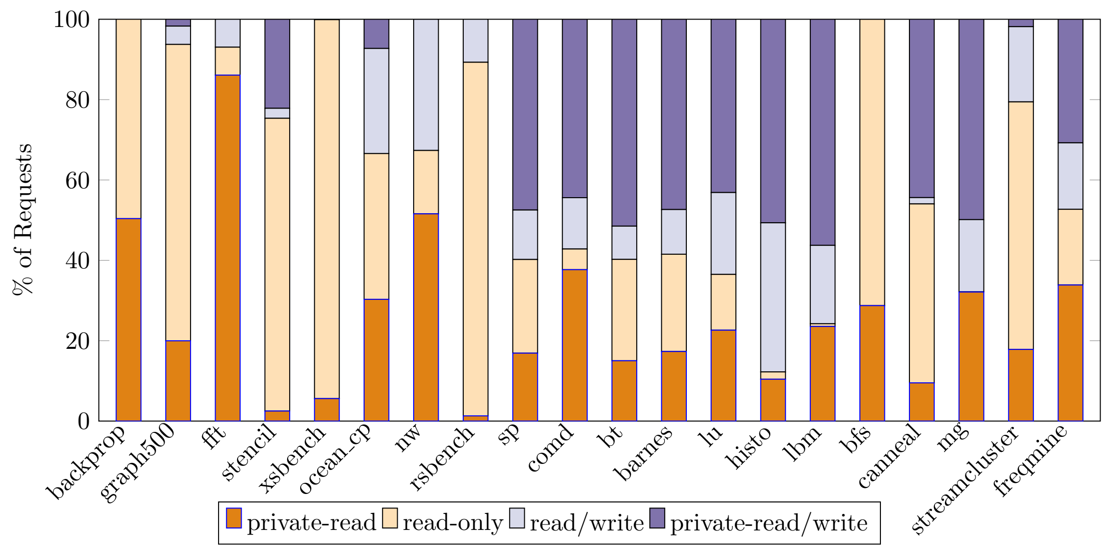

Figure 3.9 *Sharing pattern in benchmarks*

3.5.3.2 Performance Analysis - Workload Sharing Characteristics

To assess why a scheme performs better for a workload, we look at the inter-socket sharing characteristics of workloads in the baseline NUMA system. By analyzing this, we can understand the opportunity for performance gains from each scheme. We classify the requests at the home directory as one of the following – *private-read* for a GETS request to a line in I state, *read-only* for a GETS request to a line in S state, *read/write* for a GETS request to a line in M or O state or a GETX request to a line in S state, *private-read/write* for GETX request to a line in I state. Fig 3.9 shows the distribution of the above mentioned classes for each workload. Workloads that exhibit considerable private read/write behavior (greater than 46%) show higher benefits with an allow protocol. This is expected, since for such workloads on a GETX request where there is no sharing, the allow protocol is able to avoid aggressively pushing invalidates (by virtue of being a lazy, pull-based scheme) while the deny protocol is required to update the replica directory.

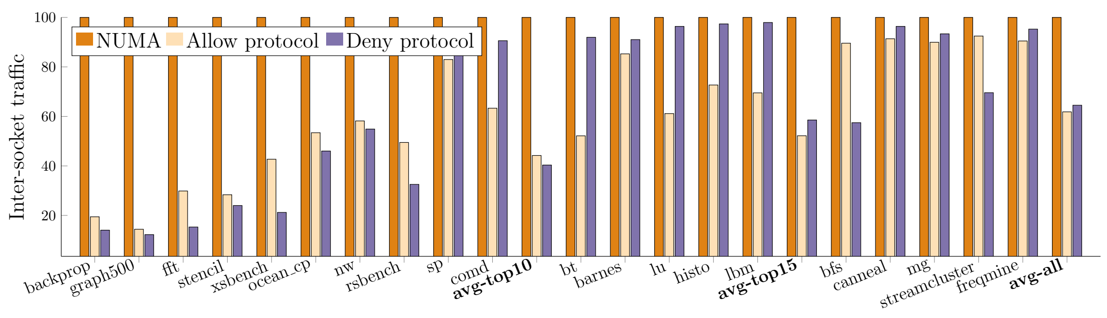

Figure 3.10 *Inter-socket traffic (normalized to NUMA)*

3.5.3.3 Performance Analysis - Inter-socket traffic

Note that both allow and deny protocols incur additional inter-socket writes to replicate data. However, the protocols reduce inter-socket reads by reading from local replica memory. As observed in the workload sharing characteristics analysis above, the workloads show significant private read/write behavior and thereby benefit from Dvé's protocols. The allow-based protocol avoids re-requesting inter-socket read permissions on a subsequent request and the deny-based protocol avoids inter-socket communication completely when a read from the local replica memory is possible. Overall, the protocols reduce the inter-socket traffic, compared to the baseline NUMA.

Fig 3.10 quantifies the reduction in inter-socket traffic when using Dvé protocols compared to baseline NUMA. We see that reduction in inter-socket traffic correlates with the performance benefits, i.e., the protocol which shows higher reduction in inter-socket traffic performs better. Backprop and graph500 notably experience a 86% and 84% reduction in inter-socket traffic as they see mostly private-read and read-only requests which can be serviced by the replica. Overall for the 20 benchmarks, allow and deny protocols reduce inter-socket communication traffic by an average of 38% and 35% respectively over the baseline NUMA architecture.

3.5.3.4 Optimizing the schemes

To gauge the ceiling of performance we can expect from an allow-based protocol, we measure the performance achievable with an oracular allow-based

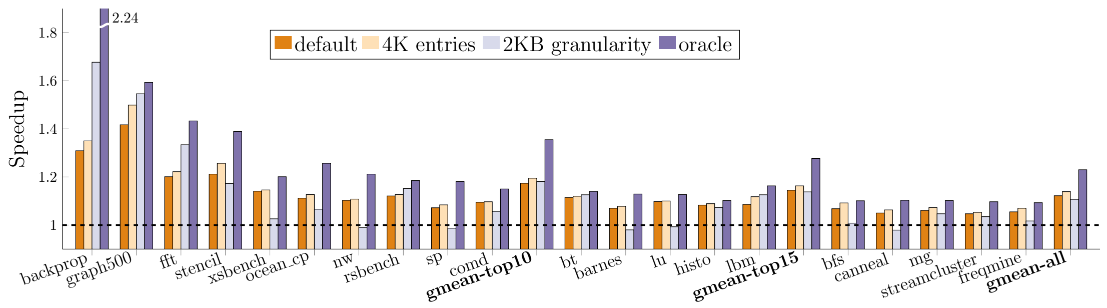

Figure 3.11 *Performance of allow-based protocol optimizations*

scheme. Intuitively, the size of replica directory structure is proportional to the efficiency of the scheme and thus, for an oracular scheme we allow the replica directory entries to be infinite and exhaustive. Further, it does not incur any latency overheads to add an entry (in the spirit of oracle knowledge) while invalidation/removal of entries does incur latency (since this can never be avoided). With such a configuration, we see that the performance of oracular scheme is 18.3% and 10.8% better than the default allow protocol for the top10 and all benchmarks respectively (Fig. 3.11, fourth bar)

In reality to achieve near oracular performance, we double the number of replica directory entries to a fully associative 4K entry structure. We see that the larger structure provides an average hit rate improvement of 32% and improves performance by 2.1% and 1.7% over the default allow protocol for top10 and all benchmarks respectively (Fig. 3.11, second bar).

Another optimization we can use is to use coarse-grain tracking at the replica directory to increase the reach. When such an optimization is employed benchmarks such as backprop, graph500, fft, rsbench show gains while stencil, ocean_cp, comd, bfs suffer compared to the default cache-line level tracking (Fig. 3.11, third bar). Further nw, sp, barnes and canneal perform worse than baseline NUMA. This is due to the additional overhead to invalidate entries from other sockets when an exclusive request is issued for a larger granularity. Overall, coarse granularity tracking performs 0.7% better for top10 benchmarks but performs worse by 1.7% over all benchmarks compared to the default allow protocol. Thus such a technique is not well suited to improve performance.

We note that the performance of the best variant of the allow protocol for each benchmark is within 12% and 7% of the oracular performance for top10 and all benchmarks (16% better than the baseline NUMA overall).

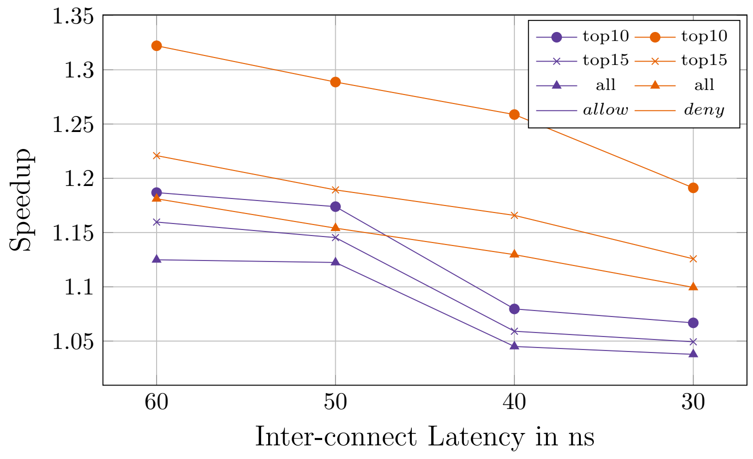

Figure 3.12 *Sensitivity to interconnect latency*

3.5.3.5 Sensitivity to inter-socket latency

A number of software-based techniques – including Carrefour [37], Shoal [91], AutoNUMA – have been proposed to mitigate NUMA effects; the net effect of each of these techniques is to reduce the average inter-socket latency. Therefore, we study the effect of inter-socket interconnect latencies on the performance of Dvé. As seen in Fig. 3.12, even with fairly low 30ns interconnect latency (each way), the deny protocol outperforms the baseline by 19%, 12% and 10% for top-10, top-15 and all benchmarks respectively. On the other end, with increased inter-socket latencies of 60ns (as in emerging scalable long range interconnects like CCIX[24], OpenCAPI[143] and GenZ[55]), Dvé's benefits increase as expected.

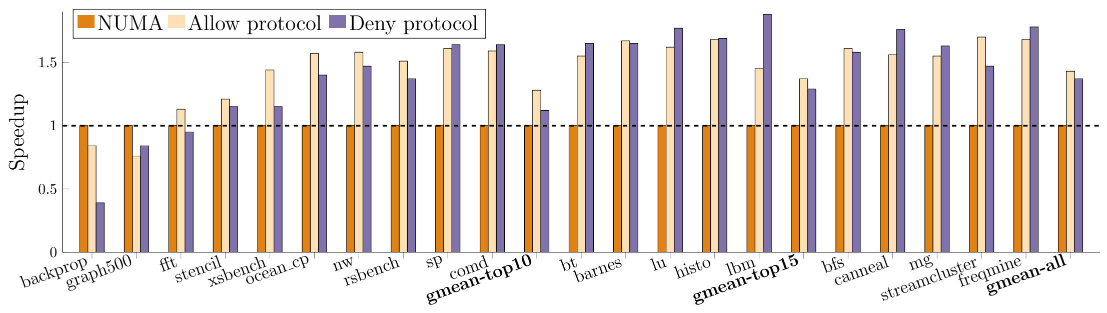

Figure 3.13 *Energy-Delay Product of DRAM subsystem*

3.5.3.6 Energy

To understand the energy overheads of maintaining a replica for each location, we measure the energy-delay product (EDP) of the DRAM subsystem. Each command to the DRAM consumes a specified amount of energy to execute, as specified in the Micron datasheet [123]. We use these values in the simulator to compute the total energy consumed by the DRAM subsystem while executing the workload. Fig. 3.13 shows the EDP of allow and deny protocols normalized to the baseline non-replicated NUMA system without idle memory. We observe that, memory-EDP for memory intensive benchmarks (backprop, graph500, fft) reduces even with the overheads of double memory capacity but the geomean over all benchmarks increases by 43% and 37% for allow and deny protocols respectively. (Note that we expect the memory-EDP of Dvé to be even lower when using idle memory as it still uses energy for refresh, even in a low power (self-refresh) state.) Moreover, typically memory consumes only a fraction of the overall system power: about 18% of the total system power in a 2-socket NUMA system [14]. Using this to calculate the system-EDP, we find that the system-EDP geomean over all benchmarks turns out to be lower by 6% and 12% for allow and deny respectively, due to shorter execution times.

3.5.4 Evaluation summary

Our experimental results indicate that Dvé provides significant performance improvement for memory intensive applications with various sharing characteristics. Dvé also reduces inter-socket traffic and DRAM energy usage and its performance gains are robust across a wide range of interconnect latencies.

3.6 Related work

State-of-the-art ECC proposals: A large body of recent work have proposed several variants of ECC schemes to deal with errors in the DRAM devices [25], [57], [88], [94], [95], [136], [177], [191]. Specifically, multi-tier ECC approaches proposed in [87], [177], [191] separate error check and error correction code.

AIECC [95] sheds light on the need for channel error protection (clock, control, command and address buses) and a holistic ECC based scheme to achieve it.

All these proposals rely on correction bits stored in the same DIMM/rank/channel and thus cannot correct memory controller or channel failures as in this work. Dvé can be flexibly paired with any proposed ECC scheme for error detection, and use the replica to recover from an error. These works also suffer performance degradation for providing additional protection while Dvé provides performance benefits.

Non-ECC based DRAM RAS schemes: To detect transmission errors on channels, DDR4 memory controllers use CRCs and retry transactions if errors are detected [123]. If errors persist, lanes are quiesced, reset and recalibrated but cannot handle hard channel failures. Aside from ECC based systems, Intel [68, 74] and IBM processors [72] allow memory to be mirrored across two channels within a memory controller or across two DIMMs within a channel. While these mechanisms can tolerate channel failures, they are still subject to faults in the single memory controller subsystem.

MemGuard [28] uses incremental log hashes for error detection and a OS created checkpoint for error recovery. However the recovery is not instant and can lead to loss of updates. Selective word level replication [137] proposes selective replication of words to mitigating large granularity failures in DRAM chips with manufacturing defects. As before, all these mechanisms cause performance degradation for providing at most channel error protection.

Stacked DRAM RAS schemes: Stacked DRAMs are prone to failures in dies and TSVs due to their organization, which resemble chip and channel failures. Adopting conventional ECC schemes causes performance pathologies due to the layout of data. To protect against such errors without causing performance/power overheads a body of work [27, 84, 89, 118, 135] calls for adding additional tier-2 code that is an XOR of the data blocks. This XOR block is then stored in the same stacked DRAM in a manner that allows regeneration of original block in case of a TSV/die failure. Although the XOR block is stored in an independent channel it cannot be accessed independently without the data block (as it not a full replica). Even if a replica were used instead of XOR block, these works would resemble at best an Intel mirroring approach [74].

Mitigating NUMA overheads: Carrefour [37] proposes OS-driven selective replication of memory read-only or read-mostly pages to alleviate NUMA overheads. Shoal [91] proposed program analysis to automatically replicate memory regions across NUMA nodes for mitigating the performance penalty of remote access.

Architectural solutions for mitigating NUMA has had a long history, most notably in the cache-only-memory architecture (COMA) and tertiary caching line of research [15, 159, 183, 195]. More recently, C3D [69] and Candy [31] use a per-socket stacked DRAM cache to reduce NUMA interconnect latencies. All of these works leverage hardware support for caching remote data and keeping them consistent. None of the aforementioned works aim to provide fault-tolerance. While Dvé aims to use the replica for improving performance, it's primary goal is to provide improved DRAM fault-tolerance.

NVM RAS mechanisms NVMs suffer from high number of hard errors due to cell wear out and stuck-at faults. ECP [161], FREE-p [190], PAYG [150] and Chipkill-correct for NVRAM [193] propose using pointers or error correction entries to remap failed bits in a line or use variants of ECC. All these schemes still target cell failures and cannot deal with larger granularity failures due to errors in shared components.

Dynamically Replicated Memory (DRM) [81] uses a read-after-write scheme to detect errors. When an error is detected, the scheme makes a copy to a compatible page with dissimilar faults. Both copies need to be accessed to service a request. We find that NVM memory RAS mechanisms are also insufficient to handle the scope of errors targeted in our work.

Exploiting underutilized memory FMR [145] proposes replicating data into idle memory – on different ranks in the same memory controller and uses it to hide DRAM maintenance latency overheads like refresh, precharge delay etc. FMR performs lock step write into the rank replicas within the same memory controller. This is very similar to the Intel-mirroring++ scheme implemented in our evaluation.

Workload co-location exploiting workload heterogeneity and variability has been proposed to improve utilization in the cloud[113]. However, this is having limited effectiveness given the dynamic bursty nature coupled with stringent tail latency, throughput guarantee requirements and increased security concerns. Memory disaggregation has also been proposed to allow workloads that require high capacity memory to take advantage of idle remote memory to improve performance [82], which is orthogonal to our approach.

3.7 Summary

In this chapter, we have presented Dvé, a hardware memory replication mechanism for achieving both reliability and performance. We have demonstrated that this unique design point offers considerably higher memory reliability and can be flexibly deployed on-demand. Furthermore, our experimental results indicate that in contrast to existing memory reliability techniques that are detrimental to performance, Dvé provides a non-trivial improvement in performance.

4

Āpta: Fault-tolerant object-granular CXL disaggregated memory for accelerating FaaS

In the previous chapter, coherent replication was employed to improve reliability and performance of memory within a datacenter server. However, as we observed in Chapter 1, in today's datacenters, main memory capacity is also located outside a server i.e., disaggregated using CXL. This disaggregated memory can also be shared between multiple servers and kept coherent with the CXL.mem protocol.

In this chapter, we present Āpta- a technique to improve the reliability of CXL disaggregated memory. Āpta specializes this disaggregated memory to improve the performance of function-as-a-service (FaaS) applications.

4.1 Overview

The FaaS model is quickly becoming the defacto standard for cloud developers. In FaaS, applications are composed as workflows of stateless functions, and the cloud provider then orchestrates and schedules the functions dynamically on a fleet of compute servers.

The stateless nature of functions is good for availability, scalability, and elasticity, but it inevitably forces state to be maintained externally. Indeed, data stores such as Amazon S3 [21] are used to maintain state and pass input/output data between the stateless functions in the workflow. These data stores are the backbone of FaaS platforms.

Splitting state and compute, however, has an intrinsic data movement cost. Our analysis of FaaS functions from the FunctionBench [96] and SeBS [35] benchmark suites shows that on average 96% of the execution time per FaaS function is spent in retrieving data from the S3 object store. Replacing the S3 object store with a RDMA-based in-memory object store improves the situation somewhat – with 51% of execution time spent in retrieving data – but the problem persists. Communication overheads still limits performance.

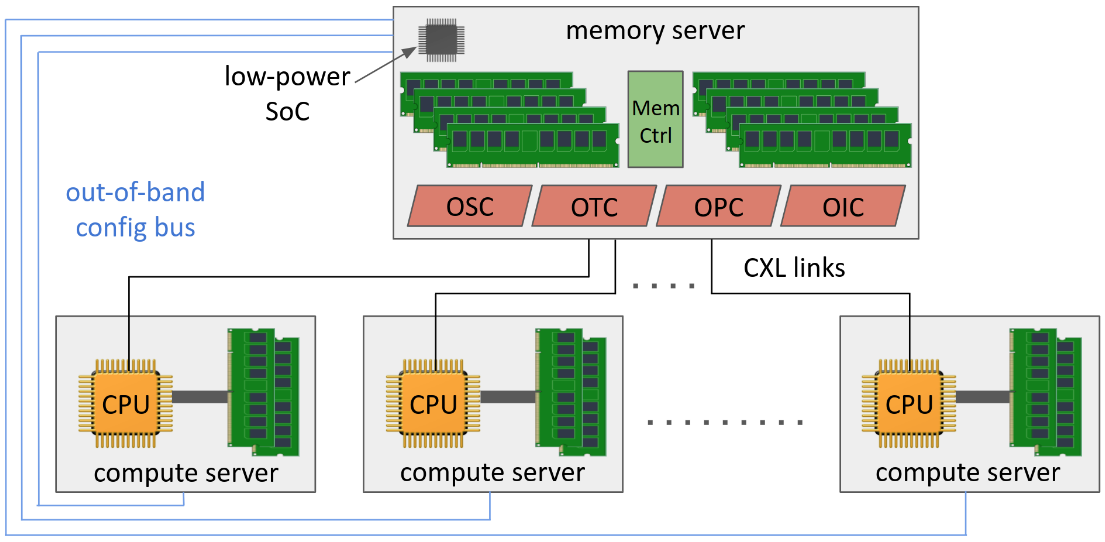

Figure 4.1 *Āpta system schematic (new controllers are shown in red). The figure shows a CXL disaggregated memory system, where compute servers are connected to and cache data from a logically centralized (physically distributed and highly-available) memory server via a hardware load/store interface. Āpta augments the memory server with new controllers to support object-granular accesses and keeps the caches consistent with a fault-tolerant coherence protocol. Sec 4.3.4.1 defines the micro-architecture of Āpta's memory server controllers: object serving controller (OSC), object tracker controller (OTC), object persistence controller (OPC), object invalidation controller (OIC).*

Insight: FaaS objects on CXL disaggregated memory. We observe that upcoming CXL-based hardware disaggregated memory [33, 143] is a promising avenue for maintaining FaaS objects. CXL pools memory resources onto a logically centralized, physically distributed, highly-available memory server, and allows compute servers to perform load/store remote memory accesses in hardware over the network. The memory server, as shown in Fig. 4.1, is equipped with specialized hardware controllers for performing frequent *data plane* operations and minimalist low-power processors to handle rarer *control plane* operations [65, 105]. Since CXL allows for loads and stores to be handled in hardware like in a traditional NUMA machine [107, 108, 149], CXL-based disaggregated memory allows for significantly lower latency and higher bandwidth compared to high-performance RDMA-based remote memory.

Furthermore, the recently announced CXL 3.0 specification [34] allows the compute server caches to transparently cache data from a *shared region* in disaggregated memory, which matches well with the access patterns of a FaaS object store. Because FaaS functions typically share objects between them, object accesses exhibit significant locality and are amenable to caching. Therefore, such object caching and the use of a locality-aware scheduling policy (schedule functions where objects it uses are cached) has the potential for significantly reducing data movement.

Our analysis shows that a FaaS object store over a CXL-based disaggregated memory system with support for object-granular accesses, coupled with a locality-aware scheduling policy can improve performance of the aforementioned FaaS functions by a significant 2.3× over the state-of-the-art RDMA-based object store. This is the performance opportunity Āpta targets.

CXL provides consistency but forgoes availability. To preserve flexibility and maximize throughput, a cloud system *dynamically* schedules a function on any available compute server. This dynamicity combined with compute-side caching results in FaaS objects being replicated, and it is imperative that the replicas be kept consistent. Because compute servers can fail or be unresponsive in the datacenter, it is important that the consistency protocol remains *available* in the presence of such failures: i.e., the protocol should not block indefinitely if any of the servers fail. Alas the CXL 3.0 protocol [154] (which is a conventional protocol that enforces the Single-Writer-Multiple-Reader (SWMR) invariant [134]), while enforcing strong consistency, fundamentally blocks in the presence of server failures: if a server sharing an object fails, a write to that sharer from any other server could indefinitely block waiting for an acknowledgment from the failed sharer. Thus, this naive application of a traditional multi-processor coherence protocol (non fault-tolerant) for distributed disaggregated memory leaves CXL systems vulnerable to system crashes.

Severity of the problem: Building system resiliency is an important problem as servers frequently fail or become unavailable in a datacenter environment. Google has observed that up to 25% of service-level disruptions are caused by machine-level failures [14]. A study of errors in even the highest reliability petascale supercomputers has shown that network link and server faults causing job failures occur with a mean time between failures (MTBF) of 160 hours [39, 85]. Consequently, fault tolerance is a key tenet of FaaS platforms. This is precisely why FaaS applications have already embraced failures via idempotent functions [5, 61]: if a function fails while executing (e.g., due to a compute server failure) the FaaS function can simply recover by re-executing. Therefore, it is imperative that the underlying CXL-based object store operates correctly in the presence of such server failures.

Consistency & availability via fault-tolerant coherence. We transform a strongly consistent SWMR-enforcing coherence protocol into a highly-available protocol in the presence of compute server failures. The idea consists of two simple steps: *lazy invalidation* and *coherence-aware scheduling*. In the first step we move the invalidations out of the critical path of the write so a writer is not blocked indefinitely when a server caching the sharer fails. But because invalidations are moved off the critical path of the write, there is a window of inconsistency where caches may hold stale values. In the second step we make a simple change to the FaaS scheduler [101] allowing it to schedule functions *only* on servers where there are no pending invalidations – thereby enforcing strong consistency as well as availability. Āpta's method for transforming the non fault-tolerant coherence protocol into a highly-available one can easily be applied to upcoming versions of CXL.

Contributions.

1. We make the case for a CXL-based object store for FaaS with object-granular reads/writes (Sec 4.2). Our analysis using stand-alone FaaS functions indicates that such a design can provide a 69× performance improvement over the Amazon S3-based FaaS system, and a 2.3× im-

Table 4.1 *Taxonomy of state-of-the-art proposals*

System	Caching support? (granularity, write-policy, inter-server sharing, coherence mechanism, sharer invalidation)	Hardware support?	Compute server fault-tolerance	Performance for object stores
S3 [21]	No	No	High	Low
Pond[108], Kona[22] ThymesisFlow [149]	Yes (cacheline, write-back, No, N/A N/A)	Yes	Low	Low
LegoOS [163]	Yes (page, write-back, No, N/A, N/A)	Yes	Low	Medium
Clio-KV [65]	Yes (object, write-through, Yes, No, N/A)	Yes	Low	Medium
MIND [105]	Yes (page, write-back, Yes, MSI, sync)	Yes	Low	Medium
OFC [133]	Yes (object, write-back, Yes, version-based - all reads require remote version match, No)	No	Low	Medium
Faa$t [155]	Yes (object, write-through, Yes, version-based - all reads require remote version match, No)	No	High	Medium
CXL 3.0 spec [34]	Yes (cacheline, write-back, Yes, MESI, sync)	Yes	Low	Medium
Āpta	Yes (object, write-through, Yes, SI, async)	Yes	High	High

provement over a RDMA-based system. We observe, however, that such a system must remain fault-tolerant, which existing CXL protocol specification falls short of.

2. We introduce Āpta (Fig. 4.1) – a CXL-based object store that allows compute server-side caching of objects without compromising consistency or availability (Sec. 4.3). Āpta is tailored for object-granular accesses and defines a fault-tolerant inter-server cache-coherence protocol that, together with the FaaS scheduler, enforces strong consistency and provides high-availability in the presence of server failures. We have verified safety and liveness of the protocol in a model checker.

3. We evaluate the performance of Āpta (Sec. 4.4) using 6 full FaaS applications (total of 26 functions) and show that it provides 21–90% execution time speedup over a state-of-the-art fault-tolerant RDMA-based object store and 15–42% speedup over a reliable CXL-based object store without caching.

4. We observe that amongst all state-of-the-art high-performance remote memories and object stores that support caching (Table 4.1), Āpta has the highest performance, and the highest availability in the presence of compute server failures.

4.2 Motivation

In this section, we first demonstrate the compelling performance reason to migrate FaaS object store to a disaggregated memory system (abbreviated as DM). Next, we illustrate why DM, while providing improved performance, falls short of providing the level of fault-tolerance required for the FaaS paradigm. Finally, we highlight inefficiencies when existing cache line granularity DM is used to design an object store.

4.2.1 The performance potential of a DM-based object store

We compare the performance of FaaS functions from FunctionBench [96] and SeBS [35] benchmark suites[1] using three different object stores: Amazon S3, RDMA-based, and DM-based object stores. The functions execute two basic operations on the object store: `obj` $\leftarrow$ `get(objID)` at the beginning and `put(objID, obj)` at the end, where `objID` is an identifier for an object `obj`. The computations in the middle of these functions are often unoptimized which hides the true bottlenecks in the system. We envision that high-performance frameworks such as Google TensorFlow [79] and Facebook PyTorch [78] will be adopted for FaaS in the future. We therefore ran the functions with Intel OneAPI [76] which applies vectorization, parallelization, cache blocking and other architecture specific optimizations. The optimized computation was run on a 16-core Intel Xeon Skylake machine. This measured computation time was kept constant across all runs while the `get` and `put` time varied based on the object store employed.

FaaS functions experience high communication overheads with Amazon S3: When using the S3 object store, a `get` operation downloads the object from a remote S3 server into the compute server memory using http network protocol; post computation the `put` operation uploads an object from the compute server into a remote S3 server using http network protocol. We take the median of 100 executions accounting for cold function and tail latency effects [51, 182].

[1]We exclude micro benchmarks and network benchmarks that are non-deterministic and sensitive to external system delays.

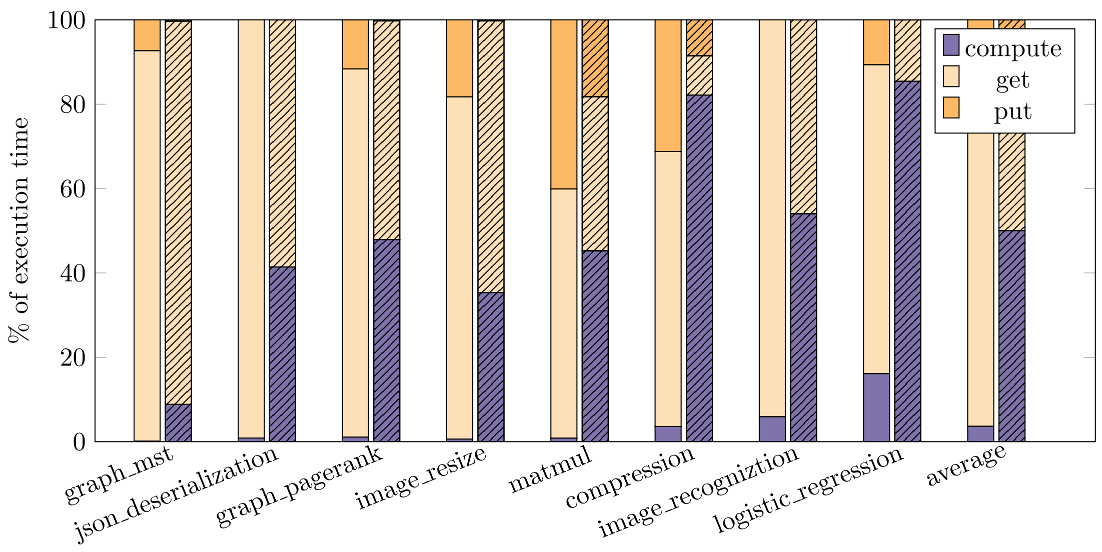

Figure 4.2 *Compute-to-communication ratio in function execution using - (a) Amazon S3 (first bar) (b) in-memory RDMA store (second, striped bar)*

We observe that on an average 96% of execution time is spent in communicating data from/to the S3 object store (Fig. 4.2, all unstriped bars). This shows that the execution of FaaS functions in the cloud today is severely limited by the latency of accessing data from object stores. While S3 is based on disk based storage servers, it employs several optimizations like replication and sharding [21] to provide the best performance among today's production object stores.

In-memory object store does not alleviate the communication overheads: High-performance RDMA-based in-memory object stores completely bypass the remote CPU to read (write) objects directly from (into) the memory of the remote object server [36, 64, 129]. For this, the `get` and `put` operations were modified to use one-sided RDMA verbs which runs over an Infiniband network (single-port Mellanox ConnectX-3 NIC on PCIe-gen3 x16) [43, 120]. Even with such modern RDMA-based data store, on average 51% of execution time is still spent in communicating objects (Fig. 4.2, all striped bars).

Overcoming RDMA's Achilles heel: The RDMA-based approach has several fundamental characteristics that limit performance – the use of software libraries like libibverbs and libmlx4, the need to perform two DMA *data copy* operations (at source and at destination, copying data to/from RNICs Memory Region) and managing the memory regions with software-initiated per-server static connection queue pairs. Several works have analyzed these and other drawbacks of RDMA [62, 65, 107]. DirectCXL [62] quantifies that even with the same underlying physical interconnect, RDMA's irreducible overheads makes `get/put` operations 2.2× slower than CXL-based DM. DM is the new approach that chip manufacturers and cloud providers are investing in. DM overcomes the drawbacks of RDMA by allowing all data plane operations to be handled in hardware, thereby providing lower latency and higher bandwidth.

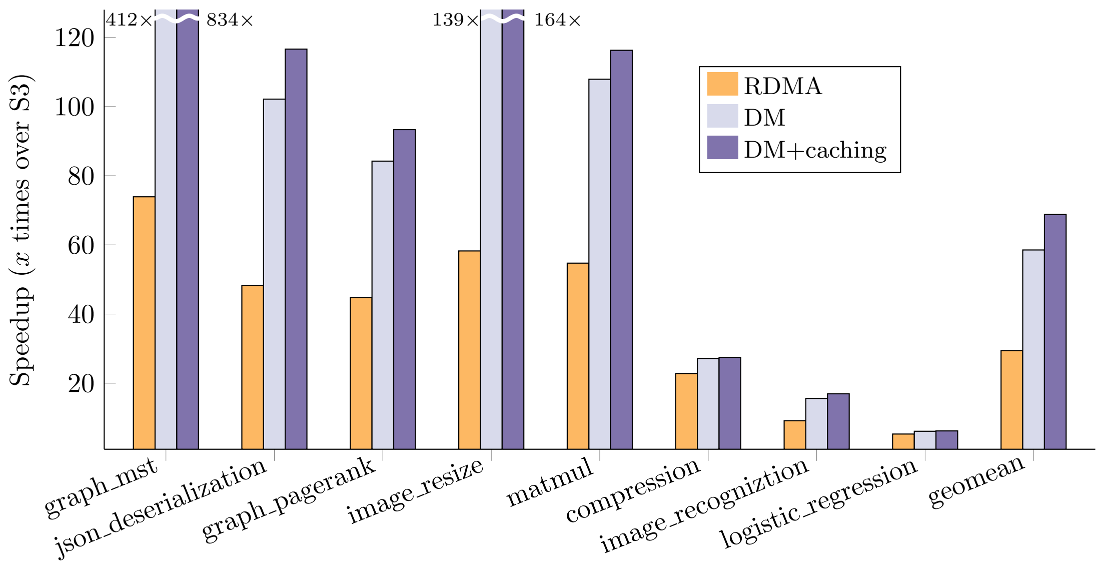

Figure 4.3 *Comparison of FaaS functions performance with various object stores (Baseline Amazon S3)*

DM reduces communication overheads: The object is retrieved from a load-/store semantic DM system. All standards for building such a DM system (GenZ[55], OpenCAPI[143]) have coalesced under the CXL umbrella due to their synergistic goals. Currently however, there exist only early prototypes: (i) OpenCAPI-based DM[149], providing RTT latencies of 950 ns and a bandwidth of 12.5 GiB/s; (ii) CXL-based DM[62], providing a lower RTT latency of 500 ns and a higher bandwidth. We pessimistically model the worst-case latency and bandwidth of OpenCAPI for our DM system. Our modeled DM system lowers latency by 3× and improves bandwidth by 10× over the RDMA system [107, 149].

With DM, the fraction of execution time spent for communication in FaaS functions reduces to 13% of the total time, on average. This translates to a large reduction in execution times of the functions. Fig. 4.3 shows that the DM-based

Table 4.2 *Object size analysis of Microsoft Azure trace data [127]*

Access Type	Median	Mean	Mode
Read	8 B	66.81 kB	28 B
Write	1.4 kB	45.16 kB	63 B
All	28 B	61.90 kB	28 B

object store is able to achieve a 59× geomean speedup over Amazon S3 and a 2× speedup over the RDMA object store.

Caching - an additional benefit of DM: A CXL DM system transparently caches object cache lines in the (on-chip SRAM or DRAM) hardware caches of the compute server, thereby being served at a lower latency compared to a remote memory server access. Such caching is extremely effective for FaaS applications which exhibit good object access locality [127]. This is because full FaaS applications, defined a state machine workflow of multiple individual functions ("function chains"), demonstrate known communication patterns like producer-consumer and broadcast within them [184]. This communication implies that successor functions can potentially access objects produced by any of its predecessors. When functions read objects from compute server caches in the DM system their execution time further speedups by 2%-100% (Fig. 4.3 DM+caching, assumes a DRAM cache of DDR4-like latency).

A relevant question is also if the hardware caches on compute servers can handle caching FaaS objects. This is particularly important to investigate as FaaS platforms optimize for cost by over-provisioning and densely packing several 100s of function instances on a single compute server. Our analysis of Microsoft Azure functions trace data reveals that the median size of objects is just 28 bytes (Table 4.2) and 80% of objects are smaller than 12KB. Therefore, objects can quite easily fit into existing caches and even more so with large DRAM caches as present in several modern processors.

Summary: Our analysis indicates that maintaining FaaS objects in DM and the caching benefits it provides largely mitigates the key performance bottleneck of the FaaS paradigm.

4.2.2 The lack of fault-tolerance in current DM systems

FaaS object stores, such as Amazon S3, are designed to provide fault-tolerant operation for a failure-prone datacenter environment. Object `get` and `put` atomically read and write entire objects, with all or nothing semantics. A `get` is also guaranteed to read the value of the most recent `put`, therefore providing a strong consistency model known as linearizability [21]. This greatly simplifies things for a FaaS developer who can simply assume that a `get` would return the object written by the most recent put in the workflow.

Enforcing strong consistency in the presence of caching. In the caching DM system, enforcing strong consistency for the FaaS execution environment can be challenging. For example, consider a simple workflow consisting of three functions: $\mathtt{f}_1 \rightarrow \mathtt{f}_2 \rightarrow \mathtt{f}_3$, where $\mathtt{f}_1$ and $\mathtt{f}_3$ read object X, while $\mathtt{f}_2$ writes to X. Further, let us assume that $\mathtt{f}_1$ is assigned to server C1 while $\mathtt{f}_2$ is assigned to C2. When $\mathtt{f}_1$ executes on C1, it would cache the object in C1. When $\mathtt{f}_2$ writes the object, it would render the value cached in C1 stale. Suppose the FaaS scheduler chooses to schedule $\mathtt{f}_3$ on C1, $\mathtt{f}_3$ would then read the stale value of X, violating strong consistency.

One way to enforce strong consistency in the presence of caching is to employ a cache coherence protocol. Conveniently, CXL 3.0 specifies an inter-server MESI-based coherence protocol [154], that enforces the SWMR invariant. In the above example, the write from $\mathtt{f}_2$ would invalidate the cached copy of X in C1, ensuring that when $\mathtt{f}_3$ is scheduled on C1, it will read the most recent value written by C2, and not the stale value.

Whither Fault tolerance? It is imperative that the aforementioned inter-server cache coherence protocol operates correctly even when compute servers fail or become unavailable. (In this work we assume that the DM server is kept highly-available using techniques such as replication [146, 163] and power redundancy.) Alas, traditional coherence protocols can block in the presence of such failures. Consider the same example where $\mathtt{f}_1$ caches object X in server C1. When $\mathtt{f}_2$ executing on C2, writes to X, the coherence protocol would send an invalidation to C1 which holds the object. Now, should C1 fail or become unreachable the write from $\mathtt{f}_2$ would simply block, waiting for an acknowledgment, thereby rendering the system unavailable. Even if C1 does not fail but is simply slow to acknowledge (e.g., due to network congestion), the write from $\texttt{f}_2$ would be impacted, which can lead to high tail latency – a critical issue for FaaS platforms [181].

4.2.3 Inefficiencies of DM for object stores

Current DM system standards specify fixed fine-grain data access, caching and coherence mechanisms. However, object reads/writes typically have widely variable sizes, ranging from bytes to MBs [44, 127]. This causes two key inefficiencies. CXL enables compute servers to read cache lines from memory server while objects frequently span multiple cache lines. Hence, reading an object will incur multiple round trips to the DM, owing to limited MSHRs (miss status handling registers).

Second, CXL permits single cache line atomic write while a `put` must atomically write an object of multiple cache lines to the DM. This incurs additional latency for software write-ahead-logging i.e., undo/redo logs. Our analysis for the above benchmarks shows that a CXL object store will incur an average of 32% and 89% higher latency for `get` and `put` respectively, compared to an optimized object granular DM (evaluation methodology in Sec. 4.4).

Summary: Supporting compute server caching mandates a fault-tolerant coherence protocol that enforces strong consistency in the presence of compute server failures. CXL-based DM systems fail to provide this. Second, existing CXL cache line granular accesses are ill suited for FaaS object granular accesses.

4.3 Āpta

Āpta's goal is to design a DM-based object store for FaaS applications (Sec. 4.3.1) that provides fault-tolerant coherence (Sec. 4.3.2) and optimum performance (Sec. 4.3.3). To accomplish this, Āpta designs DM hardware controllers and modifies runtime software (Sec. 4.3.4). Sec. 4.3.5 walks through the

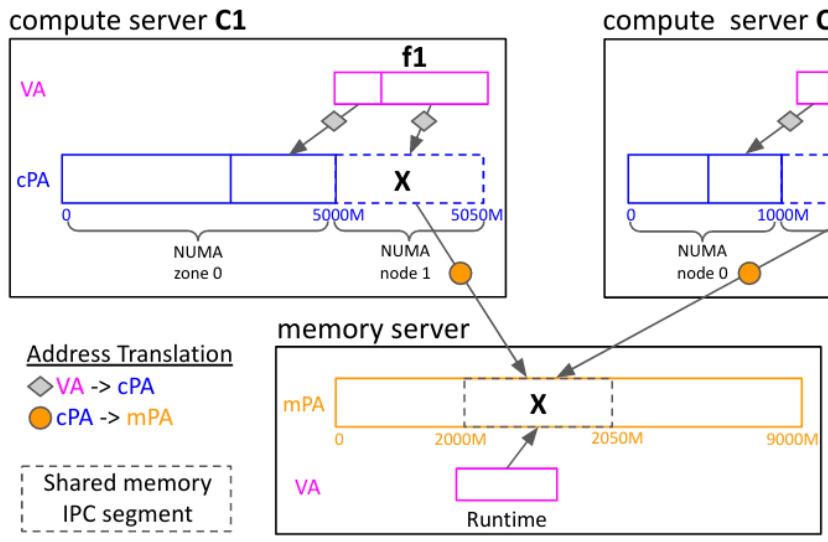

Figure 4.4 *FaaS object sharing through DM: organization and addressing*

working of the entire Āpta system when executing real-world FaaS applications.

4.3.1 Setting the stage: Designing a DM-based object store

This section describes how Āpta leverages the features of a CXL 3.0 based DM system to construct an object store.

(a_1) Sharing objects between FaaS functions through DM ▶ *Extend shared memory inter-process communication (IPC)*
CXL 3.0 allows compute servers to access a shared memory region on the memory server. The compute server OS discovers and manages this CXL memory device as per UEFI/ACPI specifications [178] and exposes the DM address space as an extended CPU-less NUMA region [107, 108, 143].

In Āpta, FaaS functions execute as independent processes on compute servers. To access a shared object, the `get` and `put` operations map a DM memory region (containing the object) into their virtual memory using shared memory inter-process communication (shmem IPC) [155]. The shmem IPC API is enhanced to allow function processes on different compute servers to mount a shared memory region.

To illustrate, Fig. 4.4 shows the two functions f_1 and f_2 executing on server C1 and C2 respectively, sharing the object X of size 50MB through the DM system.

On each compute server, the shmem IPC segment, where X resides, is located in an extended NUMA physical address space (cPA - blue dashed regions in Fig. 4.4). Just as in CXL, the access controls and page tables for end-to-end address translation from compute server process virtual address (VA) to the memory server physical address (mPA) are initialized and setup by the OS[2]. Once mapped, the object is accessed by the CPU (during the compute phase of the function) using load/store on cPA address. When these accesses miss in the LLC, the request is routed to the "home node" of the extended NUMA region (DM controller on the compute server). The DM controller uses the mapping to verify permissions and provides the memory server physical addresses (mPA) to be accessed.

(a_2) **Caching objects in compute server caches** ▶ *Defining a caching policy* Āpta introduces minimal changes to compute server caches, making them almost oblivious to disaggregation (in the spirit of CXL). The `get` operation, when mapping the shared object, sets the memory region of objects larger than size of the LLC as uncacheable using PAT or MTRR[144][3]. On an LLC miss, the cache line is read from DM and allocated in compute server caches. Similarly, objects are also write-allocated in the LLC. This policy allows retaining data in the LLC for any expected future reuse.

Importantly, the `put` operation immediately writes all modified cached lines through to DM, making the caches effectively write-through. This policy allows tolerating compute server failures since a compute server LLC never holds the only copy of the object, and the memory server always holds a valid copy. The LLC silently evicts any of the DM cache lines which are in shared state i.e., the LLC does not issue a PutS coherence request to the directory. This saves interconnect network bandwidth and avoids LLCs having to evict entire objects if one of the object's cache lines is evicted.

(a_3) **Exploiting the locality provided by caching** ▶ *Locality aware scheduling policy*

[2]CXL uses Address Translation Services (ATS) defined in PCIe Specification for translation of cPA$\rightarrow$mPA. The compute server OS sets the translation table base address register - ZMMU [56] or extended page table pointer[99].

[3]Other object cache allocation policies can be employed - e.g., fraction of LLC capacity per CPU, dynamically based on predictors of object hotness or reuse potential [133] etc. The exploration of allocation policies is orthogonal.

The FaaS runtime schedules each function invocation on compute servers. The scheduler makes intelligent heuristic decisions to achieve lowest execution latency for the functions execution by accounting for various factors [51, 116, 182]. In Āpta, this runtime scheduler is used to exploit object locality by scheduling invocations on compute servers where cached objects will be or are likely to be reused. This allows functions to benefit from lower latency for object access.

This is done on a best effort, maximization basis. For generality, in this work, the scheduler heuristically assumes that the function has a high likelihood of accessing any object consumed or produced by any of its predecessors and picks a compute server where most predecessors executed.

4.3.2 Fault-tolerant Coherence Protocol

The conventional MESI coherence protocol, specified in CXL 3.0, does not provide reliable operation in an environment where compute servers can fail independently. We now detail Āpta's highly-available fault-tolerant coherence protocol that is designed for a failure-prone cloud environment.

($\mathtt{b}_1$) Keeping the cached objects on compute servers coherent ▶ *Tailored coherence mechanism and protocol*
“Simplicity is prerequisite for reliability" - Edsger Dijkstra *Simplified coherence:* Recall that a FaaS function reads the object from the memory server and the compute server caches it in shared state; a `put` writes-through to the memory server and subsequently invalidates all sharers of the object in other compute servers. This eliminates the need for Modified or Exclusive states and reduces the inter-server protocol to two stable states - Shared and Invalid. This simplified coherence protocol, designed for the execution model of FaaS functions, hardens Āpta against compute server faults.

This Āpta protocol is layered hierarchically over and above intra-server coherence protocol. The intra-server coherence protocol is unchanged and regardless of this protocol Āpta enforces different policies in the inter-server protocol. This hierarchic organization allows Āpta to track sharers at compute server granularity (not individual caches within them). The Āpta protocol is employed for all requests from the compute server to the DM server.

Coarse granularity tracking: The use of DM in FaaS systems is restricted to sharing objects. Thus, it suffices for Āpta to use variable-sized object granularity tracking for the coherence protocol, as opposed to cache line level tracking in traditional chip level coherence protocols. In other words, we use a single state to encapsulate the state of all cache lines within the object. This is tracked using an object unique triplet of (`objID`, base mPA, size).

(b_2) Provide high-availability while enforcing strong consistency ► *Lazy invalidation of sharers with coherence-aware scheduling*
Recall, to enforce strong consistency of the caches, a `put` completes only when all servers caching that object are invalidated; therefore, `put` can block when servers fail. This invariant of any conventional coherence protocol called Single-writer-multiple-reader (SWMR) is enforced by synchronously invalidating all sharers in the critical path of the `put`.

In Āpta, the sharers are sent an invalidation message asynchronously, i.e., the `put` is acknowledged immediately without waiting for the sharers to be invalidated. The sharers that are sent invalidations are tracked off the critical path until they acknowledge the invalidation messages.

This lazy invalidation policy: (a) allows the write to be acknowledged at lower latency thereby improving performance and (b) more importantly, because writes need not wait for sharers to be invalidated, there is no risk of writes being blocked, thereby ensuring fault-tolerance.

Whither Consistency? Note, however, that this asynchronous protocol described above could violate SWMR (and hence linearizability). This is because at the instant the put is acknowledged there may be cached copies in other servers yet to be invalidated.

Lazy linearizability with scheduler support. Āpta enforces linearizability lazily using a combination of the coherence protocol and the FaaS runtime scheduler. More specifically, Āpta never schedules function invocations on servers with pending invalidations – the *§Scheduling Correctness Criterion*. This correctness criterion ensures there is no risk of reading any yet-to-be-invalidated stale objects present in the caches. More precisely, we are now in a position to assert Lemma 1.

Lemma 1. *The coherence protocol ensures that a `get` returns the value of most recent `put` to that object.*

Proof. Consider a `get` to object X. When the `get` is about to be scheduled, there are either pending invalidations to X or there are none. If there are no pending invalidations, there are no stale values and hence the `get` will return the latest value as per the original synchronous protocol. If there are one or more pending invalidations, the scheduler ensures that the function containing the `get` is not scheduled on those servers with pending invalidations, and hence there is no risk of `get` reading a stale value. □

Thus, the Āpta coherence protocol ensures that the caches on the compute servers where functions execute are strongly consistent. Meanwhile, caches in compute servers where functions are not executing can be stale without affecting consistency. Another benefit of Āpta's lazy invalidation protocol is the ability to perform coherence actions at line-rate. This is particularly important for processing packets in the data plane on DPUs or SmartNICs in the network [48, 142].

4.3.3 Addressing the inefficiencies of DM

This section describes Āpta's optimization to adapt the DM system for object idiosyncrasies.

(c_1) Object-granular reads ▶ *Via bulk cache line loads*
Recall, for each object load request that miss in the LLC, the DM controller on the compute server issues a single cache line read request over the interconnect making it inefficient for objects spanning multiple cache lines.

CXL 3.0 [34] does provides fixed block request semantics (2 or 4 contiguous cache lines) with the block request size to be specified in advance. However, objects have more variable block sizes and compute server LLCs cannot specify the block size in advance as they operate oblivious of objects.

Āpta builds on CXL 3.0 to provide variable sized, bulk cache line requests. It *bulk reads* all the objects cache lines into the compute server cache in one

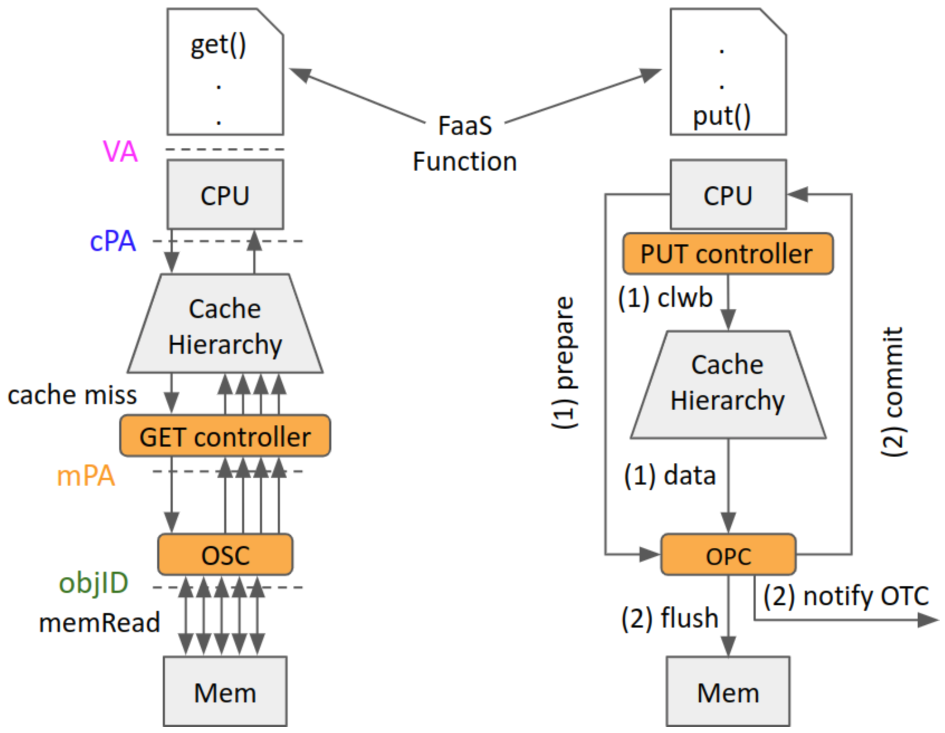

Figure 4.5 *Operation of `get` (left) and `put` (right) using controllers on the compute and memory servers (controllers shaded in orange).*

round trip to the memory server (similar to [36]), providing the lowest possible latency and maximizing the interconnect bandwidth utilization. This process is illustrated in Fig. 4.5, left. The GET controller (optimized DM controller) issues the LLC's read request over the interconnect. The memory server reads all cache lines constituting the object from DRAM memory. It replies with all these cache lines and squashes/ignores any immediate requests for this object from that compute server. The GET controller receives all the prefetched cache lines and inserts them into the respective cache sets in the LLC. The LLC forwards the demand miss cache lines to lower level caches and the CPU.

(c_2) Object-atomic writes ▸ *Transactional atomic durability*
Recall, CXL permits atomic writes of single cache lines which forces a `put` to use software transactions (*libpmemobj* API) to write an object of multiple cache lines atomically to the DM system. These transactions use software logging (undo or redo) which adds significant number of additional instructions per transaction, hurting latency and throughput.

Āpta provides hardware transactions for object atomic writes to improve performance (similar to [66, 114]). The hardware transaction ensures that when an object `put` is executed, either the entire object is persisted[4] or, in case of failures, any partial writes are collectively discarded. If the transaction succeeds, the memory server overwrites the new version into the objects memory area. If the transaction fails, it is retried, assuming the cause of the failure is transient. If a retry threshold is exceeded, the exception is reported to an external FaaS infrastructure system and the entire function execution is considered to have failed.

In the compute server, the PUT controller, co-located with the CPU, flags for persistence all the cache lines written by an object `put`. The controller orchestrates an atomic transaction, using a one-phase commit protocol with the memory server (Fig. 4.5, right). When the memory server issues a commit response, the PUT controller clears the persistence flags.

4.3.4 Realizing Āpta's architecture

We now detail the memory server components - data plane controllers, control plane software and the interaction between them required to realize Āpta. We also describe in detail the coherence protocol sketched out in the previous section.

4.3.4.1 Micro-architecture of data-plane controllers

The data plane on the memory server is composed of a conventional memory controller and the Āpta controller. The Āpta controller is composed of four modular sub-controllers as shown in Fig. 4.1. This section details the micro-architecture of these controllers, each providing a certain functionality.

Object Serving sub-controller (OSC):
≻ *Function:* Serving objects (bulk cache lines) when the GET controller requests an object's cache line.
The OSC translates the requested mPA to an object triplet. For this, OSC

[4]Recall the memory server is usually kept highly-available and persistent

walks the *object mapping data structure*, populated by the FaaS runtime object manager (See 4.3.4.2). Similar to page tables, this translation latency can be reduced by using TLBs, page walk caches, cuckoo filters [167] etc. Once the physical address of the object is retrieved, the OSC issues memory access requests to the memory controller and replies to the compute server once it receives the data from it.

Object Persistence sub-controller (OPC):

≻ *Function:* Ensures the entire object is persisted, atomically into the DM system.

Recall, an object `put` initiates a one-phase commit protocol, between the PUT controller on the compute server and OPC on the memory server, to atomically write all the objects cache lines. As shown in Fig. 4.5 (right), first, the PUT controller on the compute server CPU sends a prepare message with the `objID`, the number of cache lines and their positions within the object to be written and the . Then, it issues cacheline writeback (*clwb*[75]) for all the cache lines that are written by the `put`. The data from these cache lines, resident anywhere in the cache hierarchy of the compute server, are flushed to the memory server. OPC uses buffers to temporarily stage cache lines written back from the compute server. These buffers can be implemented either as SRAM registers at the LLC for small objects or a dedicated DRAM area for larger objects. OPC expects to receive a fixed number of cache line writes to complete the object write (received in the prepare message). Once it receives all cache lines of the object, it replies with a commit message, marking the end of transaction. OPC notifies object tracker controller (OTC) of the competition of an object write and flushes/drains the buffers to the memory controller.

Object Tracker sub-controller (OTC):

≻ *Function:* Serves as the directory for the Āpta coherence protocol.

Similar to a conventional directory, OTC maintains entries about the state and sharer vector for each object triplet. The OTC directory represents the ordering point for all requests. The directory is inclusive of all the compute server LLCs i.e., it holds directory entries for a superset of all objects cached in all the compute server LLCs. A miss in this directory cache indicates that the object is in state I. The sharer list tracked is not precise since the compute server LLC silently evicts blocks in shared state.

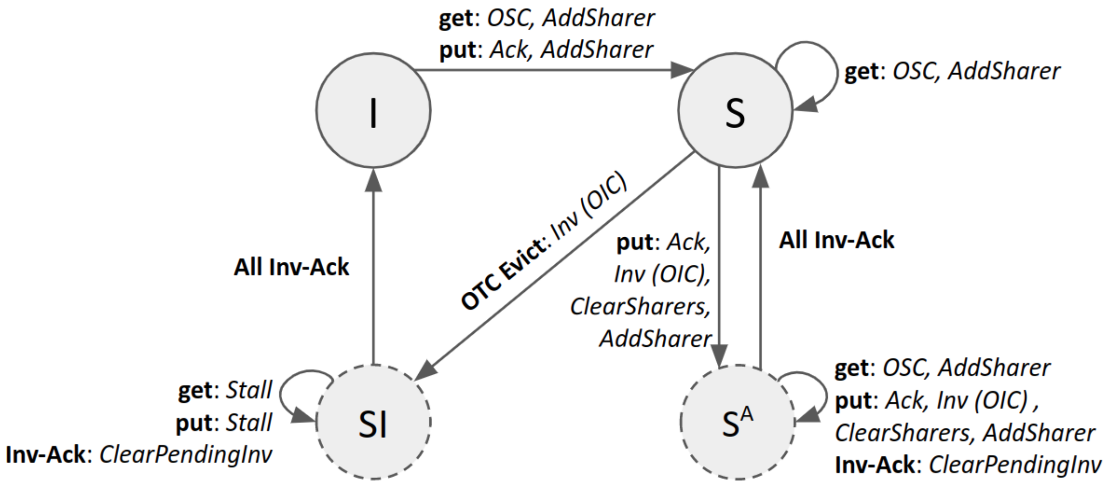

Figure 4.6 *OTC (Directory): Complete coherence protocol. I, S are stable states; SI,* $\mathtt{S}^{\mathtt{A}}$ *are transient states*

The protocol: OTC uses a simplified coherence protocol with Shared and Invalid stable states, avoiding the Modified and the Exclusive States. This is in line with the CXL specification as it flexibly allows implementations to use fewer stable states in the protocol. (We discuss further details of the CXL protocol in Sec. 4.5.1.) Most coherence protocols involve transient states since transition from one stable state to another is not typically atomic [134]. Āpta re-purposes a transient state to account for the asynchronous invalidations. Fig. 4.6 illustrates the transition diagram for the OTC directory controller (with events and actions on stable and transient states). For ease of explanation, this protocol assumes that FaaS applications are race-free, i.e., no `put` or `get` can occur during an ongoing `put`. (However, the actual Āpta protocol can handle buggy FaaS applications with races as well.) A `put` event is triggered on the completion of an object persistence transaction by the OPC; a `get` event is triggered at the beginning of an object serving request by the OSC. An “Invalid” state for an object implies it is not cached in any compute server. A “Shared” state implies the object is cached in readable state in one or more compute servers. The transient state $\mathtt{S}^{\mathtt{A}}$ signifies that the new version of the object is cached in **S**hared state in one or more compute servers and there are pending invalidation-**A**cknowledgments from one or more compute servers for the old version of the object.

Suppose the directory receives a `put` for an object currently in shared state. Once the `put` transaction completes, the directory performs 3 actions in parallel

- acknowledges the write, notifies the object invalidation controller (OIC) to send invalidation messages to all prior sharer compute servers, clears the old sharer vector and adds the compute server that requested the `put` as a sharer for the new version of the object (Recall the caching policy is write-allocate - compute server retains the object after a `put`). The directory then transitions to $\mathtt{S}^{\mathtt{A}}$ until it receives all invalidation-acknowledgments. While in this state the directory can still service a `get` or `put` request for the object. For a `get`, the OSC responds with the latest object version, satisfying Lemma #1. Once OIC notifies that all outstanding invalidation-acknowledgments are received, the directory transitions back to shared state for the object.

The organization: The directory is organized as a standalone, set-associative directory cache structure at object granularity. When a directory cache set is full, the directory controller evicts a cold, shared object in the set. It issues an invalidate to all compute server sharers that cache this object and transitions to transient state `SI`. Any requests to the object while in this state, are stalled until all invalidations are acknowledged and an entry becomes available. Sizing the directory cache appropriately and correctly identifying cold objects can ensure that operations can continue at line-rate, without stalling.

Object Invalidation sub-controller (OIC):
≻ *Function:* Invalidates stale objects cached in compute servers.
The OTC requests the OIC to invalidate an object triplet (`objID`, base mPA, size) on a set of compute servers. OIC issues invalidation messages to the compute servers for all the object's cache lines. At the compute server, the challenge however is translating the mPA address of the object to cPA address to issue invalidations to caches. This is achieved using an efficient object based reverse mapping [38], implemented in Linux for reserve mapping virtual memory (rmap chains)[5]. This reverse mapping is used by system calls like mmap, munmap, madvise etc. However, this is an expensive software call (measured to be ∼1.4μ`sec` per call) invoked by GET controller using interrupts and adds significant time overhead. Recall, in Āpta, invalidations are out of the critical path and hence this does not affect performance. Finally, the GET

[5]Originally an object in [38] referred to a memory mapped file which maps a range of data to a range of physical addresses. This works very well for our purposes since FaaS objects are also allocated contiguously within a range.

controller sends invalidation-acknowledgments.

The OIC tracks the number of invalidation-acknowledgments that are outstanding from each compute server using a counter. It notifies the OTC when all the invalidation-acknowledgments are received.

4.3.4.2 Control-plane software

Āpta modifies existing FaaS runtime control-plane software [101]. This software runs on the low-power SoC of the memory server. We outline the changes required in two of these components and describe their interface to the Āpta hardware controllers.

Executor Manager (EM):
≻ *Function:* Responsible for scheduling and tracking the execution of the state machine workflow of FaaS applications.
EM selects a suitable compute server to schedule a function invocation and passes the invocation parameters to the function sandbox. EM scheduler is guided by the performance and correctness criteria when scheduling function invocations.

≻ *Hardware interface:* When scheduling functions, if the set of all objects to be accessed by the function is unknown (not declared), the scheduler queries the OIC to exclude all compute servers which have pending invalidation-acknowledgments. If the set of objects to be accessed by a function are declared in the state machine workflow, the scheduler looks up the object in OTC to determine where scheduling can be beneficial (current sharers) and if any compute servers are to be excluded (invalidation-acknowledgment pending). The OIC/OTC would expose hardware APIs for the scheduler to read this information. Note, CXL 3.0 specification recommends that the compute server meet an average latency target of 90ns for responding to invalidation requests. With this response target and round-trip network latency, we expect that invalidations will resolve very quickly and the invalidation-acknowledgement pending nodes at any time will be very short. Therefore, we expect the impact of asynchronous invalidation on scheduling to be minimal.

Object Manager (OM):

≻ *Function:* Responsible for memory allocation and de-allocation of objects in the memory server.

The objects are allocated in mPA in fixed bucket sizes (rounded up to the nearest fixed bucket size). The buckets are allocated as a contiguous physical memory address range, aligned at the cache line boundary in the memory server. This memory allocation strategy is akin to the memcached slab allocator [121] The OM runtime stores an *object mapping data structure* of mPA to unique `objID`, at a fixed location in memory. This data structure is organized as a radix tree followed by a trie[6].

≻ *Hardware interface:* For serving objects, the OSC controller reads the *object mapping data structure* written by the OM runtime (from the fixed location in memory) and responds to the compute server mPA request with object-granularity bulk read semantics.

4.3.5 Putting it all together

Fig. 4.7 illustrates the application state machine workflow of three real world FaaS applications [100, 109] with the objects accessed by each function and annotated with an instance of scheduling decision made by the EM on a cluster of compute servers (C1 to C4) connected to access the Āpta object store.

We walk through the working of Āpta with the sentiment analysis application (Fig. 4.7, App 3) that evaluates customer reviews for products of a company and is triggered when the collated raw reviews file (csv) is uploaded to the object store.

When *read_csv* function on C1 receives a invocation trigger, the `get` call (*rdata = get("raw_data.csv")*) maps *rdata* to the shmem IPC region, located in the DM address range on C1. When *rdata* is accessed, the LLC miss triggers a request to the GET controller. The OSC controller responds with a set of cache

[6]For object mapping, a combination of space efficient radix tree and lookup time efficient trie is used (inspired from page table in virtual memory and longest prefix match in routers, respectively). The radix tree traversal first points to 4KB/2MB page. Within the page, objects are organized as a trie. This organization ensures the data structure can be read in hardware controllers.

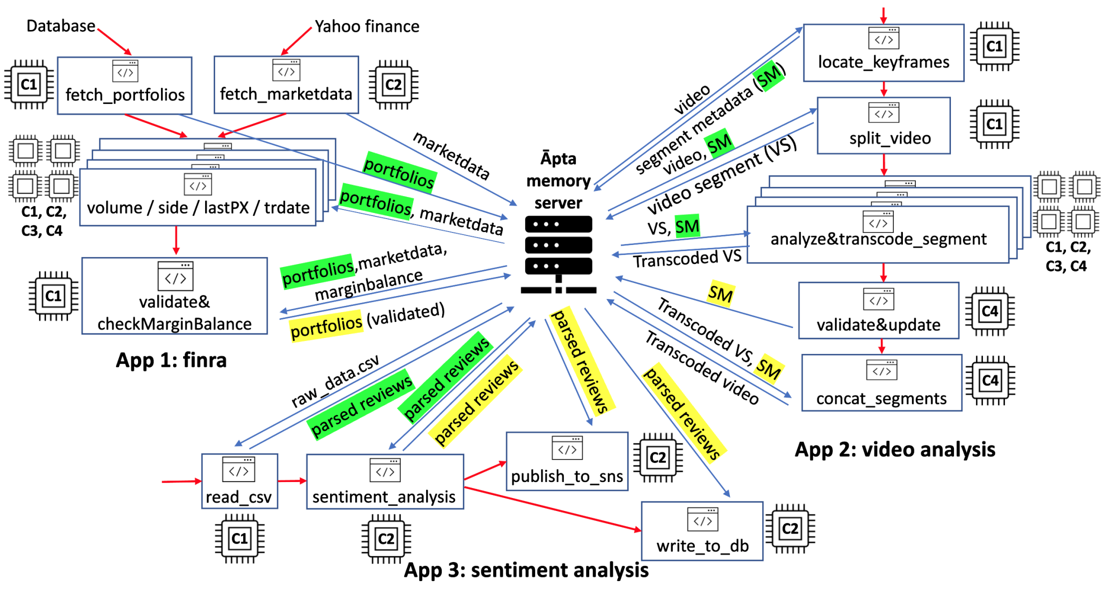

Figure 4.7 *FaaS applications annotated with object store interactions and scheduling decisions; highlight color changes indicate object writes requiring invalidation*

lines of the object. All subsequent accesses to *rdata* in the computation hit in the caches. The `put` call atomically writes *parsed_reviews* object to memory server using hardware transaction between the PUT controller and OPC. C1 caches both *raw_data.csv* and *parsed_reviews* objects and accordingly, the OTC tracks C1 as a sharer of these objects.

Next, the *sentiment_analysis* function, scheduled on C2, similarly performs a `get` on *parsed_reviews*. On access the object is brought into the LLC, making C2 a sharer for the object. After computation, a `put` call writes a new version of *parsed_reviews* to the memory server. The Āpta protocol acknowledges the write from C2 immediately and sends invalidation to C1 asynchronously, tracking C1 as having outstanding invalidation-acknowledgments. When scheduling the next set of parallel functions, the EM checks with OIC and does not schedule the functions on C1 to satisfy *§Scheduling Correctness Criterion*. Scheduling on C2 provides opportunity to exploit locality as it previously executed a predecessor function. Accordingly, *publish_to_sns* and *write_to_db* are scheduled on C2 and both functions benefit from cache hits for accesses to *parsed_reviews*.

4.4 Evaluating Āpta

4.4.1 Evaluation goals

(i) Compare performance of Āpta against the following *state-of-the-art* compute server fault-tolerant systems:

- RDMA-based object store with Faa$t caching [155]: An immutable object caching protocol run using two-sided RDMA verbs over Infiniband. On a `put`, write-through to object store with no sharer invalidation. On a `get`, if cache hit, incur one round trip to object store to ensure cached data is not stale (no object data transferred unless data is stale); if cache miss, read object from remote object store.
- *faster* RDMA-based object store with Faa$t caching: Uses the same interconnect as Āpta for RDMA, along with the above Faa$t software-based object caching protocol. This configuration allows us to isolate the benefits of improving just the underlying interconnect (transport layer).
- *faster* RDMA-based object store with Āpta caching: Uses the same interconnect and Āpta's object caching protocol but in software. This configuration allows us to quantify the benefit of our optimized coherence protocol.
- CXL uncached DM: The cache line granularity CXL DM, that achieves fault-tolerance by disabling caching of any DM data (therefore, requires no coherence protocol). This configuration allows us to quantify performance benefits of caching in DM and object semantic operations proposed for Āpta. This configuration is achieved by using gcc provided intrinsics `void _mm_stream_*()` to bypass the cache. These non-temporal writes "stream" a write from the processor directly to the memory [42]

(ii) Demonstrate the fault-tolerance and resilience of Āpta

(iii) Break-down performance gains for get and put operations (compute time is kept constant for all configurations)

(iv) Evaluate robustness of performance gains with respect to varied interconnect properties and compute server capabilities.

4.4.2 Evaluation methodology

Our evaluation of Āpta is driven by a simulator based methodology (similar to DM proposals [22, 98, 99]). We now set out the workloads and configuration parameters used in the simulation of such a system.

Table 4.3 *FaaS applications evaluated, annotated with schedule*

Application (Patterns; Input data size; Max RSS)	Functions (compute server c1-c3)
PHI data [100] (Broadcast, Pipeline; 20KB; 100MB)	identifyPHI (c1), deIdentify (c2), anonymize (c1), analytics (c1)
Sentiment Analysis [100] (Broadcast, Pipeline; 480KB; 93MB)	readcsv (c1), sentimentAnalysis (c2), publishSNS (c2), writeDB (c2)
FINRA [100] (Broadcast-Gather; 1.2MB; 23MB)	fetchMarket (c1), fetchPortfolios (c2), volume (c1), trdate (c2), lastpx (c3), side (c2), marginBalance (c1)
Video Transcode and Analysis [109] (Scatter-Gather, Pipeline; 2MB; 117MB)	locateKeyFrame (c1), splitVideo (c1), AnalyzeProcess (c1,c2), validate (c3), concat (c3)
Image Prediction [100] (Pipeline; 2.7MB; 357MB)	resize (c1), predict (c1), render (c1)
Serverless GEMM (sparse) [164] (Map-Reduce; 234KB; 943MB)	split (c1), mapper (c1, c2), split (c1), reducer (c1, c2)

4.4.2.1 Workloads

We use 6 full FaaS application workflows, totaling 26 functions, from different domains seen in FaaS - text, numeric, image, video processing. We simulate these full FaaS application workflows from start to finish to demonstrate realistic cache hit rate, invalidations and scheduling decisions. For each application, Table 4.3 shows the communication patterns in the workflow, input data size, constituent functions and a chosen instance of a schedule for an invocation. These applications cover the full range of characterized input dataset/object sizes and function communication and invocation patterns [127, 139, 184]. The applications use local DRAM main memory to store intermediate data, akin to a scratchpad. Table 4.3 shows this measured local memory usage excluding the input object (max resident set size). We report the geometric mean speedup as an aggregate statistic across all applications.

Table 4.4 *Configuration of the Āpta simulated system*

3 Compute Servers	
Processor	single socket, 3.0 GHz; Int/FP Ops: 0.02 CPI
L1 I/D Cache	256KB, 8-way, private per core, 1 cycle
L2 Cache (LLC)	32MB, 16-way, shared, 10 cycles, 128 MSHRs
Local Directory	embedded in L2, fine sharing vector (cores, cachelines)
Local Memory	2 $\times$ 8GB DDR4-2400 MHz, 1 channel
1 shared Disaggregated Memory Server	
Directory (OTC)	20-cycle, coarse sharing vector (compute servers, objects)
Memory	2 $\times$ 8GB DDR4-2400, 1 channel
Interconnect	
Latency & Bandwidth	point-to-point, 500ns, 80Gbps full-duplex

4.4.2.2 System configuration

We model a DM system with four servers (3 compute servers and 1 memory server). Each compute server has a single socket CPU with local DRAM memory. The CPU has per-core L1 and a socket-shared L2 cache, kept coherent with a directory-based MOESI protocol. Within the memory server, we simulate DDR4 DRAM memory, along with the Āpta controllers. The compute servers connect to the DM server with ordered point-to-point links of a fixed latency and bandwidth (full system config in Table 4.4).

The RDMA configurations are measured on same hardware as in Sec. 4.2. To model futuristic, faster RDMA ($\text{RDMA}_{\texttt{f}}$ - running over the same PCIe gen5 interconnect as Āpta), we add the latency overheads incurred for using RDMA operations and software coherence protocol to Āpta's network latencies, as an approximation. Object get/put operations in a key-value store using RDMA to read/write data from/to remote memory are an average of 2.2$\times$ slower than CXL [62]. Above this, fault-tolerant coherent object get/put operations require employing key-value stores like Faa$t and Hermes [93], which use complex two-sided RDMA, adding even more latencies. We measured this as the latency difference between a write in Hermes and a one-sided RDMA write ($\approx$14 μsec per call, fixed overhead irrespective of object size). We use these overhead latencies along with the respective coherence protocol actions to simulate RDMA$_{\texttt{f}}$+Faa$t and RDMA$_{\texttt{f}}$+Āpta.

4.4.2.3 Simulator setup

We simulate the identical shared memory version of the full FaaS applications written in python, compiled down to C. We generate traces of these programs using the Prism framework [140], which uses Valgrind to generates traces of compute, memory, thread create/join and barrier events. The tool produces synchronization and dependency-aware, architecture-agnostic traces. These traces are manually annotated with FaaS phases of execution i.e., get/compute/put.

We replay the traces in a modified gem5 simulator [157]. We implement the proposed inter-server Āpta coherence protocol and its hardware controllers. OSC and OTC lookup incur latency of 20 cycles each, modeled on average address translation and directory lookup latencies in modern processors [67]. Memory ops are simulated with a detailed memory hierarchy. The replay mechanism uses FaaS phase of execution annotations to apply appropriate memory access characteristics for each phase of execution i.e., caches+local memory for compute, caches+DM for get/put. The get phase of execution bulk loads cache lines of the object over the interconnect (as described in Sec. 4.3.3 $\texttt{c}_1$). The put phase of execution uses the one-phase transaction commit protocol to write-through the modified object cache lines over the interconnect (as explained in Sec. 4.3.3 $\texttt{c}_2$). Integer and floating point ops are simulated with fixed CPI. We use an aggressive CPI and larger, lower latency L1/L2 caches to represent execution with optimized libraries (as in Sec. 4.2.1). This simulator setup speeds up the computation phase of FaaS functions by 5$\times$ geomean compared to unoptimized (single thread) python functions run on a Intel i7-9700K machine. (Note this is conservative as Intel python extensions provide 200$\times$ speedup for scitkit-learn, 90$\times$ for pandas, 3$\times$ for tensorflow [76].)

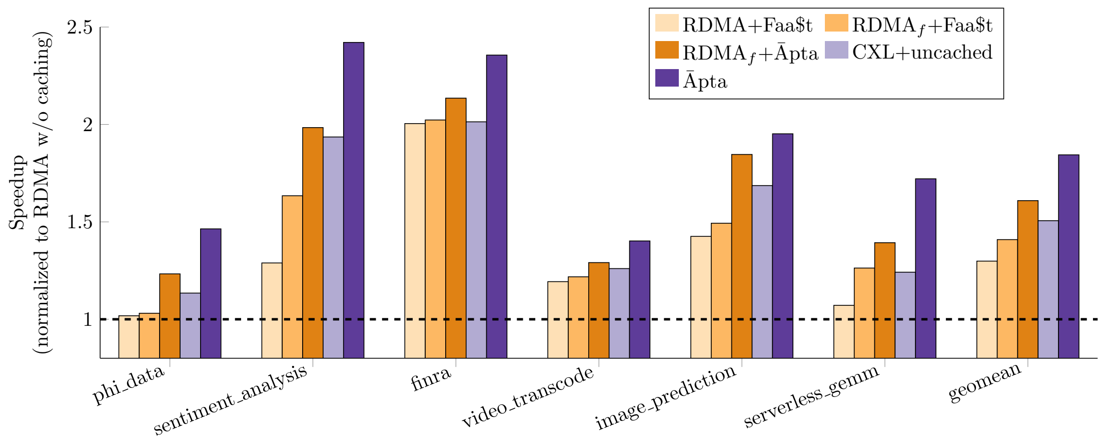

Figure 4.8 *Performance comparison of all configurations, normalized to RDMA without caching, for 6 full FaaS applications comprising of 26 FaaS functions*

4.4.3 Evaluation results

4.4.3.1 Performance benefit and analysis

Fig. 4.8 shows the performance of all configurations normalized to a baseline RDMA-based object store without caching.

Result 1: Āpta provides 42% geomean speedup over state-of-the-art RDMA+Faa$t. This performance gain comes from three sources: (a) improved network, (b) optimized Āpta coherence protocol, and (c) using hardware controllers for object access and coherence in DM. The RDMA$_\mathtt{f}$+Faa$t configuration provides 7% performance improvements over RDMA+Faa$t, showing the performance gains from just the improved network. Next, employing Āpta's coherence protocol over RDMA$_\mathtt{f}$ (RDMA$_\mathtt{f}$+Āpta) provides further 12% improvement over the previous RDMA$_\mathtt{f}$+Faa$t, showing the performance gains from our optimized coherence protocol. Finally, Āpta's use of DM hardware-controllers eliminates the irreducible software overheads of RDMA$_\mathtt{f}$, thereby providing 18% higher performance than previous RDMA$_\mathtt{f}$+Āpta.

Result 2: Āpta provides 24% geomean performance gain over CXL-uncached by using a fault-tolerant object caching protocol and object semantic reads/writes. Note that employing the CXL-uncached object store will perform worse than a faster RDMA-based object store with caching (RDMA$_\mathtt{f}$+Āpta), emphasizing the need for Āpta's design in a DM system.

Result 3: We also evaluated the performance against the non fault-tolerant cached CXL DM. Āpta provides 10% geomean speedup over this CXL-cached system (not shown in graph). This shows that there is no performance cost to Āpta's fault tolerance; in fact, Āpta shows a small improvement in performance because it addresses CXL's inefficiencies owing to its cache line granular accesses.

4.4.3.2 Fault-tolerance validation

We verify the complete Āpta protocol (with additional states to handle races, if any applications misbehave), in the Murϕ model checker [40] and exhaustively verified for liveness (deadlock-freedom) and safety (linearizability). Importantly, we also model check to prove correct and non-blocking behavior in the presence of sharer compute server failures. These Murϕ model files for the protocol are available online[7].

Result 4: Because the Āpta protocol does not wait for acknowledgments in the critical path, it has the potential for lower tail latencies. To measure this, we run the applications 50 times under variable network latencies to reflect real world rack scale networks [148] and measure the standard deviation of execution times. The network requests experience a random latency within a Gaussian distribution (40% variation around the mean as measured for an Infiniband network [92]). On average, applications exhibit 32% lower standard deviation of execution time with Āpta compared to the non fault tolerant CXL-cached system, demonstrating the resilience of Āpta.

4.4.3.3 Performance Break-down

For the simulated schedule, Table 4.5 shows the number of `get`s which hit in the cache (compulsory cache miss for first `get` request on all compute servers, while subsequent `get`s may potentially hit in the cache) and the number of `put`s that require sharer invalidations (these `put`s jeopardize DM system availability and increase latency with the blocking CXL-cached protocol).

[7]https://github.com/adarshpatil/apta

Table 4.5 *Analysis of `get` and `put` characteristics for FaaS application execution*

App ↓ / Num. of →	`gets` (cache hit,miss)	`puts` (no inv, with inv)
PHI data	4 (2,2)	4 (4,0)
Sentiment Analysis	4 (2,2)	2 (1,1)
FINRA	5 (3,2)	3 (2,1)
Video Transcode	5 (2,3)	6 (5,1)
Image Prediction	3 (2,1)	3 (3,0)
Serverless GEMM	6 (3,3)	6 (6,0)

Result 5: Āpta lowers geomean get latency by 90%, compared to RDMA+Faa$t's 57% reduction and CXL-uncached 71% reduction over baseline. Fig. 4.9, left shows the total `get` latency, normalized to baseline for each application. Although, both caching mechanisms (RDMA+Faa$t and Āpta) see same cache hit rate, Āpta lowers geomean get latency by using an improved protocol and the DM interconnect.

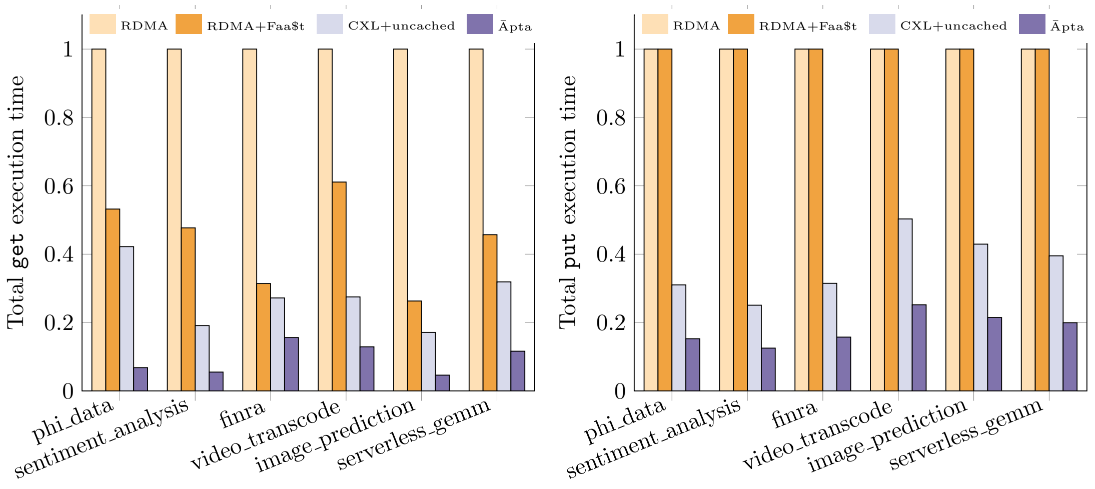

Figure 4.9 *Object `get` and `put` latencies, normalized to RDMA without caching*

Result 6: Āpta achieves the highest 81% reduction in geomean put latency, compared to 63% reduction for CXL-uncached. Fig. 4.9, right shows the baseline normalized total `put` latency. Since a `put` operation always writes through to the object store, RDMA and RDMA+Faa$t see the same `put` latencies. Āpta achieves the reduction by using optimized hardware transactions over an improved DM interconnect.

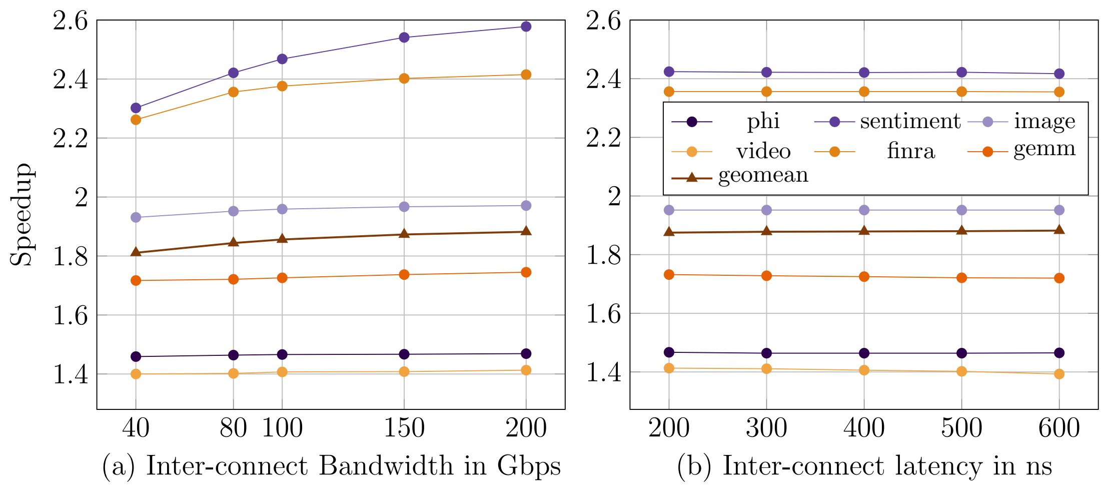

Figure 4.10 *Speedups of Āpta over baseline for varied interconnect characteristics (a) bandwidth and (b) latency*

4.4.3.4 Sensitivity studies

Āpta performance gain is subject to the compute-to-communication ratio of the application. Therefore, we study the performance sensitivity due to variations in interconnect characteristics and computation capabilities.

Result 7: The performance of Āpta improves with increase in bandwidth of the DM network, seeing 90% geomean speedup over baseline for 200Gbps. Interconnect latency has a smaller impact on the performance of Āpta. Āpta still provides a 84% geomean speedup with high latencies of 600ns, as expected for CXL switched fabrics. Fig. 4.10 a & b summarizes the speedups of Āpta over baseline, for varied interconnect network latencies and bandwidth.

Result 8: Āpta's performance gain marginally reduces to 78% geomean with lower capability compute cores, as computation segment latency dominates in the total execution time. Fig. 4.11 shows the speedups obtained as we vary integer and floating point operation CPI of the core for both baseline and Āpta.

4.4.4 Evaluation summary

Āpta provides performance gains over all types of FaaS applications – from communication to compute heavy, applications with high object reuse and those

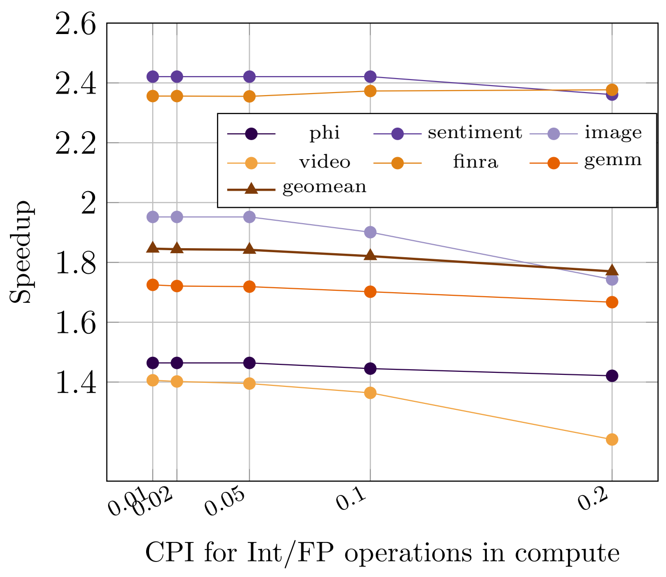

Figure 4.11 *Speedups of Āpta over baseline for varied computation capability, represented by varying integer and floating point CPI (cycles per instruction)*

with lower reuse, applications with serial and multiple parallel functions and over a range of object sizes.

4.5 Discussion

4.5.1 Specifics of CXL support for Āpta

Āpta's design introduces only minimal changes to the CXL protocol and the servers. We now discuss (i) the precise CXL protocol leveraged to design Āpta and (ii) changes needed to the CXL protocol specification to realize Āpta's fault-tolerance benefit. We refer to relevant sections in the CXL 3.0 specification [34] in the discussion.

Using CXL.mem protocol for pooled shared memory: CXL 3.0 specification defines the creation of a pooled memory device where multiple compute servers are configured to access a single memory region concurrently - called “shared FAM" (fabric attached memory) (Sec. 2.4.3). Āpta designates the coherency model for the shared FAM as “hardware coherency with a directory"

(Sec 2.4.4 [8]) and builds over the CXL.mem hardware coherence protocol (Sec 3.3). While CXL does not specify the detailed implementation of the directory, it does allow for tracking fewer states per cacheline i.e., 2 or 3 states instead of the original 4 state MESI protocol (Sec 3.3.3 Implementation Note). Accordingly, Āpta encodes the state of the cacheline using I and A stable states as per the CXL.mem parlance and implements the directory logic in the OTC. To send invalidations and receive invalidation-acknowledgments to/from compute server caches, OTC uses Back-Invalidation Snoop (BISnp) and Back Invalidation Response (BIRsp) messages (Sec 3.3.7, Sec 3.3.8), sent over dedicated S2M BISNP and M2S BIRSP channels (Sec 3.3.2).

Permitting asynchronous invalidation in CXL.mem protocol: The CXL 3.0 specification clearly defines the blocking behavior of BISnp requests (Sec 3.3.3). The synchronous invalidation behavior is further reinforced in the ordering rules (Sec. 3.4, Table 3-50 and Appendix C.1.2). Āpta requires the synchronous BISnp condition to be relaxed in the specification. This would allow implementations like Āpta to respond to write requests and other read requests immediately without waiting for the invalidation to complete.

4.5.2 Generality of proposed hardware

Designing controllers for CXL memory is currently under active development. CXL is being investigated to provide persistent memory[114], pooled remote memory to expand memory capacity [108], near-memory accelerators on CXL [70] and dynamic tiered memory [73]. Efficient implementation of these designs requires controllers for data persistence, address translation and data coherence. Āpta's controllers OPC, OSC and OTC basically provide the aforementioned services and can be adapted to suit these and other emerging application.

The design of Āpta brings *software oriented key-value stores* (employ get/put API to retrieve/store data) closer to traditional *hardware supported shared memory* (use loads/stores to memory addresses). Āpta enables a hybrid memory

[8]CXL 3.0 permits the coherency model of the shared FAM to be either hardware coherency or software-managed coherency.

system with an interface similar to that of shared memories along with the flexibility of granularity provided by key-value stores. The additional flexibility is enabled by the several controller components in the design, which can find wider applicability for other use-cases as well [70, 73, 114].

Āpta enforces coherence at an object granularity, thereby providing a strong consistency model for the DM. Āpta's per-object linearizable reads and writes consistency model is similar to today's production object stores like S3 [21], thus making it easy for adoption without needing any FaaS program changes.

Concerns over use of write-through caches: Note that every write is not written through; writes to cache lines that make a `put` are written through only at the end of each function. Such a policy is inevitably necessary for maintaining high availability and strong consistency in the presence of compute server failures, albeit at the expense of some bandwidth. It is worth noting that existing works [21, 155] have also employed a similar policy.

4.5.3 FaaS scheduler deep-dive: Case study Kubernetes

How does Āpta interface with the scheduler? Using Kubernetes as a case-study we address this question.

FaaS schedulers are complex frameworks that correctly and efficiently schedule functions on compute nodes. Notably, several custom built [50, 111, 116] and cloud provided [4, 60, 126] frameworks exist.

The *kube-scheduler* component of Kubernetes is an example of such a scheduler. It considers several factors like individual and collective resource requirements, hardware/software policy constraints, affinity/anti-affinity specifications, inter-workload interference etc., when making scheduling decisions [101]. The scheduler has two cycles: a serial scheduling cycle and a parallel binding cycle, with each cycle consisting of multiple stages. Āpta can use existing stages in the serial scheduling cycle to achieve its objectives. Specifically, the pre-filter stage can query the OIC and remove invalidation pending compute servers from scheduling decisions (for correctness). The pre-score stage can add affinity labels to nodes with locality which can then be used in the score stage to preferentially select these nodes.

With accelerated FaaS function executions, as achieved in this work, we believe that the Kubernetes scheduler would become the new bottleneck. This is because the current design of Kubernetes provides far lower throughput than what is required to operate such high-performance systems at high efficiency. Improving the scheduler is currently a subject of active research in the community and is left for future work.

4.6 Related work

Resilient coherence protocols: A class of works [1, 45, 46] design coherence protocols that can tolerate dropped and faulty messages. They reissue requests on a timeout to recover, but crucially assume all participants are alive. Āpta is the first work to handle complete node failures. The CXL specification also does not directly specify timeouts but instead recommends that components meet latency targets for various CXL Transactions (Table 13-2 of the CXL spec [34]). Additionally, designing suitable timeouts for transactions in distributed systems is a complex problem [102].

FaaS applications: A number of works [35, 96, 117, 181, 182, 192] composed function benchmarks and software stacks employed in FaaS platforms. They also demonstrate several FaaS inefficiencies: data communication, cold start etc. Āpta addresses a chief inefficiency of FaaS – data transfer overheads and provides a fault-tolerant DM system for FaaS applications.

Reducing communication overheads in FaaS: Several works [115, 133, 155, 165, 169, 186] aim to provide software-based caches at compute servers to cache objects. These works reinforce the potential of caching to improve performance despite being connected by fast networks. Āpta provides software transparent object caching using CXL-based DM.

Faastlane [100], SAND [2] co-locate functions within an application (restricting scheduling) as threads/light-weight contexts to use local shared memory for low communication latency. Āpta allows function processes, to access shared objects from local caches if co-located, but critically also permits flexible scheduling, anywhere in the DM system.

High performance remote memory: Numerous works [36, 41, 54, 64, 141, 163] have used RDMA interconnects to provide software-based remote memory for applications. Āpta overcomes inefficiencies of RDMA by using DM, providing the highest performance remote memory. MIND [105] accelerates RDMA remote memories with in-network coherence and memory management. Analogously, Āpta designs a hardware coherence protocol for a DM object store but crucially enforces availability in the presence of compute server failures.

In the cloud, RDMA is still not widely available. Cloud providers such as Microsoft [12, 152] and Alibaba [53] have shown interest in researching various techniques, pros and cons of deploying RDMA in the cloud. In production, Amazon AWS offers Elastic Fabric Adapter (EFA) [10] as an alternative to RDMA over InfiniBand. EFA is a network interface for Amazon EC2 instances that enables customers to run applications requiring high levels of inter-node communication at high performance. EFA runs an Amazon proprietary network protocol on top of commodity ethernet. However, EFA has been found to have several performance drawbacks - higher latency and lower bandwidth than RDMA [197]. Āpta demonstrates that CXL can be employed to provide better high-performance networks in the cloud, avoiding the several issues that exist in making RDMA cloud viable.

For the programming interface, the recommended interface for cloud applications to access EFA is libfabric [80]. It defines a unified communication API for high-performance distributed applications and abstracts several diverse technologies like MPI, SHMEM, PGAS etc. It also supports multiple network communication backends. Āpta can easily integrate with libfabric using the SHMEM interface and be made available to cloud applications.

Disaggregated memory: This line of work use hardware-supported operations [110] to provide remote memory. Clio [65] defines explicit virtual memory API calls for processes on compute servers to allocate, read/write and synchronize accesses to the DM. COARSE [188] uses DM to accelerate distributed deep learning training. Kona [22] and DM prototypes [107, 149] create per-server private regions on the memory server to allow compute servers to transparently extend their memory capacity. While Āpta shares some common objectives, it builds on top of a CXL 3.0 DM system, specializes it for FaaS, and ensures availability in the face of server failures.

4.7 Summary

In this chapter, we have observed that upcoming CXL-based DM systems can alleviate the communication bottlenecks of cloud-based FaaS applications but lacks the necessary fault-tolerance to operate in a failure-prone datacenter. We have proposed Āpta, a CXL-based DM system for maintaining FaaS objects that provides efficient object-granular access and allows fault-tolerant caching of objects in compute servers caches, without compromising consistency. Thus, Āpta has showcased for the first time a fault-tolerant cloud use-case for CXL-based coherent disaggregated memory.

5
Conclusion and future work

In this chapter, we first summarize the main contributions of this thesis. Next, we critically analyze the design decisions presented. Finally, we explore avenues for future work before concluding.

5.1 Summary of contributions

In Chapter 3 we motivated the need for improved DRAM reliability in modern memory systems to cope with increasing error rates and new models of failure, as reported in field studies. We reviewed existing mechanisms and saw that the incremental techniques employed are not scaling well for the increased error rates. Further, these techniques are incurring increased capacity overheads and causing performance penalties.

We addressed these shortcomings by proposing a unique design point - Dvé. Dvé offers higher memory reliability and performance using coherent replication of data. Coherent replication builds on top of existing protocols to not only maintain the replicas in sync (as required for reliability), but also provide coherent access to both of the replicas during common-case fault-free operation (for improved performance). Further, Dvé can be deployed flexibly on-demand to provide the gains, when underutilized memory capacity is available or when required by applications.

We analytically quantified Dvé's reliability benefit and showed that it provides lower uncorrectable and undetectable error rate than state-of-the-art reliability schemes. The results of our experimental evaluation showed that Dvé provides significant performance improvements for several datacenter applications.

In Chapter 4, we motivated the need for fault-tolerance and improved performance to execute applications from a new datacenter application paradigm - Function-as-a-servce (FaaS). FaaS applications are by design compute server fault-tolerant and are written as a composition of stateless, idempotent functions. However, they are severely bottlenecked by the remote object store where the objects (state) are maintained.

To eliminate these performance limitations, we proposed to use a CXL-based cache-coherent disaggregated memory to hold FaaS objects. CXL enables low-latency, high-bandwidth access to remote memory and compute server side caching of data, but lacks the requisite level of fault-tolerance necessary to operate at an inter-server scale within the datacenter. Our proposed design Āpta provides efficient object-granular accesses and allows fault-tolerant consistent caching of objects in compute servers caches. Āpta's innovation is a novel fault-tolerant coherence protocol that removes invalidations from the critical path and uses coherence aware scheduling to guarantee availability in the face of server failures.

Our experimental evaluation of Āpta using representative full FaaS application workflows showed that, compared to state-of-the-art systems, Āpta provides highest performance and the highest availability in the presence of compute server failures.

5.2 Critical analysis and takeaways

We now re-examine the design decisions presented in this thesis and scrutinize them with the benefit of hindsight.

Critical analysis: While the two systems proposed achieve the goals set out in the introduction, they however add sizeable *software complexity* to enable their design. Case in point, Āpta puts the onus of correctness on the scheduler which would now require full formal verification to validate correct behavior in all cases; Dvé relies on the OS memory manager to map replica addresses, introducing additional memory management tasks. We have demonstrated the practicality of the designs by showing similarity to existing implementations. Notwithstanding, these software frameworks are complex systems with several heuristics and introducing the necessary changes would require further detailed considerations.

Secondly, the true cost of the complexity will be evident when the system is deployed *at scale*. For instance, with hundreds of co-located FaaS functions executing concurrently, each compute server could be requesting several objects in parallel and might have pending invalidations due to any one of the executing functions. This could affect performance and scheduling decisions. When scheduling subsequent functions, if the set of all objects to be accessed by the function is unknown[1], the scheduler would conservatively exclude all compute servers which have pending invalidation-acknowledgments, causing application executions to stall. Additionally, the scheduler itself might also hit scalability bottlenecks when running such optimized functions with short runtime i.e., small compute and communication latencies. Increasing the schedulers throughput remains a challenging problem. Due to infrastructure limitations, we have not evaluated the design at a larger scale.

In summary, the designs proposed in this thesis improved system reliability and performance by using intelligent coherence protocols. However, adversarial scenarios exist where providing higher reliability can still impact performance. More importantly though, even in such worst-case scenarios, the reliability of the proposed designs would not be any worse than the baseline designs compared against.

Takeaways: We note a few key takeaways from the works in this thesis.
$\succ$ Robust reliability is key for next-generation memory.
DRAM memory technologies are being specialized for various compute paradigms (DDR, LPDDR, GDDR, HBM etc.) and each has architecture has different and varied implications on fault models and design of reliability solutions. Dvé takes a first step in architecting a technology agnostic, DRAM reliability solution. Secondly, with CXL-based hardware disaggregated memory, newer fault models have come to the fore. Āpta identifies one of the possible fault models and

[1]The support to specify function input objects in the workflow currently exists but is optional.

shows a lightweight solution to this. Several more exist and more work needs to be done to improve the RAS for such architectures.

≻ Application driven architecture.
The thesis makes the case that better computer architecture and system design can be achieved through understanding the characteristics of applications. Dvé demonstrates that a good understanding of applications memory access characteristics in NUMA is the *raison principale* for improved performance i.e., the ability to exploit performance by reading the replica for the read-only/read-mostly data. Similarly, Āpta's design was enabled by studying FaaS application communication and execution patterns which allowed the defining of favorable scheduling criteria. Secondly, applications paradigms for executing in the data center have evolved to be compute server fault tolerant. Similarly, hardware coherence protocols must advance to match this model without "blocking".

≻ Revisiting design decisions in-step with advances in technology.
Computer architecture is heavily driven by advances in electronic and semiconductor technology. As evidenced with CXL, the next generation PCIe bus in alternate protocol mode creates a new interconnect capability allowing for coherent inter-node communication. Understanding and exploiting the capabilities of this was the key cog that allowed us to design Āpta. In a similar spirit, we would need to revisit all other techniques used in a shared memory in the context of this increasingly decoupled, distributed hardware components e.g., synchronization atomics.

≻ The end-to-end arguments to system design [156] is a powerful principle to employ for current and future system designs. Employing this approach brings a fresh perspective to conventional problems and yields innovative solutions.
As a demonstration, DRAM error mitigation solutions over many years, have continually employed an incremental, bottom-up approach to protect against errors rising up in memory hierarchy. Dvé's holistic approach allows robustly handling a wide variety of errors arising from faults in any DRAM components. In the same vein, CXL only specifies what the underlying interconnect can achieve and is primarily concerned with defining its capabilities. Standards and specifications should also strive to define application use cases to employ the technologies prescribed. This will not only aid in accelerating adoption but also

create a dialogue to allow for better end-to-end system design.

≻ Complexity in system design can be a daunting challenge but can be tamed through modularization.
Case in point, to architect an object store, the Āpta system provides a multitude of features for DM. These features are provided through multiple independent modules, each dedicated for a single purpose. The modules individually can also be generalized for use in other applications currently being investigated for CXL - persistent memory [114], memory capacity expansion [108], near memory accelerators on CXL [70] and dynamic tiered memory [73].

To summarize, the need of high-reliability, high-performance systems has never been greater than it is now. Equally, the array of technologies available for designing an ideal system has never been boarder because of the rapid advances in technology, speed of processors, memory, and interconnection of components. The true challenge therefore, is in designing a balanced system according to Amdahl's Law.

5.3 Future work

The design space of datacenter memory systems remains an active area of research and there are several directions possible to advance the field. Below we briefly discuss some potential possible avenues to explore and consider how our ideas might be expanded to different settings.

Value-added accelerated DM: CXL-based DM is still in its nascency and several new application use cases of such an architecture are being explored. In these applications, various hardware-based *value-added* services like security, encryption, compression etc. can be architected in hardware using the CXL controllers on the memory server. The trend of value-added accelerated services is already prevalent in today's systems e.g., DRAMs perform processing in-memory [180] and processors embedding several accelerators in core [77]. We believe that the insight presented in designing Āpta's controllers can serve as a useful insight in this direction.

Reliable and available DM: The discontinuation of Intel Optane devices has left byte-addressable non-volatile memory (NVM) based applications to using CXL-based DM as the only viable way to achieve data persistence. Unlike local NVM based devices which only protected against power failure, the use of DM for persistence automatically brings with it the property of *availability* i.e., the data is accessible and consistent immediately on the failure of the compute node. Such available persistence devices must be hardened to be *reliable*, since memory servers too are susceptible to failures. Future work can involve defining protocol actions to enable replicated DM.

Redesigning distributed co-ordination services for modern hardware: Distributed datacenter applications use several services like Kubernetes scheduler, Chubby lock service, Zookeeper configuration service which use protocols like Raft, ZAB etc over general purpose networks like Ethernet to provide replicated highly-available co-ordination APIs. As datacenters adopt faster network technologies (like RDMA, CXL) and specialized hardware (like replicated reliable memory, hardware offloading), the design of these distributed co-ordination services needs to be revisited. Future work can investigate the efficient and performant re-design of these services taking into account modern hardware primitives like replicated reliable memory. We believe this thesis has paved and taken the first step in this direction with Āpta.

Consistency-directed shared DM: This thesis explored memory sharing DM between homogeneous CPU based servers. Going forward, we expect to see heterogeneous systems with strong memory consistency enforcing CPUs and relaxed memory consistency accelerators like GPUs, DPUs etc., use the shared DM to operate collaboratively on a shared dataset in memory. Simply using the current SWMR-based coherence protocols would be an inefficient way to ensure memory consistency. Future work can explore efficient consistency-directed coherence mechanisms e.g., temporal coherence in which FENCE instructions wait/stall, rather than every writer, for invalidations to complete or alternatively release-consistency directed coherence with special actions on acquire and release instructions.

5.4 Concluding Remarks

This thesis explored the co-design of performance and reliability for datacenter memory. We argued that this can be achieved using well-designed coherence protocols. To support this claim, we built two designs to show significant improvements in both performance and reliability for two representative organizations of memory in datacenter systems. First, using a coherence protocol for replicating data across two processors in a NUMA system can improve DRAM reliability and performance. Second, architecting a fault-tolerant CXL protocol based disaggregated memory system can accelerate Function-as-a-service applications in the datacenter.

A common underlying thread in the works presented is a holistic understanding of the entire system, to achieve better overall designs. We foresee that the demand for improved reliability in the datacenter will continue to grow, and we hope that this thesis will motivate the designs of performant, reliable next generation memory systems. .

Bibliography

[1] Konstantinos Aisopos and Li-Shiuan Peh. A systematic methodology to develop resilient cache coherence protocols. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 47–58, 2011.

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. Sand: Towards high-performance serverless computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, USA, 2018. USENIX Association.

[3] Amazon. AWS S3. https://aws.amazon.com/s3/.

[4] Amazon. AWS Step Functions. https://aws.amazon.com/step-functions/.

[5] Amazon AWS. Make a lambda function idempotent. https://aws.amazon.com/premiumsupport/knowledge-center/lambda-function-idempotent/.

[6] AMD. KDG for AMD NPT Family 0Fh Processors, 2009.

[7] AMD. KDG for AMD Family 15h Models 00h-0Fh Processors, 2013.

[8] Andy Rudoff, Intel Corporation. Persistent memory on cxl. https://www.snia.org/educational-library/persistent-memory-cxl-2021, 2021.

[9] ARM HPC. arm-hpc/comd. https://github.com/arm-hpc/CoMD.

[10] Amazon AWS. Elastic fabric adapter. https://aws.amazon.com/hpc/efa/.

[11] Amazon AWS. Error handling and automatic retries in aws lambda. https://docs.aws.amazon.com/lambda/latest/dg/invocation-retries.html.

[12] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Il-

ias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering azure storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, Boston, MA, April 2023. USENIX Association.

[13] Rajeev Balasubramonian. *Innovations in the Memory System*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2019.

[14] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2018.

[15] S. Basu and J. Torrellas. Enhancing memory use in simple coma: Multiplexed simple coma. In *HPCA*, 1998.

[16] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, and S. McIntosh-Smith. Unprotected computing: A large-scale study of dram raw error rate on a supercomputer. In *SC*, 2016.

[17] Majed Valad Beigi, Yi Cao, Sudhanva Gurumurthi, Charles Recchia, Andrew Walton, and Vilas Sridharan. A systematic study of ddr4 dram faults in the field. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 991–1002, 2023.

[18] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[19] Richard E. Blahut. *Algebraic Codes for Data Transmission*. Cambridge University Press, 2003.

[20] Spyros Blanas. Near data computing from a database systems perspective. https://www.sigarch.org/near-data-computing-from-a-database-systems-perspective.

[21] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in amazon s3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2021. Association for Computing Machinery.

[23] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. In *ISCA*, 2005.

[24] CCIX consortium. http://www.ccixconsortium.com.

[25] S. Cha, O. Seongil, H. Shin, S. Hwang, K. Park, S. J. Jang, J. S. Choi, G. Y. Jin, Y. H. Son, H. Cho, J. H. Ahn, and N. S. Kim. Defect analysis and cost-effective resilience architecture for future dram devices. In *HPCA*, 2017.

[26] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[27] Hsing-Min Chen, Carole-Jean Wu, Trevor Mudge, and Chaitali Chakrabarti. Ratt-ecc: Rate adaptive two-tiered error correction codes for reliable 3d die-stacked memory. *ACM Trans. Archit. Code Optim.*, 13(3), September 2016.

[28] L. Chen and Z. Zhang. Memguard: A low cost and energy efficient design to support and enhance memory system reliability. In *ISCA*, 2014.

[29] W. Chen, K. Ye, Y. Wang, G. Xu, and C. Xu. How does the workload look like in production cloud? analysis and clustering of workloads on alibaba cluster trace. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, 2018.

[30] R. Chien. Cyclic decoding procedures for bose- chaudhuri-hocquenghem codes. *IEEE Transactions on Information Theory*, 10(4):357–363, 1964.

[31] C. Chou, A. Jaleel, and M. K. Qureshi. Candy: Enabling coherent dram caches for multi-node systems. In *MICRO*, 2016.

[32] Google Cloud. Retrying event-driven functions. https://cloud.google.com/functions/docs/bestpractices/retries.

[33] Compute Express Link. https://www.computeexpresslink.org/.

[34] Compute Express Link 3.0 Specification. https://bit.ly/cxl3-specification, August 2022.

[35] 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 Middle-*

ware Conference, Middleware ’21. Association for Computing Machinery, 2021.

[36] Alexandras Daglis, Dmitrii Ustiugov, Stanko Novaković, Edouard Bugnion, Babak Falsafi, and Boris Grot. Sabres: Atomic object reads for in-memory rack-scale computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[37] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: A holistic approach to memory placement on numa systems. In *ASPLOS*, 2013.

[38] Dave McCracken, IBM. Object-based reverse mapping. https://landley.net/kdocs/ols/2004/ols2004v2-pages-71-74.pdf.

[39] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K. Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[40] David L. Dill. The murphi verification system. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV ’96, Berlin, Heidelberg, 1996. Springer-Verlag.

[41] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, Seattle, WA, April 2014. USENIX Association.

[42] Ulrich Drepper. What every programmer should know about memory. https://lwn.net/Articles/255364/, October 2007.

[43] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of cloudlab. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC ’19. USENIX Association, 2019.

[44] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. A review of serverless use cases and their characteristics. https://arxiv.org/pdf/2008.11110.pdf, 2021.

[45] Ricardo Fernandez-Pascual, Jose M. Garcia, Manuel E. Acacio, and Jose Duato. A low overhead fault tolerant coherence protocol for cmp architec-

tures. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 157–168, 2007.

[46] Ricardo Fernandez-Pascual, Jose M. Garcia, Manuel E. Acacio, and Jose Duato. A fault-tolerant directory-based cache coherence protocol for cmp architectures. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 267–276, 2008.

[47] Moritz Fieback. Dram reliability: Aging analysis and reliability prediction model. https://repository.tudelft.nl/islandora/object/uuid:e36c2de7-a8d3-4dfa-9da1-ac5b7e18614b, 2017.

[48] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA, April 2018. USENIX Association.

[49] G. Forney. On decoding bch codes. *IEEE Transactions on Information Theory*, 11(4):549–557, 1965.

[50] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association.

[51] 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*, ASPLOS 2021, New York, NY, USA, 2021. Association for Computing Machinery.

[52] Azure Functions. Azure functions error handling and retries. https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-error-pages.

[53] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets RDMA. In *18th USENIX*

Symposium on Networked Systems Design and Implementation (NSDI 21), pages 519–533. USENIX Association, April 2021.

[54] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. Scale-out ccnuma: Exploiting skew with strongly consistent caching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[55] Gen-Z consortium. http://genzconsortium.org.

[56] Gen-Z consortium. ZMMU and Memory Interleave. https://genzconsortium.org/wp-content/uploads/2018/05/Gen-Z-MMU-and-Memory-Interleave.pdf.

[57] S. Gong, J. Kim, S. Lym, M. Sullivan, H. David, and M. Erez. Duo: Exposing on-chip redundancy to rank-level ecc for high reliability. In *HPCA*, 2018.

[58] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, ISCA '83, page 124–131, New York, NY, USA, 1983. Association for Computing Machinery.

[59] Google. Cloud object storage. https://cloud.google.com/storage.

[60] Google. Google Cloud Workflows. https://cloud.google.com/workflows.

[61] Google Cloud Functions. Retrying event-driven functions. https://cloud.google.com/functions/docs/bestpractices/retries.

[62] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, Carlsbad, CA, July 2022. USENIX Association.

[63] Graph500. github/graph500. https://github.com/graph500/graph500.

[64] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, USA, 2017. USENIX Association.

[65] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, New York, NY, USA, 2022. Association for Computing Machinery.

[66] Siddharth Gupta, Alexandros Daglis, and Babak Falsafi. Distributed logless atomic durability with persistent memory. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52. Association for Computing Machinery, 2019.

[67] Yuchen Hao, Zhenman Fang, Glenn Reinman, and Jason Cong. Supporting address translation for accelerator-centric architectures. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[68] HP. Advanced memory protection technologies, technology brief, 5th edition. ftp://ftp.hp.com/pub/c-products/servers/options/c00256943.pdf.

[69] C. Huang, R. Kumar, M. Elver, B. Grot, and V. Nagarajan. C3d: Mitigating the numa bottleneck via coherent dram caches. In *MICRO*, 2016.

[70] Wenqin Huangfu, Krishna T. Malladi, Andrew Chang, and Yuan Xie. Beacon: Scalable near-data-processing accelerators for genome analysis near memory pool with the cxl support. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 727–743, 2022.

[71] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic rays don't strike twice: Understanding the nature of dram errors and the implications for system design. In *ASPLOS*, 2012.

[72] IBM. Power processor-based systems ras, june 27th, 2019. https://www.ibm.com/downloads/cas/2RJYYJML.

[73] MemVerge Inc. Memverge unveils first software-defined cxl memory. https://memverge.com/memverge-unveils-first-software-defined-cxl-memory-applications-to-support-4th-gen-amd-epyc-processors/.

[74] Intel. Address range partial memory mirroring. https://software.intel.com/content/www/us/en/develop/articles/address-range-partial-memory-mirroring.html and https://01.org/lkp/blogs/tonyluck/2016/address-range-partial-memory-mirroring-linux and https://www.intel.com/content/dam/develop/external/us/en/documents/memory-address-range-mirroring-validation-guide-556975.pdf.

[75] Intel. Intel Architecture ISA. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

[76] Intel. Boost pandas, scikit-learn, and TensorFlow Performance. https://www.intel.com/content/www/us/en/developer/articles/technical/code-changes-boost-pandas-scikit-learn-tensorflow.html, 2021.

[77] Intel. Intel Xeon Scalable processors featuring Accelerator Engines. https://www.intel.co.uk/content/www/uk/en/products/docs/accelerator-engines/overview.html, 2023.

[78] Intel+Facebook. Intel and Facebook Accelerate PyTorch Performance. https://community.intel.com/t5/Blogs/Tech-Innovation/Artificial-Intelligence-AI/Intel-and-Facebook-Accelerate-PyTorch-Performance-with-3rd-Gen/post/1335659.

[79] Intel+Google. Intel collaborating with Google to optimize TensorFlow. https://www.intel.com/content/www/us/en/developer/articles/news/leverage-deep-learning-optimizations-tensorflow.html.

[80] Open Fabrics Interface. Libfabric. https://ofiwg.github.io/libfabric/ and https://github.com/aws/libfabric.

[81] Engin Ipek, Jeremy Condit, Edmund B. Nightingale, Doug Burger, and Thomas Moscibroda. Dynamically replicated memory: Building reliable systems from nanoscale resistive memories. In *ASPLOS*, 2010.

[82] Maya Gokhale Ivy Peng, Roger Pearce. On the memory underutilization: Exploring disaggregated memory on hpc systems. In *2020 32st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020.

[83] JEDEC. DDR5 SDRAM - JESD79-5B, August 2022.

[84] H. Jeon, G. H. Loh, and M. Annavaram. Efficient ras support for die-stacked dram. In *2014 International Test Conference*, 2014.

[85] S. Jha, V. Formicola, C. D. Martino, M. Dalton, W. T. Kramer, Z. Kalbarczyk, and R. K. Iyer. Resiliency of hpc interconnects: A case study of interconnect failures and recovery in blue waters. *IEEE Transactions on Dependable and Secure Computing*, 15(6):915–930, Nov 2018.

[86] X. Jian, N. DeBardeleben, S. Blanchard, V. Sridharan, and R. Kumar. Analyzing reliability of memory sub-systems with double-chipkill detect/correct. In *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, 2013.

[87] X. Jian, H. Duwe, J. Sartori, V. Sridharan, and R. Kumar. Low-power, low-storage-overhead chipkill correct via multi-line error correction. In *SC*, 2013.

[88] X. Jian and R. Kumar. Adaptive reliability chipkill correct (arcc). In *HPCA*, 2013.

[89] X. Jian, V. Sridharan, and R. Kumar. Parity helix: Efficient protection for single-dimensional faults in multi-dimensional memory systems. In *HPCA*, 2016.

[90] I-Jui Sung John A. Stratton, Christopher Rodrigues, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. Parboil: A revised benchmark suite for scientificand commercial throughput computing. *IMPACT-12-01*, March 2012.

[91] Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Tim Harris. Shoal: Smart allocation and replication of memory for parallel programs. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015.

[92] M. R. Siavash Katebzadeh, Paolo Costa, and Boris Grot. Evaluation of an infiniband switch: Choose latency or bandwidth, but not both. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020.

[93] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. *Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol*, page 201–217. Association for Computing Machinery, New York, NY, USA, 2020.

[94] J. Kim, M. Sullivan, and M. Erez. Bamboo ecc: Strong, safe, and flexible codes for reliable computer memory. In *HPCA*, 2015.

[95] J. Kim, M. Sullivan, S. Lym, and M. Erez. All-inclusive ecc: Thorough end-to-end protection for reliable computer memory. In *ISCA*, 2016.

[96] Jeongchul Kim and Kyungyong Lee. Practical cloud workloads for serverless faas. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19. Association for Computing Machinery, 2019.

[97] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *ISCA*, 2014.

[98] Vamsee Reddy Kommareddy, Simon David Hammond, Clayton Hughes, Ahmad Samih, and Amro Awad. Page migration support for disaggregated non-volatile memories. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[99] Vamsee Reddy Kommareddy, Clayton Hughes, Simon David Hammond, and Amro Awad. Deact: Architecture-aware virtual memory support for fabric attached memory systems. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[100] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating function-as-a-service workflows. In *2020 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.

[101] Kubernetes. Production-Grade Container Orchestration. https://kubernetes.io/.

[102] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst.*, 6(2):254–280, apr 1984.

[103] L. A. Lastras-Montaño, P. J. Meaney, E. Stephens, B. M. Trager, J. O'Connor, and L. C. Alves. A new class of array codes for memory storage. In *2011 Information Theory and Applications Workshop*, 2011.

[104] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu. Adaptive-latency dram: Optimizing dram timing for the common-case. In *HPCA*, 2015.

[105] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. Mind: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, New York, NY, USA, 2021. Association for Computing Machinery.

[106] Larry Leemis. Probability models and statistical methods in reliability. *Department of Mathematics, College of William and Mary*, 2000.

[107] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark Hill, Marcus Fontoura, and Ricardo Bianchini. First-generation memory disaggregation for cloud platforms. In *arxiv*, 2021.

[108] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, and Ishwar Agarwal. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2023. Association for Computing Machinery.

[109] Qian Li and Christos Kozyrakis. Thousand island scanner (THIS): Scaling video analysis on AWS lambda. https://github.com/qianl15/this.

[110] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, New York, NY, USA, 2009. Association for Computing Machinery.

[111] David Liu, Amit Levy, Shadi Noghabi, and Sebastian Burckhardt. Doing more with less:orchestrating serverless applications without an orchestrator. In *Proceedings of the 20th USENIX Conference on Networked Systems Design and Implementation*, NSDI'23, USA, 2023. USENIX Association.

[112] S. Liu, B. Leung, A. Neckar, S. O. Memik, G. Memik, and N. Hardavellas. Hardware/software techniques for dram thermal management. In *HPCA*, 2011.

[113] C. Lu, K. Ye, G. Xu, C. Xu, and T. Bai. Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2884–2892, 2017.

[114] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Cache-coherent accelerators for persistent memory crash consistency. In *2022 Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2022.

[115] Taras Lykhenko, Rafael Soares, and Luis Rodrigues. Faastcc: Efficient transactional causal consistency for serverless computing. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 159–171, New York, NY, USA, 2021. Association for Computing Machinery.

[116] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. Wisefuse: Workload characterization and dag transformation for serverless workflows. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(2), jun 2022.

[117] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. Faasdom: A benchmark suite for serverless computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, DEBS '20, page 73–84, New York, NY, USA, 2020. Association for Computing Machinery.

[118] G. Mappouras, A. Vahid, R. Calderbank, D. R. Hower, and D. J. Sorin. Jenga: Efficient fault tolerance for stacked dram. In *ICCD*, 2017.

[119] P. J. Meaney, L. A. Lastras-Montano, V. K. Papazova, E. Stephens, J. S. Johnson, L. C. Alves, J. A. O'Connor, and W. J. Clarke. Ibm zenterprise redundant array of independent memory subsystem. *IBM Journal of Research and Development*, 56(1.2), Jan 2012.

[120] Mellanox Technologies . ConnectX-3 VPI Single and Dual QSFP+ Port Adapter Card User Manua. https://network.nvidia.com/pdf/user_manuals/ConnectX-3%20VPI_Single_and_Dual_QSFP+_Port_Adapter_Card_User_Manual.pdf.

[121] Memcached. Memcached internals for end users. https://github.com/memcached/memcached/wiki/UserInternals#how-much-memory-will-an-item-use.

[122] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2015.

[123] Micron. DDR4 SDRAM Datasheet. https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf.

[124] Micron. DDR5 SDRAM Whitepaper. https://www.micron.com/-/media/client/global/documents/products/white-paper/ddr5_more_than_a_generational_update_wp.pdf.

[125] Micron. Technical note: Uprating semiconductors for high-temperature applications. http://notes-application.abcelectronique.com/024/24-20035.pdf.

[126] Microsoft. Temporal Platform. https://learn.microsoft.com/en-us/azure/azure-functions/durable/.

[127] Microsoft Azure. Azure Functions Blob Access Trace 2020. https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsBlobDataset2020.md.

[128] Micrsoft. Azure Blob Storage. https://azure.microsoft.com/en-gb/products/storage/blobs.

[129] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA reads to build a fast, CPU-Efficient Key-Value store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013.

[130] Y. Mori, K. Ohyu, K. Okonogi, and R.-i. Yamada. The origin of variable retention time in dram. In *IEEE InternationalElectron Devices Meeting, 2005. IEDM Technical Digest.*, pages 1034–1037, 2005.

[131] A. Moshovos. Regionscout: exploiting coarse grain sharing in snoop-based coherence. In *ISCA*, 2005.

[132] Ireneusz Mrozek. *Multi-run memory tests for pattern sensitive faults*. Springer, 2019.

[133] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. Ofc: An opportunistic caching system for faas platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, New York, NY, USA, 2021. Association for Computing Machinery.

[134] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020.

[135] P. J. Nair, D. A. Roberts, and M. K. Qureshi. Citadel: Efficiently protecting stacked memory from large granularity failures. In *MICRO*, 2014.

[136] P. J. Nair, V. Sridharan, and M. K. Qureshi. Xed: Exposing on-die error detection information for strong memory reliability. In *ISCA*, 2016.

[137] Prashant J. Nair, Dae-Hyun Kim, and Moinuddin K. Qureshi. Archshield: Architectural framework for assisting dram scaling by tolerating high error rates. In *ISCA*, 2013.

[138] NASA Advanced Supercomputing Division. Nas parallel benchmarks. https://www.nas.nasa.gov/publications/npb.html.

[139] Nikolaos Nikitas, Ioannis Konstantinou, Vana Kalogeraki, and Nectarios Koziris. Cherry: A distributed task-aware shuffle service for serverless analytics. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 120–130, 2021.

[140] Siddharth Nilakantan, Karthik Sangaiah, Ankit More, Giordano Salvadory, Baris Taskin, and Mark Hempstead. Synchrotrace: synchronization-aware architecture-agnostic traces for light-weight multicore simulation. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015.

[141] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[142] NVIDIA. Mellanox bluefield dpu. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[143] OpenCAPI consortium. http://opencapi.org.

[144] Venkatesh Pallipadi and Suresh Siddha. Patting linux. *2008 Linux Symposium*, 2008.

[145] Gagandeep Panwar, Da Zhang, Yihan Pang, Mai Dahshan, Nathan DeBardeleben, Binoy Ravindran, and Xun Jian. Quantifying memory underutilization in hpc systems and using it to improve performance via architecture support. In *MICRO*, 2019.

[146] Adarsh Patil, Vijay Nagarajan, Rajeev Balasubramonian, and Nicolai Oswald. Dvé: Improving dram reliability and performance on-demand via coherent replication. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 526–539. IEEE Press, 2021.

[147] Adarsh Patil, Vijay Nagarajan, Nikos Nikoleris, and Nicolai Oswald. Āpta: Fault-tolerant object-granular cxl disaggregated memory for accelerating faas. In *2023 53nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 109–123, 2023.

[148] Archit Patke, Haoran Qiu, Saurabh Jha, Srikumar Venugopal, Michele Gazzetti, Christian Pinto, Zbigniew Kalbarczyk, and Ravishankar Iyer. Evaluating hardware memory disaggregation under delay and contention. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1221–1227, 2022.

[149] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H. Peter Hofstee. Thymesisflow: A software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[150] M. K. Qureshi. Pay-as-you-go: Low-overhead hard-error correction for phase change memories. In *MICRO*, 2011.

[151] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC)*, 2012.

[152] Microsoft Research. Rdma for cloud computing. https://www.microsoft.com/en-us/research/project/rdma-for-cloud-computing/publications/.

[153] Restle, Park, and Lloyd. Dram variable retention time. In *1992 International Technical Digest on Electron Devices Meeting*, pages 807–810, 1992.

[154] Robert Blankenship, Intel Corporation. CXL 1.1 Protocol Extensions: Review of the Cache and Memory Protocols in CXL. https://www.snia.org/educational-library/cxl-11-protocol-extensions-review-cache-and-memory-protocols-cxl-2020, 2020.

[155] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$t: A transparent auto-scaling cache for serverless applications. In *Proceedings of the ACM Symposium on Cloud Computing*, New York, NY, USA, 2021. Association for Computing Machinery.

[156] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4), November 1984.

[157] Karthik Sangaiah, Michael Lui, Radhika Jagtap, Stephan Diestelhorst, Siddharth Nilakantan, Ankit More, Baris Taskin, and Mark Hempstead. Synchrotrace: Synchronization-aware architecture-agnostic traces for lightweight multicore simulation of cmp and hpc workloads. *ACM Trans. Archit. Code Optim.*, March 2018.

[158] Sanghyuk Kwon, Young Hoon Son, and Jung Ho Ahn. Understanding ddr4 in pursuit of in-dram ecc. In *2014 International SoC Design Conference (ISOCC)*, 2014.

[159] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An argument for simple coma. In *HPCA*, 1995.

[160] TomsHardware Scharon Harding. Ecc memory in dram. https://www.tomshardware.com/uk/reviews/ecc-memory-ram-glossary-definition,6013.html.

[161] Stuart Schechter, Gabriel H. Loh, Karin Strauss, and Doug Burger. Use ecp, not ecc, for hard failures in resistive memories. In *ISCA*, 2010.

[162] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: A large-scale field study. *SIGMETRICS Perform. Eval. Rev.*, 37(1), June 2009.

[163] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, October 2018. USENIX Association.

[164] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 281–295, New York, NY, USA, 2020. Association for Computing Machinery.

[165] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020.

[166] Taniya Siddiqua, Athanasios E. Papathanasiou, Arijit Biswas, Sudhanva Gurumurthi, Intel Corp, and Teradata Aster. Analysis and modeling of memory errors from large-scale field data collection. In *SELSE*, 2013.

[167] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1093–1108, New York, NY, USA, 2020. Association for Computing Machinery.

[168] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[169] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. In *Proceedings of VLDB Endowment*, volume 13. VLDB Endowment, jul 2020.

[170] V. Sridharan and D. Liberty. A study of dram failures in the field. In *SC*, 2012.

[171] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *ASPLOS*, 2015.

[172] Vilas Sridharan, Jon Stearley, Nathan DeBardeleben, Sean Blanchard, and Sudhanva Gurumurthi. Feng shui of supercomputer memory: Positional effects in dram and sram faults. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13. Association for Computing Machinery, 2013.

[173] Standard Performance Evaluation Corporation. SPEC CPU 2017. https://www.spec.org/cpu2017/.

[174] Meysam Taassori, Rajeev Balasubramonian, Siddhartha Chhabra, Alaa R. Alameldeen, Manjula Peddireddy, Rajat Agarwal, and Ryan Stutsman. Compact leakage-free support for integrity and reliability. In *ISCA*, 2020.

[175] John R. Tramm, Andrew R. Siegel, Benoit Forget, and Colin Josey. Performance analysis of a reduced data movement algorithm for neutron cross section data in monte carlo simulations. In *EASC 2014 - Solving Software Challenges for Exascale*, Stockholm, 2014.

[176] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto, 2014.

[177] A. N. Udipi, N. Muralimanohar, R. Balsubramonian, A. Davis, and N. P. Jouppi. Lot-ecc: Localized and tiered reliability mechanisms for commodity memory systems. In *ISCA*, 2012.

[178] UEFI Forum. CXL: UEFI and ACPI specification enhancement. https://uefi.org/node/4093.

[179] PARSEC Group Princeton University. A memo on exploration of splash-2 input sets. https://parsec.cs.princeton.edu/doc/memo-splash2x-input.pdf.

[180] UPMEM. True Processing In-Memory (PIM) solution. https://www.upmem.com/technology/.

[181] Dmitrii Ustiugov, Theodor Amariucai, and Boris Grot. Analyzing tail latency in serverless clouds with stellar. In *Proceedings of the 2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2021.

[182] 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*, New York, NY, USA, 2021. Association for Computing Machinery.

[183] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on cc-numa compute servers. In *ASPLOS*, 1996.

[184] vHive Ecosystem. vSwarm - Serverless Benchmarking Suite. `https://github.com/ease-lab/vSwarm`.

[185] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, page 181–194, December 2003.

[186] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. InfiniCache: Exploiting ephemeral serverless functions to build a Cost-Effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, Santa Clara, CA, February 2020. USENIX Association.

[187] Wei Wang, Jack W. Davidson, and Mary Lou Soffa. Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale numa machines. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 419–431, 2016.

[188] Zixuan Wang, Joonseop Sim, Euicheol Lim, and Jishen Zhao. Enabling efficient large-scale deep learning training with cache coherentdisaggregated memory systems. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[189] R. Yeleswarapu and A. K. Somani. Sscmsd - single-symbol correction multi-symbol detection for dram subsystem. In *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2018.

[190] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. P. Jouppi, and M. Erez. Free-p: Protecting non-volatile memory against both hard and soft errors. In *HPCA*, 2011.

[191] Doe Hyun Yoon and Mattan Erez. Virtualized and flexible ecc for main memory. In *ASPLOS*, 2010.

[192] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, New York, NY, USA, 2020. Association for Computing Machinery.

[193] Da Zhang, Vilas Sridharan, and Xun Jian. Exploring and optimizing chipkill-correct for persistent memory based on high-density nvrams. In *MICRO*, 2018.

[194] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, November 2020.

[195] Z. Zhang and J. Torrellas. Reducing remote conflict misses: Numa with remote cache versus coma. In *HPCA*, 1997.

[196] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu. Mini-rank: Adaptive dram architecture for improving memory power efficiency. In *MICRO*, 2008.

[197] Tobias Ziegler, Dwarakanandan Bindiganavile Mohan, Viktor Leis, and Carsten Binnig. Efa: A viable alternative to rdma over infiniband for dbmss? In *Proceedings of the 18th International Workshop on Data Management on New Hardware*, DaMoN '22, New York, NY, USA, 2022. Association for Computing Machinery.