Scheduling Linux Threads under I/O Chiplet Wall Using `cSwitch`

Seunghyun An
University of Wisconsin–Madison
Madison, Wisconsin, USA

Joontaek Oh
University of Wisconsin–Madison
Madison, Wisconsin, USA

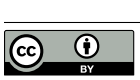
Ming Liu
University of Wisconsin–Madison
Madison, Wisconsin, USA

Abstract

Chiplet-based processors have become the dominant architecture for modern servers, partitioning functionality across compute chiplets and a centralized I/O chiplet. While this design improves scalability, it introduces a new bottleneck—the I/O Chiplet Wall—where off-chip memory and device requests contend along a shared communication path through the I/O chiplet and interconnect fabric. Our characterization of an AMD EPYC processor shows that this bottleneck significantly impacts performance: it introduces non-trivial latency overheads, caps memory bandwidth, propagates congestion back to cores, reduces per-core effective bandwidth, and lacks traffic control. However, existing OS schedulers remain unaware of this bottleneck, leading to pathological scheduling behaviors and substantial inefficiencies.

This paper presents **`cSwitch`**, a Linux scheduling framework that addresses the I/O Chiplet Wall. Our key insight is to model the I/O chiplet as a software switch that orchestrates thread-induced data movement. `cSwitch` represents threads as data flows (cFlows) annotated with communication features, transforming CPU scheduling into a flow scheduling problem and employing networking-inspired techniques. To realize this, we build an I/O Chiplet Loading Map to infer the runtime traffic load, define Chiplet Traffic Label to capture per-flow behavior, and design a cFlow scheduler that combines rate control, contention-aware refinement, and bandwidth-aware migration. Further, `cSwitch` supports coordinated workloads via MegaCFlow and provides a user-defined policy interface using `sched_ext` and eBPF. We build `cSwitch` and evaluate it against EEVDF, ARCAS, and Caladan+ across diverse workloads. Our experimental results show that `cSwitch` improves performance by up to 2.7×, reduces latency and increases throughput under contention, and achieves better scalability with modest overheads.

CCS Concepts: • **Computer systems organization → Interconnection architectures**; • **Hardware → Network on chip**; • **Software and its engineering → Scheduling**.



SOSP '26, Prague, Czech Republic

ACM ISBN 979-8-4007-2585-2/26/09
https://doi.org/10.1145/3830418.3843889

ACM Reference Format:
Seunghyun An, Joontaek Oh, and Ming Liu. 2026. Scheduling Linux Threads under I/O Chiplet Wall Using `cSwitch`. In *Symposium on Operating Systems Principles (SOSP '26), September 29–October 2, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3830418.3843889

1 Introduction

Chiplets have become the predominant approach for building modern server processors [32, 39, 87–90, 105–107]. Instead of integrating all components on a single die, chiplet processors partition functionality across compute chiplets and a centralized I/O chiplet, connected via on-package interconnects [40, 93]. Compute chiplets host CPU cores and caches, optimized for compute density, while the I/O chiplet integrates memory controllers, coherence fabrics, and device interfaces. This modular design improves yield, enables heterogeneous process technologies, and scales cores efficiently.

Chiplet processors introduce a shared communication substrate within the SoC. Off-chip requests must traverse a sequence of shared resources: the I/O chiplet, the network-on-interposer (NoI) that connects chiplets [30, 110, 120], and cross-chiplet interfaces, each provisioned with finite bandwidth and limited buffering. Thus, core-memory, core-device, and inter-chiplet coherence traffic are no longer isolated, but instead contend along a common data path. We term this emerging bottleneck the **I/O Chiplet Wall**: a fundamental performance limit imposed by the shared inter-chiplet communication path that mediates access between compute chiplets and external resources. As systems scale to higher core and chiplet counts, this wall manifests as higher memory access latency, capped effective bandwidth, and amplified interference, ultimately constraining end-to-end performance.

We characterize the I/O chiplet wall from several perspectives using a multi-chiplet AMD EPYC 9634 processor (§2.2). First, traversing the I/O chiplet introduces a non-negligible latency penalty, especially under contention, resulting in up to 2.0× higher stall cycles compared to a non-chiplet processor. Second, the I/O chiplet caps the memory throughput, constrained by the inter-chiplet links, the NoI, the I/O chiplet's internal crossbar, and the chiplet-memory interface; on our platform, bandwidth saturates at 330 GB/s, beyond which requests queue and no additional throughput is achieved. Third, congestion backpropagates through the system due to credit-based flow control in both the intra-chiplet NoC and inter-chiplet NoI, throttling request injection and stalling

core pipelines. Fourth, per-core bandwidth decreases as intra-chiplet concurrency increases, since compute-I/O chiplet links are statically provisioned; we observe 1.7–2.4× degradation for DRAM and 1.6–2.5× for CXL memory. Finally, the I/O chiplet provides no explicit traffic control, relying on best-effort arbitration, which leads to severe performance interference, up to 7.1× slowdown under congestion.

However, today's Linux OS schedulers largely ignore the I/O chiplet, treating compute chiplets as sub-NUMA domains and relying on EEVDF [6] with coarse locality heuristics [4, 14, 18]. This abstraction is far from sufficient: EEVDF lacks visibility into inter-chiplet communication and treats cores as independent resources, ignoring the I/O chiplet as a shared bottleneck. We identify several pathological cases (§3). (1) *Not all cores are equal*: performance depends on I/O path conditions, leading to 35.1–58.0% degradation when scheduling ignores link contention. (2) *Topology-aware affinity breaks*: under high load, "close" placement causes 4.3–4.5× slowdown, while remote placement performs better. (3) *Congestion expands system-wide*, degrading unrelated threads by up to 1.7×, beyond scheduler visibility. (4) *Load balancing worsens imbalance*: CPU-centric policies amplify memory contention, while end-to-end balancing improves performance by 13.6–20%+. These results reveal a mismatch: hardware topology is no longer a reliable proxy for performance, which is instead dictated by dynamic I/O chiplet contention, motivating the need for traffic-aware scheduling.

In this paper, we design and implement **`cSwitch`**, a Linux scheduling framework that tackles the I/O chiplet wall. *Our key idea is to model the I/O chiplet as a software switch that orchestrates thread-induced data movement.* We decompose the I/O chiplet into ingress channels (from compute chiplets), egress channels (to memory and devices), and a traffic manager that regulates how data traverses the system. Each thread is represented as one or more cFlows (§4.3), annotated with a chiplet traffic label that captures its communication path and demand. Rather than relying on explicit buffering, `cSwitch` adopts a bufferless design with phantom queues [21, 23, 111], inferring congestion from delay signals such as stall cycles and service latency. This abstraction transforms CPU scheduling into a flow scheduling problem[22, 51, 112], enabling `cSwitch` to leverage networking-inspired techniques to control chiplet traffic and mitigate contention.

Realizing this vision is non-trivial and imposes several challenges. First, to estimate the I/O chiplet traffic conditions with limited hardware visibility, `cSwitch` introduces the I/O Chiplet Loading Map (§4.2), a runtime telemetry structure that samples architectural counters and reconstructs cFlow–channel mappings via temporal differencing and a bipartite matching scheme. Second, to represent thread communication, `cSwitch` proposes the Chiplet Traffic Label (§4.3), a per-cFlow signature (inspired by the HASS [104] scheduler) enclosing an end-to-end communication path, performance attributes, and counter-inferred contention. Third, `cSwitch` develops a cFlow scheduler (§4.4) that jointly determines thread placement and time slice, combining egress-driven rate control, ingress-based refinement, and greedy bandwidth-aware migration. Fourth, to support coordinated workloads, `cSwitch` provides MegaCFlow (§4.5), a coflow-inspired [36–38] abstraction that groups threads and integrates seamlessly with the cFlow scheduler. Last, `cSwitch` exposes a user-defined scheduling policy interface (§4.6) built on Linux `sched_ext` [9] and eBPF.

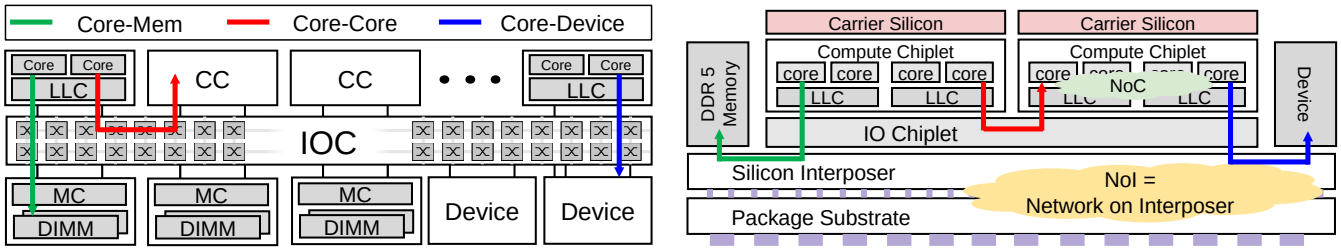
We prototyped `cSwitch` on AMD chiplet-based processors and compared it against Linux EEVDF [6] and two recent schedulers, i.e., ARCAS [48] and Caladan+ [50], over a wide range of workloads. Our evaluations show that `cSwitch` outperforms prior approaches by 1.2–2.7×, by placing threads on compute-I/O and I/O-memory/device paths with sufficient bandwidth while avoiding unnecessary I/O chiplet contention. By reacting quickly to congestion and enforcing fair bandwidth allocation, `cSwitch` achieves up to 84.5% latency reduction and 3.9× throughput improvement. Furthermore, `cSwitch` demonstrates better scalability both within and across chiplets, sustaining high utilization under increasing demand. `cSwitch` incurs modest overheads in memory footprint and CPU cycles. `cSwitch` is available at https://github.com/netlab-wisconsin/cSwitch.

2 Background

2.1 Chiplet Processors

Modern server processors [32, 39, 87–90, 105–107] increasingly adopt chiplet-based architectures to address the growing difficulty of scaling monolithic dies. Instead of integrating all processor components onto a single large die, chiplet processors partition functionality across multiple smaller dies (i.e., chiplets), interconnected through high-bandwidth on-package die-to-die fabrics (like UCIe [40] and OpenHBI [93]) operating over a compatible 2.5D/3D silicon interposer [52, 117, 118]. This modular design [59, 65, 109, 113] improves manufacturing yield, allows heterogeneous process technologies, and enables vendors to scale core counts efficiently.

A chiplet processor generally encompasses compute chiplets, connected via a centralized I/O chiplet (Figure 1-a). Compute chiplets contain CPU cores, private caches, and portions of the shared last-level cache. They are typically optimized for compute density and fabricated using advanced process nodes (like 5nm in our evaluated EPYC 9634 CPU). In contrast, the I/O chiplet integrates system interfaces such as memory controllers, coherent routing interconnects, PCIe substrates, and I/O hubs. Because I/O logic benefits less from aggressive scaling, it is often implemented in more mature fabrication processes (e.g., 6nm in EPYC 9634). In such a separated architectural organization, all off-chip communication (Figure 1-a), including DRAM accesses and device interactions, must traverse on-package interconnects (the ones connecting chiplet-chiplet and chiplet-memory/device)



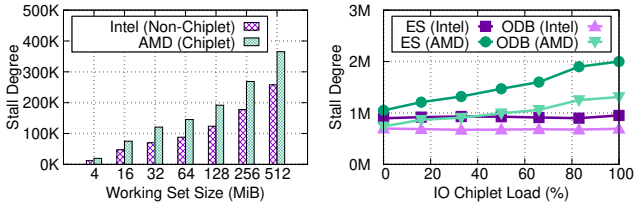
(a) Architectural Diagram of a Chiplet Processor

(b) Vertical View of a Chiplet Design

Figure 1. (a) depicts the architectural diagram of a chiplet processor. (b) shows the package substrate of a chiplet-based design.

Parameters	EPYC 9634
microarchitecture	Zen 4
L1 (per core)	64 KB
L2 (per core)	1 MB
L3 (per CPU)	384 MB
Cores per Compute Chiplet	7
Compute Chiplet (CC) #	12
Process Node (CC)	5 nm
I/O Chiplet (IOC) #	1
Process Node (IOC)	6 nm
Memory Controller #	12
PCIe Gen.	Gen 5 (CXL 1.1/2.0)
PCIe Lane	128
Base/Max Freq.	2.25 / 3.7 GHz

Table 1. Hardware specification of our evaluated processor.



(a) Stall varying working set size **(b)** Stall varying I/O chiplet load

Figure 2. We compare the application performance on a chiplet processor (AMD EPYC 9634) vs. a non-chiplet one (Intel CPU). (a) and (b) report the stall degree (defined as the number of processor stalled cycles per application request) while varying the working set size and I/O chiplet load on a single core, respectively. We run RocksDB in (a) and OrientDB and Elasticsearch in (b).

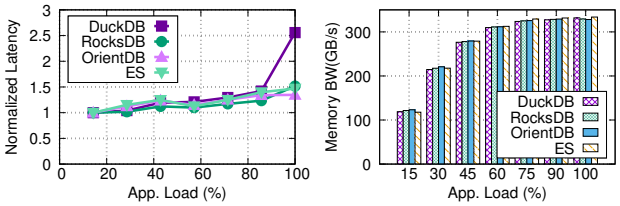
and the I/O chiplet. As a result, the I/O chiplet emerges as a gateway between compute chiplets and external resources.

2.2 I/O Chiplet Wall

Chiplet processors introduce a new shared communication structure inside the package. Cores interacting with off-chip memory and devices (Figure 1-b) must traverse (1) an I/O chiplet, (2) the network-on-interposer (NoI) that connects chiplets across the package substrate [30, 110, 120], and (3) the cross-interposer chiplet interface [60, 79]. While this design improves modularity and scalability, all core-memory, core-device traffic, and in some cases coherence traffic between chiplets must pass through the same inter-chiplet infrastructure before reaching their destinations. Because the I/O chiplet, NoI, and interfaces have finite bandwidth and internal queues, they inherently become points of contention when many cores generate traffic simultaneously.

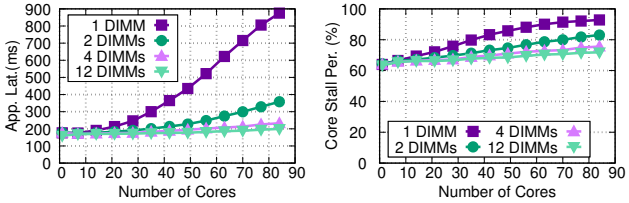
We refer to this emerging bottleneck as the **I/O Chiplet Wall**: a performance limit caused by the shared inter-chiplet communication path that mediates access between compute chiplets and external resources. As core counts and chiplet counts continue to scale, this wall increasingly constrains memory access latency, aggregate bandwidth, and overall system performance. To distinguish this effect from memory-controller saturation, we compared victim latency under noise traffic that shared the compute-to-I/O (CC-I/O) chiplet link, both w/ and w/o sharing the downstream I/O-DIMM path. Although the noise traffic remained below 39.0% of the measured saturation knee, sharing the CC-I/O path degraded performance by 9.5% w/o a shared I/O-DIMM path, compared with 13.3% when the I/O-DIMM path was also shared. These slowdowns therefore cannot be explained solely by memory-controller saturation. We use an AMD EPYC 9634 multi-chiplet CPU (Table 1) and characterize different aspects of the problem. §5.1 describes our experimental setup.

Issue #1: Traversing the I/O chiplet incurs a latency penalty. Chiplet processors physically separate compute cores from memory controllers, forcing memory accesses to cross multiple architectural components before reaching DRAM. Compared with monolithic processors, a request must undergo additional processing stages, including serialization at the inter-chiplet interface, arbitration within the I/O chiplet routing logic, and implicit queueing when requests accumulate at the chiplet boundary. As a result, crossing the I/O chiplet imposes a persistent latency tax on memory accesses that becomes more pronounced under contention. Figure 2-a quantifies this effect by comparing a chiplet-based processor (AMD) with a non-chiplet counterpart (Intel) while varying the application working set size (RocksDB). We observe 1.58× higher stalled cycles on average for a 64B random read workload. The gap widens as I/O chiplet load increases (Figure 2-b): when additional memory-intensive threads are introduced to generate more traffic, stalled cycles on the chiplet processor rise by up to 2.0×, while the non-chiplet processor increases only marginally. Increasing memory-level parallelism helps little because modern out-of-order cores already issue sufficient outstanding requests. More parallelism, accumulating in-flight requests across chiplets, cannot effectively hide the latency.



(a) Application latency **(b)** Memory bandwidth

Figure 3. (a)/(b) report the latency and memory bandwidth varying the application load. We increase the load by using more chiplets to run the workload. This experiment uses RocksDB, OrientDB, and Elasticsearch (ES) under random access, and DuckDB (with TPC-H query 21). The latency in (a) is normalized to the 15% load case.

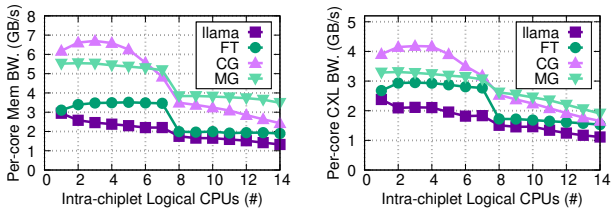


(a) Application latency **(b)** Core stall percentage

Figure 4. (a)/(b) show the application latency and core stall percentage of DuckDB w/ TPC-H varying the number of cores when using 1 DIMM, 2 DIMMs, 4 DIMMs, and all 12 DIMMs.

Issue #2: The I/O chiplet imposes a hard ceiling on memory throughput. In chiplet processors, the I/O chiplet becomes the convergence point for memory and device traffic across the die. The throughput ceiling arises because the communication entities along the path, including the inter-chiplet link, NoI, I/O chiplet's internal crossbar, and the chiplet-memory link (Global Memory Interconnect [1] in AMD), have finite resources and collectively constrain the maximum service rate. Once the application injection traffic rate exceeds the capacity, requests are queued, and no more effective bandwidth is achieved. Figure 3 demonstrates this effect by scaling the number of active cores across 12 chiplets for four data-intensive workloads: aggregate memory bandwidth initially increases but eventually plateaus at around 330 GB/s, while application latency continues to rise by up to 2.4×, 1.3×, 1.3×, and 1.5×, respectively. Therefore, the I/O chiplet fundamentally limits achievable memory throughput, and once saturated, additional concurrent requests only increase contention rather than improving performance.

Issue #3: The I/O chiplet backpropagates congestion and stalls the core pipeline. Beyond throughput limitation, the I/O chiplet propagates memory-side congestion back to compute chiplets. This behavior arises from the credit-based flow control mechanisms [29, 35, 66–68] employed by both the NoC and NoI. A sender can inject new requests only if it has sufficient credits from downstream buffers. When memory controllers or the I/O chiplet become congested, they reduce the rate at which credits are returned to the NoI.



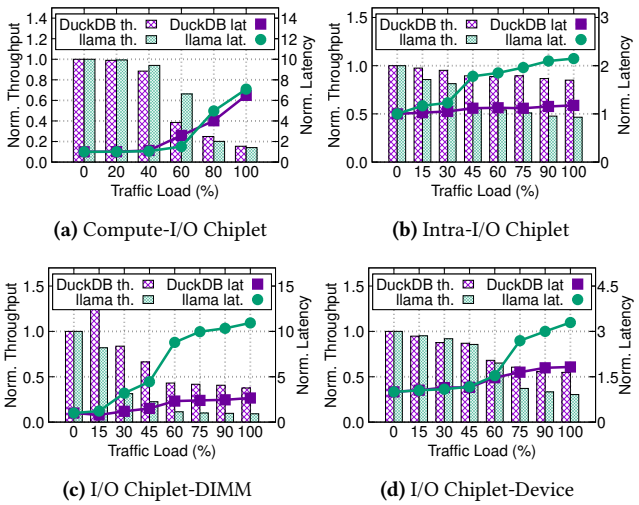
(a) Per-core memory bandwidth **(b)** Per-core device bandwidth

Figure 5. (a)/(b) present the per-core achieved memory and device (`CXL.mem`) bandwidth when scaling the number of cores within a chiplet. We run llama (llama.cpp), CG (Conjugate Gradient), MG (Multi-Grid), and FT (3D Fast Fourier Transform) applications.

This backpressure propagates upstream: the NoI throttles request injection from the compute chiplets, in turn restricting the NoC's ability to forward memory requests from cores. Because the buffering capacity in the NoI and at the chiplet interfaces is limited, it cannot absorb sustained high request rates, pushing stalls back to the cores quickly and hurting performance. We show this by using DuckDB executing TPC-H queries while scaling the number of cores. As shown in Figure 4, compared with a single DIMM case, when enabling 12 DIMMs in an interleaved mode, distributing memory requests across channels alleviates contention and improves credit return rates. This yields a 77.3% application latency reduction and a 29.2% core stall percentage drop.

Issue #4: Per-core bandwidth collapses with increasing intra-chiplet concurrency. As more threads simultaneously issue memory requests, the bandwidth available to each core drops dramatically due to the static provisioning of the compute–I/O chiplet link. This is due to physical constraints, including wire count, signaling rate, and power-delivery limits. Unlike on-chip resources that can be dynamically shared or time-multiplexed at fine granularity, these inter-chiplet links operate at a fixed data rate and lack mechanisms for dynamic scaling at runtime. Thus, intra-chiplet cores compete for a fixed bandwidth budget, yielding a proportional per-core bandwidth reduction, even when each thread's intrinsic demand remains unchanged. Figure 5 shows this by gradually enabling all cores within a compute chiplet for four workloads. We observe up to 2.1×, 1.7×, 2.4×, and 1.7× bandwidth degradation for LLaMA, FT, CG, and MG when all 14 SMT cores are used. A similar trend holds when accessing an external Type-3 CXL memory expander, where the per-core bandwidth drops by 2.2×, 1.6×, 2.5×, and 1.6×. These results highlight that scaling cores does not translate into proportional memory performance in chiplet systems; instead, the compute–I/O chiplet boundary becomes a critical bottleneck that must be explicitly considered.

Issue #5: The I/O chiplet imposes no traffic control. The I/O chiplet lacks explicit mechanisms to regulate or isolate traffic at its ingress, internal routing, and egress pipelines, leading to best-effort bandwidth allocation among competing



(a) Compute-I/O Chiplet **(b)** Intra-I/O Chiplet

(c) I/O Chiplet-DIMM **(d)** I/O Chiplet-Device

Figure 6. We co-locate a victim application (DuckDB/llama) with synthetic workloads that issue requests in a best-effort manner. We gradually increase the background traffic load and create contention at the peripheral I/O chiplet links (i.e., compute-I/O chiplet, I/O chiplet-memory, and I/O chiplet-device) as well as I/O chiplet internals. (a)/(b)/(c)/(d) report the normalized latency and throughput to the standalone case (easy visualization) under interference.

threads. Unlike network switches or QoS-aware interconnects, the I/O chiplet and memory controllers are primarily designed to maximize throughput rather than fairness or isolation. Traffic arbitration decisions are driven by low-level hardware dynamics such as request arrival timing, the aggressiveness of request injection (i.e., the number of outstanding requests per core), queue ordering, and memory controller scheduling policies (e.g., FR-FCFS [98, 99]). These policies inherently favor threads that can generate more concurrent requests or exhibit favorable access patterns (e.g., row-buffer locality), allowing aggressive senders to capture a disproportionate share of bandwidth. Further, the absence of admission control or rate limiting at the compute–I/O chiplet interface, combined with limited buffering and shared queues inside the I/O chiplet, exacerbates interference across threads. Without global coordination or enforcement of fairness across chiplets, contention at compute–I/O links, intra-I/O chiplet routing, I/O-memory links, or I/O-device links can lead to highly unpredictable performance. As shown in Figure 6, we observe up to 6.5×/7.1×, 1.2×/1.7×, 2.7×/10.9×, and 1.6×/3.0× performance degradation when co-locating DuckDB/llama with interfering background workloads in these four cases. This lack of traffic control is particularly problematic in multi-tenant environments, making performance reasoning difficult and hurting system efficiency.

3 Why I/O Chiplet Matters for Scheduling

3.1 Linux CPU Scheduling: EEVDF

Modern Linux systems employ the Earliest Eligible Virtual Deadline First (EEVDF) scheduler [6] as the default policy for fair CPU time allocation, evolving from the long-standing Completely Fair Scheduler (CFS) [3]. EEVDF retains CFS's core principle of proportional fairness but refines task selection using a virtual deadline abstraction [108]. Each task is assigned a virtual runtime that reflects the amount of CPU service it has received, weighted by priority, and a corresponding virtual deadline that determines when it should next be scheduled. At runtime, the scheduler selects the task with the earliest eligible virtual deadline, ensuring both fairness and low latency. To scale across many cores, EEVDF maintains per-CPU run queues, where tasks are enqueued locally and scheduled independently. Periodic load balancing redistributes tasks across CPUs to equalize utilization, while placement decisions are guided by heuristics such as CPU idleness [4], cache locality [14], and NUMA affinity [18].

EEVDF imposes systematic inefficiencies on chiplet-based processors due to its lack of visibility into inter-chiplet communication and the I/O chiplet. First, the scheduler remains largely unaware of chiplet topology: while Linux provides NUMA abstractions and sub-NUMA clustering (SNC) [2], these are coarse-grained and primarily capture memory locality rather than the communication costs within a socket, failing to model fine-grained latency and bandwidth costs of crossing compute chiplets or contending at the centralized I/O chiplet. Second, it relies on naive heuristics for finding out thread relationships, such as the waker-wakee mechanism, which depends on OS-visible synchronization and cannot account for asynchronous or memory-intensive behaviors, leading to suboptimal placements. Third, EEVDF treats cores as independent compute resources and is oblivious to the I/O chiplet as a shared bottleneck, and cannot address the issues characterized above (§2.2). Combined with the lack of traffic control within the I/O chiplet, this can result in severe contention and performance interference across threads. Therefore, existing Linux schedulers lack the necessary visibility and control to reason about chiplet-level costs. We characterize these cases below.

3.2 Case #1: Not All Free/Busy Cores Are Equal

The existing scheduler assumes that free or lightly loaded CPUs are interchangeable resources and prioritizes placing tasks on idle or less busy cores to maximize utilization. However, in chiplet-based architectures, this assumption does not hold. The effective memory bandwidth available to each core is non-uniform and dynamically varying. A core's performance depends not only on its local compute capacity but also on its access path to memory, including the available bandwidth on the compute-I/O chiplet links, contention within the I/O chiplet routing fabric, and utilization of downstream memory channels. As shown in Figure 7-a, both free and busy cores can exhibit different performance characteristics depending on the current I/O chiplet load. EEVDF, combined with sub-NUMA clustering, lacks visibility into these dynamics and may schedule tasks onto cores

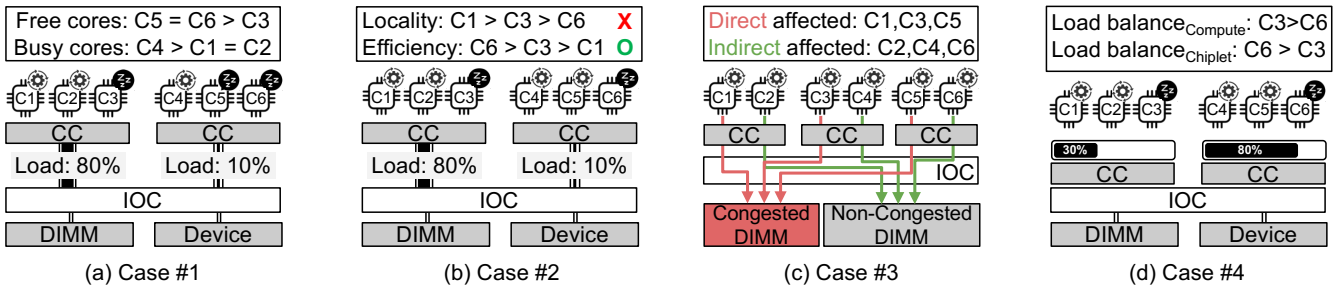


Figure 7. The four pathological scheduling cases of EEVDF+SNC on Chiplet processors. CC=Compute Chiplet. IOC=I/O Chiplet.

that appear available despite being bandwidth-starved. This mismatch leads to suboptimal decisions, where the real bottleneck (data movement capacity through the I/O chiplet) is ignored. We configure the compute-I/O chiplet link with a 25%, 50%, 75%, and 100% load, and compare how the scheduler performs when running PageRank workload. As shown in Figure 8-a, an oracle scheduler aware of compute-I/O link conditions achieves the best performance, while EEVDF and chiplet-affinity policies degrade performance by 35.1% and 24.4% even when there are free cores across chiplets. Under busy-core scenarios (Figure 8-b), we observe a 53.2% and 58.0% performance drop. This is because ignoring I/O chiplet heterogeneity yields inferior scheduling decisions.

3.3 Case #2: Topology-aware Affinity Breaks

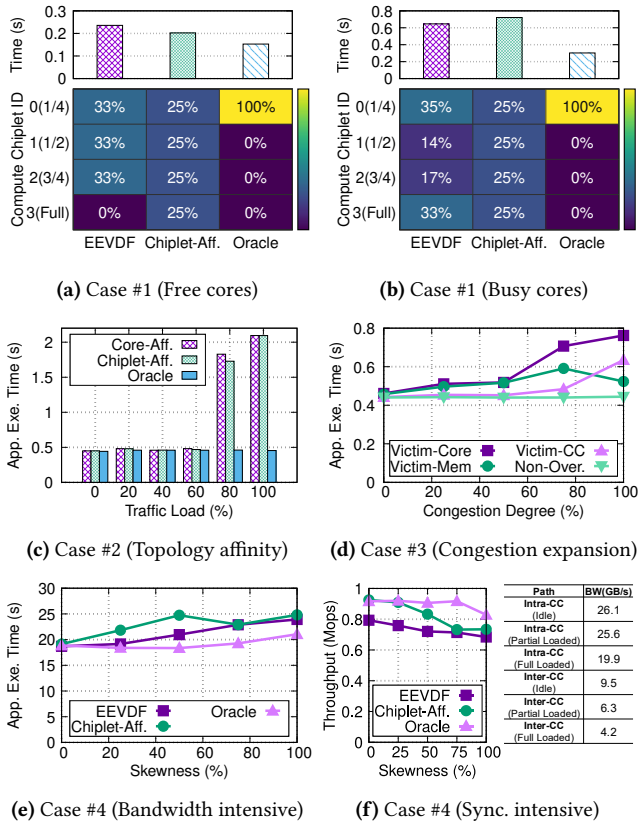
Modern schedulers rely on locality heuristics, such as cache affinity and NUMA proximity, to guide scheduling decisions, which assume that topological closeness (e.g., same core, L3, and NUMA domain) correlates with better performance. However, this assumption breaks in chiplet systems under bandwidth pressure. When the I/O chiplet becomes a bottleneck, the dominant cost shifts from cache locality to inter-chiplet contention and bandwidth availability (Figure 7-b). In such cases, placing threads "close" (e.g., within the same chiplet or NUMA) can actually worsen performance by concentrating traffic onto the same I/O chiplet links and memory channels. Conversely, placing threads farther apart may reduce contention and improve throughput. This inversion means that traditional affinity heuristics become ineffective, and even harmful, when bandwidth is under-provisioned. As shown in Figure 8-c, when sufficient bandwidth is available, affinity-based strategies perform well. However, once I/O chiplet load exceeds 80%, we observe up to 4.3× and 4.5× performance degradation in core-affinity and chiplet-affinity strategies, while placing threads on remote cores maintains stable performance. This indicates that logical proximity no longer reflects physical performance distance.

3.4 Case #3: Congestion Expands the Victim Region

As discussed in §2.2, congestion at the I/O chiplet and memory channel does not remain localized; instead, it propagates across the system, affecting threads that are otherwise unrelated. This creates a large "victim region", where threads that do not share data, cores, or even chiplets experience performance degradation (Figure 7-c). The problem is further exacerbated by the lack of fine-grained observability within the I/O chiplet: the OS cannot directly monitor internal queue states, routing congestion, or arbitration outcomes. EEVDF cannot identify which threads are causing or suffering from contention, making it difficult to isolate or mitigate interference under chiplet-induced contention. As shown in Figure 8-d, when the memory channel is overloaded, back-propagated congestion hurts the performance of indirectly affected threads by 1.7×, 1.2×, and 1.4× at the core, compute chiplet, and memory levels, respectively, while completely non-overlapping threads remain unaffected. Hence, chiplet-induced contention could cause system-wide interference that existing schedulers cannot handle.

3.5 Case #4: Load Balancing Can Increase Imbalance

Linux periodically migrates tasks to equalize CPU utilization across cores to improve fairness and overall efficiency. However, such CPU-centric load balancing can inadvertently worsen the I/O chiplet and memory system imbalance. By redistributing threads solely based on CPU load, the scheduler may consolidate memory-intensive threads onto the same compute chiplet or along the same compute-I/O chiplet paths, increasing contention at the I/O chiplet shared communication resources. While CPU utilization appears well balanced, the underlying memory subsystem becomes highly skewed (Figure 7-d), leading to degraded performance–a compute-balanced but memory-imbalanced state. EEVDF exacerbates this issue by using static topology and CPU load metrics, without visibility into dynamic I/O chiplet pressure. We demonstrate this effect in two scenarios. First, we compare EEVDF, a chiplet-affinity policy, and an oracle that performs end-to-end load balancing across cores, chiplet links, and memory channels. As memory channel skew increases (Figure 8-e), the oracle outperforms EEVDF and affinity-based policies by 13.6% and 17.9%, respectively. Second, for a synchronization-intensive workload (Figure 8-f), end-to-end load balancing achieves over 20.0% higher throughput. A bandwidth breakdown over different paths further reveals the root cause: intra-chiplet (intra-CC) bandwidth drops from 26.1 GB/s to 19.9 GB/s under load, while inter-chiplet (inter-CC) one degrades more severely from



(a) Case #1 (Free cores) **(b)** Case #1 (Busy cores)

(c) Case #2 (Topology affinity) **(d)** Case #3 (Congestion expansion)

(e) Case #4 (Bandwidth intensive) **(f)** Case #4 (Sync. intensive)

Figure 8. (a)/(b) compare the performance of EEVDF, chiplet-affinity (selecting local cores within the compute chiplet first), and oracle (being aware of compute-I/O chiplet link loading condition) schedulers. The link is 25%, 50%, 75%, and 100% loaded for the compute chiplet 0, 1, 2, and 3. (c) shows the application performance varying the I/O chiplet traffic load. (d) reports how memory channel contention impacts the performance of unrelated threads from the victim core, compute chiplet, and DIMM, comparing the ones that are non-overlapped. (e)/(f) show how a bandwidth- and synchronization-intensive application behaves under three scheduling policies varying the memory channel skewness. "Oracle" refers to the one that performs load balancing, considering the core, I/O chiplet, and memory. (f) also lists the bandwidth across different paths. We run PageRank from the GAPBS benchmark over Kronecker graphs in (a)–(e), and run a multi-threaded skiplist in (f).

9.5 GB/s to 4.2 GB/s. These results show that CPU-only load balancing can amplify chiplet-level contention, highlighting the need for joint core-I/O chiplet-memory scheduling.

3.6 Summary

These cases highlight a mismatch between existing CPU schedulers and chiplet-based architectures. Hardware topology is no longer a reliable proxy for performance. Instead, performance is increasingly limited by dynamic contention in the I/O chiplet and its link connectivity to neighboring components. Traditional CPU-centric policies (whether based on load balancing, affinity, or NUMA locality) can make poor scheduling decisions: free cores may be bandwidth-starved, "close" placement may amplify contention, interference may propagate beyond locality boundaries, and load balancing may worsen memory imbalance. Therefore, topology-aware scheduling is not only insufficient but often misleading without visibility into chiplet-level data movement, motivating the need for a new, traffic-aware scheduling approach.

4 I/O Chiplet as a Switch

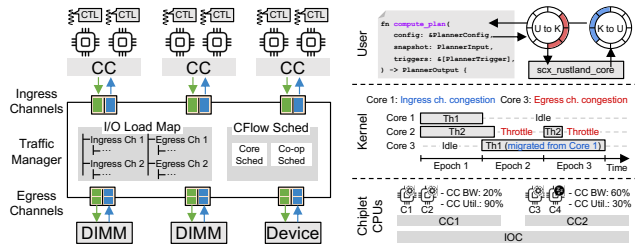
This section presents the design and implementation of `cSwitch`, a Linux scheduling framework that tackles the I/O chiplet wall (§2.2). Our system design goals are:

- **Performance.** `cSwitch` aims to maximize application performance, minimize unnecessary congestion, reduce memory access latency, and avoid pathological scheduling decisions that lead to pipeline stalls or bandwidth collapse.
- **Utilization.** `cSwitch` strives to strike a balance between concurrency and contention by carefully regulating thread placement and activation. It should adapt to runtime conditions, ensuring that neither compute resources remain idle nor the I/O chiplet becomes a persistent bottleneck.
- **Multi-tenancy.** `cSwitch` targets max-min fairness in sharing I/O chiplet and memory resources, mitigates chiplet-induced interference, and avoids a single application monopolizing bandwidth at the expense of others.

4.1 Key Idea and Overview

Key Idea. `cSwitch` *models the I/O chiplet as a software switch that orchestrates thread-induced data movement between compute chiplets and memory/devices.* In this abstraction, the I/O chiplet is decomposed into three logical components (Figure 9-a): (1) ingress channels, which capture traffic entering from compute chiplets; (2) egress channels, which forward traffic toward memory controllers and I/O devices; and (3) a traffic manager, which determines the ordering and rate at which cFlows (§4.3) traverse the switch. Rather than explicitly buffering, `cSwitch` adopts a bufferless design with phantom queues [21, 23, 111], which use a virtual capacity below the hardware limit to detect congestion before hardware saturation. Each thread generates one or more cFlows, annotated with a chiplet traffic label (§4.3) that encodes its communication path (including core, ingress/egress channels, traverse chiplets, and target memory/device) and traffic characteristics (e.g., BW and priority). This abstraction transforms CPU scheduling into a traffic scheduling problem, allowing `cSwitch` to leverage networking-inspired techniques to regulate chiplet traffic and mitigate the I/O chiplet wall.

System Overview. `cSwitch` adopts a decoupled execution architecture spanning kernel and userspace (Figure 9). Within the kernel, it introduces two key components. First,



(a) System Architecture (b) Execution Runtime

Figure 9. System architecture and execution runtime of cSwitch.

the I/O chiplet loading map (§4.2) continuously monitors congestion across ingress and egress channels by sampling architectural counters, serving as the basis for our traffic manager. Using delay-based signals (e.g., stall cycles, data serving latency), it estimates real-time contention levels without requiring direct visibility into hardware queues, as current chiplet processors do not expose the I/O chiplet or its contention state as OS-visible resources. Second, a kernel scheduler (§4.4 and §4.5) uses this information to enforce max-min fairness among competing threads, dynamically throttling, migrating, or prioritizing threads under contention. These decisions are made at fine granularity to quickly react to transient congestion. On the user side, cSwitch provides two mechanisms. Each cFlow is associated with a chiplet traffic label (§4.3), which captures its communication demand and guides scheduling decisions. Further, cSwitch exposes a scheduling policy interface (§4.6) atop eBPF for applications to define and inject a customized policy.

4.2 I/O Chiplet Loading Map

cSwitch introduces the *I/O Chiplet Loading Map*, a runtime telemetry mechanism that captures traffic conditions across the chiplet fabric. It periodically samples architectural performance counters to reconstruct the state of data movement through the I/O chiplet. We adopt a design analogous to a Linux `switchdev` device, where the I/O chiplet is abstracted into ingress channels (from compute chiplets) and egress channels (toward memory and devices). For every epoch, an ingress channel tracks (1) the number of requests (missed from LLC) and responses (returned to LLC), (2) the observed data serving (LLC miss completion) latency, and (3) the set of concurrent cFlows traversing the channel. An egress channel records (1) the number of requests/responses and (2) the congestion degree by synthesizing the service latency of responses. Table 2 summarizes the counters we use.

However, an I/O chiplet provides no direct observability into how requests from ingress channels are mapped to specific egress channels. This lack of visibility makes it difficult to attribute contention to specific threads, especially under memory interleaving, where data are distributed across multiple memory channels. To address this, cSwitch constructs a bipartite graph between ingress and egress channels using temporal differencing across snapshots. By comparing two consecutive snapshots, it computes the incremental request vectors at the ingress and egress sides. These deltas are then used to infer a mapping between ingress and egress channels: in cases such as one-to-one (1⟶1) or one-to-many (1⟶N) mappings, the correspondence can be directly inferred from matching request increments. This becomes more challenging in many-to-one (N⟶1) scenarios. cSwitch leverages delay signals as implicit weights to disambiguate contributions. The intuition is that flows experiencing higher delay increases at the ingress side are more likely to contribute to downstream congestion. Thus, cSwitch assigns weights to ingress–egress edges proportional to the observed latency inflation at the ingress channel and distributes the inferred egress load accordingly. While not perfectly accurate, in practice, we find that it provides sufficient visibility into traffic distribution and contention hotspots.

Module	Model	Architectural Counters
I/O Load Map	IngressCh.Req#	L2FillRspSrc:All
	IngressCh.Resp#	L2FillRspSrc:All
	IngressCh.Serv.Lat	L3_XiSampledLatency
		L3_XiSampledLatencyRequests
	EgressCh.Req#	DF_PERF_CTL/CTR (CS:Wr)
	EgressCh.Resp#	DF_PERF_CTL/CTR (CS:Rd)
Chiplet Traffic Label	PerfAttr.Req#	L2FillRspSrc: Cache/Dram/CCX
	PerfAttr.Resp#	L2FillRspSrc:All
	PerfAttr.Serv.Lat	L3_XiSampledLatency
		L3_XiSampledLatencyRequests
	Contention.Stall	DF_PERF_CTL/CTR (SrcDst:CS)
		ExNoRetire
		LsNotHaltedP0Cyc
	Contention.Delay	L3_XiSampledLatency:Dram/Cache
		L3_XiSampledLatencyRequests:Dram/Cache

Table 2. Architectural performance counters used by cSwitch.

4.3 Chiplet Traffic Label

In cSwitch, a **cFlow** is defined as a thread running on one core that issues memory or I/O requests to a target DIMM or device. To facilitate scheduling under the I/O chiplet wall, inspired by the HASS scheduler [104], cSwitch proposes the *Chiplet Traffic Label*, a per-cFlow signature that describes its traffic characteristics and resource demand. It represents how a thread interacts with the chiplet fabric and bridges the application behavior and the I/O Chiplet Loading Map.

Our chiplet traffic label has three components. (1) *Communication Path.* Each cFlow is associated with its inferred ingress and egress channels, derived by correlating per-thread scheduling activity with the I/O chiplet loading map. It identifies which compute-I/O links and memory/device channels the thread primarily exercises. (2) *Performance Attributes.* cSwitch tracks the number of requests issued and completed by each thread, along with the observed service latency. These metrics are collected using per-core architectural counters, e.g., L2FillRspSrc, L3_XiSampledLatency, and L3_XiSampledLatencyRequests. With L2 cache performance counters, it captures a thread's bandwidth request to each component. With L3 cache performance counters, it captures the current domain's load. (3) *Contention.* To quantify how much a thread is affected by chiplet congestion,

cSwitch combines multiple signals, including core stall cycles and latency indicators like L3-to-L3 and L3-to-DRAM access delays, as a proxy for the pressure a thread experiences on the I/O chiplet. We continuously update the label at runtime (Alg1:L8-9). When a thread is scheduled on a core, cSwitch profiles its behavior and refreshes its label using recent counters. To reduce sensitivity to transient fluctuations and adapt to dynamic workloads, cSwitch applies an exponentially weighted moving average (EWMA) over historically collected metrics and hysteresis, acting as a low-pass filter that stabilizes bandwidth estimation and contention.

4.4 The cFlow Scheduling Algorithm

cSwitch reframes CPU scheduling as a flow scheduling problem [22, 51, 112], where each thread is modeled as a flow competing for shared I/O chiplet resources. The scheduler operates in periodic epochs and uses the I/O chiplet loading map (§4.2) and chiplet traffic labels (§4.3) to jointly control (i) where threads run and (ii) how fast they inject data traffic.

Operating Regime. cSwitch is most effective when thread placement creates avoidable I/O chiplet contention. Using the I/O Chiplet Loading Map, it steers cFlows toward less-congested ingress/egress paths, and throttles overloaded egresses. As the I/O chiplet approaches global saturation, placement opportunities diminish, reducing cSwitch's gains over locality-aware schedulers, although throttling can still limit overload and mitigate imbalance.

Thread Placement. At creation, cSwitch follows a locality-first policy to preserve cache affinity: a thread is placed on an available core within the same compute chiplet as its parent (Alg1:L3). If none is available, it selects the least-utilized core globally, breaking ties randomly (as in EEVDF) (Alg1:L4–L5). The thread then registers its cFlows (based on the target memory/device), whose initial chiplet traffic label (CTlabel) is populated from per-core counters (Alg1:L8–L9).

Egress-Driven Rate Control. At each epoch, cSwitch first scans all egress channels (to memory and device endpoints) and classifies them as congested or under-subscribed using delay-based signals from the loading map (Alg1:L10–L11). Specifically, we monitor the effective bandwidth at the I/O chiplet to the memory controller interface (i.e., coherent station) and compare it against the provisioned link capacity to detect saturation. It then uses the EWMA-smoothed measured delay along with dual-threshold hysteresis to determine the contention degree. For uncongested channels, cSwitch defers to the default scheduler. For congested channels, cSwitch computes a weighted max-min fair bandwidth share across the set of traversing flows using their estimated demand (from traffic labels) and inferred channel capacity (Alg1:L12–L13). Flows that exceed their fair share are throttled by reducing their CPU time slice (thus lowering request injection rate) and, if necessary, by temporarily deprioritizing them in the runnable queue (Alg1:L14–L16). This implements a form of end-host rate limiting [20, 26, 97] analogous to network congestion control. Throttled threads might be marked as *Migratable* candidates in the next phase.

Algorithm 1 The cFlow Scheduler

```
1:  procedure cFlow_Scheduler ▷ Called Every Sched. Epoch
2:      for th ∈ newThreads do ▷ Thread Placement
3:          d ← th.parent.schedDomain
4:          while ¬d.hasIdleCores() ∧ d.upperSchedDomain do
5:              d ← d.upperSchedDomain
6:          th.assign(arg min_{core∈d}(¬core.idle(), core.util(), core))
7:      IOLoadMap.update() ▷ cSwitch Telemetry Update
8:      for cf ∈ IOLoadMap.allCFlows do
9:          cf.CTlabel←updateCTlabel(cf, IOLoadMap)
10:     for ch ∈ IOLoadMap.egChs do ▷ Egress Ch. Rate Control
11:         if ch.isCongested() then
12:             culCFlow ← arg max_{f∈ch.cflows} f.CTlabel[ch].bw
13:             newBW ← ch.bwMaxMinRealloc(culCFlow)
14:             schedPlan.append(culCFlow.th, Throttle, newBW)
15:             IOLoadMap.update(culCFlow, newBW)
16:             culCFlow.CTlabel.update(newBW)
17:     for ch ∈ IOLoadMap.inChs do ▷ Ingress Ch. Refinement
18:         if ch.isCongested() then
19:             for cflow ∈ ch ∧ cflow.isBWIntensive() do
20:                 migratePlan.append(cflow.th)
21:     migratePlan.sort(key=λcf.(cf.priority, cf.demand))
22:     inChs ← IOLoadMap.inChs.sort(key=λd.(d.AvailBW))
23:     NPlan ← bipartMatch(migratePlan.CFlows, inChs)
24:     for f ∈ NPlan do ▷ Thread Migration
25:         if f.isMegaCFlow∧¬f.demandIsSatisfied() then
26:             f ← f.decomposeMega()
27:             NPlan ←bipartMatch(migratePlan.CFlows, Chs)
28:     schedPlan.apply(); (migratePlan ← NPlan).apply()
```

Ingress-Enhanced Refinement. cSwitch then traverses each ingress channel to validate whether congestion is mitigated after applying the tentative throttling plan (Alg1:L17–L18). If it remains congested, the local rate control is insufficient, for example, due to structural hotspots or skewed placement. In this case, cSwitch identifies the dominant contributing threads–those with the highest bandwidth demand on the congested channel based on their traffic labels–and marks them as migratable candidates (Alg1:L19–L20). This is because the corresponding cFlows would benefit more from relocation to paths with greater available bandwidth. Otherwise, if congestion diminishes, cSwitch accepts the current plan without triggering migration, avoiding unnecessary thread movement. This two-phase design allows cSwitch to first apply lightweight rate control and only escalate to placement changes when congestion persists.

Thread Migration. For threads marked as migratable, cSwitch performs a bandwidth-aware bipartite matching between cFlows (threads) and candidate ingress channels. The goal is to relocate high-impact threads away from congested channels while maximizing overall bandwidth utilization. Concretely, cSwitch ranks cFlows by a synthesized score that captures bandwidth demand, priority, and congestion

contribution (derived from traffic labels and loading map signals) (Alg1:L21). In parallel, ingress channels are ranked by their residual capacity (Alg1:L22). cSwitch then applies a greedy matching algorithm (Alg1:L23): it iteratively selects the highest-demand cFlow and assigns it to the ingress channel with the largest available bandwidth that minimally overlaps with existing high-traffic flows. This approximates a maximum-weight matching [43–45] while keeping runtime overhead low. The selected channel is mapped to a target core, and the thread is migrated accordingly. During this process, cSwitch also incorporates locality constraints, such as preferring cores within nearby chiplets or NUMA domains when multiple candidates provide similar bandwidth, to reduce migration overhead and preserve cache affinity.

Schedule Plan Apply. Finally, cSwitch commits the scheduling plan by updating per-CPU run queues, adjusting time slices, and issuing migrations (Alg1:L28). The combined use of rate control (throttling) and traffic-aware placement (migration) allows cSwitch to react quickly to the I/O chiplet load condition and tackle the issues above (§3.2–§3.5).

4.5 Cooperative Group Scheduling

For scheduling groups, we propose a coflow-inspired abstraction [36–38], called a **MegaCFlow**, to coordinate threads while remaining compatible with the cFlow scheduler (§4.4). This improves inter-thread communication and avoids fragmented thread placements that cause chiplet contention.

MegaCFlow Construction. cSwitch groups corresponding threads into a MegaCFlow either using application hints (§4.6) or detecting correlated traffic patterns (e.g., synchronized behavior in Linux lockstat [11]). A MegaCFlow aggregates per-cFlow traffic labels, and is summarized by a dominant ingress (source) and dominant egress (destination), selected as the channels with the highest bandwidth consumption. This preserves the primary communication path and performance characteristics, and interaction with chiplets.

Integration with the cFlow Scheduler. Let D_M be the aggregate demand of a MegaCFlow M and $C_{ingress}$,C_{egress} be the available capacities of its dominant/egress channels. If $D_M \leq min(C_{ingress}, C_{egress})$, cSwitch schedules M as a single unit and follows the same scheduling logic as §4.4. MegaCFlows are prioritized over normal ones to preserve locality and synchronization benefits. Threads in M are preferentially co-located on cores sharing the dominant ingress path. The scheduler allocates time slices to threads in M proportionally to their contributions while enforcing a shared rate cap for the MegaCFlow. If D exceeds channel capacity, M becomes splittable (Alg1:L24-27). cSwitch then decomposes M into sub-cFlows and performs bandwidth-aware packing across ingress/egress channels. Specifically, cSwitch ranks candidate ingress–egress pairs by their residual bandwidth and greedily assigns as many cFlows from M as possible to the channel with the highest available capacity. The remaining threads are iteratively placed onto secondary channels in descending order of available bandwidth, effectively spreading the mega-flow across multiple less-contended paths. Each sub-cFlow inherits its chiplet traffic label, allowing cSwitch to apply standard rate control (throttling) to cap injection rates and migration to enforce the new placement.

4.6 User-defined Scheduling Policy

cSwitch leverages Linux's sched_ext (scx) framework [9] and eBPF to decouple policy from mechanism and enable user-defined scheduling. The kernel component uses eBPF programs to query architectural counters and collect runtime telemetry, used to build the loading map and traffic labels, which are also exposed to userspace at low overhead.

Developers inject a bespoke policy via planner.rs and policy.rs. The planner customizes a scheduling plan by invoking compute_plan(), which takes cSwitch internal states and outputs actions, such as time-slice adjustments (throttling), thread migrations, and priority updates. The policy module defines how these actions are interpreted and enforced, allowing scheduling strategies to be plugged in without modifying the kernel. The resulting plan is then communicated back to the kernel via sched_ext, which applies it by updating per-core run queues and scheduling parameters. As such, one can implement alternative scheduling objectives or completely replace existing policies. For example, we have re-implemented several representative schedulers atop cSwitch (including ARCAS [48] and Caladan [80]) to demonstrate that one can realize new scheduling policies while preserving compatibility with existing designs.

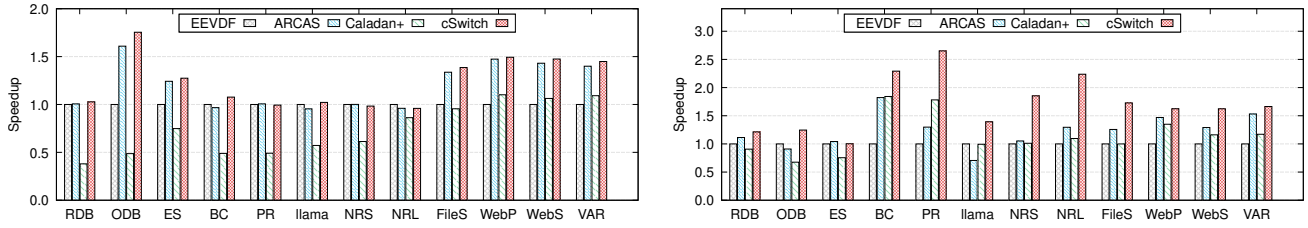
cSwitch complements, rather than replaces, existing abstractions: it respects CPU affinity, cpusets, and cgroup constraints while making chiplet-aware placement and rate-control decisions. Through sched_ext, cSwitch coexists with other scheduler classes. It currently targets intra-socket chiplet fabrics, while existing NUMA policies and memory placement remain unchanged; extending to inter-socket NUMA requires adding topology information. blkio regulates device bandwidth, whereas cSwitch manages the resulting contention within the CPU-I/O chiplet.

5 Evaluation

5.1 Experimental Methodology

Hardware Testbed. We use a Supermicro 1U server with the AMD EPYC 9634 chiplet CPU (Table 1), 1 TB DDR5, and 4× 256 GB Micron CZ120 CXL memory modules for evaluation. We disable SMT and Turbo Boosting. The server runs Ubuntu 22.04.5 LTS with Linux kernel 6.19.6. We also use a non-chiplet 2U Gigabyte server with Intel CPUs and 256GB DDR5 when examining the I/O chiplet impact (§2.2).

Benchmarks and Metrics. We use a range of workloads spanning DB, data analytics, ML, and HPC. They include Memcached[12], Redis[15], DuckDB[5], IoT-Benchmark[8],



(a) I/O chiplet is unloaded

(b) I/O chiplet is loaded

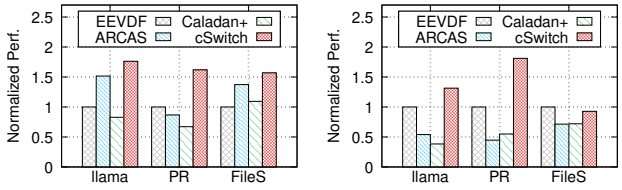
Figure 10. (a)/(b) report the speedup of cSwitch, ARCAS, and Caladan+ over the Linux EEVDF when the I/O chiplet is unloaded/loaded. RDB=RocksDB. ODB=OrientDB. ES=ElasticSearch. BC/PR∈GAPBS. NRS|NRL∈Node Replication Library. FileS|WebP|WebS|VAR∈Filebench.

Elasticsearch[7], PARSEC 3.0[31], NAS parallel benchmark[28], GAP benchmark suite (GAPBS)[27], `llama.cpp`[10], Filebench[81], Node Replication Library[33], FxMark[83], and Will it Scale[19]. We use the YCSB[41], Memtier[13], and TPC-H[17] when generating application requests. We mainly report application performance (latency and throughput) and platform microarchitectural metrics (e.g., stall cycles and memory/IO bandwidth) to capture the interaction between scheduling and chiplet-level data movement. We generated background traffic using sequential reads and controlled its rate by inserting nop instructions into the read loop.

Comparison Baselines. We compare cSwitch against Linux EEVDF and two recent schedulers. ARCAS [48] is a chiplet-aware scheduler that combines topology-aware task placement with hardware-guided memory allocation based on fine-grained monitoring. Caladan+ [50] performs μs-scale monitoring and load-aware task placement. Neither of them accounts for dynamic I/O chiplet contention. These baselines allow us to evaluate how explicit I/O chiplet awareness improves scheduling decisions under contention.

5.2 cSwitch Achieves Higher Performance

We compare cSwitch against EEVDF, ARCAS, and Caladan+ while varying the I/O chiplet load across 12 applications on DDR5 DIMMs. The speedup is defined as the ratio of application performance achieved by each scheduler relative to EEVDF. As shown in Figure 10, when the I/O chiplet is lightly loaded, cSwitch achieves performance comparable to ARCAS and outperforms EEVDF and Caladan+ by 1.2× and 1.9× on average, as both cSwitch and ARCAS exploit chiplet topology to preserve communication locality. However, as background threads inject load into the I/O chiplet, the gap widens. Leveraging the I/O Chiplet Loading Map, cSwitch captures real-time contention and uses the cFlow scheduler to steer threads toward higher-bandwidth paths and apply rate control to mitigate congestion. Thus, cSwitch achieves 1.2-2.7× speedup over EEVDF, and outperforms ARCAS and Caladan+ by 1.4× and 1.7×. The improvement stems from cSwitch 's ability to reason about dynamic data movement and contention, whereas ARCAS lacks visibility into I/O chiplet load and Caladan+ does not model it as a shared bottleneck, yielding suboptimal scheduling under contention.



(a) Free cores

(b) Busy cores

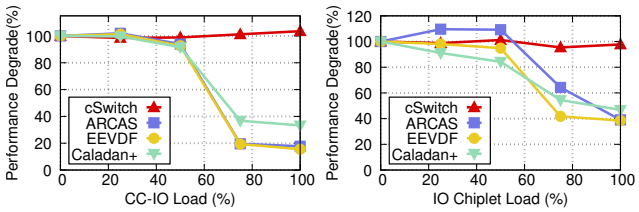
Figure 11. (a) and (b) compare the performance of cSwitch, ARCAS, EEVDF, and Caladan+ running llama.cpp, PageRank, and File Server when there are heterogeneous free/busy cores as described in §3.2. We normalize the performance to the EEVDF case.

5.3 cSwitch Chooses a Better Free/Busy Core

We evaluate how cSwitch addresses Case #1 (§3.2), where cores differ in effective performance due to heterogeneous I/O chiplet paths. Unlike existing schedulers, cSwitch estimates the end-to-end available bandwidth from each core to memory via ingress and egress channels, enabling the cFlow scheduler to place threads on cores with higher effective bandwidth. Using the same setup as §3.2, Figure 11 shows that in free-core scenarios, cSwitch outperforms ARCAS, EEVDF, and Caladan+ by 1.4×, 1.7×, and 2.0×, respectively, and achieves 2.6×, 1.4×, and 2.7× speedups under busy-core conditions. These gains arise from cSwitch's ability to avoid bandwidth-starved cores and make placement decisions based on chiplet-level visibility, whereas existing schedulers treat cores as interchangeable.

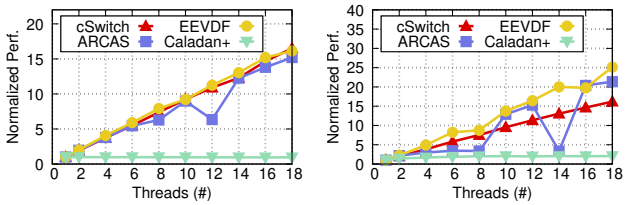
5.4 cSwitch Avoids Traffic Contention

We evaluate how cSwitch reacts to dynamic contention and avoids hotspots in the chiplet fabric: Case #2 (§3.3) and Case #4 (§3.5). We first inject background traffic on a shared compute-I/O chiplet link. As shown in Figure 12, ARCAS, EEVDF, and Caladan+ begin to degrade once load exceeds 60%, with up to 82.3%, 84.5%, and 66.7% performance loss, respectively. In contrast, cSwitch maintains stable performance by detecting congestion and using the cFlow scheduler to migrate threads to alternative ingress channels with available bandwidth. We further stress the entire I/O chiplet by introducing system-wide background traffic. cSwitch continues to sustain consistent performance by exploiting bandwidth diversity across multiple channels, whereas other



(a) Varying CC-IO load **(b)** Varying I/O chiplet load

Figure 12. (a) and (b) show the performance degradation when varying the traffic load of the compute-I/O chiplet link and the overall I/O chiplet under `cSwitch`, ARCAS, EEVDF, and Caladan+.



(a) Scaling w/o chiplet load. **(b)** Scaling w/ chiplet load.

Figure 13. (a) and (b) present the performance of PageRank when varying the number of threads under `cSwitch`, ARCAS, EEVDF, and Caladan+. We normalize the perf. to the single-thread case.

schedulers suffer more than 60% degradation. In particular, ARCAS performs poorly because it relies on LLC miss signals without visibility into available bandwidth headroom, leading to persistent placement on congested paths.

5.5 `cSwitch` Scales Well Within/Across Chiplets

We evaluate how `cSwitch` scales within and across chiplets under varying contention. As shown in Figure 13, when intra-chiplet traffic is low, `cSwitch` achieves performance comparable to EEVDF, indicating that it does not introduce unnecessary overhead in under-subscribed scenarios. However, as I/O chiplet load increases, `cSwitch` significantly outperforms prior schedulers, achieving 1.4×, 1.5×, and 3.9× higher performance than ARCAS, EEVDF, and Caladan+, respectively. This improvement stems from the fact that the cFlow scheduler, which places threads based on ingress and egress bandwidth availability inferred from the loading map, rather than relying solely on core availability or topology-based affinity. By avoiding oversubscribed paths and balancing traffic across chiplets, `cSwitch` maintains high scalability under contention. In contrast, ARCAS lacks visibility into dynamic I/O chiplet load, while Caladan+ performs worst due to the high context-switching overheads associated with its microsecond-scale scheduling model.

5.6 `cSwitch` Mitigates Backpropagation Stalls

We evaluate whether `cSwitch` can mitigate the backpropagated stalls caused by congestion at the DIMM and device side (§3.4). Using the same setup as in §3.4, we progressively increase load on the I/O-chiplet-DIMM and I/O-chiplet-CXL paths. `cSwitch` achieves up to 1.7× latency reduction compared with prior schedulers. The key reason is that `cSwitch` explicitly captures downstream congestion through the I/O Chiplet Loading Map, including delay inflation on egress channels, and feeds this information into the cFlow scheduler. When backpressure begins to propagate from memory or device endpoints, `cSwitch` reacts in two ways: it first applies egress-driven rate control to slow down aggressive threads that over-inject requests, reducing the pressure seen by the congested DIMM or CXL channel; if contention persists, it then uses bandwidth-aware migration to steer threads toward less-congested ingress and egress channels. This breaks the congestion propagation chain before it stalls the core pipeline. In contrast, EEVDF, ARCAS, and Caladan+ lack explicit visibility into end-to-end I/O chiplet backpressure, and therefore, continue to schedule threads onto saturated paths, causing stall cycles to accumulate at the cores.

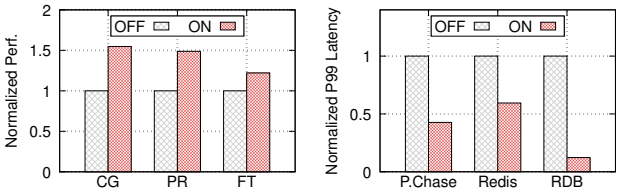
5.7 `cSwitch` Drill-Down

Programmability. We evaluate `cSwitch`'s programmability by porting two user-space schedulers, ARCAS and Caladan+, to its interface. Using `sched_ext` and eBPF, we adapt their policies to `cSwitch` 's planner while reusing its telemetry, including the I/O Chiplet Loading Map and chiplet traffic labels. This separation of policy and mechanism lets both schedulers retain their original designs while gaining chiplet-aware visibility without reimplementing low-level monitoring.

Platform Portability. We also ported `cSwitch` to an Intel Xeon Gold 6530 Emerald Rapids processor with two compute tiles. After updating the platform topology and mapping the required Intel PMCs, `cSwitch` ran without further architectural changes. Unlike AMD EPYC, Emerald Rapids exposes a single shared LLC with limited visibility and control over inter-chiplet paths. We therefore evaluate only egress throttling. On a pointer-chasing microbenchmark, throttling reduced the P99 latency of 4,096-access windows by 33%.

I/O Loading Map Accuracy. We evaluate the accuracy of `cSwitch`'s I/O Chiplet Loading Map for both ingress and egress traffic. For ingress traffic, we compare the inferred bandwidth of ten sequential-read workers against their measured bandwidth under heterogeneous rate limits. `cSwitch` estimates bandwidth with approximately 10% error. For egress traffic, we test whether `cSwitch` can identify a villain thread targeting a specific memory controller amid two background workers. Across the experiment, `cSwitch` correctly identifies the villain in 95% of high-load epochs on average.

System Ablation Study. We evaluate the contributions of `cSwitch`'s two key mechanisms by independently enabling ingress migration and egress throttling. For ingress migration, we run workloads across four compute chiplets alongside 18 heterogeneous background-noise threads. As shown in Figure 14, ingress migration improves performance by up to 1.5×, with CG benefiting the most. For egress throttling, we run 30 workload threads while injecting villain threads



(a) Ingress Migration **(b)** Egress Throttling

Figure 14. (a) shows the normalized performance improvement when ingress migration is enabled on NPB CG, FT, and PageRank. (b) presents the normalized P99 latency reduction when egress throttling is enabled on pointer chasing, Redis, and RocksDB.

that contend at memory controller 0. Egress throttling reduces P99 latency by 57% for pointer chasing and 88% for RocksDB, substantially reducing tail stalls.

`cSwitch` Overheads. `cSwitch` incurs modest CPU and memory overhead. For the 40-thread GAPBS PageRank benchmark, the userspace scheduler consumes 6.22 CPU-seconds, or 3.46% of the application's 179.60 CPU-seconds. Planner computation accounts for only 1.75 CPU-seconds (0.97%), with the remaining overhead primarily from the userspace scheduling runtime. Memory overhead is negligible: `cSwitch` uses lightweight per-core data structures, consuming approximately 0.01% of system memory even with 84 cores across 12 chiplets. Hence, `cSwitch` delivers substantial performance gains with modest runtime and memory overheads.

6 Related Work

Linux Scheduler. Linux supports diverse policies via scheduling classes [16], ordered by priority. A thread in a higher priority class can preempt threads with lower priorities, allowing applications to alter scheduling behavior to meet their performance demands. For example, [46] develops an EDF (earliest deadline first) scheduling for real-time workloads. The CFS [3] and recent EEVDF [6] schedulers target fairness. Jean-Pierre Lozi *et al.* [77] identify four performance bugs that cause core idleness even though there are runnable threads. Enoki [82] proposes a high velocity development framework for Linux kernel schedulers atop Rust. `cSwitch` develops a scheduling framework that improves the Linux scheduling efficiency under the I/O chiplet wall.

Userspace Scheduler. Researchers have developed bespoke userspace schedulers for decades. Early systems such as CPU inheritance [49] and Capriccio [114] provide user-level control over thread scheduling and resource management. Recent systems such as Shenango [91], Arachne [94], and Syrup [63] enable dynamic core allocation and application-defined scheduling policies. Similar programmability is widely used in programmable networks [71, 73, 76, 92, 95, 96, 101] and computational storage [55, 61, 64, 84–86], where userspace control planes manage scheduling and resource allocation. Likewise, `cSwitch` exposes a programmable interface for application-defined, chiplet-aware scheduling policies.

Fast Threading. Our design is inspired by pioneering efforts on building fast scheduling primitives. For example, scheduler activations [25] allow the OS kernel to expose the execution context to the userspace thread scheduler, such that it can effectively handle events, like modifying thread data structures, executing user-level threads, and making requests of the kernel. ghOSt [58] enables delegation of Linux scheduling policies to userspace processes via a customized abstraction and interface. Caladan [50] introduces a central scheduling core, collects fine-grained execution telemetry, and uses a kernel module to bypass the standard Linux scheduler for microsecond-scale task monitoring and placement. The NEST scheduler [69] runs priority threads on warm cores to harness its high frequency. Shinjuku [62] uses the hardware virtualization support to achieve fast preemption. Our upcall-downcall mechanism follows these design practices.

Chiplet Architecture and System Support. Chiplets are becoming a predominant approach to future silicon systems, spanning CPUs, accelerators, and networking devices. Architecture research explores designs such as CPElide [42] for inter-chiplet dependency management and Hetero-IF [47] for flexible die-to-die communication. Meanwhile, chiplets require new software stacks to manage on-package communication and contention [24, 70, 100, 115]. Their impact extends to network fabric [34, 53, 56, 75, 102, 103, 123, 125], disaggregated storage [74, 116, 121, 122], and rack-scale computing and memory infrastructure [54, 57, 72, 78, 119, 124], particularly for ML/AI systems. `cSwitch` takes a step in this direction by making chiplet-fabric dynamics visible to CPU scheduling, while many opportunities remain for redesigning system software around chiplet architectures.

7 Conclusion

Chiplet-based processors shift performance bottlenecks to the I/O chiplet, where shared data movement paths create the I/O Chiplet Wall and render existing CPU-centric scheduling abstractions ineffective. We build `cSwitch`, a Linux scheduling framework that addresses this challenge by modeling the I/O chiplet as a software switch and transforming CPU scheduling into a flow scheduling problem. By combining runtime telemetry (I/O Chiplet Loading Map), per-flow characterization (Chiplet Traffic Label), and a contention-aware (for cFlow&MegaCFlow) scheduler with rate control and migration, `cSwitch` effectively coordinates computation and data movement under chiplet-induced contention. Our evaluation demonstrates that `cSwitch` improves performance, scalability, and multi-tenancy with modest overheads.

Acknowledgements

We thank the anonymous reviewers and our shepherd for their comments and feedback. This work is supported in part by NSF grants CNS-2212192 and CAREER-2339755. Joontaek Oh and Ming Liu are the corresponding authors.

References

[1] 2024. 4TH GEN AMD EPYC Processor Architecture. https://www.amd.com/content/dam/amd/en/documents/products/epyc/4th-gen-epyc-processor-architecture-white-paper.pdf.

[2] 2026. Add support for Sub-NUMA cluster (SNC) systems. https://lwn.net/Articles/966963/.

[3] 2026. CFS Scheduler. https://docs.kernel.org/scheduler/sched-design-CFS.html.

[4] 2026. CPU Idle Time Management. https://docs.kernel.org/admin-guide/pm/cpuidle.html.

[5] 2026. DuckDB: an In-process SQL OLAP Database Management. https://duckdb.org.

[6] 2026. EEVDF Scheduler. https://docs.kernel.org/scheduler/sched-eevdf.html.

[7] 2026. Elasticsearch: Distributed Search & Analytics. https://www.elastic.co/elasticsearch.

[8] 2026. IoT-benchmark is a tool for benchmarking TSDB in IoT scenario. https://github.com/thulab/iot-benchmark.

[9] 2026. Linux Extensible Scheduler Class. https://docs.kernel.org/scheduler/sched-ext.html.

[10] 2026. LLM inference in C/C++. https://github.com/ggml-org/llama.cpp.

[11] 2026. lockstat–report kernel lock and profiling statistics. https://man.cx/lockstat(1).

[12] 2026. memcached: a Distributed Memory Object Caching System. https://memcached.org.

[13] 2026. memtier_benchmark: A High-Throughput Benchmarking Tool for Redis & Memcached. https://redis.io/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/.

[14] 2026. Reconsidering the scheduler's wake_wide() heuristic. https://lwn.net/Articles/728942/.

[15] 2026. Redis: an In-memory Key-value Data Store. https://redis.io.

[16] 2026. sched – overview of CPU scheduling. https://man7.org/linux/man-pages/man7/sched.7.html.

[17] 2026. The TPC-H Benchmark. https://www.tpc.org/tpch/.

[18] 2026. What is NUMA? https://www.kernel.org/doc/html/v5.6/vm/numa.html.

[19] 2026. Will It Scale Benchmark. https://github.com/antonblanchard/will-it-scale.

[20] Tarek Abdelzaher and Kang G Shin. 1998. End-host architecture for QoS-adaptive communication. In *Proceedings. Fourth IEEE Real-Time Technology and Applications Symposium*. 121–130.

[21] Yehuda Afek, Yishay Mansour, and Zvi Ostfeld. 1996. Phantom: A simple and effective flow control scheme. In *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*. 169–182.

[22] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. 2010. Hedera: dynamic flow scheduling for data center networks. In *NSDI*, Vol. 10. 89–92.

[23] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less is more: Trading a little bandwidth for {Ultra-Low} latency in the data center. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*. 253–266.

[24] Seunghyun An, Joontaek Oh, and Ming Liu. 2025. Server Chiplet Networking. In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks (HotNets'25)*. 289–299.

[25] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. 1991. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*. 95–109.

[26] Hitesh Ballani, Paolo Costa, Christos Gkantsidis, Matthew P Grosvenor, Thomas Karagiannis, Lazaros Koromilas, and Greg O'Shea. 2015. Enabling end-host network functions. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 493–507.

[27] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[28] NAS Parallel Benchmarks. 2006. Nas parallel benchmarks. *CG and IS* (2006).

[29] Srikant Bharadwaj and Tushar Krishna. 2024. InC2: Design of Interconnection Systems for Composable Chiplet Architectures. In *2024 17th IEEE/ACM International Workshop on Network on Chip Architectures (NoCArc)*. 1–6.

[30] Srikant Bharadwaj, Jieming Yin, Bradford Beckmann, and Tushar Krishna. 2020. Kite: A family of heterogeneous interposer topologies enabled via accurate interconnect modeling. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[31] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 72–81.

[32] Thomas Burd, Wilson Li, James Pistole, Srividhya Venkataraman, Michael McCabe, Timothy Johnson, James Vinh, Thomas Yiu, Mark Wasio, Hon-Hin Wong, et al. 2022. Zen3: The AMD 2nd-generation 7nm x86-64 microprocessor core. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. IEEE, 1–3.

[33] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K Aguilera. 2017. Black-box concurrent data structures for NUMA architectures. *ACM SIGPLAN Notices* 52, 4 (2017), 207–221.

[34] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, et al. 2024. Demystifying datapath accelerator enhanced off-path smartnic. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP'24)*. 1–12.

[35] Zhiqiang Chen, Wenwen Fu, Yongwen Wang, and Hongwei Zhou. 2026. A Deadlock-Free Bridge Module for Inter-Chiplet Cache-Coherent Communication in an Open Chiplet Ecosystem. In *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1–13.

[36] Mosharaf Chowdhury and Ion Stoica. 2012. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. 31–36.

[37] Mosharaf Chowdhury and Ion Stoica. 2015. Efficient coflow scheduling without prior knowledge. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 393–406.

[38] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 443–454.

[39] Ampere Computing. 2024. Breaking Boundaries: AmpereOne's Disaggregation Strategy for the Next-Gen Cloud. https://amperecomputing.com/blogs/next-gen-cloud.

[40] The UCIe Consortium. 2025. Universal Chiplet Interconnect Express (UCIe). https://www.uciexpress.org.

[41] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[42] Preyesh Dalmia, Rajesh Shashi Kumar, and Matthew D. Sinclair. 2024. CPElide: Efficient Multi-Chiplet GPU Implicit Synchronization. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 700–717.

[43] Ran Duan and Seth Pettie. 2010. Approximating maximum weight matching in near-linear time. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*. 673–682.

[44] Ran Duan and Seth Pettie. 2014. Linear-time approximation for maximum weight matching. *Journal of the ACM (JACM)* 61, 1 (2014), 1–23.

[45] Ran Duan and Hsin-Hao Su. 2012. A scaling algorithm for maximum weight matching in bipartite graphs. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*. 1413–1424.

[46] Dario Faggioli, Michael Trimarchi, Fabio Checconi, Marko Bertogna, and Antonio Mancina. 2009. An implementation of the earliest deadline first algorithm in linux. In *Proceedings of the 2009 ACM symposium on Applied Computing*. 1984–1989.

[47] Yinxiao Feng, Dong Xiang, and Kaisheng Ma. 2023. Heterogeneous die-to-die interfaces: Enabling more flexible chiplet interconnection systems. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 930–943.

[48] Alessandro Fogli, Bo Zhao, Peter Pietzuch, and Jana Giceva. 2025. ARCAS: Adaptive Runtime System for Chiplet-Aware Scheduling. arXiv:2503.11460 [cs.AR] https://arxiv.org/abs/2503.11460

[49] Bryan Ford and Sai Susarla. 1996. CPU inheritance scheduling. In *OSDI*, Vol. 96. 91–105.

[50] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 281–297. https://www.usenix.org/conference/osdi20/presentation/fried

[51] Michael R Garey, David S Johnson, and Ravi Sethi. 1976. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research* 1, 2 (1976), 117–129.

[52] Christopher Gonzalez, Huichu Liu, Mijung Noh, Eric Karl, Thomas Toifl, and Shawn Hsu. 2021. F5: Enabling New System Architectures with 2.5 D, 3D, and Chiplets. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 64. IEEE, 529–532.

[53] Zerui Guo, Jiaxin Lin, Yuebin Bai, Daehyeok Kim, Michael Swift, Aditya Akella, and Ming Liu. 2023. LogNIC: A High-Level Performance Model for SmartNICs. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*. 916–929.

[54] Zerui Guo, Emily Shriver, and Ming Liu. 2026. Building A CSFQ-Inspired Transport for Switched CXL Memory Pooling. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'26)*. 969–986.

[55] Zerui Guo, Hua Zhang, Chenxingyu Zhao, Yuebin Bai, Michael Swift, and Ming Liu. 2023. LEED: A Low-Power, Fast Persistent Key-Value Store on SmartNIC JBOFs. In *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM'23)*. 1012–1027.

[56] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. 2023. A Generic Service to Provide In-Network Aggregation for Key-Value Streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), Volume 2*. 33–47.

[57] Wentao Hou, Jie Zhang, Zeke Wang, and Ming Liu. 2024. Understanding Routable PCIe Performance for Composable Infrastructures. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*. 297–312.

[58] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghOSt: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 588–604.

[59] IDTechEx. 2025. Chiplet Technology 2025-2035: Technology, Opportunities, Applications. https://www.idtechex.com/en/research-report/chiplet-technology-2025-2035-technology-opportunities-applications/1041.

[60] Vikram Jain, Wei Tang, Zuoguo Wu, Viansa Schmulbach, Sophia Shao, Zhengya Zhang, and Borivoje Nikolic. 2024. Design approach for die-to-die interfaces to enable energy-efficient chiplet systems. In *Proceedings of the 29th ACM/IEEE International Symposium on Low Power Electronics and Design*. 1–6.

[61] Sheng Jiang and Ming Liu. 2025. Building an Elastic Block Storage over EBOFs Using Shadow Views. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*. 1137–1153.

[62] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA, 345–360.

[63] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. 2021. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 605–620.

[64] Yuyuan Kang and Ming Liu. 2025. Understanding and Profiling NVMe-over-TCP Using ntprof. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*. 1117–1136.

[65] Keysight. 2024. What is a Chiplet, and Why Should You Care? https://www.keysight.com/blogs/en/tech/sim-des/2024/2/8/what-is-a-chiplet-and-why-should-you-care.

[66] HT Kung, Trevor Blackwell, and Alan Chapman. 1994. Credit-based flow control for ATM networks: Credit update protocol, adaptive credit allocation and statistical multiplexing. In *Proceedings of the conference on Communications architectures, protocols and applications*. 101–114.

[67] HT Kung and Koling Chang. 1995. Receiver-oriented adaptive buffer allocation in credit-based flow control for ATM networks. In *Proceedings of INFOCOM'95*, Vol. 1. 239–252.

[68] NT Kung and Robert Morris. 1995. Credit-based flow control for ATM networks. *IEEE network* 9, 2 (1995), 40–48.

[69] Julia Lawall, Himadri Chhaya-Shailesh, Jean-Pierre Lozi, Baptiste Lepers, Willy Zwaenepoel, and Gilles Muller. 2022. Os scheduling with nest: Keeping tasks close together on warm cores. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 368–383.

[70] Xiao Li, Zerui Guo, Yuebin Bai, Mehash Ketkar, Hugh Willkinson, and Ming Liu. 2025. Understanding and Profiling CXL.mem Using PathFinder. In *Proceedings of the ACM SIGCOMM 2025 Conference (SIGCOMM'25)*.

[71] Ming Liu. 2020. *Building Distributed Systems Using Programmable Networks*. University of Washington.

[72] Ming Liu. 2023. Fabric-Centric Computing. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS'23)*. 118–126.

[73] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading distributed applications onto smartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'19)*. 318–333.

[74] Ming Liu, Arvind Krishnamurthy, Harsha V. Madhyastha, Rishi Bhardwaj, Karan Gupta, Chinmay Kamat, Huapeng Yuan, Aditya Jaltade, Roger Liao, Pavan Konka, and Anoop Jawahar. 2020. Fine-Grained Replicated State Machines for a Cluster Storage System . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. 305–323.

[75] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. 795–809.

[76] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*. 363–378.

[77] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*. Article 1, 16 pages.

[78] Liang Luo, Ming Liu, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. 2017. Motivating in-network aggregation for distributed deep neural network training. In *Workshop on Approximate Computing Across the Stack*.

[79] Xiaohan Ma, Ying Wang, Yujie Wang, Xuyi Cai, and Yinhe Han. 2022. Survey on chiplets: interface, interconnect and integration methodology. *CCF Transactions on High Performance Computing* 4, 1 (2022), 43–52.

[80] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. 2022. Efficient Scheduling Policies for Microsecond-Scale Tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1–18. https://www.usenix.org/conference/nsdi22/presentation/mcclure

[81] Richard McDougall and Jim Mauro. 2005. FileBench. *URL: http://www.nfsv4bat.org/Documents/nasconf/2004/filebench.pdf* (2005).

[82] Samantha Miller, Anirudh Kumar, Tanay Vakharia, Ang Chen, Danyang Zhuo, and Thomas Anderson. 2024. Enoki: High velocity linux kernel scheduler development. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 962–980.

[83] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. 2016. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 71–85.

[84] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM'21)*. 106–122.

[85] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. 2023. eZNS: An Elastic Zoned Namespace for Commodity ZNS SSDs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*. 461–477.

[86] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. 2024. eZNS: Elastic Zoned Namespace for Enhanced Performance Isolation and Device Utilization. *ACM Trans. Storage* 20, 3, Article 16 (June 2024), 41 pages.

[87] Ashley O Munch, Nevine Nassif, Carleton L Molnar, Jason Crop, Rich Gammack, Chinmay P Joshi, Goran Zelic, Kambiz Munshi, Min Huang, Charles R Morganti, et al. 2024. 2.3 emerald rapids: 5th-generation intel® xeon® scalable processors. In *2024 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 67. IEEE, 40–42.

[88] Benjamin Munger, Kathy Wilcox, Jeshuah Sniderman, Chuck Tung, Brett Johnson, Russell Schreiber, Carson Henrion, Kevin Gillespie, Tom Burd, Harry Fair, et al. 2023. "Zen 4": The AMD 5nm 5.7 GHz x86-64 microprocessor core. In *2023 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 38–39.

[89] Nevine Nassif, Ashley O Munch, Carleton L Molnar, Gerald Pasdast, Sitaraman V Lyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, et al. 2022. Sapphire rapids: The next-generation intel xeon scalable processor. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. IEEE, 44–46.

[90] The NextPlatform. 2023. AWS ADOPTS ARM V2 CORES FOR EXPANSIVE GRAVITON4 SERVER CPU. https://www.nextplatform.com/2023/11/28/aws-adopts-arm-v2-cores-for-expansive-graviton4-server-cpu/.

[91] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 361–378.

[92] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 663–679.

[93] The Open Compute Project. 2021. OpenHBI Specification Version 1.0. https://www.opencompute.org/documents/odsa-openhbi-v1-0-spec-rc-final-1-pdf.

[94] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 145–160. https://www.usenix.org/conference/osdi18/presentation/qin

[95] Yiming Qiu, Qiao Kang, Ming Liu, and Ang Chen. 2020. Clara: Performance Clarity for SmartNIC Offloading. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets'20)*. 16–22.

[96] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. 2021. Automated SmartNIC Offloading Insights for Network Functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*. 772–787.

[97] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. 2014. {SENIC}: Scalable {NIC} for {End-Host} rate limiting. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. 475–488.

[98] Scott Rixner. 2004. Memory controller optimizations for web servers. In *37th International Symposium on Microarchitecture (MICRO-37'04)*. 355–366.

[99] Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. 2000. Memory access scheduling. *ACM SIGARCH Computer Architecture News* 28, 2 (2000), 128–138.

[100] Junyeol Ryu, Ming Liu, and Matthew D. Sinclair. 2026. Understanding and Profiling the Accelerator Chiplet Network Using PingPoint. In *Proceedings of the ACM SIGCOMM 2026 Conference (SIGCOMM'26)*. 696–713.

[101] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*. 740–755.

[102] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating Fair Queueing on Reconfigurable Switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*. 1–16.

[103] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. 2020. Programmable Calendar Queues for High-speed Packet Scheduling . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. 685–699.

[104] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. 2009. HASS: A scheduler for heterogeneous multicore systems. *ACM SIGOPS Operating Systems Review* 43, 2 (2009), 66–75.

[105] Teja Singh, Spence Oliver, Sundar Rangarajan, Shane Southard, Carson Henrion, Alex Schaefer, Brett Johnson, Sarah Bartaszewicz Tower, Kathy Hoover, Deepesh John, et al. 2025. "Zen 5": The AMD High-Performance 4nm x86-64 Microprocessor Core. In *2025 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 68. IEEE, 1–3.

[106] Teja Singh, Sundar Rangarajan, Deepesh John, Carson Henrion, Shane Southard, Hugh McIntyre, Amy Novak, Stephen Kosonocky, Ravi Jotwani, Alex Schaefer, et al. 2017. 3.2 Zen: A next-generation high-performance× 86 core. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 52–53.

[107] Teja Singh, Sundar Rangarajan, Deepesh John, Russell Schreiber, Spence Oliver, Rajit Seahra, and Alex Schaefer. 2020. 2.1 Zen 2: The amd 7nm energy-efficient high-performance x86-64 microprocessor core. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 42–44.

[108] Ion Stoica and Hussein Abdel-Wahab. 1995. Earliest eligible virtual deadline first: A flexible and accurate mechanism for proportional share resource allocation. *Old Dominion Univ., Norfolk, VA, Tech. Rep. TR-95-22* (1995).

[109] Synopsys. 2025. What are Chiplets. https://www.synopsys.com/glossary/what-are-chiplets.html.

[110] Ebadollah Taheri, Sudeep Pasricha, and Mahdi Nikdast. 2024. ReD: A reliable and deadlock-free routing for 2.5-D chiplet-based interposer networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 12 (2024), 4599–4612.

[111] Ammar Tahir, Prateesh Goyal, Ilias Marinos, Mike Evans, and Radhika Mittal. 2024. Efficient policy-rich rate enforcement with phantom queues. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 1000–1013.

[112] Eric Taillard. 1993. Benchmarks for basic scheduling problems. *european journal of operational research* 64, 2 (1993), 278–285.

[113] Microchip USA. 2023. What is a Chiplet? https://www.microchipusa.com/industry-news/what-is-a-chiplet.

[114] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. 2003. Capriccio: scalable threads for internet services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. 268–281.

[115] Midhul Vuppalapati, Saksham Agarwal, Henry Schuh, Baris Kasikci, Arvind Krishnamurthy, and Rachit Agarwal. 2024. Understanding the Host Network. In *Proceedings of the ACM SIGCOMM Conference*. 581–594.

[116] Chendong Wang, Joontaek Oh, and Ming Liu. 2026. Co-Designing Traffic Control with NVMe-oF for Disaggregated Storage: A Comparative Study of Switched and Switchless SAN Architectures. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'26)*. 2651–2669.

[117] Wikipedia. 2025. 2.5D integrated circuit. https://en.wikipedia.org/wiki/2.5D_integrated_circuit.

[118] Wikipedia. 2025. Advanced packaging (semiconductors). https://en.wikipedia.org/wiki/Advanced_packaging_(semiconductors).

[119] Xincheng Xie, Wentao Hou, Zerui Guo, and Ming Liu. 2025. Building Massive MIMO Baseband Processing on a Single-Node Supercomputer. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*. 1221–1242.

[120] Jieming Yin, Zhifeng Lin, Onur Kayiran, Matthew Poremba, Muhammad Shoaib Bin Altaf, Natalie Enright Jerger, and Gabriel H Loh. 2018. Modular routing design for chiplet-based systems. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 726–738.

[121] Hua Zhang, Xiao Li, Yuebin Bai, and Ming Liu. 2026. Understanding and Optimizing Database Pushdown on Disaggregated Storage. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2141–2158.

[122] Chenxingyu Zhao, Tapan Chugh, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. 2022. Dremel: Adaptive Configuration Tuning of RocksDB KV-Store. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 2, Article 37 (June 2022), 30 pages.

[123] Chenxingyu Zhao, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. 2025. White-Boxing RDMA with Packet-Granular Software Control. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*. 427–449.

[124] Chenxingyu Zhao, Yibo Wu, Hongtao Zhang, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. 2026. Efficient and Flexible Datapaths for Fine-Grained Rack-Scale Interconnects with Elastic QP. In *Proceedings of the ACM SIGCOMM 2026 Conference (SIGCOMM'26)*. 657–675.

[125] Chenxingyu Zhao, Hongtao Zhang, Jaehong Min, Shengkai Lin, Wei Zhang, Kaiyuan Zhang, Ming Liu, and Arvind Krishnamurthy. 2026. SG-IOV: Socket-Granular I/O Virtualization for SmartNIC-Based Container Networks. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1727–1748.