Understanding Sidecars in Cloud Orchestration

Prateek Sahu
prateeks@utexas.edu
The University of Texas at Austin
Austin, USA

Shijia Wei
shijia@utexas.edu
The University of Texas at Austin
Austin, USA

Neeraja J. Yadwadkar
neeraja@austin.utexas.edu
The University of Texas at Austin
Austin, USA

Mohit Tiwari
tiwari@austin.utexas.edu
The University of Texas at Austin
Austin, USA

Abstract

Sidecars are used by organizations to implement advanced operational and security features in cloud environments. Since sidecars interpose on network traffic to provide these functionalities, they can degrade critical service level metrics such as latency and throughput. However, the precise impact of sidecars on such key metrics remains unclear. Our evaluation quantifies service-layer overheads as well as the micro-architectural implications of using sidecars in orchestration platforms – and evaluate these overheads across a range of sidecar configurations.

We show that the absolute overheads of the sidecars are independent of the workloads and depend on the filters and the microservice topology. This allows us to model performance predictably as we compose sidecar filters. Our analysis indicates very low reuse of the instruction caches (poor misses per kilo instructions) despite high-frequency reuse of sidecars. Increasing private caches from 256KB to 1.25MB across processor generations sees only a 10% improvement in the frontend stalls – this is due to high indirect branch misses and thrashing from more aggressive prefetchers and predictors that degrade the L1-I cache MPKIs up to 40%. Our study also finds that utilizing a few large pages can reduce iTLB misses and page walks by 80% at the cost of modest memory overheads.

CCS Concepts

• **General and reference** → **Performance**; **Evaluation**; • **Computer systems organization** → **Cloud computing**.

Keywords

Service Mesh, Orchestration, Performance

ACM Reference Format:
Prateek Sahu, Shijia Wei, Neeraja J. Yadwadkar, and Mohit Tiwari. 2025. Understanding Sidecars in Cloud Orchestration. In *The 3rd Workshop on SErverless Systems, Applications and MEthodologies (SESAME' 25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3721465.3721862

ACM ISBN 979-8-4007-1557-0/2025/03
https://doi.org/10.1145/3721465.3721862

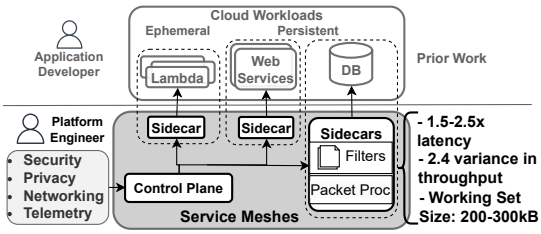


Figure 1: Platform engineers use service mesh for critical fleet wide security and networking functions. We quantify the impact of sidecar components on instruction caches and TLB to model composable performance overheads and identify optimization opportunities. Our analysis shows that we can reduce iTLB miss rate by 80% using 5 hugepages.

1 Introduction

Cloud frameworks like service-meshes [14, 40, 52] and container orchestrators [45, 48, 73] have enabled large scale deployment of distributed microservice applications [35, 86]. Orchestration relies on a robust control plane to provide features such as efficient scheduling [13], transparent service discovery [30, 90], and on-demand workload and resource scaling [34, 88, 96]. While control plane developments have helped application developers provide higher quality of service [74] and improve datacenter utilization [12, 83], data plane components allow platform engineers to implement service-level functionality to improve security and improve monitoring of deployed workloads. Sidecars accomplish this by applying sets of filters to all traffic associated with the microservice. Containerized sidecar processes do not require application rewrites or recompilations, and can implement complex logic for application layer security and management, making them the preferred choice over in-kernel [14] or library-based [57, 76] sidecars.

Scaling-up data plane policies from implementing simple cross-service encryption to a larger number of more complex mechanisms is severely curtailed due to the unpredictable and significant performance impacts of filter choices on application workloads. Fig. 1 depicts the role of sidecars in cloud environments and the performance penalties associated with its use. Service mesh vendors [15, 42, 53] report 2×-6× variance in latency, while academic studies [99] report an increase in latency and utilization by 30-185% and 41-92% across different benchmark applications. This can cost millions in additional computing resources [76] to support production workloads. Our study confirms these variances with the instruction and CPU

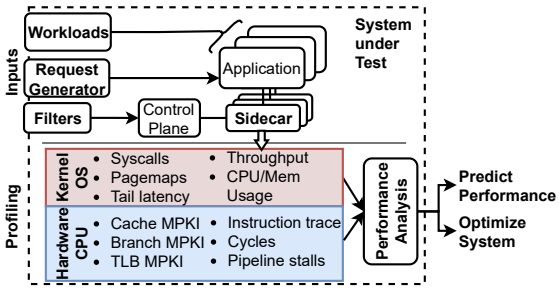


Figure 2: Our framework can deploy and profile various configurations, and provides analytics.

usage increasing by up to 76% and 91% respectively for simple TCP filters. Furthermore, we note that seemingly simple changes, such as a switch from TCP to HTTP filters for access control, result in a 2.4× decrease in throughput. A methodical understanding of factors influencing sidecar performance and resource usage is needed for operators to navigate the deployment choices with a balance between performance and operational functionalities.

In this paper we present a methodical study of sidecars in modern cloud frameworks. Our characterization improves upon existing white-box methodology used in literature [99] by a) incorporating diverse and complex sidecar filters and b) microarchitectural profiling of orchestration frameworks (Fig. 2).

Existing studies, primarily conducted by practitioners [42, 53, 62], focus on performance degradation for applications and lack comprehensive analysis for different filter configurations. Therefore, these studies offer isolated findings with minimal comparisons between frameworks, forcing operators to conduct their own evaluations of custom sidecar configurations. In contrast, by leveraging a broad and diverse set of filters, our study enables us to model sidecar performance as a combination of individual filters and other components, such as packet reads/writes and protocol parsing.

Platform operators also aim to minimize performance overheads while deploying such rich filters in bulk across production environments. Since sidecars and filters operate at microsecond scales, it is imperative to understand their interaction with the processor microarchitecture to systematically analyze and mitigate performance overheads. Our micro-architectural profiling of Envoy sidecars [28] show that different filters spend up to 25% of all cycles waiting for valid instructions, caused by a poor reuse of the instruction cache (iCache) – indicated by 45-75 MPKI, and stalls incurred due to very high instruction translation lookaside buffer (iTLB) misses. Binary instrumentation reveal a high iCache working set size (WSS) of 200kB-270kB for TCP and HTTP based filters. Further, an irregular access pattern results in sidecars, touching over 300 4k pages each request which contributes to a high iTLB miss rate. We explore solutions of mixed page sizes based on access pattern and show that using just 4-6 2M-hugepages can help reduce the iTLB miss rates by 50%-85% in simulation environments.

We make the following key contributions in our paper:

(1) We improve on the profiling methodology for evaluating orchestration frameworks by incorporating a diverse and representative set of sidecar filters. This provides visibility into performance critical components of sidecars as well as account for performance variability across different types of filters.
(2) We provide the first microarchitectural analysis of sidecar performance in modern orchestration frameworks. We show how such characterizations can reveal critical microarchitectural bottlenecks and guide domain-specific solutions.

We open-source our framework and toolkit for broader use at https://github.com/utspark/sidecar-characterization, that can inform platform engineers and architects of performance trade-offs and motivate future optimizations and cloud-native architectures.

2 Background

2.1 Microservices, Functions and Service Meshes

Microservices rely on cloud orchestration frameworks to abstract server nodes and handle tasks such as scheduling, fault tolerance and service discovery. Increasing cluster sizes, service volume and complex capabilities, increases the communication traffic multi-fold, making it challenging to debug [58] and update infrastructures.

Service meshes solve this challenge by giving platform engineers flexibility to augment microservices with rich filters that provide operational visibility, application security and management. A service mesh easily conforms to any microservice topology by injecting data plane *sidecars* to manage intra-service communication as depicted in Fig. 3. In contrast to microservice applications and function workloads, sidecars are persistent *infrastructure components* that have a consistent runtime behavior, unlike application workloads that exhibit runtime differences and FaaS functions that are ephemeral processes. Frameworks like Knative [47] include sidecars in the same pod as a FaaS function, and hence sometimes scale down to zero. However, during warm states, the runtime behavior of infrastructure sidecar processes are separate from user workload functions and exhibit well-defined patterns. Recent surveys [16, 17] found over 60% of organizations use service meshes, making it essential to study their performance.

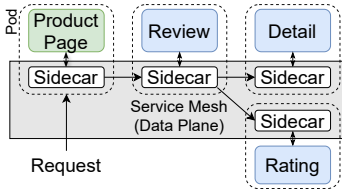


Figure 3: BookInfo [8] deployed with a service mesh.

2.2 Sidecars

Sidecars processes are co-located with each microservice instance within a shared network namespace called pod. This allows the sidecar to intercept any traffic for the service instance and apply configured set of filters. These filters are generally aimed towards network management (packet management [37, 39, 72], load balancing [22, 23], service discovery [30, 90]), security (mTLS [60], isolation [82], service-level authorization [7, 29, 65]) or observability (tracing [21, 43, 100], logging [4, 89], stats [50]). To enable rich application layer filters [27], sidecars usually run in the user-space.

eBPF-based [14] sidecars limit functionality due to kernel restrictions, while consolidation of sidecars [3] raises security challenges. Although the use of sidecars as libraries [57, 76] offers performance

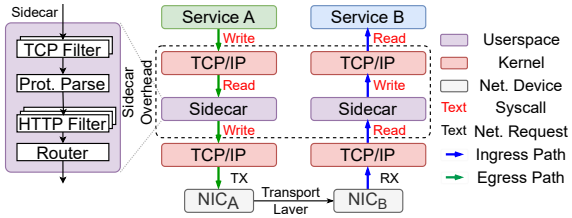


Figure 4: Life of a request between two services via sidecars.

benefits without sacrificing functionality, its adoption is low because it needs recompilation of binaries and services. Hence we expect process-based sidecars to continue to dominate in the future.
Packet lifetime: Sidecars intercept both ingress and egress traffic for an application (Fig. 4), resulting in a single request between two service instances being broken into three network requests. Outbound packets from service A are written to network buffers before it is read by its sidecar. The sidecar processes the packet, invoking any configured TCP and HTTP/gRPC filters on its egress path. The sidecar completes its action by utilizing an HTTP or TCP router to write back to the network stack with service B as the destination. The packet undergoes a similar life-cycle before it is sent out to the destination over the network.

3 Motivation

Service meshes provide a powerful tool to platform engineers to independently and dynamically configure services with operational logic as a set of filters, allowing operators to move from traditional network management to rich security and telemetry tasks.

However, platform engineers face a significant challenge in deployment of these filters because of the cost of using these filters are often large and unpredictable. Black-box performance analysis are often application specific and cannot be generalized. The variance in overheads is high across different filters making it challenging for architects and engineers to either predict the performance or work towards optimizing the penalty.

3.1 'Side'cars are often the dominant modules

We study user-visible metrics like tail latency and throughput to identify the service level degradation experienced due to the use of sidecar processes. We evaluate this using an off-the-shelf service mesh (Istio [40]) with a default predefined sidecar configuration [41]. We use three commonly used [33, 99] benchmark applications – DeathstarBench's Hotel Reservation [35], BookInfo [8] and Online Boutique [66]. We also quantify these overheads as we scale the cluster to larger sizes. Figure 5 quantifies the p90 tail latency of applications and associated networking activities with and without a sidecar configured orchestration. "Base app" includes the application processing time and all network related latency in the system. "Sidecars" only quantifies the sum of latencies incurred when a request is being processed in a sidecar associated with any of the microservices. We notice that applications experience 5-25ms of tail latency increase across benchmark applications, indicating that sidecar proxies can sometimes dominate the overall application workload. Although larger clusters see an increase in overall application latency due to networking overheads, the penalty incurred

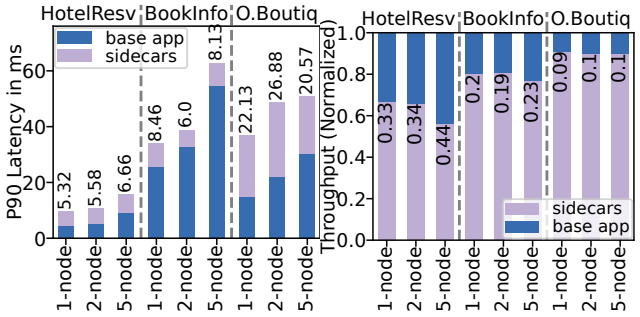


Figure 5: Performance measurements across applications with different cluster size show no significant change in the performance penalty of using sidecars in larger cluster sizes. Throughput degradation of using sidecars is shown normalized to maximum QPS without service mesh.

due to sidecars in each application remains unchanged as we go to larger clusters. The variance in overheads are due to differences in application topologies and sidecar configurations that we explore in detail in Section 5. Increasing compute demands results in higher utilization and a throughput degradation of up to 45%. Since using sidecars decreases throughputs, we show a normalized view of the throughput in Fig. 5, where the "sidecars" block is lower than "base app". This demand compounds as we move towards applications with 1000s of microservices [76] and is fueled by two trends of the industry. First, agile development encourages smaller application components [55] and "offload" any operational task to the orchestrator and the sidecar components. This leads to communication traffic increase that results in higher time spent in sidecars rather than application processing. Second, flexibility of dynamic configuration comes with the complexity of design in sidecar processes which require more CPU cycles and resources. With notable processing times for sidecars, it is imperative for service providers and architects to understand the micro-architectural impacts to design efficient systems.

3.2 Sidecar's performance variance

Our analysis reveals two key factors affecting these variances:
Sidecar Configuration A sidecar can be customized with one or more sets of filters during application deployment. Depending on the filter's compute requirements and complexity, it can affect the overall performance quite differently. Figure 6 shows the overall latency and throughput experienced by a simple echo server when we configure the sidecar with a set of commonly used TCP (TCP, RBAC, TLS) and HTTP (HTTP, Log and Mix) filters. We discuss details about these filters in Section 4. Our results indicate that across a similar protocol (TCP or HTTP), latency shows little variance. However, based on complexity, throughput degrades by about 2.4× as we switch from TCP to HTTP protocols.
Processor micro-architecture Figure 6, evaluates latency and throughput of a simple echo-server applications configured with different sidecar filters across two machines with host processors that are 5 generations apart. Across both generations, we see that

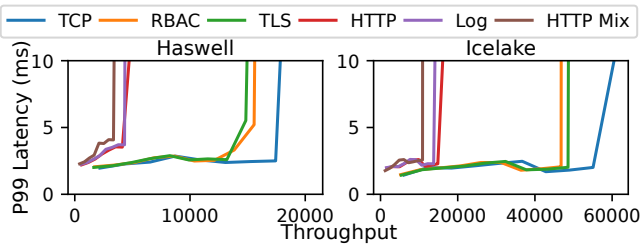


Figure 6: Tail latency vs throughput for various configurations across processor generations. Throughput decreases by 2.4× when operators switch to HTTP-based from TCP-based policies for service access control.

throughput change significantly as we configure sidecars with different policies. Although, the trend across the policies are same, newer processors experience a 3× boost in maximum query capacity as seen in Fig. 6. We also observe a distinct improvement of around 20% on tail latency across policies too. This motivates our interest in evaluating and understanding micro-architectural implications on sidecars.

4 Experimental Setup

Our methodology includes diverse sidecar filters, applications and deployment settings like platform hardware, cluster size and message sizes to quantify sidecar performance.

Application micro-benchmarks: We study sidecar's performance using two single-service microbenchmarks.

(1) Socketify EchoServer [85]: Fast Python based echo-server – augmented to modify response size based on the request parameter.
(2) MySQL [61]: A relational database configured with synthetic tables for use with SQL query filters.

Filter Microbenchmarks: We use Envoy [28] as our sidecar, given its widespread adoption in service meshes [14, 40, 49] and extensive library of 80+ filters. For our evaluation, we select 9 representative filters – 2 TCP filters and 7 application-layer filters, including rate-limiting, TLS, logging, and MySQL. These cover a range of compute and I/O intensive tasks. While these filters suffice for our characterization, the testbed can easily extend to other filters.

(1) `TCP`: Simple L3/L4 Router.
(2) `RBAC`: TCP-IP/Port based access control policies to rejects packets from all or range of IP/Ports. Variants used: "Accept All", "Reject 1", "Reject 100" and "Reject 1000".
(3) `SQL`: MySQL filter that parses queries and emits metadata.
(4) `HTTP`: Simple L7 API Router.

Node	Intel Xeon E5-2683v3	Intel Xeon Silver 4314
Cores	28-core dual socket	16-core (SMT Off)
I-Cache	32KB L1	32KB L1
D-Cache	32KB L1;256KB L2	48KB L1;1.25MB L2
iTLB	128-entry 4-way Shared	128-entry 8-way Shared
LLC	70 MB (56 cores)	24 MB (16 cores)
Freq.	2GHz (no cstates)	2.4GHz (no cstates)
Memory	256G 2133MHz DDR4	128G 3200MHz DDR4
Kernel	Linux 5.4 (ubuntu20)	Linux 5.15 (ubuntu20)

Table 1: Node Specifications.

(5) `Header Read`: Reads packets headers and sets sidecar metadata appropriately. It does not modify the packet.
(6) `Ratelimit`: Limits the request rate to a service. Filter terminates excess requests with error responses.
(7) `IP Tag`: Modifies packet and inserts new headers with value based on client's IP. Variants used: "IP Tag1","IP Tag 5","IP Tag 10" based on number of inserted headers.
(8) `Logging`: Configurable logs of service invocations.
(9) `TLS`: Operator configuration commonly used to provide en(de-)cryption of the payload at pod boundaries.
(10) `HTTP Mix 1`: Combined `Ratelimit` and `IP Tag`.
(11) `HTTP Mix 2`: All policies combined except `TLS`.

Hardware and software details: All of our experiments and results are conducted on Cloudlab [26] using Intel Xeon Silver 4314 (IceLake) nodes. Details about node configuration can be found in Table 1. For brevity we limit our analysis to IceLake machines here and provide extended evaluations on dual socket Intel Xeon E5-2683 v3 (Haswell) in Appendix B. For container orchestration we use Istio (v1.20.3) [40] on Kubernetes (v1.28), and wrk2 [93] as our load generator operating on the worker node to mitigate datacenter network latency. All benchmark applications run at maximum throughput while the microbenchmark experiments run at varying the query rates to achieve low (20%), medium (50%) and high (100%) CPU utilization. Our study utilizes a 100-byte payload for most of the experiments as shown to be a representative message size in prior studies [99]. However, we also study the impact of larger message sizes in Appendix B. All microbenchmark studies have sidecar processes pinned to 1 CPU core with no SMT enabled to reduce interference from workloads and benefit from a warm micro-architecture. We use `perf` (performance counters), `strace` and `pmu-tools` [71] to collect profiling information and Intel PIN [54](v3.30-98830-1d7b601b3) to collect execution traces.

5 Characterization and Evaluation

Here we discuss the evaluation results of sidecar processes by addressing two noted limitations of existing studies – 1. Lack of diverse sidecar policies by incorporating representative filters in our microbenchmark for comprehensive study, and 2. Micro-architecture visibility gaps by evaluating extensive low-level metrics across platforms, revealing patterns for design exploration and optimization.

5.1 Application Topology

Service meshes incorporate sidecar processes for each service instance, making a user request traverse multiple sidecars as the request gets processed by the microservice application. Each user request pays the performance overhead of each sidecar process in its call path, that aggregates to significant overhead for large applications. Prior work [97] have shown such call paths to be typically 8-10 microservice long. Since the microservice graph and communication patterns do not influence sidecar behavior as we scale-up applications, we do not see a significant change in sidecar overheads as we scale out to large cluster sizes. We also note that sidecar processes being decoupled from underlying microservice, exhibiting performance behavior that is influenced by configured policies rather than the service they are associated with. We detail these experiments and results in Appendix A.

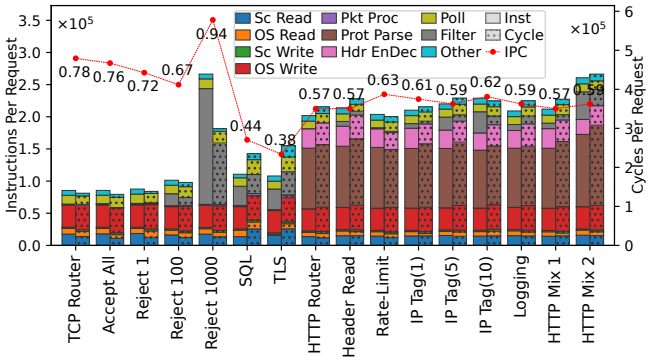


Figure 7: Performance overhead across different sidecar configurations per request. Filters contribute from 1%-10% cycles overhead, while protocol parsing and OS write are the largest contributors of overhead.

5.2 Configuration Microbenchmarking

As we saw in Section 2, a network packet traverses through various sidecar components before it is processed by the operator-configured filters. We categorize this lifecycle into functional components described in Table 2. We use diverse microbenchmark filters described in Section 4 to understand their performance impact.

Fig. 7 shows instruction and cycle distribution across benchmarks. HTTP policies spend 50% of CPU cycles on protocol parsing, with write syscalls as the next largest contributor. For TCP policies, write syscalls dominate. Most components, except filters, show uniform latency across microbenchmarks. Filters, though small (<5% instruction overhead), can use up to 15% of CPU cycles.

Although cycles and instructions overhead go hand-in-hand, we see that the instruction-per-cycle (IPC) metric reveals efficiency differences in the TCP and HTTP filters. IPC estimates the instruction throughput of a processor and is indicative of architectural execution efficiency. Modern processors are designed for higher IPC (1.5-2 [69]) for benchmark applications by leveraging locality and program patterns but our results show TCP policies exhibiting an IPC of 0.7-0.9, while HTTP-based policies are lower than 0.6.

These results bring out two interesting facets of how filters and configuration can affect overall performance. First, we see a gradual decrease in IPC across the TCP and HTTP policies as we increase complexity of the filters (e.g. larger allow/deny lists and filter-mixes). But as we increase the filters significantly in "Reject 1000", the IPC increases because of highly reused logic that benefits from both cache reuse and branch predictors. Other components see the program behavior switch quicker than the hardware is able to warm up and predict. Second, Table 3 shows that the performance overhead of filters is additive in nature. Composing the cycle latencies of individual filter components gives a close upper-bound on the cycle latency for aggregated filter configuration.

Component	Description
OS Read	Kernel network stack read on packet receive.
Sidecar Read	Userspace packet read by sidecar.
Packet Processing	TCP/IP packet processing in sidecar.
HTTP Protocol Parse	HTTP protocol parsing in sidecar for L7 filters.
Header En/Decoder	En(De-)coder logic for any header-specific filter.
Sidecar Filter	User defined filter logic applied to messages.
Event Poll	I/O notif processing & polling event handler.
Sidecar Write	Reassembly of packet & invoke write syscall.
OS Write	Update destination & write to network buffer.

Table 2: Functional components of a sidecar proxy.

	Filter cycles		Total cycles	
Policy	Actual	Estimate	Actual	Estimate
HTTP	-	-	350.5k	-
Header Proc.	9.67k	-	364.6k	360.2k(+1.22%)
Rate Limit	4.16k	-	331.7k	354.7k(-8.2%)
IP Tag (1)	8.52k	-	338.7k	359.1k(-8.7%)
Logging	20.45k	-	358.9k	370.9k(-9.6%)
HTTP Mix 1	12.9k	12.68k(1.72%)	373.9k	363.4k(-0.66%)
HTTP Mix 2	41.83k	42.8k(-2.33%)	439.1k	392.3k(1.23%)

Table 3: Actual and estimated cycles for HTTP filters.

5.3 Micro-architectural Investigation

Hardware events can yield fine-grained visibility into IPC bottlenecks.

Topdown Analysis: A top-down approach quantifies the time spent across various stages in a processor, namely 1. Frontend: which fetches and decodes instructions; 2. Backend: which processes and computes the instruction data; 3. Retiring: which finally commits processed instructions and 4. Speculation: which represents pipeline cycles lost due to a bad branch prediction. Fig. 8 plots the cycle breakdown across the four stages of a processor pipeline which reveals more than half of cycles are spent in the frontend irrespective of the filter and load. This indicates lack of valid instructions, resulting in stalls for useful instructions.

Frontend Behavior: We investigate frontend stalls by analyzing instruction cache misses and branch behavior, including branch misses and branch types. Misses per kilo-instructions (MPKI) metrics indicate efficiency of micro-architectural components with high-performance systems aiming for low miss rates. Prior work [35] report about 10-12 MPKI for iCaches in microservices workloads.

Our experimental results in Fig. 9 reveal that sidecar processes exhibit a significantly higher iCache miss rate of up to 75 MPKI on Icelake machines. Fig. 10 indicates these machines face up to 13% cycles stalled in the processor frontend waiting for valid instructions, likely due to iCache misses. Our analysis also show that of all fetched instructions that encounter an iCache miss, only 25% actually execute and retire. This is indicative of a large number of outstanding misses prefetch from a wrong target and end up thrashing the iCache, resulting in higher miss rate.

Interestingly, we observe low branch miss rates across platform generations, with under 5 MPKI on Icelake machines, as shown in Fig. 9. In the interest of space, we list the

Branches	Total	Miss
All Retired	56.5k	4.5k
Cond. Taken	12.14k	1.3k
Cond. Not Taken	20.64k	1k
Indirect	3.6k	2k
Near Taken	35.4k	3.6k

Table 4: Breakdown of branch accesses and misses per request.

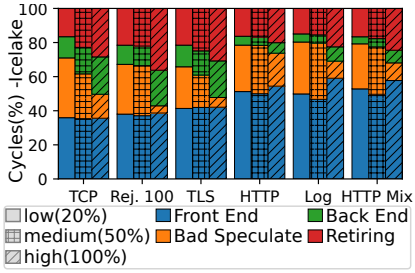


Figure 8: Cycle breakdowns reveal the processor frontend as the primary bottleneck, followed by the backend for data-intensive filters.

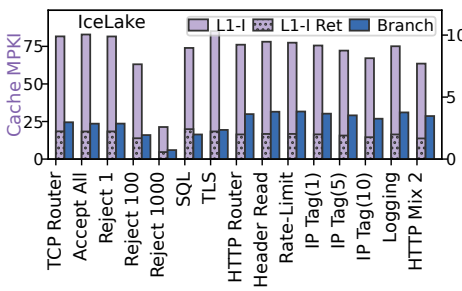


Figure 9: iCaches have high MPKI while retired iCache misses are only 25% of all misses. Branch predictors have low MPKI.

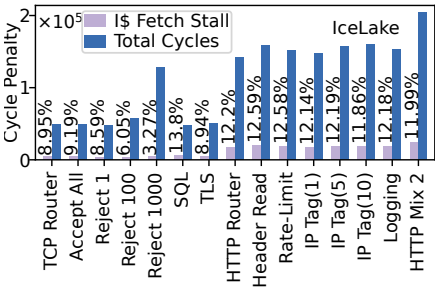


Figure 10: I-Cache misses cause frontend stalls that take 9-12% of all cycles in Icelake processors.

types of branch instructions for one policy (RateLimit) in Table 4. Branches account for less than 20% of instructions, mostly conditional. Indirect branches, under 5% of all branches, cause over 50% of branch misses. Poor predictability and deep pipeline resolution lead to costly flushes, validating that mispredicting indirect branches has high performance cost due to wasted cycles and iCache pollution.

In summary, we identify two key contributors to frontend bottlenecks. First, indirect branches dominate branch misses, severely impacting performance by wasting pipeline slots, triggering deep wrong-path executions, and polluting the iCache, leading to higher cache misses and stalls. While newer processors with larger L2 caches (1.25MB) mitigate some fetch miss costs, aggressive prefetchers still result in elevated L1 miss rates. Second, simple branch predictors perform well due to the prevalence of conditional branches. Enhancements targeting indirect branch prediction could yield significant improvements in frontend efficiency.

5.4 Dynamic code analysis

Our characterization highlights two avenues to improve performance for sidecar developers and architects aiming to provide efficient cloud-native hardware – reducing instruction fetch latency and improving branch misprediction costs.

Instruction fetch: Microservice applications typically use a containerized sidecar for each service instance. While each request processing requires some heap space for thread local storage, most of the code are read-only pages shared as copy-on-write. Envoy sidecars [28] reports two large shared code regions spanning 21MB and 40MB along with two private writable areas that are 2288kB and 248kB. However, each request typically touches a very small portion of the entire memory, called the working set size (WSS).

Existing tools for calculating working set sizes estimate by counting the referenced flags in the page map kept by Linux systems. However, for highly segmented applications like the sidecars, it does not provide accurate estimates. Our evaluation instruments the proxy during the run time using Intel PIN [54] to get an instruction trace. This trace is then used to calculate the actual number of cache lines used by the sidecar process. Since PIN only instruments retired instructions we get a lower bound estimate of the WSS.

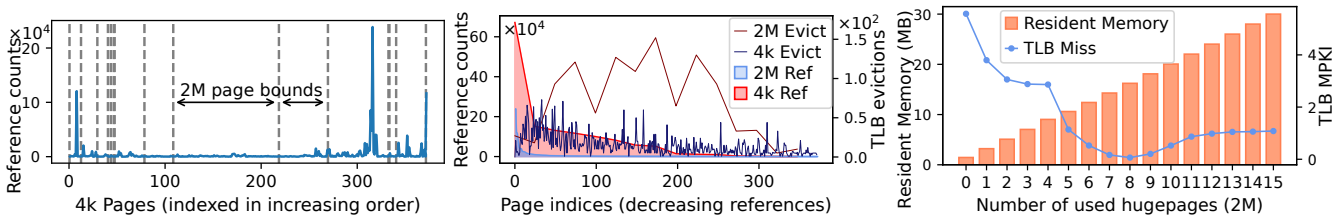
Although prior work [6] show an estimate of 4MB of WSS for cloud functions, our analysis puts a sidecar's WSS at 275-300kB across various microbenchmark policies – small enough for LLCs and large L2s but too large for L1! We also note that these cache lines map to about 350 4kB pages in the envoy binaries and shared libraries. We map the access pattern of the hot code regions in Fig. 11a. We also note huge-page boundaries for these instruction traces.

350 page accesses impart significant pressure on small L1-I iTLBs (64-entry 8-way 4k TLB, 8-entry fully associative hugepage iTLBs per thread [38]). Performance counters show about 1.5k iTLB misses per request of which about 1.2k misses do a page walk which can be an expensive process and introduce frontend stalls. We use a simple TLB simulator with an LRU (least recently used) replacement policy to simulate the iTLB behavior using extracted instruction traces. We note a similar rate of $\simeq$ 1.1k miss per request. Simulation results show that over 95% of the iTLB accesses occur in under 20 pages. However, 19 of these pages get swapped out as often as 6-60x in each request shown in Fig. 11b. Fig. 11c modifies the simulator to estimate iTLB misses while using hugepages. We see that misses decrease up to 80% when we use all huge pages (2M) but such a model would have very high memory usage. Using a combination of 4k and 2M huge pages on demand, reduce iTLB misses by up to 80% and 90% with 5 and 6 huge-page entries respectively. These have a moderate memory overhead of 8.5MB-11MB.

We also profile the WSS of Envoy functions to understand the cache requirements of high latency sections like the protocol parsing. Protocol parsing uses $\simeq$ 40-50kB of instruction memory indicating a constant thrash of caches that provides no discernible iCache reuse. While these are larger than current iCaches, future generations are slated to support 64k of iCaches. Core isolation with cache partitioning for critical sections can provide performance boosts.

6 Related Work

Microservices and Functions Rise of distributed application designs [45] have been driven by advances in containerization [18, 24, 56] and orchestration frameworks [25, 48] which provide users with a *serverless* experience by abstracting the hardware layer. Microservices differ from traditional cloud services, posing challenges in tail latency [32, 63, 73], performance [34, 86, 94], reliability [68], and security [2, 9, 10, 31]. Functions, popularized by AWS Lambda, excel in short, parallel tasks but struggle with startup latency. Significant prior research aims at understanding FaaS platforms [78, 81, 87, 88, 92, 98], performance [19, 77, 79], and isolation [1, 44, 83, 91]. However, learning from microservices workloads



(a) Proxy's Page Accesses Patterns **(b) iTLB evictions per page** **(c) iTLB misses while using using huge pages**

Figure 11: Instruction and iTLB access patterns across a sidecar invocation.

do not apply to sidecars. Microservices [35] and functions [79] have low frontend bottlenecks on warm executions, and checkpointing benefits such workloads but fails to address sidecars, which exhibit high frontend stalls even in back-to-back executions.

Understanding cloud platforms Characterization studies of orchestration platforms [75, 80, 99] reveal significant communication layer overheads, prompting novel communication schemes [19, 84] and network optimizations [80]. While we observe similar issues, our focus is on hardware offload designs for infrastructure components. Warehouse-scale studies [36, 46] highlight overheads from system tasks like compression, allocation, and growing instruction footprints, aligning with our observations on sidecar behaviors. However, sidecars, being highly replicated, require targeted program analysis and hardware optimizations for significant gains. Existing performance models [99] rely on tail-latency metrics, which vary across architectures. We propose microarchitectural metrics as a more accurate basis for predictive models and are working to integrate these into existing frameworks.

Offloading sidecars With efforts from academia [5, 51, 59, 70, 95] and industry [11, 64] to offload infrastructure tasks, there is an opportunity to bring insights from our micro-architectural evaluations for accelerating sidecar functionality to enable large and rich logic filters. Recent works [67] have also looked into offloading software network switches for high-performance data planes.

7 Conclusion

This paper brings attention to performance penalties associated with the use of service meshes and sidecars. We provide an in-depth characterization that quantifies sidecar overheads to build a performance model for predictable overheads when composing sidecar configurations. We also explore hardware bottlenecks in supporting such frameworks highlighting severe frontend bottlenecks and evaluating solutions that can reduce iTLB miss rates up to 85%.

Acknowledgments

We thank the reviewers for their insightful comments, Anjo Vahldiek-Oberwagner and Mattan Erez for their feedback, and members of SPARK Lab for the regular discussions and inputs. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[2] Angeliki Aktypi, Dimitris Karnikis, Nikos Vasilakis, and Kasper Rasmussen. Themis: A secure decentralized framework for microservice interaction in serverless computing. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, pages 1–11, 2022.

[3] Ambient mesh - simplify operations of the istio service mesh. https://www.solo.io/products/ambient-mesh/.

[4] Application logging — envoy 1.32.0-dev-bfa0e0 documentation. https://www.envoyproxy.io/docs/envoy/latest/configuration/observability/application_logging.

[5] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in {High-Speed}{NICs}. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 93–109, 2020.

[6] Grant Ayers, Nayana Prasad Nagendra, David I August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 462–473, 2019.

[7] Basic auth — envoy 1.32.0-dev-bfa0e0 documentation. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/basic_auth_filter.

[8] Istio / bookinfo application. https://istio.io/latest/docs/examples/bookinfo/.

[9] Ferdinand Brasser, Patrick Jauernig, Frederik Pustelnik, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Trusted container extensions for container-based confidential computing. *arXiv preprint arXiv:2205.05747*, 2022.

[10] Stefan Brenner, Tobias Hundt, Giovanni Mazzeo, and Rüdiger Kapitza. Secure cloud micro services using intel sgx. In *Distributed Applications and Interoperable Systems: 17th IFIP WG 6.1 International Conference, DAIS 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19–22, 2017, Proceedings 17*, pages 177–191. Springer, 2017.

[11] Brad Burres, Dan Daly, Mark Debbage, Eliel Louzoun, Christine Severns-Williams, Naru Sundar, Nadav Turbovich, Barry Wolford, and Yadong Li. Intel's hyperscale-ready infrastructure processing unit (ipu). In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–16. IEEE, 2021.

[12] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Shuang Chen, Christina Delimitrou, and José 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*, ASPLOS '19, page 107–120, New York, NY, USA, 2019. Association for Computing Machinery.

[14] Cilium - linux native, api-aware networking and security for containers. https://cilium.io/.

[15] Cni benchmark: Understanding cilium network performance. https://cilium.io/blog/2021/05/11/cni-benchmark/.

[16] CNCF. Cncf_service_mesh_microsurvey_final.pdf. https://www.cncf.io/wp-content/uploads/2022/05/CNCF_Service_Mesh_MicroSurvey_Final.pdf.

[17] CNCF. Cncf_survey_report_2020.pdf. https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf.

[18] containerd – an industry-standard container runtime with an emphasis on simplicity, robustness and portability. https://containerd.io/.

[19] Marcin Copik, Konstantin Taranov, Alexandru Calotoiu, and Torsten Hoefler. rfaas: Rdma-enabled faas platform for serverless high-performance computing. *arXiv preprint arXiv:2106.13859*, 2021.

[20] Cross-origin resource sharing (cors) - http | mdn. https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

[21] Getting started with apm tracing. https://docs.datadoghq.com/getting_started/tracing/.

[22] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2013.

[23] Christina Delimitrou and Christos Kozyrakis. HCloud: Resource-Efficient Provisioning in Shared Cloud Systems. In *Proceedings of the Twenty First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2016.

[24] Docker: Accelerated container application development. https://www.docker.com/.

[25] Swarm mode overview | docker docs. https://docs.docker.com/engine/swarm/.

[26] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*, pages 1–14, 2019.

[27] Http filters — envoy 1.27.0-dev-f2a6dc documentation. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/http_filters.

[28] Envoy proxy - home. https://www.envoyproxy.io/.

[29] Role based access control (rbac) filter — envoy 1.27.0-dev-70be00 documentation. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/rbac_filter.

[30] xds rest and grpc protocol — envoy 1.32.0-dev-bfa0e0 documentation. https://www.envoyproxy.io/docs/envoy/latest/api-docs/xds_protocol.

[31] Eduardo Falcão, Matteus Silva, Ariel Luz, and Andrey Brito. Supporting confidential workloads in spire. In *2022 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 186–193. IEEE, 2022.

[32] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

[33] Yu Gan and Christina Delimitrou. The Architectural Implications of Cloud Microservices. In *Computer Architecture Letters (CAL), vol.17, iss. 2*, Jul-Dec 2018.

[34] Yu Gan, Sundar Dev, David Lo, and Christina Delimitrou. Sage: Leveraging ML To Diagnose Unpredictable Performance in Cloud Microservices. In *Workshop on ML for Computer Architecture and Systems (MLArchSys)*, June 2020.

[35] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[36] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. Profiling hyperscale big data processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–16, 2023.

[37] Header mutation — envoy 1.32.0-dev-bfa0e0 documentation. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/header_mutation_filter.

[38] Intel icelake. https://www.7-cpu.com/cpu/Ice_Lake.html.

[39] Ip tagging — envoy 1.27.0-dev-70be00 documentation. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/ip_tagging_filter.

[40] Istio. https://istio.io/latest/.

[41] Istio / installation configuration profiles. https://istio.io/latest/docs/setup/additional-setup/config-profiles/.

[42] Istioldie 1.11 / performance and scalability. https://istio.io/v1.11/docs/ops/deployment/performance-and-scalability/.

[43] Jaeger: open source, distributed tracing platform. https://www.jaegertracing.io/.

[44] Shannon Joyner, Michael MacCoss, Christina Delimitrou, and Hakim Weatherspoon. Ripple: A Practical Declarative Programming Framework for Serverless Compute. In *arXiv:2001.00222 [cs.DC]*, January 2020.

[45] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat S. Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. Service fabric: a distributed platform for building microservices in the cloud. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[46] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 158–169, New York, NY, USA, 2015. Association for Computing Machinery.

[47] Home - knative.

[48] Kubernetes. https://kubernetes.io/.

[49] Kuma. https://kuma.io/.

[50] Joshua Levin and Theophilus A Benson. Viperprobe: Rethinking microservice observability with ebpf. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pages 1–8. IEEE, 2020.

[51] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 1–14, 2016.

[52] The world's most advanced service mesh. | linkerd. https://linkerd.io/.

[53] Benchmarking linkerd and istio: 2021 redux | linkerd. https://linkerd.io/2021/11/29/linkerd-vs-istio-benchmarks-2021/.

[54] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, jun 2005.

[55] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 412–426, New York, NY, USA, 2021. Association for Computing Machinery.

[56] lxc/lxc: Lxc - linux containers. https://github.com/lxc/lxc.

[57] Marblerun: the service mesh for confidential computing. https://www.edgeless.systems/products/marblerun.

[58] Challenges in microservices: Testing, monitoring, and debugging. https://asyx.com/2024/02/09/challenges-in-microservices-testing-monitoring-and-debugging/#:~:text=Debugging%20microservices%20can%20pose%20a,the%20asynchronous%20communication%20between%20microservices.

[59] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. {AccelTCP}: Accelerating network applications with stateful {TCP} offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 77–92, 2020.

[60] Rfc 8705 - oauth 2.0 mutual-tls client authentication and certificate-bound access tokens. https://datatracker.ietf.org/doc/html/rfc8705.

[61] Mysql. https://www.mysql.com/.

[62] Netflix. Netflix architecture: How much does netflix's aws cost? https://www.cloudzero.com/blog/netflix-aws.

[63] Minh Nguyen, Zhongwei Li, Feng Duan, Hao Che, and Hong Jiang. The tail at scale: how to predict it? In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[64] What is a dpu? | nvidia blog. https://blogs.nvidia.com/blog/whats-a-dpu-data-processing-unit/.

[65] Oauth2 — envoy 1.32.0-dev-bfa0e0 documentation. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/oauth2_filter.

[66] Googlecloudplatform/microservices-demo: Sample cloud-first application with 10 microservices showcasing kubernetes, istio, and grpc. https://github.com/GoogleCloudPlatform/microservices-demo.

[67] Heng Pan, Peng He, Zhenyu Li, Pan Zhang, Junjie Wan, Yuhao Zhou, XiongChun Duan, Yu Zhang, and Gaogang Xie. Hoda: a high-performance open vswitch dataplane with multiple specialized data paths. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 82–98, New York, NY, USA, 2024. Association for Computing Machinery.

[68] Aurojit Panda, Mooly Sagiv, and Scott Shenker. Verification in the age of microservices. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 30–36, 2017.

[69] Reena Panda, Shuang Song, Joseph Dean, and Lizy K. John. Wait of a decade: Did spec cpu 2017 broaden the performance horizon? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282, 2018.

[70] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A programming system for {NIC-Accelerated} network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, 2018.

[71] andikleen/pmu-tools: Intel pmu profiling tools. https://github.com/andikleen/pmu-tools.

[72] Proto message extraction — envoy 1.32.0-dev-bfa0e0 documentation. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/proto_message_extraction_filter.

[73] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. {FIRM}: An intelligent fine-grained resource management framework for {SLO-Oriented} microservices. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 805–825, 2020.

[74] Francisco Romero and Christina Delimitrou. Mage: Online and Interference-Aware Scheduling for Multi-Scale Heterogeneous Systems. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT18)*, November 2018.

[75] Prateek Sahu, Lucy Zheng, Marco Bueso, Shijia Wei, Neeraja J Yadwadkar, and Mohit Tiwari. Sidecars on the central lane: Impact of network proxies on microservices. *arXiv preprint arXiv:2306.15792*, 2023.

[76] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. ServiceRouter: Hyperscale and minimal cost service mesh at meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 969–985, Boston, MA, July 2023. USENIX Association.

[77] David Schall, Artemiy Margaritov, Dmitrii Ustiugov, Andreas Sandberg, and Boris Grot. Lukewarm serverless functions: Characterization and optimization. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 757–770, New York, NY, USA, 2022. Association for Computing Machinery.

[78] David Schall, Andreas Sandberg, and Boris Grot. Warming up a cold front-end with ignite. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 254–267, 2023.

[79] David Schall, Andreas Sandberg, and Boris Grot. Warming up a cold front-end with ignite. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 254–267, New York, NY, USA, 2023. Association for Computing Machinery.

[80] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. A cloud-scale characterization of remote procedure calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 498–514, 2023.

[81] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, pages 1063–1075, 2019.

[82] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.

[83] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433, 2020.

[84] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 138–152, 2021.

[85] cirospaciari/socketify.py: Bringing http/https and websockets high performance servers for pypy3 and python3. https://github.com/cirospaciari/socketify.py.

[86] Akshitha Sriraman and Thomas F Wenisch. μ suite: a benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12. IEEE, 2018.

[87] Jovan Stojkovic, Chunao Liu, Muhammad Shahbaz, and Josep Torrellas. umanycore: A cloud-native cpu for tail at scale. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[88] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. Mxfaas: Resource sharing in serverless environments for parallelism and efficiency. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[89] Tap — envoy 1.32.0-dev-bfa0e0 documentation. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/tap_filter.

[90] FreeWheel Biz-UI Team. Microservice traffic management. In *Cloud-Native Application Architecture: Microservice Development Best Practice*, pages 109–152. Springer, 2024.

[91] Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. Clemmys: Towards secure remote execution in faas. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 44–54, 2019.

[92] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 559–572, 2021.

[93] giltene/wrk2: A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2.

[94] Juncheng Yang, Yao Yue, and KV Rashmi. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)*, 17(3):1–35, 2021.

[95] Jie Zhang, Hongjing Huang, Lingjun Zhu, Shu Ma, Dazhong Rong, Yijun Hou, Mo Sun, Chaojie Gu, Peng Cheng, Chao Shi, et al. Smartds: Middle-tier-centric smartnic enabling application-aware message split for disaggregated block storage. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–13, 2023.

[96] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, Ed Suh, and Christina Delimitrou. Sinan: Data-Driven Resource Management for Interactive Microservices. In *Workshop on ML for Computer Architecture and Systems (MLArchSys)*, June 2020.

[97] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothfy Sherwood, and Milind Chabbi. {CRISP}: Critical path analysis of {Large-Scale} microservice architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 655–672, 2022.

[98] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Everywhere all at once: Co-location attacks on public cloud faas. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, page 133–149, New York, NY, USA, 2024. Association for Computing Machinery.

[99] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. Dissecting overheads of service mesh sidecars. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 142–157, New York, NY, USA, 2023. Association for Computing Machinery.

[100] Openzipkin · a distributed tracing system. https://zipkin.io/.

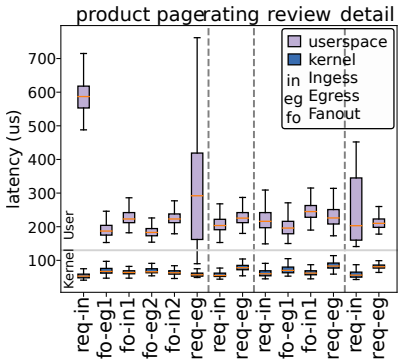


Figure 13: Sidecar latency per request in Bookinfo. Kernel accounts 20% latency; userspace don't vary across services.

A Appendix: System Level Implications

This section explores system behavior across various deployment settings to isolate and quantify the variability in performance degradation in sidecars to provide visibility into the trade-offs of using service meshes across various operational settings.

For these experiments, we use three benchmark workloads and two benchmark configurations for sidecars as provided by Istio [41].

- W1 BookInfo [8]: Small library web-app consisting of 4 services that allows users to get reviews and information about books.
- W2 Online Boutique [66]: E-commerce app consisting 11 services that allows browsing, shopping cart, and a payment page.
- W3 Hotel Reservation [35]: An app with 17 services providing hotel search based on location, rates and availability.
- C1 Default: Configuration combines HTTP-based RBAC filters, stats collection, header management [20], and some request tracing.
- C2 Demo: Showcase configuration that combines all 'Default' filters while logging and tracing all requests.

A.1 Deployment configurations and settings

We evaluate how real world datacenter settings and operator choices impact the overall service level objective using our benchmark applications. In Fig. 5, we analyze P90 latency and throughput overheads as cluster sizes scale from 1 to 5 nodes. Latency remains consistent but slightly increases due to cross-node network factors while throughput slightly degrades at larger scales, independent of sidecars. To quantify penalties, we measure CPU instructions and cycles, which reflect the additional work and resources consumed in Fig. 12. Experimental setups compare benchmark applications with and without service mesh, where sidecars are configured with no traffic control policy with minimal TCP-based access control and observability policy – ensuring similar application behavior.

Application behavior remains steady across cluster sizes if nodes have sufficient resources. However, deployment settings show variable degradation. The "Default" configuration incurs 20–85% cycle and 13–75% instruction penalties, while the "Demo" configuration nearly doubles these overheads. Sidecar performance remains consistent across cluster sizes for each benchmark but varies across applications and configurations. Instructions and cycles emerge as key metrics for precisely and uniformly measuring overheads.

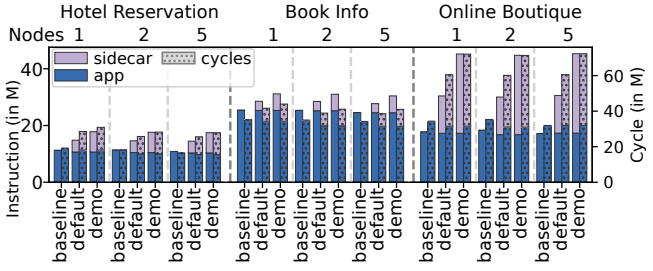


Figure 12: Instructions and cycle counts across different clusters and different sidecar configurations shows constant overhead across cluster sizes, but varies across different configuration profiles. Instruction overhead vary from 13%-76% for 'default' and 23%-162% for 'demo' while cycles see a range of 21%-91% for 'default' and 28%-128% for 'demo'.

A.2 Application Topology

Here we evaluate how service topologies of applications might impact sidecar latency. Fig. 13 plots the latency distribution of system-calls for all invocations of sidecar instances in the BookInfo application. We use `request_in(e-)gress` and `fanout_in(e-)gress` terms to indicate ingress and egress paths to and from a service instance from the downstream client or upstream instances respectively. We find that requests typically take between 200-250 ms for a sidecar invocation, except ProductPage. ProductPage's sidecar that handles external ingress, adds 200–400 ms for socket setup and longer egress times due to a 6KB HTML payload.

Sidecar invocation counts per user request vary for different services in an application, e.g. 6 for ProductPage, 4 for Review and 2 each for Rating and Detail in BookInfo app. Fig. 14 shows average instructions and cycles per sidecar invocation across benchmarks, confirming uniform performance penalties across services. "Default" configurations incur 200–300K instructions and 500–600K cycles, while "Demo" configurations face higher penalties of 400–500K instructions and 650–800K cycles. Frontend and Search (Hotel Reservation) and Review (BookInfo) are outliers due to amortization effects due to higher invocation counts, unlike ProductPage and Home, which handle HTML payloads, raising their averages.

Takeaway: Performance varies by topology and call path but sidecar latency is predictable per configuration. Overall penalty correlates with sidecar policies and configurations.

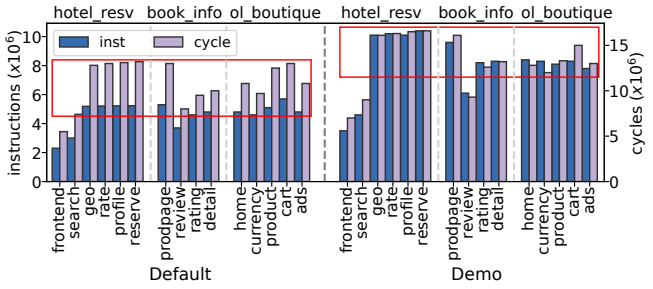


Figure 14: Instructions and latency across different configurations per sidecar invocation show roughly similar overhead independent of service.

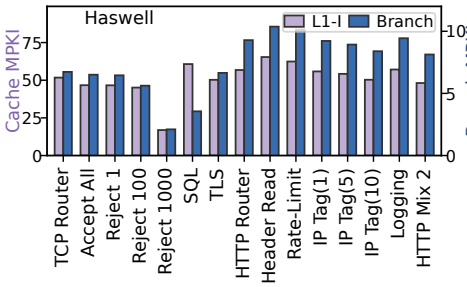


Figure 15: iCaches have high MPKIs. Branch predictors have low MPKI.

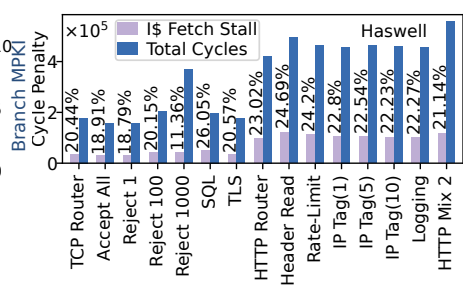


Figure 16: Haswell systems see 17-21% of cycle stalls due to iCache miss.

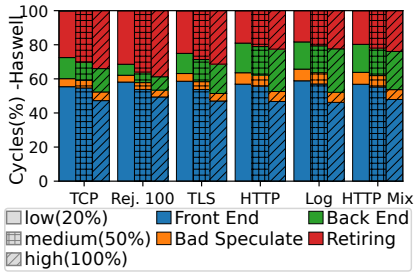


Figure 17: Haswell processors exhibit frontend and backend behavior similar to Icelake servers.

B Appendix: Extended Characterization Results

B.1 Comparing hardware generations

We use this section to present our microarchitectural evaluations on an older Haswell machine and compare them to the results we previously discussed for Icelake servers. Figs. 15 to 17 plots branch and instruction cache miss behavior, indicating significant frontend stalls and bottlenecks. Although, these trends align with our analysis in Section 5.3, we would like to point to a few interesting observations.

We note that the iCache fetch stalls for Haswell are much higher (12-25%) compared to newer generation processors. However, the L1 iCache MPKI trends lower than Icelake servers. This indicates that better predictors and prefetchers do not work well for infrastructure components like sidecars, and might even contribute to larger cache pollution. We also note that significantly larger private L2 caches helps with such workloads that are common, repetitive and have large instruction footprints.

B.2 Message Size

Across our benchmark applications, we note a diverse range of message sizes from a few bytes of message for Details (in BookInfo) to 532KB of HTML data for Online Boutique's home page. This section provides visibility into the performance overhead experienced by the sidecar components as we vary the message sizes. Our microbenchmarks evaluate a large range of messages starting from 16KB up to 1MB. Figure 18 plots the number of instructions and cycles as we increase the payload sizes across a set of filter configurations. At large payload sizes, we see an interesting interplay between the performance impact and the underlying microarchitecture. For brevity, we omit components that do not show much variance. "Protocol Parsing" also shows low variability since these program phases mainly processes the packet header portions of the messages which remains unaffected in this setting.

Filter: Both the HTTP and TCP filters work on packet headers and see no variance with message sizes. Data intensive configurations like TLS encounter a steep increase in computation and latency since this filter works on the entire payload message for encryption/decryption before further processing.

OS Read/Writes: OS read and write functions also show similar exponential growth in instructions and cycles. However, we notice that the IPC remains fairly consistent until 16KB message size, when it starts to sharply decrease to almost 0.25 IPC for 1MB payloads. This is interesting because the IPC drop aligns with our cache sizes. Both kernel read and write functions iterate over the entire packet while copying between kernel and userspace buffers. This results in both cold and capacity misses in private data caches as we increase the packet sizes above 16KB. TLS write filter deviates from the observed trends. With a single application-sidecar instance, packet decryption happens at kernel read on req-ingress, and encryption happens at kernel write on req-egress. Only the response from sidecar to client undergoes a variable message encryption which gets reflected on the performance of kernel write. Since this is the only filter that actively works on data, the data is actually in the caches when the kernel writes attempts to copy and hence it sees an elevated and constant IPC.

Sidecar Read/Writes: These functions mainly initiate a read/write system call on event triggers and updates buffer data for further processing in the sidecar. As they do not work on the data themselves, we see a nearly constant IPC. With a slightly predictable loop behavior with higher packet sizes, we notice an uptick in the IPC across different configurations.

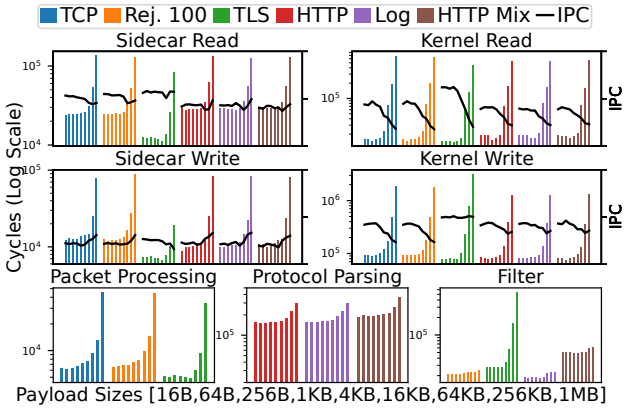


Figure 18: Impact of payload size on configurations: Compute heavy filters (TLS) see exponential overheads. Performance/IPC drops at cache level sizes during data copy depending on data use in the sidecars.