A Harmony of Ice and Fire: Mitigating Cold Last-Level Cache Effects in Serverless Computing on Commodity Hardware

Qi Ling Ajay Rawat Pedro Fonseca Kazem Taram
Purdue University
West Lafayette, IN, United States
{ling102, rawat10, pfonseca, kazem}@purdue.edu

***Abstract*—Serverless computing has become a popular paradigm for deploying event-driven services, allowing developers to focus purely on their service logic and leave infrastructure management to cloud providers. However, serverless functions are short-lived and heavily interleaved, causing them to execute with cold microarchitectural state and lose much of the performance benefit normally delivered by modern caches. We show that ineffective management of the last-level cache (LLC) is a primary contributor to this overhead. This paper presents HarmoniCa, a dynamic, fine-grained LLC management system that requires no hardware or application changes. HarmoniCa introduces a slice-aware LLC partitioning mechanism that leverages the linear structure of Intel's slice-hashing function to construct lightweight set–slice partitions, improving partitioning granularity and reducing partitioning overheads by up to 7× compared to existing techniques. Building on a detailed characterization of serverless cache behavior and using our novel partitioning technique, HarmoniCa learns each function's sensitivity to cold LLC, and dynamically allocates shared and private LLC partitions. Across a range of serverless workloads, HarmoniCa improves the performance of cache-sensitive functions by 34.7%, without notably harming the performance of other co-located functions.**

***Index Terms*—serverless computing, cache partitioning.**

I. Introduction

Major cloud providers continue to invest heavily in serverless computing, particularly Function-as-a-Service (FaaS) [1]. Amazon AWS [2], Azure [3], Google Cloud [4], and Alibaba Cloud [5] all operate large-scale FaaS platforms that power microservices [6], data pipelines [7], [8], and a growing class of ML workloads [9]–[11]. This paradigm benefits both developers and providers: developers express applications as stateless, event-triggered *functions* without managing deployment, while providers handle infrastructure, placement, and scaling, enabling higher resource utilization and lower operational costs.

However, modern processors that execute these serverless functions still rely on microarchitectural optimizations originally designed decades ago for monolithic, long-running programs. In contrast, serverless workloads exhibit vastly different behavior that renders many of these traditional optimizations far less effective [12]–[14]. In particular, serverless workloads are extremely short-lived (sub-second, often sub-millisecond) [15]–[17], and yet, have long inter-arrival time (IAT) measured in seconds or minutes [18]. For example, Huawei reports a median execution time of 5ms, while over 97% of functions have an average IAT longer than 1s [19]. Azure reports that 81% of its serverless functions are invoked no more than once per minute, with execution times over two orders of magnitude shorter than their average IAT [18]. Similar trends are also observed on Amazon AWS [20].

The short-lived nature of these functions leaves too little time for traditional stateful microarchitectural optimizations, such as caches and branch predictors, to warm up and produce meaningful benefit. Meanwhile, the long inter-arrival times of these functions prevent any useful microarchitectural state from carrying across invocations, as intervening workloads running on the same processor pollute these microarchitectural structures. Thus, serverless functions often run on cold microarchitectural state, leading to over 2× slowdown compared to execution with warm microarchitectural state [12]–[14], [21]–[23].

Unfortunately, existing approaches to address the cold microarchitectural state hinge on hardware changes. For instance, recent work proposes either designing entirely new microarchitectures tailored to serverless workloads [24], modifying existing microarchitectural mechanisms by partitioning hardware resources [13], or restoring state into these structures [12], [14], [21]. These approaches, however, require hardware changes that are costly and slow to deploy, and maintaining different processor designs for different workload types is undesirable.

In this paper, we take a different approach: we show that much of the cold microarchitectural state problem can be mitigated using software-only techniques on commodity hardware, without changes to applications or hardware. We propose HarmoniCa, a novel fine-grained, software-only cache management system that delivers the bulk of performance benefits demonstrated in prior work [12], [13] without requiring hardware modifications.

We first conduct a systematic study of the microarchitectural resources that contribute to the cold state problem. Prior work [12], [14], [21] identifies CPU frontend stalls, due to icache misses and branch mispredictions, as the primary source of the performance degradation, and thus prioritizes optimizing front-end resources. In contrast, our analysis shows that the LLC plays a more significant role: while at the pipeline level most stalls stem from frontend issues, mitigating cold LLC state accelerates the warmup of many structures, alleviating

both frontend and backend stalls.

While LLC partitioning [25]–[32] is a well-established technique for isolating workloads and mitigating interference, applying it to serverless remains a significant challenge. We show that existing cache partitioning solutions fail to deliver any performance gain, as both their *partitioning mechanisms* and their *management strategies* are ill-suited for serverless workloads.

Intel Cache Allocation Technology (CAT) and page-coloring set partitioning [27], [28], the primary partitioning mechanisms used today, both provide only a limited number of partitions. CAT exposes less than 16 classes of service, while traditional cache-coloring set partitioning approaches [27], [28] provide at most 32 for unmodified applications on our platform. Moreover, both set and way partitioning significantly degrade performance as the number of partitions increases. Way partitioning reduces the associativity available to each partition, while set partitioning limits usable L2 capacity, as L2 index bits are often shared with the LLC.

To address these limitations, we develop a novel partitioning mechanism with the highest granularity to date. By leveraging the LLC *slice* as an additional partitioning dimension and combining it with CAT's way partitioning and traditional set partitioning, HarmoniCa provides fine-grained control over LLC capacity (with up to 1536 partitions) without restricting each function's usable L2 cache or sacrificing associativity in most scenarios.

Existing software-based cache management strategies target long-running, concurrent programs that contend for LLC capacity, where underutilized cache can be dynamically resized to improve overall utilization [25]–[29], [33]. However, such an approach is unsuitable for serverless workloads, where preserving a function's state across invocations is critical, as reassignment would discard that state. Moreover, the large number of inactive functions, often on the order of hundreds to thousands per node [34], makes replacement decisions among contending functions crucial. We observe that serverless functions exhibit streaming and thrashing access patterns that render simple LRU-based policies ineffective.

HarmoniCa overcomes these limitations by introducing a runtime cache management system that dynamically manages LLC resources using novel policies derived from our study of serverless workloads. HarmoniCa employs a stream- and thrash-resistant allocation strategy, adaptively sizes partitions to match functions' demands, and exploits opportunities to share cache across functions with shared memory (e.g., those using the same language runtime).

Our evaluation on a recent Intel processor with Azure invocation traces shows that HarmoniCa improves the median latency of cache-sensitive serverless functions by up to 87.3%, with a 34.7% average improvement. Further, HarmoniCa remains effective across different system loads and service densities.

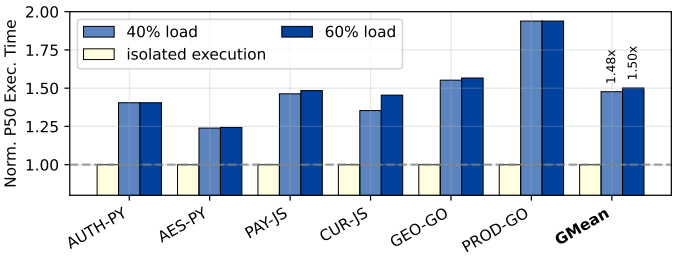


Fig. 1. Slowdown of functions due to interleaving.

II. Motivation

In this section, we analyze how cold microarchitectural states cause slowdowns and explain why LLC replacement policies fail for serverless functions. Finally, we show why prior cache partitioning approaches fall short in serverless.

A. Cold Microarchitectural State

Prior work [12], [13], [21]–[23], [35] observed that serverless functions receive little benefit from microarchitectural structures such as caches and branch predictors, as core oversubscription and frequent context switches pollute these microarchitectural state. More specifically, the inter-arrival time of functions is often so long that many other intervening functions execute between two consecutive invocations of the same function. According to Mosaic [13], for 21% of consecutive invocations of a given function, its invocations are separated by the execution of at least eight other functions on the same core. Unfortunately, executing those eight or more intervening functions is enough to evict most microarchitectural state, leaving little to no useful information for the next invocation. Furthermore, unlike long-running monolithic programs, the cost of microarchitecture warming up can hardly be amortized by the short runtime of serverless functions, which often last for milliseconds or less [18], [20], leading to severe performance overhead.

To begin, we design an experiment to quantify the impact of interleaved function execution on the performance of serverless workloads. We use 13 functions from the vSwarm benchmark [36]. For each function, 3 instances are created, adding to 39 services in total. Each service is invoked using exponential distribution. We measure each function's latency under three scenarios: (1) running each function alone without interleaving with other functions (isolated execution), (2) interleaving functions to reach 40% CPU load, and (3) interleaving to reach 60% CPU load. The full experimental setup, including the details of our server, is presented in §VII. Figure 1 summarizes the results, showing up to $1.8\times$ slowdown (with a geometric mean of $1.50\times$) under 60% CPU load compared to isolated execution. This is consistent with the results reported by previous work [12], [21], [23].

In addition, we quantify how the coldness of different microarchitectural structures contributes to overall performance degradation. To that end, we use hardware performance counters to measure miss rates of major stateful components, including caches, branch predictors, and translation lookaside buffers

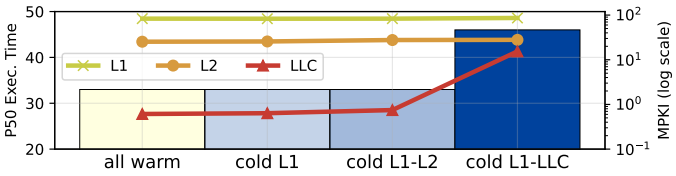


Fig. 2. Impact of cold caches on AUTH-PY. The left y-axis shows p50 latency under thrashing at different cache levels, and the right y-axis shows corresponding miss rates (MPKI). Cold LLC causes the largest slowdown (40%), while cold L1 and L2 have minimal impact.

(TLBs). We observe a sharp increase in LLC miss rates across all benchmarks once interleaving begins. For instance, AUTH-PY shows a $4.5\times$ increase. Branch predictor accuracy also degrades, with an average $1.3\times$ increase in mispredictions across all the functions, while TLBs show only a $\sim$1% rise in miss rate.

Performance counter results hint that memory-related structures (both instruction and data), contribute most to the observed slowdown. To quantify this effect, we repeat the same experiment but prevent the interleaving functions from evicting LLC state by assigning each target function an exclusive LLC partition. Note that this setup keeps the interleaving impact on on-core structures (e.g., L1/L2 caches and branch predictors) unchanged, while isolating only the effect on the LLC. With this isolation, the overall performance overhead drops significantly. For example, the slowdown of AUTH-PY decreases from 24% to only 9%. This result indicates that most of the interference responsible for performance degradation is caused by the LLC state loss across invocations.

To confirm this, we conduct an experiment that isolates the impact of coldness at different cache levels on function performance. We introduce controlled cache pollution by inducing thrashing between consecutive invocations, using buffers of increasing sizes to target specific cache levels. Figure 2 shows results for AUTH-PY. A cold LLC incurs the largest overhead (40%), while cold L1 and L2 have minimal impact, with largely unchanged miss rates. Because L1 and L2 are small with low miss penalties (tens of cycles), they warm up quickly and their cost is easily amortized. In contrast, the larger LLC has higher miss penalties (hundreds of cycles due to DRAM access) and takes longer to warm up. Similar trends hold across other functions as can be observed in Table I.

We also find that both instruction and data caches contribute significantly to cold-start overhead. By controlling the physical pages mapped to different memory regions with page coloring, we independently pollute instruction and data caches. Results are shown in Table I. Overall, data cache pollution dominates for Python and Node.js, whereas instruction cache pollution is more costly for Go.

Unlike prior work [12], [14], [21], which focuses mainly on frontend stalls, we show that mitigating cold LLC state accelerates the warmup of several structures, alleviating both frontend and backend stalls.

B. Inefficiency of LLC Management Policies

Next, we discuss why existing hardware-based cache replacement policies remain ineffective for serverless workloads, even with their built-in stream- and thrash-resistant mechanisms [37], [38]. Streaming behavior refers to a memory access pattern in which a program walks through a large data structure such that each cache block is accessed once and never reused before eviction. This pattern is common in real workloads such as I/O streaming [39], database column scans in analytical queries [40], and garbage-collection passes in managed runtimes [41].

Under a standard least-recently-used (LRU) policy, streaming behavior introduces fresh cache blocks that, due to LRU's recency bias, remain in the cache for a long time despite never being reused. These blocks effectively outlive more valuable data and reduce effective cache capacity. To address this, prior work has proposed replacement mechanisms with stream-resistant properties, including RRIP [37], which has been shown in Intel processors [38], [42]. RRIP assigns newly inserted cache blocks the oldest age so they remain the first replacement candidate until they are accessed again.

For example, consider a program with a streaming access pattern that touches one block in every LLC set. If the program performs a second scan, it will only displace the blocks inserted during the first scan, rather than evicting other useful data. As a result, the program can stream through a working set several times larger than the LLC without evicting useful lines residing in the cache.

To examine the streaming access pattern of serverless workloads, we run an experiment where we gradually reduce the LLC available to each function and measure both latency and LLC occupancy using Intel's pqos tool [43]. Figure 3 shows the result for a video analytics function. As shown, the function occupies over 16MB of LLC, yet its performance remains unchanged when only 3MB is available. In other words, over 13MB of the LLC is polluted by accesses with poor locality. Similar streaming access patterns are observed in other functions as in Table II.

With the stream-resistance properties described above, we would expect existing hardware replacement policies to prevent low-locality data from displacing more useful cache blocks. However, stream resistance has a fundamental limit: with each insertion of low-locality data, the ages of all other blocks increase as well. Eventually, after enough insertions, an older but useful block reaches the same age as the newly inserted streaming data and is evicted. This reflects an inherent tradeoff between recency and stream resistance. Prior work [37] estimates that the limit of RRIP stream resistance is roughly $< 3\times$ LLC size. In addition, prior studies [42] report that Intel processors dynamically switch between recency-biased and stream/thrash-resistant modes depending on observed patterns, which may further reduce their ability to protect useful data under mixed or irregular access patterns.

Many serverless functions exhibit streaming behavior. When several such functions run between two invocations of another function, due to long inter-arrival times, they can generate

TABLE I
CACHE MPKI AND P50 SLOWDOWN UNDER DIFFERENT THRASHING. FUNCTIONS INSENSITIVE TO COLD CACHE ARE NOT SHOWN.

Condition	**AUTH** (33μs)				**PROD** (37μs)				**GEO** (60μs)				**PAY** (89μs)				**PRANK** (193μs)				**AES** (207μs)			
	L1	L2	LLC	Slow.	L1	L2	LLC	Slow.	L1	L2	LLC	Slow.	L1	L2	LLC	Slow.	L1	L2	LLC	Slow.	L1	L2	LLC	Slow.
Warm cache	83.2	25.4	0.6	1.00	60.5	16.8	1	1.00	55.4	15.4	0.7	1.00	103.9	43.2	4.1	1.00	75.4	23.4	1	1.00	48.8	14.9	0.4	1.00
Cold L1	82.9	25.7	0.6	1.00	61.7	17.8	1	1.00	54.9	15.9	0.7	1.00	95.7	41.3	4.3	1.00	75.4	23.8	1	1.00	48.7	15	0.4	1.00
Cold L1∼L2	83.2	27.6	0.7	1.00	60.9	19.5	1.1	1.03	53.9	17.1	0.8	1.00	101.2	43.1	4.6	1.06	75.6	25	1.1	1.01	48.7	16.1	0.4	1.00
Cold L1∼LLC-D	86.2	28.2	7	1.30	64.7	20.9	3.3	1.30	55.3	17.7	2.6	1.10	105.8	44.9	12.2	1.25	78.7	26	7.2	1.26	51.5	16.5	4.7	1.24
Cold L1∼LLC-I	83.7	27.5	10.2	1.15	64.4	20.6	13	1.65	53	16.8	9.6	1.35	102.2	43.3	12.1	1.12	76.3	25.3	8.6	1.10	49.2	16	5.7	1.02
Cold L1∼LLC	86.2	27.9	15.4	1.39	63.7	20	14.1	1.81	55.5	17.5	11.2	1.40	102.5	43.3	18.8	1.29	79.1	26	14.1	1.34	51.3	16.3	9.5	1.26

aggregate cache traffic many times larger than the LLC capacity, exceeding the limitations of LLC stream-resistance.

Serverless functions are often minimalistic to maximize scalability and parallelism. As a result, each function implements a small, self-contained unit of computation. This stateless design naturally leads to streaming access patterns, since data processed in one invocation is discarded and never reused in the next. L1 and L2 caches filter any immediate temporal reuse within their small capacities (majority of accesses), so from the perspective of the LLC the workload often appears purely streaming: each cache line is touched once and never revisited. This behavior is evident in our video analytics function: each invocation opens a different video file, reads its frames into heap buffers, preprocesses them into tensors, and feeds them through the model. These temporary buffers and tensors are specific to one invocation and become useless afterward, resulting in substantial LLC pollution, as shown in Figure 3. MXFaaS [44] confirms that on average there's 8.7MB per-invocation private data.

Even without the streaming behavior, invocations of multiple functions can produce a combined working set far larger than the LLC's stream-resistance limit. In this case, the LLC cannot preserve useful blocks across invocations, causing functions to begin with a cold LLC and effectively lose its performance benefits.

Even more advanced LLC replacement policies fail to preserve useful cache blocks across serverless invocations. We collect traces of serverless functions and simulate three state-of-the-art policies [45]–[47] in ChampSim [48]. After a training period, we invoke `AUTH-PY`, followed by N other functions, and measure the fraction of `AUTH-PY`'s reusable LLC blocks that remain. As shown in Figure 4, although Belady's MIN preserves all reusable state, practical policies lose over 99% of it after three invocations.

State-of-the-art replacement policies fail in serverless for several reasons. First, although many policies [46], [47], [49] strive to emulate Belady's MIN [50], they have a short and narrow vision due to hardware budget, limiting their capabilities to capture invocation patterns in serverless. Second, most policies [45]–[47], [49], [51]–[53] rely on additional predictors to learn programs' memory access patterns, which require warmup and are susceptible to pollution from interleaved function executions. Third, most policies [45]–[47], [49], [51] classify each memory access into different reuse distances. Yet, memory accesses of serverless functions exhibit different reuse distances at different phases: short reuse distance within each invocation, and long reuse distance after each invocation.

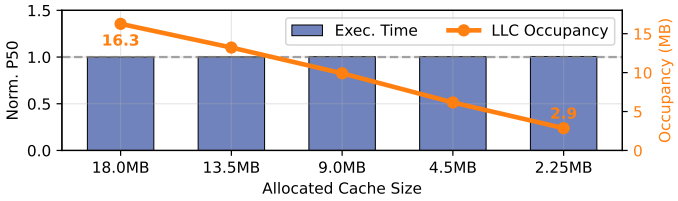


Fig. 3. Estimated LLC occupancy and execution time of `VIDEO-ANALYTICS` under different LLC constraints. Performance remains the same as the LLC occupancy is reduced, indicating LLC is polluted by poor-locality accesses.

TABLE II
BREAKDOWN OF LLC OCCUPANCY (MB) INTO REQUIRED WORKING SET (WS) AND STREAMING POLLUTION (SP) FOR ALL FUNCTIONS.

Fn.	AUT	PRO	GEO	PAY	PRA	AES	RNN	VID	ZLI	IMG	CNN	WCN
WS	2.3	2.3	1.7	9.0	2.3	1.7	1.7	1.1	0.3	0.6	1.1	0.6
SP	0.6	5.3	5.5	1.1	2.5	2.5	3.2	13.6	12.8	15.2	15.4	16.1

C. Inefficiency of Existing Partitioning Systems

Although existing LLC management systems [25]–[33] avoid most cache contention among concurrently running programs, they do not preserve cache states for inactive functions, because they reassign the cache space of inactive programs to maximize cache utilization. Therefore, serverless functions will still experience a cold LLC at each invocation under these LLC management systems.

Adapting existing cache management systems for serverless computing is challenging. A naive adaptation is to reserve cache

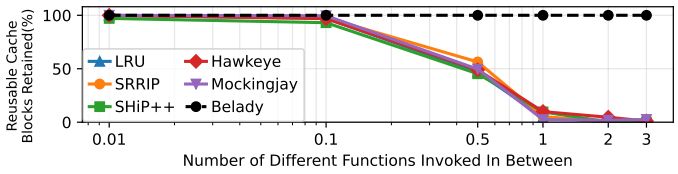


Fig. 4. Existing cache replacement policies fail to preserve function's cache state across serverless invocations.

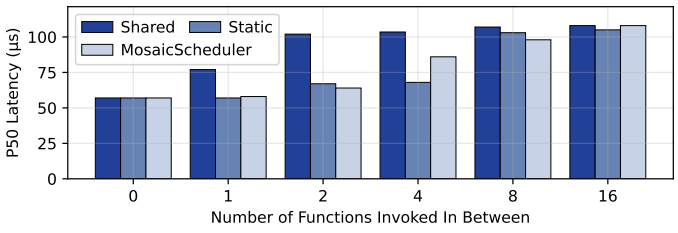


Fig. 5. `GEO` execution time across LLC management schemes under increasing co-location density. Static partitioning and MosaicScheduler exhibit poor scalability in high-density serverless environments.

$(f_1, f_2, \ldots, f_k)^N$

(a) Thrashing Invocation Pattern ($k >$ #max partitions)

$(f_1, f_2, f_3, f_4, \ldots, f_k)^1$

(b) Streaming Invocation Pattern ($k \to \infty$)

Fig. 6. Invocation patterns of serverless functions.

space for all services, whether active or not. As in Figure 5, this solution does not scale with the high-density co-location of serverless functions for two reasons. First, existing cache partitioning techniques (e.g., cache way and set partition) incur significant overhead due to reduced LLC associativity and L2 capacity (more in §III). Second, with more and more functions scheduled onto the same CPU, LLC capacity will become a bottleneck and each function will be assigned less than it needs.

Stojkovic et al. designed Mosaic [13] to overcome both challenges and partitioned cache for serverless computing. They first proposed a new cache design called cache chunks to allow fine-grained yet low-cost cache partitioning. To avoid the cache capacity bottleneck, they designed a new cache allocation strategy to keep track of **the least-recently used (LRU) services** and evict them at cache space shortage.

Although Mosaic shows strong potential on future serverless-specialized CPUs, it does not scale well on current hardware (Figure 5). Beyond the lack of fine-grained, low-overhead cache partitioning, its LRU-based function replacement policy fails to preserve function state under streaming and thrashing invocation patterns.

- **Thrashing invocation patterns**: Figure 6a presents a cyclic invocation pattern of k functions that repeats N times. When k exceeds the number of partitions available, LRU will always evict each function before its next invocation, providing no cache hit at invocation level.
- **Streaming invocation patterns**: Figure 6b presents a streaming invocation pattern, where each function is invoked only once and never again, or functions with little cross-invocation cache state reuse are invoked repeatedly (more in §IV-A). Under LRU, streaming invocations will evict the cache space of other functions, reducing cache hit at invocation level.

Further, while existing cache partitioning improves allocation efficiency by minimizing per-application footprints, it largely ignores shared cache state across applications. This leads to inefficiency in serverless environments, where functions often share common language runtimes [54], [55]. Although Mosaic [13] recognizes this, its solution depends on unavailable hardware extensions.

III. HarmoniCa's Slice-Aware Partitioning

This section presents a novel slice-aware cache partitioning approach. Leveraging LLC slice architecture, it achieves finer-grained partitions while minimizing negative impacts on the L2 cache.

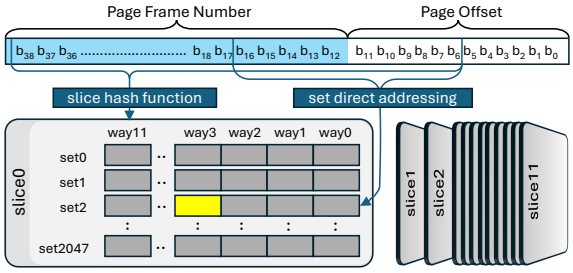


Fig. 7. LLC organization of a 12-core Ice Lake CPU.

A. Organization of Modern LLCs

Figure 7 shows the organization and addressing of the last-level cache in a typical Intel CPU. Modern Intel CPUs divide the LLC into multiple slices to improve the throughput, typically one slice per core [56]. Each physical address is mapped to a slice using a complex hash function that aims to balance the traffic across slices. Within each slice, cache sets are indexed directly using specific physical-address bits, and each set contains W cache ways, where W is the associativity of the LLC.

B. Limitations of Current Partitioning Mechanisms

One common technique is cache way partitioning using Intel Cache Allocation Technology (CAT) [26], [57], [58], which assigns different cache ways to different classes of service. The minimum granularity, therefore, is one cache way across the entire LLC, typically between 1/8 to 1/20 of the total LLC on modern Intel processors. On our Ice Lake machine, this is approximately 1.5 MB. This is too coarse-grained for serverless workloads, as some of the functions we evaluate use only a few hundred kilobytes of LLC. Moreover, way partitioning reduces the effective associativity of each partition. As the number of partitions increases, each receives fewer ways, which can significantly degrade performance for applications that rely on higher associativity.

Another common technique is cache set partitioning [27], [58]–[60], which partitions the LLC by assigning each program memory pages whose addresses map to disjoint cache sets. The advantage of this approach is that it can be implemented entirely in software, since the OS can control physical page allocation without requiring hardware support such as Intel CAT.

However, set partitioning has several drawbacks. First, supporting unmodified applications limits which address bits we can control. The OS can only manipulate bits in the physical frame number that contribute to the LLC set index (e.g., bits 12-16 in Figure 7). On our machine, this gives us control over only 5 bits, yielding a granularity of 576 KB. While this is better than way partitioning, it is still too coarse for serverless functions that require only a few hundred kilobytes of LLC.

Second, set partitioning introduces significant overhead on partitioned functions because the same set index bits also index into the L2 cache. On our machine, 4 of the 5 bits are used for L2 set indexing. As a result, allocating a function 1/32 of the LLC (576 KB) effectively reduces its available L2 capacity to

$$f^0_{\text{slice}}(x) = x_6 \oplus x_8 \oplus x_9 \oplus x_{10} \oplus x_{14} \oplus x_{15} \oplus x_{17} \oplus x_{18} \oplus x_{20} \oplus x_{23} \oplus x_{27} \oplus x_{30} \oplus x_{31} \oplus x_{34} \oplus x_{36}$$

$$f^1_{\text{slice}}(x) = x_6 \oplus x_7 \oplus x_8 \oplus x_{12} \oplus x_{16} \oplus x_{17} \oplus x_{20} \oplus x_{21} \oplus x_{22} \oplus x_{23} \oplus x_{24} \oplus x_{25} \oplus x_{26} \oplus x_{28} \oplus x_{30} \oplus x_{33} \oplus x_{35}$$

Fig. 8. Cache slice indexing function on our Ice Lake processor. x_i shows bit i of the address. We show only the lowest two (linear) index bits useful for cache slice partitioning.

1/192 of the total L2 (∼80 KB). This dramatic and unbalanced reduction in L2 capacity can severely degrade performance.

C. Cache Slice Indexing Function

To address the limitations of set partitioning discussed above, HarmoniCa extends partitioning to also take advantage of the sliced structure of Intel LLCs. We, therefore, first explain how physical addresses map to slices.

We reverse-engineered the cache slice indexing function of a 12-core Ice Lake machine, using the method from prior work [61]. Figure 8 shows the two lower bits of the resulting index function. We observe that these bits are computed as an XOR chain of some of the physical address bits that include both the page offset and the page frame number. At first glance, this appears to make slice partitioning infeasible, since slice selection depends on page offset bits that we cannot restrict without modifying applications. However, in the following subsection we show that XOR's linearity over GF(2) allows us to construct slice-aware partitions without restricting page offsets. We omit the remaining slice-index bits, which involve non-linear functions and therefore cannot be leveraged by our partitioning scheme.

D. Slice-Aware Cache Partitioning

We make the key observation that, even though lower address bits participate in the cache slice index, we can still improve our partitioning granularity without controlling those lower bits. The core insight is that while we cannot force two physical pages to get mapped to disjoint cache slices, we can guarantee that no two addresses from two physical pages get mapped to the same $\langle set \mid slice \rangle$ tuple, which is enough for achieving partitioning.

To that end, we use the physical frame number (PFN) part of the slice indexing function (the highlighted parts in Figure 8) as the slice partition ID (SID). For arbitrary addresses from two physical pages with different slice partition IDs, there are two possible scenarios:

- ▶ They have *different* page offset (PO) bits $x_6, x_7, ...x_{11}$. In this case, the two addresses do not conflict because they are mapped to different cache sets.
- ▶ They have the *same* page offset (PO) bits $x_6, x_7, ...x_{11}$. In this case, the two addresses are guaranteed to not conflict either because two addresses with different SIDs and same page offset are guaranteed to map to different slices. This comes from the linearity of the slice indexing function. Formally, if $\text{SID}_1 \neq \hat{\text{SID}}_1$, then we can guarantee $\text{SID}_1 \oplus f(\text{PO}) \neq \hat{\text{SID}}_1 \oplus f(\text{PO})$.

Figure 9 illustrates this partitioning. With two linear bits of slice hashing functions, there are 4 possible slice IDs, dividing all physical pages into 4 colors. For addresses of different colors that are mapped to the same cache set, they are mapped to different cache slice groups, avoiding cache conflicts.

slice partition ID for the 12-core Ice Lake CPU	Color0	Color1	Color2	Color3
ID_0=b_{36}^b_{34}^b_{31}^b_{30}^b_{27}^b_{23}^b_{20}^b_{18}^b_{17}^b_{15}^b_{14}	0	1	0	1
ID_1=b_{35}^b_{33}^b_{30}^b_{28}^b_{26}^b_{25}^b_{24}^b_{23}^b_{22}^b_{21}^b_{20} ^b_{17}^b_{16}^b_{12}	0	0	1	1

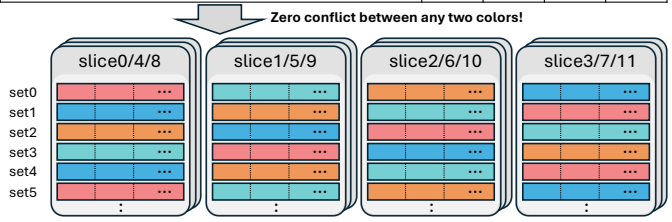


Fig. 9. Illustration of HarmoniCa's slice partitioning.

This cache-slice partitioning technique does not degrade performance by altering the distribution of addresses across slices. Because page-offset bits still determine the slice index, each partitioned program continues to access all cache slices, preserving the available LLC bandwidth. Unlike way partitioning, it does not reduce LLC associativity, because each partition still has access to all cache ways. Because cache-slice partitioning does not partition the address bits used for L2 indexing (e.g., $x_{12} \ldots x_6$), as it can achieve the same partitioning effect using higher bits, each partition retains full L2 capacity, which is essential given L2's performance impact (§III-E). With traditional set partitioning, we can create only two partitions without reducing the effective L2 capacity; with slice-aware partitioning, we can increase this to eight.

Further, this cache slice partitioning technique works across all recent Intel processors, as it relies on only the following assumptions on the cache indexing function: (1) At least one bit of the slice index is computed using a linear function (e.g., an XOR chain) over page–offset (PO) and page–frame–number (PFN) bits. (2) Any page–offset bit used in the slice–index hash also appears in the set–index computation.

Both assumptions hold for all recent Intel CPUs. According to a recent work [62], nearly all Intel client and server CPUs, regardless of the number of cores (from 2 to 24) and the microarchitecture (from Broadwell to Meteor Lake), have at least one linear bit in the slice indexing function (at most 4). The only exception is an old Broadwell CPU with 18 cores released in 2016.

E. Comparison with Existing Techniques

To demonstrate the effectiveness of our slice partitioning compared to existing techniques, we use the `AUTH-PY` benchmark as a case study. We progressively reduce the LLC available to this function under different partitioning mechanisms and measure the resulting slowdown. Figure 10 shows the results.

Way partitioning incurs 43.8% overhead when the LLC is reduced to roughly 2 MB, primarily because it sharply lowers effective associativity. Set partitioning performs even worse, with a 62.5% overhead; performance-counter analysis shows that this degradation is caused by the reduction in effective L2

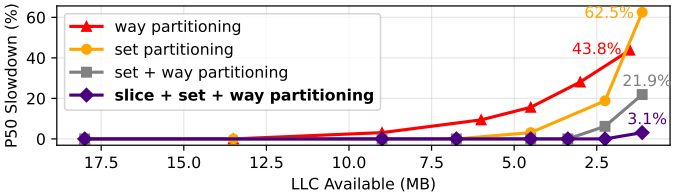


Fig. 10. HarmoniCa's slice-aware partitioning reduces the overhead on AUTH-PY from 21.9% to 3.1% by alleviating bottlenecks on LLC associativity and L2 capacity.

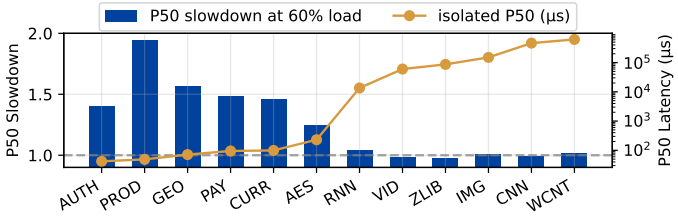


Fig. 11. Functions with longer execution time in general are less sensitive to cold cache states. Functions longer than 10ms don't experience any slowdown.

capacity caused by set partitioning. Combining way and set partitioning offers only modest improvement, but still results in 21.9% overhead because it continues to restrict the L2 capacity available to the function.

In contrast, slice partitioning alleviates pressure on both LLC associativity and L2 capacity, reducing the slowdown to just 3.1%—a 7× reduction in partitioning overhead relative to existing techniques. Importantly, slice partitioning also improves the partitioning granularity and allows more partitions. On our Ice Lake machine, compared to set+way partitioning, our slice-aware partitioning increases the number of supported partitions from 384 to 1536 and improves the partition granularity from 48 KB to 12 KB.

IV. Principles for Efficient LLC Management

With cache-slice partitioning at our disposal, we now discuss how to apply it effectively to address LLC inefficiencies in serverless.

A. Stream- and Thrash-Resistant Allocation

To overcome the limitations of static partitioning and LRU function replacement policy, we propose an allocation strategy that is both stream- and thrash-resistant.

Stream-resistance. We define a streaming invocation pattern as one in which some functions that benefit little from preserved cache state may evict the useful state of others, degrading overall performance. We identify two main sources of such patterns.

First, we observe that functions with longer execution time are insensitive to cold LLC and gain minimal benefit from cache preservation. Figure 11 quantifies each function's sensitivity by measuring slowdown under cold-cache conditions (using a setup similar to Figure 1). We observe that sensitivity decreases with execution time: functions longer than 10ms experience negligible slowdown, as cache warm-up costs are amortized over their relatively longer runtime.

Second, infrequently invoked functions benefit little from cache state preserved across invocations. The analysis of an Azure serverless trace [18] shows that 45% of applications are invoked at most once per hour, and 81% at most once per minute. Overall, these 81% of applications account for only 0.4% of total invocations. Allocating cache to such functions not only yields limited benefit, but also leaves cache space idle for long periods between invocations, preventing cache space allocation for frequently invoked functions.

To prevent streaming invocation patterns from evicting useful functions, functions that are insensitive to cold LLC or are infrequently invoked must be excluded from cache allocation. To learn each function's sensitivity to cold LLC, we perform online/offline learning (described in §V-A).

Thrash-resistance. Thrashing invocation patterns occur when the number of repeatedly invoked functions exceeds cache capacity, causing cyclic evictions that eliminate cross-invocation cache reuse.

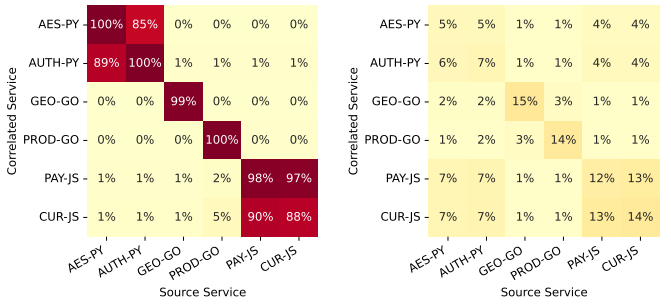
To provide thrash-resistance, we must preserve cache space only for a subset of functions whose combined cache demand fits within the cache capacity and partitioning limits. Other functions might be given temporary space while active, but their allocations must be reclaimed and reassigned when they finish execution and idle.

B. Shared Cache Partition

We observe that across instances there are many functions that access identical physical memory pages, leading to many shared cache states. In Docker and gVisor, containers that share an OverlayFS layer (e.g., official Python image) reuse the same page-cache entries. In VMs and microVMs, the hypervisor may map identical pages to the same physical frame through memory deduplication [63] or snapshot-based sharing [64]–[66]. Until a copy-on-write event occurs, all instances reading these pages access the same physical addresses, leading to shared cache states.

To quantify the amount of potentially shared cache states between instances, we use Intel PT [67] to trace all the instructions executed by each instance, and use Perf-mem [68] to sample all the data accessed by each instance for a sufficiently long interval. For each pair of instances, we compare their instruction and data traces and compute the fraction of memory accesses that target the same physical addresses. Figure 12 shows the results as heatmaps, where darker colors indicate a higher degree of cache state sharing.

As shown in Figure 12a, instruction cache states are commonly shared among functions. Nearly all instruction cache states are shared between instances of the same function. Even between instances of different functions, a large portion of instruction cache is shared depending on their execution environment (e.g., Python or Go runtimes). For interpreted languages (e.g., AUTH-PY and AES-PY), over 80% of instruction cache is shared because both functions execute the same Python interpreter and common libraries like gRPC. For languages with JIT compilation (e.g., PAY-JS



(a) Instruction cache (b) Data cache

Fig. 12. Cache line sharing across serverless instances. Over 85% instruction cache lines are shared between instances of the same function or language runtime.

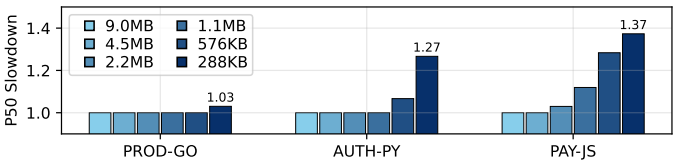


Fig. 13. Different functions need different LLC sizes to achieve optimal performance.

and CUR-JS), most instruction cache is also shared, likely because most executed instructions originate from the shared language runtime rather than the per-function JIT-compiled code. However, for compiled languages (e.g. GEO-GO and PROD-GO), different functions share less than 1% instruction cache, as different functions execute different compiled binaries. Further, data cache state sharing is less common because most data accesses target private heap and stack regions. This is consistent with Table II, which shows that each invocation accesses many temporary data buffers that are neither shared across invocations nor across instances.

To use LLC efficiently, we must avoid over-allocating cache space for shared cache states. If each instance is allocated cache independently based on its own needs, the shared cache states will be redundantly reserved multiple times. To prevent this, we differentiate between shared and private cache states and only allocate once for shared cache states (§V-B).

C. Customized Cache Partition Size

We observe that different serverless functions need different amounts of LLC to be performant. To compare the cache needs of functions, we measure the latency of functions with gradually reduced LLC capacity. As can be observed in Figure 13, different functions need various LLC capacities for the best performance. To achieve 95% performance, PAY-JS needs as large as 2.2 MB LLC, while PROD-GO only needs 288 KB. This difference in working set sizes arises from the function workload as well as from the language runtime.

To use LLC efficiently, we must tailor the cache partition size to each serverless function. Assigning the same amount of LLC to all functions wastes cache for functions with small working sets while degrading performance of functions with large working sets. To prevent this, we learn the minimum LLC needed by each function to achieve best performance through online or offline learning (§V-A).

Although inputs can affect function behavior, they do not change the optimal cache configuration for any function in our evaluation. Most functions keep similar control flow and working sets across inputs. For AES and PAGERANK, larger inputs only increase loop iterations, not LLC demand: AES streams over the input once at 16B granularity, while PAGERANK's per-invocation graph nodes fit in the unpartitioned L2. With larger graphs, PAGERANK becomes insensitive to cold LLC and can remain in the common cache pool. If a function does exhibit input-dependent cache demand, HarmoniCa can detect this by monitoring function latencies, and maintain function instances with separate cache configurations for invocations of different input-size ranges.

V. HarmoniCa

Based on the three strategies, we introduce HarmoniCa, a last-level cache management framework that orchestrates LLC resources among serverless functions. For each newly deployed function, HarmoniCa operates in two phases. First, HarmoniCa learns the function's cache behavior through online or offline learning (§V-A). This includes its sensitivity to cold LLC, shared cache states, and the minimum LLC capacity required for optimal performance.

Second, HarmoniCa customizes the LLC allocation for each function based on its cache behavior and invocation frequency (§V-B). Functions insensitive to cold LLC are placed in a common cache pool, whose size is dynamically adjusted based on the services inside. For cache-sensitive functions, HarmoniCa first identifies shared cache states and allocates a single partition. Next, HarmoniCa reserves an additional partition that provides the minimum LLC for the function's non-shared data. When LLC becomes scarce, HarmoniCa prioritizes services with higher invocation rates, higher sensitivity to cold LLC, and lower cache needs, while moving the remaining services back to the common pool.

A. Function Characterization

For each serverless function, two cache properties are learned: performance sensitivity to a cold LLC and minimum LLC size to deliver good performance, because both are important to achieve efficient LLC partitioning as discussed in Section IV. To measure the sensitivity to a cold LLC, two latencies are measured: (1) lat_{cold} when the function shares LLC with other functions, and (2) lat_{warm} when the function is allocated an exclusive cache partition of a maximum size.

We use $sensitivity = lat_{cold}/lat_{warm}$ to quantify the cold start sensitivity of a serverless function. The larger the sensitivity, the more a function benefits from an exclusive cache partition. A sensitivity close to 1 indicates that a function doesn't suffer from a cold LLC and won't benefit from an exclusive cache partition.

If a function demonstrates high sensitivity to cold LLC, HarmoniCa continues learning the cache needs of the function. First, we identify sharing of instruction cache states between

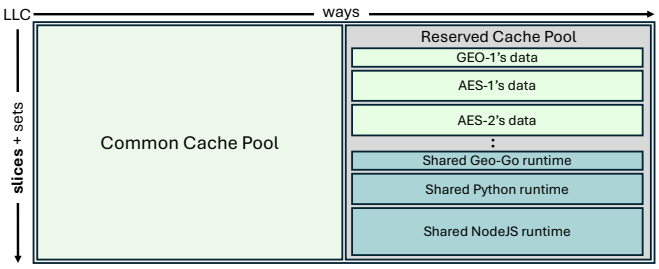


Fig. 14. An example of HarmoniCa LLC allocation scheme. Insensitive functions are assigned to a dynamically adjusted common pool while we allocate exclusive shared and private partitions for the sensitive functions.

the target function and other functions. For the most accurate result, one needs to trace the instruction addresses executed by each function or compare the physical pages mapped for each function, which is both time- and memory-consuming. To avoid these overheads, we observe that for most serverless functions, the binary in the container's default command contains most of the executed instructions. Therefore, we infer that two functions share many instruction cache states if their default command binaries are the same file in the same docker layer. This heuristic correctly identifies all pairs of functions with over 85% instruction cache sharing in Figure 12.

Finally, HarmoniCa measures, separately for the shared instructions and the private data, the minimum LLC allocation needed to deliver good performance. A naive approach is to try all possible LLC sizes and choose the smallest one without exceeding a performance threshold, which is time-consuming given the fine-grained partitioning technique we propose. To reduce the test time, we observe that the performance of serverless functions degrades monotonically with reduced LLC size. Therefore, we perform a binary search over the ordered set of candidate LLC sizes. This reduces the search time from $O(n)$ to $O(logn)$, where n is the number of LLC sizes. In most cases, the minimum LLC size can be identified within 1000 invocations.

To further accelerate learning of function cache needs, HarmoniCa supports offline learning. Given a set of representative inputs provided by the programmer, HarmoniCa first learns the sensitivity to cold LLC with controlled colocated workloads (e.g., using stressors such as stress-ng [69] or a set of test functions). Then, HarmoniCa learns the cache needs similar to online learning, but with an increased invocation rate to collect samples faster. Although the offline learning environment differs from the online deployment in invocation rate and colocated workloads, we show that the cache behavior of each function remains the same (§VIII-C), making the knowledge learned offline directly transferable to an online setting.

B. Priority-Based LLC Allocation

The overall allocation scheme is shown in Figure 14. To begin with, all functions insensitive to cold LLC (with a sensitivity close to 1) are assigned to a common cache pool. This common pool is partitioned from the rest of the cache using way-partitioning, allowing us to adjust its size dynamically as needed, without significant reallocation costs.

Next, to maximize the benefits of the reserved cache pool, HarmoniCa prioritizes the cache demands of services based on three factors: (1) invocation rate, (2) sensitivity to cold LLC, and (3) LLC demands. HarmoniCa keeps satisfying the LLC demands for services with higher priorities until there is no more free cache in the reserved pool. For each service, HarmoniCa allocates LLC separately for shared instruction and private data cache states, avoiding redundant reservations. Then, all untended services are assigned to the common pool.

When multiple new functions have not yet been characterized, HarmoniCa prioritizes their learning based on the same factors as cache allocation: invocation rate and expected cold LLC sensitivity. While invocation rate can be measured directly, HarmoniCa estimates cold LLC sensitivity using function latency, because short-running functions are generally more sensitive to cold LLC as in Figure 11. Learning is performed in parallel when resources permit, and these heuristics prioritize functions with the highest expected benefit.

To partition the LLC, HarmoniCa combines cache slice, set, and way partitioning, as illustrated in Figure 14. Cache way partitioning is best suited to partition between the common and reserved cache pool, because it incurs little overhead when used for such coarse-grained partitioning (see Figure 10). Within the reserved pool, HarmoniCa applies slice and set partitioning to achieve fine-grained cache allocation. To prevent bottlenecks caused by limited L2 capacity, HarmoniCa performs slice partitioning before set partitioning, ensuring each service can utilize as much L2 capacity as possible. Services in the common cache pool are not partitioned by slices or sets and can access all cache slices and sets.

VI. Implementation

We extend Linux with page coloring to support cache slice/set partitioning (§VI-A), which causes no overhead to serverless workloads. We then implement HarmoniCa using Python and Bash (§VI-B). No modifications are required to the serverless functions or hardware.

A. Page coloring in Linux

We implement page coloring in Linux 6.12 with around 1k lines of code. The code is publicly available at https://doi.org/10.5281/zenodo.21548955.

Interfaces. We add two syscalls: `set_color` and `get_color`, which change and query the page color mask of a process, respectively. A member field is added to `task_struct` to record the page color mask of each process. Similar to `taskset`, we also implemented a command-line utility called `colorset` to assign or change the page colors of a process.

Color-aware buddy allocator. Similar to prior work [28], Linux's buddy allocator is extended to honor the page color requirements when allocating physical pages. Instead of a single linked list of free pages, the buddy allocator keeps track of multiple free lists, one for each color. At a page request, the

buddy allocator selects pages of allowed colors in a round-robin manner. When a page is freed, the buddy allocator returns the page to the corresponding free list. We observe that the customized memory allocator doesn't affect the performance of serverless functions.

Page recoloring. When a process's cache partition changes, we use a Linux workqueue [70] to handle page migration and page-table updates in the background. A workqueue thread scans all physical pages of the target process to identify the minimal set that must be migrated, moves those pages to newly allocated pages with the desired colors, and updates the corresponding page-table entries. In our implementation, migrating a short service (sub-1ms execution time) typically takes around 50 ms, and over 100 ms for larger functions. This overhead is acceptable because recoloring events are rare and function invocation rate is low. Still, it can be further reduced through lazy migration [71] or migrating only hot pages [72]. For all functions, performance stabilizes within five invocations after a page recoloring.

Shared page identification. To support shared cache partitions, we assign separate page color masks to shared and private pages, using Linux's `folio_likely_mapped_shared` to classify pages into shared and private. This correctly classifies over 99% instruction and data accesses into shared or private cache partitions.

Deployment constraints. We prioritize allocation success and NUMA locality over exact page coloring to avoid failures and slowdowns. Although coloring can restrict huge pages, we observe no overhead due to functions' small memory footprints.

B. HarmoniCa

We implement HarmoniCa with 3K lines of Python and Bash scripts. To begin with, HarmoniCa takes in an optional list of files containing function cache behavior learned offline. For functions without offline knowledge, HarmoniCa performs online learning. To collect service metrics, HarmoniCa queries a Zipkin [73] server which traces the execution of all services using an OpenTelemetry plugin [74]. Next, HarmoniCa adjusts the LLC scheme based on the knowledge and collected metrics. Finally, HarmoniCa deploys the LLC scheme onto the worker server by adjusting cache way partitioning using pqos [43], slice and set partitioning using colorset (§VI-A), and core pinning using Linux cgroup.

VII. Evaluation Setup

Testbeds. We evaluate HarmoniCa on an Intel Xeon Silver 4310 processor with 12 cores, 80 KiB of private L1, 1.25 MiB private L2 per core, and an 18 MiB shared LLC (12-way, non-inclusive). The server has 136 GB of DDR4 memory across five channels. To avoid interference from client programs, we place all clients on a separate machine with a similar configuration. To avoid network latency impact, the two machines are co-located with a round-trip time of 100 µs. For stable performance, simultaneous multithreading (SMT) is disabled and the core frequency is fixed at 2.10GHz.

Serverless functions. We use 12 functions from two serverless benchmarks [36], [75]. These functions cover common use cases such as web services, API backends, file/image processing, and ML inference. We randomize function inputs to match real-world user inputs. For each serverless function, we create 3 instances to mimic the auto-scaling behavior of serverless orchestrators.

Invocation trace. Similar to prior work [13], [44], we drive the functions with real-world invocation traces released by Microsoft Azure [18]. From the 7-day trace, we first select a 10-minute window that exhibits a representative (P50) invocation load. We then randomly map the trace functions in this interval to our benchmark functions, after filtering them by invocation rate. This filtering step is necessary to ensure that the resulting workload matches the capacity of our server machine, since the Azure trace is sampled across Azure's entire infrastructure rather than a single machine. Finally, given the per-minute invocation rate of each function, we generate its arrival timestamps using a Poisson distribution like prior research on serverless systems [13], [22], [44], [76]–[82].

Evaluated designs. We compare with three baseline LLC management schemes: (1) *Shared*: all instances share the last-level cache; (2) *Static*: the LLC is statically partitioned for all instances; (3) *MosaicScheduler*: the LLC is managed by the software support of Mosaic with cache way partitioning. In all three setups, for a fair comparison, cache-sensitive instances are pinned to cores, while cache-insensitive instances can be freely scheduled on the least-loaded cores by the kernel scheduler to avoid CPU bottleneck. To minimize interference from background tasks, all serverless instances run under Linux's real-time scheduler [83] with a fixed priority of 10.

VIII. Evaluation Results

For function performance, we measure the server-side end-to-end latency using gRPC OpenTelemetry plugin [74]. This includes the time from when the server receives a request until it responds.

A. Performance Impact

To understand how the partitioning mechanism and strategy contribute to the overall improvement, we evaluate HarmoniCa under a 40% system load with 3 variants: (1) *Adv. Static* enhances *Static* partitioning with cache slice partitioning; (2) *HC-Coarse* is only stream-resistant, as it performs coarse-grained way partitioning between streaming invocations and others; (3) *HarmoniCa* is also thrash-resistant and performs fine-grained partitioning. All results include page recoloring overhead and use offline learning for a fair comparison. Online learning overhead is evaluated in Section VIII-C.

Median latency. Figure 15 and Figure 16a show the median latency improvement for all 12 functions. On average, HarmoniCa improves the median latency of cache-sensitive functions by 34.7% over shared cache on average, without notably hurting the performance of cache-insensitive functions (0.1%).

Moreover, each component contributes to the overall improvement. Using `PROD` as an example: slice partitioning

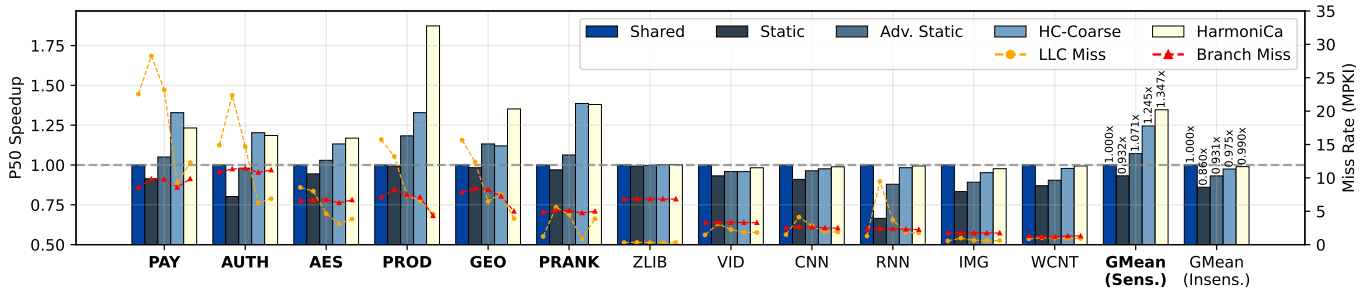
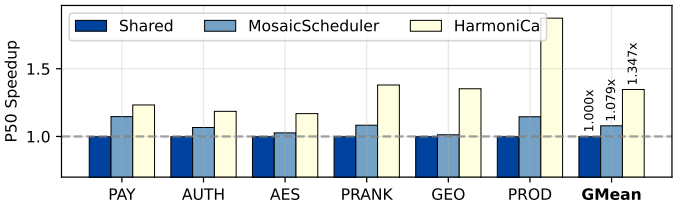
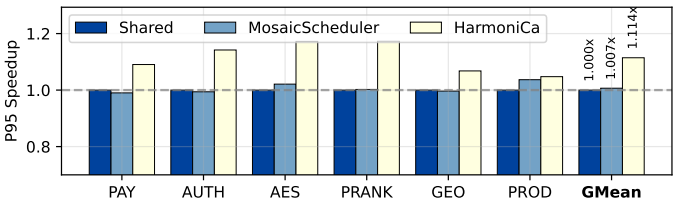


Fig. 15. HarmoniCa P50 improvement. On average, HarmoniCa improves the P50 latency of cache-sensitive functions by 35%, without notably slowing down cache-insensitive functions (0.1%).

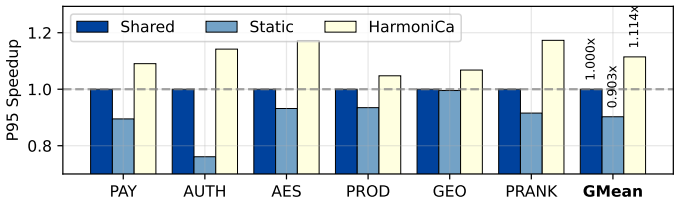


(a) P50 improvement.

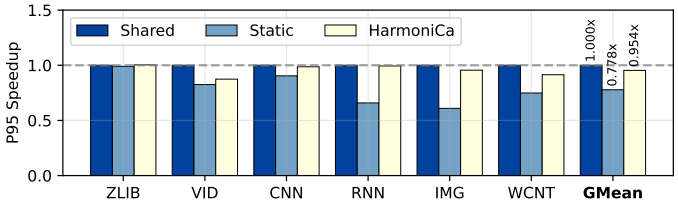


(b) P95 improvement.

Fig. 16. HarmoniCa improvement over MosaicScheduler.



(a) Cache-sensitive functions.



(b) Cache-insensitive functions.

Fig. 17. HarmoniCa improves P95(P99) latency of cache-sensitive functions by 11.4%(7.7%). CPU bottlenecks cause a 4.6%(5.1%) slowdown for cache-insensitive functions.

alleviates reduced L2 capacity and L3 associativity, reducing LLC misses by 45% and improving latency by 19%. With stream-resistant allocation, *HC-Coarse* reduces misses by 59% and improves latency by 15% compared to a shared cache. Finally, fine-grained cache management makes *HarmoniCa* robust to thrashing invocation patterns, further reducing misses by 30% and improving latency by 54%. On average, slice partitioning improves static partitioning by 14%, while *HC-Coarse* and *HarmoniCa* improve shared cache by 25% and 10%.

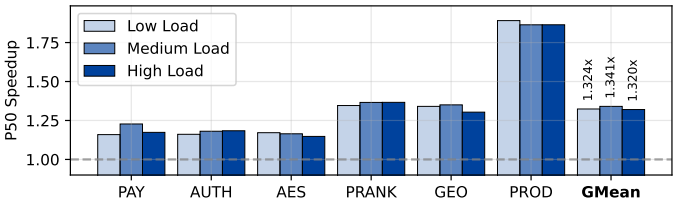
Finally, HarmoniCa outperforms static partitioning by 44.5% and *MosaicScheduler* by 25.8%. While static partitioning preserves cache state for all functions, it severely reduces L2 capacity (to 1/8 or 160KB) and L3 associativity (to 4 ways), incurring significant overhead. Although *MosaicScheduler* provides sufficient cache space per function, its LRU policy is not effective under streaming and thrashing invocation patterns, leaving 70% of invocations with a cold LLC and yielding a 7.9% improvement.

Tail latency. Figure 17 and Figure 16b show the P95 tail latency improvement. HarmoniCa improves the tail latency of cache-sensitive functions by up to 17% on `AES` and 11.4% on average. Notably, despite the lower speedup factor, the absolute tail latency reduction exceeds the median latency reduction for all functions. Because tail latency is affected by many other factors beyond cold cache states, such as I/O interference and kernel interrupts, the relative speedup from HarmoniCa at the tail latency is naturally lower than the improvement observed for the median latency.

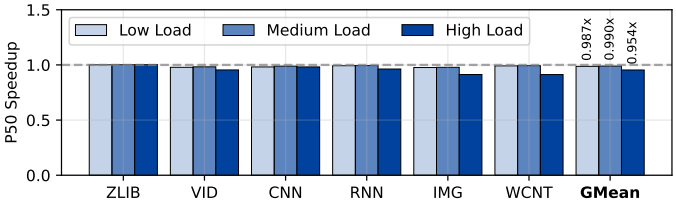
On the other hand, we see a slowdown in the tail latency of cache-insensitive functions (4.6% on average). Rather than an LLC capacity bottleneck, we attribute this slowdown to a CPU bottleneck resulting from HarmoniCa's core allocation scheme. Unlike the baseline where cache-insensitive functions can be scheduled on any of the 12 cores, HarmoniCa constrains them to 9 cores for cache way partitioning, leading to CPU bottlenecks at bursts of invocations. To verify this, we pin all cache-insensitive functions to 9 cores but allow them to use the full LLC. We observe a similar slowdown in the tail latency, confirming the CPU bottleneck. This bottleneck is not inherent to HarmoniCa because cache ways can be assigned to tasks rather than cores through Linux support [84].

Finally, static partitioning significantly worsens tail latency due to reduced L2 capacity and LLC associativity, while *MosaicScheduler* achieves a modest 0.7% improvement with its LRU policy.

Impact on aggregate CPU load. Across all invocations, HarmoniCa slightly increases total CPU time by 1.75% over



(a) Cache-sensitive functions benefit at all loads.



(b) Cache-insensitive functions slightly degrade at higher loads.

Fig. 18. Median latency at different system loads.

the shared-cache baseline, much less than static partitioning's 16.5% increase. This overhead stems from the earlier CPU bottleneck in long, cache-insensitive functions, which task-level way partitioning can avoid [84]. Although HarmoniCa does not significantly reduce aggregate CPU load, it improves the responsiveness of short, latency-sensitive interactive requests, which dominate user-perceived performance.

B. Sensitivity Analysis

Sensitivity to system load. We next study how HarmoniCa performs under different system loads. We maintain the same experiment setup as the main evaluation experiment in Figure 15 and scale the invocation trace to achieve 25%, 40%, and 60% system loads similar to prior works [13], [44], which is representative in cloud to guarantee SLO [85]–[88]. Figure 18 shows the median latency of each function with HarmoniCa normalized to the baseline latency under each system load configuration, respectively.

HarmoniCa consistently improves the performance of all cache-sensitive functions under all system loads. However, cache-insensitive functions degrade with higher loads, with the slowdown increasing to 4.6% at 60% system load. Similar to the tail latency slowdown in Figure 17b, this is due to the CPU bottleneck when HarmoniCa reduces the cores available from 12 to 9.

Sensitivity to number of cache-sensitive services. We now evaluate how HarmoniCa scales as the number of cache-sensitive services requiring cache reservations increases. In this experiment, we keep the overall system load fixed at 50%, using the same cache-insensitive functions as in Figure 15. We then gradually increase the number of Python authentication services `AUTH` deployed on the machine, with each service invoked at an identical rate. Figure 19 reports the average median latency of `AUTH`. In addition to the *baseline* and *HarmoniCa*, we include *HC-noShare*, a variant that ignores cache sharing between services and over-allocates cache space for the shared cache states.

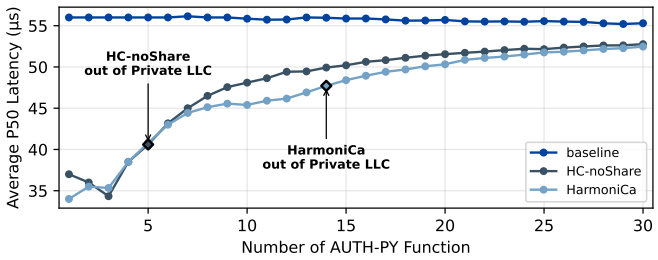


Fig. 19. Median latency at different service densities.

We observe three trends in HarmoniCa. First, when the number of services is small (≤ 3), the median latency remains the lowest at 35 µs, because each service is assigned a dedicated core in addition to benefiting from cache management. Second, as the number of services grows, the latency increases linearly before flattening around 45 µs. This reflects the point where the advantage of exclusive cores disappears and the remaining gains come purely from cache management. Finally, once the number of services exceeds 14, the average latency degrades as an inverse function. At this point, HarmoniCa exhausts the reserved cache pool, forcing additional services to fall back to the common cache pool. 14 services are not the scalability limit of HarmoniCa, but rather an LLC capacity bottleneck from the machine we used (9MB for private LLC).

Although *HC-noShare* provides significant speedup compared to *baseline*, it degrades earlier when the reserved cache pool is exhausted at 5 services. This results from an inefficient use of LLC, where shared cache states are redundantly reserved multiple times, showing the importance of properly managing shared cache states.

Sensitivity to cache sharing configuration. To evaluate under stricter security constraints, we test disabling all sharing versus restricting sharing to functions from the same user. Disabling all sharing drops geomean improvement from 34.1% to 23.9%, while restricting sharing to the same user limits this drop to 29.8%.

Sensitivity to network latency. Experiments with 500μs RTT and injected 1–5ms delays show negligible changes, confirming that server-side LLC effects dominate rather than network latency.

Sensitivity to invocation trace. Across ten traces generated with different seeds and Azure-published trace windows, HarmoniCa achieves stable speedups from 1.31× to 1.39× with 1.34× on average.

C. Cache Learning Analysis

Learning time. We evaluate how fast HarmoniCa learns a function's cache behavior online. In this experiment, we remove `AES` from the knowledge library and measure the time HarmoniCa needs to infer its minimum cache size from scratch. We compare this against a brute-force baseline that gradually shrinks the cache size until the function's performance falls below a predefined threshold. Both methods are tested with the same threshold and sampling setup using the same invocation trace. Figure 20 shows the results.

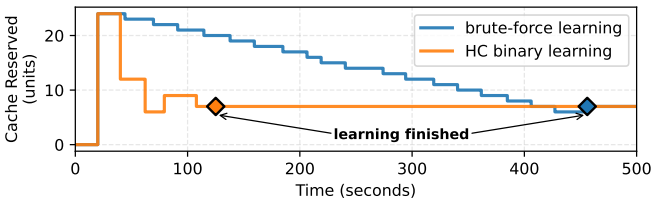


Fig. 20. HarmoniCa learns the optimal cache configuration for AES over 3 times faster than a brute-force search.

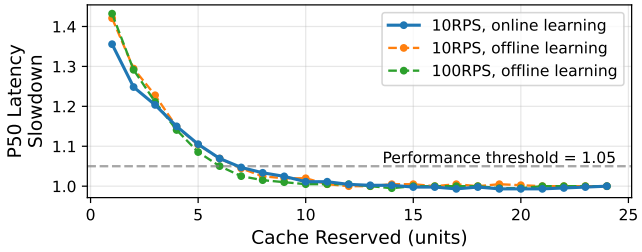


Fig. 21. Faster offline learning with a higher invocation rate closely matches the online learning result on AES.

With binary learning, HarmoniCa significantly reduces the learning time for each cache-sensitive function by over 3× compared to a brute-force learning approach. In total, around 1000 invocations are needed for HarmoniCa to determine the minimum cache size without breaking the performance threshold. To further accelerate learning, we perform offline training with a higher invocation rate so that we collect samples faster. For example, increasing AES to 100 RPS reduces the learning time to under 15 seconds.

We further evaluate HarmoniCa's online learning overhead by repeating Figure 15 without prior knowledge of any function. HarmoniCa learns all 12 functions in under 240s, a negligible fraction of their typical 9-day production lifetime (estimated from Azure trace [18]). Consequently, the geomean P50 improvement for cache-sensitive functions drops only slightly from 1.347× to 1.290×. This minimal impact is due to HarmoniCa prioritizing short, latency-sensitive, and frequently invoked functions (Section V-B), which enables near-optimal scheduling within 120s.

Learning Accuracy. Offline learning may introduce inaccuracy because the co-running workloads and invocation rates differ from the online setting. To quantify this effect, we modify HarmoniCa to measure function performance at every cache configuration, allowing a one-to-one comparison. Figure 21 shows each learning result and how its performance degrades as the reserved cache decreases. For each curve, all latencies are normalized to the minimum latency observed at the highest cache capacity on that curve. Despite the differences in co-running workloads and invocation rate, the offline learning results closely match the online learning result, as all curves in Figure 21 align well with each other.

Further, binary search may introduce inaccuracy as it assumes latency degrades monotonically with LLC allocation. We compare binary search against brute-force search and observe 100% agreement on all functions, with no over- or under-allocation.

IX. Related Work

Cache partitioning. Many works have explored cache partitioning for efficiency or security. Software-only approaches use hardware way partitioning and set partitioning [25]–[28], [30]–[32], [56]–[60], [72], [89]–[96], but they primarily target traditional long-running workloads and offer insufficient granularity for serverless workloads. Although HardHarvest [97] preserves cache states for microservices across invocations, it assumes no over-subscription and relies on way partitioning, which doesn't scale to serverless. Hardware proposals [13], [98], [99] enable finer control but remain unadopted by CPU vendors.

Unlike works [25], [30], [99] that optimize cache management with a clear objective function, HarmoniCa uses serverless-specific heuristics. Incorporating a formal utility model is promising future work.

While several works reverse-engineer Intel's LLC slice-indexing function [61], [62], [100], [101], slice-aware partitioning has received limited attention. Yarom et al. [101] suggested that slice knowledge could improve page coloring without implementation or evaluation. Scolari et al. [28] later proposed slice-aware partitioning, but their technique requires slice selection to depend only on PFN bits, which doesn't hold on all Intel CPUs since Broadwell [62], [100]. In contrast, HarmoniCa works on nearly all Intel CPUs.

Microarchitecture for serverless. Many works have shown that serverless functions degrade due to cold microarchitecture states. In response, recent works [12], [14], [21] design new hardware prefetcher or restoration to prewarm instruction cache and branch predictors before an invocation. Some works [13], [24] propose new microarchitectures for serverless. In contrast, we propose a software-only approach to preserve cache states through careful partitioning.

X. Conclusion

This paper shows that a cold LLC is a major bottleneck for serverless performance, and existing software and hardware mechanisms are ill-suited to address it. We present HarmoniCa, an LLC management system for commodity Intel processors that recovers performance previously requiring hardware changes. By combining fine-grained slice partitioning with dynamic, function-aware allocation, HarmoniCa restores cache efficiency and improves latency of serverless workloads by 34.7%.

Acknowledgment

This work was funded in part by NSF awards CNS-2145888 and CNS-2140305 and gifts from Ampere Computing and Longview Philanthropy. We thank anonymous reviewers for their valuable feedback. The artifact evaluation for this paper was conducted on the Chameleon testbed, supported by the National Science Foundation.

REFERENCES

[1] Grand View Research, Inc., "Serverless computing market size, industry report, 2025-2030," https://www.grandviewresearch.com/industry-analysis/serverless-computing-market-report, 2025, accessed: 2025-11-18.

[2] "Serverless Function, FaaS Serverless - AWS Lambda - AWS." [Online]. Available: https://aws.amazon.com/lambda/

[3] "Serverless on Azure | Microsoft Azure." [Online]. Available: https://azure.microsoft.com/en-us/solutions/serverless

[4] "Serverless." [Online]. Available: https://cloud.google.com/serverless

[5] "Function Compute - Alibaba Cloud." [Online]. Available: https://www.alibabacloud.com/product/function-compute

[6] Amazon Web Services, "Architecting a Highly Available Serverless, Microservices-Based Ecommerce Site | AWS Architecture Blog," Jul. 2021. [Online]. Available: https://aws.amazon.com/blogs/architecture/architecting-a-highly-available-serverless-microservices-based-ecommerce-site/

[7] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A Serverless Video Processing Framework," in *Proceedings of the ACM Symposium on Cloud Computing*. Carlsbad CA USA: ACM, Oct. 2018, pp. 263–274. [Online]. Available: https://dl.acm.org/doi/10.1145/3267809.3267815

[8] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 475–488. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/fouladi

[9] "Cloud Run GPUs are now generally available." [Online]. Available: https://cloud.google.com/blog/products/serverless/cloud-run-gpus-are-now-generally-available

[10] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated Model-less Inference Serving," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 397–411. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/romero

[11] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "INFless: a native serverless system for low-latency, high-throughput inference," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, Feb. 2022, pp. 768–781. [Online]. Available: https://dl.acm.org/doi/10.1145/3503222.3507709

[12] D. Schall, A. Margaritov, D. Ustiugov, A. Sandberg, and B. Grot, "Lukewarm serverless functions: characterization and optimization," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*. New York New York: ACM, Jun. 2022, pp. 757–770. [Online]. Available: https://dl.acm.org/doi/10.1145/3470496.3527390

[13] J. Stojkovic, E. Choukse, E. Saurez, I. Goiri, and J. Torrellas, "Mosaic: Harnessing the micro-architectural resources of servers in serverless environments," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 1397–1412.

[14] H. Volos, S. Vassiliou, G. Antoniou, D. B. Bartolini, and Y. Sazeides, "Leveraging control-flow similarity to reduce branch predictor cold effects in microservices," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA '25. New York, NY, USA: Association for Computing Machinery, Jun. 2025, pp. 616–630. [Online]. Available: https://dl.acm.org/doi/10.1145/3695053.3731059

[15] Z. Jia and E. Witchel, "Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Virtual USA: ACM, Apr. 2021, pp. 152–166. [Online]. Available: https://dl.acm.org/doi/10.1145/3445814.3446701

[16] A. Sriraman and T. F. Wenisch, "µTune: Auto-Tuned threading for OLDI microservices," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 177–194. [Online]. Available: http://www.usenix.org/conference/osdi18/presentation/sriraman

[17] Y. Zhang, D. Meisner, J. Mars, and L. Tang, "Treadmill: attributing the source of tail latency through precise load testing and statistical inference," *SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 456–468, Jun. 2016. [Online]. Available: https://dl.acm.org/doi/10.1145/3007787.3001186

[18] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 205–218. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shahrad

[19] A. Joosen, A. Hassan, M. Asenov, R. Singh, L. Darlow, J. Wang, Q. Deng, and A. Barker, "Serverless Cold Starts and Where to Find Them," Oct. 2024, arXiv:2410.06145 [cs]. [Online]. Available: http://arxiv.org/abs/2410.06145

[20] Datadog, "The State of Serverless 2020," Feb. 2020. [Online]. Available: https://www.datadoghq.com/state-of-serverless-2020/

[21] D. Schall, A. Sandberg, and B. Grot, "Warming Up a Cold Front-End with Ignite," in *56th Annual IEEE/ACM International Symposium on Microarchitecture*. Toronto ON Canada: ACM, Oct. 2023, pp. 254–267. [Online]. Available: https://dl.acm.org/doi/10.1145/3613424.3614258

[22] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural Implications of Function-as-a-Service Computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus OH USA: ACM, Oct. 2019, pp. 1063–1075. [Online]. Available: https://dl.acm.org/doi/10.1145/3352460.3358296

[23] T. Asheim, T. A. Khan, B. Kasicki, and R. Kumar, "Impact of microarchitectural state reuse on serverless functions," in *Proceedings of the Eighth International Workshop on Serverless Computing*. Quebec City, Canada: ACM, Nov. 2022, pp. 7–12. [Online]. Available: https://dl.acm.org/doi/10.1145/3565382.3565879

[24] J. Stojkovic, C. Liu, M. Shahbaz, and J. Torrellas, "μManycore: A Cloud-Native CPU for Tail at Scale," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*. Orlando FL USA: ACM, Jun. 2023, pp. 1–15. [Online]. Available: https://dl.acm.org/doi/10.1145/3579371.3589068

[25] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, Dec. 2006, pp. 423–432, ISSN: 2379-3155. [Online]. Available: https://ieeexplore.ieee.org/document/4041865/

[26] M. Shahrad, S. Elnikety, and R. Bianchini, "Provisioning Differentiated Last-Level Cache Allocations to VMs in Public Clouds," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 319–334. [Online]. Available: https://dl.acm.org/doi/10.1145/3472883.3487006

[27] Y. Ye, R. West, Z. Cheng, and Y. Li, "COLORIS: A dynamic cache partitioning system using page coloring," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Aug. 2014, pp. 381–392. [Online]. Available: https://ieeexplore.ieee.org/document/7855915/

[28] A. Scolari, D. B. Bartolini, and M. D. Santambrogio, "A Software Cache Partitioning System for Hash-Based Caches," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, pp. 57:1–57:24, Dec. 2016. [Online]. Available: https://dl.acm.org/doi/10.1145/3018113

[29] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: improving resource efficiency at scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. Portland Oregon: ACM, Jun. 2015, pp. 450–462. [Online]. Available: https://dl.acm.org/doi/10.1145/2749469.2749475

[30] 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 *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 104–117. [Online]. Available: https://ieeexplore.ieee.org/document/8327002/

[31] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang, "DCAPS: dynamic cache allocation with partial sharing," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1–15. [Online]. Available: https://dl.acm.org/doi/10.1145/3190508.3190511

[32] X. Wang, S. Chen, J. Setter, and J. F. Martínez, "SWAP: Effective fine-grain management of shared last-level caches with minimum hardware support," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 121–132, ISSN: 2378-203X. [Online]. Available: https://ieeexplore.ieee.org/document/7920819/

[33] S. Chen, C. Delimitrou, and J. F. Martínez, "PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. Providence RI USA: ACM, Apr. 2019, pp. 107–120. [Online]. Available: https://dl.acm.org/doi/10.1145/3297858.3304005

[34] A. Agache, M. Brooker, A. Florescu, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: lightweight virtualization for serverless applications," in *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'20. USA: USENIX Association, 2020, p. 419–434.

[35] Y. Zhu, D. Richins, M. Halpern, and V. J. Reddi, "Microarchitectural implications of event-driven server-side web applications," in *Proceedings of the 48th International Symposium on Microarchitecture*. Waikiki Hawaii: ACM, Dec. 2015, pp. 762–774. [Online]. Available: https://dl.acm.org/doi/10.1145/2830772.2830792

[36] "vhive-serverless/vSwarm," Oct. 2025. [Online]. Available: https://github.com/vhive-serverless/vSwarm

[37] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 60–71. [Online]. Available: https://doi.org/10.1145/1815961.1815971

[38] S. Jahagirdar, V. George, I. Sodhi, and R. Wells, "Power management of the third generation intel core micro architecture formerly codenamed ivy bridge," in *2012 IEEE Hot Chips 24 Symposium (HCS)*, Aug. 2012, pp. 1–49. [Online]. Available: https://ieeexplore.ieee.org/document/7476478/

[39] J. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," *http://www. cs. virginia. edu/stream/*, 2006.

[40] A. Ailamaki, D. J. DeWitt, and M. D. Hill, "Data page layouts for relational databases on deep memory hierarchies," *The VLDB Journal*, vol. 11, no. 3, p. 198–215, Nov. 2002. [Online]. Available: https://doi.org/10.1007/s00778-002-0074-9

[41] K. Wang, J. Liu, and F. Chen, "Put an elephant into a fridge: optimizing cache efficiency for in-memory key-value stores," *Proc. VLDB Endow.*, vol. 13, no. 9, p. 1540–1554, May 2020. [Online]. Available: https://doi.org/10.14778/3397230.3397247

[42] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "Reload+refresh: abusing cache replacement policies to perform stealthy cache attacks," in *Proceedings of the 29th USENIX Conference on Security Symposium*, ser. SEC'20. USA: USENIX Association, 2020.

[43] "intel/intel-cmt-cat," Oct. 2025. [Online]. Available: https://github.com/intel/intel-cmt-cat

[44] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas, "MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*. Orlando FL USA: ACM, Jun. 2023, pp. 1–15. [Online]. Available: https://dl.acm.org/doi/10.1145/3579371.3589069

[45] V. Young, C. Chou, A. Jaleel, and M. Qureshi, "SHiP++: Enhancing signature-based hit predictor for improved cache performance," in *The 2nd Cache Replacement Championship (CRC-2)*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:43689204

[46] A. Jain and C. Lin, "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2016, pp. 78–89, ISSN: 1063-6897. [Online]. Available: https://ieeexplore.ieee.org/document/7551384

[47] I. Shah, A. Jain, and C. Lin, "Effective mimicry of Belady's MIN policy," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 558–572, ISSN: 2378-203X. [Online]. Available: https://ieeexplore.ieee.org/document/9773195/

[48] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, "The championship simulator: Architectural simulation for education and competition," 2022. [Online]. Available: http://arxiv.org/abs/2210.14324

[49] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus OH USA: ACM, 2019, pp. 413–425. [Online]. Available: https://dl.acm.org/doi/10.1145/3352460.3358319

[50] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966. [Online]. Available: https://ieeexplore.ieee.org/document/5388441/

[51] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 430–441. [Online]. Available: https://ieeexplore.ieee.org/document/7851492

[52] N. Beckmann and D. Sanchez, "Maximizing cache performance under uncertainty," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Austin, TX: IEEE, 2017, pp. 109–120. [Online]. Available: http://ieeexplore.ieee.org/document/7920818/

[53] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 389–400, ISSN: 2379-3155. [Online]. Available: https://ieeexplore.ieee.org/document/6493636

[54] Google Cloud. (2026) Cloud run runtime lifecycle. [Online]. Available: https://docs.cloud.google.com/run/docs/runtime-support

[55] Amazon Web Services. (2026) Lambda runtimes. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html

[56] A. Farshin, A. Roozbeh, G. Q. Maguire, and D. Kostić, "Make the Most out of Last Level Cache in Intel Processors," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, Mar. 2019, pp. 1–17. [Online]. Available: https://dl.acm.org/doi/10.1145/3302424.3303977

[57] H. Park, J. Lou, S. Lee, Y. Yuan, K. Park, Y. Son, I. Jeong, and N. S. Kim, "A4: Microarchitecture-Aware LLC Management for Datacenter Servers with Emerging I/O Devices," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA '25. New York, NY, USA: Association for Computing Machinery, Jun. 2025, pp. 679–693. [Online]. Available: https://dl.acm.org/doi/10.1145/3695053.3731114

[58] S. Volos, C. Fournet, J. Hofmann, B. Köpf, and O. Oleksenko, "Principled Microarchitectural Isolation on Cloud CPUs," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, Dec. 2024, pp. 183–197. [Online]. Available: https://dl.acm.org/doi/10.1145/3658644.3690183

[59] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, Jun. 2011, pp. 194–199, ISSN: 2325-6664. [Online]. Available: https://ieeexplore.ieee.org/document/5958812/

[60] H. Kim and R. R. Rajkumar, "Real-time cache management for multi-core virtualization," in *Proceedings of the 13th International Conference on Embedded Software*. Pittsburgh Pennsylvania: ACM, Oct. 2016, pp. 1–10. [Online]. Available: https://dl.acm.org/doi/10.1145/2968478.2968480

[61] J. D. McCalpin, "Address Hashing in Intel Processors," The University of Texas at Austin, Tech. Rep., 2018. [Online]. Available: https://repositories.lib.utexas.edu/handle/2152/86210

[62] M. Rainer, L. Hetterich, F. Thomas, T. Hornetz, L. Trampert, L. Gerlach, and M. Schwarz, "Rapid Reversing of Non-Linear CPU Cache Slice Functions: Unlocking Physical Address Leakage," in *2025 IEEE Symposium on Security and Privacy (SP)*, May 2025, pp. 3497–3515, ISSN: 2375-1207. [Online]. Available: https://ieeexplore.ieee.org/document/11023462/

[63] I. Eidus and H. Dickins, "Kernel Samepage Merging — The Linux Kernel documentation — docs.kernel.org," https://docs.kernel.org/admin-guide/mm/ksm.html, 2009, [Accessed 06-04-2026].

[64] L. Ao, G. Porter, and G. M. Voelker, "FaaSnap: FaaS made fast using snapshot-based VMs," in *Proceedings of the Seventeenth European Conference on Computer Systems*. Rennes France: ACM, Mar. 2022, pp. 730–746. [Online]. Available: https://dl.acm.org/doi/10.1145/3492321.3524270

[65] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Virtual USA: ACM, Apr. 2021, pp. 559–572. [Online]. Available: https://dl.acm.org/doi/10.1145/3445814.3446714

[66] N. Lazarev, V. Gohil, J. Tsai, A. Anderson, B. Chitlur, Z. Zhang, and C. Delimitrou, "Sabre: Hardware-Accelerated snapshot compression

for serverless MicroVMs," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 1–18. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/lazarev

[67] Intel Corporation, "Intel® Xeon® Processors and Intel® Processor Trace (Intel® PT)... — intel.com," https://www.intel.com/content/www/us/en/support/articles/000059254.html, 2025, [Accessed 06-04-2026].

[68] M. Kerrisk, "perf-mem(1) - Linux manual page — man7.org," https://man7.org/linux/man-pages/man1/perf-mem.1.html, 2025, [Accessed 06-04-2026].

[69] C. I. King, "ColinIanKing/stress-ng," Nov. 2025. [Online]. Available: https://github.com/ColinIanKing/stress-ng

[70] T. Heo and F. Mickler, "Workqueue — The Linux Kernel documentation — docs.kernel.org," https://docs.kernel.org/core-api/workqueue.html, 2010, [Accessed 06-04-2026].

[71] 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 *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, Feb. 2008, pp. 367–378, ISSN: 2378-203X. [Online]. Available: https://ieeexplore.ieee.org/document/4658653/

[72] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European conference on Computer systems*, ser. EuroSys '09. New York, NY, USA: Association for Computing Machinery, Apr. 2009, pp. 89–102. [Online]. Available: https://dl.acm.org/doi/10.1145/1519065.1519076

[73] "openzipkin/zipkin," Nov. 2025. [Online]. Available: https://github.com/openzipkin/zipkin

[74] "OpenTelemetry Metrics." [Online]. Available: https://grpc.io/docs/guides/opentelemetry-metrics/

[75] J. Kim and K. Lee, "FunctionBench: A Suite of Workloads for Serverless Cloud Function Service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, Jul. 2019, pp. 502–504, ISSN: 2159-6190. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8814583

[76] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas, "SpecFaaS: Accelerating Serverless Applications with Speculative Function Execution," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2023, pp. 814–827, ISSN: 2378-203X. [Online]. Available: https://ieeexplore.ieee.org/document/10071120/

[77] H. Qiu, W. Mao, A. Patke, C. Wang, H. Franke, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer, "SIMPPO: a scalable and incremental online learning framework for serverless resource management," in *Proceedings of the 13th Symposium on Cloud Computing*. San Francisco California: ACM, Nov. 2022, pp. 306–322. [Online]. Available: https://dl.acm.org/doi/10.1145/3542929.3563475

[78] Y. Zhang, I. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and Cheaper Serverless Computing on Harvested Resources," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 724–739. [Online]. Available: https://dl.acm.org/doi/10.1145/3477132.3483580

[79] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, and A. Akella, "Atoll: A scalable low-latency serverless platform," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 138–152. [Online]. Available: https://dl.acm.org/doi/10.1145/3472883.3486981

[80] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, "Fifer: Tackling resource underutilization in the serverless era," in *Proceedings of the 21st International Middleware Conference*, ser. Middleware '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 280–295. [Online]. Available: https://dl.acm.org/doi/10.1145/3423211.3425683

[81] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, "Kraken: Adaptive container provisioning for deploying dynamic DAGs in serverless platforms," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 153–167. [Online]. Available: https://dl.acm.org/doi/10.1145/3472883.3486992

[82] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance serverless computing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 923–935. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/akkus

[83] Red Hat, "Red Hat Enterprise Linux for Real Time 7 Tuning Guide: Realtime-Specific Tuning," accessed: Nov. 13, 2025. [Online]. Available: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux_for_real_time/7/html/tuning_guide/chap-realtime-specific_tuning

[84] User interface for resource control feature (resctrl) — the linux kernel documentation. [Online]. Available: https://docs.kernel.org/filesystems/resctrl.html

[85] K. Wang, Y. Li, C. Wang, T. Jia, K. Chow, Y. Wen, Y. Dou, G. Xu, C. Hou, J. Yao, and L. Zhang, "Characterizing Job Microarchitectural Profiles at Scale: Dataset and Analysis," in *Proceedings of the 51st International Conference on Parallel Processing*. Bordeaux France: ACM, Aug. 2022, pp. 1–11. [Online]. Available: https://dl.acm.org/doi/10.1145/3545008.3545026

[86] A. Sahraei, S. Demetriou, A. Sobhgol, H. Zhang, A. Nagaraja, N. Pathak, G. Joshi, C. Souza, B. Huang, W. Cook, A. Golovei, P. Venkat, A. Mcfague, D. Skarlatos, V. Patel, R. Thind, E. Gonzalez, Y. Jin, and C. Tang, "XFaaS: Hyperscale and Low Cost Serverless Functions at Meta," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, Oct. 2023, pp. 231–246. [Online]. Available: https://dl.acm.org/doi/10.1145/3600006.3613155

[87] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 412–426. [Online]. Available: https://dl.acm.org/doi/10.1145/3472883.3487003

[88] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who limits the resource efficiency of my datacenter: an analysis of alibaba datacenter traces," in *Proceedings of the International Symposium on Quality of Service*, ser. IWQoS '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 1–10. [Online]. Available: https://dl.acm.org/doi/10.1145/3326285.3329074

[89] Y. Yuan, M. Alian, Y. Wang, R. Wang, I. Kurakin, C. Tai, and N. S. Kim, "Don't Forget the I/O When Allocating Your LLC," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2021, pp. 112–125, ISSN: 2575-713X. [Online]. Available: https://ieeexplore.ieee.org/document/9499850/

[90] J. Park, H. Yeom, and Y. Son, "Page Reusability-Based Cache Partitioning for Multi-Core Systems," *IEEE Transactions on Computers*, vol. 69, no. 6, pp. 812–818, Jun. 2020. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8964429

[91] A. Garcia-Garcia, J. C. Saez, F. Castro, and M. Prieto-Matias, "LFOC: A Lightweight Fairness-Oriented Cache Clustering Policy for Commodity Multicores," in *Proceedings of the 48th International Conference on Parallel Processing*. Kyoto Japan: ACM, Aug. 2019, pp. 1–10. [Online]. Available: https://dl.acm.org/doi/10.1145/3337821.3337925

[92] J. Ahn, C. Hyun, D. Lee, and S. H. Noh, "Cache-aware block allocation for memory-technology storage targeted file systems," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. Limassol Cyprus: ACM, Apr. 2019, pp. 1424–1431. [Online]. Available: https://dl.acm.org/doi/10.1145/3297280.3297423

[93] M. Xu, L. Thi, X. Phan, H.-Y. Choi, and I. Lee, "vCAT: Dynamic Cache Management Using CAT Virtualization," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2017, pp. 211–222. [Online]. Available: https://ieeexplore.ieee.org/document/7939041/

[94] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gómez, "Application Clustering Policies to Address System Fairness with Intel's Cache Allocation Technology," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2017, pp. 194–205. [Online]. Available: https://ieeexplore.ieee.org/document/8091245/

[95] 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 *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Mar. 2016, pp. 406–418, ISSN: 2378-203X. [Online]. Available: https://ieeexplore.ieee.org/document/7446082

[96] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, "Making shared caches more predictable on multicore platforms," in *2013 25th Euromicro Conference on Real-Time Systems*, 2013, pp. 157–167.

[97] J. Stojkovic, C. Liu, M. Shahbaz, and J. Torrellas, "HardHarvest: Hardware-Supported Core Harvesting for Microservices," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. Tokyo Japan: ACM, Jun. 2025, pp. 708–722. [Online]. Available: https://dl.acm.org/doi/10.1145/3695053.3731071

[98] G. Dessouky, E. Stapf, P. Mahmoody, A. Gruler, and A.-R. Sadeghi, "Chunked-Cache: On-Demand and Scalable Cache Isolation for Security Architectures," in *Proceedings 2022 Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, 2022. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2022-110-paper.pdf

[99] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2011, pp. 57–68, ISSN: 1063-6897. [Online]. Available: https://ieeexplore.ieee.org/document/6307780

[100] L. Gerlach, S. Schwarz, N. Faroß, and M. Schwarz, "Efficient and Generic Microarchitectural Hash-Function Recovery," in *2024 IEEE Symposium on Security and Privacy (SP)*, May 2024, pp. 3661–3678, ISSN: 2375-1207.

[101] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, "Mapping the Intel Last-Level Cache," 2015. [Online]. Available: https://eprint.iacr.org/2015/905

[102] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons learned from the chameleon testbed," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

APPENDIX
ARTIFACT APPENDIX

A. Abstract

The artifact contains the HarmoniCa prototype, page coloring implementation in Linux, experiment and analysis scripts used for the main evaluation. It reproduces the results of all LLC management configurations in Section VIII-A: *Shared*, *MosaicScheduler*, *Static*, *Adv. Static*, *HC-Coarse*, and *HarmoniCa*. The workflow deploys three instances of each of 12 functions on a serverless function server, replays the paper's invocation trace from a separate client, collects function latencies and hardware counters, and reproduces Figures 15, 16, 17.

B. Artifact check-list (meta-information)

- **Program:** HarmoniCa prototype, Linux kernel with page coloring, serverless benchmark functions from vSwarm [36] and FunctionBench [75]
- **Compilation:** GCC and Go compiler
- **Run-time environment:** Ubuntu 24.04 LTS, Docker 29.6.2
- **Hardware:** Chameleon [102] *compute_icelake_r650* node with Intel Xeon Platinum 8380 CPU and 256GB memory, and a client machine.
- **Metrics:** Server-side P50/P95 latency, LLC and branch MPKI.
- **Output:** Raw data and Figures.
- **Experiments:** Figures 15, 16, 17.
- **How much disk space is required (approximately)?:** 64GB
- **How much time is needed to prepare workflow (approximately)?:** 6h
- **How much time is needed to complete experiments (approximately)?:** 6h
- **Publicly available?:** Yes.
- **Code licenses?:** MIT License
- **Archived?:** 10.5281/zenodo.21548955

C. Description

1) How to access: The artifact is available at https://doi.org/10.5281/zenodo.21548955. It includes:

- Page coloring implementation in Linux.
- All serverless function benchmark Docker images.
- Instructions and scripts to prepare the machines.
- HarmoniCa prototype and scripts to run the experiments.

2) Hardware dependencies: The experiment uses a server machine at Chameleon [102] *compute_icelake_r650* node with Intel Xeon Platinum 8380 CPU and 256GB memory, and a client machine reachable from the server machine. Although HarmoniCa was evaluated on 12-core Ice Lake CPUs, we ported the artifact for 40-core Ice Lake CPUs that are widely available on Chameleon Cloud. To better match the paper results, only 20 cores and 30MB LLC are used.

3) Software dependencies: The artifact uses Docker Engine with Swarm mode to orchestrate serverless functions. It requires Python 3 to run the HarmoniCa prototype and analyze experimental results, GCC and the Go compiler to build the Linux kernel and supporting utilities, and other standard Linux tools for resource management and profiling. `HarmoniCa-artifact/README.md` provides detailed setup instructions.

D. Installation

Download the artifacts to the home directory on both the server and client machines. Follow the instructions in `HarmoniCa-artifact/README.md` and install the artifact on both the server and client machines. First, install software dependencies, including the customized Linux kernel and all Docker images. Second, configure both machines as Docker Swarm nodes. Third, configure the deployment destination of serverless functions and deploy all services.

E. Experiment workflow

From the client artifact root, activate the virtual environment and run the automated experiment script:

```
source .venv/bin/activate
SERVER_HOST=<server-ip> \
SERVER_USER=<server-user> \
bash evaluation/scripts/run_experiments.sh all
```

Generate the paper figures after all runs:

```
python3 \
  evaluation/scripts/generate_paper_figures.py
```

F. Evaluation and expected results

A successful run creates one directory for each of the six LLC management setups. Each directory contains `zipkin_analysis/summary.json` and `perfs/`. The figure generator produces Figures 15, 16, 17.

Absolute latency is machine-dependent, but the expected qualitative result is that *HarmoniCa* improves the median and tail latency of cache-sensitive functions compared to *Shared*, *Static*, and *MosaicScheduler*, without notably harming the performance of cache-insensitive functions.

Moreover, each component contributes to the overall performance improvement. Slice-aware cache partitioning enables *Adv. Static* to outperform *Static* on cache-sensitive functions. Stream-resistant allocation allows *HC-Coarse* to outperform *Shared*, while thrash resistance further improves performance, enabling *HarmoniCa* to outperform *HC-Coarse*.