# Understanding Acceleration Opportunities for Microservice Overheads at Hyperscale

**Hailiang ZHAO @ ZJU-CS**

*http://hliangzhao.me*

June 13, 2024

*A lecture slide based on the paper —— Sriraman et al., Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale, in: ASPLOS '20.*

# Outline

# Outline

# The Microservice Deployment Model

There is growing trend towards microservice implementation models, wherein **a complex application is decomposed into *distributed microservices***, that each provide specialized functionality, such as

- HTTP connection termination,
- key-value serving,
- protocol routing,
- ad serving, ....

At hyperscale, this deployment model uses standardized Remote Procedure Call (RPC) interfaces to invoke several microservices to serve a user's query.[1]

[1] Communication paradigms include RPC, internal communication, and message queue.

# The Question

Upon receiving an RPC, a microservice must often perform operations such as I/O processing, decompression, deserialization, and decryption, **before it can execute its core functionality** (e.g., key-value serving).

Important microservices grow to account for an enormous *installed base* of physical hardware. Then we should ask the question ——

1. *Which microservice operations consume **the most** CPU cycles?*
2. *Are there **common overheads** across microservices that we might address when designing future hardware?*

# Always Characterization First!

The authors undertake a comprehensive characterization of microservices' CPU overheads on *Facebook production systems* serving live traffic.

1. Since microservices must invoke common leaf functions at the end of a call trace (e.g. `memcpy()`), the authors first characterize the leaf function overheads.
2. Then, the authors characterize microservice functionalities to determine (1) *whether diverse microservices execute* ***common types of operations*** *(e.g., compression, serialization, and encryption), and (2) the overheads they induce.*

# Characterization Result

## Observation

*Several microservices (despite their diversity in core logic) spend only* ***a small fraction*** *of execution time serving* ***core application logic****, squandering signifcant cycles facilitating the core logic via* ***orchestration work*** *that is not core to the application logic (e.g., compression, serialization, and I/O processing).*

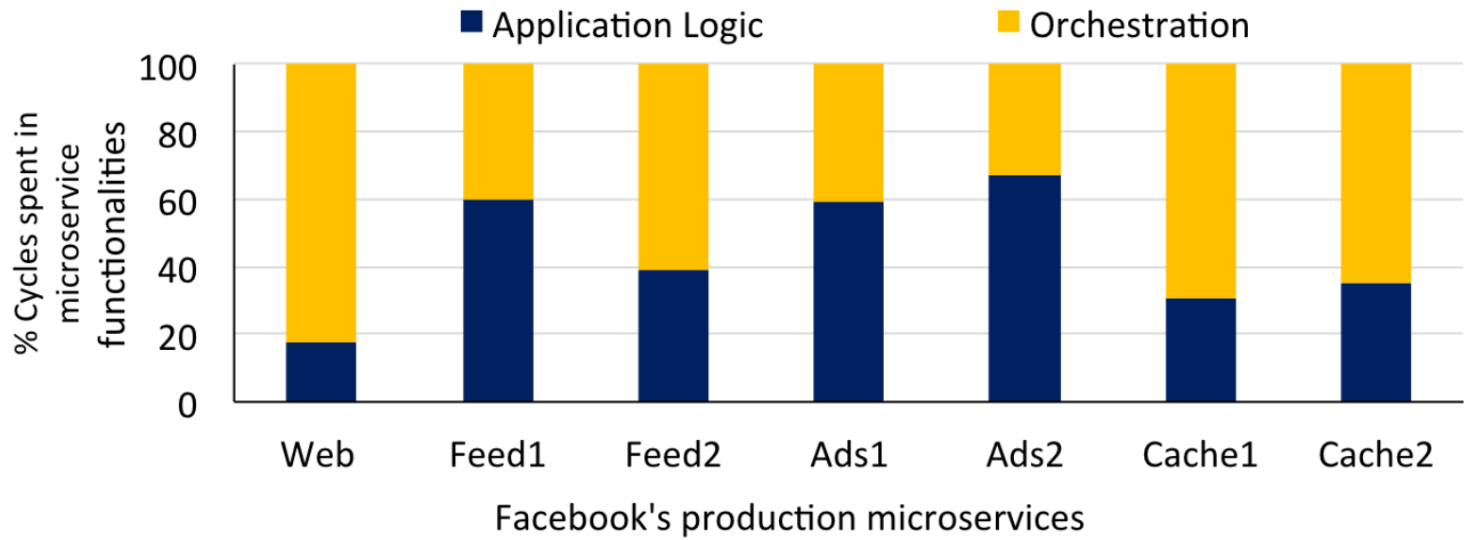


Figure 1: Breakdown of cycles spent in application logic vs. orchestration work.

# The Analytic Model

Before introducing hardware acceleration, **there is a need for simple analytical models** that identify performance bounds early in the hardware design phase to project gains from accelerating overheads.

The authors then develop an analytical model for hardware acceleration, *Accelerometer*, that identifes performance bounds to project microservice speedup.

1. *Accelerometer considers that the offload operates can be* ***asynchronous****.*
2. *Accelerometer realistically models microservice speedup for various hardware acceleration strategies (e.g.,* ***on-chip*** *vs.* ***off-chip****).*

# Outline

# Characterization Setup

**Production microservice.** There are 7 microservices in four diverse service domains that account for a large portion of Facebook's data center fleet.

- Web: *serve web requests*
- Feed1 and Feed2: *generate new feeds with model inference*
- Ad1 and Ad2: *maintain user-specifc and ad-specifc data and do ads recommendation via model inference*
- Cache1 and Cache2: *large distributed memory object caching service*

# Characterization Setup

**Hardware platforms.** The authors run Web, Feed1, Feed2, and Ad1 on the 18-core Skylake, and Ad2, Cache1, and Cache2 on the 20-core Skylake. They study IPC scaling across 3 CPU generations.

| | GenA | GenB | GenC |
|---|---|---|---|
| **$\mu$architecture** | Intel Haswell | Intel Broadwell | Intel Skylake |
| **Cores / socket** | 12 | 16 | 18 or 20 |
| **SMT** | 2 | 2 | 2 |
| **Cache block size** | 64 B | 64 B | 64 B |
| **L1-I$ / core** | 32 KiB | 32 KiB | 32 KiB |
| **L1-D$ / core** | 32 KiB | 32 KiB | 32 KiB |
| **Private L2$ / core** | 256 KiB | 256 KiB | 1 MiB |
| **Shared LLC** | 30 MiB | 24 MiB | 24.75 or 27 MiB |

Figure 2: GenA, GenB, and GenC CPU platforms' attributes.

# Characterization Setup

**Experimental setup.** The authors measure each microservice in production environment's default deployment, i.e., *standalone with no co-runners on bare metal hardware.*

There are no cross-service contention or interference effects. The authors study each system *at peak load* to stress performance bottlenecks.

# Characterization Setup

**Experimental setup (cont'd).** The authors first use Strobelight[2] to

- *collect all function call traces of a microservice (a function call trace is a call sequence starting with cloning a thread and ending with a leaf function such as* `memcpy()`*), and*
- *measure cycles and instructions spent in each call trace.*

Then, they feed the function call traces and their cycle counts to an internal tool[3] that *buckets each function call trace into a microservice functionality category* (e.g., I/O, serialization, and compression); it then aggregates cycles spent in each category.

[2] See `https://github.com/facebookincubator/strobelight`.

[3] Any similar open-sourced tools that we can use?

# Leaf Function Characterization

To determine a category's IPC, the authors determine the ratio of aggregated instruction and cycle counts for functions in that category.

The categories of leaf functions are presented in Table 3.

| Leaf category | Examples of leaf functions |
|---|---|
| Memory | Memory copy, allocation, free, compare |
| Kernel | Task scheduling, interrupt handling, network communication, memory management |
| Hashing | SHA & other hash algorithms |
| Synchronization | User-space C++ atomics, mutex, spin locks, CAS |
| ZSTD | Compression, decompression |
| Math | Intel's MKL, AVX |
| SSL | Encryption, decryption |
| C Libraries | C/C++ search algorithms, array & string compute |
| Miscellaneous | Other assorted function types |

Figure 3: Categorization of leaf functions.

# Breakdown of Cycles Spent in Leaf Functions

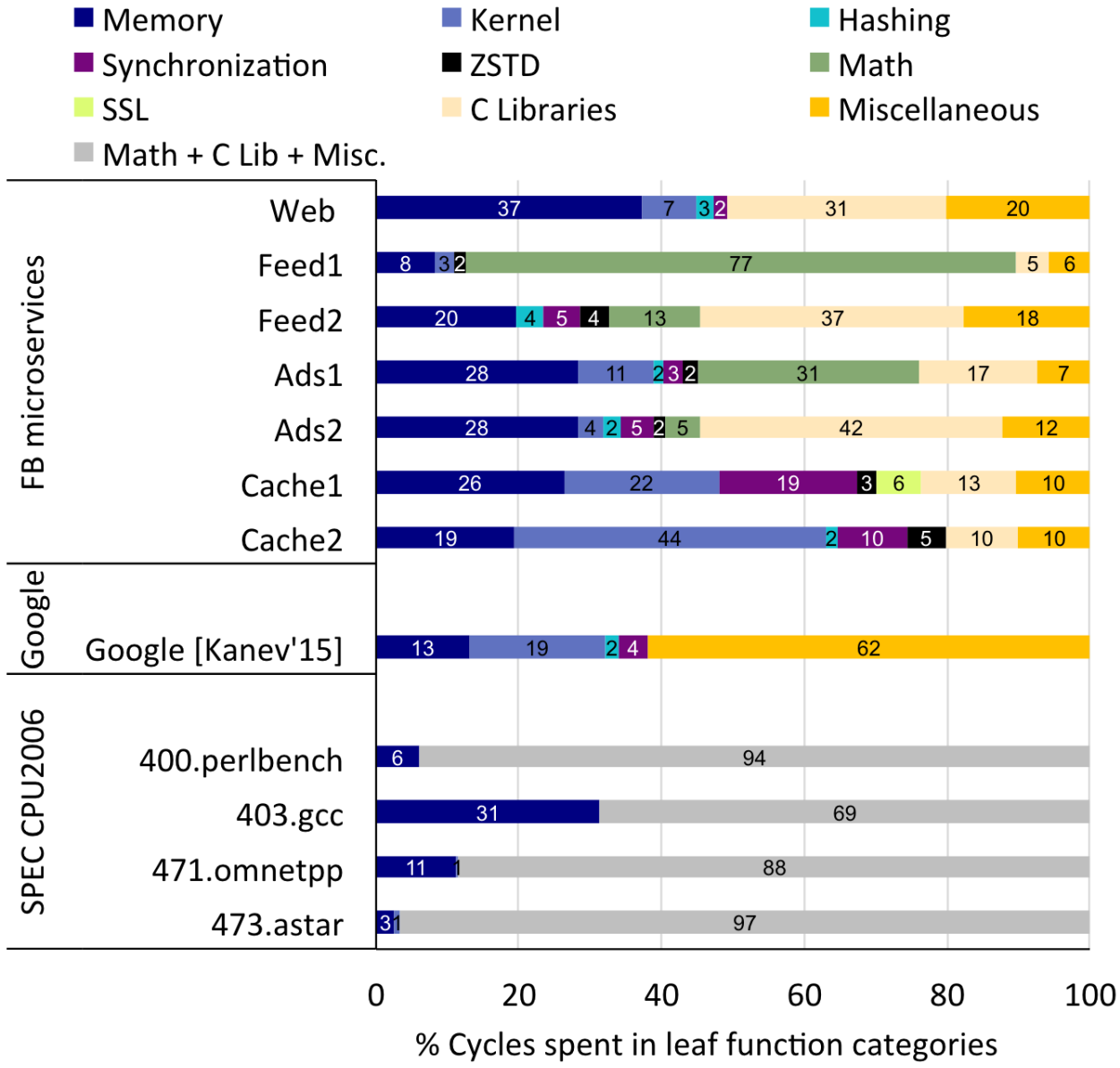


Figure 4: Breakdown of cycles spent in leaf functions.

# Observations

We make several observations from Fig. 4.

- Most microservices spend a signifcant fraction of cycles on **memory functions** (e.g., copy and allocation) and **kernel operations**.
- Cache1 and Cache2 spend more cycles in the **kernel** as they frequently incur context switches due to a high service throughput (invoke scheduler frequently).
- ML microservices such as Ad2 and Feed2 spend **only up to 13%** of cycles on mathematical operations that constitute ML inference using Multilayer Perceptrons.[4]

[4] ML services can also beneft from optimizations to C libraries. Details will be presented in the following.

# Observations (Cont'd)

We make several observations from Fig. 4 (Cont'd).

- Cache1 and Cache2 spend signifcant cycles **synchronizing frequent communication** between distinct thread pools.
- Cache1 spends 6% of cycles in leaf **encryption** functions since it encrypts a high number of QPS.

# Leaf Function Characterization in Detail

Many leaf function overheads are signifcant and *common across services*. In the following, we present the characterization of leaf functions in greater detail to identify acceleration opportunities.

- Memory
- Kernel
- Synchronization
- C Libraries

# Memory

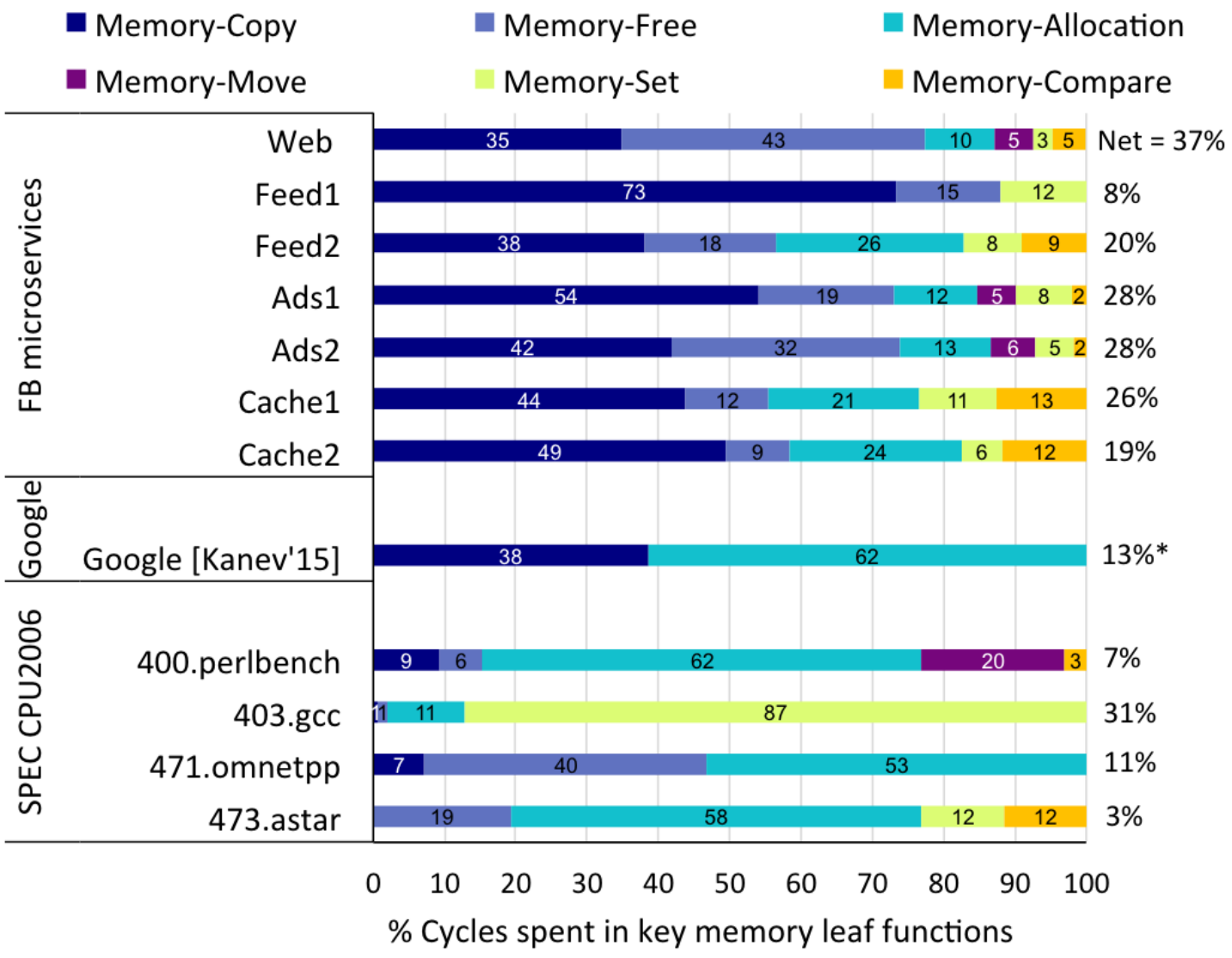


Figure 5: Breakdown of cycles spent in memory leaf functions.

# Memory

**Observations.**

- **Memory copies** are by far the greatest consumers of memory cycles.
- Google's services also spend 5% (0.38 $\times$ 13%) of total feet cycles on memory copies.

Memory copy optimizations include (1) reducing copies in network protocol stacks, (2) performing dense memory copies vis SIMD, (3) moving data in DRAM, (4) minimizing I/O copies using Intel's I/O Acceleration Technology (IO AT), (5) processing in memory, (6) optimizing memory-based software libraries, and (7) building hardware accelerators.

# Memory

**Observations (Cont'd).** Freeing memory incurs a high overhead for several microservices, as the `free()` function does not take a memory block size parameter, *performing extra work to determine the size class to return the block to.* TCMalloc performs a **hash lookup** to get the size class. This hash tends to cache poorly, especially in the TLB, leading to performance losses.[5]

Although C++11 ameliorates this problem by allowing compilers to invoke `delete()` with a parameter for memory block size, overheads can still arise from (1) *removing pages faulted in when memory was written to*, and (2) *merging neighboring freed blocks to produce a more valuable large free block.*

[5] A translation lookaside buffer (TLB) is a memory cache that stores the recent translations of virtual memory to physical memory.

# Memory

**Memory Copy Origins.** We observe diversity in dominant service functionalities that invoke copies across microservices.

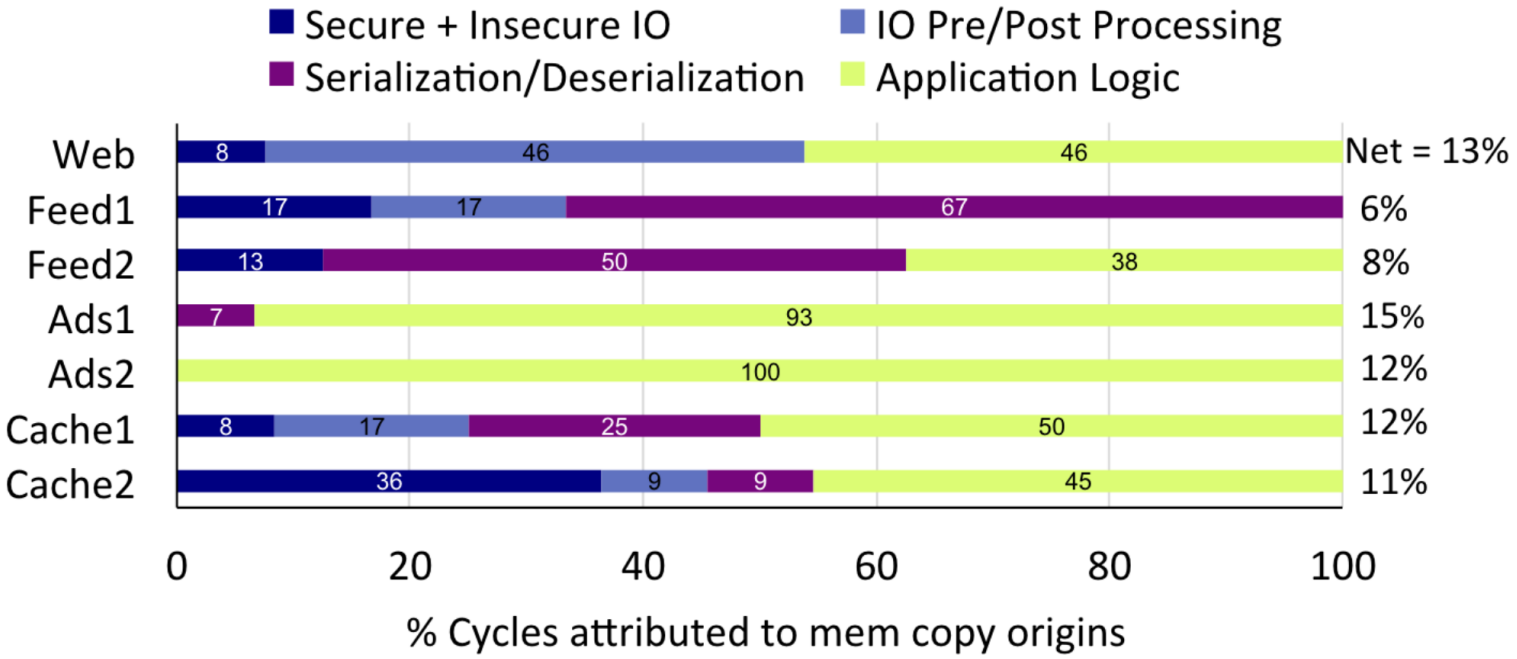


Figure 6: Breakdown of service functionalities that invoke memory copies.

For example, Web can beneft from *reducing copies in I/O pre- or post-processing*, whereas Cache2 can gain from *fewer copies in network protocol stacks*.

# Kernel

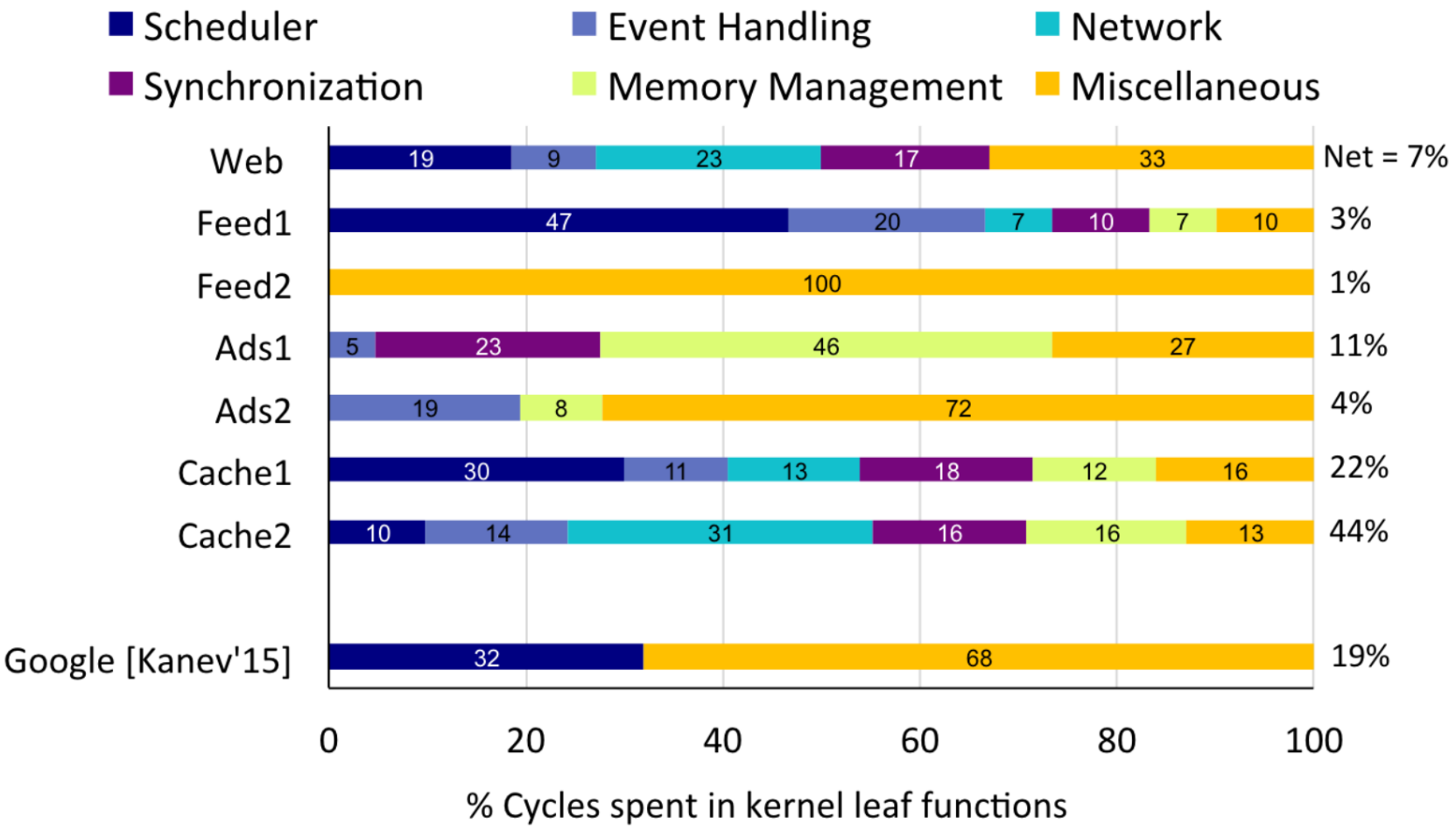


Figure 7: Breakdown of cycles spent in various kernel leaf functions: kernel scheduler, event handling, and network overheads can be high.

# Kernel

**Observations.**

- Microservices with a high kernel overhead —— Cache1 and Cache2 —— invoke scheduler functions frequently. *Software/hardware optimizations that reduce scheduler latency (e.g., **intelligent thread switching and coalescing I/O**) might considerably improve Cache performance.*
- Cache2 spends signifcant cycles in I/O and network interactions. *Optimized systems that incorporate **kernel-bypass and multi-queue NICs** might minimize* Cache2*'s kernel overhead.*

# Synchronization

Microservices such as Cache **oversubscribe threads** to improve service throughput. Hence, such microservices frequently synchronize various thread pools.

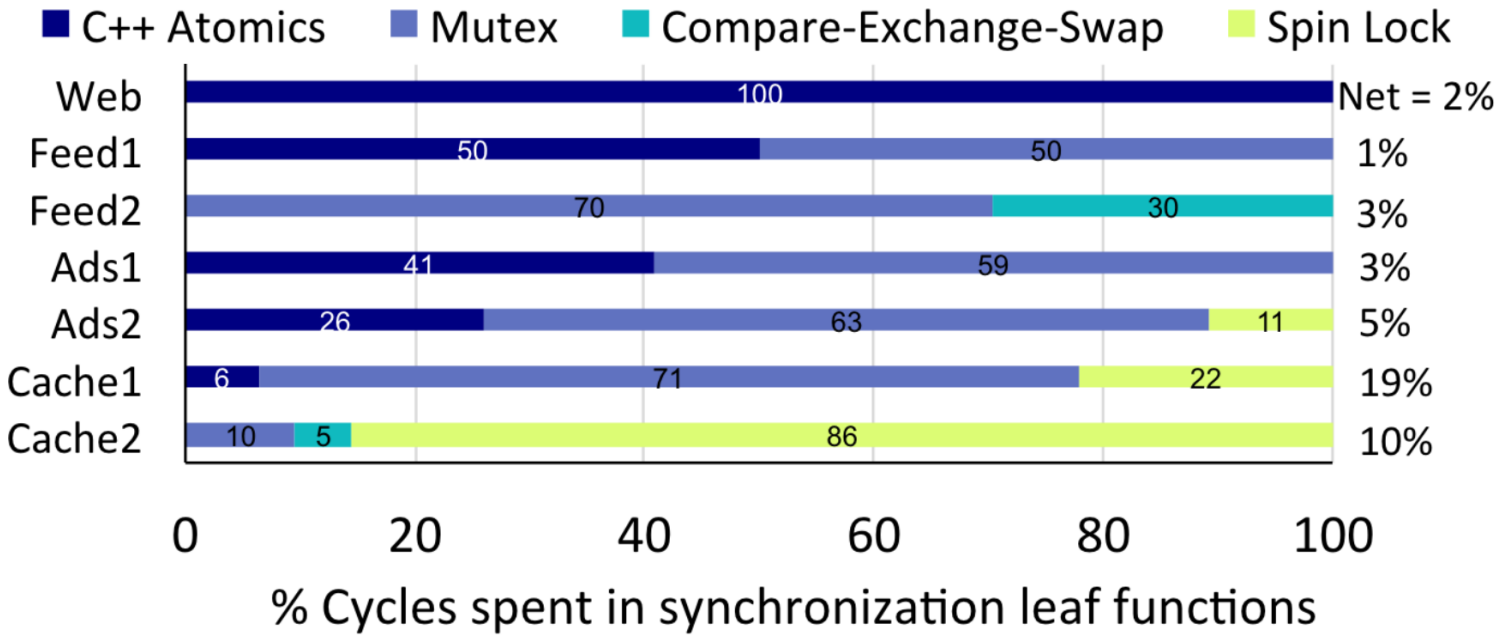


Figure 8: Breakdown of CPU cycles spent in synchronization functions: Cache frequently uses spin locks to avoid thread wakeup delays.

# Synchronization

**Observations.**

- Cache, which exhibits a high synchronization overhead, spends several cycles in **spin locks** that are typically deemed performance ineffcient.
- However, Cache implements spin locks since it is a $\mu$s-scale microservice, and is hence more prone to $\mu$s-scale performance penalties that can otherwise arise from *thread re-scheduling, wakeups, and context switches.*[6]

[6] The primary disadvantage of a spinlock is that, while waiting to acquire a lock, it wastes time that might be productively spent elsewhere. There are two ways to avoid this: (1) Do not acquire the lock. (2) Switch to a different thread while waiting.

# C Libraries

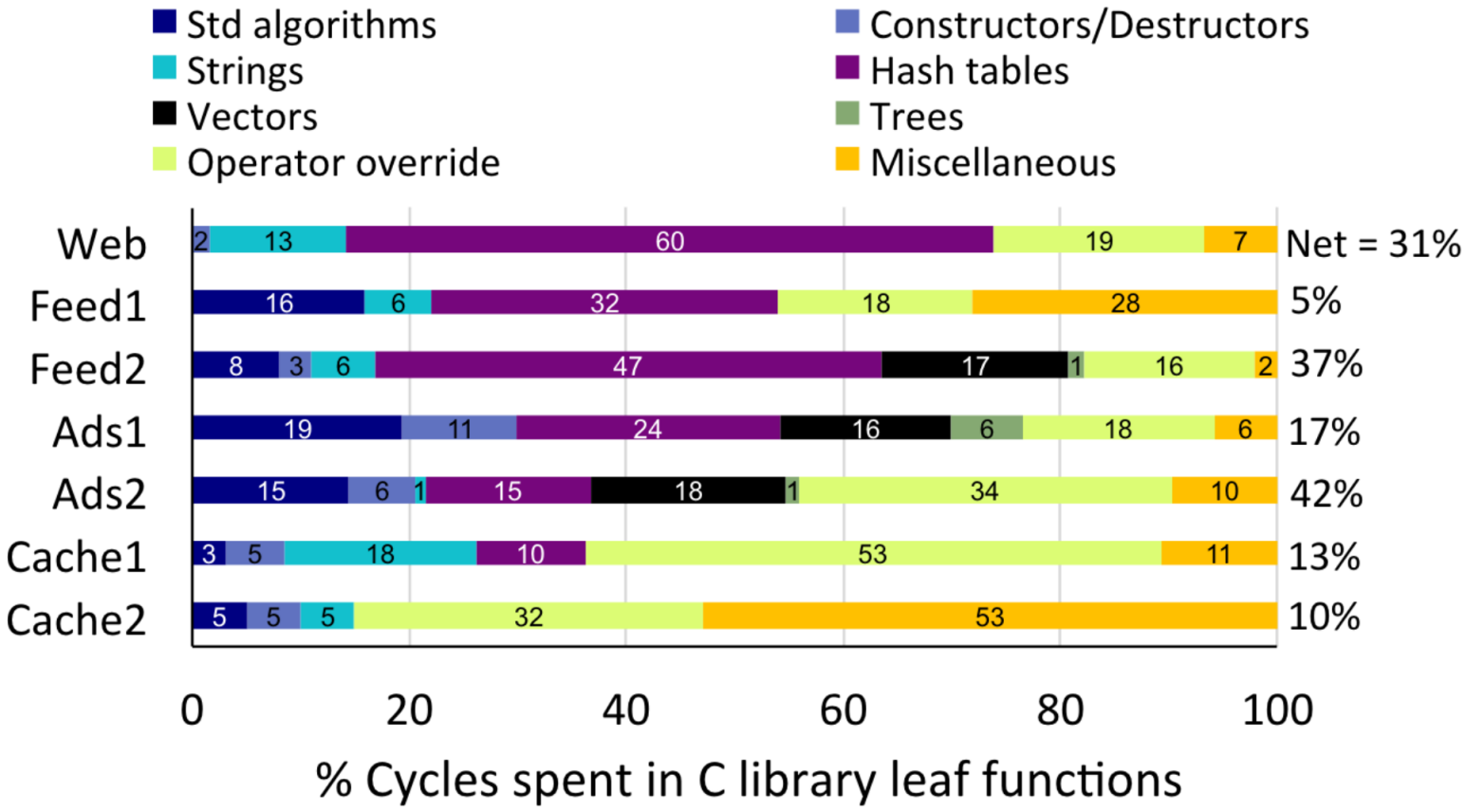


Figure 9: Breakdown of CPU cycles spent in C libraries: ML services perform several vector operations while dealing with large feature vectors.

# C Libraries

**Observations.**

- Feed2, Ad1, and Ad2 perform several vector operations as they deal with large feature vectors.
- Web spends signifcant cycles parsing and transforming strings to process queries from the many URL endpoints it implements.
- Web also performs several hash table look-ups to (1) maintain query parameters, (2) identify services to contact, and (3) merge responses.

Obviously, many microservices can beneft from optimizing **vector operations, string computations, and hash table look-ups**.

# IPC Scaling for Leaf Functions

Cache1's per-core IPC scaling for key leaf functions is depicted in Fig. 10.

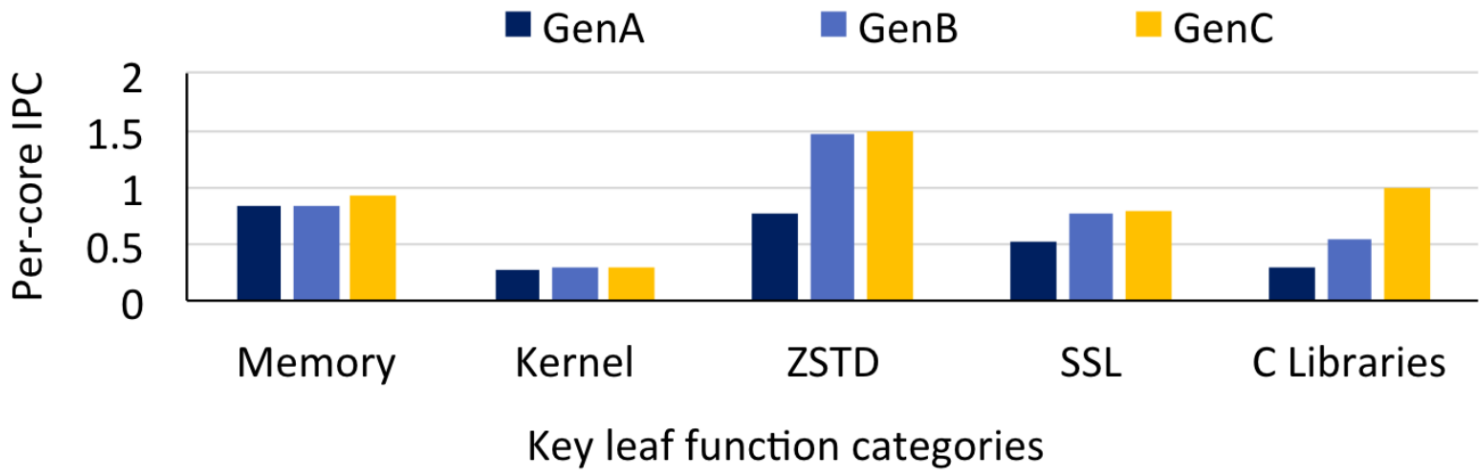


Figure 10: Cache1's IPC scaling across three CPU generations for key leaf funcs.: kernel IPC is typically low & scales poorly.

# IPC Scaling for Leaf Functions

**Observations.**

- Each leaf function type uses less than half of the theoretical execution bandwidth of a GenC CPU (theoretical peak IPC of 4.0). As such, **simultaneous multithreading** is efective for these microservices and is enabled in our CPUs.
- Kernel IPC is typically low and also scales poorly. Accelerating the kernel is non-trivial as it is neither small, nor self-contained, and cannot be easily optimized in hardware. However, software optimizations that minimize scheduler, I/O, and network overheads can improve kernel IPC.
- C libraries' IPC scales well across CPU generations.
- Only a small IPC gain from GenB to GenC.

# Service Functionality Characterization

Below is the categorization of microservice functionalities.

| Functionality category | Examples of service operations |
|---|---|
| Secure and insecure I/O | Encrypted/plain-text I/O sends & receives |
| I/O pre/post processing | Allocations, copies, etc before/after I/O |
| Compression | Compression/decompression logic |
| Serialization | RPC serialization/deserialization |
| Feature extraction | Feature vector creation in ML services |
| Prediction/ranking | ML inference algorithms |
| Application logic | Core business logic (e.g., Cache's key-value serving) |
| Logging | Creating, reading, updating logs |
| Thread pool management | Creating, deleting, synchronizing threads |

**Each functionality category typically includes several leaf function categories.** For example, despite ML inference being heavy on math leaf functions, it can also comprise memory movement and C library leaves.

# Service Functionality Characterization

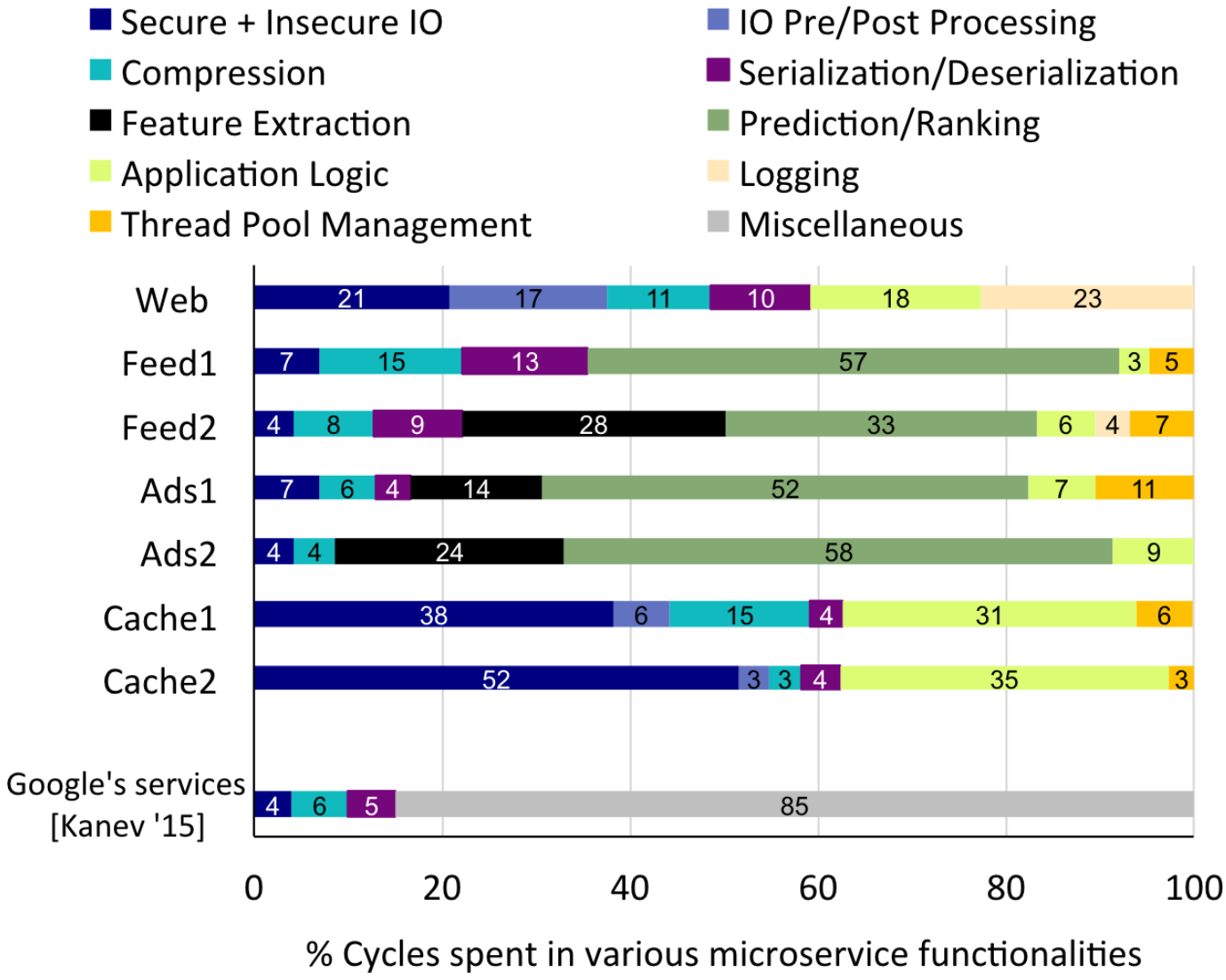


Figure 11: Breakdown of CPU cycles spent in various microservice functionalities: orchestration overheads are signifcant & fairly common.

# Service Functionality Characterization

**Observations.**

- Several microservices face **signifcant orchestration overheads** from performing operations that are **not core to** the application logic, but instead facilitate application logic such as compression, I/O, and logging (e.g., 33% of cycles on ML inference in Feed2).
- Several orchestration overheads are **common** across microservices; accelerating them can signifcantly improve the global feet's performance.
- Web spends only 18% of cycles in core web serving logic (parsing and processing client requests), consuming 23% of cycles in reading and updating logs.
- Ad1, Feed2, Cache1, and Feed1 incur a high thread pool management overhead. Intelligent **thread scheduling and tuning** can help these services.

# IPC Scaling for Functionalities

Cache1's per-core IPC for key microservice functionalities across 3 CPU generations are depicted in Fig. 12.

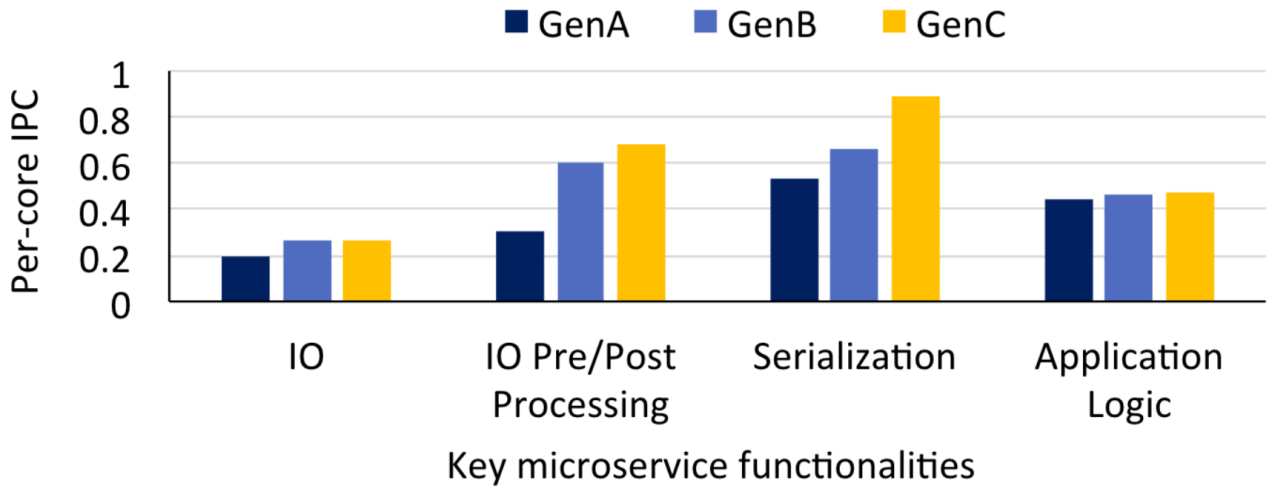


Figure 12: Cache1's IPC scaling across three CPU generations.

- **I/O IPC remains low across CPU generations.** Since I/O calls primarily invoke kernel functions, the low kernel IPC (see Fig. 10) contributes to the low I/O IPC.
- **Application logic IPC also remains low.** Since key-value stores are typically memory intensive, the low memory IPC (Fig. 10) results in a low key-value store IPC.

# Summary of Findings

- **Significant orchestration overheads.** Software and hardware acceleration for orchestration rather than just app. logic.
- **Several common orchestration overheads.** Accelerating common overheads (e.g., compression) can provide fleet-wide wins.
- **Poor IPC scaling for several functions.** Optimizations for specific leaf/service categories.
- **Memory copies & allocations are significant.** Dense copies via SIMD, copying in DRAM, Intel's I/O AT, DMA via accelerators, PIM.
- **Memory frees are computationally expensive.** Faster software libraries, hardware support to remove pages.

# Summary of Findings (Cont'd)

- **High kernel overhead and low IPC.** Coalesce I/O, user-space drivers, in-line accelerators, kernel-bypass.
- **Logging overheads can dominate.** Optimizations to reduce log size or number of updates.
- **High compression overhead.** Bit-Plane Compression, Buddy compression, dedicated compression hardware.
- **Cache synchronizes frequently.** Better thread pool tuning and scheduling, Intel's TSX, coalesce I/O, vDSO.
- **High event notification overhead.** FRDMA-style notification, hardware support for notifications, spin vs. block hybrids.

# Outline

# Accelerometer

Overheads identifed can be accelerated in the hardware via CPU optimizations (e.g., specialized hardware instructions) or custom accelerator devices (e.g., ASICs).

- **On-chip.** On-chip acceleration optimizes components *on the CPU die* (e.g., wider SIMD units, Intel's AES-NI hardware encryption instruction, and CPU modifcations). Offoad latencies are typically ns-scale.
- **Off-chip.** Off-chip accelerators are typically contacted *via PCIe and coherent interconnects* (e.g., GPUs, smart NICs, and ASICs). Offoad latencies are $\sim$ $\mu$s-scale.
- **Remote.** Remote accelerators are *off-platform devices contacted via the network*. Examples include remote ML inference units, network switches, remote encryption units, and remote GPUs. Offload latencies are typically ms-scale when using commodity ethernet.

# System Abstraction

We consider an abstract system with 3 components:

- **host:** a general-purpose CPU
- **accelerator:** a custom hardware to accelerate a kernel
- **interface:** the communication layer between the host and the accelerator (e.g., a PCIe link)

The interface helps define overheads from dispatching work to an accelerator (e.g., preparing the kernel for offload, offload latency, and queuing delays).

# System Model

**Assumptions.**

1. The kernel's execution time is a function of granularity $g$, i.e., the data offload size.
2. The host and accelerator use kernels of the same complexity.
3. Data offload is unpipelined (i.e., the accelerator requires the entire block to start operating); it considers the average latency of such an offload, $L$.

**(Other) Model parameters.**

- $C$: Total cycles spent by the host to execute all logic in a fixed time unit.
- $\alpha$: A constant such that the host spends $\alpha C$ cycles executing the kernel and $(1-\alpha)C$ cycles executing the non-kernel logic.
- $A$: The peak achievable accelerator speedup factor.

# System Model

**Other notations.** $o_0$ denotes cycles the host spends in setting up the kernel prior to a single offload. $Q$ denotes avg. cycles spent in queuing between host and accelerator for a single offload. $o_1$ denotes cycles spent in switching threads (due to context switches and cache pollution) for a single offload.

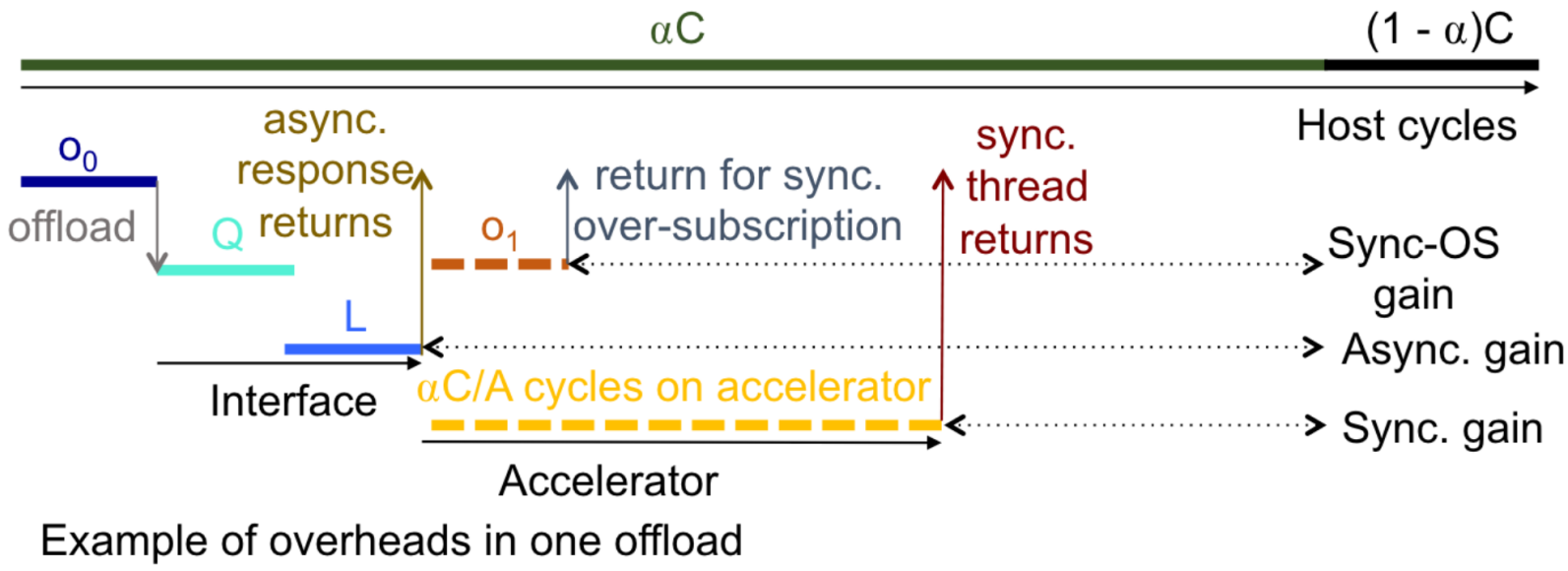


Figure 13: Example timeline of host & accelerator (explain the colors!).

# System Model

Accelerometer models the microservice **throughput speedup** and the microservice **per-request latency speedup**.

- To model **speedup**, Accelerometer identifies how many fewer *host cycles* are needed to execute the kernel when there is acceleration spending fewer host cycles on the kernel frees up host cycles to do more work, improving throughput. $\triangleright\ C/C_S$
- To model **per-request latency reduction**, it identifies the *total cycles taken* to execute a request when there is acceleration; spending fewer cycles for a request due to acceleration reduces per-request latency. $\triangleright\ C/C_L$

# Offloading models

The authors consider 3 offloading models in Accelerometer.

- Synchronous offloading (Sync). When a host thread offloads work to an accelerator synchronously, it waits in the blocked state for the accelerator's response.
- Synchronous offloading with Oversubscription (Sync-OS). Oversubscription allows a host to schedule an available thread to process new work, while the thread that offloaded work blocks awaiting the accelerator's response. The host continues to perform useful work instead of wasting cycles in awaiting the accelerator's response.
- Asynchronous offloading (Async). The host continues to process new work without awaiting the accelerator's response.

# Synchronous Offloading

Consider $n$ synchronous offloads occur (*the microservice runs one thread per core, the host's core waits for the accelerator's response*), then we have:

$$\text{Sync } \frac{C}{C_s} \text{ or } \frac{C}{C_L} = \frac{C}{(1-\alpha)C + \frac{\alpha C}{A} + n(o_0 + L + Q)}. \quad (1)$$

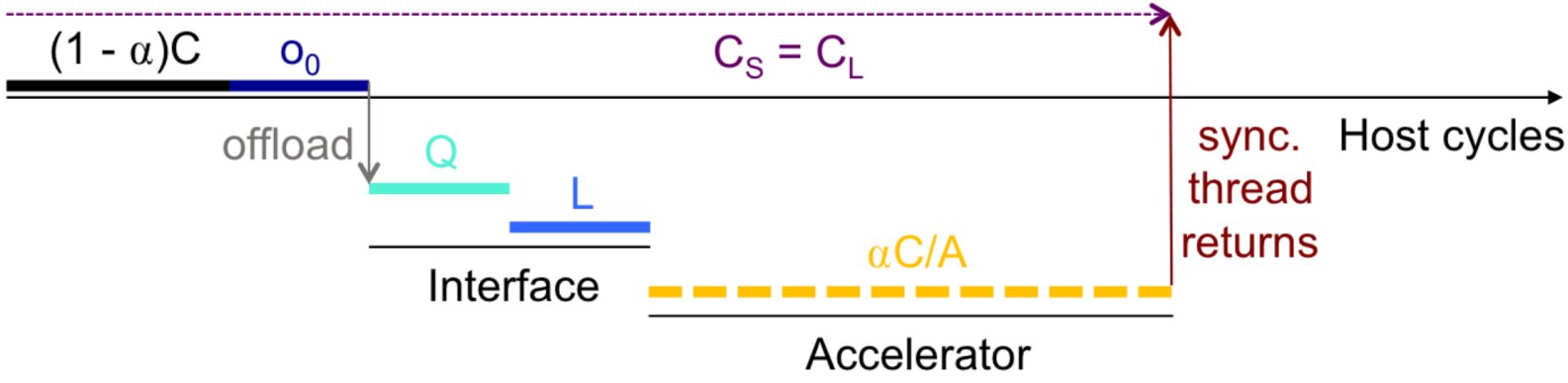


Figure 14: Modeling Sync $C_S$ and $C_L$ for one offload.

# Synchronous Offloading

To determine whether a kernel offload improves speedup, we consider the offload granularity, $g$, such that the host spends $C_b$ cycles per byte of $g$.

A single offload improves speedup when

$$C_b \times g > \frac{C_b \times g}{A} + o_0 + L + Q.^{7} \tag{2}$$

Note that Assumption 2 is applied here.

[7] Use $C_b \times g^{\beta}$ (allow $\beta \neq 1$) to derive a non-linear kernel's complexity.

# Synchronous Offloading with Oversubscription

With Oversubscription, the host continues to perform (other) useful work instead of wasting cycles.

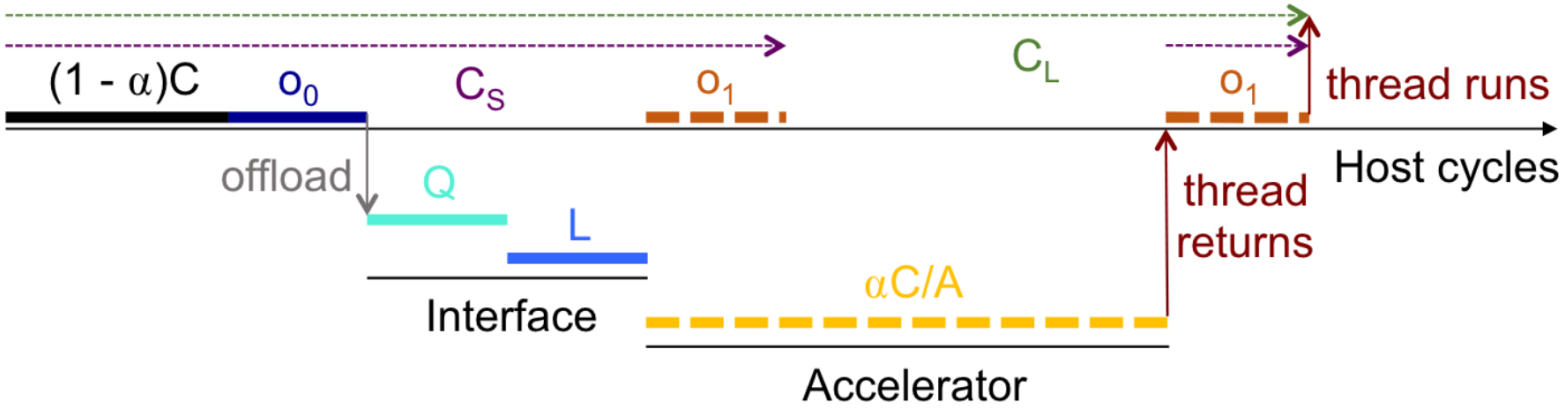


Figure 15: Modeling Sync-OS $C_S$ and $C_L$ for one offload.

The speedup in this case is

$$\text{Sync-OS}\ \frac{C}{C_S} = \frac{1}{(1-\alpha) + \frac{n}{C}(o_0 + L + Q + 2o_1)}, \tag{3}$$

where $o_1$ is the cycles spent in switching threads (due to context switches and cache pollution) for a single offload.

# Synchronous Offloading with Oversubscription

On the other hand,

$$\text{Sync-OS}\ \frac{C}{C_L} = \frac{1}{(1-\alpha) + \frac{\alpha}{A} + \frac{n}{C}(o_0 + L + Q + o_1)}, \tag{4}$$

since $C_L$ includes cycles spent on the accelerator. The $\mu$s-scale $o_1$ overhead can dominate in $\mu$s-scale microservices such as Cache, making it feasible to incur a throughput gain at the cost of a per-request latency slowdown.

Obviously, a single offload improves *throughput speedup* when

$$C_b \times g > o_0 + L + Q + 2o_1. \tag{5}$$

A single offload reduces *latency* when

$$C_b \times g > \frac{C_b \times g}{A} + (o_1 + L + Q + o_1). \tag{6}$$

# Asynchronous Offloading

When a host thread offloads work asynchronously, it continues to process new work *without awaiting the accelerator's response* after issuing the offloading. When the response arrives, it can be picked up by

- the same thread that sent the request,
- a distinct thread dedicated to pick up responses.

**Case I.** When a distinct thread picks up the response, the speedup equation is the same as (3) with only one thread switching overhead $o_1$. The latency reduction equation remains the same as (4).

# Asynchronous Offloading

**Case II.** If the response is picked up by the same thread that sent the request, $o_1 = 0$ since the OS does not switch threads. The speedup is

$$\text{Async}\ \frac{C}{C_S} = \frac{1}{(1-\alpha) + \frac{n}{C}(o_0 + L + Q)}. \tag{7}$$

A single offload improves speedup when

$$C_b \times g > o_0 + L + Q. \tag{8}$$

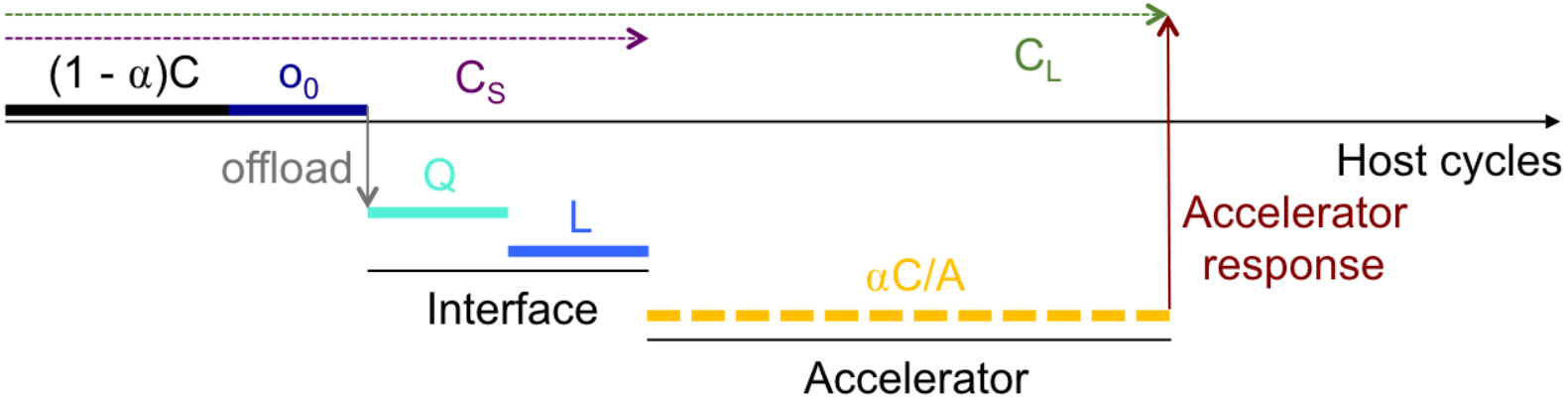


Figure 16: Modeling Async $C_S$ and $C_L$ for one offload.

# Asynchronous Offloading

On the other hand, the latency reduction is

$$\text{Async}\ \frac{C}{C_L} = \frac{1}{(1-\alpha) + \frac{\alpha}{A} + \frac{n}{C}(o_0 + L + Q)}. \tag{9}$$

A single offload reduces latency when

$$C_b \times g > \frac{C_b \times g}{A} + (o_0 + L + Q). \tag{10}$$

# Asynchronous Offloading

In some asynchronous designs, the host does not require the accelerator's response for further processing, eliminating $o_1$ (*e.g., when a host sends requests to an encryption accelerator, which then sends encrypted requests to the next microservice*). Hence, the speedup equation remains the same as (7).

Latency reduction depends on whether acceleration is off-chip or remote since remote accelerator latencies $\frac{\alpha C}{A}$ will not affect a microservice's request latency and will instead show up in the overall application's e2e latency. The authors define the Async off-chip per-request latency reduction as (9) and the remote latency reduction as (7).

# Outline

# Validation Methodology

A 5 step process for Accelerometer's validation:

1. Identify offload sizes $g$ that improve speedup.
2. Determine the number of such offloads in one second, $n$, and the fraction of cycles they constitute, $\alpha$.
3. Use Accelerometer to estimate speedup from these $n$ offloads.
4. Compare Accelerometer estimated speedup with real production speedup.
5. Present a functionality breakdown for both the accelerated and unaccelerated microservices to show how throughput improves.

# Measure Real Production Speedup

For each case study, the authors first measure *the real production speedup* using an internal tool via A/B testing (by comparing the throughput of two identical servers that differ only in terms of whether they accelerate the kernel).

# Measure Accelerometer's Speedup

The authors measure model parameters using (1) *tools such as Strobelight, bpftrace, and bcc-tools*, (2) *roofline estimates from device specification sheets*, and *micro-benchmarks that measure execution time on the host and the accelerator.*

- Measure the unaccelerated host's busy frequency to calculate $C$ for one second
- Use bpftrace to measure $g$'s size range and the number of invocations of each granularity
- Compute $n$ by aggregating invocations of those offload sizes that improve speedup
- To determine $\alpha$, the authors first use the service functionality breakdown (see Fig. 11) to estimate host cycles spent in the kernel under study. They then use $n$ and these total host cycles to estimate the fraction of kernel cycles that must be offloaded, $(\alpha \cdot C)$

# Case Study: AES-NI for Cache1

The authors study encryption in Cache1 with Intel's AES-NI instruction (an on-chip optimization).

Cache1 uses a Sync threading design. The authors use AES from the OpenSSL cryptography library to build micro-benchmarks to measure $L$, $o_0$, and $A$. Since the same host thread executes the AES-NI instruction, $Q = 0$. Then,

$$\frac{C}{(1-\alpha)C + \frac{\alpha C}{A} + n(o_0 + L + Q)} \approx 15.7\% \tag{11}$$

with $C = 2.0 \times 10^9$ cycles, $\alpha = 0.165844$, $n = 298951$, $L = 3$, $o_0 = 10$, and $A = 6$.

Correspondingly, the real production speedup is $14\%$.

# Outline

# Model Application

The authors apply the Accelerometer model to project speedup for the acceleration recommendations derived from three key common overheads identified by the characterization: **compression**, **memory copy**, and **memory allocation**.

For compression, the authors apply existing on-chip and off-chip compression acceleration with Sync, Sync-OS, and Async.

| Overhead | Acceleration | C ($10^9$ cycles) | $\alpha$ | n | L (cycles) | $o_1$ (cycles) | A |
|---|---|---|---|---|---|---|---|
| Compression | On-chip: Sync | 2.3 | 0.15 | 15,008 | 0 | NA | 5 |
| Compression | Off-chip: Sync | 2.3 | 0.15 | 9,629 | 2,300 | NA | 27 |
| Compression | Off-chip: Sync-OS | 2.3 | 0.15 | 3,986 | 2,300 | 5,750 | 27 |
| Compression | Off-chip: Async | 2.3 | 0.15 | 9,769 | 2,300 | NA | 27 |
| Memory Copy | On-chip: Sync | 2.3 | 0.1512 | 1,473,681 | 0 | NA | 4 |
| Memory Allocation | On-chip: Sync | 2.0 | 0.055 | 51,695 | 0 | NA | 1.5 |

Figure 17: Parameters used to model speedup and latency reduction. Here $\alpha = 0.15$ because Feed1 spends 15% of cycles in compression.

# Apply to Compression

Ideally, for on-chip offloads, the overhead $o_0 + L$ can be negligible. We also assume $Q = 0$. Then, for compression, Feed1 can achieve an ideal speedup of $17.6\%$.

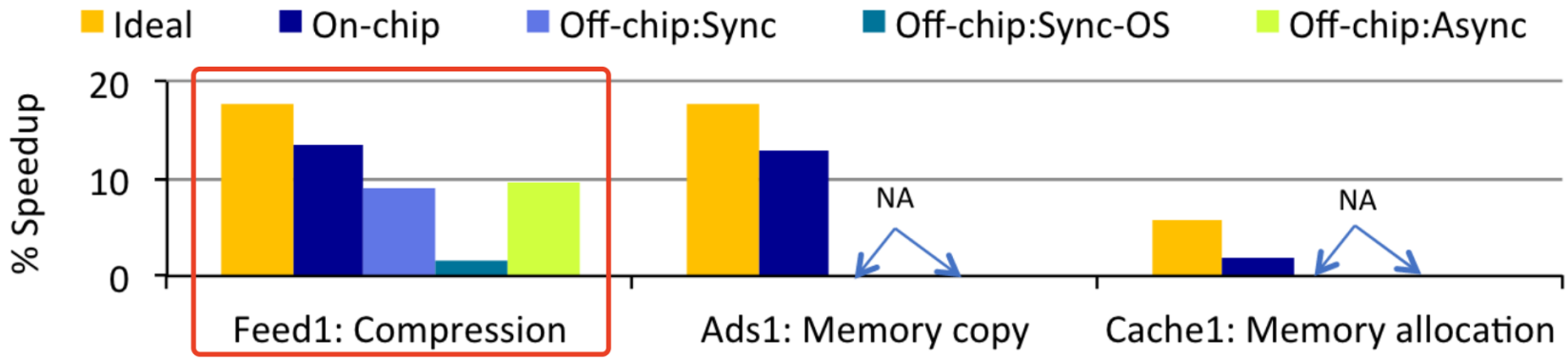


Figure 18: Accelerometer-estimated speedup.

**Recommendation.** Even though on-chip yields a higher speedup, there might be value in off-chip acceleration as it is easier to design than modifying CPUs.