Optimizing Data Movement Through Software Control of General-Purpose Hardware Caches

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Brian C. Schwedock

B.S., Computer Engineering and Computer Science, University of Southern California
M.S., Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

July 2023

Acknowledgments

To start off, nothing would have been possible without my research advisor, Professor Nathan Beckmann. From guiding me toward exciting research topics to helping me refine my communication and presentation skills, Nathan has been with me every step of the way. My favorite Ph.D memories are spitballing wild ideas with Nathan when starting new projects. Nathan has been a wonderful mentor and friend through this long journey.

I would like to thank my committee members: Professor Brandon Lucia, Professor James Hoe, and Dr. Carlos Villavieja. I have enjoyed many lunch meetings and gatherings with Brandon and his Abstract research group. James' leadership of the CALCM organization has provided me with meaningful collaborations with CMU's computer architecture community. Carlos advised me on the aspects of industry research during my internships at Google and helped me to realize impactful change on a massive scale.

I am immensely grateful for PDL's community and computing infrastructure, which enabled me to evaluate and present my research ideas. In addition to PDL's many compute resources used to run my experiments, the IT team is top-notch. Chad Dougherty and Jason Boles, in particular, solved any and every issue I threw their way.

I am fortunate to have received multiple sources of funding throughout my Ph.D. The NSF Graduate Research Fellowship, CMU ECE Ann and Martin McGuinn Graduate Fellowship, CMU CIT Bertucci Fellowship, and NSF grant CCF-1845986 all provided valuable assistance in supporting my studies.

My friends and colleagues at CMU have made my time here all the more enjoyable, both through intellectual discussions and casual socializing. The CORGi group has been by my side the whole time: Graham Gobieski, Elliot Lockerman, Nathan Serafin, Sara McAllister, Souradip Ghosh, Nikhil Agarwal, Mitchell Fream, Xuesi Chen, and the honorable corgis themselves, Arya and Baphy. Our partner Abstract group has been awesome too: Alexei Colin, Kiwan Maeng, Vignesh Balaji, Emily Ruppel, Brad Denby, Milijana Surbatovich, Harsh Desai, McKenzie van der Hagen, Zhuo Cheng, and Kyle McCleary. I want to also thank Matt Butrovich, Wan Lim, Sam Arch, Kartikeya Bhardwaj, Anderson Sartor, Maia Blanco, Elliott Binder, Upasana Sridhar, Ziqi Wang, Kaiyang Zhao, and Patrick Coppock.

My undergraduate and master's mentees enabled me to grow as a researcher and educator. Thank you to Amolak Nagi, Hanchen Yang, Jennifer Seibert, Bas Yoovidhya, and Jennifer Brana for allowing me to share my experiences with you.

Finally, I am fortunate to have a loving family that has supported me through the highs and the lows these many years. My parents are always there for me, no matter the time or reason. My brother, Nathan, always has my back when I need him. And a special thanks to all my wonderful cousins, aunts, uncles, and grandparents.

Abstract

Computer systems are increasingly burdened by the rising cost of data movement. Moving data across chip in a modern processor consumes orders-of-magnitude more energy than performing an arithmetic operation on the data. On-chip caches also constitute more than half of a chip's area. The severity of these problems will continue to grow alongside rising core counts and data-processing requirements.

The underlying issue is that chip multiprocessors (CMPs) provide a compute-centric programming interface where software views the entire memory hierarchy as a black box. Software issues loads and stores, and it is entirely up to hardware to manage all data movement between the core and main memory. Although this interface simplifies software, hardware is forced to resort to overly general application-agnostic optimizations.

To overcome the limitations of compute-centric CMPs, prior work has proposed specialized hierarchies which add custom logic to the hierarchy to enable novel data-movement-reducing features. Specialized hierarchies reduce data movement by either moving data closer to compute (data placement) or moving compute closer to data (near-data computing). These data-centric systems often provide significant benefits by customizing data movement within the memory hierarchy to specific applications. Unfortunately, adding custom logic to CMPs for every possible application domain is not a scalable solution.

The goal of this thesis is making specialized hierarchies practical by letting software customize data movement, eliminating the need for application-specific custom hardware. Our proposed systems address both data placement and near-data computing (NDC). First, Jumanji shows how software-controlled data placement for distributed last-level caches enables optimizing for a variety of application objectives on a single processor. Specifically, Jumanji targets a datacenter environment where co-running applications either care about tail latency or throughput, and all applications care about security. Second, täkō demonstrates how a major NDC paradigm, data-triggered computation, can be implemented by letting software observe and manipulate data as it traverses the cache hierarchy. In täkō, applications register software callbacks that execute in response to in-cache data-movement (i.e., cache misses, evictions, and writebacks), a novel data-centric mechanism that supports many optimizations which each previously required custom hardware. Finally, Leviathan unifies multiple NDC paradigms under a single architecture and programming interface to provide a truly practical NDC system. Together, these contributions exhibit the feasibility of programmable data movement in general-purpose processors.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The performance and energy efficiency of computer systems are increasingly burdened by the rising cost of data movement [53, 82, 88, 128]. Over the last few decades, processor speeds have improved orders of magnitude faster than memory speeds [61]. Some systems already treat computation as practically free compared to the cost of data movement [80]. It is evident that computer architectures must treat data movement as a first-order concern.

However, whereas processors often add new compute features and capabilities (e.g., SIMD [180] and cryptographic instructions [9]), the memory hierarchy has not changed much in recent years. Figure 1.1 illustrates a representative chip multiprocessor (CMP). Processors contain an ever-growing number of cores, each of which is accompanied by multiple levels of private caches (L1s and L2) and one bank of the shared last-level cache (LLC). The distribution of LLC banks across the chip, called non-uniform cache architecture (NUCA), results in variable network latency between the core and LLC bank depending on the network distance. Main memory is additionally split into multiple banks to serve more requests in parallel. This overall composition has remained fairly stagnant over several generations of hardware, with changes mostly to the size and number of levels of the cache hierarchy. And since caches already consume more than 50% of chip area [124], there is limited ability to scale them further.

Despite the importance of data movement, processors still provide a compute-centric programming interface where software views the entire memory hierarchy as a black box. Software issues loads and stores (Figure 1.2), and it is entirely up to hardware to manage all data movement between the core and main memory. The goal of this interface is to simplify software since it does not need to worry about the distributed memory hierarchy. But the problem is that *hardware is forced to resort to overly general microarchitectural optimizations* (e.g., cache replacement policies and data prefetchers) that are hidden from software and thus cannot optimize for any specific application.

Customized data movement is necessary. The fact is that the traditional memory hierarchy cannot sustain the growing cost of data movement. Many others have realized this, resulting in a wave of proposals for *specialized memory hierarchies* which add custom logic to the hierarchy to enable novel data-movement-reducing features (detailed in

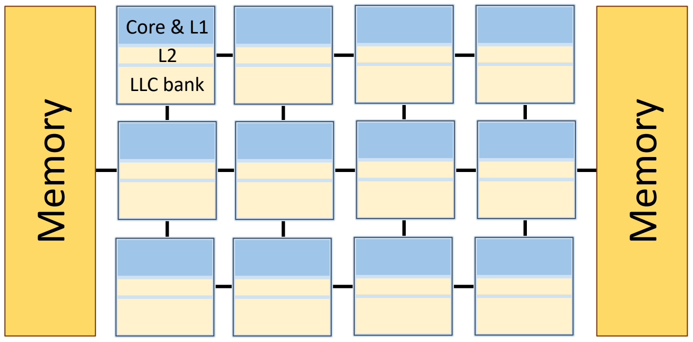


Figure 1.1: Representative chip multiprocessor (CMP). Each tile contains a core, private L1s and L2 caches, and one bank of the shared last-level cache (LLC). Tiles are connected through a network-on-chip (NoC).

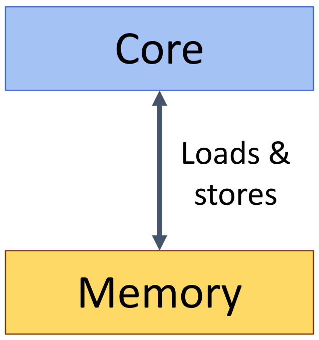


Figure 1.2: Memory interface exposed by processor instruction set architectures (ISA).

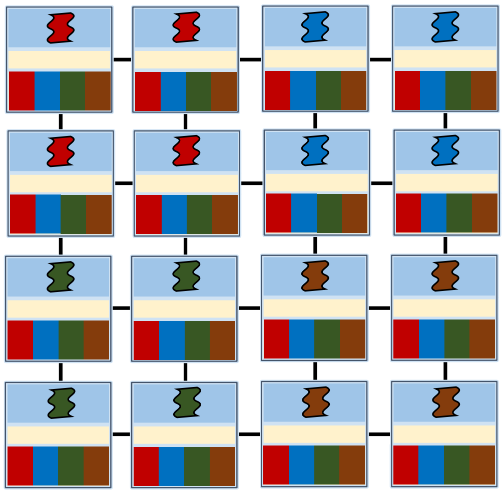
Sections 2.3 – 2.7). Specialized hierarchies generally reduce data movement in one of two ways: moving data closer to compute (Figure 1.3), or moving compute closer to data (Figure 1.4).

Moving data closer to compute involves placing data in specific LLC banks to minimize the network distance between data and the cores accessing them (Figure 1.3b). Dynamic NUCA (D-NUCA) designs actively place data in cache banks close to their cores, typically by replicating data [26, 41, 48, 79, 105, 155, 264] or restricting where data can reside [17, 27, 28, 42, 49]. Data placement reduces the network distance between threads and their data, improving both latency and energy to access data. Importantly, data placement allows system size to gracefully scale because data can be placed locally no matter how big the system.

The other type of specialized hierarchies moves compute closer to data, often with near-data computing (NDC) where compute logic is placed near cache or memory banks. These designs mitigate the cost of data movement by operating on data where it resides within the memory hierarchy. Avoiding the transfer of data across levels of the memory hierarchy and across on-chip networks can provide substantial performance and energy improvements. For example, Figure 1.4b demonstrates how NDC reduces data movement on a pointer chasing application.

Unfortunately, prior work on both data placement and NDC suffer from the same issue: they all require adding custom hardware to a general-purpose processor, yet each only benefits a narrow application domain. Prior D-NUCAs generally optimize only for throughput-oriented applications (and as a result, harm other applications). And proposed NDCs typically target very specific workloads (e.g., pointer chasing [87, 91] or data compression [12, 62, 178, 179, 200, 226]). Taken literally, prior work suggests that memory hierarchies should contain an ever-growing number of specialized hardware optimizations to benefit all potential applications. But this is unrealistic because each hardware addition and change to the hardware-software interface requires large, up-front investment in both hardware and software to be effective.

Adding custom hardware to the memory hierarchy of general-purpose processors is only practical if enough


(a) Commercial processors spread data evenly across LLC banks (i.e., S-NUCA).

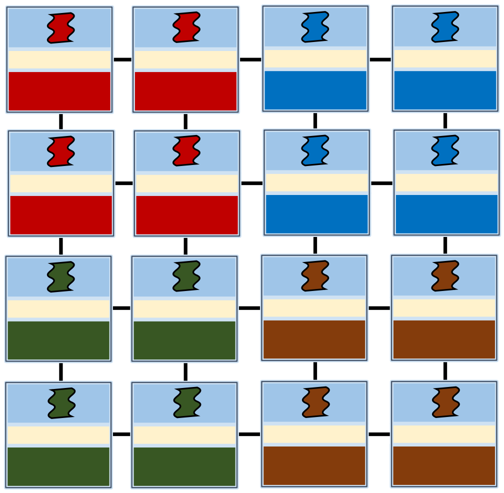
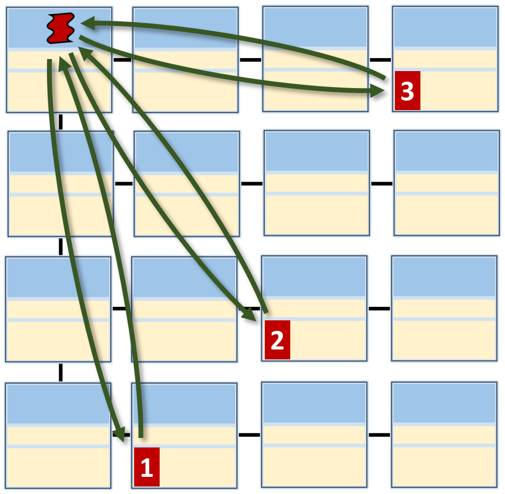
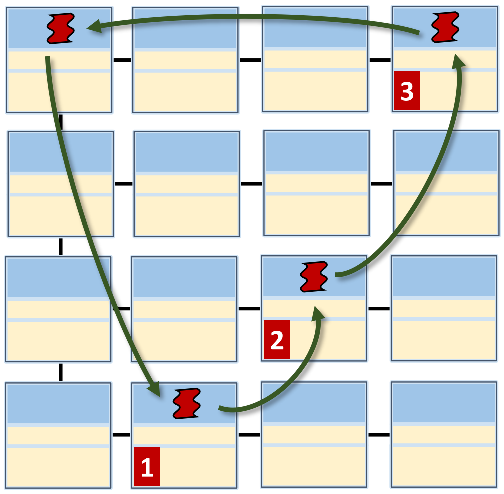

(b) D-NUCA places data in LLC banks near their threads.

Figure 1.3: Reducing data movement by *moving data closer to compute*.



(a) Traditional thread performing pointer chasing needs to pull all data to the core, one at a time.



(b) NDC performing pointer chasing offloads work to the data's location, reducing network traversals.

Figure 1.4: Reducing data movement by *moving compute closer to data*.

applications will benefit. Accordingly, this thesis aims to generalize data-movement optimizations for both data placement and NDC, allowing software to optimize data movement for different applications on a single architecture. Jumanji shows how software customization of prior D-NUCA hardware can benefit the objectives of all co-running applications on a single processor. täkō demonstrates how a major NDC paradigm, data-triggered computation, can be implemented by triggering software callbacks when data moves through the hierarchy. Leviathan then leverages täkō as one component in a general-purpose NDC system that supports a variety of different NDC paradigms with a single architecture and programming interface.

The massive quantity and impressive performance of recently proposed specialized hierarchies implies that the computer architecture community has strong faith in their potential. However, adoption of specialized hierarchies in commodity processors still requires a lot of work. The object of this thesis is to address the challenges in designing practical, specialized hierarchies.

1.1 Challenges

The end goal is developing a general-purpose system that addresses the growing cost of data movement head on. This endeavor involves fundamental changes to the traditional memory hierarchy, a path met with heavy resistance by chip manufacturers. Accordingly, we must overcome major challenges to the practical adoption of data-centric computer architectures.

Challenge 1: Applications know what to optimize for, but hardware controls all data movement. Many decades ago, when cores connected directly to memory, the load-store memory interface was a natural choice for software to access data (Figure 1.2). However, memory has evolved into a complex, distributed hierarchy of cache and main memory banks (Figure 1.1), yet the memory interface has remained unchanged. Data movement is entirely managed by hardware while software continues to view memory as a black box through loads and stores. While the decision to maintain the interface enables software to ignore the complexity of the memory hierarchy, it also means that hardware has no knowledge of the applications running on the cores and must resort to conservative optimizations (e.g., cache replacement policies and data prefetchers). This is a problem because these optimizations provide unimpressive performance benefits and data-movement reductions (e.g., a state-of-the-art replacement policy yields only 15% average speedup [100]).

To overcome this limitation, recent architecture conferences have been filled with proposals for specialized memory hierarchies which include custom hardware *within* the hierarchy to enable optimizations unrestricted by the traditional load-store interface. The reason these designs provide significant performance and energy benefits is because they tailor data movement within the hierarchy to the needs of the applications. When applications can express their access patterns, data structures, and performance objectives to hardware, then the hardware can use this additional knowledge to appropriately partition caches, relocate data, move compute closer to data, or perform any other relevant optimization. The key is the ability to customize data movement to the needs of the application. The difficulty is designing the mechanisms to express and optimize application goals.

Challenge 2: Application-specific custom hardware is not scalable. Despite the large benefits promised by specialized memory hierarchies, prior proposals all suffer a severe drawback: they each require adding their own custom hardware. This mirrors the fundamental tradeoff between programmable processors and ASICs—specialization improves performance but limits applicability. Although these proposals promise massive reductions in data movement, they each benefit a small subset of the immense application space. It is just not feasible to incorporate additional custom hardware and a new hardware-software interface in a general-purpose CPU for each proposed optimization. Accordingly, the objective should be designing a single system that provides application-specific benefits across a wide range of application domains using the same hardware.

We believe that the solution is incorporating general-purpose, programmable hardware within the cache hierarchy along with a simple and flexible programming interface. Although prior specialized hierarchies used custom hardware, programmable resources can often achieve the same effect if flexible hardware provides software with the ability to tailor data movement. Thus, the real challenges are designing the hardware mechanisms that enable programmable control over data movement and providing a programming interface that enables software to express its application goals and specialized data-movement requirements.

1.2 Contributions

The objective of this work is designing a general-purpose, programmable cache hierarchy. This thesis tackles data movement optimizations from both angles: moving data closer to compute and moving compute closer to data. Each contribution focuses on hardware-software co-design, where hardware opens the memory interface to enable software control over data movement, and software libraries provide applications with an easy-to-use programming interface. In total, these works support the following thesis statement: *Application-specific data-movement optimizations are achievable and practical on general-purpose cache hierarchies with a simple and flexible programming interface.*

Jumanji (Chapter 3). Jumanji demonstrates how software control over data placement enables co-optimization of multiple application objectives simultaneously. Whereas prior works proposed mechanisms for implementing software-controlled data placement on NUCAs [28, 30], they only targeted a single application objective: throughput. Jumanji builds upon these works to demonstrate that software-controlled data placement can optimize for a variety of competing application objectives in the context of datacenter workloads. With only simple software changes, Jumanji reuses general-purpose data-placement hardware to co-optimize for latency-critical applications and batch applications while also providing strong security guarantees across untrusted applications. Across mixes of datacenter workloads, Jumanji enforces tail-latency requirements, improves batch speedup by 14%, and provides stronger security than prior work.

täkō (Chapter 4). täkō's goal is generalizing the wide range of data-triggered NDC systems. Many proposed data-movement optimizations involve computation triggered in response to data movement, but the current memory interface hides all data movement from software (see Figure 1.2). As a result, prior works use specialized hardware to implement their data-triggered functionality. But as discussed, application-specific specialized hardware is not a scalable solution.

täkō's solution is a new hardware-software interface that lets software observe and manipulate data as it traverses the cache hierarchy. Specifically, täkō lets applications register software callbacks which execute on near-data engines in response to cache misses, evictions, and writebacks. Unlike traditional NDC in which cores offload compute into the hierarchy, with täkō, caches trigger compute as data *moves*. This interface enables täkō's single architecture to cover a wide range of data-movement features and optimizations which previously each required their own custom hardware. We demonstrate täkō's ability to implement a wide range of optimizations through five different case studies, across which täkō provides up to 3.5× speedup.

Leviathan (Chapter 5). Leviathan's objective is to make NDC truly practical. While täkō enables software to implement one type of NDC (data-triggered), it fails to support the wide range of other NDCs, limiting its usefulness. We first build a taxonomy of NDC to identify the commonalities and differences across NDC designs. The result of this study leads us to design a single architecture which supports multiple types of NDC with a unified programming interface. Leviathan further performs all necessary data management to ensure data locality for near-data actions and executes actions near-data at the right time and place. Across a range of NDC-specialized applications, Leviathan improves performance by up to 3.7×.

1.3 Outline

The remainder of this thesis is organized as follows. Chapter 2 discusses relevant background on data movement, cache architecture, optimizing caches for application-specific objectives, and specialized memory hierarchies. Chapter 3 describes Jumanji, our system on customizing D-NUCA for datacenter workloads. Chapter 4 covers täkō, a polymorphic cache hierarchy that enables software to operate on data as it traverses the hierarchy. Chapter 5 presents Leviathan, a general-purpose NDC system that supports many types of NDC optimizations with a single architecture. Chapter 6 proposes directions for further work on programmable specialized hierarchies, and Chapter 7 concludes.

Chapter 2

Background

This chapter presents background information and relevant prior work on topics upon which this thesis builds. This chapter is broken down into the following sections: managing memory latency, cache architecture, cache partitioning, non-uniform cache architectures, caching for tail latency, cache security, near-data computing, and actor-based reactive programming.

2.1 Managing Memory Latency

Since processor speeds improve significantly faster than memory speeds [61], minimizing the impact of memory latency on application performance is a first-order concern. A large portion of modern processors is devoted to this challenge, both by hiding and reducing memory latency.

2.1.1 Hiding memory latency

One method to limit the impact of memory latency is by continuing forward progress of a thread concurrently with its long-latency memory accesses (i.e., hiding the latency behind other work). The two common mechanisms for hiding latency are prefetching [65] and out-of-order (OOO) execution [220].

Prefetching. Prefetching attempts to predict and issue a core's future memory accesses so that data is always available nearby (e.g., in private caches) when the core requests it. Regular access patterns (e.g., strided and affine) are easy to detect and predict, and their access latency can thus easily be hidden by simple prefetchers [65]. However, irregular access patterns are much more difficult to predict and are the focus of recent work on more complex prefetchers [8, 99, 218, 254].

Out-of-order (OOO) execution. OOO cores take a different approach to hiding memory latency by executing unrelated instructions concurrently with memory accesses [220]. Conceptually, when an instruction issues a memory request, the

OOO core continues executing later instructions which are not dependent on the memory request while waiting for the request to complete. This enables forward progress during memory requests. Unfortunately, this simple idea requires complex and large hardware support (e.g., large instruction windows [181, 190] and runahead execution [164, 165, 190]) to provide observably sequential execution and sufficient performance improvements.

Despite the performance benefits of latency-hiding mechanisms, *they do not reduce the amount of data movement*. In fact, prefetchers can only increase total data movement, by fetching data that is never used or evicting useful data from caches. As a result, these mechanisms trade improved performance for increased energy usage and memory accesses. Systems must actually reduce total data movement to efficiently scale, not simply hide it.

2.1.2 Reducing memory latency

Of the methods to reduce memory latency, by far the most common is caching.

Caching. Modern processors contain distributed, multilevel cache hierarchies that sit between the cores and main memory (Figure 1.1). The main purpose of caches is to take advantage of the inherent temporal locality common in applications by keeping frequently accessed data in smaller and faster memories compared to main memory. Serving memory requests from caches instead of DRAM provides latency and energy benefits. Ideally, as many requests as possible should be served from the smallest caches because data-movement reductions differ notably even across levels of the cache hierarchy (as explained in Section 2.2).

Although traditionally a microarchitectural mechanism hidden from software, minor control over cache behavior has been exposed to software, enabling further data-movement benefits. Sections 2.3 – 2.6 cover prior work on software-managed caches and how they provide application-specific benefits. In this line, our work heavily relies on the underlying cache hierarchy for its implicit data-locality benefits. But whereas prior work has only provided the slightest opportunities for explicit software management of caches, our work provides novel interfaces which enable considerable software control of data-movement optimizations throughout the entire cache hierarchy.

Data-movement accelerators. To perform data-movement optimizations not possible with only caches, prior work has proposed incorporating custom hardware within the memory hierarchy for further reduction in data movement. These optimizations typically reduce data movement either by moving data closer to compute (Section 2.4) or by moving compute closer to data (Section 2.7).

Data-movement accelerators have promised massive performance improvements and energy savings for decades, yet they are only starting to be incorporated into commercial hardware. Most notable is Intel's recent Sapphire Rapids processor line, which incorporates accelerators for data streaming, load balancing, and in-memory analytics [32]. While a step in the right direction, these accelerators are still severely restrictive in types of supported optimizations. Our work proposes frameworks for general-purpose data-movement optimizations that provide software with a much wider range of potential benefits.

2.2 Cache Architecture

Modern CMPs contain multilevel, distributed cache hierarchies (Figure 1.1). Each core typically has two levels of fast, private caches (L1s and L2) backed by a much larger, shared last-level cache (LLC). The LLC, which can often occupy more than half of the chip's entire area [124], is shared among all cores in order to fit the largest possible working set.

The major difference between the LLC and private caches is that the LLC is distributed across the chip, typically by colocating one bank with each core. Splitting the LLC into multiple banks is necessary as systems scale in both number of cores and cache size to avoid a single, slow bank from bottlenecking concurrent accesses from many cores. The banks are connected over an on-chip network (NoC), which exposes variable network latency between a core and different LLC banks. This non-uniform cache access (NUCA) architecture results in accesses to far away banks incurring longer latency and more energy than closer banks.

Although distributing the LLC enables systems to scale, it introduces a new challenge in *data placement* across the banks. For example, a representative LLC bank could have an access latency of 8 cycles. For an 8×8 mesh network connecting the banks with 3 cycles per hop, the round trip network latency between corners of the chip is 84 cycles. As a result, over 90% of the latency to access the far-away bank comes from just the network. Energy costs for network versus bank access paint a similar story.

This extreme difference between accessing and transferring data would seem to warrant careful efforts at optimizing LLC data placement. However, modern CPUs simply spread data evenly across cache banks [248], opting for a simpler design over improved efficiency. Nevertheless, abundant prior work has tackled this challenge, as will be discussed more in Section 2.4.

2.3 Cache Partitioning

Cache partitioning enables software to divide the shared cache space among different entities, e.g., threads, cores, or data. Partitioning can be used to benefit many different application objectives, e.g., reserve enough space to fit a working set, enforce quality-of-service for critical applications, or isolate applications to defend cache attacks. The two components of cache partitioning are the *partitioning policy* to decide on partition sizes and the *partitioning scheme* to enforce the partitions.

2.3.1 Partitioning schemes

A partitioning scheme's purpose is to enforce software-defined partition sizes. Common techniques involve restricting the physical location where data can reside [47, 95, 191, 227], modifying the allocation or replacement policy [148, 198, 231, 239, 240], or page coloring [134, 219]. Due to simplicity of implementation and ability to strictly enforce partitions, location restriction through way partitioning [47, 95] is the most popular approach and the scheme we focus on in this work.

Way partitioning restricts which *ways* of a set-associative cache a particular cache line can be inserted into [47]. In set-associative caches, a line's address determines which set it maps to, but the line can be inserted into any way of the set, as selected by the replacement policy. Way partitioning further limits which ways may contain the line, letting the cache control capacity allocated to different partitions.

Despite way partitioning's benefits, it suffers from two major drawbacks. Cache banks are typically designed with small associativities (e.g., 8 to 32 ways) to limit hardware overheads. This restricts both the granularity and number of possible partitions. As the number of cores rapidly rises across processor generations, the small associativity forces increasing numbers of cores to share partitions, greatly limiting the benefits. The other drawback is that way partitioning does not provide any data-placement advantages because the address to bank mapping is unmodified. As a result, cache lines are still spread evenly across banks, and the partitions are enforced identically within each bank.

Unfortunately, *the only support in most modern CPUs for software control of data movement is way partitioning*, through Intel's Cache Allocation Technology (CAT) [95]. Intel CAT, while providing some benefits from partitioning, still suffers the aforementioned drawbacks of way partitioning. There is currently no hardware support in modern CPUs for software-controlled data placement across the distributed banks of a NUCA cache.

2.3.2 Partitioning policies

Partitioning policies have two components: one or more monitors, which analyze performance with previous partitions to predict the relationship between partition size and performance, and a controller, which uses the monitors' predictions to decide new partitions. Monitors can take different forms depending on the performance metric of interest (e.g., software request-latency queue [45, 111, 141] or hardware miss-curve utility monitors [28, 30, 186]). The controller occasionally repartitions the cache based on updated feedback from the monitors to optimize a target objective (e.g., throughput or tail latency). Controllers are often implemented in software because the overhead to compute new partitions is negligible if repartitioning is infrequent enough.

Although prior work on cache partitioning is often applied to way partitioning, that need not be the case. The same techniques can be applied to other partitioning schemes, such as with distributed NUCA caches [28, 30]. Jumanji uses both hardware and software monitors for NUCA-aware data placement, as detailed in Section 3.2.1 and Section 3.3.2, respectively.

2.4 Non-Uniform Cache Access (NUCA) Architectures

As discussed in Section 2.3, cache partitioning provides limited benefits because it does not consider data placement across distributed NUCAs. NUCA-aware data-placement policies enable performance to scale with an increasing number of cores by keeping data close to their cores.

2.4.1 Static NUCA

The simplest mechanism for implementing a distributed cache is static NUCA (S-NUCA) [118], where data is spread evenly across all cache banks. There is no consideration for data placement with S-NUCA, making it very simple to implement. Data can only ever reside in a specific cache bank, which is typically determined by hashing the address. Unfortunately, commercial processors employ S-NUCA designs, incurring severe data-movement costs.

2.4.2 Dynamic NUCA

Dynamic NUCA (D-NUCA) designs improve on S-NUCA by actively placing data in cache banks close to their cores. Early D-NUCAs treated LLC banks as a hierarchy [26,27,41,48,155,185,264], e.g., by checking the local bank before a global "home" bank. In contrast, *single-lookup D-NUCAs* restrict each memory address to live in a single LLC bank at a time [17,28,42,49,79,105], avoiding LLC directories and multiple lookups. These D-NUCAs typically control placement at page granularity and cache page locations in the TLB.

Though single-lookup D-NUCAs originally used the page table out of convenience, this design lets *software control where data is placed.* Software scheduling algorithms can find near-optimal data placements that would be too expensive to find in hardware alone [17,28,30,222,223]. Single-lookup D-NUCAs thus significantly reduce data movement over other D-NUCAs, at the cost of modest complexity in the operating system (OS).

Jigsaw [28, 30] is a single-lookup D-NUCA that introduced the concept of virtual caches, software-managed collections of NUCA banks. Virtual caches enable software to partition the LLC both within and across banks, limiting a core's data to reside within a subset of nearby banks. Jigsaw's ability to enforce fine-grained data placement across distributed LLCs provides substantial data movement reductions both across the NoC and through improved cache hit ratios.

Unfortunately, the sole focus of prior D-NUCA designs, including Jigsaw, has been to minimize total data movement. But applications have increasingly diverse objectives which do not always benefit from strictly minimizing system-wide data movement. Today's dominant computing environment, the datacenter, runs applications that care about tail latency in addition to more traditional throughput-oriented batch applications. We show that minimizing total data movement on a system running mixes of latency-critical and batch applications results in violations of tail-latency deadlines because data-movement-oriented metrics naturally prioritize batch speedup over tail latency (Section 3.1).

Table 2.1: Comparison of Jumanji to prior LLC designs.

		Batch speedup	Tail latency	Security
Prior work	D-NUCA [28, 30, 264]	✓	✗	✗
	Tail-aware [45, 111, 141]	✗	✓	✗
	Secure [74, 170, 172, 173]	✗	✗	✓
	Jumanji	✓	✓	✓

Further, following several high-profile breaches [121, 135], security has become a first-order concern for many datacenter customers. With cloud services colocating customers' jobs on the same machine, it is vital to provide hardware-security guarantees. But prior D-NUCAs unknowingly colocate data from different customers' virtual machines (VMs) on the same LLC banks, leaving the system vulnerable to cache side-channel attacks (Section 3.1). Just minimizing total data movement is clearly insufficient for the diverse application objectives of datacenter workloads.

2.4.3 Our contributions vs. prior D-NUCAs

Fortunately, the silver lining is that prior D-NUCAs can easily be tailored for different application objectives, and *the key is that data placement is controlled by software*. We propose Jumanji (Chapter 3), a new D-NUCA that optimizes for the diverse application objectives of the datacenter. Jumanji requires only software changes to the prior Jigsaw [28, 30], reusing its programmable data-placement hardware. Whereas prior D-NUCAs strictly focused on minimizing total data movement (i.e., improving speedup), Jumanji ensures that each application's data placement optimizes for its specific needs (Table 2.1). Importantly, since software controls the data placement, the same hardware can be programmed to optimize for any application objective, even new objectives that we do not know about yet.

2.5 Caching for Tail Latency

User-facing applications in the datacenter increasingly drive growth in computing [97]. Unlike traditional computer systems that run scientific, analytic, or other batch workloads, these user-facing applications care about response latency, which must be short (e.g., 100 ms) to keep users engaged [56, 201]. Moreover, since serving a request requires completing many tasks, the overall response latency is set by the longest of these tasks, making systems sensitive to *tail latency*.

Prior work has re-designed systems for tail latency in many ways [21, 58, 152]. Systems minimize power through dynamic voltage and frequency scaling (DVFS) [92, 110, 140, 141, 143, 242, 255], varying parallelism as load fluctuates [78, 167, 182], or finding jobs that can safely run alongside latency-critical applications [57, 58, 63, 150, 245]. This work is complementary to this thesis and can be combined with techniques we propose.

A few systems focus on the effect of the LLC on tail latency. Ubik [111] partitions the LLC to safely colocate batch and latency-critical applications. Similar to DVFS, Ubik gives idle latency-critical applications minimal LLC space and "boosts" the allocation once a request arrives. Since latency-critical applications are mostly idle, Ubik non-trivially increases batch allocations.

Heracles [141] and Parties [45] control LLC space, core DVFS, memory bandwidth, and network traffic to meet tail-latency deadlines. These systems manage resources through feedback control and partition the LLC using Intel CAT [95] (i.e., way partitioning). We compare Jumanji with a similar scheme; however, we compare them only at the LLC.

The drawback of these prior works is that they only consider cache partitioning's affect on tail latency. As discussed in Section 2.4, partitioning alone suffers from excessive data movement in the NoC due to randomized data placement. Yet data placement's affect on tail latency has not been previously addressed. Consequently, these works overallocate cache space to latency-critical applications, providing subpar performance for co-running best-effort batch applications (Table 2.1).

2.5.1 Our contributions vs. prior work on tail latency

Jumanji extends prior work on tail latency to incorporate the benefits of NUCA data placement. Not only does moving data closer to latency-critical threads result in less total data movement, it also reduces the amount of cache space needed to meet tail-latency deadlines, freeing resources for co-running batch applications. To determine the amount of cache space needed, Jumanji takes inspiration from prior work by using simple feedback control. We show that Jumanji's simple data-placement mechanisms, which build on prior work on tail latency and D-NUCA, efficiently co-optimize for multiple application objectives.

2.6 Cache Security

Hardware security has become a hotbed of discussion following the discovery of several severe microarchitectural vulnerabilities (e.g., Spectre [121] and Meltdown [135]). As cloud services colocate increasing numbers of customers on the same machines to improve resource utilization, the potential for cross-customer vulnerabilities also increases. Traditional hardware abstractions that provide software with the illusion of running on a dedicated system (e.g., virtual machines) fail to provide true isolation. This is particularly true for processor LLCs, which are physically shared by all cores. Isolation mechanisms such as disabling hyperthreading and reserving cores cannot prevent contention in the LLC. Accordingly, shared-cache attacks are major concern for commercial CMPs. This section addresses background on shared-cache attacks which allow an attacker either to learn a victim's access pattern through side channels [114, 115, 173] or harm a victim's performance.

2.6.1 Conflict attacks

Prior work mainly considers content-based timing side-channel attacks, specifically *conflict attacks* where an attacker attempts to detect a victim's access pattern. The prime+probe attack [138], a popular conflict attack for detecting which cache sets a victim accesses, operates as follows. The attacker prepares by creating an *eviction set*, a group of cache lines that fills all the ways of a target cache set. After *priming* the eviction set by loading it into the cache, the attacker waits until it expects the victim to have accessed its data. The attacker then times the access latency to *probe* each line in the eviction set. If any access goes to main memory, the long access latency indicates to the attacker that the victim also accessed the set, evicting a line from the eviction set. The prime+probe attack is quite effective, as it has been shown to leak entire AES keys [74, 98, 132].

Prior work offers many defenses for conflict attacks. Randomization-based defenses prevent an attacker from identifying which addresses a victim has accessed by randomizing the data layout within each shared cache bank [137, 188, 189]. Preservation-based defenses isolate the attacker's and victim's data, preventing the attacker from inducing conflict misses. Sharp [244] defends conflict attacks by modifying the LLC replacement policy, and RIC [114] defends conflict attacks on thread-private or read-only data by relaxing inclusion. However, way partitioning (i.e., Intel CAT [95]) is the simplest and by far the most common defense. Way partitioning restricts different processes to different cache ways, preventing partitioned processes from evicting each other's data and thus eliminating conflict attacks. Unfortunately, way partitioning reduces associativity, so *only a few partitions can be used before performance drops precipitously*. Consequently, prior way partitioning designs can only defend a small amount of data, which must be explicitly designated as sensitive by the OS [119, 136, 232]. Many alternatives to way partitioning face similar limitations [134, 139, 250] or do not guarantee isolation [148, 198, 231].

2.6.2 Additional cache attacks

The above techniques address conflict attacks, but they leave other LLC attacks undefended. In particular, *port attacks* exploit shared structures to leak information, as queueing delay reveals when a victim uses the shared structure [13, 31]. Caches' limited ports make them vulnerable to port attacks, which have been demonstrated in CPU L1 caches [103] and GPUs [104]. In Section 3.4.2, we demonstrate that CPU LLCs are also vulnerable to port attacks.

Moreover, in Section 3.4.3 we show that way partitioning offers incomplete performance isolation due to shared microarchitectural state in the replacement policy. This allows untrusted processes to harm a victim's performance, e.g., by causing missed deadlines.

The only prior defense against these attacks is IRONHIDE [170]. IRONHIDE is a secure enclave that splits a multicore into two clusters of tiles, "trusted" and "untrusted", and prevents all resource sharing across them. IRONHIDE defends LLC attacks, but it comes at a high price and with some disadvantages. For example, the enclave approach has limited scalability, since, e.g., each cluster requires its own memory controller (IRONHIDE supports just two clusters). Finally, IRONHIDE ignores application objectives such as tail latency and does not optimize data placement within each cluster to reduce data movement.

2.6.3 Our contributions vs. prior work on security

Our work tackles cache security from two different angles, providing stronger security than prior work and without harming, and actually even improving, application performance. täkō shows how opening the memory interface to software enables applications to detect attacks before any secure data is leaked. In täkō, applications can register their data with software callbacks that execute when the data is inserted into or evicted from the cache. Applications can then use the eviction callback to detect when their secure data is evicted, implying that, e.g., an attacker might be priming the cache for a prime+probe attack. Since no information is leaked before the probe stage, catching an attacker during the prime stage allows the victim process to interrupt itself before it accesses its secure data. This visibility over data movement enables applications to detect attacks before any secure information is leaked and impose no performance overheads in the common case.

Jumanji goes a step further by using data placement to isolate untrusted applications to fully defend against attacks. Since D-NUCAs physically separate data into different banks, Jumanji gives a complete defense against all of the above attacks. Moreover, we show that Jumanji defends conflict attacks with complete performance isolation and without the associativity-induced performance problems of prior partitioning defenses. To the contrary, Jumanji offers significant performance *gains*, as we leverage existing D-NUCAs to minimize data movement while meeting applications' tail-latency and security goals, unlike prior works on security which sacrifice performance (see Table 2.1).

2.7 Near-Data Computing (NDC)

Given the enormous cost of moving data to compute, alternatively, many architectures instead move compute closer to data. These data-centric designs, in contrast to traditional compute-centric designs, mitigate the costs of data movement by operating on data where they reside within the memory hierarchy. Avoiding the transfer of data across levels of the memory hierarchy and across on-chip networks can provide substantial performance and energy improvements when pulling all data to the core incurs excess data movement.

Data-centric architectures typically fall within two models: *near-data computing (NDC)* or *processing-in-memory (PIM)*. Whereas NDC places compute logic near cache or memory banks, PIM directly modifies data storage (e.g., data array or DRAM row buffer) [16, 46, 70, 107, 122, 123, 133, 168, 171, 176, 210, 216]. PIM can potentially achieve the highest performance and energy efficiency since operating directly on the bit cells essentially avoids all data movement, but it comes with a heavy drawback. The analog characteristics of PIM computation, especially on newer, less-studied memory technologies such as memristors, suffer from poor reliability [129, 249]. The need for abundant redundancy and error correction lessens PIM's attractiveness.

NDC, on the other hand, places more traditional compute logic (e.g., in-order core, application-specific integrated circuit (ASIC), field-programmable gate array (FPGA), or coarse-grained reconfigurable array (CGRA)) close to cache or memory banks [16]. This design still eliminates significant data movement with much less integration effort. Additionally, the compute is often simpler and more efficient than beefy out-of-order cores because they do not need to support the ability for executing entire complex applications. As a result, near-data computing is a very appealing solution for minimizing data movement.

2.7.1 In-cache NDC

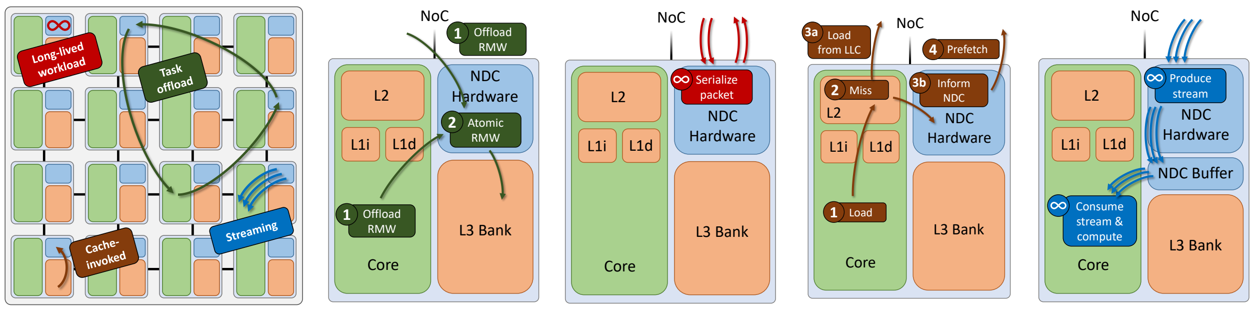
Where to add compute resources within the memory hierarchy is not a trivial decision. Modern memory hierarchies contain multiple levels of caches distributed across an on-chip network which are further backed by multiple memory banks (see Figure 1.1). Many NDC architectures are *in-memory* in that they place compute near main memory [33, 36, 67, 184]. This has become increasingly popular recently due to advances in high-bandwidth, 3D-stacked memory where a logic layer has direct access to many memory layers [10, 33, 34, 91, 184]. In-memory NDC performs very well for applications that exhibit minimal data reuse by avoiding the constant transfer of data over the memory bus. However, these designs forgo all the locality benefits provided by the cache hierarchy when workloads exhibit data reuse and can even perform worse than a conventional cache hierarchy without NDC [6, 90, 142, 224, 258]. Further, they complicate system integration by adding complexity to coherence and synchronization mechanisms [142].

In-cache NDC, on the other hand, benefits applications with data locality by augmenting a cache hierarchy with processing capabilities [2, 4, 7, 8, 12, 40, 55, 62, 89, 109, 113, 125, 127, 142, 145, 162, 178, 179, 200, 204, 209, 218, 226, 246, 254, 256, 257, 259–262]. These designs get the best of both worlds by moving compute closer to data while also exploiting locality, unlocking the full potential of data-centric computing. Despite the growing popularity of in-cache NDC, in-memory NDC has received much more attention. Accordingly, there is still plenty of available work in designing and refining in-cache NDC systems.

In-cache NDC still poses a challenge in deciding where in the cache hierarchy to compute. With cache banks spanning sizes and locations, the optimal location to compute can depend on application and workload characteristics. One example where location matters is with hardware compression, where data is stored compressed in main memory, and possibly also lower cache levels, to increase effective storage capacity [12, 62, 178, 179, 200, 226]. The more cache levels that are compressed, the more data they can store, but also the more often that data needs to be decompressed. Thus, the optimal cache level to perform decompression depends on application and workload characteristics. Clearly, an optimal NDC system would provide ample flexibility in where near-data computation can execute.

Table 2.2: Prior work involving near-data computing within the memory hierarchy.

NDC paradigm	**Prior work**
Task offload	remote memory operations (RMOs) [116, 205], Minnow [256, 257], hash tables [262], memoization [261], BSSync [127], pointer chasing [87, 91], data remapping [10], Compute Caches [2], Livia [142], Dist-DA [22]
Long-lived workloads	PageForge [214], SerDes [109, 183], garbage collection [145], COREx [66]
Data-triggered actions	prefetching [7, 8, 218, 254], compression [12, 62, 178, 179, 200, 226], HTM [259], coherence and synchronization [4, 55, 89, 125, 192, 209, 246, 260], Impulse [40], Tvarak [113], PHI [162], täkō [204]
Streaming	Stream Dataflow [169], Stream ISA [234], Stream Floating [236], Near-Stream Computing [235], Task Stream [51], Infinity Stream [233], HATS [159], SpZip [247], Cohort [237]



(a) Paradigms differ in *when*/*where* actions execute as well as communication patterns with cores.

(b) Task offload: cores push short operations to caches, e.g., atomic add.

(c) Long-lived: near-data thread avoids cache pollution, e.g., serialization.

(d) Data-triggered: compute when data moves, e.g., prefetching.

(e) Streaming: caches push data to cores, e.g., CSR traversal.

Figure 2.1: Breakdown of NDC taxonomy across paradigms.

2.7.2 A taxonomy of NDC

With the goal of developing a unified NDC system (Figure 2.1a), we explored the diverse prior work on near-data computing. We found that prior designs largely fall into one of four main paradigms:

- ***Task offload***. A core explicitly offloads a small amount of work into the memory hierarchy (e.g., atomic read-modify-write) and often expects a response quickly.
- ***Long-lived workloads***. A long-running thread within the memory hierarchy, typically processing a large amount of data (e.g., packet serialization) without frequent communication to or from cores.
- ***Data-triggered tasks***. Computation triggered when data moves through the hierarchy (e.g., prefetching). Tasks are short-lived and do not communicate at all with cores.
- ***Streaming***. A near-data producer generates a stream of data to be processed by a separate consumer (e.g., decoupled access-execute). Tasks are long-lived and communicate continuously with cores.

Table 2.2 gives examples of recent NDC designs and where they fit in this taxonomy. We will now take a deeper look into each paradigm.

Task offload. Task offload encompasses designs where a core or other near-data task offloads a small amount of work into the memory hierarchy to execute closer to a specific piece of data. The traditional example is remote memory operations (RMOs), where a core requests a single atomic operation to execute directly on the data within the cache or main memory [116, 205] (Figure 2.1b). This avoids the expensive ping-ponging of data between cores for heavily shared data. Over time, offloaded tasks have become increasingly complex, potentially involving many operations, multiple locations in the memory hierarchy, and tasks spawning additional tasks (e.g., for pointer chasing [87, 91]). A major challenge in these designs is dynamically determining the right location to execute a task; e.g., where is the data *now*?

Long-lived workloads. In contrast to task offload, long-lived workloads execute long computations that operate on large amounts of data. They run independently of cores and do not communicate directly with them (Figure 2.1c). Typically, applications in this paradigm perform some background processing, and run low in the cache hierarchy to avoid polluting private caches. One example is serialization/deserialization (SerDes), where an object is transformed near memory while the core continues to operate asynchronously [109, 183]. Long-lived workloads often want to execute at a fixed location in the memory hierarchy (e.g., LLC or memory controller). Accordingly, the NDC system needs to allow software to request a specific location for execution.

Data-triggered actions. These are actions triggered implicitly by data movement within the memory hierarchy, not explicitly by cores (as in the first two paradigms). Typically, the triggering mechanism is when data is inserted or evicted from a cache bank. A popular example is hardware prefetching (Figure 2.1d), where the prefetcher monitors cache misses and optionally triggers additional data requests before the underlying core needs the data.

The benefits of data-triggered actions are increased visibility and control over data movement. For example, hardware compression has been proposed to decompress data as it moves from main memory to the core, improving the effective capacity of main memory while avoiding the need to decompress data on cache hits [12, 62, 178, 179, 200, 226].

The unique characteristic of data-triggered actions is that they execute when data moves, which is traditionally invisible to software. Hardware support is required to trigger actions when data moves across levels of the memory hierarchy.

Streaming. Streaming is when applications access data in a pattern that can be decoupled from other application logic. Typically confined to simple affine patterns, recent work has proposed general-purpose streaming engines [169, 234–236] and sophisticated stream logic that supports complex, irregular access patterns [159, 254]. The benefits of streaming are that the stream producer can run ahead of consumers to hide memory latency, control flow is regularized on the consumer, and stream generation can often use simplified hardware logic.

The unique characteristic of streaming as an NDC paradigm is that the stream is long-lived within the memory hierarchy and communicates frequently with cores, pushing data and waiting for an acknowledgment that data has been finally consumed. Streams benefit from explicit ISA support for this frequent communication [234].

2.7.3 Benefits of supporting multiple NDC paradigms

Figure 2.1 separates the four NDC paradigms, but they need not operate independently. Prior NDC designs have demonstrated significant benefits from executing multiple paradigms at once.

PHI [162] combines task offload and data-triggered paradigms. PHI offloads atomic updates near-data (**task offload**) to avoid ping-ponging of data between private caches, which is important because PHI expects frequent concurrent updates. PHI also modifies cache insertion and eviction (**data-triggered**) to decide upon eviction how to apply updates, saving memory bandwidth.

Similarly, Near-Stream Computing (NSC) [235] combines both task offload and streaming. NSC observed that it is more efficient to process stream output near the cache than on a core. So NSC offloads tasks to the location where streams are produced, reducing data movement and avoiding pollution of cores' private caches.

Finally, Dist-DA [22] provides a flexible design for supporting task offload and long-lived workloads by providing a common mechanism for cores to offload work near caches.

2.7.4 Limitations of prior work

Despite providing large benefits, prior NDC designs suffer two major deficiencies: *scope* and *hardware abstraction*.

Limited scope. Every NDC design requires new hardware and interfaces across the system stack. The simplest designs are ISA extensions that enable single operations on cached data (e.g., RMOs); these are broadly useful and easy to implement. However, more complex tasks (e.g., SerDes) cannot be efficiently reduced to RMOs and require much more disruptive changes that benefit fewer applications. Recent programmable designs require the most disruptive changes of all and still only target a subset of the NDC design space [8, 22, 142, 169, 204, 234, 236, 257].

To justify the cost of adding new features to a general-purpose processor, features must benefit a wide range applications. It is simply infeasible to re-design hardware and software for every potential application of NDC.

Poor hardware abstraction. One of the consistent lessons in the history of computer architecture is the importance of *ease of programming* to the real-world success of hardware. Hardware details are typically abstracted away from application software, so that the programmer can focus on developing application features and only rarely worry about microarchitecture for performance-critical code. Exposing microarchitectural details, such as the cache's line size, is unnatural for a programming interface, but that is exactly what prior work on NDC does.

Since in-cache NDCs operate at cache banks, it is highly desirable for an action's data to reside entirely in one cache bank or tile. Prior NDCs have placed that burden on the programmer, forcing applications to properly align and pad data to cache lines or suffer massive performance penalties [40, 87, 142, 162, 204, 261, 262]. This low-level programming interface limits NDC to a narrow subset of programmers and adds additional burden when porting application code across microarchitectures.

2.7.5 Cost of specialization

Near-data computing is clearly a very popular design choice for reducing data movement, so why has it not been adopted in commercial products? The elephant in the room is that each proposed NDC requires adding custom logic to a general-purpose memory hierarchy, a very expensive endeavor. Taken literally, prior work suggests that memory hierarchies should contain an ever-growing number of custom accelerators. But this is unrealistic because each change to the hardware-software interface requires large, up-front investment in both hardware and software to be effective. Most accelerators benefit too few applications to justify such investment, creating innovation deadlock where large potential speedups cannot be realized in practice. Thus, optimizations are mostly limited to those that preserve the load-store interface, such as cache replacement policies or prefetchers.

A theme of our work is that the solution to this deadlock is to find a *general-purpose architecture* that supports a wide variety of data-movement features and optimizations. Only with wide applicability can the necessary hardware and software investments be justified. Architectures should expose more data movement to software, so that software can observe and optimize data movement itself. Software control of data movement offers enormous advantages over a hardware-only approach. Solutions can be better tailored to individual applications, and development cycles go from years to days. Although the upfront costs of a new hardware-software interface are formidable, *these costs are paid only once*, after which the marginal cost is reduced by orders of magnitude.

2.7.6 Programmable NDC

Some prior works have incorporated programmability with NDC designs. Early programmable data-triggered NDCs were explored in the '90s and focused on distributed cache coherence [4, 40, 89, 125, 192, 209]. More recent designs have added some programmability to the memory hierarchy for specific purposes: e.g., prefetching [8, 218] or compression [247].

Many stream accelerators enable the application to specify its access patterns [169, 234–236] These accelerators are designed to support common access patterns such as, e.g., affine, indirect, and pointer chasing. The issue is that their hardware only supports a hardened subset of patterns, so uncommon patterns (e.g., bounded depth-first-search in HATS [159]) are not possible.

The most flexible designs can execute code on programmable engines placed throughout the memory hierarchy [22, 142, 257]. These engines typically contain simple, programmable compute logic (e.g., in-order core, FPGA, or CGRA) to support arbitrary code without incurring excessive area or power overheads. Although these programmable designs are the closest to a general-purpose NDC system, they typically target only the task-offload NDC paradigm, greatly limiting their applicability.

2.7.7 Our contributions vs. prior programmable NDC

täkō's goal is providing a single architecture that can support a wide range of data-triggered NDC designs. Whereas prior work on data-triggered NDC either requires application-specific hardware or only targets a single objective (e.g., prefetching or compression), täkō enables software to execute arbitrary code in response to data movement. Taking inspiration from prior works that add near-data engines [22, 142, 257], täkō executes application code in response to cache misses and evictions, a mechanism sufficient for supporting many prior data-triggered NDCs. By demonstrating how software can implement data-triggered NDC, täkō is a step towards a general-purpose NDC system.

Leviathan then builds upon täkō and other programmable NDCs to provide an architecture that unifies all NDC paradigms discussed in Section 2.7.2. We observe that the near-data engines used in prior work can support each paradigm with simple hardware extensions and a simple programming interface. Leviathan is a truly general-purpose NDC system that gives software control over data movement within the cache hierarchy.

2.8 Actor-Based Reactive Programming

In the search for a single programming model that could enable software to implement all the NDC paradigms discussed in Section 2.7.2, we found that actor-based reactive programming provides a fitting framework for NDC. Reactive programming is a model for designing event-driven applications [18]. While traditionally geared towards large-scale distributed applications [194], reactive programming can be a good fit for any application that breaks down into units of work that often execute asynchronously from each other. Accordingly, we found that reactive programming could enable a clean description of NDC functions.

There are different variations of reactive programming, including, but not limited to, object-oriented [196, 202], actor-based [84, 194], functional [64], and imperative [59]. Object-oriented and actor-based are the best candidates for NDC because they focus on executing asynchronous functions on individual objects, as NDC often does. However, whereas object-oriented triggers work when an object's data is modified, NDC triggers work when data moves (data-triggered) or when explicitly requested (task offload, long-lived, and streaming).

In actor-based reactive programming, messages are sent to actors to trigger actions on the actors' data. Similarly, each NDC paradigm involves triggering actions, typically on a specific piece of data. Also, the flexibility permitted in message creation (e.g., core-triggered vs. data-triggered) and composition (e.g., variable number of arguments) enable all NDC paradigms to fit within the model.

2.8.1 Our contributions vs. prior actor-based reactive programming

Leviathan adopts an actor-based reactive programming model to enable software to implement NDC functionality. The actor design of data associated with asynchronously triggered actions accurately encompasses the NDC model.

Whereas prior work on actor-based reactive programming often targets large-scale distributed applications [194] or low-latency user-facing applications [18], Leviathan leverages the model for software-programmable NDC. An original motivation for reactive programming was simplifying the programming effort for designing complex, event-driven workloads [18]. In contrast, we use reactive programming because it provides a single, unified model under which different NDC paradigms all fit.

Chapter 3

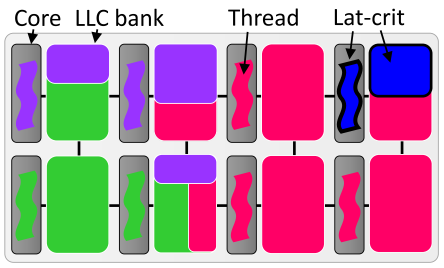
Jumanji: The Case for Dynamic NUCA in the Datacenter

The datacenter has become the dominant computing environment for many applications and will remain so for the foreseeable future. Its massive scale and multi-tenancy introduce new demands that systems were not originally designed for. Specifically, many datacenter applications are sensitive to **tail latency**, not raw computing throughput, since the slowest request out of many determines end-to-end performance [56]. Simultaneously, multi-tenancy means that applications now commonly run alongside untrusted applications. Following several high-profile breaches [121, 135], **security** has become a first-order concern for many datacenter customers.

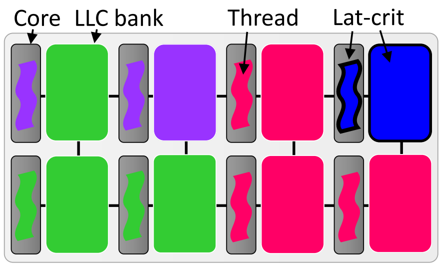
This work focuses on data movement at the shared last-level cache (LLC), a major factor in both tail latency and security. Data movement has a first-order effect on tail latency, as the time spent accessing data often sets the tail, and on security, as many attacks target shared state in caches.

The problem. Abundant prior work has tried to address these challenges, but offers incomplete or unsatisfactory solutions. Prior work in tail latency meets deadlines by reserving resources for latency-critical applications [45, 141], harming the performance of co-running "batch" applications without such deadlines (see Section 2.5). Prior work in security defends against microarchitectural side channels [121, 135–137, 173, 188, 232, 243, 253], but leaves some attacks undefended, harms application performance, or increases system complexity (see Section 2.6).

Setting tail latency and security aside, the most successful data movement techniques exploit the distributed nature of multicore caches, i.e., non-uniform cache access (NUCA), to keep data in close physical proximity to cores that access it [17, 28, 30, 79, 160, 222, 223]. *Dynamic NUCA* (D-NUCA) techniques yield large improvements in system throughput and energy efficiency. Unfortunately, as discussed in Section 2.4.2, prior D-NUCAs focus solely on reducing data movement, causing them to violate tail-latency deadlines and expose unnecessary cache side channels. This work revisits these techniques to see what they can offer in the datacenter. Our central message is that, with just a few small



(a) Jigsaw [28, 30].



(b) Jumanji.

Figure 3.1: Dynamic NUCA has natural advantages in the datacenter because it meets performance targets with fewer resources and physically isolates data from attackers. (a) However, Jigsaw, a state-of-the-art D-NUCA, is oblivious to tail latency and security, leading to missed deadlines and potential cache side channels. (b) With simple changes, Jumanji enforces tail-latency deadlines and defends side channels at similar performance to Jigsaw.

changes, *D-NUCA offers a superior solution for both tail latency and security* at the last-level cache.

Prior D-NUCAs ignore tail latency and security

Figure 3.1a illustrates how Jigsaw [28, 30], a state-of-the-art D-NUCA, places data in a multicore LLC. It depicts threads and data in an 8-core system, using colors to indicate different processes. Jigsaw tries to minimize overall data movement (both off-chip cache misses and on-chip network traversals) by placing applications' data in nearby LLC banks. We observe the following pros and cons of prior D-NUCAs:

Tail latency. By intelligently placing data near cores, D-NUCAs can meet a given performance target while using fewer resources. This frees up cache banks for other applications to use. We find that, with the right allocations, D-NUCA can *meet tail-latency deadlines with significantly higher batch performance* than prior techniques.

However, prior D-NUCAs like Jigsaw are oblivious to applications' goals (e.g., tail-latency deadlines), so they perform poorly on latency-critical applications. For example, at low load, latency-critical applications generate few LLC accesses, so Jigsaw tends to shift resources away from them to reduce data movement of batch applications. While such decisions may make sense from a data movement perspective, they cause latency-critical applications to miss their deadlines, harming overall system performance. It is therefore inadequate for D-NUCAs to focus exclusively on data movement—*D-NUCAs must incorporate applications' goals.*

Security. By clustering data near cores, D-NUCA naturally avoids sharing cache state between applications. As a result, D-NUCA can offer *stronger isolation* between applications than conventional cache partitioning, since data reside in physically separate cache banks. This makes it difficult for attackers to observe or interact with victims' cache accesses, simply because they do not share any cache with them.

D-NUCAs can thus solve two security flaws with NUCA-oblivious LLC designs. First, as we show in Section 3.4, LLCs are vulnerable to timing attacks on shared cache ports. Prior secure LLC designs do not defend this attack. Second, we show that standard partitioning defenses offer imperfect performance isolation due to leakage through the shared cache replacement policy, and also significantly harm performance by lowering associativity. D-NUCA can avoid all of these problems by placing untrusted applications' data in different LLC banks.

Unfortunately, prior D-NUCAs do not specifically target security, so these benefits so far arise only as a happy accident and cannot be relied upon by datacenter customers.

Jumanji: Redesigning D-NUCA for tail latency and security

We design a new D-NUCA called Jumanji to capitalize on the above advantages while addressing the disadvantages. Figure 3.1b shows how Jumanji's allocations differ from Jigsaw. Jumanji enforces tail latency by reserving enough cache space for each latency-critical application to meet its deadlines, using feedback control [45, 141]. Since data placement significantly reduces data movement, Jumanji actually meets deadlines with much less cache space than prior work, freeing cache space to accelerate batch applications. Jumanji enforces security by placing data from untrusted applications, e.g., from different virtual machines (VMs) [136], in different banks, guaranteeing strong isolation between untrusted applications. Jumanji further optimizes data placement within each VM's allocation to minimize data movement for each application.

Table 2.1 compares Jumanji against prior LLC designs in terms of tail latency, security, and batch performance. Jumanji gets the best of all worlds: it meets tail-latency deadlines, defends a wide range of cache attacks, and nearly matches Jigsaw's speedup. Jumanji is the only design that meets all of these objectives. Moreover, Jumanji achieves these benefits by leveraging prior D-NUCAs to simplify its implementation, requiring only a few, simple changes in software over Jigsaw.

Contributions

Our message is that D-NUCA offers superior performance and security for datacenter applications than existing techniques. Specifically, we contribute the following:

- We present Jumanji, the first D-NUCA designed for tail latency and security. Jumanji achieves these goals with ***better performance and energy efficiency*** than prior solutions. Moreover, Jumanji is ***practical***, requiring only a few simple software changes to existing D-NUCAs.
- We show that Jumanji ***meets tail-latency deadlines*** with significantly less cache capacity than prior work, freeing space for other applications. As a result, Jumanji significantly improves batch performance.
- We show that Jumanji offers ***stronger security*** than prior secure LLC designs. We give the first demonstration of an LLC port attack and of performance leakage in a strictly partitioned LLC. Jumanji defends all LLC attacks,

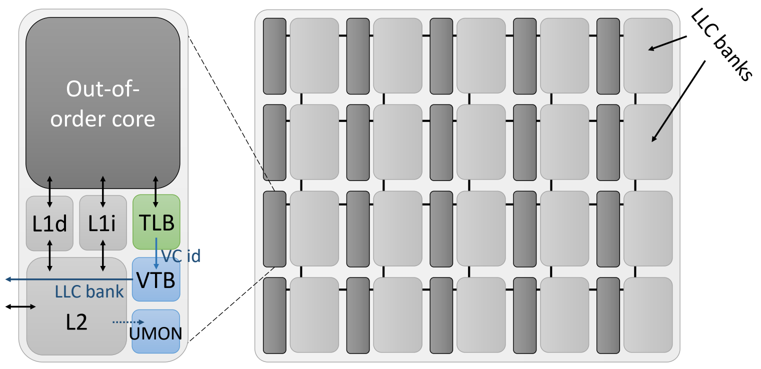


Figure 3.2: A 20-core system with a distributed LLC (20×1 MB banks). Jumanji adds simple hardware to control data placement, borrowed from Jigsaw [28, 30]. Green indicates **modified components**, and blue indicates **new components**.

including conventional content-based attacks and these new ones, with much better performance than prior designs.

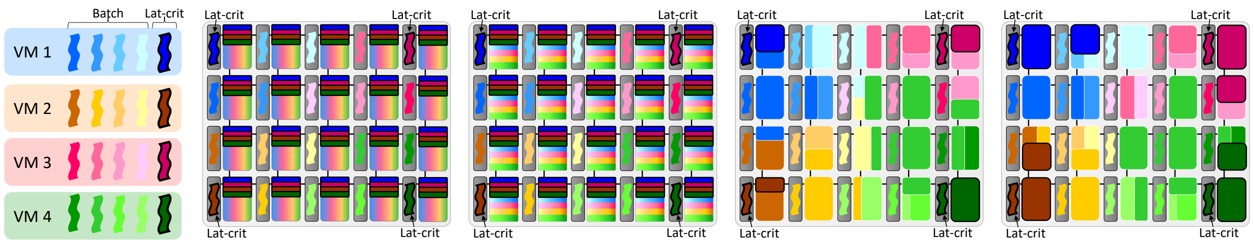
- We evaluate Jumanji in microarchitectural simulation on a 20-core CMP running mixes of batch and latency-critical applications. We show that Jumanji speeds up batch applications by 11%–15%, vs. 11%–18% for Jigsaw and 0%–4% for NUCA-oblivious designs, and that Jumanji comes within 2% of the batch performance of an idealized design that eliminates competition between batch and latency-critical applications.

3.1 Motivation

System context. Jumanji is focused on the datacenter environment, where applications often run in virtual machines (VMs) or containers alongside other untrusted VMs. This *multi-tenancy* is important to improve utilization and reduce costs, since datacenter applications often run at low utilization to keep queueing low for latency-critical applications [21, 58, 152].

Multi-tenancy causes challenges for performance and security. Datacenters run a wide range of workloads with different goals and characteristics, demanding architectures that perform well in a wide range of scenarios. Co-running VMs can cause performance interference, particularly at the tail, demanding effective resource partitioning. Co-running VMs are also untrusted, demanding robust and universal security.

This work presents Jumanji, a new D-NUCA that meets all of these demands in a single, simple design. Figure 3.2 shows the multicore that we consider in this work: 20 out-of-order cores share a 20 MB LLC that is distributed into 20 banks over a mesh network-on-chip (NoC). (See Section 3.5 for details.) To this base multicore, Jumanji adds D-NUCA hardware that controls where data is placed in the LLC, as detailed in Section 3.2.1.



(a) Workload. (b) Adaptive [45, 141]. (c) VM-Part [136, 173]. (d) Jigsaw [28, 30]. (e) Jumanji.

Figure 3.3: Representative data placements for a workload (a) with four VMs running a mix of latency-critical and batch applications on different LLC designs. (b) Adaptive dynamically adjusts latency-critical allocations to meet deadlines with minimal LLC space, but data is far away from threads. (c) VM-Part additionally partitions LLC space between VMs to avoid conflict side-channel attacks, but is vulnerable to new attacks on other shared cache structures. (d) Jigsaw places data to minimize data movement, ignoring tail latency and security. And (e) Jumanji, which places data to meet deadlines and defend side channels with minimal data movement.

3.1.1 Case study

To see how Jumanji improves upon prior work, we now consider an extended case study, illustrated in Figure 3.3. The workload is shown in Figure 3.3a: four VMs share the 20-core system, each running one latency-critical application (`xapian` from TailBench [112]) and four batch applications (randomly chosen from SPEC CPU2006); we show that other workloads yield the same conclusions in Section 3.6.

To see how different LLC designs behave on this workload, the remaining plots in Figure 3.3 depict where threads and data are placed in each design. Each VM is represented as a different color (blue, brown, pink, and green), and applications within each VM as different shades of this color. Threads are clustered in quadrants, with latency-critical applications running in the corners. LLC banks are colored to show where data is placed. Latency-critical applications are highlighted with a black border.

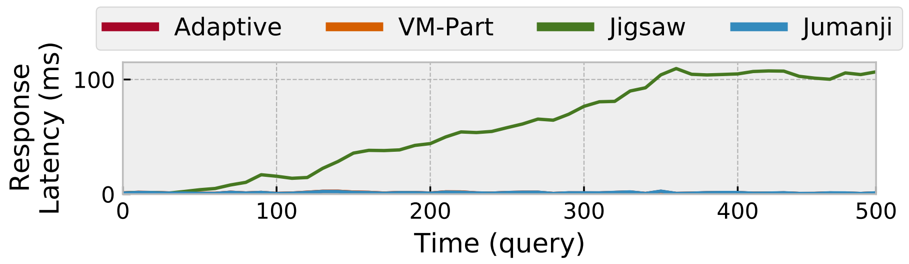
We consider the following LLC designs:

• **Adaptive** (Figure 3.3b) reserves space in each bank using way partitioning [95] and dynamically adjusts the allocations through feedback control [45, 141]. Adaptive partitions latency-critical data to guarantee tail latency is kept low, but it does not partition batch data because doing so lowers LLC associativity. Note that Adaptive is a *static* NUCA design (i.e., it spreads each application's data across all LLC banks), so data is far away from cores on average.

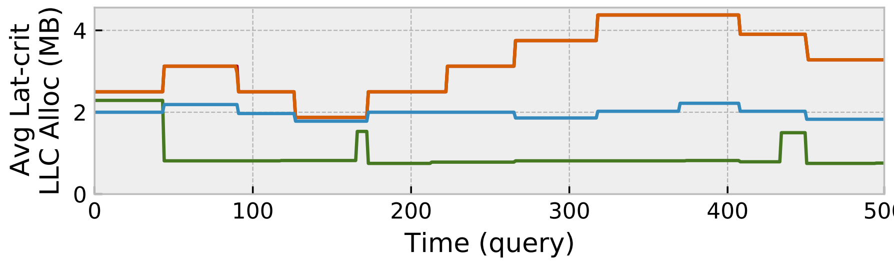
• **VM-Part** (Figure 3.3c) is a similar static NUCA design that, in addition to reserving space for `xapian` exactly like Adaptive through feedback control, also partitions batch data from each VM within each LLC bank. This partitioning defends against conflict timing attacks (Section 2.6.1), but lowers LLC associativity and thus harms batch performance.

• **Jigsaw** (Figure 3.3d) is a state-of-the-art D-NUCA that minimizes data movement [28, 30], but ignores tail latency and security. Jigsaw places data in LLC banks near threads, and partitions data within each bank for performance isolation.

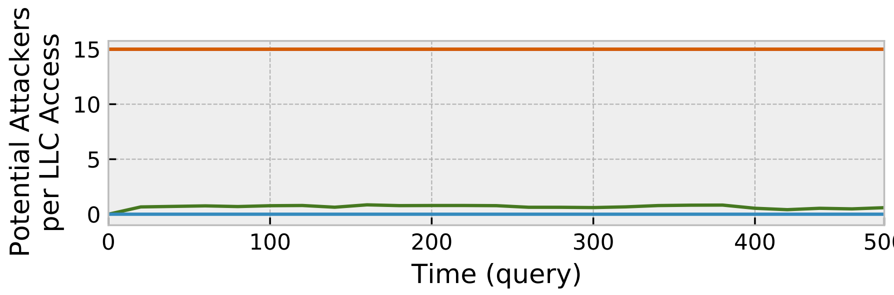
• **Jumanji** (Figure 3.3e) is our new D-NUCA design that targets applications' tail-latency and security goals. Like



(a) Avg end-to-end query latency for 4 `xapian` instances.



(b) Avg LLC allocation for 4 `xapian` instances.

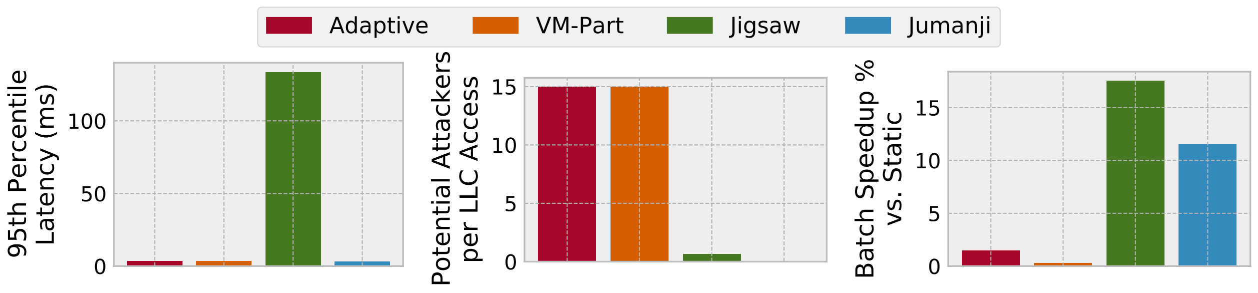


(c) Vulnerability to shared-cache-structure attacks (Section 3.4.1).

Figure 3.4: How different LLC designs behave over time. All but Jigsaw meet tail deadlines, but Adaptive and VM-Part need more space than Jumanji. Jigsaw and Jumanji improve security by physically isolating VMs' data.

Adaptive, Jumanji reserves space for latency-critical applications using feedback control to meet their deadlines. Like VM-Part, Jumanji isolates data from different VMs; in fact, Jumanji gets stronger isolation by *never sharing LLC banks* across VMs. Like Jigsaw, Jumanji places data near threads to minimize data movement.

Jumanji meets tail deadlines with much less LLC space, whereas Jigsaw badly violates deadlines. Figure 3.4 quantifies how each LLC design behaves over time, in terms of tail latency, LLC space, and security. Figure 3.4a shows `xapian`'s request latencies. All designs maintain low tail latency, except for Jigsaw, whose latency grows increasingly large over time. Figure 3.4b explains why by plotting how much LLC space is reserved for `xapian` in each design (averaged across VMs). Unlike the others, Jigsaw gives `xapian` very little space. This is because latency-critical applications run at low utilization to avoid queueing, and thus tend to generate little data movement. So Jigsaw, which cares only about data movement, tends to de-prioritize latency-critical applications and allocate them little LLC space. Jumanji fixes this problem by giving `xapian` enough space to keep the tail low. Moreover, Jumanji meets tail-latency deadlines with less space than Adaptive or VM-Part because Jumanji places data close to threads, letting a smaller



(a) Tail latency. (b) Security. (c) Batch speedup.

Figure 3.5: Jumanji meets tail-latency deadlines and defends side channels much more efficiently than non-NUCA approaches.

allocation achieve equivalent performance (see Section 3.3.1).

Jumanji improves security by physically separating VMs' data in distinct LLC banks. Next we consider security. All designs except for Adaptive partition LLC space among VMs, and so defend against conventional conflict timing attacks (Section 2.6.1). However, since VM-Part is an S-NUCA design with limited associativity, it pays for this security with lower batch performance. This work also considers an attacker that observes a victims' LLC accesses through other shared cache structures, e.g., cache ports (see Section 3.4.1). For such attacks to succeed, *the attacker only needs to share an LLC bank with the victim.* Figure 3.4c quantifies how vulnerable each design is to such an attack by plotting the number of untrusted applications that share an LLC bank when a victim accesses it, averaged across all applications and LLC accesses (higher is worse). Way partitioning is no defense against such attacks, so S-NUCA designs fare badly—*all* untrusted applications can potentially observe *every* access. D-NUCA offers a natural mitigation against this attack by clustering data near threads. In Jigsaw, many fewer untrusted applications can observe each access, but this benefit arises only as a by-product of minimizing data movement. Jumanji strongly enforces this constraint, never sharing banks between untrusted VMs, so that no untrusted application can ever observe an access.

Jumanji gets the best of all worlds. Figure 3.5 shows end-to-end results for this case study. Both Adaptive and VM-Part meet tail-latency deadlines, but get negligible batch speedup. Jigsaw improves batch performance, but causes unacceptable tail-latency violations. Jumanji meets tail-latency deadlines, nearly matches Jigsaw's speedup, and improves security by never sharing banks across VMs. Jumanji is thus a superior design for tail latency and security: it meets deadlines with better batch performance, while defending more attacks.

3.2 Jumanji's Design in a Nutshell

Figure 3.6 gives a high-level overview of Jumanji's design. Several VMs run in userspace, each with some mix of latency-critical and batch applications. Latency-critical applications inform Jumanji's low-level OS/hypervisor runtime of their tail-latency requirements and when each request completes, and all applications inform Jumanji of their "trust

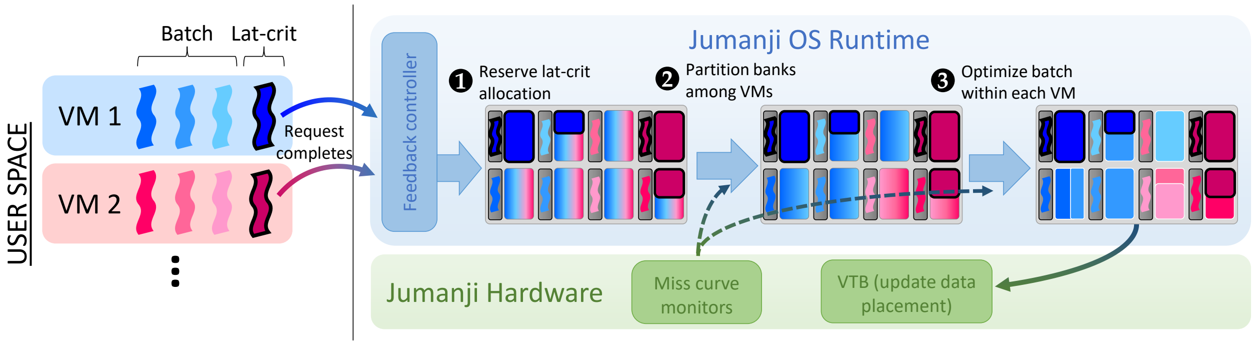


Figure 3.6: Jumanji periodically reoptimizes data placement to ensure that tail-latency deadlines are met and side channels are eliminated. ❶ A feedback controller reserves space for latency-critical applications to meet their deadlines. ❷ Data from untrusted applications is physically separated into different LLC banks. ❸ Data placement is optimized within each VM to minimize data movement.

domain" (e.g., the VM they belong to [136]).

Jumanji has software and hardware components. Jumanji's hardware is simple and borrowed from Jigsaw, as described below. Jumanji's changes lie in the software layer, where the OS periodically (every 100 ms) places data to enforce applications' tail-latency and security goals while minimizing data movement. Jumanji is designed to be practical by reusing proven techniques to simplify its design. Jumanji's placement algorithm has three broad steps:

❶ Jumanji reserves space for each latency-critical application in nearby LLC banks, using a feedback controller. *This step ensures tail-latency deadlines are met.*

❷ Jumanji partitions the remaining LLC banks among VMs. *This step defends against cache attacks while minimizing off-chip data movement.*

❸ Jumanji optimizes batch data placement within each VM's banks. *This step minimizes on-chip data movement.*

These steps complete quickly in software, taking a negligible fraction of system cycles. Once the new allocation is found, Jumanji installs the new placement in its hardware.

3.2.1 Jumanji hardware

Jumanji borrows Jigsaw's D-NUCA hardware without modification [28, 30]. For readers unfamiliar with Jigsaw, we now briefly describe how it works. Figure 3.2 depicts the hardware Jumanji adds to control data placement. Jumanji is a single-lookup D-NUCA that places data at page granularity (Section 2.4.2).

Controlling data placement. Jumanji maps each page to a *virtual cache* (VC), a new OS abstraction for managing data placement. For this work, it suffices to think of there being one VC per application [160, 223]. The OS controls page mappings via the page table, and the TLB is extended to store each page's VC id. Each core also contains a *virtual-cache translation buffer* (VTB) that determines which LLC bank holds a memory address for a given VC. Its operation is illustrated in Figure 3.7. The VTB maps a VC id to a *placement descriptor*, a 128-entry array of bank ids.

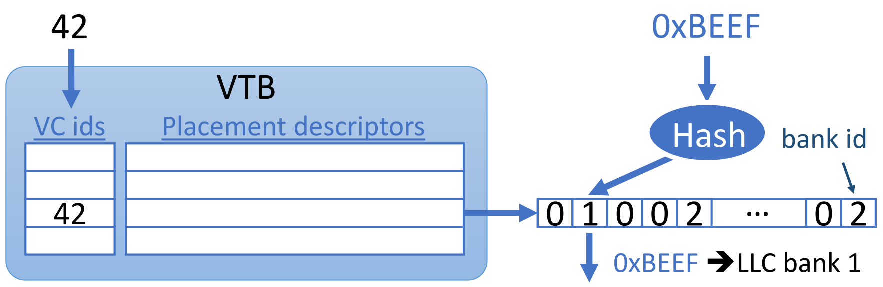


Figure 3.7: The virtual-cache translation buffer (VTB) controls data placement. Addresses are hashed to index into the VC's placement descriptor, yielding the address's unique LLC location.

The LLC bank is determined by hashing the address to index into the VC descriptor. Software can therefore control where each VC is placed in the LLC by setting the entries in its VC descriptor.

Coherence. Jumanji maintains coherence when pages change VCs or data placement changes without pausing thread execution. Each LLC bank walks its array and, in the background, invalidates lines that have moved. This requires lightweight hardware; for details, see [28, 30].

Monitoring miss curves. To intelligently place data, Jumanji needs to know how each VC is accessed. Jumanji profiles this in hardware through *utility monitors* (UMONs) [30, 186], which sample $\approx$1% of accesses to determine how many LLC misses the VC would incur at different allocation sizes.

Hardware overheads. In all, Jumanji adds small hardware overheads to the baseline multicore. The necessary logic is simple (e.g., table lookups and hashing), and the area of the VTB and UMONs are dominated by data arrays, which store 9 KB of state in total (8 KB for UMONs and less than 1 KB for the VTB). This is less than 1% of the per-tile cache storage.

System hardware changes. To better reflect production systems, we use way partitioning [95] and DRRIP replacement [101] within each LLC bank, instead of Vantage partitioning [198] and LRU as in Jigsaw's evaluation. We approximate DRRIP's miss curve by taking the convex hull of LRU's miss curve, which can be measured much more cheaply [29, 230].

3.2.2 OS integration

Software runtime. Jumanji operates as a low-level software runtime tightly integrated with the VM hypervisor. Every 100 ms, Jumanji's runtime executes a data placement algorithm (detailed in Section 3.3 and Section 3.4) to reconfigure all applications' LLC allocations and placements. New data placements are installed by updating each core's VTB, and allocations within each bank are enforced through way partitioning (e.g., Intel CAT). When threads migrate across cores, Jumanji migrates their LLC allocations along with the threads, like prior D-NUCAs [30, 79].

Jumanji requires integration with the VM hypervisor to know which applications run within each VM. This allows

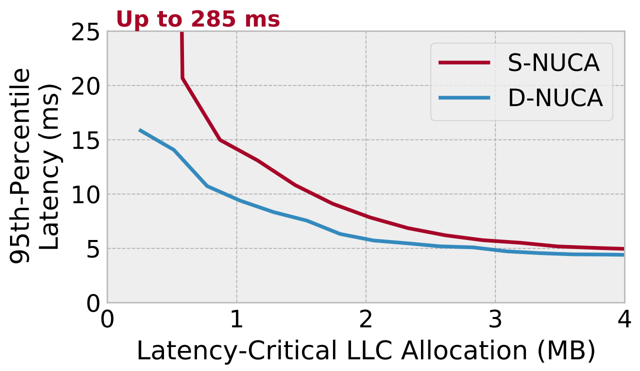


Figure 3.8: How `xapian`'s tail latency varies with its cache allocation, with and without D-NUCA. D-NUCA lets `xapian` meet its tail-latency deadline with much less LLC space.

Jumanji to isolate VMs into distinct LLC banks. When launched, new VMs are provided a small LLC allocation (e.g., one bank) along with their core(s) until Jumanji's next reconfiguration determines their LLC allocation. If VMs exceed the number of LLC banks, then multiple VMs by necessity must share an LLC bank, potentially compromising security (Section 3.4.1). Like other secure designs [170], Jumanji handles this by flushing the shared cache on context switch—but note that only the LLC banks shared with the swapped-in VM must be flushed.

Software overheads. Jumanji's overheads are small. We measured the execution time for Jumanji's placement algorithm across all evaluated experiments. Jumanji's placement algorithm runs once every 100 ms and takes 11.9 Mcycles on average. This corresponds to negligible execution overhead of 0.22% of system cycles ($= 11.9\,\text{Mcycles}/[20\,\text{cores} \times 100\,\text{ms} \times 2.66\,\text{GHz}]$), which only affects batch performance and is included in our results. More frequent reconfigurations do not improve results.

3.3 Jumanji for Tail Latency

We first motivate D-NUCA for meeting tail-latency deadlines. Then we introduce a simple software algorithm which uses Jumanji's hardware to manage latency-critical applications more efficiently than prior, non-NUCA approaches.

3.3.1 Motivation: Why D-NUCA for tail latency?

First, we evaluate the effect of D-NUCA on `xapian`'s tail latency independent of batch applications. Figure 3.8 shows `xapian`'s tail (95th-percentile) latency when `xapian` is allocated different portions of the LLC. The red line shows tail latency when allocations are set using way partitioning (i.e., Intel CAT), which spreads data around all LLC banks. The blue line shows tail latency when allocations are reserved in the closest LLC banks. (See Figure 3.3b and Figure 3.3e.)

Figure 3.8 illustrates two important points. First, as found in prior work [111, 141], cache allocations have a large impact on tail latency. In S-NUCA, moving from a large allocation to a small allocation degrades tail latency by up to 50×. This is because the request arrival rate exceeds the system's service rate, yielding unbounded queueing latency.

```
def RequestCompleted(latency, app):
  latencies[app].append(latency)
  if latencies[app].size() > configurationInterval:
    tail = getPercentile(latencies[app], 95)
    latAppSize[app] = ctrl.update(tail, deadline, app)
    latencies[app].clear()
```

Figure 3.9: The OS is updated every time a latency-critical request completes. Once the number of completed requests exceeds a configurable interval, the feedback controller updates the size allocated for that latency-critical application.

Second, which has not been considered in prior work, *NUCA also has a large impact on tail latency.* When allocations are placed in nearby banks, `xapian` can meet tail-latency deadlines with much less space than in S-NUCA. For example, `xapian`'s tail latency with a 2 MB D-NUCA allocation is the same as 3 MB with S-NUCA. D-NUCA thus frees 1 MB for other applications to use, while saving energy by reducing on-chip data movement. Tail latency also degrades more gracefully with D-NUCA than without; D-NUCA's worst-case latency is roughly 18$\times$ lower than S-NUCA's.

3.3.2 Jumanji's OS interface

Similar to prior work on tail latency (Section 2.5), Jumanji extends the system-call interface to let system administrators register latency-critical applications and let these applications report their tail-latency deadline and when requests begin and complete. Jumanji asks applications to share their performance goals, not desired resource allocations, to reduce waste from over-provisioning [58]. Jumanji takes responsibility for allocating resources to meet these goals. Jumanji runs multiple latency-critical applications together on the same multicore system and places them as far apart as possible to minimize LLC contention. A better mapping may be possible [30], but that is outside the scope of this work.

3.3.3 How much LLC space do latency-critical applications need?

Jumanji uses a simple feedback controller to decide how much of the LLC to allocate to each latency-critical application. When a request completes, the OS buffers its response latency (including queueing delay). If it has seen enough requests to determine the tail latency of recent requests (e.g., 20 requests for 95th-percentile latency), then it updates the feedback controller with this tail latency and adjusts the application's allocation. Figure 3.9 gives pseudocode for this procedure.

The controller increases the application's allocation by 10% if tail latency exceeds 95% of the deadline, and reduces it by 10% if it is below 85% of the deadline. If tail latency exceeds the deadline by 10%, the controller "panics" and boosts the allocation to a canonical, safe size (one-eighth of the LLC in our experiments). This boost is necessary

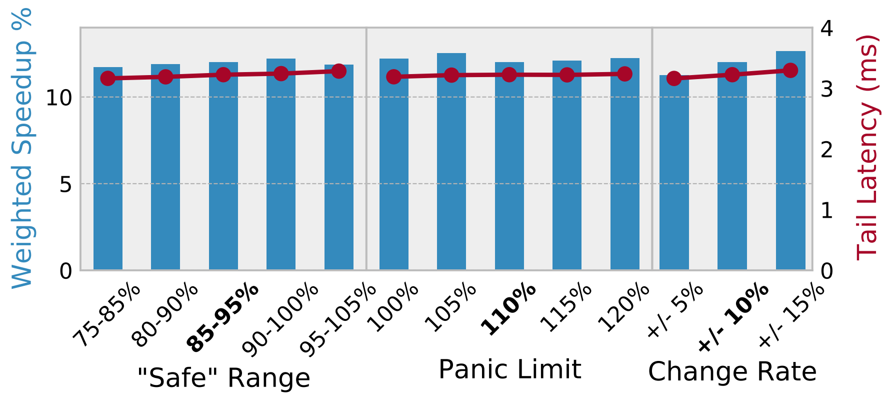


Figure 3.10: Variation in speedup and latency as parameter values for the feedback controller change. Jumanji is insensitive to values.

because we find that even very short spikes in queueing latency frequently set the tail. Alternatively, we could use queue length, but that would require additional information from applications [110].

Controller sensitivity. Jumanji is minimally sensitive to the feedback controller's parameters, letting Jumanji use a single set of parameters across many different latency-critical applications. Figure 3.10 shows gmean weighted speedup (bars) and tail latency (lines) for the same workload as Figure 3.5, varying one parameter at a time. The first group varies the target latency range, the second group varies the panic threshold, and the last group varies the step size. Results change very little across parameter values; we use the bolded parameter values in our experiments.

3.3.4 Placing latency-critical allocations in the LLC

Once Jumanji's software knows how much space to give each latency-critical application, Jumanji next greedily places latency-critical allocations to prevent batch applications from claiming the space. This placement is sub-optimal for batch throughput, but it ensures that tail-latency deadlines are met.

Figure 3.11 gives pseudocode for Jumanji's algorithm. Jumanji's `LatCritPlacer` first sorts LLC banks for each latency-critical application by distance from the application according to the NoC topology. (Jumanji's algorithms are topology-agnostic.) Then it simply grabs space in the closest banks until it has placed all `latAppSize`'s space. All remaining space is left for batch applications (we will optimize batch placement below, ensuring security across VMs).

This greedy algorithm is simple, but surprisingly effective and *leaves little room for improvement*. We explored a more sophisticated (and significantly more complicated) algorithm that trades cache space between batch and latency-critical applications after placing batch data, moving batch data closer while compensating latency-critical applications. We omit this algorithm because its gains were marginal over the much simpler `LatCritPlacer` and because, as Section 3.6.3 shows, Jumanji's batch performance with this greedy placement is already close to an idealized design.

```
def LatCritPlacer(bankBalance): # capacity per bank
  orderedBanks = sortBanksByDistance(latApps)
  foreach latApp:
    preferredBanks = orderedBanks[latApp]
    latAppAlloc = latAppSize[latApp] # set by feedback controller
    while latAppAlloc > 0: # allocate greedily
      bestBank = preferredBanks.next()
      allocSize = min(bankBalance[bestBank], latAppAlloc)
      allocs[bestBank][latApp] = allocSize
      latAppAlloc -= allocSize
  return allocs
```

Figure 3.11: Jumanji uses feedback control to determine each latency-critical application's allocation, then places this allocation nearby. All remaining cache space is allocated for batch applications. Every 100 ms, the OS invokes this algorithm to produce a matrix `allocs[b][a]` which denotes how much space application a is allocated in LLC bank b.

The resulting placement meets tail-latency deadlines, unlike prior D-NUCAs. However, `LatCritPlacer` provides no better security than Jigsaw and has not yet optimized batch data placement. We address these limitations next.

3.4 Jumanji for Security

After ensuring tail-latency deadlines are met, Jumanji defends against cache attacks. This section describes our threat model, discusses why D-NUCA improves security, demonstrates a new LLC attack, and explains how Jumanji defends against LLC attacks while improving performance.

3.4.1 Threat model

Jumanji targets datacenters where a single machine is shared by several VMs. As in prior work [136], processes in the same VM trust each other, but not processes from other VMs. VMs allow users to share hardware, but users expect their data to be secure from attack by other users.

We are concerned with cache attacks across VMs at the shared LLC, which is distributed into banks over a NoC. LLC architecture is complex and shares several components. Figure 3.12 illustrates the attacks that we consider.

1 **Conflict attacks.** An attacker exploits the presence or absence of data in the shared cache to determine the victim's access pattern. This is the standard cache side-channel attack, which has many defenses (as discussed in Section 2.6.1). Jumanji's advantage for conflict attacks is that it has much higher effective associativity than conventional way partitioning, letting it defend *all* data while maintaining high performance.

2 **Port attacks.** An attacker exploits queueing at shared cache ports to determine when a victim accesses a cache bank. To the best of our knowledge, we are the first to demonstrate a port attack at the LLC. Port attacks are not defended by prior defenses for conflict attacks.

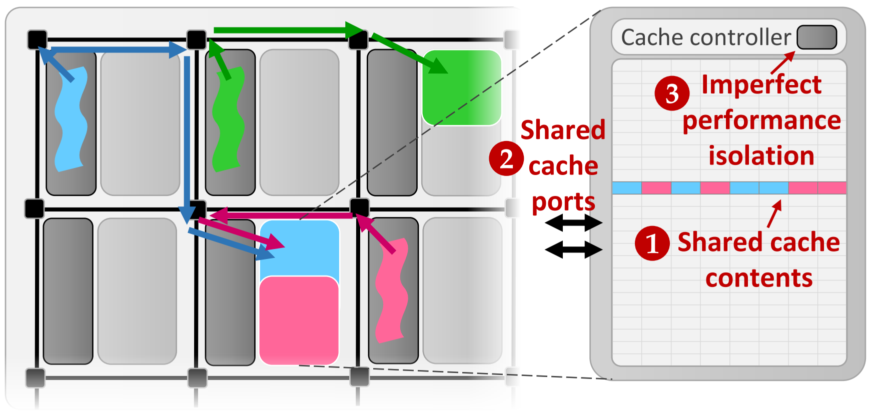


Figure 3.12: The three shared cache components considered in this work. ❶: *Conflict attacks* through shared cache sets. ❷: *Port attacks* through shared bank ports. And ❸: *Imperfect performance isolation* through adaptive cache replacement state.

❸ **Performance leakage.** Finally, we discovered that standard partitioning-based defenses do not offer strong performance isolation due to shared microarchitectural state in the replacement policy. This can allow an attacker to, e.g., cause a victim to miss its tail-latency deadlines.

Note that way partitioning like Intel CAT [95] does *not* defend against attacks ❷ and ❸, since it does not separate data into different banks. The rest of this section explores these attacks and explains how Jumanji defends them.

3.4.2 Demonstration of an LLC port attack

Cache banks have a limited number of ports [118]. Independent of attacks that depend on shared state within a bank, contention on shared cache ports is another timing side channel that lets an attacker observe a victim's memory accesses. Since prior preservation and randomization defenses (Section 2.6.1) build on an S-NUCA baseline, untrusted applications still share LLC banks, leaving *port attacks undefended*. Such an attack is noisier than conflict attacks because it relies on queueing, but we now show it is feasible on current processors.

Figure 3.13 demonstrates an LLC port attack on an Intel Xeon E5-2650 v4. An attacker thread constantly floods a target cache bank, using the algorithm in [138], and records the time to complete every 100 LLC accesses (to amortize timing overheads). Figure 3.13 displays measured access times vs. wall-clock time, where darker color indicates a larger number of measurements at that value. Outliers (<0.1% of accesses) are excluded.

The victim is a multi-threaded process (with 3 threads) that, for demonstration purposes, rotates through flooding each LLC bank, pausing in between banks for several million cycles. Since the Xeon E5-2650 has twelve LLC banks, this gives rise to twelve peaks in Figure 3.13. Note that the victim accesses a *different* cache set from the attacker to guarantee that contention does not occur from the cache contents.

Figure 3.13 shows that latency increases whenever the victim is active due to NoC contention, but delay is noticeably

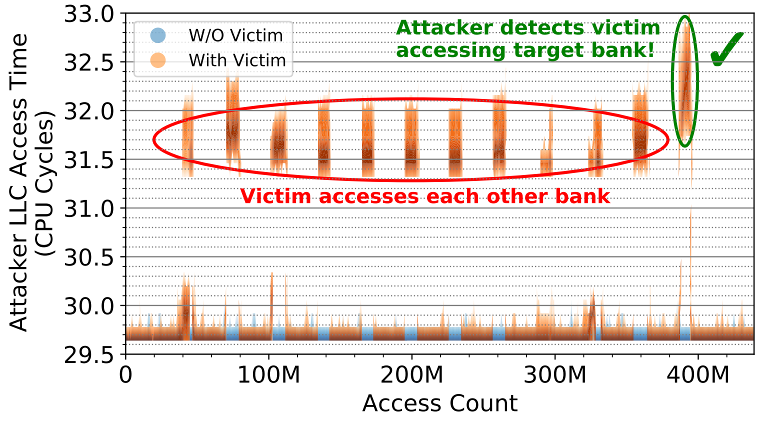


Figure 3.13: LLC access times for an attacker flooding a target LLC bank with accesses (with and without a co-running victim process). The victim accesses each LLC bank, causing 12 spikes in latency for the attacker. The attacker detects victim accesses to a target bank by higher access times due to port conflicts.

higher when the victim accesses the same bank as the attacker (avg. time $>$ 32 cycles). This result is clear and consistent across runs, demonstrating that port attacks are viable at the LLC. Such port contention could be realized in practice through microarchitectural replay attacks [215], frequent coherence misses to shared data among victims, or frequent evictions if the victim has a small LLC partition.

3.4.3 Performance leakage and degradation in way partitioning

Way partitioning is the most common defense against LLC attacks. Here we briefly discuss some additional limitations of way partitioning as an LLC defense, which Jumanji solves “for free” (i.e., at no performance loss or added complexity) while defending port attacks.

Performance leakage, even with partitioning. Modern, adaptive cache replacement policies dynamically switch policies using set-dueling [101, 187]. Since set-dueling chooses between policies at cache-bank granularity, all applications accessing a bank both influence which policy is used and are impacted by the chosen policy, regardless of partitioning mechanisms. Hence, interactions between processes in set-dueling’s shared counters can let VMs affect each others’ performance *even when the VMs are isolated into partitions*.

Figure 3.14 demonstrates the impact of this performance leakage. `img-dnn`, a latency-critical application from Tailbench [112], is run alongside different mixes of batch applications with DRRIP replacement [101]. The red line plots tail latency, normalized to `img-dnn` running alone, on S-NUCA with a 2.5 MB fixed LLC partition, sorted from best to worst for 40 random mixes of SPEC CPU2006 applications. For each mix, `img-dnn` has the same, static LLC partition, yet its tail latency varies significantly depending on the co-running batch applications. The result is tail-latency violations, sometimes exceeding 10%.

In contrast, the blue line plots tail latency when `img-dnn` is allocated the two closest 1 MB banks (like Jumanji with

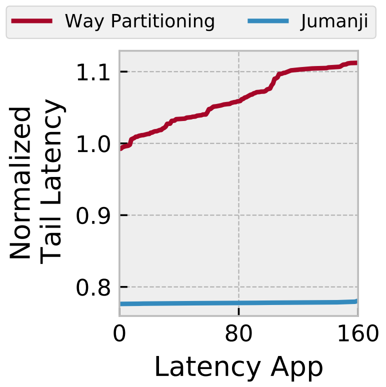


Figure 3.14: Tail-latency distribution for four instances of `img-dnn` when run alongside 40 batch mixes with a fixed LLC partition.

a fixed allocation). Tail latency is stable, independent of co-running batch applications, and 20% lower than S-NUCA, even with a smaller partition.

Universal defense of conflict attacks at high performance. Like prior work (Section 2.6), Jumanji defends against conflict attacks by partitioning the cache. However, there is a major difference between Jumanji and prior defenses: Jumanji is not limited by associativity, so it can easily protect all applications' data while maintaining high performance.

This is a consequence of how D-NUCAs place data across all banks: In a 20-core system with highly associative, 32-way LLC banks, conventional way partitioning limits applications to 1 or 2 ways when each core is given its own partition. As a result, prior work requires the OS to designate one or a few applications as security-sensitive, and only defends their data [119, 136]. This security model is inadequate in the datacenter, where no customer wants to be the one left with poor security.

Jumanji instead places data across all banks, giving 20 banks $\times$ 32 ways/bank $=$ 640 ways to partition among applications. Jumanji can thus afford to give each application its own partition while maintaining high associativity.

3.4.4 Jumanji: Defending all LLC attacks at high performance

The message of the discussion thus far is that ***it is unwise for untrusted applications to share LLC banks***. LLC banks contain many architectural and microarchitectural components, which expose a large attack surface when shared among untrusted processes. Isolating VMs into separate cache banks protects against all bank attacks and mitigates uncontrollable performance impacts. However, though D-NUCA has natural advantages as an LLC defense mechanism, prior D-NUCAs only realize these advantages heuristically.

Jumanji's approach. Jumanji improves prior D-NUCAs to completely defend LLC attacks while maintaining high performance. Jumanji defends these attacks by preventing untrusted applications (e.g., from different VMs) from sharing banks.

```
 1 def JumanjiPlacer(bankBalance): # capacity per bank
 2   latAppAllocs = LatCritPlacer(bankBalance)
 3   batchBalance = sum(bankBalance) - sum(latAppAllocs)
 4   vmCurves = CalculateMissCurve(VMs)
 5   sizeOfVMs = JumanjiLookahead(batchBalance, vmCurves, latAppAllocs)
 6   foreach VM:
 7     sizeofVMs[VM] += latAppAllocs[VM]
 8   while VMs not all placed:
 9     AllocatePreferredBankToNextVM()
10   foreach VM:
11     allocs[VM] = latAppAllocs[VM]
12     allocs[VM] += Jigsaw(batchApps[VM])
13   return allocs
```

Figure 3.15: Jumanji's D-NUCA data-placement algorithm first reserves space for latency-critical applications to meet deadlines, then allocates entire banks among VMs to defend against cache attacks. Finally, it uses Jigsaw's data-placement algorithm to optimize batch applications within each VM.

We propose the `JumanjiPlacer`, which guarantees bank isolation between VMs, and efficiently meets tail-latency deadlines by building on `LatCritPlacer` (Figure 3.11). Jumanji achieves these benefits through a two-tiered placement algorithm which only allows shared banks between applications in the same VM, as shown in Figure 3.15.

`JumanjiPlacer` starts by calling `LatCritPlacer` to obtain the allocations for latency-critical applications. Next, it computes a combined miss-rate curve for each VM's batch applications using the model in [160, Appendix B]. Remaining LLC capacity is then divided among batch applications using a slightly modified version of the `Lookahead` algorithm [186] that guarantees each VM gets a bank-granular allocation. For example, if a latency-critical application needs 1.3 LLC banks, then `JumanjiLookahead` will allocate batch applications in the same VM either 0.7, 1.7, 2.7, . . . , or 18.7 banks so that the total LLC space allocated to the VM is a whole number.

Jumanji next places allocations in banks. Jumanji prioritizes meeting tail-latency deadlines over batch data movement by starting with the latency-critical allocations from `LatCritPlacer`. `JumanjiPlacer` assigns remaining banks in a round-robin fashion, letting each VM take the closest remaining bank (according to NoC topology).

Finally, Jumanji optimizes batch data placement within each VM. To do this, Jumanji simply calls Jigsaw's batch placement algorithm within each VM's allocation (Figure 3.15, line 12).

Putting it all together. Jumanji guarantees that latency-critical applications meet their deadlines by reserving them space in the LLC, and then partitions LLC banks across VMs to avoid new security threats that we identify. With these simple software changes, Jumanji generalizes Jigsaw to support the needs of modern datacenter applications.

3.5 Experimental Methodology

We evaluate Jumanji through detailed microarchitectural simulation using ZSim [199]. Our experimental methodology is similar to prior work [111, 222] and is detailed below.

Table 3.1: System parameters in our experimental evaluation.

Cores	20 cores, x86-64 ISA, 2.66 GHz OOO Nehalem [213]
L1	32 KB, 8-way set-associative, split data and instruction caches, 3-cycle latency
L2	128 KB private per-core, 8-way set-associative, inclusive, 6-cycle latency
Coherence	MESI, 64 B lines, no silent drops; sequential consistency
LLC	20 MB shared LLC, 5×4 1 MB banks; 32-way set-associative, 13-cycle bank latency
NoC	mesh, 128-bit flits and links, X-Y routing, 2-cycle pipelined routers, 1-cycle links
Memory	4 memory controllers at chip corners; 120-cycle latency

System. Parameters are shown in Table 3.1. We model a 20-core system with a 20 MB shared LLC, with out-of-order cores modeled on Nehalem [213]. We focus on data placement in the LLC, which is distributed into 20 banks connected by a 5×4 mesh. NoC delays are taken from prior work [79,83,149,222]. Each LLC bank uses way partitioning (i.e., Intel CAT [95]) and DRRIP replacement [101]. Main memory models bandwidth partitioning with fixed latency [96,141].

Applications. We use latency-critical applications from Tailbench [112] and batch applications from SPEC CPU2006. Each experiment runs four latency-critical applications with a random mix of sixteen SPEC applications.[1] The latency-critical applications evaluated are `masstree`, `xapian`, `img-dnn`, `silo`, and `moses`. Tailbench integrates a client and server together in one process. The client issues a stream of requests with exponentially distributed interarrival times at a given rate [153,154]. We run experiments with both *(i)* random mixes of multiple latency-critical applications and *(ii)* multiple instances of the same latency-critical application.

VM environment. Except where stated otherwise, we consider a datacenter scenario where four VMs share the resources of a single system. Each VM occupies five cores in one corner of the chip and runs one latency-critical application and four batch SPEC applications. All applications within a VM trust each other, and all applications from other VMs are untrusted.

Security metrics. We report vulnerability to port attacks by computing the average number of potential attackers per LLC access, as in Figure 3.4c. Specifically, for a single LLC access, we calculate the average number of applications from other VMs which occupy any space in the LLC bank being accessed, and then average across all LLC accesses.

Performance metrics. We measure 95th-percentile latency for Tailbench applications and weighted speedup for batch applications. (Higher latency percentiles would require prohibitively long simulations.) We compute weighted speedup using a fixed-work methodology similar to FIESTA [85]: we profile how many instructions each SPEC application completes in 15 B instructions when running in isolation and run all programs until all finish. We profile the latency-critical applications to determine request interarrival rates at low (10%) and high (50%) load, shown in Table 3.2. For all experiments, the deadline for a latency-critical application is determined by the 95th percentile tail latency when the application is run in isolation on high load with four cache ways using way partitioning. This

[1]SPEC applications are chosen from `401`, `403`, `410`, `429`, `433`, `434`, `436`, `437`, `454`, `459`, `462`, `470`, `471`, `473`, `482`, and `483`.

Table 3.2: Workload configuration for latency-critical applications. Queries per second (QPS) at different loads and total queries issued for each application.

QPS	Low	High	Number of queries
`masstree`	300	1475	3000
`xapian`	130	570	1500
`img-dnn`	28	135	350
`silo`	375	1750	3500
`moses`	34	155	300

corresponds to allocating the four latency-critical applications half of the LLC.

LLC designs. We primarily compare the four designs already introduced in Section 3.1:

1. Adaptive: an S-NUCA design that tunes the latency-critical allocation via feedback control.
2. VM-Part: an S-NUCA design that additionally partitions VMs' batch data to defend (only) conflict attacks.
3. Jigsaw: a D-NUCA that minimizes data movement, but ignores tail latency and security.
4. Jumanji: our proposed D-NUCA that meets tail-latency deadlines, defends all LLC attacks, and minimizes data movement.

We additionally consider other configurations of Jumanji as sensitivity studies, described in context below. All designs are normalized to a naïve Static allocation, where each latency-critical application is simply allocated four ways in the LLC and the sixteen batch applications share all remaining ways.

3.6 Evaluation

This section evaluates Jumanji to show the following:

1. Jumanji consistently meets tail-latency deadlines, whereas prior D-NUCAs do not.
2. Jumanji completely defends against port attacks and performance leakage, unlike most prior secure cache designs.
3. Jumanji significantly reduces data movement over prior S-NUCA designs for tail latency and security.
4. Jumanji gets similar batch speedup to Jigsaw and is close to an idealized batch placement.
5. Jumanji's performance scales well with number of VMs.

3.6.1 Security vulnerability

Jumanji fully defends LLC attacks: Port attacks and performance leakage are both a consequence of untrusted processes sharing cache banks. S-NUCA way partitioning, like Intel CAT [95], only defends against conflict attacks. Since Adaptive and VM-Part allocate space for every process in every LLC bank, they are both fully susceptible to these attacks. Figure 3.16 shows this vulnerability, plotting the average number of untrusted processes sharing a bank

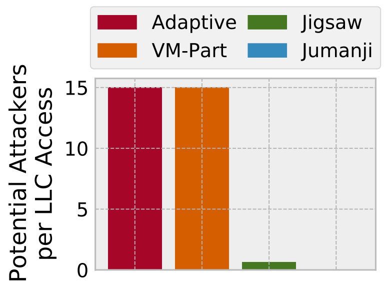


Figure 3.16: Each LLC design's vulnerability to port attacks, averaged over all experiments.

when a victim accesses it. All LLC accesses with Adaptive and VM-Part have 15 potential attackers—i.e., all untrusted applications are potential attackers.

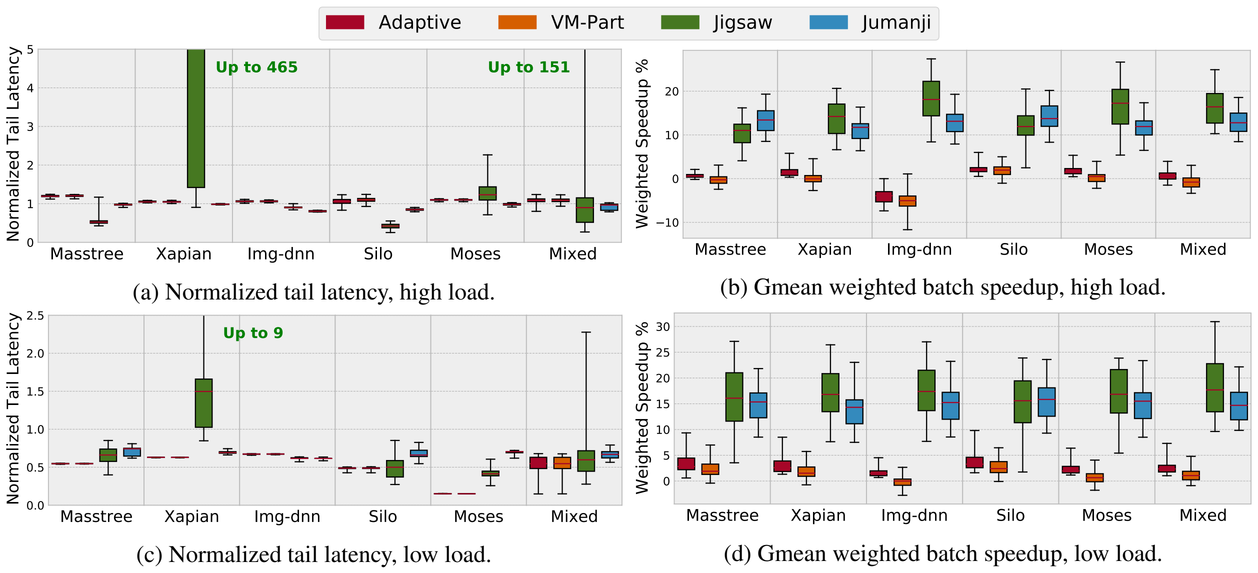
Jigsaw heuristically mitigates port attacks, and has just 0.63 potential attackers per access on average. However, heuristic mitigations are unreliable. Jumanji's placement algorithm isolates VMs into separate banks, giving a complete defense against both port attacks and performance leakage.

3.6.2 Tail latency and batch speedup

Jumanji meets deadlines with minimal data movement. Figure 3.17 shows the normalized tail latency and gmean batch weighted speedup results for each policy when running copies of one latency-critical application or a random mix of latency-critical applications. These box-and-whisker plots show the distribution of tail latency and weighted speedup (both normalized to Static) over all batch workload mixes (see caption for details). Figure 3.17 shows that all tail-latency-aware policies (Adaptive, VM-Part, and Jumanji) meet tail-latency deadlines, with rare exceptions. On the other hand, Jigsaw massively violates deadlines (by up to 465× on `xapian` and 151× on `Mixed`). Even at low load, when latency-critical applications need minimal space, Jigsaw still violates deadlines for `Xapian` and `Mixed`. Additionally, Jigsaw sometimes overprovisions latency-critical applications (e.g., `masstree` and `silo` at high load), unnecessarily harming batch applications.

Jumanji accelerates batch significantly. Figure 3.17 further shows that the D-NUCAs (Jumanji and Jigsaw) significantly accelerate batch workloads. The speedup graphs show the distribution of gmean speedup for batch applications in each workload mix compared to Static. Jumanji improves batch weighted speedup by 11%–15%, and Jigsaw improves speedup by 11%–18%. Jumanji does not quite match Jigsaw because it (correctly) reserves LLC space so that latency-critical applications meet their deadlines, whereas Jigsaw does not.

Adaptive and VM-Part barely improve batch weighted speedup, with max gmean speedups of 4% and 3%. The S-NUCAs do not perform well because, although they can give space to batch applications in periods of low load, they must take this space back when load increases. There is little net benefit except when latency-critical applications are grossly over-provisioned.



(a) Normalized tail latency, high load.

(b) Gmean weighted batch speedup, high load.

(c) Normalized tail latency, low load.

(d) Gmean weighted batch speedup, low load.

Figure 3.17: Normalized tail latency (lower is better) and gmean weighted batch speedup (higher is better) relative to a naïve static allocation over 40 random batch mixes, at high and low latency-critical load. Results are shown as box-and-whisker plots. Boxes show the lower to upper quartile of values (over 40 workload mixes); whiskers show the furthest data points. This figure summarizes 969 trillion simulated cycles.

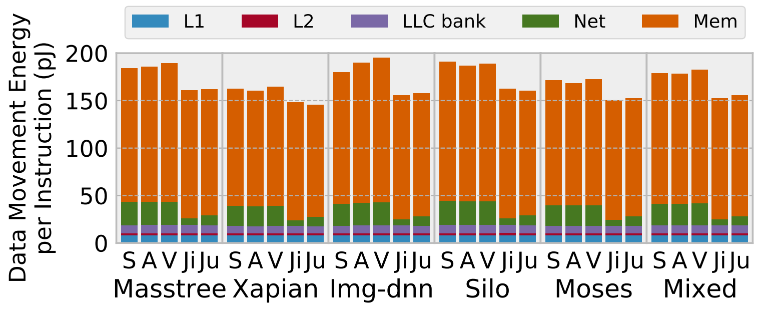
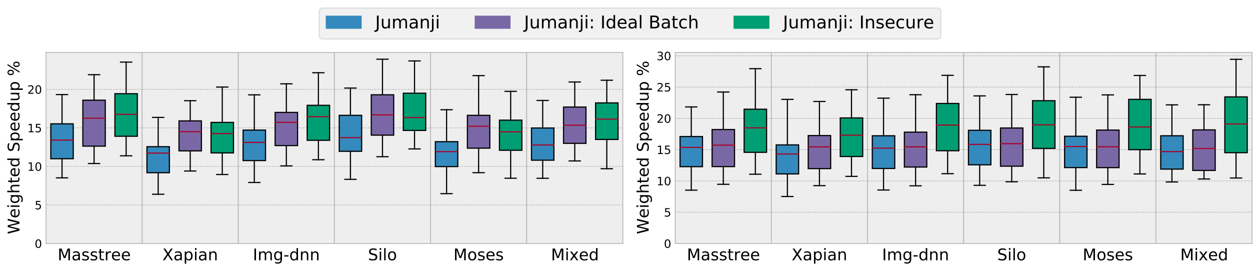


Figure 3.18: Dynamic data movement energy for latency-critical applications at high load over 40 random batch mixes. S=Static, A=Adaptive, V=VM-Part, Ji=Jigsaw, Ju=Jumanji. Jumanji matches Jigsaw's data movement reductions without violating tail-latency deadlines.

Jumanji supports multiple different latency-critical applications. Prior 20-core results evaluated multiple instances of the same latency-critical application. Figure 3.17 also shows that Jumanji meets deadlines when running mixes of *different* applications, whereas Jigsaw violates deadlines for one-third of applications at high load. Jumanji also manages to achieve 14% gmean batch speedup over all workload mixes, comparable to Jigsaw's 17% gmean speedup, whereas Adaptive and VM-Part do not even obtain 3% speedup.

Jumanji significantly reduces data movement. Figure 3.18 shows average dynamic data movement energy for each workload at high load in Figure 3.17. Data movement energy is split between the L1, L2, LLC banks, on-chip network, and memory, using numbers from prior work [222].

Overall, D-NUCA designs achieve significantly lower data movement than S-NUCA designs. This is due to fewer memory accesses from LLC partitioning and fewer network hops from data placement. Compared to Static, both



(a) Gmean weighted batch speedup, high load. (b) Gmean weighted batch speedup, low load.

Figure 3.19: Batch speedup for Jumanji vs. *(i)* "Jumanji: Insecure", which does not enforce bank isolation, and *(ii)* "Jumanji: Ideal Batch", which eliminates competition with latency-critical applications during placement. On average, Jumanji is within 3% of Insecure and within 2% of Ideal Batch.

Jumanji and Jigsaw reduce average data movement energy by 13% whereas Adaptive actually *increases* it by 0.1%, and VM-Part also *increases* it by 2.4% (both due to extra LLC misses from limited associativity in way partitioning).

3.6.3 Sensitivity studies

Jumanji defends cache attacks at low cost. Figure 3.19 shows that Jumanji's bank-isolation defense against LLC attacks costs little batch performance. We compare Jumanji to "Jumanji: Insecure", a version of Jumanji that does not enforce strict bank isolation but is otherwise identical. Jumanji gets 11%–15% gmean batch speedup, vs. 14%–19% for Insecure, and is within 3% of Insecure on average.

Jumanji's simple algorithms are nearly ideal for batch speedup. As described in Section 3.3, Jumanji prioritizes latency-critical applications, giving them as much space as they need and placing their allocations first. One might wonder: how much does this simple, greedy approach penalize batch applications?

Figure 3.19 shows that Jumanji is in fact nearly ideal for batch speedup. "Jumanji: Ideal Batch" is an infeasible, idealized design that eliminates competition between latency-critical and batch applications, letting batch applications get their preferred data placement. It does this by placing batch and latency-critical data in separate copies of the LLC, while ensuring the total capacity allocated to applications does not exceed the original LLC size. (E.g., if latency-critical applications claim 8 MB, then it allocates the remaining 12 MB among batch applications but places these allocations in a copy of the 20 MB LLC reserved for batch applications.) Latency-critical data is still placed in nearby banks in their own LLC, maximizing the space available to batch applications (e.g., Figure 3.8). Ideal Batch also isolates VMs for security. The result is an infeasibly optimal batch placement unconstrained by any latency-critical placement.

Figure 3.19 shows that Jumanji's simple algorithms are within 2% of Ideal Batch (by gmean batch speedup). Jumanji's greedy placement is effective because moving allocations further away from latency-critical applications would require giving them larger allocations to meet deadlines. This is rarely a net win for batch applications. In fact, we implemented an algorithm that tries to improve batch placement by trading allocations with latency-critical

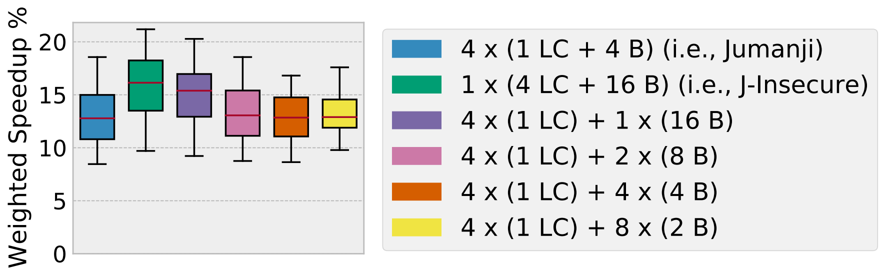


Figure 3.20: Jumanji's batch speedup when varying from 1 VM (all apps) to 12 VMs. Jumanji scales well as VMs increase.

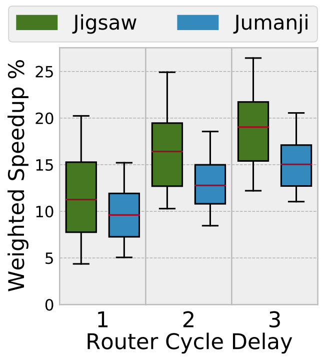


Figure 3.21: NoC sensitivity.

applications, similar to [30]. In contrast to [30], we found that trades were very rare and yielded little speedup: [30] only tries to reduce data movement, whereas Jumanji must also meet latency-critical deadlines. This latter requirement imposes a strict constraint on trades (i.e., they cannot penalize latency-critical applications), which greatly reduces the number of beneficial trades. As a result, the algorithm generally behaves like Jumanji's simple `LatCritPlacer` in practice.

Jumanji scales well as the number of VMs increases. We next consider how Jumanji scales with different VM configurations, shown in Figure 3.20. Results thus far have used four VMs, each with one latency-critical application and four batch applications, denoted "$4 \times (1\ \text{LC} + 4\ \text{B})$" in the figure. Figure 3.20 explores six different configurations, ranging from a single VM (i.e., no bank isolation) up to twelve VMs (one per latency-critical application and per pair of batch applications). Increasing VMs further causes missed deadlines, since VMs become restricted to a single LLC bank.

Figure 3.20 shows that Jumanji scales well with more VMs. Jumanji's gmean speedup varies from 16% with one VM to 13% with twelve VMs. Increasing VMs from four (the default used in other experiments) to twelve shows no degradation in batch speedup. Jumanji is effective with many VMs because placing data in nearby banks is sufficient for most applications, and Jumanji retains enough flexibility to increase allocations for the few applications that benefit a lot. While Figure 3.20 only shows results for mixed latency-critical applications at high load, results are similar for other configurations too.

NoC sensitivity. Finally, we see how speedups vary with NoC latency. Results so far use 2-cycle router delay to model modest NoC congestion. Figure 3.21 shows that Jumanji's speedup on random mixes increases from 9% to 15% as routers go from 1 to 3 cycles.

Summary. Jumanji shows that systems can meet tail-latency deadlines and defend a wide range of attacks while *improving* performance. D-NUCAs can benefit all applications by placing data in nearby LLC banks, where it belongs. By considering all applications' goals, Jumanji excels where previous solutions fail.

3.7 Summary

We have shown that D-NUCAs offer significant advantages in tail latency and security over prior LLC designs. However, to realize these benefits, D-NUCAs cannot focus exclusively on reducing data movement. We developed Jumanji, the first data placement algorithm for tail latency and security, and demonstrated that it meets tail-latency deadlines and defends against previously undefended cache attacks, yet still significantly reduces data movement compared to NUCA-oblivious designs. Jumanji thus achieves the best of all worlds. Moreover, Jumanji requires only simple hardware and software, making it a practical approach to scale future systems.

Chapter 4

täkō: A Polymorphic Cache Hierarchy for General-Purpose Optimization of Data Movement

Hardware and software do not work together to optimize data movement, despite its rapidly increasing importance. Mainstream instruction set architectures were designed at a time when data movement was inexpensive and do not emphasize it — software reads and writes data, and hardware decides when and where to move it. Although memory hierarchies have evolved over time, this memory interface has remained unchanged.

Lacking visibility and control over data movement, software cannot implement many attractive features or optimizations, and instead resorts to overly conservative and wasteful solutions. Recognizing this, there has been a wave of proposals for specialized memory hierarchies that give software the ability to observe and manipulate data as it moves [2, 8, 10–12, 40, 55, 62, 66, 77, 87, 91, 113, 127, 145, 157, 159, 162, 178, 179, 183, 200, 214, 218, 225, 226, 247, 256, 257, 259–262]. These designs are highly effective, often reporting speedups of 2$\times$ or more, so there is clearly potential to massively reduce data movement. However, as discussed in Section 2.7.5, adding per-application custom logic to a general-purpose CPU memory hierarchy is not a scalable solution.

We argue that the solution to this problem is to find a *single, general-purpose architecture* that supports a wide variety of data-movement features and optimizations. Only with wide applicability can the necessary hardware and software investment be justified. Additionally, we observe that the key to many prior optimizations is the ability to perform simple computations in response to data movement. Hence, the theme of this work is that *architectures should expose more data movement to software, so that software can observe and optimize data movement itself.* In other words, the hardware-software interface is the problem, and often *specialized hardware is not needed* with a richer interface. The missing ingredient is feedback from hardware to software when data moves. We call this idea a *polymorphic cache hierarchy*, and we propose the *täkō* architecture to realize it.

Software control of data movement offers enormous advantages over a hardware-only approach. Solutions can be

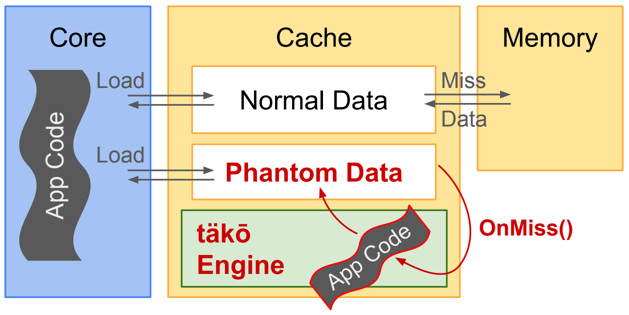


Figure 4.1: täkō in action: An application registers an address range whose semantics are defined by software callbacks. These callbacks run in-cache on programmable engines.

better tailored to individual applications, and development cycles go from years to days. Although the upfront costs of a new hardware-software interface are formidable, *these costs are paid only once*, after which the marginal cost is reduced by orders of magnitude.

Figure 4.1 illustrates täkō in action. Software (e.g., an application, domain-specific framework, or library) registers a *phantom address range* with täkō, whose data only lives in-cache and is not backed by off-chip memory [40]. Instead of fetching data from memory, misses to this address range are served by *software callbacks*. Evictions and writebacks are handled similarly. These callbacks thus define the semantics of loads and stores in this address range, letting software re-purpose the caches as desired.

Like recent near-data computing architectures [8, 142, 177, 236, 257], täkō adds programmable *engines* near caches to execute callbacks efficiently. In täkō, engines contain scheduling logic and a spatial dataflow fabric to run callbacks [68, 93, 175, 221, 228, 238]. With this microarchitectural support, täkō gets close to the performance of fully specialized hardware — software programmability adds little overhead because data movement costs dominate and callbacks are short. The critical difference from prior work is that whereas *cores* invoke tasks in prior near-data architectures, *caches* invoke callbacks in täkō. This difference is the crux of the architecture: täkō closes the loop between hardware and software, letting software finally observe and optimize data movement.

This work explores the programming interface and system architecture of a polymorphic cache hierarchy. täkō's goal is to enable optimizations that otherwise require custom hardware, and as such it currently provides a low-level interface for expert programmers. This work focuses on *(i)* an initial set of callbacks that covers many, but not all, data-movement features and optimizations; and *(ii)* an architecture that implements these callbacks correctly and efficiently.

Contributions. This work contributes the following:

- *Problem.* We identify the need for an improved hardware-software interface to unlock the performance and efficiency gains demonstrated by recent specialized cache hierarchies.
- *Programming Interface.* We propose a simple, flexible, and effective programming interface to give software

visibility and control over data movement.

- *Architecture.* We discuss the architectural constraints and features needed to implement a polymorphic cache hierarchy correctly and with good performance, with similar hardware overhead to prior near-data computing architectures.

Summary of results. We present five case studies for täkō, demonstrating that a general-purpose, programmable data-movement architecture can enable new functionality while approaching the performance of custom hardware.

- *In-cache data transformation:* täkō enables software-defined transformations (e.g., decompression) when data moves. With good locality, täkō eliminates redundant work to get 2.2× speedup and 61% energy savings.
- *Commutative scatter-updates:* täkō implements PHI [162], transforming the caches to use push-based semantics to accelerate commutative scatter-updates in graphs. täkō gets 3.5× speedup, similar to [162].
- *Decoupled graph traversals:* täkō implements HATS [159] as a representative decoupled streaming application. täkō accelerates graph traversals and gets a 43% speedup and 17% energy savings.
- *Transactions on non-volatile memory:* täkō's improved visibility over data movement eliminates wasteful work in NVM transactions. If no data is evicted before commit [158], täkō eliminates journaling overhead and achieves up to 2.1× speedup and 47% energy savings.
- *Detecting cache side-channel attacks:* täkō exposes data movement to software, letting applications detect and prevent cache side-channel attacks [138].

Unlike prior work that requires custom hardware for each feature and optimization, täkō implements these applications on a single, general-purpose hardware design. täkō adds just ≈5% area overhead, similar to prior near-data systems. Further, we show that täkō's hardware achieves performance within 1.8% of an idealized design.

4.1 täkō Overview

täkō consists of software and hardware components. In software, täkō's programming interface gives software visibility and control over data movement via cache-triggered callbacks. In hardware, täkō adds architectural support for scheduling and executing callbacks efficiently near data.

Design rationale. Caches exist to shield systems from expensive operations. Traditionally, these are reads and writes to larger memories lower in the cache hierarchy, but in principle they could be anything. täkō opens up the cache hierarchy by letting *software* define what happens on a cache miss and, similarly, what to do with evictions.

Opening up the cache hierarchy yields two distinct benefits:

(a) Software can leverage existing cache hardware to memoize expensive computations or buffer updates; and

(b) Software can observe data movement as it happens and interpose as necessary.

Table 4.1: täkō callback semantics.

Callback	**Semantics**	Side effects?*
`onMiss`	Generates data for requested address.	✗
`onEviction`	Handles eviction of unmodified data.	✗
`onWriteback`	Handles eviction of modified data.	✓

* ✗ — can only write local state and/or the affected cache line; see Section 4.3.3.

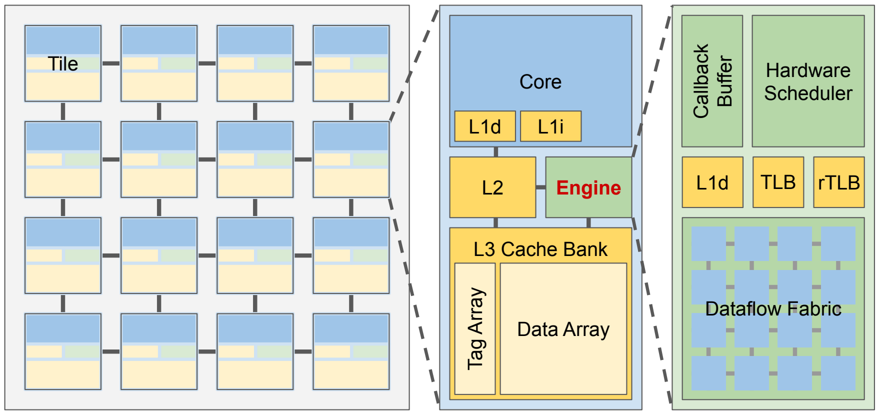


Figure 4.2: täkō adds programmable *engines* to each tile of a CMP. Engines schedule callbacks in response to cache events and execute them in parallel with conventional threads.

Both of these benefits are essential to implementing many data movement features and optimizations. For example, PHI [162] *(a)* buffers graph updates in-cache, and *(b)* decides on eviction whether to apply updates in-place or log them (Section 4.7.1).

Interface. Table 4.1 summarizes täkō's interface. Callbacks are registered only on selected addresses, and täkō does not affect loads and stores to other addresses. `onMiss` is invoked on cache misses, letting software fill in the requested cache line. Values are then cached normally; i.e., cores can read and write them, with hits handled like any other data. `onEviction` and `onWriteback` handle evictions for clean and dirty data, respectfully.

Architecture. Figure 4.2 shows a high-level view of a täkō system. On top of a baseline, cache-coherent multicore, each tile is augmented with an *engine* that contains hardware scheduling logic and a programmable dataflow fabric to execute callbacks. täkō tracks which lines have callbacks registered and adds no latency or energy to traditional loads and stores.

The engine microarchitecture is guided by constraints and characteristics of täkō callbacks. To compete with specialized hardware, callbacks must exploit memory-level parallelism but should not add much area. Callbacks tend to be short, re-execute repeatedly, and perform the same operation across entire cache lines. These considerations led us to a dataflow fabric (to avoid re-fetching the same instructions) with SIMD functional units (for repeated operations).

Summary. täkō hardware enables visibility and control over data movement *in software* via its general-purpose programming interface. The architecture changes *only once, up front*, rather than for each individual data-movement

```
int64 bases[N / 8]    # one base per line, or 8 values
int8 deltas[N]        # 4-bit exponent, 4-bit mantissa

total = 0

for idx in indices:
    # 1. decompress data
    base = bases[idx >> 3]
    delta = deltas[idx]
    mantissa = delta & 0b1111
    exponent = delta >> 4

    data = base + (mantissa << exponent)

    # 2. compute average
    total += data

avg = total / len(indices)
```

Figure 4.3: Example program written in traditional software.

feature or optimization. täkō thus massively reduces the barrier for optimizing data movement.

4.2 Motivation

Memory hierarchies currently suffer from *innovation deadlock*: though specialization offers large benefits, it also requires prohibitively large, up-front investments in both hardware and software. Without strong demand from software, hardware vendors are reluctant to design, verify, and support new features; but without hardware support, software vendors will not rewrite applications. As a result, architects are limited to optimizations that preserve the load-store interface but leave significant gains on the table. The goal of täkō is to break this deadlock by providing a general-purpose architecture that frees software to optimize data movement itself.

To motivate täkō, we begin with an example of how polymorphic cache hierarchies enable data-movement optimization in software. The purpose of this example is to introduce the basic components of a polymorphic cache hierarchy. Later case studies will show the full power of polymorphic cache hierarchies to transform cache behavior.

4.2.1 Example program: Lossy compression

Prior work has studied many optimizations that transform data as it moves through the cache, e.g., to compress [12, 62, 157, 178, 179, 200, 226, 247], decrypt [73, 108, 195], prefetch [8, 218, 256], change layout [10, 40], memoize [11, 66, 261, 262], or serialize/de-serialize [183] data. We motivate täkō by observing how its `onMiss` callback enables arbitrary data transformations while improving performance, saving energy, and reducing overall work.

Figure 4.3 shows our example program, which computes the average value of a data set that is stored in an approximate, compressed format in memory as a base plus offset value, similar to [179]. Unlike standard compressed

```
# define new cache semantics to pack data densely
class Decompressor extends tako::Morph:
    int64* data  # actually in base Morph class
    int64* bases
    int8* deltas

    # decompress data when it moves into cache
    void onMiss(int64* phantomAddress):
        idx = phantomAddress - &this.data[0]

        base = bases[idx >> 3]
        delta = deltas[idx]
        mantissa = delta & 0b1111
        exponent = delta >> 4

        *phantomAddress = base + (mantissa << exponent)
```

```
int64 bases[N / 8]   # one base per line, or 8 values
int8 deltas[N]       # 4-bit exponent, 4-bit mantissa

# allocate new phantom address range with callbacks
morph = tako::registerPhantom(Decompressor, PRIVATE, sizeof(int64) * N)
morph.bases = bases
morph.deltas = deltas

total = 0

# code now reads decompressed data directly
for idx in indices:
    total += morph.data[idx]

avg = total / len(indices)

morph.unregister()
```

Figure 4.4: Pseudocode for the same program in täkō.

caches, this lossy compression cannot be implemented in hardware without application knowledge [156], motivating the need for software in the loop. (The details of the compression algorithm are immaterial; the point is that software can transform data however it likes.)

This program has two major problems. Cores are inefficient at data transformations, wasting time and energy [183, 247]. And if data are re-used, then the program re-executes the same transformation many times. However, there is currently no good alternative in software, as alternative implementations waste memory, add data movement, or perform even more work.

4.2.2 täkō to the rescue!

Figure 4.4 illustrates how täkō solves these problems. Rather than operate on the raw compressed data, the program allocates a new "phantom" address range for decompressed data. These addresses only live in the caches and are not backed by physical memory. The program defines an `onMiss` callback that decompresses data whenever a new cache

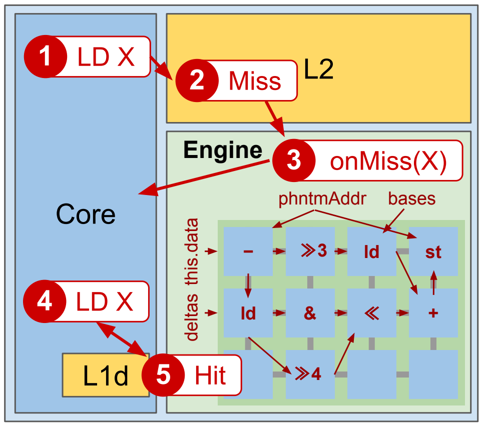


Figure 4.5: Running the example program in täkō. Data is decompressed automatically on a cache miss, executing application `onMiss` code on a near-cache spatial dataflow fabric. Decompressed data is also cached for future use to eliminate redundant work.

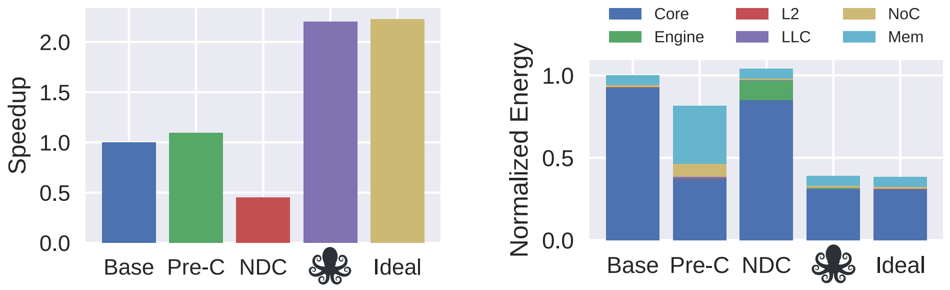


Figure 4.6: Results for the example program. täkō improves performance by 2.2× and reduces energy by 61%.

line in the phantom range is requested.

The callbacks are grouped in a `Morph` object that collects the data and methods for this polymorphic cache hierarchy — in this example, a `data` pointer to the phantom address range and pointers to the `bases` and `deltas` arrays. The `onMiss` callback takes the phantom address that triggered the miss and decompresses the requested data. All operations execute in parallel across the full cache line, shown in data-parallel pseudocode for brevity.

The modified program first registers the `Morph` at the private L2 cache, allocating a phantom address range for it. It then simply reads the decompressed data and computes the average, now using even simpler code. Figure 4.5 illustrates its execution. The first time the program reads a phantom address `X`, there is an L2 miss, which triggers `onMiss` on the spatial dataflow engine to decompress the full cache line. The decompressed line is then cached so that any subsequent read of the same line (due to spatial or temporal locality) is a cache hit, *eliminating redundant work* from decompressing the same data many times.

4.2.3 Results and comparison to prior work

The täkō version of this program improves performance, saves energy, and reduces redundant work. Figure 4.6 shows results with 32 K `indices` for the baseline software implementation, a software version that pre-computes the

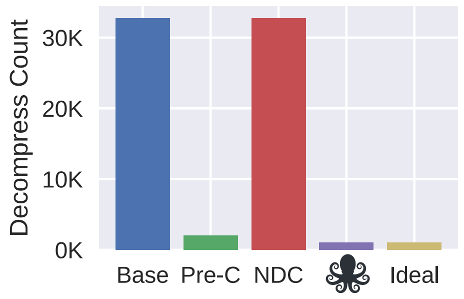


Figure 4.7: Number of decompressions.

decompressed data in a separate array, a near-data computing (NDC) implementation, and the täkō (🐙) implementation. The pre-compute version uses vector instructions to decompress a full cache line (eight values) at a time. The NDC version is similar to [142], where the core offloads decompressions to execute at an L2 engine. Indices are randomly generated following a Zipfian distribution [38] over 16 K values. (Full experimental methodology is in Section 4.6.)

täkō reduces execution time by 55% vs. the software baseline and by 50% vs. software pre-computation, and it reduces energy by 61% and 52%, respectively. Moreover, täkō comes within 1.1% performance and 1.3% energy of an idealized engine with unlimited, instantaneous, and energy-free compute.

täkō achieves these gains by memoizing decompressions of frequently accessed data (Figure 4.7), greatly reducing the number of total decompressions. Although the pre-compute version avoids decompressing the same value multiple times, it decompresses values which are never accessed and also allocates memory for the entire decompressed array, incurring significant memory overheads. With täkō, decompression runs on in-cache engines, in parallel with software threads, similar to prior near-data computing (NDC) architectures. However, unlike NDC, *täkō triggers computation by data movement, not from cores:* instead of decompressing data every time it is requested, täkō decompresses data only on a miss and caches it thereafter, exploiting locality to eliminate redundant work [261, 262].

This optimization is not possible in prior NDC systems, which move computation closer to data but do not improve software's visibility over data movement. Figure 4.6 shows that NDC actually *hurts* performance and energy efficiency on this decompression example. This is because decompressing at the L2 fails to exploit locality in the L1s; in other words, offloading computation near-data is not always an optimization [142]. In contrast, täkō's *cache-triggered* computation gets the best of all worlds by executing computations near-data, eliminating wasteful work, and preserving locality.

4.2.4 Discussion

Decompression is representative of many prior optimizations that transform data as it moves through the cache hierarchy. Such transformations are easily implemented by writing `onMiss`, `onEviction`, and `onWriteback` callbacks. These callbacks are written *in software* and execute on täkō's general-purpose hardware. Compared to adding custom hardware, täkō reduces the innovation barrier by orders of magnitude.

It bears emphasizing that a polymorphic cache hierarchy is *not* purely microarchitectural. This is by design: the entire point is to give software visibility and control over data movement. Callbacks should be thought of as part of the application code, which execute as hardware-scheduled threads in parallel with conventional software threads. A well-structured application splits functionality appropriately between the two.

Finally, while this example showed how täkō can leverage caches to eliminate redundant work, täkō is capable of more radical transformations of cache behavior. These will be explored in Section 4.7.

4.3 täkō Programming Interface

täkō's programming interface is designed to let software optimize data movement in ways that would otherwise require custom hardware. Our goal is to massively reduce implementation effort vs. the custom hardware required by prior specialized cache hierarchies. This section describes the interface and restrictions that make it easier to reason about program behavior. Though täkō is available to application programmers, it currently targets experts; we envision täkō code being shipped as part of domain-specific frameworks or libraries.

Overview. täkō breaks the address space into different *address ranges*, each with their own semantics. Software can register *callbacks* that execute in response to specific cache events — misses, evictions, and writebacks. By default, addresses retain load-store semantics and have no callbacks registered.

Software defines the behavior of a polymorphic cache hierarchy by providing a `Morph` data type and registering it with a specific address range. Often, the `Morph` allocates a new *"phantom"* address range that is not backed by physical off-chip memory [40], but `Morphs` can also be registered on "real" addresses. Phantom callbacks define the results of loads and stores to the address range, since there is no backing memory to load or store. Figure 4.8 gives pseudocode for täkō's basic interface, discussed in detail below.

4.3.1 `register/unregister`

Registering the `Morph` associates callbacks with an address range. Software provides a morph type (a child class of `täkō::Morph`), the location in the cache hierarchy to register the `Morph`, and the address range. The location can be `PRIVATE` (at the L2) or `SHARED` (at the L3). Currently, täkō does not support `Morphs` at the L1 because L1s are very tightly integrated with cores; nor does it support `Morphs` at memory because memory controllers are below the cache coherence protocol, complicating consistency in callbacks.

Phantom address ranges are requested only by their size, and `registerPhantom` allocates and assigns the address range. To support `Morphs` on existing data, `registerReal` accepts an arbitrary base and bound and attempts to register

```
Morph registerPhantom(morphType, location, size)
Morph registerReal(morphType, location, base, bound)
void flushData(morph)
void unregister(morph)

class Morph:
    void* data              # base of address range
    int size                # size of address range
    Morph[] views           # engine-local state

    # callbacks
    void onMiss(addr)       # loads
    void onEviction(addr)   # clean evictions
    void onWriteback(addr)  # dirty evictions
```

Figure 4.8: täkō's interface for a polymorphic cache hierarchy.

the Morph on this range. täkō only allows *one* Morph *to be registered on an address at a time*. This restriction simplifies translation hardware (see below), but it is not fundamental.

The Morph remains in effect until unregistered. When a Morph is registered or unregistered, its address range is flushed from the cache. unregister de-allocates phantom address ranges.

4.3.2 Morph objects

A Morph object represents an instance of a particular polymorphic cache hierarchy. Multiple instances of a Morph type, or of different types, can be registered at the same time, each operating on their own distinct address ranges (e.g., see Section 4.7.3). register returns a Morph object, letting software threads control it (e.g., by unregistering it).

Callbacks execute on engines, not cores, and each engine also has its own *view* (i.e., copy) of the Morph object. This is important because each view may have local state, similar to conventional thread-local state, but shared by all threads running on that engine. Local state is allocated in memory, and engines access it via coherent loads and stores. PRIVATE Morphs have a single view (at the L2), but SHARED Morphs have one view per L3 bank. The views are gathered in the views array to, e.g., allow initialization of local state.

4.3.3 Callbacks

Cache-triggered callbacks are the heart of täkō's design. By defining callbacks in the Morph, software transforms the semantics of that address range. Callbacks are flexible to maximize täkō's applicability, but they must obey certain restrictions for correctness and performance.

Semantics. täkō callbacks allow software to modify cache behavior, as summarized in Table 4.1. For phantom address ranges, onMiss and onWriteback directly define the results of loads and stores. When there is a miss to a phantom address, the cache controller allocates a line, zeroes it, and then invokes onMiss. When evicting a phantom cache line,

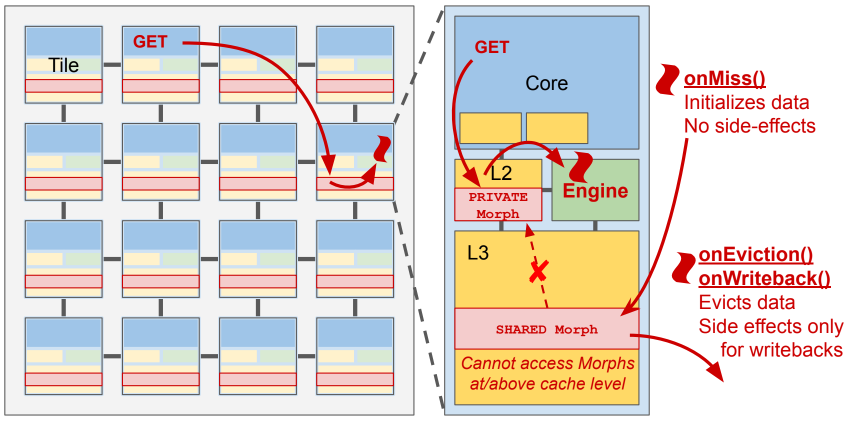


Figure 4.9: Callbacks are scheduled by hardware in response to cache misses, evictions, and writebacks, and run on the engine closest to the data. Only writebacks should have side effects.

the cache controller invokes `onEviction` (if clean) or `onWriteback` (if dirty) and then discards the line. Intervening memory operations (i.e., cache hits) simply read and write the data normally, without invoking callbacks.

Callbacks on real address ranges operate similarly, except that the cache controller reads and writes the backing memory, maintaining load-store semantics by default. `onMiss` begins executing in parallel with reading `addr`. `onWriteback` executes before writing back `addr` to let the callback interpose.

`onMiss` is on the critical path of software threads, but `onEviction` and `onWriteback` are not. This difference is important for performance: it is best to keep `onMiss` short, and push work into the other callbacks (e.g., see Section 4.7.1).

Execution model. Callbacks are short threads that are created and scheduled *entirely by hardware* and run in parallel with conventional software threads (Figure 4.9). Because callbacks are triggered by cache hardware, *they can occur spontaneously* from the perspective of a software thread. This spontaneity can be unintuitive: cache misses can be triggered by speculative loads or prefetches, so an `onMiss` may not correspond to any committed instruction in a program. Similarly, data can be evicted from caches at any time, triggering `onWriteback` even when no corresponding software thread is active.

Restrictions. Given these considerations, it is best practice to write callbacks that behave similarly to conventional reads, evictions, and writebacks. That is, `onMiss` *and* `onEviction` *should be free of side effects*,[1] since they can be triggered at any time, whereas `onWriteback` *can have side effects*, since modified data must correspond to a committed store in some software thread. These restrictions make it easier to reason about callback behavior, but täkō does not strictly enforce them because misses/evictions are sometimes part of correctness (e.g., for security; see Section 4.7.4).

Ignoring side effects, callbacks can reference nearly any memory address. The remaining exception is that *callbacks cannot access data with a* `Morph` *registered at the same or higher level of the cache hierarchy* (Figure 4.9). Without

[1]We define a side effect as a modification to non-local state, i.e., a store to any location except the engine's local `Morph` object or the `addr` itself.

this restriction, deadlock is possible as callbacks trigger further callbacks, quickly exhausting the engine's hardware scheduler. A `SHARED` callback is not allowed to trigger a `PRIVATE` callback because the `PRIVATE` callback could trigger `onMiss` in the shared cache. But a `PRIVATE` callback *can* trigger a `SHARED` callback, since there is no cyclic dependence. This constraint was not problematic in any of our case studies.

Callback code. täkō is designed for short callbacks, which we find to be natural in our case studies. Callback code executes in SIMD fashion across entire cache lines. For long code paths or error conditions, callbacks can raise a user-space interrupt to preempt a software thread (e.g., see Section 4.7.4). For simulation convenience, callback code is currently written in C++, and instructions are mapped onto the dataflow fabric when they first execute; in practice, one could compile code statically [217, 238].

Coherence and consistency. täkō leverages the cache-coherence protocol in the baseline multicore to provide a consistent view of memory. A callback is just another thread in the system, from a consistency perspective. Engines have coherent L1d caches, implemented using clustered coherence within each tile to avoid increasing directory state [75, 130, 151]. In brief, the L2 and engine L1d snoop on coherence traffic within each tile so that the directory behaves exactly as if the engine L1d is part of the L2 cache on that tile.

Callbacks thus enjoy the same coherence and consistency as any other thread in the system. Additionally, *the address that triggered the callback is locked for the duration of callback execution*; i.e., no other thread (or callback) can access the data until the callback completes. Locking is strictly enforced by the cache controller, which serializes operations on each address. Callbacks therefore do not need to worry about racing accesses to `addr`, but races to other addresses are possible, so callbacks should be data-race free [1] to maintain consistency.

4.3.4 `flushData`

`flushData` enables synchronization between callbacks and conventional threads without completely unregistering a `Morph`. By flushing all of a `Morph`'s data from the cache, programs are guaranteed that there will be *no further racing writes from callbacks*. `flushData` signals cache controllers at the appropriate level of the hierarchy to walk their tag arrays and flush any lines belonging to the `Morph`'s address range, triggering `onWriteback` or `onEviction`. `flushData` blocks the software thread until all callbacks complete.

4.3.5 Discussion and roads not taken

We found the above callbacks to be a logical starting point for a polymorphic cache hierarchy that covers a wide range of use cases. As discussed in Section 4.1, the basic intuition is to generalize caches by letting software provide an `onMiss` handler [89], and the rest of the interface and its restrictions follow naturally. We arrived at this interface early in the design, and it proved useful, self-contained, and consistent. Although the semantics are not trivial, writing täkō software has been fairly straightforward in our experience. For most applications, there is a clear separation of concerns across misses and clean or dirty evictions (Section 4.7).

That said, more callbacks are certainly possible. `onReplacement` would allow software to optimize the eviction policy for particular workloads [19, 244]. `onHit` would allow customization of the cache coherence protocol, among other applications. We did not pursue `onHit` because programmable cache coherence has been explored extensively [4, 40, 55, 125, 127, 192, 209, 259, 260] and because it seemed that `onHit` would often be needed in the L1, requiring disruptive core changes. Finally, one could make cache indexing programmable, letting software re-purpose the tag array [188, 189, 197, 207, 208, 262]. We did not explore this direction to avoid adding any latency to conventional loads and stores — täkō has *no* performance impact on legacy applications.

4.4 täkō Architecture

Similar to recent near-data-computing architectures [8, 142, 177, 236, 257], täkō extends a baseline multicore with near-cache engines to run callbacks efficiently (Figure 4.2). Engines are placed on each tile of the multicore, near the L2 and L3 caches. The engines consist of *(i)* a hardware scheduler that buffers callbacks and runs them when they are ready, and *(ii)* a spatial dataflow fabric that executes callbacks efficiently. Figure 4.10 shows `onMiss` and `onWriteback` callbacks, which are referenced throughout the text below.

4.4.1 Core modifications for täkō

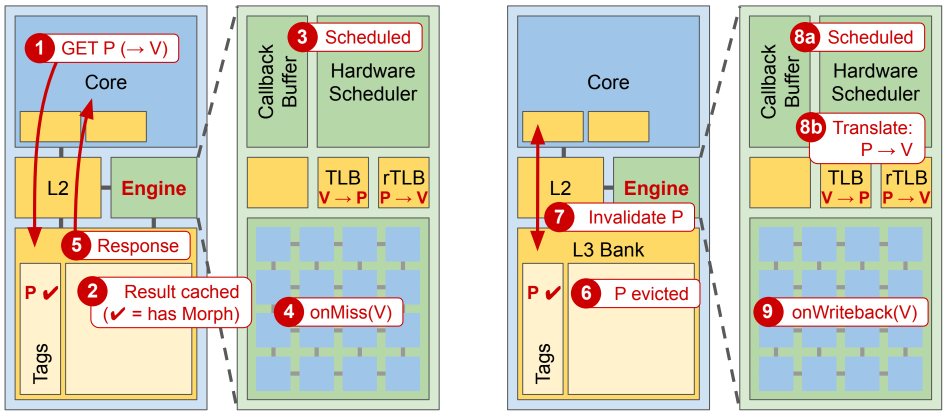
Tracking `Morphs`. täkō tracks which addresses have a registered `Morph` via the TLB. TLBs are augmented with two bits indicating whether a `Morph` is registered and, if so, whether it is registered at `PRIVATE` or `SHARED`. When a load or store misses, the core augments the `GET` request with these bits, giving the `Morph`'s location ❶. Alternatively, täkō could keep a separate table of registered `Morphs`, but this would limit the number of `Morphs` that could be registered concurrently.

ISA. täkō adds one new cache flush instruction, corresponding to the `flushData` API, that flushes a particular address range in the `PRIVATE` or `SHARED` cache.

4.4.2 Cache modifications for täkō

State. Tags are extended with one bit to track whether a `Morph` is registered for the line at that cache level ❷. This bit is set on insertion using the two registration bits in the `GET` request.

Triggering a callback. Engines are tightly integrated with the cache controller. When serving a cache miss, eviction, or writeback, the controller checks whether a `Morph` is registered and, if so, sends a request to the local engine along



(a) `onMiss`. (b) `onWriteback`.

Figure 4.10: Example callback execution in täkō on a phantom address. Engines schedule and execute callbacks in hardware. TLBs and cache tags track where `Morph`s are registered. The accompanying text walks through the steps of callback execution.

with the `addr` and operation type. The engine's scheduler enqueues a request in its callback buffer 3 8a, which starts executing it as soon as the fabric is available and the callback configuration is loaded 4 9. (Usually, the fabric is ready immediately.) For `onEviction` and `onWriteback`, the registered line occupies an entry in the cache's writeback buffer until a callback buffer entry is available. When the callback completes, the cache controller responds to the original request 5. Other cache operations (i.e., all hits and any operation with no `Morph` registered) work normally and do not go through the engines at all.

Avoiding deadlock. Without additional mechanisms, deadlock can occur in the engine scheduler: e.g., suppose the engine's callback buffer is full, an executing callback suffers a cache miss, and every line in the set is waiting to grab a callback buffer spot (e.g., to execute `onMiss`). Nothing can be evicted because the callback buffer is full, so the callback buffer cannot drain.

Luckily, it is easy to avoid this deadlock by ensuring that *there is always a cache line in every set with no* `Morph` *registered at this cache or any child cache*. This constraint guarantees forward progress, as there will always be a line that can be evicted without triggering a callback. täkō enforces this constraint by modifying its eviction policy, tr̃rîp (see below). For similar reasons, täkō enforces that there is always at least one MSHR and writeback buffer entry not waiting on a callback.

Avoiding cache pollution from callbacks. Callbacks often translate a phantom address to some real address that is accessed during the callback, but is not accessed afterwards (e.g., `deltas[idx]` in Figure 4.4). To avoid cache pollution, täkō modifies its RRIP-based [101] replacement policy, tr̃rîp, to insert accesses from engines at lower priority (i.e., closer to eviction). This optimization can significantly improve cache utilization; e.g., in a simple `Morph` that maps array-of-structs to struct-of-arrays, we have observed speedup of $>4\times$.

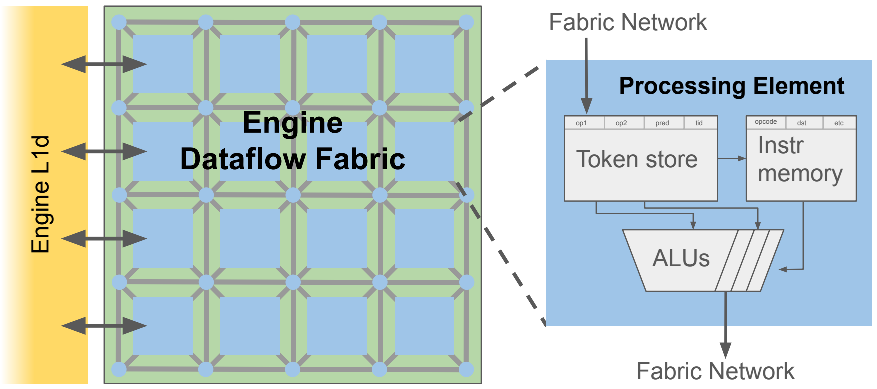


Figure 4.11: Sketch of engine dataflow fabric microarchitecture. A fabric of simple processing elements (PE) are connected by an on-chip network. Each PE holds a small number of instructions, which are issued to the ALU when input operands (with matching thread id) are available. A small number of PEs also connect to memory through the engine's L1 data cache.

4.4.3 Engine microarchitecture

täkō adds one engine to each tile of the CMP. The engine runs all callbacks for the L2 and L3 bank on that tile. It has its own cache-coherent L1 data cache, a small TLB and reverse TLB for address translation, and a spatial dataflow engine to execute callbacks.

Scheduling callbacks in hardware. The scheduler consists of simple logic in hardware and a buffer of pending requests. Upon receiving a callback request, the engine enqueues it in its callback buffer, assigns the callback a unique id, and loads the callback bitstream into the fabric (if necessary). The engine maintains a small bitstream cache, which maps `Morph`s' registered address ranges to their callbacks' bitstreams and tracks which callbacks are loaded on the fabric. Callbacks begin executing once the fabric is ready and all earlier callbacks on the same `addr` have finished.

Dataflow fabric. Callbacks execute on a small dataflow fabric; see Figure 4.11. The fabric is an array of simple processing elements (PEs) connected by an on-chip network. Each PE contains an instruction memory that holds a small number (e.g., 16) of static instructions, a token store that holds intermediate values, and ALUs. PEs issue operations using asynchronous dataflow firing, supporting concurrently executing callbacks via dynamic tag matching [93, 175, 228, 238] on callback ids. Operations work in SIMD fashion across entire cache lines at a time.

Our workloads require only a small fabric (e.g., 5×5) with simple integer operations and few (e.g., 8) concurrent callbacks (see Section 4.8). Our largest `Morph`, for HATS (Section 4.7.2), contains 94 instructions across all its callbacks, less than one-quarter of fabric resources. Our next-largest application contains only 46 instructions. Moreover, across all applications, there are no more than 19 average live tokens when an engine is active (summing across concurrent callbacks). There is thus plenty of room for co-running applications to share engines, even without mechanisms to limit contention (see Section 4.5).

We chose dataflow fabrics for täkō engines because *(i)* callbacks are typically short, *(ii)* callbacks are frequently

Table 4.2: Hardware overhead (state per L3 bank).

L3 tags	8K lines $\times$ 1 bit = 1 KB
Engine L1d, TLB, rTLB	8 KB + 2 KB + 2 KB = 12 KB
Callback buffer	8 lines $\times$ 64 B = 0.5 KB
Token store	25 PEs $\times$ 8 tokens / PE $\times$ 64 B = 12 KB
Instruction Memory	25 PEs $\times$ 16 instr / PE $\times$ $\approx$4 B = 1.6 KB
Total per L3 bank	27.1 KB / 512 KB = 5.3%

executed in parallel, and *(iii)* callbacks are executed repeatedly. Short callbacks map easily onto a small, dynamic dataflow fabric, letting täkō run callbacks near-data with low area overhead. A dataflow fabric can easily run callbacks in parallel by assigning each a unique tag. Alternatively, täkō could execute callbacks on reserved SMT threads [235, 259], but this would either sequentialize callbacks or require multiple, heavy-weight thread contexts. Moreover, constantly re-fetching and decoding the same instructions would be wasteful. Preliminary exploration of SMT threads showed severe performance penalties, and Section 4.8 finds that in-order cores, as proposed in prior work [8, 142]. perform very poorly in täkō.

4.4.4 Putting it all together

täkō's hardware support adds little area to the baseline multicore system (Table 4.2). With 512 KB L3 banks and 64 B lines, the L3 tags need 1 KB to track `Morph` registration. The engines have 8 KB L1d caches, 2 KB TLB and rTLBs (see below), and a 5 $\times$ 5 dataflow fabric with integer functional units. Conservatively overprovisioning the token and instruction memory yields state overhead of 5.3% over an L3 bank. This is comparable to recent fabrics [166, 193, 238], which add roughly 5% area overhead.

4.5 System Integration

By opening up the cache hierarchy to software, täkō touches many aspects of the system stack. This work does not solve every issue, but here we discuss some of the major implications of polymorphic cache hierarchies.

Address translation. Caches use physical addresses, but täkō callbacks need virtual addresses. The engines maintain a reverse TLB (rTLB) for this purpose. The rTLB is eagerly filled when an `onMiss` is scheduled 3; however, we found that this optimization makes little difference in our workloads because rTLB hit ratios are so high. When a callback is scheduled, the engine recovers the virtual `addr` using the rTLB and the physical address from the cache tags 8b. Synonyms (i.e., ambiguity in reverse translation) are not an issue because only one `Morph` can be registered on an address at a time. The engine also keeps a conventional L1 TLB for other data accessed by callbacks, sharing the L2 TLB with the main core.

Table 4.3: System parameters in our experimental evaluation.

Cores	16 cores, x86-64 ISA, 2.4 GHz, OOO Goldmont uarch [5]
Engines	16 engines, 15 int FUs (1-cycle latency), 10 mem FUs, 256-entry rTLB
L1	32 KB, 8-way set-assoc, split data and instruction caches
L2	128 KB, 8-way set-assoc, 2-cycle tag, 4-cycle data array, t̃r̃îp repl., strided prefetcher
LLC	8 MB (512 KB per tile), 16-way set-assoc, 3-cycle tag, 5-cycle data array, inclusive, t̃r̃îp repl.
NoC	mesh, 128-bit flits and links, X-Y routing, 2-cycle pipelined routers, 1-cycle links
Memory	4 controllers, 100-cycle latency, 11.8 GB/s per controller

täkō has several nice features with respect to address translation. Phantom addresses are not backed by physical memory, making huge pages easier to use because fragmentation is less of a concern than in conventional memory allocators [126]. Moreover, the engines' rTLB only needs to cover data currently in the cache, since `onEviction` and `onWriteback` can only be triggered on cached data. Both of these observations mean that the engine rTLB can be small (Section 4.8). We assume that engine TLBs are kept coherent using shootdowns when translations change (e.g., when a `Morph` is registered or unregistered).

OS support. täkō requires operating system support to manage `Morph` registration. The operating system needs to track which address ranges currently have a `Morph` registered along with a pointer to the callback code. Phantom address ranges may require an independent data structure from the page tables, since they use physical addresses that do not correspond to physical memory. `Morphs` also complicate thread scheduling because eviction callbacks can still run even if a process is de-scheduled from cores. In many cases, this is not problematic. But if a process must be fully de-scheduled for some reason, then it is necessary to also flush its `Morphs`' data (i.e., using the `flushData` API). Doing this is feasible but takes time and energy, especially for `Morphs` at the `SHARED` cache.

Multi-tenancy, virtualization, and security. In heavily shared systems with many active `Morphs`, further potential problems arise with thrashing in engines, possible security issues between concurrent callbacks, and virtualizing shared resources. These issues are outside the scope of this work, but we think partitioning application data across L3 banks is a promising solution [170, 203]. That is, the operating system can prevent unwanted contention or interaction between callbacks by preventing them from sharing cache space in the first place.

4.6 Experimental Methodology

Simulation framework. We evaluate täkō in execution-driven microarchitectural simulation. Our simulator shares infrastructure with SwarmSim [102], but supports cycle-level timing throughout the memory hierarchy and models täkō's interface and engines.

System parameters. Except where specified otherwise, our system parameters are given in Table 4.3. We model a tiled multicore system with 16 cores connected in a mesh on-chip network. Each tile contains a conventional out-of-order core (modeled after Intel Goldmont), one bank of the shared LLC, and a täkō engine. Section 4.8 varies these parameters and shows that täkō is effective across a variety of system configurations.

We assume the out-of-order cores support atomic exchange operations (e.g., LL/SC) along with other relaxed atomics. Except where noted, we evaluate engines with a 5×5 dataflow fabric (15 integer PEs and 10 memory PEs) with 1-cycle PE latency. We also evaluate an *idealized engine* with unlimited, 0-cycle latency PEs; i.e., callback latency is only affected by memory latency and data dependencies.

Metrics. We present results for speedup and dynamic execution energy (energy parameters from [193, 222]). We focus on dynamic energy because täkō has negligible impact on static power and to clearly distinguish täkō's impact on data movement energy from its overall performance benefits.

4.7 Evaluation — Case Studies on täkō

täkō's flexible programming interface enables a wide variety of optimizations on the same, general-purpose hardware. We evaluate a sample of four applications that can benefit from täkō to demonstrate:

- täkō supports prior specialized cache hierarchies. We implement two prior designs that accelerate graphs in very different ways [159, 162].
- täkō enables features in software that are impossible without fine-grain visibility over data movement. Specifically, täkō lets the system eliminate unnecessary writes in direct-access NVM and detect suspicious activity.
- täkō's performance is fairly insensitive to its microarchitectural parameters (Section 4.8) and close to an idealized design.

Our case studies depend on being able to observe and interpose on data movement, and are thus not implementable on prior near-data computing (NDC) architectures. täkō provides the missing interface and mechanisms to implement these data-movement optimizations in software.

4.7.1 Accelerating commutative scatter-updates

We begin with an example of how täkō can ***redefine cache semantics*** to accelerate data movement. This study implements PHI [162], a *push-based hierarchy* for commutative scatter-updates, e.g., in graph applications. PHI turns the cache into a large write-combining buffer for commutative operations (e.g., addition). In PHI, the cache contains updates (e.g., deltas), not raw data. When a cache line is evicted, PHI either immediately applies the update in-place or logs the update to be applied later [25, 120]. PHI minimizes memory bandwidth by choosing between these two policies, using the number of updates in the line to decide which is best.

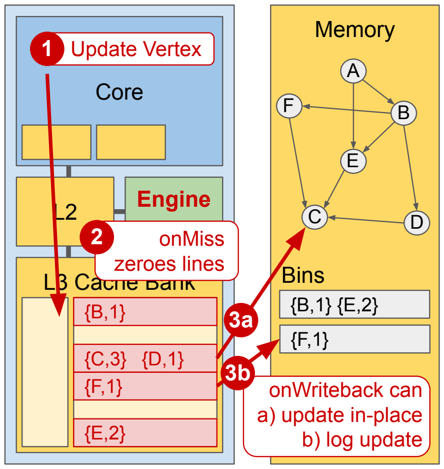


Figure 4.12: täkō lets software re-purpose the cache to accelerate applications. PHI accelerates scatter-updates by buffering updates in-cache and applying them when evicted. Writebacks either apply updates in-place or log updates to be applied later. These optimizations are naturally implemented in täkō via `onMiss` and `onWriteback`.

Table 4.4: täkō callbacks for PHI.

Callback	**Semantics**
`onMiss`	Sets line to identity element (e.g., zero).
`onEviction`	—
`onWriteback`	If # updates > threshold, apply updates immediately; otherwise, log updates for application in “binning” phase.

Description. Figure 4.12 illustrates how täkō implements PHI. The application starts by allocating a phantom address range the same size as the graph’s vertex data. In the first phase, updates are pushed to the phantom region using remote memory operations (RMO) (i.e., relaxed atomic add [212]). If updates 1 miss in the cache, they trigger `onMiss` 2 to initialize the lines with an identity element (e.g., zero for addition), without making any requests down the cache hierarchy. The application then pushes commutative updates to the cache (i.e., write hits). When a line is evicted from the cache, `onWriteback` either directly applies the updates to backing memory 3a or appends them to a “bin” 3b, depending on the number of non-identity values in the line. After completing the edge phase, the main thread calls `flushData` and then streams through the bins to apply deferred updates (not shown).

Why täkō?. PHI’s design fits very well with täkō’s interface. Its implementation requires application- and data-dependent operations on cache lines as they are allocated and evicted. This is exactly the type of data-movement control that täkō enables in software. Moreover, PHI is a prime example of the limitations of prior NDC: PHI requires the ability to intercept misses and writebacks and modify their behavior, which is not possible in traditional NDC.

Evaluation. Figure 4.13 shows results for PageRank with 16 threads pushing updates to a single `Morph` registered at `SHARED`,[3] comparing täkō to a baseline software implementation, a software implementation of update batching

[2]Results are updated from [204] to reflect more accurate modeling.
[3]Due to simulator limitations, we can currently only run PHI at a single level. But täkō’s design allows hierarchical PHI as described in [162],

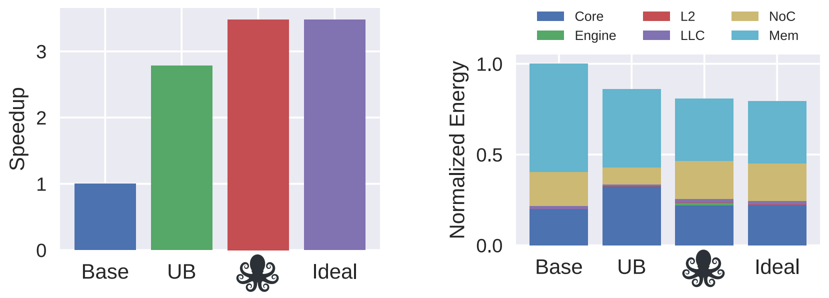


Figure 4.13: PHI results for PageRank on a 8M vertex, 80M edge synthetic graph. täkō improves performance by 3.5×.[2]

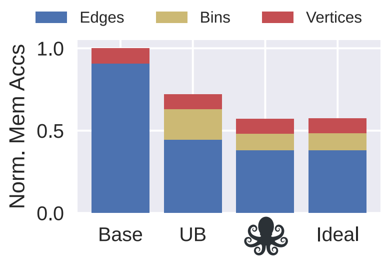


Figure 4.14: DRAM accesses per phase.

(UB) [25, 120], and an ideal dataflow engine. We see similar results as the PHI paper [162]: UB in software gets 2.8× speedup, but täkō gets 3.5× speedup. täkō also reduces energy by 19%, compared to 14% for UB.

täkō achieves its benefits by *(i)* writing to phantom data, which does not incur a memory access on miss; *(ii)* binning updates off the critical path of the main threads on writeback; and *(iii)* reducing memory accesses (by 20%) and core computation (by 32%) compared to UB by buffering updates in the cache and sometimes applying them in-place. Figure 4.14 breaks down memory accesses for each implementation between the edge, bin, and vertex phases of PageRank. UB reduces total accesses by 28% by improving spatial locality via binning. täkō reduces total accesses by 43% by buffering updates in-cache and only binning when there is poor spatial locality, lowering accesses in both edge and bin phases. Further, täkō incurs negligible overheads compared to an ideal engine because `onWriteback` is short (85 cycles and 19 instructions on average), off the critical-path, and most of the latency comes from memory accesses (0.30 accesses per `onWriteback` on average).

4.7.2 Accelerating graph traversals via streams

This second study takes a much different view of accelerating graph applications by using täkō to implement a ***programmable, decoupled stream***. Architectures have long had special support for streaming access patterns [52, 60, 71, 117, 169, 229, 234, 236], many of which use dedicated engines to stream data to the main cores. We demonstrate täkō's support for programmable streams by implementing HATS (hardware-accelerated traversal scheduling) [159], which computes an efficient graph traversal to improve data locality in graph applications.

which would show even better results.

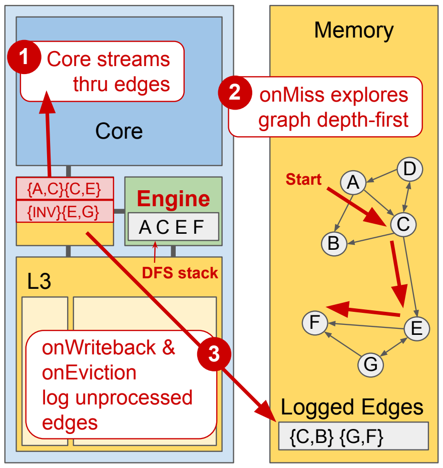

1 Core streams thru edges
Core
{A,C}{C,E}
{INV}{E,G}
Engine
A C E F
DFS stack
L3
onWriteback & onEviction log unprocessed edges
3
Memory
2 onMiss explores graph depth-first
Start
A
D
B
C
F
E
G
Logged Edges
{C,B} {G,F}

Figure 4.15: täkō's stream implementation supports complex decoupled pipelines. HATS improves locality in graphs by traversing the graph in bounded depth-first order so that communities are visited together. `onMiss` provides a simple, data-movement triggered approach to filling a stream.

Table 4.5: täkō callbacks for HATS.

Callback	**Semantics**
`onMiss`	Fills line with edges in BDFS order.
`onEviction`	Logs unprocessed edges.
`onWriteback`	Logs unprocessed edges.

Description. HATS observed that, without expensive pre-processing, it is inefficient to process edges in the order they are laid out in memory. Many graphs exhibit strong community structure [23, 131], so it is much better to process graphs one community at a time. A bounded, depth-first search (BDFS) is a simple traversal order that significantly improves locality. The challenge is that BDFS is a poor fit for cores due to unpredictable control flow, so HATS adds a dedicated hardware engine.

Figure 4.15 illustrates the täkō implementation of HATS. The application initially allocates a phantom address range large enough to hold every edge of the graph (recall that no physical memory is allocated). This phantom address range acts as a stream, where the core reads edges sequentially and the engine supplies edges when requested by `onMiss`. HATS's `onMiss` keeps a small stack and walks the graph in BDFS order, as described in the original paper [159]. Our current implementation of HATS sequentializes all `onMisses` to simplify contention on the shared stack. While the core processes one part of the stream, the prefetcher triggers `onMiss` for subsequent edges. Note that `onMiss` is not guaranteed to be called in strictly sequential order, but this is fine in HATS because minor re-orderings have minimal impact on locality.

However, a more serious concern is that phantom lines can be evicted before the core has processed them. Although this occurs exceedingly rarely, the application cannot tolerate any lost edges. täkō solves this problem by logging unprocessed edges to memory in `onWriteback` and `onEviction`. To know which edges have been processed, the core

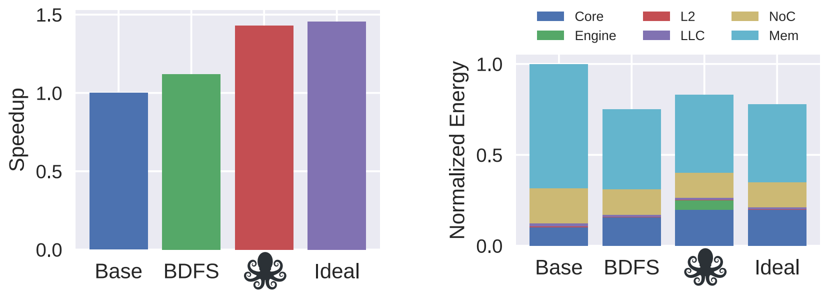


Figure 4.16: HATS results for one iteration of PageRank on uk-2002 graph [54]. täkō improves performance by 43% and reduces energy by 17% vs. the software baseline.

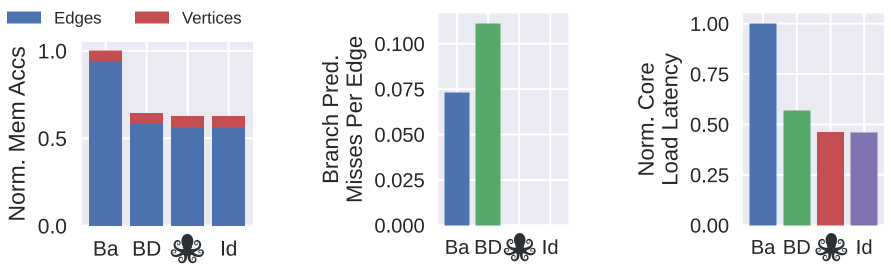


Figure 4.17: HATS performance breakdown. Left: DRAM accesses split by PageRank phase. Mid: core branch mispredictions per graph edge processed. Right: cumulative core load latency.

assigns an INVALID value to processed edges using an atomic exchange (e.g., LL/SC). Any unprocessed edges are logged during `onWriteback` and `onEviction`, and the core processes the logged edges at the end of the iteration.

Why täkō?. HATS is a good example of a streaming computation that runs inefficiently on cores, motivating the need for separate streaming hardware [8, 236, 257]. This case study shows how täkō can support this important class of workloads. For performance, HATS relies on decoupling between graph traversal (on engines) and edge processing (on cores); this is awkward if not impossible to implement in NDC. Moreover, implementing HATS in täkō software lets it support a wide range of graph data formats or traversal heuristics [39, 161], unlike fixed-function hardware.

Evaluation. Figure 4.16 presents speedup and energy results for a single thread of PageRank with the baseline "vertex-ordered" traversal, a baseline software BDFS implementation, täkō, and an ideal engine. Baseline BDFS provides minimal benefits due to extra instructions with complex control flow. In contrast, täkō provides substantial speedup of 43%, approaching the 46% speedup of an ideal engine. täkō also reduces energy by 17%, compared to 22% for ideal.

This speedup is due to *(i)* better cache locality; *(ii)* regularizing control flow on the core; and *(iii)* decoupling edge traversal from the core. Figure 4.17 quantifies these points. All versions incur the same number of accesses during the vertex phase, but the BDFS traversal (also used by täkō) reduces misses to vertex data during the edge phase by 40%. täkō also eliminates branch mispredictions by turning the complex BDFS traversal into simple loop over a sequential stream, whereas software BDFS increases mispredictions by 52%. Finally, täkō reduces memory latency seen at the core by 19% over BDFS by decoupling the edge traversal.

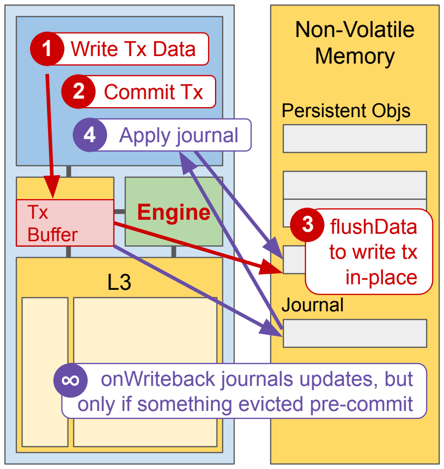

1 Write Tx Data
2 Commit Tx
4 Apply journal
Tx Buffer
Engine
L3
Non-Volatile Memory
Persistent Objs
3 flushData to write tx in-place
Journal
∞ onWriteback journals updates, but only if something evicted pre-commit

Figure 4.18: täkō increases software's visibility over data movement, making some operations dramatically more efficient. When caches are persistent, täkō lets transactions avoid journaling if there are no writebacks before commit. Better visibility thus greatly reduces transaction overheads.

Table 4.6: täkō callbacks for NVM support.

Callback	Semantics
`onMiss`	Sets line with INVALID value.
`onEviction`	—
`onWriteback`	If transaction committed, apply writes immediately; otherwise, journal writes.

täkō achieves significant speedups on HATS, but somewhat lower than reported in [159]. This is because we sequentialize the calls to `onMiss`, whereas [159] re-orders the trace to exploit locality by traversing multiple neighbors in parallel and processing whichever data returns first.

4.7.3 System support: Transactions in direct-access NVM

We next show how ***better visibility over data movement*** enables new features and optimizations. There are many applications where it would be useful to know when data moves in or out of caches: e.g., for immutable data structures [35], intermittent computing [50, 144, 146, 147], checking data integrity [113, 241, 263], debugging and logging [44, 158], etc. This study considers a filesystem on non-volatile memory (NVM) with battery-backed caches, like Intel eADR [94]. The major challenge is to avoid inconsistent states on failure. For this purpose, NVM filesystems employ transactions using journaling, logging, or shadow paging [241, 263].

Description. Figure 4.18 illustrates efficient journal-based transactions in täkō. Like prior transactional memory designs [158], the idea is that if a transaction's writes complete before any have been evicted from cache, then it is safe to push the updates directly to NVM without journaling. (In a sense, the cache *is* the journal.) The application writes all updates to a phantom address range ❶. To commit a transaction, the thread simply flushes the `Morph`'s phantom data from the cache ❷. `onWriteback` either writes directly to NVM (if the transaction has committed) or journals the

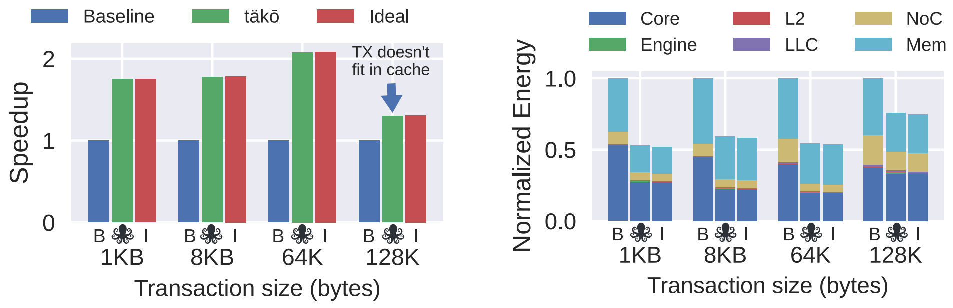


Figure 4.19: Results for NVM journaling microbenchmark at different transaction sizes. täkō improves performance by up to 2.1× and reduces energy by up to 47%.

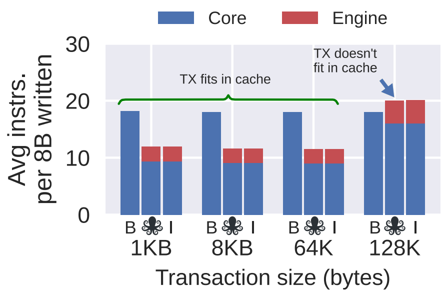
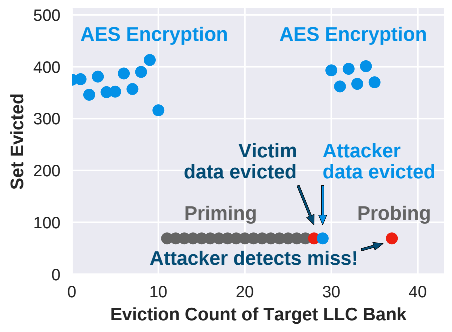


Figure 4.20: Instructions executed for each 8B written in application.

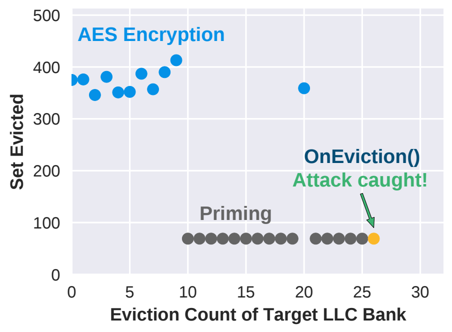
writes (if not) ∞. In the common case where no data is evicted, täkō adds minimal overhead ③. But if data is evicted before commit, then the application must apply the journaled writes to commit the transaction ④. This design permits one in-flight transaction per `Morph` instance, but an application can register many instances. We register at `PRIVATE` because each transaction needs to flush all the phantom data, which is more efficient in the L2.

Why täkō?. Current NVM filesystems must implement transactions conservatively because they cannot observe when data enters or leaves caches. Journaling avoids writing directly to data, but adds instructions and NVM writes. täkō lets filesystems only resort to journaling if data falls out of the cache. Prior NDC systems do not improve visibility over data movement and hence do not enable this feature.

Evaluation. Figure 4.19 shows results for a workload of append-only transactions of different sizes, from 1 KB to 128 KB. As long as transactions fit in the L2, täkō provides up to 2.1× speedup by eliminating unnecessary journaling. täkō executes ≈50% fewer core instructions and ≈36% fewer total instructions (Figure 4.20), yielding large speedup and up to 47% energy savings. täkō achieves the same gains as the ideal engine because the engine mainly performs very simple data copies. When the transaction size exceeds the cache size (i.e., 128 KB), `onWriteback` falls back to journaling and performs closer to the baseline. However, täkō still outperforms the baseline by filling the journal in `onWriteback`, off the critical-path of the core.



(a) Attack succeeds in baseline.



(b) Attack detected in täkō.

Figure 4.21: Prime+probe attack on AES encryption tables at the L3. Without täkō, the attack succeeds with the victim unaware. täkō detects the attack immediately.

Table 4.7: täkō callbacks for detecting side-channel attacks.

Callback	Semantics
`onMiss`	—
`onEviction`	Interrupt main thread.
`onWriteback`	—

4.7.4 Detecting side-channel attacks

Finally, we demonstrate täkō's ***security*** benefits by showing how it can defend against prime+probe attacks [138] at the shared cache. This study emphasizes the additional functionality enabled by better visibility over data movement. Specifically, we demonstrate that täkō enables fine-grain monitoring of data for side-channel attacks [20, 74, 81, 98, 115, 132, 138, 163, 211, 252].

Threat model. We consider a scenario with attacker and victim threads running on separate cores in a CMP with shared last-level cache. The attacker detects when the victim accesses a vulnerable data structure (e.g., AES tables) to reverse engineer secure data (e.g., AES keys). We consider a prime+probe attack, but prior work has used similar techniques to defend flush+reload, evict+time, and cache+collision [43, 72].

Description. The prime+probe attack [138] leaks information about a victim process simply by detecting which cache sets the victim accesses, as shown in Figure 4.21a. The attacker starts by priming a target cache set with its own data. After the victim has accessed its secure data, the attacker then monitors how long it takes to probe its own data. Long latency (due to cache misses) reveals to the attacker which sets the victim has accessed, and thus leaks the victim's access pattern. This prime+probe attack has been shown to leak entire AES keys [74, 98, 132].

täkō gives the victim visibility over movement of their secure data. Specifically, to detect a prime+probe attack, the victim needs to know when data is evicted. The application registers a "real data" `Morph` for the address range of its secure data (e.g., AES tables). The `Morph` only implements one callback, `onEviction`, which simply interrupts the main thread whenever any cache line containing the AES tables is evicted. This interrupt lets the victim defend

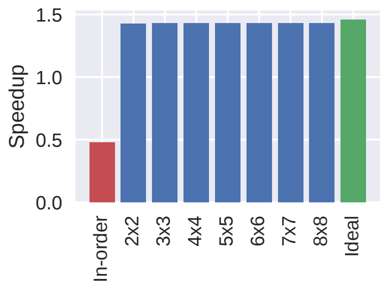


Figure 4.22: Sensitivity to engine fabric with HATS.

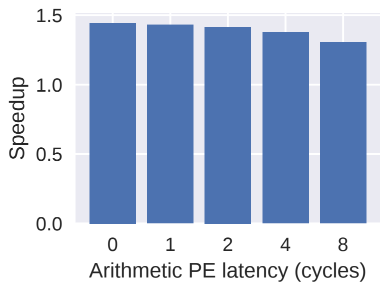


Figure 4.23: Sensitivity to arithmetic PE latency with HATS.

itself from attack [20, 172, 211]. Figure 4.21b shows a cache-eviction trace of an attack that is successful without täkō (left) and unsuccessful with täkō (right). täkō interrupts the victim during the probe phase of the attack *before any information is leaked.*

Why täkō?. täkō exposes software to previously invisible data movement. Although active attackers can time cache accesses to expose microarchitectural state, passive victims might never even know they were attacked. täkō provides victim applications the tools to monitor data movement for cache attacks. This allows victims to take control over their data and defend themselves pro-actively. Like transactions above, visibility over data movement is the key to this defense, and prior NDC systems offer no solution.

4.8 Evaluation — Sensitivity Studies

Engine microarchitecture. We study täkō's sensitivity to engine microarchitecture on HATS. HATS is most sensitive to the fabric because its `onMiss` is the longest callback among our benchmarks. Figure 4.22 evaluates different dataflow-fabric sizes, as well as an in-order core and ideal. Dataflow vastly outperforms in-order, but performance plateaus with small fabrics. We use a 5×5 fabric, which is within 1.8% of ideal. Figure 4.23 evaluates HATS on a 5×5 fabric, varying arithmetic PE execution latency. We use single-cycle latency, but even at eight cycles speedup only decreases to 30% from 43%. This is because memory-level parallelism, not arithmetic throughput, is what matters most for täkō (Section 4.4.3).

Core microarchitecture. Figure 4.24 evaluates PageRank on PHI with different core microarchitectures. Speedup is unchanged because PageRank is memory-bound. Beefier cores improve performance in absolute terms on decompression and HATS, but täkō's speedup is affected little.

Scalability. Figure 4.25 evaluates PageRank on PHI across different system sizes. (Memory bandwidth scales proportionally with cores.) täkō outperforms update batching by 7.0%, 25%, and 19% at 8, 16, and 36 cores, respectively. Hierarchical PHI would improve PHI's speedup further at larger core counts by reducing cross-chip coherence traffic.

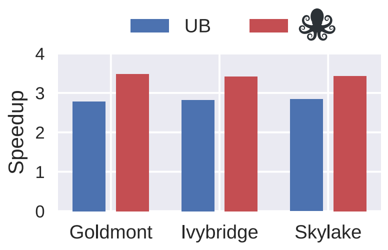


Figure 4.24: PHI results for PageRank on a 80M-edge synthetic graph. täkō performs similarly with all core microarchitectures.

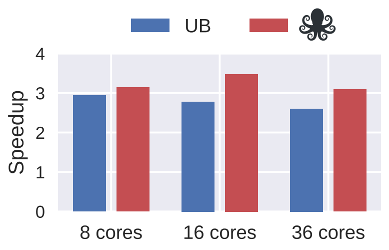


Figure 4.25: PHI results for PageRank across different numbers of cores/threads. täkō performs well across configurations.

Callback-buffer size. The NVM journaling benchmark invokes many concurrent `onWritebacks` when flushing data, stressing the callback buffer. Varying the callback buffer from 1 to 64 entries, performance plateaus at 4 entries. Accordingly, we use 8 entries as a practical but sufficient size in our evaluation.

rTLB size. Finally, we swept rTLB size from 256 to 1024 entries with both 4 KB and 2 MB pages, and found that performance varied by at most 2.1%. We use 256 entries with 2 MB pages.

4.9 Summary

Many inefficiencies in current systems are the result of an outdated hardware-software interface that gives software too little visibility and control over data movement. *Polymorphic cache hierarchies* expand the hardware-software interface to expose more data movement to software. täkō is an efficient, general-purpose implementation of a polymorphic cache hierarchy that massively reduces the innovation barrier for data movement features and optimizations. Through multiple case studies, we demonstrated täkō's ability to specialize data movement for different application domains with a single hardware architecture.

Chapter 5

Leviathan: A Unified System for General-Purpose Near-Data Computing

Near-data computing (NDC) is a popular approach to reduce data movement. Instead of pulling data closer to compute as with conventional memory hierarchies, NDC architectures move compute closer to data. Traditional in-memory NDC shows large benefits for applications with little data reuse, but it fails to exploit the locality present in most workloads. Blindly moving all compute to memory can actually harm performance [6, 90, 142, 224, 258]. This limitation is addressed by *in-cache* NDC [2, 4, 7, 8, 12, 40, 55, 62, 89, 109, 113, 125, 127, 142, 145, 162, 178, 179, 200, 204, 209, 218, 226, 246, 254, 256, 257, 259–262] which augments a cache hierarchy with processing capability. In-cache NDC allows systems to move compute closer to data while also exploiting locality, unlocking the full potential of data-centric computing.

The problem: Prior NDC is too limited and hard to use. Despite the large benefits promised by NDC, there remain significant roadblocks to its practical adoption in general-purpose systems. Most NDC proposals target a narrow range of application domains and only support a limited subset of NDC's design paradigms (Table 2.2). But it is not scalable or practical to add new hardware for every potential application domain. Some recent work has started to address this challenge via *programmable* NDC, where software can configure the operations that execute near data [22, 142, 159, 204, 234]. However, *existing programmable NDC is still insufficient because it only targets a limited subset of the broad NDC design space.*

Beyond limited scope, prior designs also expose too many microarchitectural details to the programmer. Specifically, since existing NDCs rely on the underlying caches or DRAM for data storage, their designs often require data to fit within and align to cache lines [40, 87, 142, 162, 204, 261, 262]. Exposing such microarchitectural details to software, let alone forcing programmers to reason about them, increases programming difficulty and makes NDC unapproachable.

Opportunity and insight. We observe that neither of these issues is fundamental to NDC. With the goal of designing a

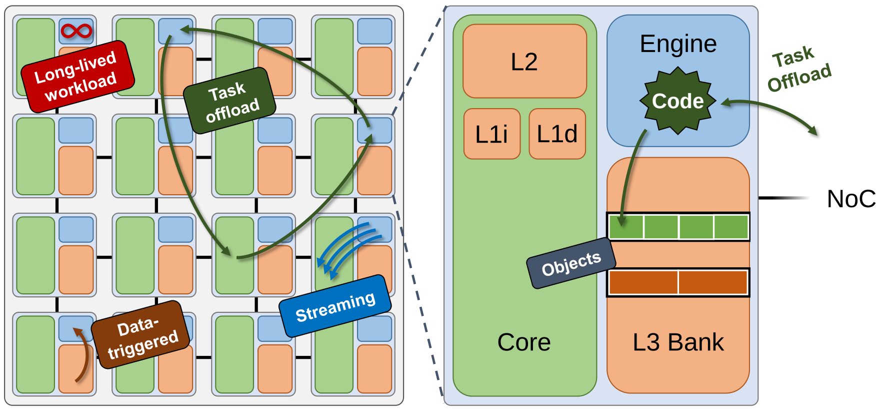


Figure 5.1: We divide prior near-data computing (NDC) into four paradigms. Leviathan supports all NDC paradigms, executing code on near-data engines at the time and location dictated by the paradigm. Programmers write Leviathan programs via a simple reactive-programming interface, and Leviathan hardware ensures that objects are efficiently packed within cache banks.

practical NDC system, we perform an extensive study on prior NDC proposals and build a taxonomy that captures their similarities and differences (detailed in Section 2.7.2). We find that prior designs largely fall into one of four main paradigms (Figure 5.1).

Prior work has treated each paradigm separately, but we observe that a similar structure underlies them. Each paradigm can be roughly broken down into three components: *what* to execute, *where* to execute, and *when* to execute. By placing general-purpose hardware near caches, programmable NDC addresses "what," but "when" and "where" remain unsolved. A system can support all paradigms only if it has the flexibility to trigger computation at the right time and place.

The other challenge is to avoid exposing microarchitectural details to the programmer. The main issue is that NDC requires data to be entirely within a single cache bank to maximize locality. Prior work put this burden on the programmer [40, 87, 142, 162, 204, 261, 262], requiring them to be aware of and optimize for the system's cache-line size, but there is no reason why this should be the case. Instead, the programmer can tell the NDC system the granularity of its data, and the system itself can optimize locality.

Our approach. We propose Leviathan, a polymorphic cache hierarchy that unifies a wide range of prior NDC designs under a simple, actor-based reactive-programming interface. Figure 5.1 illustrates a Leviathan system executing exemplar NDC workloads from each paradigm in our taxonomy. **Task offload** involves short tasks explicitly invoked by a core (or another NDC action) to execute near a target object in the hierarchy. **Long-lived** workloads perform large tasks independently from cores and run near-memory or -cache to avoid polluting cores' caches. **Data-triggered** actions are implicitly executed on objects as they move through the cache hierarchy. And **streaming** allows a decoupled, near-data producer to continually feed a core with data.

To support all paradigms, Leviathan provides a reactive-programming interface. In actor-based reactive program-

```
class Actor: # combines data and near-data action
  int data

  # action executes near ‘‘data’’ in the hierarchy
  void action(int update):
    atomicAdd(data, update)

# core offloads an action to execute on an ‘‘actor’’
invoke actor->action(newUpdate)
```

Figure 5.2: Example implementation of a remote memory operation (RMO) in Leviathan using the actor interface. The actor encompasses a near-data action and the data which the action accesses. A core (or other action) explicitly invokes the action to execute near the data.

ming, an *actor* is an *object* associated with specific *actions* that are invoked by external triggers [194]. In Leviathan, actions are NDC functions executed near-data in response to paradigm-specific triggers. Figure 5.2 shows an example actor which implements a remote memory operation (RMO). The actor includes the data to be accessed and a function that implements the desired RMO, atomic RMW in this example.

A key contribution of Leviathan is providing data locality transparently to applications. Leviathan can perform data management itself because it knows an action's access granularity — i.e., the actor itself. Moreover, Leviathan provides a custom memory allocator that ensures objects are located entirely within one cache bank to maximize locality (right side of Figure 5.1).

To support multiple NDC paradigms on the same hardware, Leviathan takes inspiration from prior NDCs which incorporated programmable compute within the cache hierarchy [22, 142, 204, 235] by distributing *near-data engines* to execute actions on actors. The engines also contain hardware scheduling logic that, in combination with microarchitectural support in the cores and caches, execute code near-data at the right time and place. We explain how each NDC paradigm maps to a combination of actions and triggers, and describe the necessary runtime and microarchitectural support for each.

The end result is a polymorphic cache hierarchy that unifies prior NDC systems on the same hardware while providing a simple-to-use programming interface.

Contributions. This work contributes the following:

- *NDC taxonomy.* We analyze prior NDCs to identify their similarities and differences (detailed in Section 2.7.2). This leads us to the necessary mechanisms for a practical, unified NDC system.
- *Programming interface.* We propose a simple and flexible reactive-programming interface which allows programmers to implement a wide range of NDC applications without worrying about hardware details.
- *Architecture.* We propose a single architecture that supports all four NDC paradigms and provides microarchitectural support to control object placement so that all data resides within the same cache bank.

Summary of results. We evaluate Leviathan on four case studies to demonstrate *(i)* the importance of supporting multiple NDC paradigms on a single system, and that *(ii)* hiding microarchitectural details from the programmer eases programming effort and improves performance.

- *Commutative scatter-updates:* Leviathan implements PHI [162], a graph accelerator which uses multiple NDC paradigms to improve the performance of commutative scatter-updates in graphs. Leviathan is the first system to provide all the necessary NDC support in a general-purpose way, achieving 3.7$\times$ speedup.
- *In-cache data transformation:* Leviathan enables decompressing objects as the data moves through the hierarchy. Leviathan's programming interface abstracts away microarchitectural details to handle objects of any size without any programming complexity, and Leviathan achieves up to 2.4$\times$ speedup across object sizes.
- *Pointer chasing:* Leviathan reduces network overheads on linked-list searches by offloading lookups to the cache banks containing the data. Leviathan performs well across a wide range of object sizes, achieving up to 2.1$\times$ speedup.
- *Decoupled graph traversals:* Leviathan implements HATS [159], a complex decoupled streaming application, achieving 1.7$\times$ speedup. Leviathan's streaming interface allows arbitrary data access patterns, unlike prior affine-based designs [235, 236], and its stream interface is much simpler to program and more effective than prior general-purpose NDC designs that do not explicitly support streams [204].

Leviathan adds just $\approx$6% area overhead to a baseline multicore, similar to prior near-data systems, and achieves performance within 11% of an idealized design.

5.1 Leviathan Overview

Leviathan's goal is to provide a truly *polymorphic cache hierarchy* that unifies prior NDC paradigms and is easy to program. Similar to recent programmable NDC systems [8, 22, 142, 169, 204, 234, 236, 257], Leviathan adds general-purpose engines near the cache banks of a multicore, letting software run arbitrary compute near-data. To support all four NDC paradigms, Leviathan further adds microarchitectural support to execute software at the right time and place. And Leviathan exposes all this capability to programmers via a simple programming interface that hides unnecessary microarchitectural detail from software.

Programming interface. The programming model comprises an object-oriented memory allocator and an actor-based interface for each of the NDC paradigms. Each paradigm operates on actors provided by the allocator to ensure that Leviathan maintains intra-bank data locality.

NDC paradigms mainly consist of three components: *what* action to execute, *when* to execute it, and *where* to execute it. In Leviathan, the application provides the actions to execute and indicates the NDC paradigm to use. It is the responsibility of Leviathan's runtime and hardware support to correctly execute the action, depending on the paradigm.

Table 5.1 breaks down the actions associated with each paradigm. Task offload and long-lived workloads both

Table 5.1: Actions associated with each NDC paradigm.

Paradigm	**Actions**
Task offload	Arbitrary actor-specific function
Long-lived	Arbitrary actor-specific function
Data-triggered	Actor constructor & destructor
Streaming	Actor-specific producer function

involve actor-specific actions explicitly triggered by a core or another near-data action, so Leviathan needs to execute the action when requested at the appropriate location. Data-triggered NDC involves two actions—actor constructors and destructors—that are triggered on specified actors when they are either inserted or evicted from the cache. And streaming involves a producer (long-lived workload) and consumer (any other thread) along with additional support to push and pop objects from a shared communication channel.

Hardware. On top of a baseline, cache-coherent multicore, each tile is augmented with a near-data engine (Figure 5.1). The commonality across NDC paradigms is the ability to execute an application-defined action on a specified object, so Leviathan's engine contains a lightweight, programmable processor to execute actions. The difference across paradigms is the way in which actions are triggered. This is handled by the engine's hardware scheduler, which provides microarchitectural support for each paradigm. The other main engine components are a small, coherent cache and thread context buffer to manage local state for currently running actions.

5.2 Motivation

We demonstrate the power of a unified NDC system by implementing an application that requires functionality from multiple paradigms. Prior work focuses on a single application or paradigm, and no prior system covers all paradigms. Leviathan's unification of all four paradigms is essential to providing a truly polymorphic cache hierarchy.

5.2.1 Accelerating commutative scatter-updates

PHI [162] is a push-based cache hierarchy optimized for commutative scatter-updates, e.g., in graph applications. In PHI, each cache is a large write-combining buffer for commutative operations (e.g., addition) that contains partial updates (e.g., deltas) instead of raw data. When cache lines are evicted, PHI either immediately applies the updates in-place or logs them for later processing [25, 120], dynamically choosing the policy that minimizes memory bandwidth.

PHI spans multiple NDC paradigms. PHI is mostly **data-triggered**: PHI changes cache insertion to zero-out lines and changes eviction to perform updates in-place or log them. However, a large portion of PHI's benefits come **task offload** by using remote memory operations (RMOs) to apply partial updates. This aspect of PHI is not emphasized in prior work, which assumes that the caches support whichever RMOs are needed by each graph application. Given the

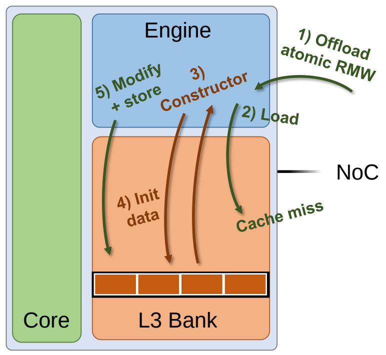


Figure 5.3: Leviathan implements PHI [162] by enabling multiple NDC paradigms to work together. The figure demonstrates how an **offloaded RMW task** leads to a **data-triggered action** that implements PHI's insertion semantics. A similar process happens on cache evictions.

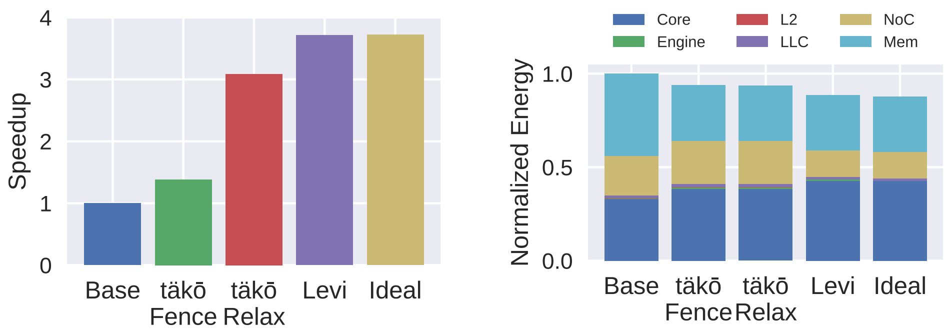


Figure 5.4: PHI results for PageRank on a 4M vertex, 40M edge synthetic graph. Leviathan improves performance by 3.7×.

diversity of graph applications [24], it is essential that multiple NDC paradigms are supported in a general-purpose system to make techniques like PHI practical.

5.2.2 Leviathan's implementation of PHI

Figure 5.3 illustrates Leviathan's implementation of PHI, where task offload and data-triggered actions work together to treat the L3 cache as a write-combining buffer.

1) A core offloads a **RMW task** to this L3 bank to perform an atomic RMW on an object (e.g., updating a vertex's rank in PageRank). **2)** The **RMW task** loads an object which is not cached. **3)** The cache miss triggers a second **insertion action**; several objects are packed into one cache line. **4)** The **insertion action** (i.e., object constructor) initializes the objects with zeros and completes. **5)** The **RMW task** now updates the object.

5.2.3 Evaluation

We evaluate Leviathan to demonstrate the benefits of multi-paradigm support. The comparisons are a baseline implementation of push-based PageRank and tākō's implementation of PHI. tākō is a programmable NDC for only **data-triggered actions**. Since tākō does *not* support task offload, it approximates the benefits of RMOs by assuming cores support all necessary atomic instructions without memory fences (i.e., relaxed atomics [15, 212]). We evaluate täkō with and without relaxed atomics to demonstrate the importance of this dimension of PHI.

Figure 5.4 shows results for PageRank with 16 threads. (Methodology in Section 5.5.) Leviathan achieves 3.7× speedup, whereas täkō gets 3.1× and 1.4× speedup with and without fenceless atomics, respectively. Leviathan also reduces energy by 11%, vs. 6% for täkō. And Leviathan comes within 1% speedup and energy of an idealized engine.

Leviathan achieves its benefits by *(i)* reducing memory accesses with **data-triggered actions**; *(ii)* eliminating memory-fence overheads with **task offload**; and *(iii)* reducing NoC traffic with **task offload**. Both Leviathan and täkō reduce memory accesses by conditionally binning updates on cache evictions using data-triggered actions. The benefits of eliminating memory fences is shown by comparing täkō Fence and täkō Relax in Figure 5.4. Fences serialize memory accesses and impose a severe performance penalty. Relaxed atomics are essential for täkō to realize large benefits.

Meanwhile, Leviathan simply offloads tasks near-data, improving performance vs. täkō without requiring major ISA extensions. Leviathan reduces NoC traffic by 40% vs. täkō. These benefits are unachievable in täkō because it does not support task-offload.

Discussion. All in all, Leviathan's ability to support multiple NDC paradigms enables a variety of performance optimizations that are unsupported by prior NDC systems. Leviathan is the only multi-paradigm, general-purpose NDC system, and thus the first truly polymorphic cache hierarchy.

5.3 Leviathan Programming Interface

Leviathan's programming interface works to overcome the two major limitations of prior work: *scope* and *hardware abstraction*. Leviathan extracts the commonalities across NDC paradigms while supporting their differences. The commonalities are actors (objects) with paradigm-specific, near-data actions (methods) that execute asynchronously from the main thread and communicate results via futures. The key differences across paradigms are when and where to execute the actions. Leviathan's interface abstracts hardware by letting applications specify the data it wants to access, and then Leviathan performs all data management behind the scenes via a custom memory allocator.

We start by covering common features before describing each paradigm's interface.

5.3.1 Building block: Actors

The underlying mechanism for implementing all paradigms is the actor model [84, 194]. An actor is an object (i.e., class) associated with one or more near-data actions (i.e., methods). Note that we distinguish "actor" and "object", where object just refers to data, because not all objects in Leviathan are actors (specifically, with streams, as discussed in Section 5.3.6).

A programmer uses Leviathan by defining an actor class which implements the necessary actions for the paradigm of interest (e.g., Figure 5.2). All actor instances are allocated with Leviathan's allocator (Section 5.3.3) so that data management is hidden from the application. Near-data actions are then executed on allocated actor instances at a time and place in the cache hierarchy according to the designated paradigm.

5.3.2 Building block: Communicating results with Futures

```
class Future<R>:
  R wait()             # for receiver
  void send(R result) # for sender
```

Figure 5.5: Leviathan's Future interface.

All paradigms except data-triggered require the ability to communicate results from near-data actions back to a core. For this functionality, Leviathan provides a Future<R> (Figure 5.5) which is filled with an object of type R from an associated action running asynchronously from the core. To receive a result, the core simply waits on the Future<R> until the object is available.

5.3.3 Building block: Memory allocator

The purpose of Leviathan's memory allocator is to abstract away microarchitectural details so that the application can specify the actors it wants to operate on, and Leviathan manages packing and padding of their data into cache lines.

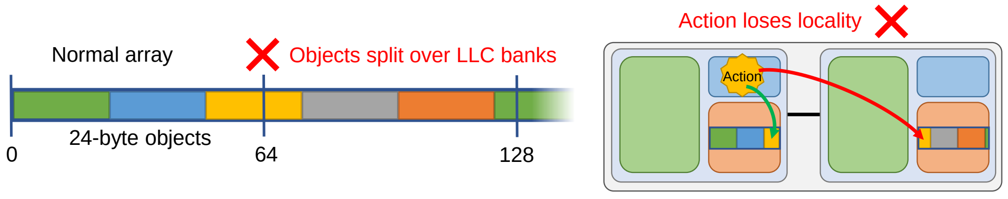
```
class Allocator<T>:
  T* allocate()
  void deallocate(T* object)
```

Figure 5.6: Leviathan's object-oriented memory allocator.

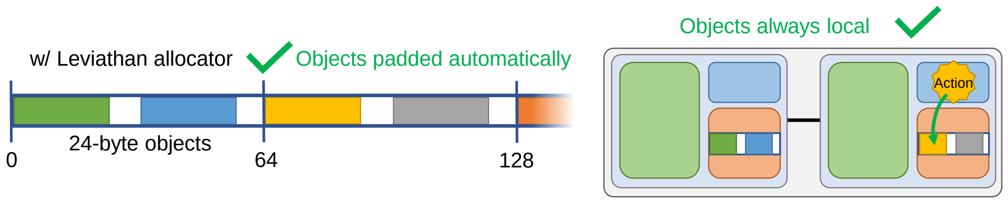
Application interface. Leviathan's Allocator<T> (Figure 5.6) provides simple methods to allocate and deallocate objects of type T. Depending on NDC paradigm, applications may not use the allocator directly; data-triggered actors are allocated and freed implicitly by hardware.

Implementation. The allocator has three jobs: padding objects to be cache-aligned; mapping large objects to the same LLC bank; and packing objects to not waste main memory.

Small objects. When small objects do not evenly divide a cache line, allocating an array of objects normally will result in some objects spanning multiple lines (see Figure 5.7a). This hurts NDC in particular because actions are forced to fetch part of the object from another cache bank, rather than finding all data locally. To avoid this issue, Leviathan's allocator pads objects to the next power-of-two size (see Figure 5.7b).



(a) Allocating a normal array of objects splits objects across LLC banks, losing data locality for NDC actions.



(b) Leviathan's allocator pads objects to maintain data locality for NDC actions.

Figure 5.7: Padding objects in the cache is necessary to maintain data locality for NDC actions. This example demonstrates allocating 24B objects for a cache with 64B lines.

Large objects. Large objects reside on multiple banks because consecutive cache lines typically map to distinct banks [248]. This is impossible to solve in software alone. Leviathan maps large objects to the same bank by modifying the LLC's bank-index function to ignore LSBs of an address, depending on the object size. For example, for an object that is four cache lines in size, ignoring two ($\log(4) = 2$) LSBs will map all lines of the object to the same bank.

Memory compaction. Padding causes fragmentation that wastes memory. Our insight is that padding matters for NDC *in the cache*, but is unnecessary in memory. Leviathan thus aligns objects to cache lines in the cache, but packs them densely in memory to avoid fragmentation.

Leviathan uses a one-to-one translation between cache addresses and memory addresses, similar to a phantom address or memory overlay [40, 204, 206]. The translation is a simple computation which only requires the object size and base array address both for cache and memory addresses (see Figure 5.12). This dynamic padding is impossible in conventional software alone, because software has no control over the cache-to-memory interface.

This design requires contiguous address ranges in both cache and memory. Accordingly, Leviathan's allocator is pool-based and allocates from a large, contiguous physical memory range. Alternatively, one could add an additional page-level translation layer between the LLC and memory at some additional overhead and complexity [206].

5.3.4 Paradigms: Task offload & long-lived NDC

We now discuss how Leviathan deploys these building blocks to support the four NDC paradigms in Section 2.7.2.

The first NDC paradigms we discuss are task offload and long-lived workloads. We observe that, although their usage and underlying hardware can differ, the software interface is essentially the same [22]. They both involve a core or action explicitly invoking another action, be it short- or long-lived. Task offload frequently requires a return value and long-lived frequently requires a specific location in the hierarchy to execute, but neither is for certain. Accordingly, we group both paradigms into a single interface with options to distinguish the aforementioned differences.

```
class A:   # example actor
  U f(...) # action 1
  V g(...) # action 2

A* a = Allocator<A>::allocate()

# invoke creates a future, which holds a return value
Future<U> u = invoke a->f(...)         # location is dynamic
Future<V> v = invoke[REMOTE] a->g(...) # vs. static
```

Figure 5.8: Leviathan's task offload actor interface.

Invoking tasks. Offloaded tasks operate on an object, which is expressed in Leviathan as actions on an actor. The application first allocates an actor and triggers an action using the invoke keyword (see Figure 5.8), similarly to Livia [142]. In the figure, invoke offloads the method f to execute near the provided actor a , returning a Future<U> that is filled when the task completes.

Offloaded tasks can take any number of arguments and return any type, including void (no return value). The optional [location] parameter indicates in which level of cache hierarchy the task should execute. There are three options:

- LOCAL: The invoker's local engine.
- REMOTE: The engine near the object's LLC bank.
- DYNAMIC (default): Leviathan probes down the cache hierarchy to locate the object, and executes the task nearby.

The user can also indicate a task wants EXCLUSIVE permissions as hint to DYNAMIC scheduling.

Offloaded tasks can themselves invoke further tasks in continuation-passing style, eventually returning a single value to the original caller by calling send on the Future.

5.3.5 Paradigm: Data-triggered actions

Data-triggered NDC often involves interposing on cache misses and evictions to perform application-specific handling of the data being moved. As täkō identified, letting software fill in data on misses and process data on evictions (instead of fetching from or evicting to the next level of the hierarchy, respectively) is a mechanism that can replicate the functionality of custom data-triggered NDCs. Accordingly, this "phantom" data only resides in-cache and is not backed by physical off-chip memory [40, 204], since it is constructed when filling a cache line and destructed when evicting the cache line. In Leviathan, actors' data is phantom for data-triggered NDC, and their actions are the actors' constructor and destructor (Figure 5.9).

```
class A: # example actor
  # actions
  A(Morph<A>* view)                # constructor
  ~A(Morph<A>* view, bool isDirty) # destructor

class Morph<T>:
  Morph<T>[] views        # engine-local state

  T& getActor(int offset) # for cores
  int getOffset(T* actor) # for actions

Morph<T> register(Type morphType, Type actorType, Location location,
  int numActors)
void unregister(Morph<T>* morph)
```

Figure 5.9: Leviathan's data-triggered actor interface.

We already demonstrated an example with phantom data in Leviathan's implementation of PHI, where the actor's data is initialized with zero on a cache miss and conditionally written back to memory or logged on a cache eviction (Section 5.2.2). Another later example in Section 5.6.1 will show a constructor accessing compressed data to decompress into a phantom actor on a miss.

Registration. Applications use data-triggered NDC by registering a Morph object that encapsulates a phantom address range for a given actor type. On registration, the morph allocates an address range (not actual memory space) for the actors' phantom data. Registration uses an Allocator behind the scenes so that actions always have their data in a single cache bank. Registration returns a Morph object that encapsulates the address range and actor instances. Each engine has a separate view (i.e., copy) of the morph which may contain engine-local state for actions running on that engine, similar to conventional thread-local state.

Actions. The two actions are the actor's constructor and destructor, which correspond to onMiss and onEviction/onWriteback in täkō. Both actions are provided with a pointer to the morph's view for the local engine to access shared data. On a cache miss to the actor's data, the constructor is executed to initialize the data in-place. Similarly, the destructor is executed when the data is evicted, and it is also passed a boolean argument denoting whether the data was clean or dirty.

The major difference from täkō is that code can be much simpler because actions execute on objects instead of cache lines. The application just needs to handle construction and deconstruction of single objects, not worrying about number of objects per cache line and alignment of data within cache lines.

5.3.6 Paradigm: Streaming

Our streaming interface takes inspiration from decoupled streaming accelerators in which a near-data thread pushes data to a core [159, 234–236, 247] (see Figure 5.10). But unlike prior work, Leviathan is not restricted to a specific data

```
class Stream<T> extends Morph: # actor base class
  # consumer interface
  Stream<T>(int queueSize)
  Future<T> next() # consume stream
  void terminate()

  # producer interface
  void genStream()    # action: generate stream
  void push(T object) # called by genStream, blocks when buffer is full
```

Figure 5.10: Leviathan's stream actor interface.

size for stream entries and streams can execute arbitrary logic for any desired pattern (vs. pre-defined affine or indirect patterns).

Streams are essentially long-lived NDC threads, but they are so ubiquitous and their communication pattern with cores so regular, that it is worth treating them as a separate paradigm with a custom interface and microarchitectural support. In fact, Leviathan's stream implementation actually uses both long-lived and data-triggered paradigms under the hood. The crux of the stream is a long-lived thread that pushes new entries onto a circular buffer in shared memory. Consuming the stream, however, involves loading consecutive phantom actors registered with data-triggered actions that construct the actors by copying from the corresponding entry of the circular buffer. The benefits of this data-triggered mechanism are *(i)* enabling the consumer thread to simply issue loads to read off the stream and *(ii)* being able to stall load-triggered constructors at the engine when stream generation is slower than consumption, a mechanism not possible with loads to non-phantom data. Importantly, Leviathan's interface hides all the data-triggered details from the application, exposing only a simple Future-based function to consume stream entries.

Initialization. A stream is initialized by specifying the object type and the size of the buffer which will hold objects produced but not yet consumed. The buffer is a circular queue in shared memory that contains objects (*not* actors), using the Leviathan allocator.

Producer. Data is pushed onto the stream by a long-lived thread, genStream, running on the tile's local engine. genStream calls push, a blocking function, to push onto the buffer. When the buffer is full, the function blocks until the core consumes an entry.

Consumer. next provides a Future<T> which will contain the next stream entry when available. Under the hood, next performs two actions: *(i)* initializes the Future<T> with the next entry and *(ii)* pops the entry off the stream. To fill the Future<T>, next loads from a phantom address range, which is also allocated using the Leviathan allocator, through a Morph hidden from the application. The load triggers the phantom actor's constructor to read from the stream's buffer when it is not empty. After the load is issued, next pops the entry off the stream by incrementing the core's stream head pointer and sending a message to the engine when the head pointer has incremented to a new cache line, unblocking push to allow the producer to continue.

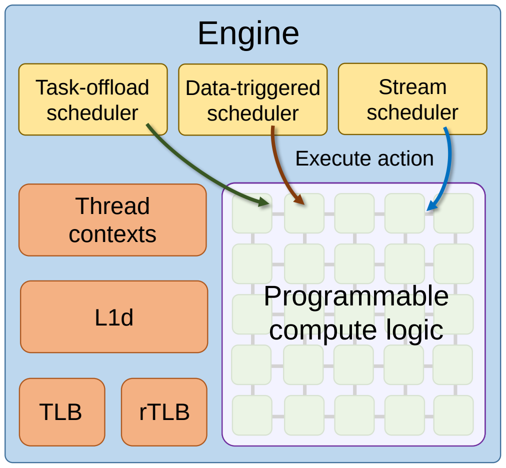


Figure 5.11: Each near-cache engine contains programmable compute to execute actions, a thread context for each running action, schedulers for each NDC paradigm, and an L1d, TLB, and rTLB.

5.3.7 Interaction across paradigms

One of Leviathan's major strengths is the ability to support applications in which multiple NDC paradigms directly interact with each other. We already demonstrated an example with PHI [162], which combines task offload with data-triggered actions (Section 5.2). It is possible to further combine PHI with streaming by decoupling the graph traversal from the cores to improve cache locality (see Section 5.6.3). Leviathan's streaming mechanism itself is implemented through a combination of long-lived workloads and data-triggered actions. These examples demonstrate the importance of supporting multiple paradigms to have a truly polymorphic cache hierarchy.

5.4 Leviathan Architecture

Leviathan's hardware support includes a near-cache engine for executing each NDC paradigm's actions along with core and cache modifications to assist in both executing actions at the right time and place, and managing object placement throughout the memory hierarchy. We start by discussing hardware support shared by all paradigms, and then cover each paradigm's individual support.

5.4.1 Shared infrastructure: Near-cache engines

Similar to recent NDC architectures [8, 142, 177, 204, 236, 257], Leviathan extends a baseline multicore processor with near-cache engines (Figure 5.11). The compute logic, which can be any programmable resource (e.g., core, FPGA, dataflow fabric), is used to execute all application-provided NDC actions. We evaluate Leviathan with engines as dataflow fabrics due to their high performance per area on short, repeated functions (as demonstrated with täkō). The L1d and TLB give engines coherent access to the shared memory space. The rTLB is used to translate cached physical addresses back to virtual addresses for action execution.

Finally, a thread buffer stores local state for all executing actions. To prevent deadlock, there must always be at least one thread context not reserved by an offloaded task. Otherwise, all tasks might be waiting for a data-triggered constructor to execute, but the constructor is waiting for a free context. In our evaluation, we evenly split contexts between offloaded and data-triggered actions.

5.4.2 Shared infrastructure: Support for Futures

The Future send function communicates a result from a near-data task to the thread waiting on the future through a store-update instruction [86, 142]. The store-update instruction, which executes on an engine, sends a message containing the future pointer and value over the NoC to the waiting thread. The message instructs the thread to perform the store itself so that the result becomes immediately available without waiting for any additional coherence traffic.

5.4.3 Shared infrastructure: Support for data packing and padding

There are three main hardware mechanisms in support of Leviathan's data management: LLC object mapping, DRAM object compaction, and a memory controller cache.

LLC object mapping. As discussed in Section 5.3.3, it is important for objects larger than a cache line to map entirely to the same LLC bank. This is accomplished by modifying the input to the hash function that maps from cache-line address to LLC bank. With Leviathan, every cache line belonging to the same object provides the same hash-function input. This is accomplished by zeroing out LSBs of the address (e.g., for objects spanning two cache lines, zeroing out the single LSB is sufficient). Since Leviathan supports objects up to 256 B (i.e., 4 cache lines), two ($log(4) = 2$) bits are needed to indicate the number of LSBs that should be ignored for a particular object. Page table entries and L2 tags are appended with these two bits.

DRAM object compaction. Although we pad objects in the cache to improve locality, we do not want to waste DRAM. Prior NDCs required software to manually pad data, leading to an unattractive tradeoff between locality and memory capacity. However, since Leviathan has full control over padding, it can eliminate DRAM fragmentation with minor hardware support, invisibly to applications.

On an LLC miss or writeback, the LLC controller checks a small translation buffer to determine if the address needs translating. Figure 5.12 shows the breakdown for determining the DRAM address of an object based on its cache address. Since all objects of a given type are addressed contiguously both in the cache and DRAM (see Section 5.3.3), the translation is simply a matter of calculating offsets, which adds no latency on misses by running in parallel with LLC tag lookups. Each translation buffer entry contains the cache address base and bound, DRAM address base, and object size, totaling 25 B.

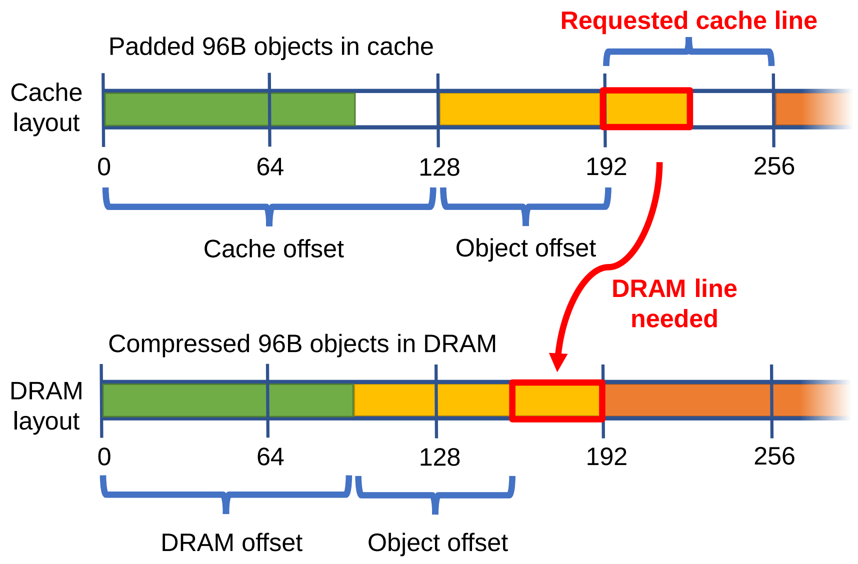


Figure 5.12: Leviathan pads objects in the cache but stores them compressed in DRAM. Simple computation translates between the cache and DRAM addresses for an object.

Table 5.2: Per-paradigm microarchitecture support across the system.

Paradigm	**Core**	**Cache**	**Engine**
Task offload	invoke instruction & buffer	N/A	DYNAMIC scheduling
Data-triggered	TLB bits	tag bits	actor buffer, vtable map
Streaming	pop instruction	N/A	push instruction, stream metadata

Memory controller cache. Because we store objects compacted in DRAM, lines fetched from DRAM will frequently contain portions of multiple objects. See, e.g., the second DRAM line in Figure 5.12. When an application iterates through objects sequentially, loading the second and third *cache* lines will both incur a memory access to the same *DRAM* line. To alleviate these excess DRAM accesses, we place a small FIFO cache (32 lines) at each memory controller. This small cache can reduce DRAM accesses by up to $2\times$.

5.4.4 Paradigm support: Task offload

Table 5.2 breaks down the microarchitecture additions necessary to support each paradigm, which are explained in detail here and throughout the following subsections.

invoke. A new ISA instruction corresponding to the invoke function is added to the cores. If the location is designated as LOCAL, then the core sends a message to the engine on the local tile; if it is REMOTE, then the core maps the object pointer to an LLC bank and sends a message to its engine.

If the location is DYNAMIC, the operation is more involved as the objective is to execute near the action's actor wherever it lies in the cache hierarchy. The instruction first probes the L1D and executes the action locally if the data is cached. Otherwise, the instruction assembles a packet containing the actor pointer, action function pointer, flags, and function arguments. The packet is sent to the task-offload scheduler on the tile's local engine, which checks whether the actor is cached in the L2 and then executes the action on the engine if so. If the actor is not cached in the L2 either, the scheduler forwards the packet to the LLC bank to which the actor maps. If the LLC's directory indicates that another L2 has the data with exclusive coherence permissions and the packet was assembled with the EXCLUSIVE flag, the packet is forwarded to that L2's engine. Else, the action executes at the LLC bank's local engine. The operation of invoking is similar to that in Livia [142] except that Livia only supports the DYNAMIC functionality.

Backpressure. Each core also contains a small "invoke buffer" to apply backpressure when cores offload tasks faster than they can execute. The invoke buffer is similar to a store buffer: task-offload requests first enter the invoke buffer and drain to engines. If a task-offload request arrives at an engine with no free space in the engine's task context buffer, the engine NACKs the invoke, spilling the task back to the core [142]. Otherwise, the engine ACKs the task-offload request, and it is removed from the invoke buffer. Finally, invoke instructions cannot commit in the core unless there is space in the invoke buffer. However, when offloaded tasks include a Future, the invoke buffer is skipped because waiting on futures generally provides sufficient backpressure.

Migrating data. In order to allow objects to settle at their natural location in the cache hierarchy, whenever a DYNAMIC task would be executed remotely, the scheduler will instead with small probability $(1/8)$ execute locally to pull the data up the hierarchy. This allows objects with high temporal locality to gradually move to the private caches.

5.4.5 Paradigm support: Data-triggered actions

Data-triggered actions are executed *when* and *where* data moves, and hence most of the changes are in the cache controllers. Leviathan adds minor core modifications to track which data is phantom and where it is registered; see Section 4.4.1 on täkō for details.

The data-triggered scheduler in the engine manages a buffer containing the actors on which actions are currently executing, since the actors are not accessible by any other threads during that time. The scheduler also contains a small cache that maps address ranges to their associated actions, i.e., the Morph's vtable.

Cache modifications. GETs are augmented with two bits indicating if a cache miss should trigger the data's constructor at the L2 or LLC, respectively. The L2 and LLC tags are augmented with one extra bit to indicate whether the destructor should be triggered on eviction.

With this extra information, the cache controller triggers actions when data is inserted or evicted. For small objects, the scheduler executes the actions on all the objects within the line in parallel. For large objects, only one action is triggered, which inserts (or evicts) multiple lines at once. Construction inserts multiple lines to fit the entire object, and destruction evicts all lines corresponding to the object.

5.4.6 Paradigm support: Streams

As discussed in Section 5.3.6, whereas the stream's data is stored in a circular buffer in shared memory, the core reads from the stream by accessing a contiguous phantom array which maps to the buffer through data-triggered constructors. Managing the stream and buffer involves support at both the core (consumer) and engine (producer).

Core modifications. Streams require a new ISA instruction, which corresponds to a pop function called by next. It increments a register containing the head pointer for the phantom stream. When the head pointer increments to a new cache line, it sends a request (a new message type) to the local engine (where the stream is generated) to bump the stream's head pointer forward. The request also invalidates the old stream head at the L2 since it will not be used anymore.

Engine stream scheduler. For each active stream, the engine needs to track the buffer size and phantom head/tail pointers. The reason for tracking the tail pointer is to stall the core if it loads data after the tail (i.e., stream entries not yet pushed). The head pointer is used to NACK prefetches and throw exceptions on loads to data before the head (i.e., stream entries already popped). When the core sends a pop message, the head is incremented, and if an NDC action is blocked on push it is unblocked to generate the next element of the stream.

Deadlock prevention. Out-of-order cores need to be careful to avoid deadlock with streams. Speculatively reordered loads could reserve all L1 MSHRs, without any load able to proceed because they all are past the end of the currently generated stream. This condition is exceedingly rare, but possible in principle. To prevent this, systems could NACK speculative loads to addresses past the end of the current stream buffer, and re-execute them on commit, when they must point to the current stream head. A NACK to a load at the head of the commit queue triggers an exception.

5.4.7 Handling very large objects

As previously mentioned, Leviathan can only support objects up to a microarchitecturally defined size, as it is impractical to support individual objects of many KBs, MBs, or GBs with lightweight hardware extensions. It also simply impossible to preserve the benefits of near-data computing as object sizes scale very large. However, to present a simple, unified programming interface, Leviathan offers a functionally correct fallback implementation of each NDC paradigm for arbitrary object sizes.

Task offloading works like normal, except the allocator just resorts to malloc, so objects are spread across LLC banks and padded in DRAM. For data-triggered actions, all constructors are triggered on the core when a page of objects is paged in, and destructors are triggered on the core when paged out. For streams, the producer and consumer are spawned as conventional threads with a message-passing queue between them.

Table 5.3: Hardware overhead (state per L3 bank).

L3 tags	8K lines × 3 bits = 3 KB
L3 translation buffer	8 entries × 25 B = 200 B
Engine L1d, TLB, rTLB	8 KB + 2 KB + 2 KB = 12 KB
Data-triggered buffer	16 objects × 256 B = 4 KB
Dataflow fabric [204]	13.6 KB
Total per L3 bank	32.8 KB / 512 KB = 6.4%

Table 5.4: System parameters in our experimental evaluation.

Cores	16 cores, x86-64 ISA, 2.4 GHz, OOO Skylake uarch [5], 4-entry invoke buffer
Engines	16 engines, dataflow fabric, 15 int FUs (1-cycle latency), 10 mem FUs, 8 KB L1d, 256-entry rTLB, 32 thread contexts
L1	32 KB, 8-way set-assoc, split data and instruction caches
L2	128 KB, 8-way set-assoc, 2-cycle tag, 4-cycle data array, t̃r̃îp repl. [204], strided prefetcher
LLC	8 MB (512 KB per tile), 16-way set-assoc, 3-cycle tag, 5-cycle data array, inclusive, t̃r̃îp repl. [204]
NoC	mesh, 128-bit flits and links, 2/1-cycle router/link delay
Memory	4 controllers, 100-cycle latency, 11.8 GB/s per controller, 32 entry FIFO cache

5.4.8 Putting it all together

In total, Leviathan's hardware adds relatively small area overheads to a baseline multicore. The total per-tile storage cost, when modeling a dataflow fabric with parameters from täkō (Section 4.4.3), totals 32.8 KB, or 6.4% compared to the data array of an L3 bank (Table 5.3). This is comparable to recent fabrics [166, 193, 204, 238].

5.5 Experimental Methodology

Simulation framework. Our execution-driven simulator shares infrastructure with SwarmSim [102, 251], but supports cycle-level timing throughout the memory hierarchy, implements Leviathan's interface, and models near-data engines. For simulation convenience, near-data instructions are mapped onto the engines' dataflow fabric when they first execute, but in practice one could compile code statically [217, 238].

System parameters. Except where specified otherwise, our system parameters are given in Table 5.4. We model a tiled multicore system with 16 cores connected in a mesh on-chip network. Each tile contains a conventional out-of-order core (modeled after Intel Skylake), one bank of the shared LLC, and a Leviathan engine (to ease implementation, our simulator models engines at both the L2 and LLC bank). We take energy parameters from [76, 193, 222]. Section 5.7 varies these parameters and shows that Leviathan is effective across a variety of system configurations.

Leviathan engines are modeled with a 5 × 5 dataflow fabric (15 integer PEs and 10 memory PEs) and 1-cycle PE latency. When comparing against täkō, we do not evaluate the SIMD PEs from its evaluation in order to compare both systems with the same engine compute resources. We also evaluate an *idealized engine* with unlimited, 0-cycle latency

```
class Decompressor extends Leviathan::Morph<Pixel>:
  int16* bases[3]
  int8* deltas[3]

# actor with data (colors) and an action (constructor)
class Pixel: # Leviathan is agnostic to object size
  int16 colors[3] # 3 ints does not divide a cache line

  Pixel(Decompressor* decomp): # action: constructor
    idx = decomp->getOffset(this)
    bases = decomp->bases
    deltas = decomp->deltas

    for i in range(numColors):
      base = bases[i][idx >> 3] # 1 base per 8 pixels
      delta = deltas[i][idx + i]
      mantissa = delta & 0b1111
      exponent = delta >> 4
      colors[i] = base + (mantissa << exponent)
```

Figure 5.13: Leviathan uses data-triggered actions to decompress objects of different sizes when the data is loaded by the core.

and energy-free PEs; i.e., action latency is only affected by memory latency and data dependencies.

5.6 Evaluation — Case Studies

Leviathan is a polymorphic cache hierarchy that unifies prior NDC paradigms without exposing microarchitectural details to the programmer. We now evaluate three additional applications that benefit from Leviathan to demonstrate:

- Leviathan provides strong performance and energy benefits across NDC paradigms.
- Leviathan's actor-based interface is intuitive to program and provides benefits across object sizes.
- Leviathan scales well across system and data sizes (Section 5.7) and is close to an idealized design.

5.6.1 In-cache data transformation

Prior work on hardware compression has shown significant memory and cache savings [12, 62, 178, 179, 200, 226]. But prior designs fix the (de)compression mechanism in hardware, so there is no flexibility of scheme or data sizes. In this study, we analyze Leviathan's ability to transform data by using data-triggered actions to decompress objects of *different sizes* as they are brought into a core's private cache.

Decompression with Leviathan. Figure 5.13 shows the code for a data-triggered NDC application that uses a Leviathan Morph to decompress data stored in a lossy, compressed format in memory as a base plus offset value, similar to [179]. The application registers the morph at the L2 (not shown). The actor's constructor is then triggered for each object that is accessed by the core.

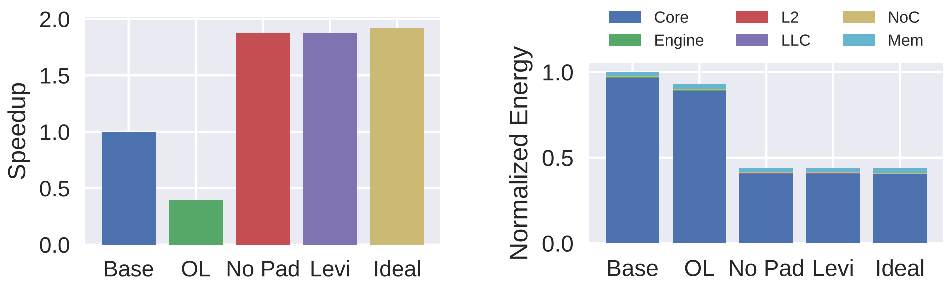


Figure 5.14: Results when decompressing 8B objects. Leviathan improves performance by 1.9× and reduces energy by 56%.

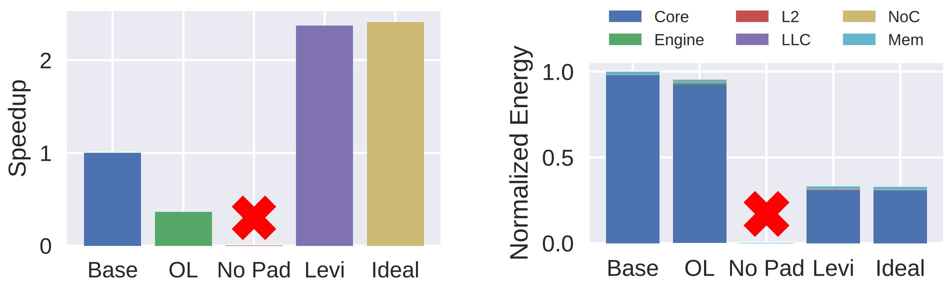


Figure 5.15: Results when decompressing 6B objects. Leviathan improves performance by 2.4× and reduces energy by 67%.

To decompress data of different types, the programmer simply implements the constructor to perform the type's decompression algorithm. Prior work requires decompressed data to evenly fit into cache lines (e.g., täkō decompresses one cache line at a time in Section 4.2.2), restricting the programmer to a limited subset of data types and requiring careful alignment and padding. By contrast, Leviathan simply asks the programmer to provide the data type of interest (see line 6). Figure 5.13 shows decompression for a 6 B Pixel, which does not evenly divide a cache line.

Evaluation. Leviathan improves performance, saves energy, and reduces redundant work *across different object sizes*. We evaluate an application which computes an average over an array of 16 K decompressed objects (either a single 8 B integer, or the 6 B Pixel in Figure 5.13). The array is indexed using a Zipfian distribution [38] of 32 K accesses.

We evaluate a baseline software implementation that decompresses on every access, an NDC version that uses task offload (OL) to decompress at the local engine, and Leviathan with and without padding, which accesses decompressed data through the Morph in Figure 5.13. Figure 5.14 shows results for decompressing 8 B objects, and Figure 5.15 for 6 B objects.

Observation #1. Task-offload actually *worsens* performance (by up to 2.8×). Task-offload hurts performance because decompressing at the L2 loses the locality in the L1s, without reducing the number of compressions.

Observation #2. Data-triggered actions *do not work* without padding. 8 B evenly divides a cache line, so decompression works, but 6 B does not, so cache lines contain partial objects. Constructors cannot initialize a portion of an object.

Leviathan addresses both issues and works for all object sizes while significantly outperforming the baseline.

```
 1  # actor with data (value) and an action (Lookup)
 2  class Node:
 3    Node* next, prev
 4    int64 value[N]                 # large objects are fine!
 5    # int64 padding[LINE_SIZE-2-N] # no padding needed!
 6
 7    Node* Lookup(key, idx): # action: runs near-data implicitly
 8      if value[idx] == key:
 9        return this
10
11      if next == nullptr:
12        return nullptr
13
14      return invoke next->Lookup(key, idx)
```

Figure 5.16: Leviathan uses task offloading for linked-list searches with different node sizes.

Leviathan improves performance by 1.9×/2.4× for 8 B/6 B objects, and Leviathan reduces energy by 56%/67%. Moreover, Leviathan comes within 4.2% speedup and 0.2% energy of ideal.

Discussion. This study demonstrates *(i)* polymorphic hierarchies need to support all paradigms because not all paradigms are right for an application; *(ii)* data-triggered actions *require* padding; and *(iii)* Leviathan provides good performance while easing programming effort by padding automatically.

5.6.2 Pointer chasing

Linked lists are a popular data structure due to fast insertion and removal of elements, but they are notoriously slow due to their sequential, unpredictable access pattern. Prior NDC work offloads lookups into the memory hierarchy, avoiding constant round-trips between core and cache.

Linked-list pointer chasing with Leviathan. Figure 5.16 shows the code for an application that uses task offloading for linked-list pointer chasing with Leviathan. Lines 7-14 implement an offloaded task that compares a single linked-list node against a key. If the node contains the key, a future is notified that the key was found (by returning the node's address). Otherwise, if the node is not at the end of the list, the task invokes another task to check the next node.

The major benefit of Leviathan for this example is that the application can use any node type desired without concern for microarchitecture. It is important for each node to reside entirely within a single tile for a task to maintain the locality benefits of NDC (see Figure 5.7). As a result, prior work required the application to manually pad and align linked-list nodes to cache lines, an unnecessary exposure of microarchitecture to programmers. Instead, with Leviathan, the application simply allocates each linked-list node using Leviathan's allocator without concern for object size or alignment. Further, previous NDCs cannot provide spatial locality for nodes larger than a cache line, whereas Leviathan's LLC mapping mechanism easily maps entire objects to the same cache bank (Section 5.4.3).

Evaluation. Linked-list pointer chasing with Leviathan significantly reduces runtime and energy *across different object*

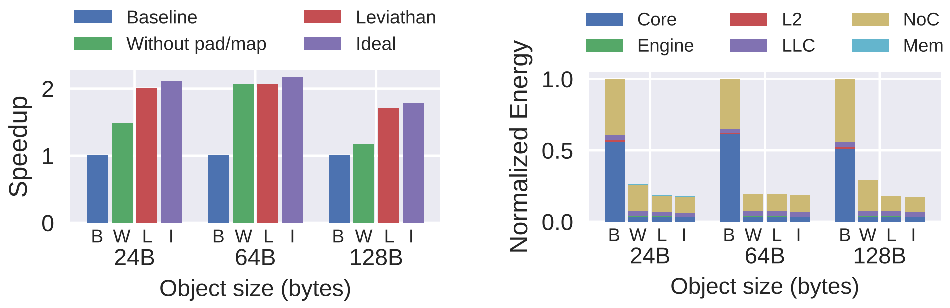


Figure 5.17: Results when performing linked-list search across different object sizes with a uniform distribution over keys. Leviathan performs well across object sizes, improving performance up to 2.1× and reducing energy by up to 82%.

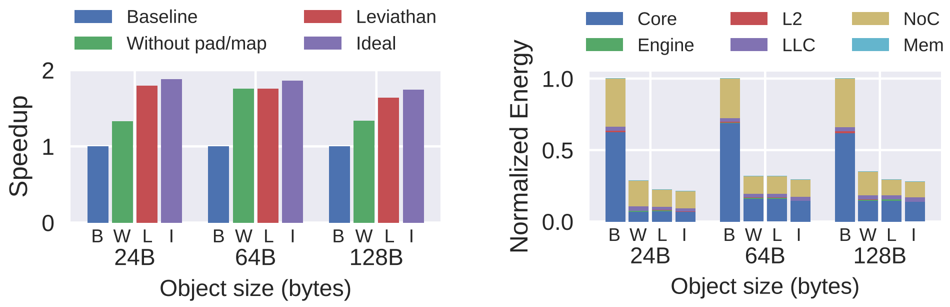


Figure 5.18: Results when performing linked-list search across different object sizes with a zipf distribution over keys. Leviathan performs well across object sizes, improving performance up to 1.8× and reducing energy by up to 77%.

sizes, even when larger than a cache line.

We evaluate an application with 16 threads each performing 1 K lookups across different object sizes (24 B, 64 B, and 128 B; see line 3 in Figure 5.16). We initialize an array of 32-Node linked lists whose (padded) size totals 4 MB. To perform a lookup, we generate a key from a uniform or Zipfian [38] distribution and scan the corresponding list. We evaluate a baseline software implementation and Leviathan, with and without Leviathan's padding and LLC object mapping support.

Leviathan performs similarly across all object sizes and both distributions (Figure 5.17 and Figure 5.18), achieving up to 2.1× speedup and 82% energy savings. A majority of the benefits come from reducing NoC traffic by offloading a chain of tasks within the LLC, instead of constant round-trips to the caches to fetch each Node. The linked lists fit in the LLC, but not L1d or L2, so almost all lookups in the baseline require pulling data from the LLC down into the private caches.

The benefits of Leviathan's padding and large-object LLC mapping are demonstrated by the 24 B and 128 B results, respectively. Without padding, 24 B performance is reduced to 1.3×, and without mapping 128 B performance is reduced to 1.2×.

Another benefit of Leviathan is compact storage in DRAM for nodes padded in the cache. Specifically, padding the 24 B nodes to 32 B would cause 25% memory fragmentation in prior work. Leviathan performs padding in-cache and

```
struct Edge { int src, int dst } # object can be anything

# actor with an action (genStream)
class LeviathanHATS extends Leviathan::Stream<Edge>:
  Stack bdfs = {Vertex* vec, int top}

  void genStream(): # action: fill stream
    while True:
      if bdfs.top == 0:
        root = G.getNextRootVertex()
        if root == INVALID: return INVALID
        bdfs.vec[++bdfs.top] = root
        active[root++] = false

      dst = bdfs.vec[bdfs.top]
      while dst.nextNeighbor < inDegree:
        src = dst.neighbors[dst.nextNeighbor++]
        push(Edge(src, dst)) # stalls when full

        if bdfs.top < depth and !active[src]:
          bdfs.vec[++bdfs.top] = src
          active[src] = false

      --bdfs.top

# main thread reads off stream
for range(G.numEdges):
  # get future for next edge and process when ready
  Future<Edge> future = stream.next()
  processEdge(future.wait())
```

Figure 5.19: Leviathan implementation of HATS.

compacts objects in DRAM, getting the best of both worlds.

Discussion. Leviathan provides many qualitative and quantitative benefits for this pointer-chasing application. Qualitatively, programming is much simpler to due Leviathan's reactive-programming interface where offloaded tasks can execute on any actor allocated using Leviathan's allocator. Quantitatively, Leviathan achieves significant speedup, energy savings, and memory savings across a range of different node sizes, even when nodes are larger than a cache line. Leviathan thus makes NDC both easier to use and more effective.

5.6.3 Decoupled graph traversal via streaming

Lastly, we demonstrate streaming in Leviathan, which, behind the scenes, leverages the task-offload and data-triggered paradigms (Section 5.3.6). We demonstrate streaming on HATS [159], a recent optimization for locality in graphs. HATS observed that, without expensive pre-processing, it is inefficient to process the edges in the order they are laid out in memory. Many graphs exhibit strong community structure [23, 131], so it is much better to process graphs one community at a time. A bounded, depth-first search (BDFS) is a simple traversal order that significantly improves locality. The challenge is that BDFS executes inefficiently on cores due to unpredictable control flow and

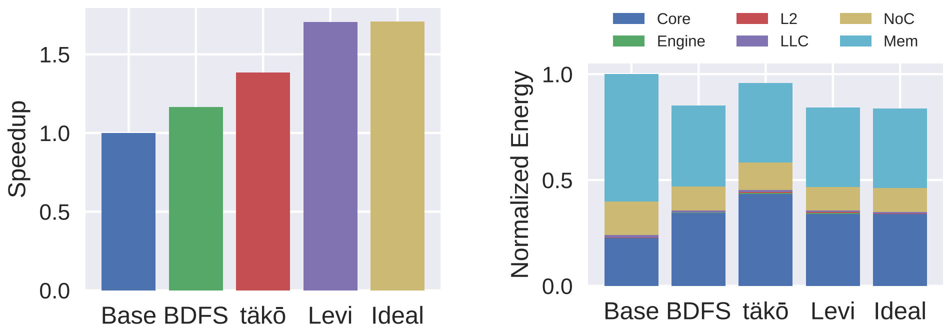


Figure 5.20: HATS results for one iteration of PageRank on uk-2002 graph [54]. Leviathan improves performance by 1.7× and reduces energy by 16% vs. the software baseline.

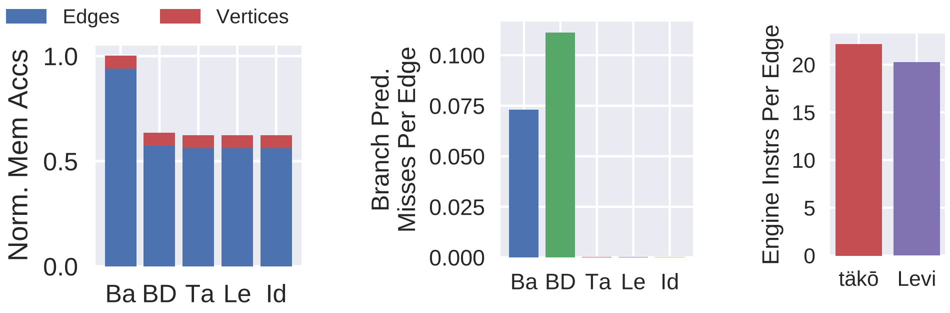


Figure 5.21: HATS performance breakdown. Left: DRAM accesses split by PageRank phase. Middle: core branch mispredictions per graph edge processed. Right: avg engine instrs per edge.

coupling of the graph traversal with vertex processing. However, BDFS is infeasible for many prior streaming NDC designs because it cannot be easily reduced to a combination of simple affine or indirect patterns.

BDFS streaming with Leviathan. Figure 5.19 shows the code for streaming BDFS on Leviathan. The application registers a Stream with an Edge type, without worrying about padding, alignment, or size of the Edge. genStream is populated with the BDFS algorithm, which continually generates Edges and pushes them onto the stream. The main thread running on the core processes edges with next.

Evaluation. Leviathan's streaming implementation of BDFS provides significant performance and energy benefits. We compare baseline PageRank, software BDFS, BDFS in täkō, and Leviathan. täkō only supports data-triggered actions, and implements HATS by having constructors on cache misses trigger BDFS traversal (instead of stream pushing). The täkō version of BDFS is more complex and has unintuitive corner cases; e.g., it cannot guarantee that the stream is generated sequentially, since it depends on the order of misses generated by the core.

Figure 5.20 presents speedup and energy results for one iteration of PageRank across the modes. Whereas software BDFS and täkō achieve modest speedups of 1.2× and 1.4×, Leviathan achieves 1.7× speedup (nearly identical to ideal). Additionally, Leviathan reduces energy by 16%.

This speedup is due to *(i)* better cache locality; *(ii)* regularizing control flow on the core; *(iii)* an efficient push-based streaming interface; and *(iv)* decoupling of stream producer and consumer. Figure 5.21 quantifies the first three points.

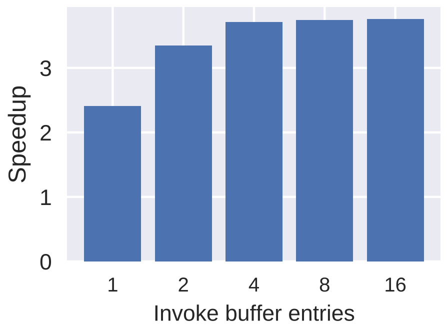


Figure 5.22: Sensitivity to invoke buffer with PHI.

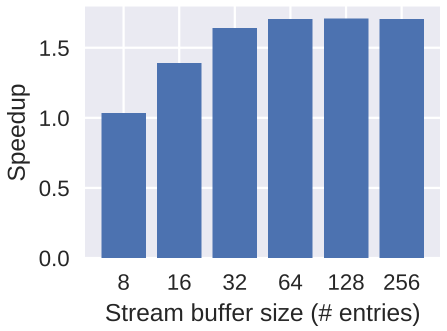


Figure 5.23: Sensitivity to stream buffer with HATS.

All versions incur the same number of memory accesses during the vertex phase, but the versions that execute the BDFS traversal reduce total accesses by 40%. täkō and Leviathan both eliminate branch mispredictions by turning the complex BDFS traversal into a simple loop over a sequential array.

However, täkō's disadvantages begin when looking at average engine instructions per edge generated. Since täkō's implementation triggers a new action to resume the BDFS traversal every eight edges (one cache line), it adds many additional instructions to "reinitialize" the BDFS stack each time. In contrast, Leviathan's stream producer is a continually running action, reducing average instructions per edge. Leviathan also lets the stream run far ahead, whereas täkō streams are implicitly triggered by loads and thus dependent on the consumer.

Discussion. Leviathan's decoupled producer-consumer streaming interface is simple to program and enables significant performance gains. A major benefit of Leviathan over prior streaming accelerators is that Leviathan's stream producer is not limited to simple access patterns (e.g., affine or indirect) [169, 233–236, 247]. Leviathan allows the application to implement any desired pattern in software. Moreover, Leviathan can implement streams more easily and efficiently than NDCs that support only a single paradigm, like täkō.

5.7 Evaluation — Sensitivity Studies

Invoke buffer. PHI is the most sensitive to the invoke buffer because it offloads tasks rapidly and does not wait for them to complete. Figure 5.22 evaluates Leviathan across different buffer sizes. With one or two entries, Leviathan slows due to queueing effects causing backpressure, but performance plateaus after four entries.

Stream buffer. Figure 5.23 evaluates HATS' performance across stream-buffer sizes. Performance plateaus at 64 entries. Note that the stream buffer resides in memory, not a separate hardware structure, so its overhead is negligible.

Input size. Figure 5.24 evaluates linked list with a uniform distribution across total linked-list sizes. As long as the data fits in the LLC, Leviathan performs well. Once the data is larger than the LLC, Leviathan's performance drops as the NoC savings are swamped by DRAM latency. Leviathan outperforms the baseline, except with 24 B objects when

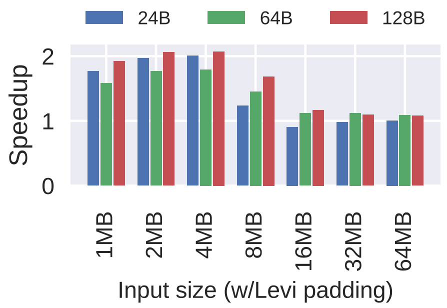


Figure 5.24: Sensitivity to input size with linked-list uniform.

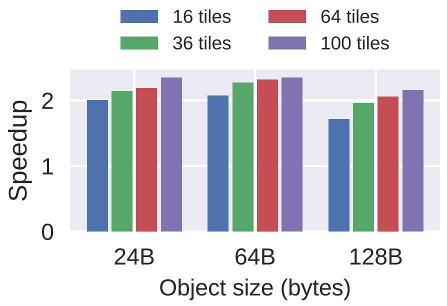


Figure 5.25: Sensitivity to number of tiles with linked-list uniform.

Leviathan's padding causes the working set to exceed LLC capacity.

System size. Finally, Figure 5.25 evaluates linked-list on a uniform distribution across system sizes. Leviathan performs better with larger systems due to the increased NoC savings.

5.8 Summary

Near-data computing is essential to tackle the rising cost of data movement. Prior work has proven that NDC yields large gains in performance and energy efficiency. Unfortunately, prior designs are impractical because they have limited applicability and each propose their own difficult-to-use programming model. Leviathan overcomes these challenges by unifying prior NDC techniques in a single, polymorphic cache hierarchy with a simple, reactive-programming interface.

Chapter 6

Future Work

Programmable data movement is an exciting and broad field with plenty of room for future work. There is ample opportunity to explore both additional high-level programmable features and low-level implementation details for full-system integration.

Unified system for data placement and NDC. The output of this thesis is two separate mechanisms: one that moves data closer to compute (Jumanji) and one that moves compute closer to data (täkō and Leviathan). An ideal data-centric processor would support both mechanisms without requiring separate hardware for each. A possible approach is using data-triggered NDC to redirect accesses to specific LLC banks and memory locations through software address translation (i.e., cache miss at L2 loads from a nearby LLC bank, and cache miss at LLC loads from the real data in memory). One requirement for this approach to work is enforcing coherence between phantom data and real data, another future work topic. Whether this design is feasible and imposes acceptable overheads with software address translation requires further analysis.

Another possibility is integrating Jumanji's data placement with Leviathan's large-object mapping. Leviathan only uses the mapping mechanism to place entire objects in a single bank, not necessarily close to a specific core, but it could be extended for NUCA-aware data placement too.

Transformed-data coherence. A shortcoming of our software implementation of data-triggered NDC (introduced in täkō) is that phantom data is not coherent with its real-data source. For example, in the decompression application evaluated by both täkō and Leviathan, the decompressed data is not coherent with the compressed data. After the data is decompressed due to a cache miss, the compressed data can be modified while the decompressed data remains unaffected. Ideally, the system would maintain coherence between phantom data and the real data used as input to generate the phantom data (i.e., invalidate the phantom data when the real data is modified, and vice versa).

Maintaining this phantom data to real data coherence is not trivial. One phantom line can be generated from many real lines, and one real line can help generate many phantom lines. This many-to-many relationship could be very difficult to track in hardware and too slow to maintain in software. Further, an application might not want this additional coherence mechanism due to excess invalidations deemed unnecessary by the programmer. How to properly and efficiently implement transformed-data coherence remains an open problem.

NDC near memory. This thesis only addressed data movement within the cache hierarchy, but there is significant opportunity to optimize data movement beyond the cache near main memory. Near-memory NDC has garnered extra popularity recently with 3D-stacked memory, where a logic layer has high-bandwidth access to its local memory layers [10, 33, 34, 91, 184]. These systems thrive with workloads that exhibit minimal data locality, so involving the caches provides no benefit.

Many of the techniques we presented can be extended to near-memory NDC. For example, offloaded tasks can dynamically execute in the logic layer near its data in memory if the data is not found in any cache. We could also take an approach similar to Livia [142], which places an engine at each on-chip memory controller to execute near a memory bank while confining modifications to the processor. Further, we could extend data-placement techniques from Jumanji to allocate data in LLC banks near specific memory banks and incorporate ideas from Jenga [222] on allocating DRAM too.

Additional NDC features. Although Leviathan attempts to support as many prior NDC designs as possible, it is not yet fully comprehensive. There are additional features, both inside and outside of the paradigms we discussed in Section 2.7.2, that require additional support to implement. We identify and briefly discuss two new opportunities here.

Our data-triggered NDC design does not currently support triggering actions in response to a common cache operation—cache hits. Whereas täkō and Leviathan trigger computation in response to cache insertions and evictions, they lack support for applications that would also benefit from responding to cache hits. A notable example is programmable prefetchers, which want to observe every access to a specific cache level, whether hit or miss. While prefetchers can be partially supported using only insertion-based data-triggered actions, the prefetcher only knows about accesses to the next cache level, not the current one.

Another possible NDC feature is software-managed replacement policies. Although we stated in Chapter 1 that state-of-the-art replacement policies provide limited benefits, we were addressing application-agnostic policies (e.g., Hawkeye [100]) that try to optimize performance across all types of workloads. Instead, prior work has also proposed application-specific replacement policies that target a single workload pattern to provide near-ideal performance (e.g., P-OPT [19] for graph applications and TCOR [106] for graphics rendering). These policies provide near-ideal performance because they can actually determine the order of future caches accesses by observing data structures used by the application.

In the theme of our work, we could move control of replacement decisions from hardware to software. As a result, each application could customize its own replacement policy when the access pattern can be determined. Triggering replacement “actions” in style of Leviathan (Section 5.1) fits well with our NDC model, where the actions could access the application’s data structures through shared memory. But there remain many open questions (e.g., How to store per-entry replacement metadata? How to communicate this data to the actions? How to prevent actions from recursively triggering additional replacements?).

Engine microarchitecture. Detailed design exploration of dataflow fabrics, along with their communication to other engine components, is warranted for the programmable compute logic used in täkō’s and Leviathan’s near-data engines. We evaluated a high-level dataflow model that abstracts away many intricate details because the microarchitecture of the engine’s compute logic was not the main focus of those systems. But Leviathan is now in a sufficient state to dive deeper into the hardware mechanisms supporting its programming interface.

Many prior dataflow architectures provide a foundation to fuel our exploration. Early dataflow designs focused on replacing von Neumann architectures as the central processing unit (e.g., WaveScalar [217], TTDA [14], Raw [229], and Monsoon [174]). These works provide valuable, fundamental concepts for dataflow architectures. However, they were designed two to three decades ago when computer systems faced different bottlenecks, and they also focused on the ability to run everything (operating system and applications) whereas we only require the ability to execute relatively small portions of applications.

Co-processor dataflow designs are more relevant to our needs (e.g., REVEL [238], UE-CGRA [221], SNAFU [68], and RipTide [69]). But they generally target very-low-power objectives or specific algorithm paradigms. We are more concerned about providing high performance for multiple co-running threads with potentially variable algorithm designs.

Consequently, one should explore the design space between these two groups of dataflow designs. Our engines need to provide good performance and have enough resources to support multiple threads, yet they do not need to execute entire applications. This area warrants further research to develop an engine microarchitecture suitable for NDC.

Compiler support. There are two areas in which compiler support is needed for our programmable NDC. The first area is compiling from high-level languages to a dataflow binary for code executing on the near-data engines. As mentioned earlier, both täkō and Leviathan evaluated a high-level dataflow model. They dynamically mapped x86 micro-operations to processing elements (PEs) of the dataflow fabric, but this is not a fully realistic design. In practice, the code for near-data actions needs to be compiled to the dataflow’s ISA and then mapped to PEs either statically or dynamically. There is prior work on dataflow compilation for full programs [3, 14, 217, 229] and more recently for tiny kernels on very-low-power dataflow fabrics [68, 69], but there is still missing work for compiling medium-sized kernels for high-performance dataflow fabrics.

The other area warranting compiler support is converting traditional applications to NDC applications without programmer involvement. täkō and Leviathan require programmers to use the provided software libraries, but often a compiler could automatically convert code itself. One possible paradigm to incorporate with compilation is streaming due to the use of common access patterns (e.g., indirect or affine), which prior work has already explored for streaming accelerators [234, 236]. Compilers could also detect operations to offload due to low expected reuse of data that should not be brought into private caches. Automatically converting code suitable to data-triggered NDC is likely a more challenging, but also interesting, endeavor.

Engine cache coherence. Near-data engines in täkō and Leviathan contain their own L1d caches which are intended to use clustered coherence within each tile, but both evaluations actually simulated separate engines for a tile's L2 and LLC bank (i.e., one engine L1d is a child of the L2, and the other L1d is a child of the LLC). But with only one engine per tile, it is undesirable for the engine L1d to either always be a child of the L2 or always be a child of the LLC. The L2 issues are the engine polluting the core's private cache and adding L2 access latency before LLC accesses, and the LLC issues are adding additional directory state and increasing the shared-memory communication latency between a core and its local engine.

Accordingly, the optimal coherence protocol should be a hybrid policy that allows local communication between the core and engine without involving the LLC, but also avoids cache pollution without increasing LLC directory state. These objectives exhibit similar qualities to clustered coherence [75, 130, 151] but also introduce new challenges such as the cache pollution issue. We have started developing a new coherence protocol [37], but a full implementation is still in the works.

Chapter 7

Conclusion

This thesis has presented techniques to enable practical specialization of data movement within the cache hierarchy. We have shown that general-purpose architectures can provide application-specific data-movement optimizations when combined with a flexible programming interface. Concretely, we have presented the following contributions:

- **Jumanji** is a dynamic NUCA design for software-managed data placement. We demonstrated that data placement can be easily tailored for a variety of application objectives, even at the same time, when software is given full control. By reusing the same hardware proposed by prior throughput-oriented D-NUCAs, Jumanji's software runtime optimizes data placement for a variety of datacenter objectives — tail latency, security, and throughput. Consequently, Jumanji provides similar throughput to prior work while ensuring quality-of-service for latency-critical applications and guaranteeing stronger security than prior work.

- **täkō** is a polymorphic cache hierarchy that expands the hardware-software interface by enabling software to observe and manipulate data as it traverses the cache hierarchy. The key mechanism is triggering software callbacks in response to cache misses, evictions, and writebacks. This feedback from hardware to software enables many application-specific optimizations that previously each required custom hardware. By opening the cache hierarchy to software, täkō demonstrates that specialized data movement is achievable on general-purpose hardware.

- **Leviathan** builds on täkō to realize a fully-programmable NDC system. Leviathan supports many NDC designs on a single architecture while hiding hardware details from the programmer. Its actor-based programming model encompasses multiple NDC paradigms and handles data management behind the scenes, providing programmers with a familiar and easy-to-user interface. The resulting system demonstrates how a general-purpose, polymorphic cache hierarchy can provide software with the means to implement many NDC optimizations.

These contributions compose a polymorphic cache hierarchy that transfers control of data movement from hardware to software. By enabling software to move both data closer to compute and compute closer to data, these architectures remove the need for application-specific custom hardware, paving the path for general-purpose optimization of data movement.

Bibliography

[1] S. V. Adve and K. Gharachorloo, “Shared memory consistency models: A tutorial,” *IEEE Computer*, 1996.

[2] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, “Compute caches,” in *Proc. of the 23rd IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-23)*, 2017.

[3] A. Agarwal, S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, D. Srikrishna, and M. Taylor, “The raw compiler project,” in *Proceedings of the Second SUIF Compiler Workshop*, 1997.

[4] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, “The MIT Alewife machine: architecture and performance,” *Proc. of the 22nd annual Intl. Symp. on Computer Architecture*, 1995.

[5] Agner Fog, “The microarchitecture of Intel, AMD and VIA CPUs,” https://www.agner.org/optimize/microarchitecture.pdf, 2020.

[6] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, “Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture,” in *Proc. of the 42nd annual Intl. Symp. on Computer Architecture (Proc. ISCA-42)*, 2015.

[7] S. Ainsworth and T. M. Jones, “Graph prefetching using data structure knowledge,” in *Proc. of the Intl. Conf. on Supercomputing (Proc. ICS’16)*, 2016.

[8] S. Ainsworth and T. M. Jones, “An event-triggered programmable prefetcher for irregular workloads,” in *Proc. of the 23rd intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXIII)*, 2018.

[9] K. Akdemir, M. Dixon, W. Feghali, P. Fay, V. Gopal, J. Guilford, E. Ozturk, G. Wolrich, and R. Zohar, “Breakthrough aes performance with intel aes new instructions,” *Intel White paper, June*, 2010.

[10] B. Akin, F. Franchetti, and J. C. Hoe, “Data reorganization in memory using 3d-stacked dram,” in *Proc. of the 42nd annual Intl. Symp. on Computer Architecture (Proc. ISCA-42)*, 2015.

[11] I. Akturk and U. R. Karpuzcu, "Amnesiac: Amnesic automatic computer," in *Proc. of the 22nd intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXII)*, 2017.

[12] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *Proc. of the 31st annual Intl. Symp. on Computer Architecture (Proc. ISCA-31)*, 2004.

[13] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[14] Arvind and R. Nikhil, "Executing a program on the mit tagged-token dataflow architecture," *IEEE Transactions on Computers*, 1990.

[15] A. Asgharzadeh, J. M. Cebrian, A. Perais, S. Kaxiras, and A. Ros, "Free atomics: Hardware atomic operations without fences," in *Proc. of the 49th annual Intl. Symp. on Computer Architecture (Proc. ISCA-49)*, 2022.

[16] K. Asifuzzaman, N. R. Miniskar, A. R. Young, F. Liu, and J. S. Vetter, "A survey on processing-in-memory techniques: Advances and challenges," *Memories-Materials, Devices, Circuits and Systems*, 2022.

[17] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter, "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches," in *Proc. of the 15th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-15)*, 2009.

[18] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, "A survey on reactive programming," *ACM Computing Surveys (CSUR)*, 2013.

[19] V. Balaji, N. Crago, A. Jaleel, and B. Lucia, "P-opt: Practical optimal cache replacement for graph analytics," in *Proc. of the 27th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-27)*, 2021.

[20] S. Bandara and M. A. Kinsy, "Adaptive caches as a defense mechanism against cache side-channel attacks," *Journal of Cryptographic Engineering*, 2020.

[21] L. Barroso and U. Holzle, "The case for energy-proportional computing," *IEEE Computer*, vol. 40, no. 12, 2007.

[22] S. Baskaran, M. T. Kandemir, and J. Sampson, "An architecture interface and offload model for low-overhead, near-data, distributed accelerators," in *Proc. of the 55th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-55)*, 2022.

[23] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an Ivy Bridge server," in *Proc. of the IEEE Intl. Symp. on Workload Characterization (Proc. IISWC)*, 2015.

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

[25] S. Beamer, K. Asanović, and D. Patterson, "Reducing pagerank communication via propagation blocking," in *Proc. of the 31st IEEE Intl. Parallel and Distributed Processing Symp. (Proc. IPDPS)*, 2017.

[26] B. Beckmann, M. Marty, and D. Wood, "ASR: Adaptive selective replication for CMP caches," in *Proc. of the 39th intl. symp. on Microarchitecture*, 2006.

[27] B. Beckmann and D. Wood, "Managing wire delay in large chip-multiprocessor caches," in *Proc. of the 37th intl. symp. on Microarchitecture*, 2004.

[28] N. Beckmann and D. Sanchez, "Jigsaw: Scalable Software-Defined Caches," in *Proc. of the 22nd intl. conf. on Parallel Architectures and Compilation Techniques*, 2013.

[29] N. Beckmann and D. Sanchez, "Talus: A Simple Way to Remove Cliffs in Cache Performance," in *Proc. of the 21st IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-21)*, 2015.

[30] N. Beckmann, P.-A. Tsai, and D. Sanchez, "Scaling distributed cache hierarchies through computation and data co-scheduling," in *Proc. of the 21st IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-21)*, 2015.

[31] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: exploiting speculative execution through port contention," *arXiv preprint arXiv:1903.01843*, 2019.

[32] A. Biswas, "Sapphire rapids," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021.

[33] B. Black, "Die Stacking is Happening!" in *MICRO-46 Keynote*, 2013.

[34] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso *et al.*, "Die stacking (3d) microarchitecture," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[35] G. E. Blelloch, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun, "The parallel persistent memory model," in *Proc. of the 30th ACM Symp. on Parallelism in Algorithms and Architectures (Proc. SPAA)*, 2018.

[36] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Proc. of the 23rd intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXIII)*, 2018.

[37] J. Brana, B. C. Schwedock, Y. A. Manerkar, and N. Beckmann, "Kobold: Simplified cache coherence for cache-attached accelerators," *IEEE Computer Architecture Letters*, 2023.

[38] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *IEEE INFOCOM*, 1999, pp. 126–134.

[39] A. Buluç and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *Proc. of the 22nd IEEE Intl. Parallel and Distributed Processing Symp. (Proc. IPDPS)*, 2008.

[40] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama, "Impulse: Building a smarter memory controller," in *Proc. of the 5th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-5)*, 1999.

[41] J. Chang and G. Sohi, "Cooperative caching for chip multiprocessors," in *Proc. of the 33rd Intl. Symp. on Computer Architecture*, 2006.

[42] M. Chaudhuri, "PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches," in *Proc. of the 15th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-15)*, 2009.

[43] S. Chen, F. Liu, Z. Mi, Y. Zhang, R. B. Lee, H. Chen, and X. Wang, "Leveraging hardware transactional memory for cache side-channel defenses," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018.

[44] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, "Flexible hardware acceleration for instruction-grain program monitoring," in *Proc. of the 35th annual Intl. Symp. on Computer Architecture (Proc. ISCA-35)*, 2008.

[45] S. Chen, C. Delimitrou, and J. F. Martinez, "PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services," in *Proc. of the 24th intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXIV)*, 2019.

[46] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-43)*, 2016.

[47] D. Chiou, P. Jain, L. Rudolph, and S. Devadas, "Application-specific memory management for embedded systems using software-controlled caches," in *Proc. of the 37th annual Design Automation Conf.*, 2000.

[48] Z. Chishti, M. Powell, and T. Vijaykumar, "Optimizing replication, communication, and capacity allocation in CMPs," in *Proc. of the 32nd Intl. Symp. on Computer Architecture*, 2005.

[49] S. Cho and L. Jin, “Managing distributed, shared L2 caches through OS-level page allocation,” in *Proc. of the 39th intl. symp. on Microarchitecture*, 2006.

[50] A. Colin and B. Lucia, “Chain: Tasks and channels for reliable intermittent programs,” in *Proc. of the ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (Proc. OOPSLA)*, 2016.

[51] V. Dadu and T. Nowatzki, “Taskstream: Accelerating task-parallel workloads by recovering program structure,” in *Proc. of the 27th intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXVII)*, 2022.

[52] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J.-H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck, “Merrimac: Supercomputing with streams,” in *Proc. of the ACM/IEEE conf. on Supercomputing (Proc. SC03)*, 2003.

[53] W. J. Dally, “GPU Computing: To Exascale and Beyond,” in *Supercomputing ’10, Plenary Talk*, 2010.

[54] T. A. Davis and Y. Hu, “The University of Florida sparse matrix collection,” *ACM TOMS*, vol. 38, no. 1, 2011.

[55] C. De Sa, M. Feldman, C. Ré, and K. Olukotun, “Understanding and optimizing asynchronous low-precision stochastic gradient descent,” in *Proc. of the 44th annual Intl. Symp. on Computer Architecture (Proc. ISCA-44)*, 2017.

[56] J. Dean and L. A. Barroso, “The tail at scale,” *Communications of the ACM*, vol. 56, no. 2, 2013.

[57] C. Delimitrou and C. Kozyrakis, “Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters,” in *Proc. of the 18th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2013.

[58] C. Delimitrou and C. Kozyrakis, “Quasar: Resource-efficient and QoS-aware cluster management,” in *Proc. of the 19th intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XIX)*, 2014.

[59] C. Demetrescu, I. Finocchi, and A. Ribichini, “Reactive imperative programming with dataflow constraints,” in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2011.

[60] J. M. Domingos, N. Neves, N. Roma, and P. Tomás, “Unlimited vector extension with data streaming support,” in *Proc. of the 48th annual Intl. Symp. on Computer Architecture (Proc. ISCA-48)*, 2021.

[61] D. Efnusheva, A. Cholakoska, and A. Tentov, “A survey of different approaches for overcoming the processor-memory bottleneck,” *International Journal of Computer Science and Information Technology*, vol. 9, no. 2, pp. 151–163, 2017.

[62] M. Ekman and P. Stenstrom, “A robust main-memory compression scheme,” in *Proc. of the 32nd annual Intl. Symp. on Computer Architecture (Proc. ISCA-32)*, 2005.

[63] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, “Kpart: A hybrid cache partitioning-sharing technique for commodity multicores,” in *Proc. of the 24th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-24)*, 2018.

[64] C. Elliott and P. Hudak, “Functional reactive animation,” in *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, 1997.

[65] B. Falsafi and T. F. Wenisch, *A primer on hardware prefetching*. Springer Nature, 2022.

[66] A. Fuchs and D. Wentzlaff, “Scaling datacenter accelerators with compute-reuse architectures,” in *Proc. of the 45th annual Intl. Symp. on Computer Architecture (Proc. ISCA-45)*, 2018.

[67] M. Gao, G. Ayers, and C. Kozyrakis, “Practical near-data processing for in-memory analytics frameworks,” in *Proc. of the 24th Intl. Conf. on Parallel Architectures and Compilation Techniques (Proc. PACT-24)*, 2015.

[68] G. Gobieski, A. O. Atli, K. Mai, B. Lucia, and N. Beckmann, “Snafu: An ultra-low-power, energy-minimal cgra-generation framework and architecture,” in *Proc. of the 48th annual Intl. Symp. on Computer Architecture (Proc. ISCA-48)*, 2021.

[69] G. Gobieski, S. Ghosh, M. Heule, T. Mowry, T. Nowatzki, N. Beckmann, and B. Lucia, “Riptide: A programmable, energy-minimal dataflow compiler and architecture,” in *Proc. of the 55th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-55)*, 2022.

[70] M. Gokhale, B. Holmes, and K. Iobst, “Processing in memory: The Terasys massively parallel PIM array,” *Computer*, vol. 28, no. 4, 1995.

[71] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, “PipeRench: A coprocessor for streaming multimedia acceleration,” in *Proc. of the 26th annual Intl. Symp. on Computer Architecture (Proc. ISCA-26)*, 1999.

[72] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, “Strong and efficient cache side-channel protection using hardware transactional memory,” in *Proceedings of the 26th USENIX Conference on Security Symposium*, 2017.

[73] S. Gueron, “A memory encryption engine suitable for general purpose processors,” *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 204, 2016.

[74] D. Gullasch, E. Bangerter, and S. Krenn, “Cache games–bringing access-based cache attacks on aes to practice,” in *2011 IEEE Symposium on Security and Privacy*, 2011.

[75] A. Gupta, W.-D. Weber, and T. Mowry, “Reducing memory and traffic requirements for scalable directory-based cache coherence schemes,” in *Scalable shared memory multiprocessors*. Springer, 1992, pp. 167–192.

[76] J. Haj-Yihia, A. Yasin, Y. B. Asher, and A. Mendelson, “Fine-grain power breakdown of modern out-of-order cores and its implications on skylake-based systems,” *ACM Transactions on Architecture and Code Optimization*, 2016.

[77] S. Han, X. Liu, H. Mao, J. Pu, A. Pdream, M. A. Horowitz, and W. J. Dally, “Eie: Efficient inference engine on compressed deep neural network,” in *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-43)*, 2016.

[78] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley, “Exploiting heterogeneity for tail latency and energy efficiency,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 625–638.

[79] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, “Reactive NUCA: near-optimal block placement and replication in distributed caches,” in *Proc. of the 36th Intl. Symp. on Computer Architecture*, 2009.

[80] U.-U. Haus, “The Brave New World of Exascale Computing: Computation is Free, Data Movement is Not,” in *TRR154/MINOA ’21, Talk*, 2021.

[81] Z. He and R. B. Lee, “How secure is your cache against side-channel attacks?” in *Proc. of the 50th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-50)*, 2017.

[82] J. Hennessy and D. Patterson, “A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development,” in *Turing Award Lecture*, 2018.

[83] E. Herrero, J. González, and R. Canal, “Elastic cooperative caching: an autonomous dynamically adaptive memory hierarchy for chip multiprocessors,” in *Proc. of the 37th annual Intl. Symp. on Computer Architecture (Proc. ISCA-37)*, 2010.

[84] C. Hewitt, P. Bishop, and R. Steiger, “A universal modular actor formalism for artificial intelligence,” in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, 1973.

[85] A. Hilton, N. Eswaran, and A. Roth, “FIESTA: A sample-balanced multi-program workload methodology,” in *MoBS*, 2009.

[86] H. Hoffmann, D. Wentzlaff, and A. Agarwal, “Remote store programming,” in *Proc. of the 5th intl. conf. on High Performance Embedded Architectures and Compilers (Proc. HiPEAC)*, 2010.

[87] B. Hong, G. Kim, J. H. Ahn, Y. Kwon, H. Kim, and J. Kim, “Accelerating linked-list traversal through near-data processing,” in *Proc. of the 25th Intl. Conf. on Parallel Architectures and Compilation Techniques (Proc. PACT-25)*, 2016.

[88] M. Horowitz, “Computing’s energy problem (and what we can do about it),” in *ISSCC*, 2014.

[89] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith, “Informing memory operations: Providing memory performance feedback in modern processors,” in *Proc. of the 23rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-23)*, 1996.

[90] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O’Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, “Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems,” in *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-43)*, 2016.

[91] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, “Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation,” in *Proc. of the 34th Intl. Conf. on Computer Design (Proc. ICCD)*, 2016.

[92] C.-H. Hsu, Y. Zhang, M. Laurenzano, D. Meisner, T. Wenisch, L. Tang, J. Mars, and R. Dreslinski, “Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting,” in *Proc. of the 21st IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-21)*, 2015.

[93] Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu, “Elastic cgras,” in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays (FPGA)*, 2013.

[94] Intel, “Intel Optane Persistent Memory 200,” 2020.

[95] Intel corporation, “Improving real-time performance by using cache allocation technology,” *Intel Whitepaper*, 2015.

[96] Intel corporation, “Are noisy neighbors keeping in your data center keeping you up at night?” https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-rdt-infrastructure-paper.pdf, 2018, [Online; accessed 5-December-2018].

[97] Intel corporation, “Earnings report,” Q3 2018.

[98] G. Irazoqui, T. Eisenbarth, and B. Sunar, “S$a: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes,” in *2015 IEEE Symposium on Security and Privacy*, 2015.

[99] A. Jain and C. Lin, “Linearizing irregular memory accesses for improved correlated prefetching,” in *Proc. of the 46th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-46)*, 2013.

[100] A. Jain and C. Lin, “Back to the future: Leveraging Belady’s algorithm for improved cache replacement,” in *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-43)*, 2016.

[101] A. Jaleel, K. Theobald, S. C. S. Jr, and J. Emer, “High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP),” in *Proc. of the 37th annual Intl. Symp. on Computer Architecture*, 2010.

[102] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, “A scalable architecture for ordered parallelism,” in *Proc. of the 48th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-48)*, 2015.

[103] Z. H. Jiang and Y. Fei, “A novel cache bank timing attack,” in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2017, pp. 139–146.

[104] Z. H. Jiang, Y. Fei, and D. Kaeli, “Exploiting bank conflict-based side-channel timing leakage of gpus,” *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, Nov. 2019. [Online]. Available: https://doi.org/10.1145/3361870

[105] L. Jin and S. Cho, “SOS: A software-oriented distributed shared cache management approach for chip multiprocessors,” in *Proc. of the 18th Intl. Conf. on Parallel Architectures and Compilation Techniques (Proc. PACT-18)*, 2009.

[106] D. Joseph, J. L. Aragón, J.-M. Parcerisa, and A. González, “Tcor: A tile cache with optimal replacement,” in *Proc. of the 28th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-28)*, 2022.

[107] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, , and J. Torrellas, “FlexRAM: Towards an intelligent memory system,” in *Proc. of the 17th Intl. Conf. on Computer Design (Proc. ICCD)*, 1999.

[108] D. Kaplan, J. Powell, and T. Woller, “AMD Memory Encryption,” AMD, Tech. Rep., 2016.

[109] S. Karandikar, C. Leary, C. Kennelly, J. Zhao, D. Parimi, B. Nikolic, K. Asanovic, and P. Ranganathan, “A hardware accelerator for protocol buffers,” in *Proc. of the 54th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-54)*, 2021.

[110] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast Analytical Power Management for Latency-Critical Systems," in *Proc. of the 48th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-48)*, 2015.

[111] H. Kasture and D. Sanchez, "Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads," in *Proc. of the 19th intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XIX)*, 2014.

[112] H. Kasture and D. Sanchez, "TailBench: A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications," in *Proc. of the IEEE Intl. Symp. on Workload Characterization (Proc. IISWC)*, 2016.

[113] R. Kateja, N. Beckmann, and G. R. Ganger, "Tvarak: software-managed hardware offload for redundancy in direct-access nvm storage," in *Proc. of the 47th annual Intl. Symp. on Computer Architecture (Proc. ISCA-47)*, 2020.

[114] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "Ric: Relaxed inclusion caches for mitigating llc side-channel attacks," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2017, pp. 1–6.

[115] M. Kayaalp, D. Ponomarev, N. Abu-Ghazaleh, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2016, pp. 1–6.

[116] R. E. Kessler and J. L. Schwarzmeier, "CRAY T3D: A new dimension for Cray Research," in *Compcon Spring'93, Digest of Papers.*, 1993.

[117] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner, "Imagine: media processing with streams," *IEEE Micro*, vol. 21, no. 2, 2001.

[118] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proc. of the 10th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[119] T. Kim, M. Peinado, and G. Mainar-Ruiz, "{STEALTHMEM}: System-level protection against cache-based side channel attacks in the cloud," in *Proc. USENIX Security (USENIX Security-12)*, 2012.

[120] V. Kiriansky, Y. Zhang, and S. Amarasinghe, "Optimizing indirect memory references with milk," in *Proc. of the 25th Intl. Conf. on Parallel Architectures and Compilation Techniques (Proc. PACT-25)*, 2016.

[121] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *40th IEEE Symposium on Security and Privacy (S&P’19)*, 2019.

[122] P. M. Kogge, “EXECUBE-A new architecture for scaleable MPPs,” in *Proc. of the intl conf. on Parallel Processing (ICPP)*, 1994.

[123] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick, “Scalable processors in the billion-transistor era: IRAM,” *Computer*, vol. 30, no. 9, 1997.

[124] N. Kurd, S. Bhamidipati, C. Mozak, J. Miller, T. Wilson, M. Nemani, and M. Chowdhury, “Westmere: A family of 32nm IA processors,” in *IEEE Intl. Solid-State Circuits Conf.*, 2010.

[125] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, “The Stanford FLASH multiprocessor,” in *Proc. of the 21st annual Intl. Symp. on Computer Architecture (Proc. ISCA-21)*, 1994.

[126] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, “Coordinated and efficient huge page management with ingens.” in *Proc. of the 12th USENIX symp. on Operating Systems Design and Implementation (Proc. OSDI-12)*, 2016.

[127] J. H. Lee, J. Sim, and H. Kim, “Bssync: Processing near memory for machine learning workloads with bounded staleness consistency models,” in *Proc. of the 24th Intl. Conf. on Parallel Architectures and Compilation Techniques (Proc. PACT-24)*, 2015.

[128] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, “There’s plenty of room at the top: What will drive computer performance after moore’s law?” *Science*, vol. 368, no. 6495, 2020.

[129] O. Leitersdorf, R. Ronen, and S. Kvatinsky, “Making memristive processing-in-memory reliable,” in *2021 28th IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, 2021.

[130] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, “The directory-based cache coherence protocol for the dash multiprocessor,” in *Proc. of the 17th annual Intl. Symp. on Computer Architecture (Proc. ISCA-17)*, 1990.

[131] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, “Statistical properties of community structure in large social and information networks,” in *Proc. of the intl. World Wide Web conf. (WWW-17)*, 2008.

[132] B. Li and B. Jiang, "Cache attack on aes for android smartphone," in *Proceedings of the 2nd International Conference on Cryptography, Security and Privacy*, 2018.

[133] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016, p. 173.

[134] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *Proc. of the 14th intl. symp. on High Performance Computer Architecture*, 2008.

[135] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[136] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *Proc. of the 22nd IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-22)*, 2016.

[137] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, 2016.

[138] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, 2015.

[139] M. Liu and T. Li, "Optimizing virtual machine consolidation performance on numa server architecture for cloud workloads," in *Proc. of the 41st annual Intl. Symp. on Computer Architecture (Proc. ISCA-41)*, 2014.

[140] D. Lo, L. Cheng, R. Govindaraju, L. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Proc. of the 41st annual Intl. Symp. on Computer Architecture (Proc. ISCA-41)*, 2014.

[141] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: improving resource efficiency at scale," in *Proc. of the 42nd annual Intl. Symp. on Computer Architecture (Proc. ISCA-42)*, 2015.

[142] E. Lockerman, A. Feldmann, M. Bakhshalipour, A. Stanescu, S. Gupta, D. Sanchez, and N. Beckmann, "Livia: Data-centric computing throughout the memory hierarchy," in *Proc. of the 25th intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXV)*, 2020.

[143] J. Lorch and A. Smith, “Improving dynamic voltage scaling algorithms with PACE,” *SIGMETRICS PER*, vol. 29, no. 1, 2001.

[144] B. Lucia and B. Ransford, “A simpler, safer programming and execution model for intermittent systems,” in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (Proc. PLDI)*, 2015.

[145] M. Maas, K. Asanovic, and J. Kubiatowicz, “A hardware accelerator for tracing garbage collection,” in *Proc. of the 45th annual Intl. Symp. on Computer Architecture (Proc. ISCA-45)*, 2018.

[146] K. Maeng, A. Colin, and B. Lucia, “Alpaca: Intermittent execution without checkpoints,” in *Proc. of the ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (Proc. OOPSLA)*, 2017.

[147] K. Maeng and B. Lucia, “Adaptive dynamic checkpointing for safe efficient intermittent computing,” in *Proc. of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’18. Berkeley, CA, USA: USENIX Association, 2018. [Online]. Available: http://dl.acm.org/citation.cfm?id=3291168.3291178

[148] R. Manikantan, K. Rajan, and R. Govindarajan, “Probabilistic shared cache management (PriSM),” in *Proc. of the 39th Intl. Symp. on Computer Architecture*, 2012.

[149] A. Margaritov, S. Gupta, R. Gonzalez-Alberquilla, and B. Grot, “Stretch: Balancing qos and throughput for colocated server workloads on smt cores,” in *Proc. of the 25th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-25)*, 2019.

[150] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. Soffa, “Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations,” in *Proc. of the 44th intl. symp. on Microarchitecture*, 2011.

[151] M. Martin, M. D. Hill, and D. J. Sorin, “Why on-chip cache coherence is here to stay,” *Commun. ACM*, 2012.

[152] D. Meisner, B. Gold, and T. Wenisch, “PowerNap: eliminating server idle power,” *Proc. of the 14th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2009.

[153] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, “Power management of online data-intensive services,” in *Proc. of the 38th annual Intl. Symp. on Computer Architecture (Proc. ISCA-38)*, 2011.

[154] D. Meisner and T. F. Wenisch, “Stochastic queuing simulation for data center workloads,” in *EXPERT*, 2010.

[155] J. Merino, V. Puente, and J. Gregorio, “ESP-NUCA: A low-cost adaptive non-uniform cache architecture,” in *Proc. of the 16th intl. symp. on High Performance Computer Architecture*, 2010.

[156] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger, “Doppelgänger: A cache for approximate computing,” in *Proc. of the 48th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-48)*, 2015.

[157] J. S. Miguel and N. E. Jerger, “The anytime automaton,” in *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-43)*, 2016.

[158] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, “Logtm: log-based transactional memory.” in *Proc. of the 12th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-12)*, 2006.

[159] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, “Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling,” in *Proc. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-51)*, 2018.

[160] A. Mukkara, N. Beckmann, and D. Sanchez, “Whirlpool: Improving dynamic cache management with static data classification,” in *Proc. of the 21st intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXI)*, 2016.

[161] A. Mukkara, N. Beckmann, and D. Sanchez, “Cache-Guided Scheduling: Exploiting Caches to Maximize Locality in Graph Processing,” in *1st International Workshop on Architectures for Graph Processing (AGP 2017), held in conjuntion with ISCA-44*, 2017.

[162] A. Mukkara, N. Beckmann, and D. Sanchez, “PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates,” in *Proc. of the 52nd annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-52)*, 2019.

[163] M. Mushtaq, A. Akram, M. K. Bhatti, R. N. B. Rais, V. Lapotre, and G. Gogniat, “Run-time detection of prime+ probe side-channel attack on aes encryption algorithm,” in *2018 Global Information Infrastructure and Networking Symposium (GIIS)*, 2018.

[164] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, “Runahead execution: an alternative to very large instruction windows for out-of-order processors,” in *Proc. of the 9th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-9)*, 2003.

[165] A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout, “Vector runahead,” in *Proc. of the 48th annual Intl. Symp. on Computer Architecture (Proc. ISCA-48)*, 2021.

[166] Q. M. Nguyen and D. Sánchez, “Fifer: Practical acceleration of irregular applications on reconfigurable architectures,” in *Proc. of the 54th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-54)*, 2021.

[167] R. Nishtala, P. Carpenter, V. Petrucci, and X. Martorell, “Hipster: Hybrid task manager for latency-critical cloud workloads,” in *Proc. of the 23rd IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-23)*, 2017.

[168] M. D. Noakes, D. A. Wallach, and W. J. Dally, “The J-machine multicomputer: an architectural evaluation,” in *Proc. of the 20th annual Intl. Symp. on Computer Architecture*, 1993.

[169] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, “Stream-dataflow acceleration,” in *Proc. of the 44th annual Intl. Symp. on Computer Architecture (Proc. ISCA-44)*, 2017.

[170] H. Omar and O. Khan, “Ironhide:a secure multicore that efficiently mitigates microarchitecture state attacks for interactive applications,” in *Proc. of the 26th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-26)*, 2020.

[171] M. Oskin, F. Chong, and T. Sherwood, “Active pages: A model of computation for intelligent memory,” in *Proc. of the 25th annual Intl. Symp. on Computer Architecture (Proc. ISCA-25)*, 1998.

[172] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: the case of aes,” in *Cryptographers’ Track at the RSA Conference*. Springer, 2006, pp. 1–20.

[173] D. Page, “Partitioned Cache Architecture as a Side-Channel Defence Mechanism,” *IACR Cryptology ePrint archive*, no. 2005/280, 2005.

[174] G. M. Papadopoulos and D. E. Culler, “Monsoon: an explicit token-store architecture,” in *Proc. of the 17th annual Intl. Symp. on Computer Architecture (Proc. ISCA-17)*, 1990.

[175] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer, “Triggered instructions: A control paradigm for spatially-programmed architectures,” in *Proc. of the 40th annual Intl. Symp. on Computer Architecture (Proc. ISCA-40)*, 2013.

[176] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, “A case for intelligent ram,” *IEEE micro*, vol. 17, no. 2, pp. 34–44, 1997.

[177] A. Pattnaik, X. Tang, O. Kayiran, A. Jog, A. Mishra, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, “Opportunistic computing in gpu architectures,” in *Proc. of the 46th annual Intl. Symp. on Computer Architecture (Proc. ISCA-46)*, 2019.

[178] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly compressed pages: a low-complexity, low-latency main memory compression framework," in *Proc. of the 46th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-46)*, 2013.

[179] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Proc. of the 21st Intl. Conf. on Parallel Architectures and Compilation Techniques (Proc. PACT-21)*, 2012.

[180] A. Peleg and U. Weiser, "Mmx technology extension to the intel architecture," *IEEE Micro*, 1996.

[181] M. Pericas, A. Cristal, R. Gonzalez, D. Jimenez, and M. Valero, "A decoupled kilo-instruction processor," in *Proc. of the 12th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-12)*, 2006.

[182] V. Petrucci, M. A. Laurenzano, J. Doherty, Y. Zhang, D. Mosse, J. Mars, and L. Tang, "Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers," in *Proc. of the 21st IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-21)*, 2015.

[183] A. Pourhabibi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. P. Drumond, B. Falsafi, and C. Koch, "Optimus prime: Accelerating data transformation in servers," in *Proc. of the 25th intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXV)*, 2020.

[184] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, and V. Srinivasan, "NDC: Analyzing the Impact of 3D-Stacked Memory + Logic Devices on MapReduce Workloads," in *Proc. of the IEEE Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[185] M. Qureshi, "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs," in *Proc. of the 10th intl. symp. on High-Performance Computer Architecture*, 2009.

[186] M. Qureshi and Y. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. of the 39th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-39)*, 2006.

[187] M. Qureshi, A. Jaleel, Y. Patt, S. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. of the 34th annual Intl. Symp. on Computer Architecture (Proc. ISCA-34)*, 2007.

[188] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *Proc. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-51)*, 2018.

[189] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *Proc. of the 46th annual Intl. Symp. on Computer Architecture (Proc. ISCA-46)*, 2019.

[190] T. Ramírez, A. Pajuelo, O. J. Santana, and M. Valero, "Kilo-instruction processors, runahead and prefetching," in *Proceedings of the 3rd Conference on Computing Frontiers*, 2006.

[191] P. Ranganathan, S. Adve, and N. Jouppi, "Reconfigurable caches and their application to media processing," in *Proc. of the 27th annual Intl. Symp. on Computer Architecture*, 2000.

[192] S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Tempest and Typhoon: User-level shared memory," in *Proc. of the 21st annual Intl. Symp. on Computer Architecture (Proc. ISCA-21)*, 1994.

[193] T. J. Repetti, J. P. Cerqueira, M. A. Kim, and M. Seok, "Pipelining a triggered processing element," in *Proc. of the 50th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-50)*, 2017.

[194] R. Roestenburg, R. Williams, and R. Bakker, *Akka in action*. Simon and Schuster, 2016.

[195] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, J. A. Joao, and M. K. Qureshi, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," in *Proc. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-51)*, 2018.

[196] G. Salvaneschi and M. Mezini, *Towards Reactive Programming for Object-Oriented Applications*. Springer Berlin Heidelberg, 2014.

[197] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling Ways and Associativity," in *Proc. of the 43rd intl. symp. on Microarchitecture*, 2010.

[198] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning," in *Proc. of the 38th annual Intl. Symp. on Computer Architecture (Proc. ISCA-38)*, 2011.

[199] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *Proc. of the 40th annual Intl. Symp. on Computer Architecture (Proc. ISCA-40)*, 2013.

[200] S. Sardashti and D. A. Wood, "Decoupled compressed cache: exploiting spatial locality for energy-optimized compressed caching," in *Proc. of the 46th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-46)*, 2013.

[201] E. Schurman and J. Brutlag, "The user and business impact of server delays, additional bytes, and HTTP chunking in web search," in *Velocity*, 2009.

[202] C. Schuster and C. Flanagan, "Reactive programming with reactive variables," in *Companion Proceedings of the 15th International Conference on Modularity*, 2016.

[203] B. C. Schwedock and N. Beckmann, "Jumanji: The case for dynamic nuca in the datacenter," in *Proc. of the 53rd annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-53)*, 2020.

[204] B. C. Schwedock, P. Yoovidhya, J. Seibert, and N. Beckmann, "täkō: A polymorphic cache hierarchy for general-purpose optimization of data movement," in *Proc. of the 49th annual Intl. Symp. on Computer Architecture (Proc. ISCA-49)*, 2022.

[205] S. L. Scott, "Synchronization and communication in the T3E multiprocessor," in *Proc. of the 7th intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-VII)*, 1996.

[206] V. Seshadri, G. Pekhimenko, O. Ruwase, O. Mutlu, P. B. Gibbons, M. A. Kozuch, T. C. Mowry, and T. Chilimbi, "Page overlays: An enhanced virtual memory framework to enable fine-grained memory management," in *Proc. of the 42nd annual Intl. Symp. on Computer Architecture (Proc. ISCA-42)*, 2015.

[207] A. Seznec, "A case for two-way skewed-associative caches," in *Proc. of the 20th annual Intl. Symp. on Computer Architecture*, 1993.

[208] A. Seznec, "Decoupled sectored caches: conciliating low tag implementation cost," in *Proc. of the 21st annual Intl. Symp. on Computer Architecture (Proc. ISCA-21)*, 1994.

[209] O. Shacham, Z. Asgar, H. Chen, A. Firoozshahian, R. Hameed, C. Kozyrakis, W. Qadeer, S. Richardson, A. Solomatnikov, D. Stark, M. Wachs, and M. Horowitz, "Smart memories polymorphic chip multiprocessor," in *Proc. of the 46th Design Automation Conf. (Proc. DAC-46)*, 2009.

[210] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-43)*, 2016.

[211] C. Shen, C. Chen, and J. Zhang, "Micro-architectural cache side-channel attacks and countermeasures," in *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2021.

[212] M. D. Sinclair, J. Alsop, and S. V. Adve, "Chasing away rats: Semantics and evaluation for relaxed atomics on heterogeneous systems," in *Proc. of the 44th annual Intl. Symp. on Computer Architecture (Proc. ISCA-44)*, 2017.

[213] R. Singhal, "Inside intel® core microarchitecture (nehalem)," in *2008 IEEE Hot Chips 20 Symposium (HCS)*. IEEE, 2008, pp. 1–25.

[214] D. Skarlatos, N. S. Kim, and J. Torrellas, "Pageforge: a near-memory content-aware page-merging architecture," in *Proc. of the 50th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-50)*, 2017.

[215] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "Microscope: Enabling microarchitectural replay attacks," in *Proc. of the 46th annual Intl. Symp. on Computer Architecture (Proc. ISCA-46)*, 2019.

[216] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," in *Proc. of the 24th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-24)*, 2018.

[217] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *Proc. of the 36th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-36)*, 2003.

[218] N. Talati, K. May, A. Behroozi, Y. Yang, K. Kaszyk, C. Vasiladiotis, T. Verma, L. Li, B. Nguyen, J. Sun *et al.*, "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," in *Proc. of the 27th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-27)*, 2021.

[219] D. Tam, R. Azimi, L. Soares, and M. Stumm, "Managing shared l2 caches on multicore systems in software," in *Proc. of the Workshop on the Interaction between Operating Systems and Computer Architecture*, 2007.

[220] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, 1967.

[221] C. Torng, P. Pan, Y. Ou, C. Tan, and C. Batten, "Ultra-elastic cgras for irregular loop specialization," in *Proc. of the 27th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-27)*, 2021.

[222] P.-A. Tsai, N. Beckmann, and D. Sanchez, "Jenga: Software-Defined Cache Hierarchies," in *Proc. of the 44th annual Intl. Symp. on Computer Architecture (Proc. ISCA-44)*, 2017.

[223] P.-A. Tsai, N. Beckmann, and D. Sanchez, "Nexus: A New Approach to Replication in Distributed Shared Caches," in *Proc. of the 26th Intl. Conf. on Parallel Architectures and Compilation Techniques (Proc. PACT-26)*, 2017.

[224] P.-A. Tsai, C. Chen, and D. Sanchez, "Adaptive Scheduling for Systems with Asymmetric Memory Hierarchies," in *Proc. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-51)*, 2018.

[225] P.-A. Tsai, Y. L. Gan, and D. Sanchez, "Rethinking the Memory Hierarchy for Modern Languages," in *Proc. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-51)*, 2018.

[226] P.-A. Tsai and D. Sanchez, "Compress objects, not cache lines: An object-based compressed memory hierarchy," in *Proc. of the 24th intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXIV)*, 2019.

[227] K. Varadarajan, S. Nandy, V. Sharda, A. Bharadwaj, R. Iyer, S. Makineni, and D. Newell, “Molecular Caches: A caching structure for dynamic creation of application-specific Heterogeneous cache regions,” in *Proc. of the 39th intl. symp. on Microarchitecture*, 2006.

[228] D. Voitsechov and Y. Etsion, “Single-graph multiple flows: Energy efficient design alternative for gpgpus,” in *Proc. of the 41st annual Intl. Symp. on Computer Architecture (Proc. ISCA-41)*, 2014.

[229] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, “Baring it all to software: Raw machines,” *IEEE Computer*, vol. 30, no. 9, 1997.

[230] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, “Cache modeling and optimization using miniature simulations,” in *Proc. of USENIX ATC*, 2017.

[231] R. Wang and L. Chen, “Futility scaling: High-associativity cache partitioning,” in *Proc. of the 47th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-47)*, 2014.

[232] Z. Wang and R. B. Lee, “New cache designs for thwarting software cache-based side channel attacks,” in *Proc. of the 34th annual Intl. Symp. on Computer Architecture (Proc. ISCA-34)*, 2007.

[233] Z. Wang, C. Liu, A. Arora, L. John, and T. Nowatzki, “Infinity stream: Portable and programmer-friendly in-/near-memory fusion,” in *Proc. of the 28th intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXVIII)*, 2023.

[234] Z. Wang and T. Nowatzki, “Stream-based memory access specialization for general purpose processors,” in *Proc. of the 46th annual Intl. Symp. on Computer Architecture (Proc. ISCA-46)*, 2019.

[235] Z. Wang, J. Weng, S. Liu, and T. Nowatzki, “Near-stream computing: General and transparent near-cache acceleration,” in *Proc. of the 28th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-28)*, 2022.

[236] Z. Wang, J. Weng, J. Lowe-Power, J. Gaur, and T. Nowatzki, “Stream floating: Enabling proactive and decentralized cache optimizations,” in *Proc. of the 27th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-27)*, 2021.

[237] T. Wei, N. Turtayeva, M. Orenes-Vera, O. Lonkar, and J. Balkind, “Cohort: Software-oriented acceleration for heterogeneous socs,” in *Proc. of the 28th intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXVIII)*, 2023.

[238] J. Weng, S. Liu, Z. Wang, V. Dadu, and T. Nowatzki, “A hybrid systolic-dataflow architecture for inductive matrix algorithms,” in *Proc. of the 26th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-26)*, 2020.

[239] C. Wu and M. Martonosi, “A Comparison of Capacity Management Schemes for Shared CMP Caches,” in *7th Workshop on Duplicating, Deconstructing, and Debunking*, 2008.

[240] Y. Xie and G. H. Loh, “PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches,” in *Proc. of the 36th Intl. Symp. on Computer Architecture*, 2009.

[241] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff, “NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System,” in *Proc. of the 26th Symp. on Operating System Principles (Proc. SOSP-26)*, 2017.

[242] R. Xu, C. Xi, R. Melhem, and D. Moss, “Practical PACE for embedded systems,” in *EMSOFT*, 2004.

[243] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, “Invisispec: Making speculative execution invisible in the cache hierarchy,” in *Proc. of the 45th annual Intl. Symp. on Computer Architecture (Proc. ISCA-45)*, 2018.

[244] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, “Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks,” in *Proc. of the 44th annual Intl. Symp. on Computer Architecture (Proc. ISCA-44)*, 2017.

[245] H. Yang, A. Breslow, J. Mars, and L. Tang, “Bubble-Flux: Precise online QoS management for increased utilization in warehouse scale computers,” in *Proc. of the 40th annual Intl. Symp. on Computer Architecture (Proc. ISCA-40)*, 2013.

[246] Q. Yang, G. Thangadurai, and L. Bhuyan, “Design of an adaptive cache coherence protocol for large scale multiprocessors,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 3, 1992.

[247] Y. Yang, J. S. Emer, and D. Sanchez, “Spzip: Architectural support for effective data compression in irregular applications,” in *Proc. of the 48th annual Intl. Symp. on Computer Architecture (Proc. ISCA-48)*, 2021.

[248] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, “Mapping the intel last-level cache,” *Cryptology ePrint Archive*, 2015.

[249] R. Yasuhara, T. Ono, R. Mochida, S. Muraoka, K. Kouno, K. Katayama, Y. Hayata, M. Nakayama, H. Suwa, Y. Hayakawa *et al.*, “Reliability issues in analog reram based neural-network processor,” in *2019 IEEE International Reliability Physics Symposium (IRPS)*, 2019.

[250] Y. Ye, R. West, Z. Cheng, and Y. Li, “Coloris: a dynamic cache partitioning system using page coloring,” in *Proc. of the 23rd Intl. Conf. on Parallel Architectures and Compilation Techniques (Proc. PACT-23)*, 2014.

[251] V. A. Ying, M. C. Jeffrey, and D. Sanchez, “T4: Compiling sequential code for effective speculative parallelization in hardware,” in *Proc. of the 47th annual Intl. Symp. on Computer Architecture (Proc. ISCA-47)*, 2020.

[252] Y. A. Younis, K. Kifayat, and A. Hussain, “Preventing and detecting cache side-channel attacks in cloud computing,” in *Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing*, 2017.

[253] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, “Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data,” in *Proc. of the 52nd annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-52)*, 2019.

[254] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, “IMP: Indirect memory prefetcher,” in *Proc. of the 48th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-48)*, 2015.

[255] W. Yuan and K. Nahrstedt, “Energy-efficient soft real-time CPU scheduling for mobile multimedia systems,” in *Proc. of the 19th Symp. on Operating System Principles (Proc. SOSP-19)*, 2003.

[256] D. Zhang, X. Ma, and D. Chiou, “Worklist-directed Prefetching,” *IEEE Computer Architecture Letters*, 2016.

[257] D. Zhang, X. Ma, M. Thomson, and D. Chiou, “Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching,” in *Proc. of the 23rd intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXIII)*, 2018.

[258] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, “Top-pim: Throughput-oriented programmable processing in memory,” in *Proc. HPDC*, 2014.

[259] G. Zhang, V. Chiu, and D. Sanchez, “Exploiting Semantic Commutativity in Hardware Speculation,” in *Proc. of the 49th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-49)*, 2016.

[260] G. Zhang, W. Horn, and D. Sanchez, “Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems,” in *Proc. of the 48th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-48)*, 2015.

[261] G. Zhang and D. Sanchez, “Leveraging Hardware Caches for Memoization,” *Computer Architecture Letters (CAL)*, vol. 17, no. 1, 2018.

[262] G. Zhang and D. Sanchez, “Leveraging caches to accelerate hash tables and memoization,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 440–452.

[263] L. Zhang and S. Swanson, “Pangolin: A Fault-Tolerant Persistent Memory Programming Library,” in *Proc. of the USENIX Annual Technical Conf. (Proc. USENIX ATC)*, 2019.

[264] M. Zhang and K. Asanovic, “Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors,” in *Proc. of the 32nd Intl. Symp. on Computer Architecture*, 2005.