Jumanji: The Case for Dynamic NUCA in the Datacenter

Brian C. Schwedock
Carnegie Mellon University
bschwedo@andrew.cmu.edu

Nathan Beckmann
Carnegie Mellon University
beckmann@cs.cmu.edu

***Abstract*—The datacenter introduces new challenges for computer systems around *tail latency* and *security*. This paper argues that dynamic NUCA techniques are a better solution to these challenges than prior cache designs. We show that dynamic NUCA designs can meet tail-latency deadlines with much less cache space than prior work, and that they also provide a natural defense against cache attacks. Unfortunately, prior dynamic NUCAs have missed these opportunities because they focus exclusively on reducing data movement.**

We present Jumanji, a dynamic NUCA technique designed for tail latency and security. We show that prior last-level cache designs are vulnerable to new attacks and offer imperfect performance isolation. Jumanji solves these problems while significantly improving performance of co-running batch applications. Moreover, Jumanji only requires lightweight hardware and a few simple changes to system software, similar to prior D-NUCAs. At 20 cores, Jumanji improves batch weighted speedup by 14% on average, vs. just 2% for a non-NUCA design with weaker security, and is within 2% of an idealized design.

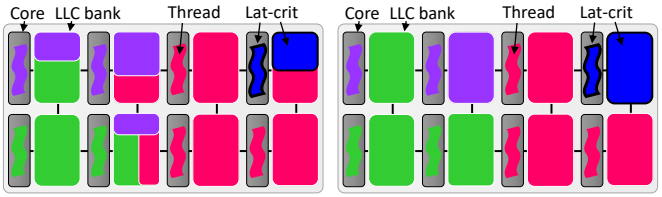
***Index Terms*—Multicore caching, non-uniform cache access (NUCA), tail latency, hardware security.**

I. Introduction

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 [16]. Simultaneously, multi-tenancy means that applications now commonly run alongside untrusted applications. Following several high-profile breaches [42, 44], **security** has become a first-order concern for many datacenter customers.

This paper 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 [12, 51], harming the performance of co-running "batch" applications without such deadlines. Prior work in security defends against microarchitectural side channels [42, 44, 45, 47, 65, 71, 83, 87, 91], but leaves some attacks undefended, harms application performance, or increases system complexity.



(a) Jigsaw [6, 8]. (b) Jumanji.

Fig. 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.

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 [2, 6, 8, 23, 61, 79, 80]. *Dynamic NUCA* (D-NUCA) techniques yield large improvements in system throughput and energy efficiency. Unfortunately, prior D-NUCAs focus solely on reducing data movement, causing them to violate tail-latency deadlines and expose unnecessary cache side channels. This paper revisits these techniques to see what they can offer in the datacenter. Our central message is that, with just a few small 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

Fig. 1a illustrates how Jigsaw [6, 8], 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,

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00061

TABLE I: Comparison of Jumanji to prior LLC designs.

		Tail latency	Security	Batch speedup
Prior work	Tail-aware [12, 35, 51]	✓	✗	✗
	Secure [20, 63, 64, 65]	✗	✓	✗
	D-NUCA [6, 8, 93]	✗	✗	✓
	Jumanji	✓	✓	✓

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 Sec. VI, 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 avoids 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. Fig. 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 [12, 51]. 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) [45], 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 I 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

This paper's 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, 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 multicore system 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.

Road map: Sec. II discusses prior work on NUCA, tail latency, and security. Sec. III motivates Jumanji with an extended case study. Sec. IV presents a high-level overview of Jumanji's design. Sec. V discusses how Jumanji enforces tail-latency deadlines, and Sec. VI how Jumanji eliminates cache bank attacks. Sec. VII gives our experimental methodology, and Sec. VIII evaluates Jumanji. Finally, Sec. IX concludes.

II. Background

A. Data movement and multicore caching

Data movement is more expensive than compute, and is becoming only more so [15, 39, 76]. This fact has led to a resurgence in cache research to reduce data movement.

Non-uniform cache access (NUCA): To improve scaling, large caches are implemented via many smaller cache banks connected over an on-chip network [40]. Commercial processors use a "static NUCA" (S-NUCA) design that simply stripes data across banks. S-NUCA exposes non-uniform latency, but suffers from a large average distance to data.

Dynamic NUCA designs try to place data closer to cores. Early D-NUCAs treated LLC banks as a hierarchy [4, 5, 10, 13, 59, 68, 93], e.g., by checking the local bank before a global "home bank." In contrast, *single-lookup D-NUCAs* restrict each memory address to live at a single LLC bank at a time [2, 6, 11, 14, 23, 33], 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 [2, 6, 8, 79, 80]. Single-lookup D-NUCAs thus significantly reduce data movement over other D-NUCAs, at the cost of modest complexity in the operating system (OS).

Beyond reducing data movement: Applications often care about objectives other than raw performance or energy efficiency. Hardware alone cannot manage data movement, as *only software knows to optimize for*. To achieve a wide range of application goals, hardware must yield control of the cache to software. Cache partitioning mechanisms [27, 69, 73, 82] let systems allocate the shared LLC among applications to manage tail latency [21, 35, 51], improve fairness [60, 66], eliminate side channels [45, 65], or minimize data movement [69, 84, 85]. However, these partitioning mechanisms ignore NUCA, needlessly increasing data movement.

Jumanji vs. prior work on caching: A key insight of this paper is that, because they place data in software, single-lookup D-NUCAs can also optimize for high-level objectives like tail latency and security, while still greatly reducing data movement vs. cache partitioning techniques. Jumanji is the first D-NUCA to realize this opportunity. Jumanji thus *generalizes prior D-NUCAs* to support modern datacenter workloads.

B. Redesigning systems for tail latency

User-facing applications in the datacenter are increasingly driving growth in computing [29]. 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 [16, 75]. 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 [3, 18, 56]. Systems minimize power through dynamic voltage and frequency scaling (DVFS) [26, 34, 50, 51, 52, 86, 92], varying parallelism [22, 62, 67] as load fluctuates, or finding jobs that can safely run alongside latency-critical applications [17, 18, 19, 55, 89]. This work is complementary to Jumanji and falls outside the scope of this paper.

Caching for tail latency: A few systems focus on the effect of the LLC on tail latency. Ubik [35] partitions the LLC to safely co-locate 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 [51] and Parties [12] 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 [27] (i.e., way-partitioning). We compare Jumanji with a similar scheme; however, we compare them only at the LLC.

Jumanji vs. prior work on tail latency: We echo this broad body of work in showing that D-NUCA must be designed for tail latency explicitly; designing for overall system efficiency is insufficient (Sec. V). Like prior work, Jumanji focuses on the LLC's impact on tail latency and uses feedback control. However, no prior work has considered NUCA, which we show leaves significant performance on the table.

C. Security and cache attacks

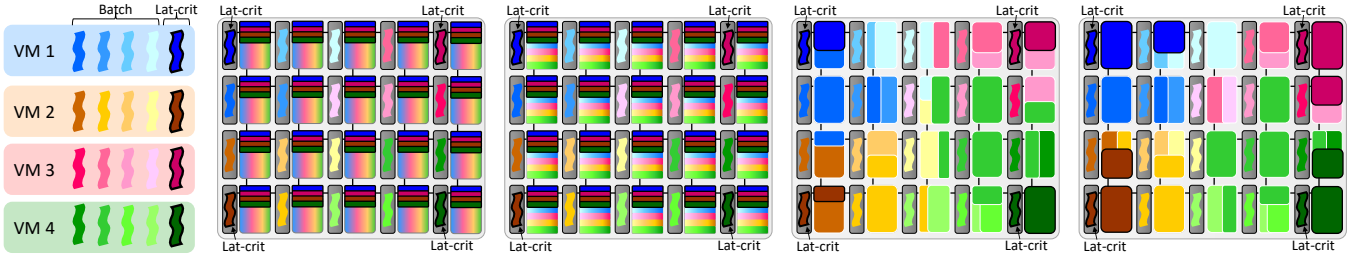
Recent work has demonstrated many microarchitectural security vulnerabilities. This paper focuses on shared-cache attacks which allow an attacker either to learn a victim's access pattern through side channels [37, 38, 65] or harm a victim's performance. Prior work considers content-based timing side-channel attacks, specifically *conflict attacks* where an attacker primes the cache so that a victim's access will evict the attacker's data [46, 64, 88]. The attacker detects what data the victim accesses by timing its own cache accesses.

Defending conflict attacks: Prior work offers many defenses for conflict attacks [37, 47, 71, 72, 88]. However, way-partitioning (i.e., Intel CAT [27]) is the simplest and by far the most common defense. Way-partitioning restricts different processes to different cache ways, 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 [41, 45, 83]. Many alternatives to way-partitioning face similar limitations [43, 49, 90] or do not guarantee isolation [53, 73, 82].

Other 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 [1, 9]. Caches' limited ports make them vulnerable to port attacks, which have been demonstrated in CPU L1 caches [31] and GPUs [32]. In Sec. VI, we demonstrate that CPU LLCs are also vulnerable to port attacks.

Moreover, 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 [63]. 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 tail latency and does not optimize data placement within each cluster to reduce data movement.



(a) Workload. (b) Adaptive [12, 51]. (c) VM-Part [45, 65]. (d) Jigsaw [6, 8]. (e) Jumanji.

Fig. 2: 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.

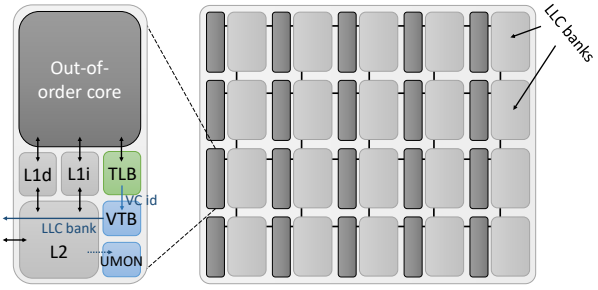


Fig. 3: A 20-core system with a distributed LLC (20×1 MB banks). Jumanji adds simple hardware to control data placement, borrowed from Jigsaw [6, 8]. Green indicates modified components, and blue indicates new components.

Jumanji vs. prior work on security: 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.

III. 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 [3, 18, 56].

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 paper presents Jumanji, a new D-NUCA that meets all of these demands in a single, simple design. Fig. 3 shows the multicore that we consider in this paper: 20 out-of-order cores share a 20 MB LLC that is distributed into 20 banks over a mesh network-on-chip (NoC). (See Sec. VII for details.) To this base multicore, Jumanji adds D-NUCA hardware that controls where data is placed in the LLC, as detailed in Sec. IV-A.

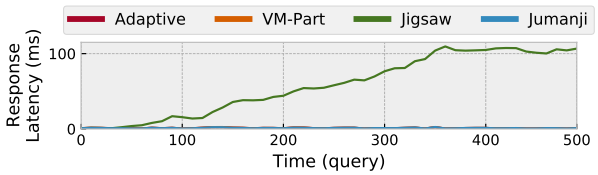
A. Case study

To see how Jumanji improves upon prior work, we now consider an extended case study, illustrated in Fig. 2. The workload is shown in Fig. 2a: four VMs share the 20-core system, each running one latency-critical application (`xapian` from TailBench [36]) and four batch applications (randomly chosen from SPEC CPU2006); we show that other workloads yield the same conclusions in Sec. VIII.

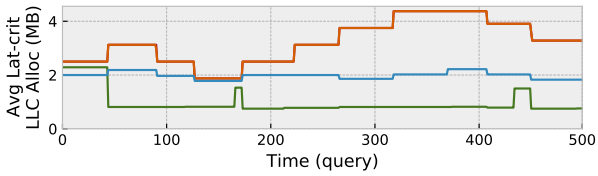
To see how different LLC designs behave on this workload, the remaining plots in Fig. 2 depict where threads and data are placed in each design. Each VM is represented as a different color (blue, brown, pink, and green), and applications within each VM as different shades of this color. Threads are clustered 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:

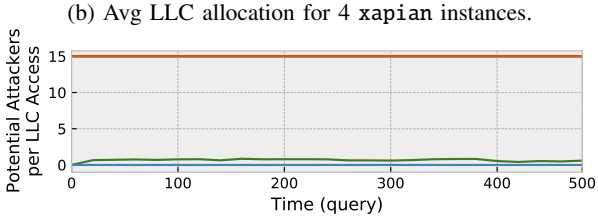
- **Adaptive** (Fig. 2b) reserves space in each bank using way-partitioning [27] and dynamically adjusts the allocations through feedback control [12, 51]. Adaptive partitions latency-critical data to guarantee tail latency is kept low, but it does not partition batch data because doing so lowers LLC 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** (Fig. 2c) 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 (Sec. II-C), but lowers LLC associativity and thus harms batch performance.
- **Jigsaw** (Fig. 2d) is a state-of-the-art D-NUCA that minimizes data movement [6, 8], but ignores tail latency and security. Jigsaw places data in LLC banks near threads, and partitions data within each bank for performance isolation.



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



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



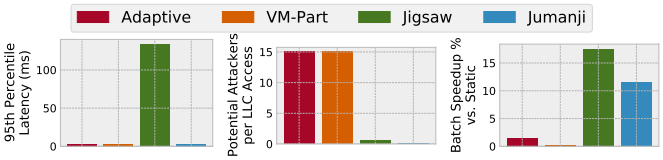
(c) Vulnerability to shared-cache-structure attacks (Sec. VI-A).

Fig. 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.

• **Jumanji** (Fig. 2e) is our new D-NUCA design that targets applications' tail-latency and security goals. Like 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: Fig. 4 quantifies how each LLC design behaves over time, in terms of tail latency, LLC space, and security. Fig. 4a shows `xapian`'s request latencies. All designs maintain low tail latency, except for Jigsaw, whose latency grows increasingly large over time. Fig. 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 allocation achieve equivalent performance (see Sec. V-A).

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 (Sec. II-C). However, since VM-Part is an S-NUCA design with limited associativity, it pays for this security with lower batch performance. This paper also considers an attacker that observes a victims' LLC accesses through other shared cache structures, e.g., cache ports (see Sec. VI-A). For such attacks to succeed, *the attacker only needs to share an LLC bank with the victim*. Fig. 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.



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

Fig. 5: Jumanji meets tail-latency deadlines and defends side channels much more efficiently than non-NUCA approaches.

Jumanji gets the best of all worlds: Fig. 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. The rest of this paper expands on this motivation, describing how Jumanji works and evaluating it across many applications.

IV. Jumanji's Design in a Nutshell

Fig. 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 domain" (e.g., the VM they belong to [45]).

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:

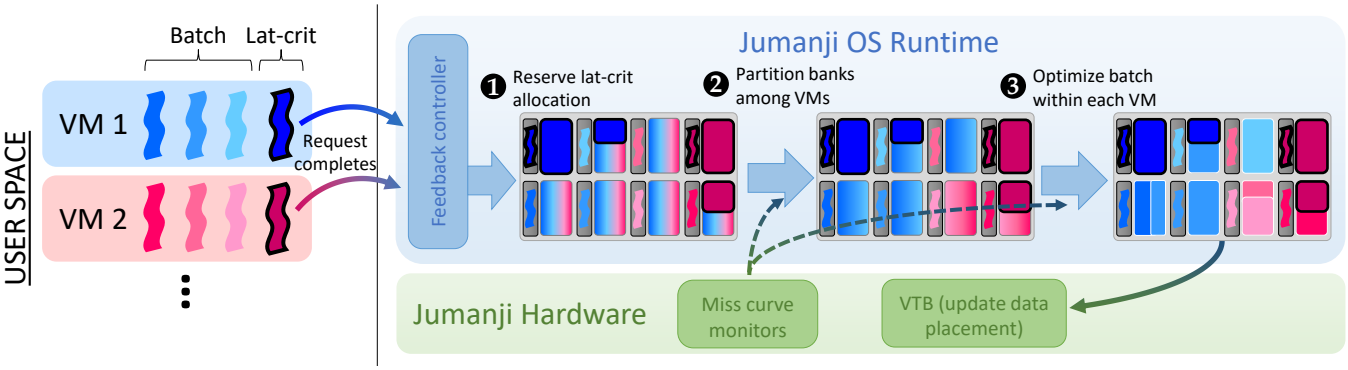


Fig. 6: Jumanji periodically reoptimizes data placement to ensure that tail-latency deadlines are met and side channels are eliminated. ❶ A feedback controller reserves space for latency-critical applications to meet their deadlines. ❷ Data from untrusted applications is physically separated into different LLC banks. ❸ Data placement is optimized within each VM to minimize data movement.

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.

A. Jumanji hardware

Jumanji borrows Jigaw's D-NUCA hardware without modification [6, 8]. For readers unfamiliar with Jigsaw, we now briefly describe how it works. Fig. 3 depicts the hardware Jumanji adds to control data placement. Jumanji is a single-lookup D-NUCA that places data at page granularity (Sec. II-A).

Controlling data placement: Jumanji maps each page to a *virtual cache* (VC), a new OS abstraction for managing data placement. For this paper, it suffices to think of there being one VC per application [61, 80]. 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 Fig. 7. The VTB maps a VC id to a *placement descriptor*, a 128-entry array of bank ids. 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 [6, 8].

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) [8, 69], which

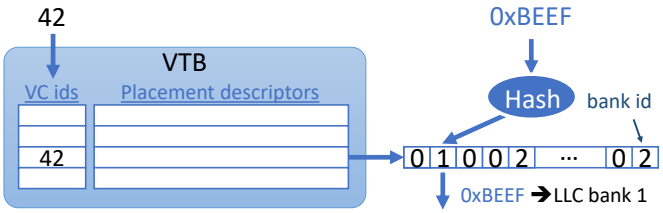


Fig. 7: The virtual-cache translation buffer (VTB) controls data placement. Addresses are hashed to index into the VC's placement descriptor, yielding the address's unique LLC location.

sample ≈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 [27] and DRRIP replacement [30] within each LLC bank, instead of Vantage partitioning [73] 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 [7, 81].

B. 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 Secs. V and VI) 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 [8, 23].

Jumanji requires integration with the VM hypervisor to know which applications run within each VM. This allows 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 (Sec. VI-A). Like other secure designs [63], 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.

V. 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.

A. Motivation: Why D-NUCA for tail latency?

First, we evaluate the effect of D-NUCA on `xapian`'s tail latency independent of batch applications. Fig. 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 Fig. 2b and Fig. 2e.)

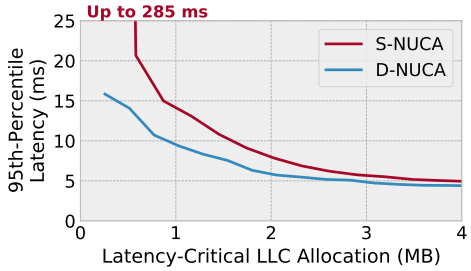


Fig. 8: How `xapian`'s tail latency varies with its cache allocation, with and without D-NUCA. D-NUCA lets `xapian` meet its tail-latency deadline with much less LLC space.

Fig. 8 illustrates two important points. First, as found in prior work [35, 51], 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.

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× lower than S-NUCA's.

B. Jumanji's OS interface

Similar to prior work on tail latency (Sec. II-B), 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 [18]. 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 [8], but that is outside the scope of this work.

C. 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. Listing 1 gives pseudocode for this procedure.

Listing 1: 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.

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

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 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 [34].

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. Fig. 9 shows gmean weighted speedup (bars) and tail latency (lines) for the same workload as Fig. 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.

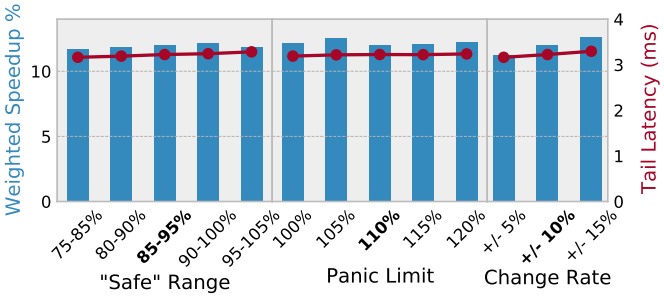


Fig. 9: Variation in speedup and latency as parameter values for the feedback controller change. Jumanji is insensitive to values.

Listing 2: 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 cache bank `b`.

```
1  def LatCritPlacer(bankBalance): # capacity per bank
2    orderedBanks = sortBanksByDistance(latApps)
3    foreach latApp:
4      preferredBanks = orderedBanks[latApp]
5      latAppAlloc = latAppSize[latApp] # Set by feedback
6      while latAppAlloc > 0: # allocate greedily
7        bestBank = preferredBanks.next()
8        allocSize = min(bankBalance[bestBank],latAppAlloc)
9        allocs[bestBank][latApp] = allocSize
10       latAppAlloc -= allocSize
11   return allocs
```

D. 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.

Listing 2 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 Sec. VIII-C shows, Jumanji's batch performance with this greedy placement is already close to an idealized design.

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.

VI. 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.

A. Threat model

Jumanji targets datacenters where a single machine is shared by several VMs. As in prior work [45], 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. Fig. 10 illustrates the attacks that we consider.

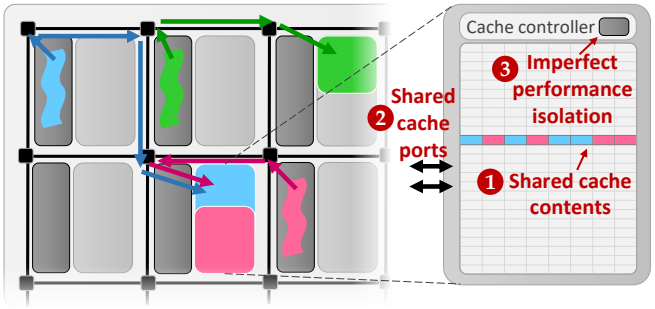


Fig. 10: The three shared cache components considered in this paper. 1: *Conflict attacks* through shared cache sets. 2: *Port attacks* through shared bank ports. And 3: *Imperfect performance isolation* through adaptive cache replacement state.

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 Sec. II-C). 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.

3 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 [27] does *not* defend against attacks 2 and 3, since it does not separate data into different banks. The rest of this section explores these attacks and explains how Jumanji defends them.

B. Demonstration of an LLC port attack

Cache banks have a limited number of ports [40]. 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 (Sec. II-C) 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.

Fig. 11 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 [48], and records the time to complete every 100 LLC accesses (to amortize timing overheads). Fig. 11 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.

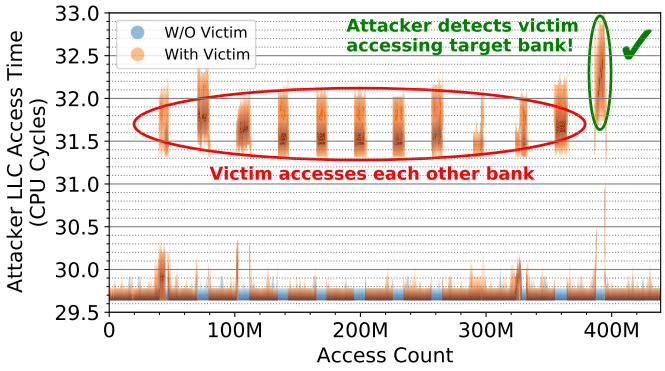


Fig. 11: LLC access times for an attacker flooding a target LLC bank with accesses (with and without a co-running victim process). The victim accesses each LLC bank, causing 12 spikes in latency for the attacker. The attacker detects victim accesses to a target bank by higher access times due to port conflicts.

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 Fig. 11. Note that the victim accesses a *different* cache set from the attacker to guarantee that contention does not occur from the cache contents.

Fig. 11 shows that latency increases whenever the victim is active due to NoC contention, but delay is noticeably 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 [78], frequent coherence misses to shared data among victims, or frequent evictions if the victim has a small LLC partition.

C. Performance leakage and degradation in way-partitioning

Way-partioning 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 [30, 70]. 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*.

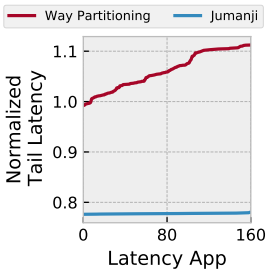


Fig. 12: Tail-latency distribution for four instances of `img-dnn` when run alongside 40 batch mixes with a fixed LLC partition.

Fig. 12 demonstrates the impact of this performance leakage. `img-dnn`, a latency-critical application from Tailbench [36], is run alongside different mixes of batch applications with DRRIP replacement [30]. 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 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 (Sec. II-C), 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 [41, 45]. 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.

D. 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.

Listing 3: 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.

```
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
```

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.

We propose the `JumanjiPlacer`, which guarantees bank isolation between VMs, and efficiently meets tail-latency deadlines by building on `LatCritPlacer` (Listing 2). Jumanji achieves these benefits through a two-tiered placement algorithm which only allows shared banks between applications in the same VM, as shown in Listing 3.

`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 [61, Appendix B]. Remaining LLC capacity is then divided among batch applications using a slightly modified version of the `Lookahead` algorithm [69] 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 (Listing 3, 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.

Cores	20 cores, x86-64 ISA, 2.66 GHz OOO Nehalem [77]
L1 caches	32 KB, 8-way set-associative, split data and instruction caches, 3-cycle latency
L2 caches	128 KB private per-core, 8-way set-associative, inclusive, 6-cycle latency
Coherence	MESI, 64 B lines, no silent drops; sequential consistency
Last-level cache	20 MB shared LLC, 5×4 1 MB banks; 32-way set-associative, 13-cycle bank latency; mesh NoC, 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

TABLE II: System parameters in our experimental evaluation.

VII. Methodology

We evaluate Jumanji through detailed microarchitectural simulation using ZSim [74]. Our experimental methodology is similar to prior work [35, 79] and is detailed below.

System: Parameters are shown in Table II. We model a 20-core system with a 20 MB shared LLC, with out-of-order cores modeled on Nehalem [77]. 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 [23, 24, 54, 79]. Each LLC bank uses way-partitioning (i.e., Intel CAT [27]) and DRRIP replacement [30]. Main memory models bandwidth partitioning with fixed latency [28, 51].

Applications: We use latency-critical applications from Tailbench [36] 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 [57, 58]. 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 Fig. 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 applica-

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

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

TABLE III: Workload config. for latency-critical applications.

tions. (Higher latency percentiles would require prohibitively long simulations.) We compute weighted speedup using a fixed-work methodology similar to FIESTA [25]: 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 III. 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 corresponds to allocating the four latency-critical applications half of the LLC.

LLC designs: We primarily compare the four designs already introduced in Sec. III:

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.

VIII. 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.

A. 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 [27], 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. Fig. 14 shows this vulnerability, plotting the average number of untrusted processes sharing a bank 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.

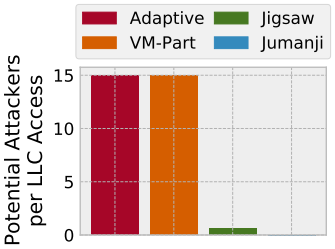


Fig. 14: Each LLC design's vulnerability to port attacks, averaged over all experiments.

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.

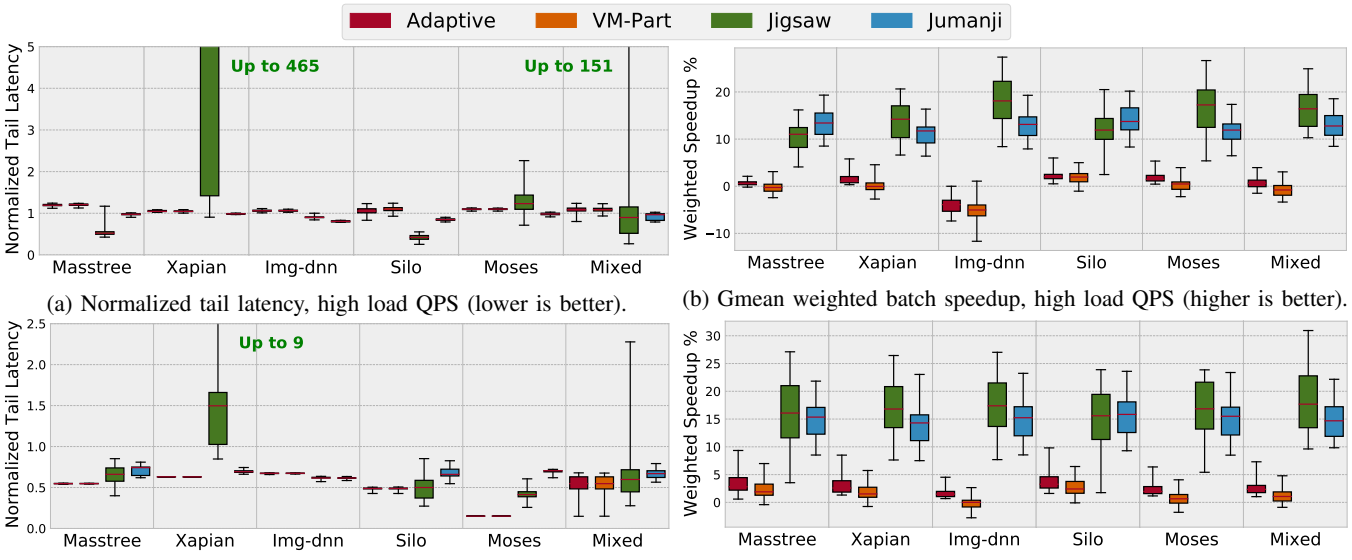
B. Tail latency and batch speedup

Jumanji meets deadlines with minimal data movement: Fig. 13 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). Fig. 13 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: Fig. 13 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.

Jumanji supports multiple different latency-critical applications: Prior 20-core results evaluated multiple instances of the same latency-critical application. Fig. 13 also shows that Jumanji meets deadlines when running mixes of *different* applications, whereas Jigsaw violates deadlines for one-third of



(a) Normalized tail latency, high load QPS (lower is better).

(b) Gmean weighted batch speedup, high load QPS (higher is better).

(c) Normalized tail latency, low load QPS.

(d) Gmean weighted batch speedup, low load QPS.

Fig. 13: Normalized tail latency and gmean batch weighted batch speedup (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.

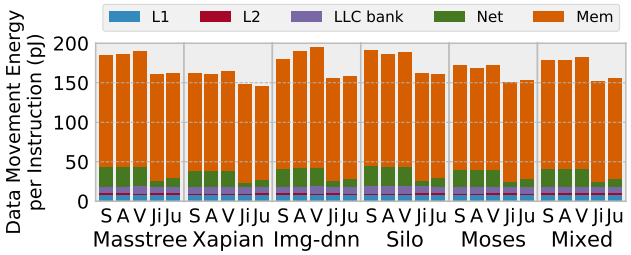


Fig. 15: Dynamic data movement energy for latency-critical applications at high load over 40 random batch mixes. S=Static, A=Adaptive, V=VM-Part, Ji=Jigsaw, Ju=Jumanji. Jumanji matches Jigsaw's data movement reductions without violating tail-latency deadlines.

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: Fig. 15 shows average dynamic data movement energy for each workload at high load in Fig. 13. Data movement energy is split between the L1, L2, LLC banks, on-chip network, and memory, using numbers from prior work [79].

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 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).

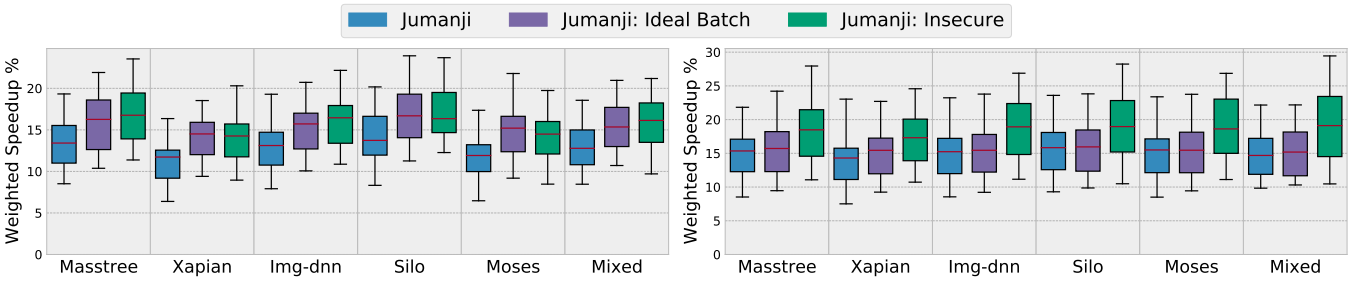
C. Sensitivity studies

Jumanji defends cache attacks at low cost: Fig. 16 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 Sec. V, 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?

Fig. 16 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., Fig. 8). Ideal Batch also isolates VMs for security. The result is an infeasibly optimal batch placement unconstrained by any latency-critical placement.

Fig. 16 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



(a) Gmean weighted batch speedup, high load QPS.

(b) Gmean weighted batch speedup, low load QPS.

Fig. 16: 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.

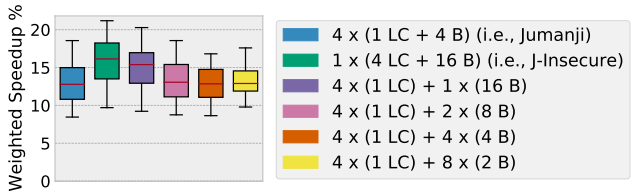


Fig. 17: Jumanji's batch speedup when varying from 1 VM (all apps) to 12 VMs. Jumanji scales well as VMs increase.

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 applications, similar to [8]. In contrast to [8], we found that trades were very rare and yielded little speedup: [8] 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 Fig. 17. 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. Fig. 17 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.

Fig. 17 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 Fig. 17 only shows results for mixed latency-critical applications at high load, results are similar for other configurations too.

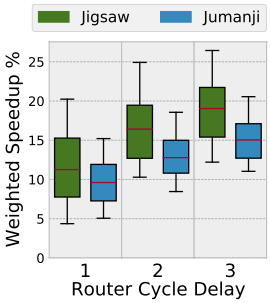


Fig. 18: NoC sensitivity.

NoC sensitivity: Finally, we see how speedups vary with NoC latency. Results so far use 2-cycle router delay to model modest NoC congestion. Fig. 18 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.

IX. Conclusion

This paper has 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.

Acknowledgments

We thank the anonymous reviewers, Brandon Lucia, Graham Gobieski, Elliot Lockerman, and Bryan Parno for their feedback. Brian Schwedock is supported by an NSF Graduate Research Fellowship, and this work is further supported by NSF grant CCF-1845986.

REFERENCES

[1] 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.

[2] 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.

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

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

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

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] W. J. Dally, "GPU Computing: To Exascale and Beyond," in *Supercomputing '10, Plenary Talk*, 2010.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] F. Guo, Y. Solihin, L. Zhao, and R. Iyer, "A framework for providing quality of service in chip multi-processors," in *Proc. of the 40th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-40)*, 2007.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] Intel corporation, "Improving real-time performance by using cache allocation technology," *Intel Whitepaper*, 2015.

[28] 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].

[29] Intel corporation, "Earnings report," Q3 2018.

[30] 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.

[31] 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.

[32] 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

[33] 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.

[34] 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.

[35] 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*, 2014.

[36] 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.

[37] 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.

[38] 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.

[39] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, 2011.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] F. Liu and R. B. Lee, "Random fill cache architecture," in *Proc. of the 47th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-47)*, 2014.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] J. Lorch and A. Smith, "Improving dynamic voltage scaling algorithms with PACE," *SIGMETRICS PER*, vol. 29, no. 1, 2001.

[53] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management (PriSM)," in *Proc. of the 39th Intl. Symp. on Computer Architecture*, 2012.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] D. Meisner and T. F. Wenisch, "Stochastic queuing simulation for data center workloads," in *EXPERT*, 2010.

[59] 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.

[60] M. Moreto, F. J. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero, "FlexDCP: A QoS framework for CMP architectures," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, 2009.

[61] 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.

[62] 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.

[63] H. Omar and O. Kahn, "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.

[64] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *CryptographersâĂŹ Track at the RSA Conference*. Springer, 2006, pp. 1–20.

[65] D. Page, "Partitioned Cache Architecture as a Side-Channel Defence Mechanism," *IACR Cryptology ePrint archive*, no. 2005/280, 2005.

[66] A. Pan and V. Pai, "Imbalanced cache partitioning for balanced data-parallel programs," in *Proc. of the 46th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-46)*, 2013.

[67] 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 *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015, pp. 246–258.

[68] 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.

[69] 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*, 2006.

[70] 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.

[71] 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.

[72] 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.

[73] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning," in *Proc. of the 38th annual Intl. Symp. in Computer Architecture*, 2011.

[74] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *Proc. of the 40th Intl. Symp. on Computer Architecture*, 2013.

[75] E. Schurman and J. Brutlag, "The user and business impact of server delays, additional bytes, and HTTP chunking in web search," in *Velocity*, 2009.

[76] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *Proc. High Performance Computing for Computational Science (VECPAR)*, 2011.

[77] R. Singhal, "Inside intel® core microarchitecture (nehalem)," in *2008 IEEE Hot Chips 20 Symposium (HCS)*. IEEE, 2008, pp. 1–25.

[78] 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.

[79] 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.

[80] 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.

[81] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, "Cache modeling and optimization using miniature simulations," in *Proc. of USENIX ATC*, 2017.

[82] 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.

[83] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," 2007.

[84] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang, "Dcaps: Dynamic cache allocation with partial sharing," in *Proc. of the EuroSys Conf. (Proc. EuroSys)*, 2018.

[85] C. Xu, K. Rajamani, A. Ferreira, W. Felter, J. Rubio, and Y. Li, "dcat: Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service," in *Proc. of the EuroSys Conf. (Proc. EuroSys)*, 2018.

[86] R. Xu, C. Xi, R. Melhem, and D. Moss, "Practical PACE for embedded systems," in *EMSOFT*, 2004.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

[92] 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.

[93] 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.