μManycore: A Cloud-Native CPU for Tail at Scale

Jovan Stojkovic
University of Illinois at Urbana-Champaign
jovans2@illinois.edu

Chunao Liu
Purdue University
liu2849@purdue.edu

Muhammad Shahbaz
Purdue University
mshahbaz@purdue.edu

Josep Torrellas
University of Illinois at Urbana-Champaign
torrella@illinois.edu

ABSTRACT

Microservices are emerging as a popular cloud-computing paradigm. Microservice environments execute typically-short service requests that interact with one another via remote procedure calls (often across machines), and are subject to stringent tail-latency constraints. In contrast, current processors are designed for traditional monolithic applications. They support global hardware cache coherence, provide large caches, incorporate microarchitecture for long-running, predictable applications (such as advanced prefetching), and are optimized to minimize average latency rather than tail latency.

To address this imbalance, this paper proposes *μManycore*, an architecture optimized for cloud-native microservice environments. Based on a characterization of microservice applications, *μManycore* is designed to minimize unnecessary microarchitecture and mitigate overheads to reduce tail latency. Indeed, rather than supporting manycore-wide hardware cache coherence, *μManycore* has multiple small hardware cache-coherent domains, called *Villages*. Clusters of villages are interconnected with an on-package leaf-spine network, which has many redundant, low-hop-count paths between clusters. To minimize latency overheads, *μManycore* schedules and queues service requests in hardware, and includes hardware support to save and restore process state when doing a context-switch. Our simulation-based results show that *μManycore* delivers high performance. A cluster of 10 servers with a 1024-core *μManycore* in each server delivers 3.7× lower average latency, 15.5× higher throughput, and, importantly, 10.4× lower tail latency than a cluster with iso-power conventional server-class multicores. Similar good results are attained compared to a cluster with power-hungry iso-area conventional server-class multicores.

CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**; *Cloud computing*;

KEYWORDS

Microservices; cloud computing; manycore architecture

ISCA '23, June 17–21, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589068

ACM Reference Format:
Jovan Stojkovic, Chunao Liu, Muhammad Shahbaz, and Josep Torrellas. 2023. μManycore: A Cloud-Native CPU for Tail at Scale. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3579371.3589068

1 INTRODUCTION

Cloud computing is undergoing a paradigm shift, as large monolithic applications are being replaced by compositions of many lightweight, loosely-coupled microservices [64]. In these "cloud-native" workloads, each microservice is implemented and deployed as a separate program, and executes a portion of the application's logic, such as HTTP connection termination, key-value serving [75], protocol routing [93], or ad serving [29]. This composable application design simplifies development and enables programming language and framework heterogeneity. Additionally, each microservice can be shared among multiple applications, while being scaled and updated independently of other microservices. This new paradigm is being embraced by all the major cloud providers, such as Amazon, Netflix, Alibaba, Twitter, Uber, Facebook, and Google [10, 14, 28, 50, 51, 59, 72, 73, 81, 82, 84, 94]. In addition, there is a proliferation of open-source environments that manage microservice workloads (e.g., Kubernetes [42] and Docker Compose [17]).

Microservice environments have new characteristics that impact the system and hardware architecture of the platforms on which they run. Specifically, requests for microservices in an application are typically short-running and may execute on different machines. Requests for different microservices share no memory state and interact with one another via remote procedure calls (RPCs) [27, 38]. Further, requests have small working sets and are often invoked in bursts, frequently waiting in queues before being executed. Finally, the decomposition of an application places tight sub-ms latency Service Level Objectives (SLOs) on individual services [74, 75]. As a result, while reducing average latency and improving throughput are important, the key performance target in these environments is now *minimizing tail latency* [16] (e.g., improving the 99th-percentile responses). Many of these characteristics are also found in emerging deployment methods based on microservices, such as Function-as-a-Service (FaaS) environments [2, 25, 33, 54].

Current processors are not expressly designed for these environments. Indeed, multicores invest significant hardware and design complexity to support global hardware cache coherence. They have large caches to capture the working sets of long-running applications. They are relatively unconcerned with supporting short-running, RPC-communicating programs. Instead, they incorporate

microarchitectural optimizations for long-running, predictable applications, such as advanced prefetchers and branch predictors. These optimizations add significant hardware complexity and are, at best, marginally effective for microservices. Perhaps most importantly, current processors are highly optimized to minimize the average latency of programs or transactions, and ignore tail-latency considerations.

How should one change the design of processors so that they match microservice requirements? First, some of the hardware optimizations that introduce design complexity and are hardly needed by microservices, such as global hardware cache coherence, should be reconsidered. Second, there should be a comprehensive effort to optimize for tail-latency reduction. Optimizations should target both inefficiencies affecting all requests, and contention-based overheads that may affect a subset of requests. While the resulting processor will not be competitive for general-purpose loads, it can be the CPU of choice for microservice-heavy datacenters.

In this paper, we propose a processor architecture highly optimized for cloud-native microservice workloads. We call it *µManycore*. *µManycore* is not an accelerator; it retains general-purpose processor capabilities, although it may not be as competitive for monolithic applications.

To design *µManycore*, we start by characterizing production-level microservice traces from Alibaba [50] and microservice applications from DeathStarBench [23]. Our analysis shows that bursty service requests create periods of high demand where long waiting queues are likely to appear. In addition, requests spend most of their time blocked, waiting for the completion of their accesses to storage or their calls to other services. In the meantime, CPUs context switch frequently, introducing overhead. Moreover, service-initiated messages between cores experience the latency of interconnection networks (ICNs), often suffering contention delays that further increase tail latency. Finally, while requests have small working sets, microservices benefit from a large nearby pool of memory that stores per-microservice read-mostly state.

Based on these findings, we design a chiplet-based *µManycore*. Rather than supporting package-wide hardware cache-coherence, *µManycore* is built with multiple small hardware cache-coherent domains called *Villages*. Microservices are assigned to individual villages. A few villages, together with a memory chiplet (storing read-mostly state), are grouped in a cluster. Clusters are interconnected with a leaf-spine ICN [12, 20]. This topology has many redundant, low-hop-count paths between any two clusters—hence, minimizing contention between multiple messages with the same source and destination clusters and reducing tail latency. To minimize scheduling overheads, *µManycore* enqueues, dequeues, and schedules service requests in hardware. Finally, to minimize the overhead of frequent context switching, cores include hardware support to save and restore process state.

Our simulation-based results show that *µManycore* delivers high performance for microservice workloads. We compare a 1024-core *µManycore* to two conventional server-class multicores: one with the same power and one with the same area as *µManycore*. A cluster of 10 servers with *µManycores* delivers 3.7× lower average latency, 15.5× higher throughput, and, importantly, 10.4× lower tail latency than a cluster with the iso-power conventional multicores. Similar good results are attained compared to a cluster with the power-hungry iso-area conventional multicores. Finally, on-package leaf-spine ICN, request scheduling in hardware, and hardware context switching are highly effective in improving the performance of microservice workloads.

This paper's contributions are as follows:

- A characterization of microservice workload behavior in conventional processors.
- *µManycore*, a processor architecture that is highly optimized for microservice workloads.
- An evaluation of *µManycore*, comparing it to two conventional server-class multicores: one with the same power and one with the same area.

2 BACKGROUND AND MOTIVATION

2.1 Microservice Environments

In microservice environments (e.g., managed by Kubernetes [42] or Docker Compose [17]), large complex applications are organized as workflows of multiple interdependent services. Each service executes a separate functionality, serves requests of its type, and is deployed as a separate instance. Service requests often perform reads and writes to remote storage, which are costly and may stall program execution for a significant time.

Often, a service request invokes one or more other services that perform simple operations and then aggregates the obtained data. Studies by Alibaba [50] and Facebook [73] show that such a multi-tier paradigm is popular in production-level microservice architectures. Services communicate with each other via RPC/HTTP protocols, such as gRPC [27] and eRPC [38]. When a service request calls another service synchronously, it waits on the results before continuing with its own execution. This operation also introduces potentially significant stall times.

Individual microservices are significantly simpler than their monolithic counterparts. They have a smaller memory footprint and working set, less pressure on instruction fetching, and orders of magnitude shorter execution time. However, in reality, these environments have substantial performance challenges. Short execution times and frequent, costly remote storage accesses and communication between services induce overheads that cannot be overlooked [11, 15, 32, 36, 55, 56, 61, 62, 74, 75, 77].

2.2 The Need for a Cloud-Native CPU

The ever-increasing complexity of software systems has kept pushing forward processor design. For example, researchers have proposed numerous prefetching, branch prediction, and cache replacement schemes. These proposals introduce custom microarchitectural structures that increase processor area, power consumption, and design complexity in order to improve application performance.

However, many of these optimizations hardly benefit cloud-native microservice workloads. To validate this hypothesis, we consider four published microarchitectural optimizations for which the simulator and applications used in the publications are open sourced. For each of the optimizations, we first run the original applications [13, 60, 78, 79, 88] on the original simulator and record the performance with and without the optimizations. The results are depicted as bars *Baseline* and *Optimized* in *Mono* (for Monolithic)

in Figure 1, normalized to *Baseline*. We then run a set of microservice applications—SocialNetwork from DeathStarBench [23], and Router and SetAlgebra from μSuite [74]—on the original simulator and record the performance with and without the optimizations. The results are depicted as bars *Baseline* and *Optimized* in *Micro* (for Microservice) in Figure 1, normalized to *Baseline*.

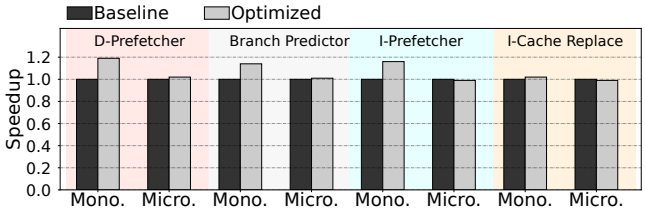


Figure 1: Performance improvements of four recently-proposed microarchitectural optimizations using monolithic (*Mono*) and microservice (*Micro*) applications. For each optimization and application set, the bars are normalized to *Baseline*.

The optimizations are as follows:

D-Prefetcher shows the impact of the Pythia reinforcement-learning data prefetcher [8]. Pythia speeds-up monolithic applications by 19% on average over a system without a prefetcher. However, it brings only marginal benefits of 2% to microservices.

Branch Predictor shows the impact of a perceptron-based branch predictor [35]. The predictor speeds-up monolithic applications by 14% on average over a system with a simple g-share predictor. On the other hand, the predictor speeds-up microservice applications by only 1% on average over the g-share predictor.

I-Prefetcher shows the impact of the I-SPY context-driven instruction prefetcher [40]. The prefetcher speeds-up monolithic applications by 16% on average over a system without instruction prefetcher. On the other hand, it does not speed-up microservice applications.

I-Cache Replace shows the impact of the Ripple profile-guided instruction cache replacement algorithm [41]. The algorithm speeds-up monolithic applications by 2% on average over a system with LRU replacement. However, it does not bring any benefits to microservices.

The reason for the discrepancy in the effectiveness of the proposed optimizations is the reduced data and instruction memory footprint of the microservice workloads compared to the monoliths, as well as their increased cache hit rates and different branch behavior. This data shows that a different type of processor microarchitecture is needed to speed-up microservice applications.

3 CHARACTERIZING MICROSERVICE APPLICATIONS ON CURRENT PROCESSORS

To guide the design of *μManycore*, we first characterize the behavior of microservice applications on current processors. We execute the DeathStarBench [23], TrainTicket [96], and μSuite [74] open-source microservice application suites, as well as real-world production-level microservice execution traces from Alibaba [50]. Our main conclusions are described next.

3.1 Monolithic Cache Coherence Provides Limited Advantage

To enable high availability, fast scalability, and fault tolerance, microservice applications are implemented as sets of services. Each service is built as a standalone RPC/HTTP server—in our workloads, a TThreadedServer [80], RestController [71], HTTPServer [24], or gRPCServer [27]. Upon service instance initialization, network connections are set up, libraries are loaded, and preparation code is executed. Upon service request arrival, the service instance spawns a new worker (a process, thread, or co-routine) or reuses an existing one to serve the request.

Different services or different instances of the same service do not share any modifiable memory. A worker can update its private state, the local state of its service instance and, with RPC calls, the state in global storage. This is in contrast to traditional multi-threaded applications, where concurrently-running threads are often free to share memory.

Given this environment, conventional monolithic hardware cache coherence, as it is used in current large multicores, is hard to justify. Cache coherence is only needed inside a service instance, which typically uses only a few cores. One could argue that global coherence would still be needed if we allowed service instances to migrate across any cores. However, unimpeded migration of service instances across a large 1K-core multicore is unlikely to deliver performance improvements and, in fact, is likely to increase tail latency. Hence, given the well-known hardware complexity and scalability challenges of large-scale hardware cache-coherence, it is more reasonable to support only small-scale cache-coherence domains among the cores used by individual service instances. Service requests for a given instance can still migrate between the cores used by the service instance if needed for load balance—resulting in a more efficient environment.

3.2 Bursty Requests Increase Tail Latency

We use Alibaba's production-level traces [50] to characterize the arrival rate of service requests. The traces include requests directed to 10,000 servers. In each server, service requests arrive in bursts, creating periods of high and low request demands. Figure 2 shows the CDF of the number of Requests per Second (RPS) arriving at a server [85, 92]. We can see that a server that gets and processes a median of ≈500 RPS, sometimes gets multiple times these many RPS—i.e., 20% of the time, it receives 1,000 RPS or more, and in 5% of the time, it receives 1,500 RPS or more. When these large numbers of requests are received, they have to wait in queues.

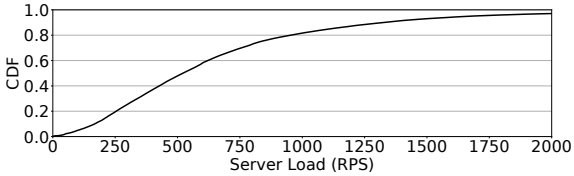


Figure 2: CDF of Requests per Second (RPS) received by a server.

Given this environment, it is important to design queuing systems that have minimal overhead. Previous proposals have considered a fully-centralized First-Come-First-Serve (FCFS) queue [36,

61]. However, under high concurrency, such an approach induces high synchronization overheads. At the other extreme, one can have fully-decentralized FCFS queuing, with a per-core queue. This approach is equally undesirable, as it leads to load imbalance and head-of-line blocking.

Any suboptimal queuing structure will lead to increased average response time for the service requests. Most importantly, it will have a major impact on the *tail* response time.

To see this effect, we take the DeathStarBench applications [23] and run them on a simulated 1024-core ScaleOut manycore (described in Section 5). We issue requests using a Poisson distribution with 50K RPS, on average. Figure 3 shows the average and tail response time of the requests as we vary the number of queues in the manycore. The leftmost point (1024) means that each core has a dedicated queue, and the next one (512) that every two cores share one queue, and so on. In the rightmost point, all cores share a single queue. Requests are assigned to queues randomly. In addition, we evaluate a system that allows a core to steal requests from other queues when its assigned queue is empty.

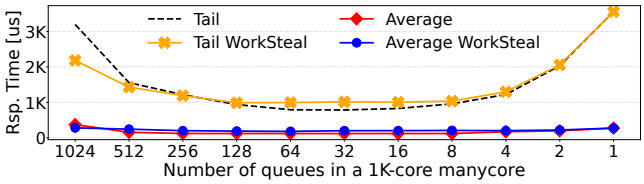


Figure 3: Average and tail response time of requests for different numbers of queues in a 1024-core manycore.

The figure shows that the average response time increases modestly as we go from the best scenario (32 queues with 32 cores per queue) to the worst ones (1024 queues or one queue). However, the tail response time changes dramatically. With 1024 queues and with one queue, the tail is 4.1× and 4.5× higher, respectively, than the tail with 32 queues.

Work stealing significantly reduces the tail when the system has one queue per core. This is because it mitigates load imbalance. However, as we increase the number of cores per queue, and thus reduce the load imbalance, work-stealing becomes less useful and even increases the tail due to the added overheads. Work stealing does not change the average latency.

3.3 Context Switching Hurts Tail Latency

We now use Alibaba's traces to characterize the execution of service requests. We find that requests are typically very short: 36.7% of the dynamic invocations take less than 1ms; the geometric mean duration of the remaining dynamic invocations is 2.8ms. In addition, service requests spend most of that time waiting (i.e., blocked) on I/O. Figure 4 shows the CDF of CPU utilization per dynamic request. The median CPU utilization is only ≈14%. Further, 99% of the requests utilize the CPU less than 60%. The reason for the low utilization is the frequent execution stalls due to RPC invocations: the request execution is blocked waiting for the completion of storage requests or calls to other services. Figure 5 shows the CDF of the number of RPC invocations per dynamic request. A request performs a median of ≈4.2 RPC invocations. Moreover, about 5% of the requests invoke 16 or more RPCs.

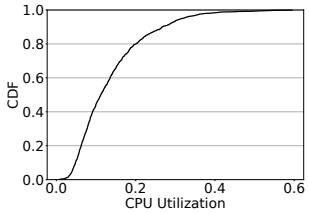


Figure 4: CDF of CPU utilization per request.

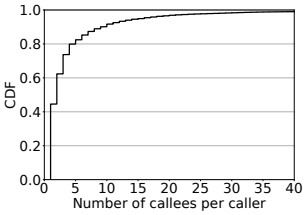


Figure 5: CDF of number of RPC invocations per request.

As another data point, in the DeathStarBench applications [23], the average execution time of a service request is 120μs, and the average request performs 3.1 RPC invocations.

To use resources efficiently in microservice environments, CPUs need to context switch every time a request blocks on an RPC invocation. The cost of a context switch is ≈5K cycles in Linux-based systems and ≈2K cycles in state-of-the-art software schedulers [36]. This overhead may be negligible for monolithic applications, where the time between context switches is much larger than the context switch overhead. However, this is not true for microservices.

To assess the impact of context-switch overhead on the request tail latency, we simulate the execution of the 1024-core ScaleOut manycore invoking the services of the SocialNetwork application from DeathStarBench with a Poisson distribution with 5K, 10K, and 50K RPS. We add a certain amount of Context Switch overhead cycles (*CS*) every time they suffer a context switch. We vary *CS* from zero to 8K cycles. Figure 6 shows the tail latency of the requests. The tail latency is normalized to the one with zero *CS* cycles. The figure shows the range of *CS* cycles that are typical for Linux, and for the state-of-the-art Shenango, Shinjuku, and ZygOS software schedulers [36].

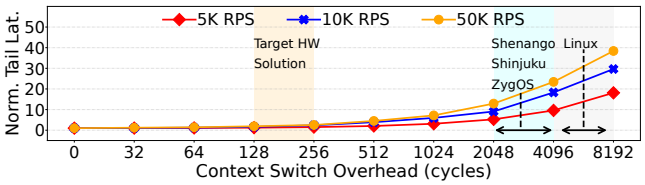


Figure 6: Impact of the context switch overhead on the tail latency with different requests per second (RPS).

These context-switch overhead cycles delay request processing. We see that the impact is significant, especially for larger loads. For 50K RPS, the context-switch overhead of Linux degrades the tail latency of requests by 26–38×; the context switch overhead of state-of-the-art software schedulers degrades it by 13–23×. Ideally, we would like a *CS* of around 128–256 cycles which, as shown in the figure, barely impacts the tail latency. Such *CS* requires hardware support.

3.4 The Interconnect Impacts Tail Latency

In microservice environments, request execution triggers interconnection network (ICN) messages, as it issues storage requests and calls to other services. Such messages compete for ICN links, and potentially suffer contention delays. Such delays directly impact

the tail latency of requests. Consequently, the design and implementation of the ICN play a significant role in determining the tail latency. In this paper, we are interested in the on-package ICN.

To assess this effect, we take the DeathStarBench applications and run them on the simulated 1024-core ScaleOut manycore, issuing Poisson-distributed requests with 1K, 5K, 10K, and 50K RPS. Cores are grouped in 32-core clusters, and the clusters are interconnected with either a 2D mesh or a fat-tree ICN. The contention-free hop-to-hop latency of the ICN is 5 cycles. Service requests are issued to cores randomly. Figure 7 shows the resulting request tail latency. Each bar is normalized to the tail latency of the same environment without ICN contention.

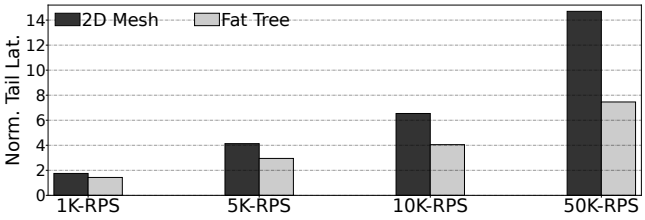


Figure 7: Impact of contention in the on-package interconnection network (ICN) on the tail latency of requests. Each bar is normalized to the tail latency without ICN contention.

The figure shows that contention in the ICN has a substantial impact on tail latency. With 50K RPS, contention in the 2D mesh ICN increases the tail latency by 14.7× on average. For the fat-tree ICN, the increase is 7.5× on average. Therefore, the ICN should be carefully designed to minimize contention.

3.5 Large Read-Mostly Memories of Service Instances & Small Working Sets of Requests

When a service instance is created, it initializes its state, which includes its container, runtime, and libraries. To save initialization overhead, microservice systems may store *Snapshots* of services in memory with all the initialization state. This is especially important in FaaS, where containers are created much more frequently [1, 18, 26]. Then, when a new instance is created, all that it needs to do to initialize is to simply read its corresponding snapshot. Hence, for performance reasons, it is important to keep snapshots in a *near read-mostly memory*. For DeathStarBench applications, snapshots reduce the boot time of a service instance from over 300ms to less than 10ms, while using less than 16MB of memory per service [18].

In addition, every time a request is received for a service, the service instance spawns a new handler. All handlers of a service instance read some of the instance's initialization data. Moreover, as the handlers execute the same code, they read mostly the same instructions. As a result, different handlers of the same service instance have very similar instructions and read-data footprints.

A handler's memory footprint is small. On average for the DeathStarBench applications, it is only 0.5 MB. Figure 8 considers the memory footprint of a handler (normalized to 1). In the *Handler-Handler* bars, it shows what fraction of the footprint is *common* (and hence can be read-shared) with another handler of the same service instance. In the *Handler-Init* bars, the figure shows what fraction of the handler footprint is common (and hence can be read-shared) with the initialization process of the service instance. In each of the two groups, from left to right, the normalized bars show the data footprint in pages, the data footprint in cache lines, the instruction footprint in pages, and the instruction footprint in cache lines. Pages are 4KB and cache lines are 64B. All bars are averaged across all the DeathStarBench applications.

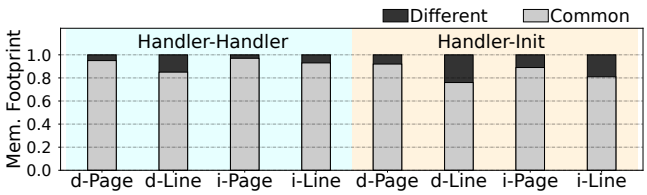


Figure 8: Handler-handler and handler-initialization sharing of data and instruction pages and cache lines.

The figure shows that, on average, the fraction of pages or cache lines that are common between two handlers or between a handler and its initialization process is 78–99%. Consequently, a manycore architecture for microservices can benefit from having read-shared memories that are accessed by multiple requests of the same service instance.

Because of the small footprint of handlers, requests put little pressure on the cache hierarchy. This is in contrast to monolithic applications, which require ever bigger caches [40, 41, 70]. Figure 9 shows the average hit rates of L1 and L2 TLBs and caches, both for data and instructions, for the architecture in Table 2, which will be discussed later. We observe that, for the L1 TLB and cache, the hit rates of both data and instructions are above 95%. Hence, the working sets fit in L1 TLB and cache. The L2 TLB and cache have lower hit rates; this is because the L1 structures act as filters, intercepting the high-locality accesses. As a result, a manycore architecture for microservices can use small caches and reduced-depth cache hierarchies (e.g., hierarchies of only two levels of caching). The resources saved can be invested in supporting more parallelism.

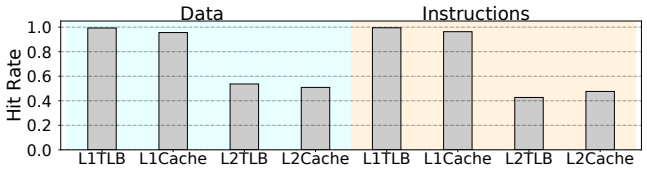


Figure 9: L1 and L2 data/instructions TLB and cache hit rates.

4 *μMANYCORE*: A CLOUD-NATIVE CPU

This paper proposes *μManycore*, a processor designed for microservices. In microservice environments, a key objective is to minimize the tail latency of requests. Hence, *μManycore* is designed to minimize the primary overheads that contribute to the tail latency. Some of these overheads impact both tail and average latency—i.e., overheads that, to a large extent, affect all service requests. Other overheads impact mainly tail latency—i.e., overheards that disproportionately impact some requests, such as overheads resulting from contention effects. *μManycore* addresses both types of overheads.

The characterization of Section 3 gives insights into the main sources of tail latency in microservice environments. Table 1 lists such sources, the reason why they exist, and how the *μManycore*

Table 1: Main sources of tail latency.

Source	Reason	*µManycore* Solution
Monolithic cache coherence	Remote directory/cache/network accesses (some due to migration) and contention	Multiple small cache coherent domains
Request scheduling	Synchronization and queuing of requests	Request enqueuing, dequeuing, and scheduling in hardware
Context switching	OS invocation and saving & restoring state	Hardware-based context switching
On-package network	Network link/router latency (some due to contention)	On-package hierarchical leaf-spine network

design avoids them. In the following, we consider each of these sources in turn. We assume a large manycore with 1024 cores.

1. Monolithic Cache Coherence. As indicated in Section 3.1, requests for different service instances do not share memory state. They communicate through remote storage accesses and through service calls, both of which use RPCs. Hence, they do not require monolithic cache coherence. Providing monolithic cache coherence in a manycore typically results in remote directory and network accesses, which increase tail latency. The only reason to provide monolithic cache coherence would be to support service instance migration across cores for load balance. However, unrestricted instance migration across a large manycore results in (1) remote cache accesses to obtain data from caches in cores where the instance used to run, (2) more remote directory accesses, (3) additional network traffic, and (4) increased contention. The result would be increased tail latency.

In practice, there are some reasons to support modest-size cache coherence domains. First, some services are multithreaded. Second, allowing requests for a given service instance to migrate between the cores used by the instance can improve load balance. Finally, supporting some hardware cache coherent domain ensures that the manycore remains general purpose. Consequently, in *µManycore*, we eliminate monolithic hardware cache coherence and, instead, have multiple small hardware cache-coherent domains. These domains are called *Villages*. Each service instance is assigned to a village. A service request is allowed to migrate between the cores of its village for load balance, and to execute in parallel on the cores of its village for speed.

Message-passing designs such as Intel's Single Chip Cloud [30, 83] and Sony's Cell Processor [21] completely abandon hardware cache coherence. Such designs could also be used to run microservice environments. However, they are suboptimal. Beyond not supporting multithreaded services efficiently, they also fail to efficiently handle request migration in the presence of frequent context switches. Specifically, recall that a request is frequently blocked on I/O. When the request gets restarted, to better utilize CPUs, the system may want to run it on another core. Unless there is cache coherence support, the state left by the request in the caches before blocking will not be automatically reused after restarting.

Section 3.5 showed that handlers executing requests for the same service instance share substantial read-only data and instruction state. *µManycore* takes advantage of this fact, as it maps requests for the same service instance to the same village. Their handlers read the same cache state, thereby improving overall performance.

When a village fills to capacity, the system may need to allocate a new instance of the same service in another village. Such new instance will be initialized faster if it can read a *Snapshot* of the service (Section 3.5). A snapshot takes 10s of MBs. Consequently, *µManycore* provides a large *Memory Pool* of fast mostly-read SRAM next to the villages to keep snapshots. Service instances in nearby villages can access the memory pool.

2. Request Scheduling. As indicated in Section 3.2, requests come in bursts, potentially creating queues of requests to be processed. Given that request execution granularity is often in the scale of microseconds, the overheads of request queuing and scheduling are noticeable.

To provide efficient request handling, *µManycore* supports request enqueuing, dequeuing, and scheduling in hardware. Each village has its own hardware queue for requests to local service instances. When a request external to the *µManycore* package arrives at the *µManycore*'s top-level NIC or a request is generated internally in the *µManycore* package, the request is routed in hardware to the village that runs the corresponding service instance and enqueued in a queue. Then, a local core dequeues it. Both enqueuing and dequeuing are performed in hardware, without any OS or other software involvement.

3. Context Switching. A request spends most of its execution time blocked on I/O, waiting on remote storage accesses or calls to other services (Section 3.3). Cores avoid stall time by frequently switching between requests. However, each context switch involves thousands of cycles, directly degrading the tail latency.

To address this problem, *µManycore* has hardware support for context switching. A core saves and restores state in a context switch without any OS or software intervention.

4. On-package Interconnection Network (ICN). Messages between different villages and memory pools traverse ICN links and routers. Network traversal can take substantial time, especially if compounded by contention effects. The resulting latency directly affects the tail latency.

To minimize this latency, *µManycore* uses an on-package *Leaf-Spine* ICN topology [12, 20] (Figure 12), which has many redundant, low-hop-count paths between any given source and destination villages. Messages are less likely to suffer contention than in other networks. Even multiple messages with the same source and destination villages can proceed in parallel without delaying one another.

In the following, we describe these four main components of *µManycore* in detail.

4.1 *µManycore* Organization

Villages and Clusters. The basic unit of a *µManycore* is a hardware cache-coherent village. A village contains a set of cores (e.g., 8-16) with private caches and a shared L2, a *Request Queue* module that will be described later, and two I/O ports. Since the working set of service requests is small (Section 3.5), there is no need for a deeper cache hierarchy.

The combination of a few villages (e.g., 4), a memory pool, and a network hub forms a cluster. Figure 10 shows a cluster. We envision the combined villages, the memory pool, and the network hub to be implemented as three different chiplets. Finally, a *µManycore* package is composed of many clusters (Figure 11 shows two of them) interconnected with a hierarchical leaf-spine ICN (Figure 12).

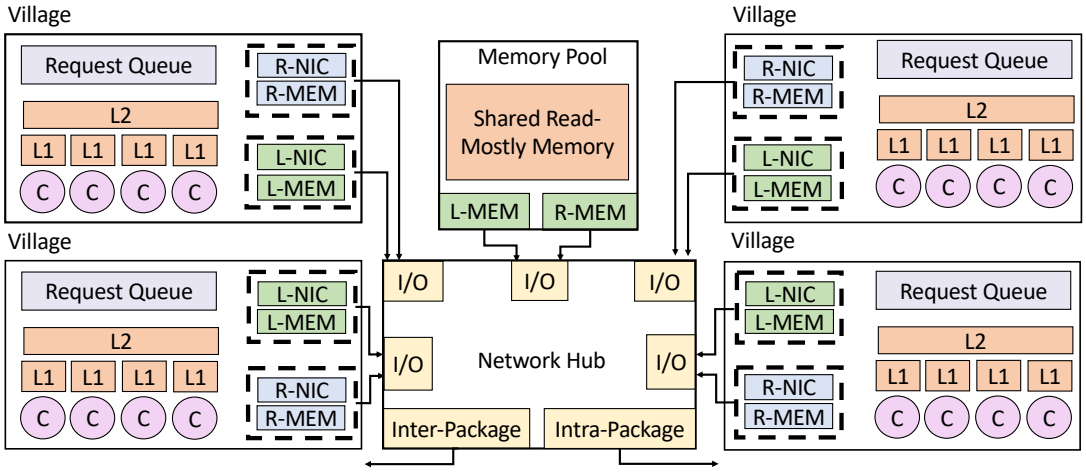


Figure 10: Organization of a *μManycore* cluster.

Communication Modules. In a village, the local (*L*) I/O port is for communication within the *μManycore*, and the remote (*R*) I/O port is for communication outside the *μManycore*. Each port contains a NIC and a hardware module to perform bulk memory transfers (*MEM*)—useful to prefetch or write-back data chunks. The reason why a village has two NICs is that the L-NIC is simpler. The L-NIC runs on a lossless on-package network and, therefore, does not need to support complicated transports (e.g., TCP) for re-transmissions and congestion control. The network has back-pressure support; the source waits for the network to become available before sending messages. There is never the need for retransmission to handle loss or for flow or congestion control. On the other hand, the R-NIC operates on a lossy network when communicating with the external world, and needs to support retransmission, reordering, flow control (to avoid hogging the sender), and congestion control (to avoid saturating the network). It estimates congestion using acknowledgment (ACK) packets (e.g., in TCP or RDMA).

The *Network Hub* (NH) connects to the local and remote ports of all the villages in the cluster. In addition, it is connected to the on-package ICN (via the intra-package port) and to the *μManycore*'s top-level NIC (via the inter-package port) to communicate with the outside world (Figure 12). The local ports of the villages communicate with the intra-package port; the remote ports of the villages communicate with the inter-package port.

Figure 11 shows two clusters with their NHs as leaf switches of the ICN, connected to another NH that acts as a second-level switch. As we will see, groups of non-leaf NHs are placed in chiplets. All the chiplets in clusters, plus the non-leaf NH chiplets, form the processor package. This design can scale up to one thousand cores or more.

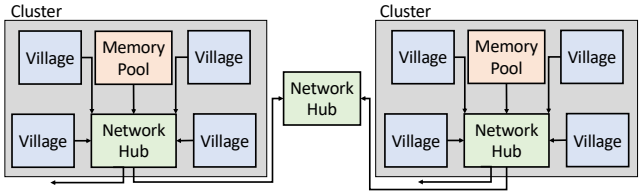


Figure 11: Two clusters connected via non-leaf network hub.

Memory Pool. A memory pool (implemented as a separate SRAM chiplet) contains a large volume of fast-access mostly-read data that multiple service instances in the villages of the local cluster may read (Figure 10). As indicated before, it contains snapshots of services so that, when a new service instance is created in the cluster, it can fetch the snapshot and skip instance boot-up and initialization overheads. The memory pool also has hardware modules to perform bulk memory transfers to and from on-package memory (*L-MEM*) or to and from off-package memory (*R-MEM*).

Resource Allocation. Each village runs its own light-weight operating system such as a microkernel, and communicates with other villages using messages. A service instance always stays within one village. When the number of concurrent requests for a given instance exceeds the capacity of the village, the system creates another instance of that service in another village. The two instances are independent and serve different arriving requests.

A village can also run instances of multiple different services. In this case, *μManycore* partitions the cores within the village across colocated service instances based on the instances' load. Each core is assigned a Service ID, which is stored in a separate register. In this way, the system ensures a more predictable performance and minimizes any negative interference between services.

A security-sensitive service instance can exclusively own a village. In this way, we reduce the chances that a malicious program performs side-channel attacks.

4.2 Hierarchical Leaf-Spine Interconnection Network

The network hub (NH) in each cluster is a leaf switch of an on-package hierarchical *Leaf-Spine* interconnection network (ICN). The leaf-spine is a topology that provides high connectivity between nodes [12, 20]. The left part of Figure 12, inside a dashed box, shows the leaf-spine topology. The per-cluster NHs are the leaf switches inside the box. Each NH is connected all-to-all to a set of fewer, second-level NHs (4 in the figure). These second-level NHs are standalone (i.e., not associated with any cluster), as shown in Figure 11. This topology allows any pair of leaf NHs to communicate

in two hops, and using as many different paths as there are switches in the second level of the tree.

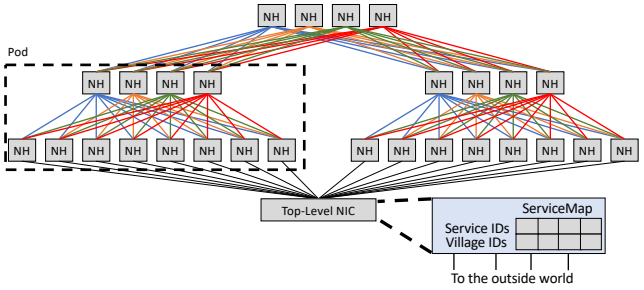


Figure 12: Hierarchical leaf-spine interconnection network, where NH stands for network hub. The figure also shows the connection to the package top-level NIC.

Since a *µManycore* has many clusters, we build the leaf-spine topology hierarchically. Figure 12 shows how the original topology (now called a Pod) is connected to other pods with a third level of NHs. This is the topology used in a *µManycore*. In our 1024-core *µManycore* design, we have 4 pods and 8 third-level NHs. Thanks to this topology's ability to connect many clusters with low hop counts and with redundant paths, this topology minimizes tail latency.

To provide connectivity to the outside world, the leaf NHs are also directly connected to the top-level NIC of the package (Figure 12). Incoming external requests flow from the top-level NIC to a leaf NH and then to the remote I/O port of a village. Outgoing requests flow in the opposite direction. The top-level NIC schedules incoming requests to the villages in hardware. Specifically, it maintains a *ServiceMap* table that stores, for each service ID, the set of villages that host an instance of that service. The ServiceMap is populated by the system software every time a new service instance is initialized in any village. When a request for a given service arrives at the top-level NIC, the hardware checks which villages are able to serve the request. Then, the hardware forwards the request to one of those villages in a round-robin manner.

4.3 Hardware Support for Request Queuing and Scheduling

As a village receives requests to execute locally, it is important to minimize the overheads of (i) depositing them on a queue and (ii) picking them up from the queue and executing them on local cores. Minimizing these overheads reduces request tail latency. Consequently, *µManycore* performs these operations in hardware.

To support these operations, each village includes a hardware *Request Queue* (RQ) (Figure 13). The RQ is implemented as a circular buffer, with head and tail pointers. Each full RQ entry corresponds to a service request that is executing or wants to execute in the local village. Each RQ entry contains three fields. The first one, *Status*, is the status of the request, which can be: running, ready to run, blocked on an RPC, or finished. The second field, *Service ID*, is the ID of the service that the request invokes. Recall that a village can have instances of multiple services. The third field, *Req Ptr*, is a pointer to a local memory called *Request Context Memory* that contains the context of the request. The context includes: the input data, the destination service that should receive the results of this request's execution, the ID of the process assigned to execute the request, and the core where this request runs (if known).

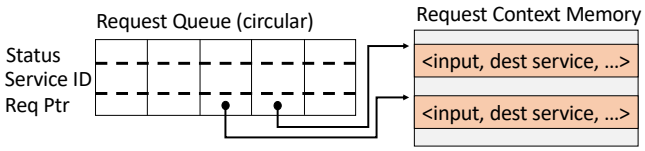


Figure 13: Hardware-based Request Queue.

On request arrival, the village NIC hardware performs all the RPC layer processing, such as header parsing, payload de-serialization, and service dispatching [62]. Then, it places the request at the tail of the RQ. Idle cores spin on a per-core local *Work* flag that is automatically set when the RQ contains work to do. When the flag is set, a core executes a *Dequeue* instruction that takes as argument the ID of the service that the core is tasked to execute. Recall from Section 4.1 that, when multiple services are co-located in a village, individual cores are assigned to specific services. The Dequeue instruction atomically accesses the RQ and returns the highest-priority entry (i.e., the one closest to the RQ head) that matches the service ID and is ready to run. It also sets the entry's status to running.

After a core completes the execution of a request, it executes a *Complete* instruction, passing as argument a pointer to the RQ entry. The hardware atomically accesses the RQ, sets the request status to finished and, if the entry was at the RQ head, advances the head to the first unfinished entry. With this hardware, *µManycore* minimizes the tail latency effects of request queuing and scheduling. Moreover, by processing requests in FCFS order, this scheme further minimizes tail latency.

An alternative scheduling policy to use is Shortest Remaining Processing Time First (SRPT). However, in microservice environments, SRPT is unlikely to improve much over FCFS for two reasons. First, requests for a given service tend to have similar execution times. Second, request execution is frequently interrupted by I/O calls, which in our case will provide frequent opportunities to schedule other ready-to-run requests.

If a request finds a full RQ, it is temporarily queued in the NIC. If the NIC has exhausted its buffering space, it rejects the request.

A more advanced design of the RQ would involve dynamically partitioning it into multiple RQs—each partition devoted to a different service. Specifically, when the system co-locates a second service instance in a village, the system would partition the RQ and record the new RQ structure in an RQ_Map hardware table. The proportion of entries assigned to each service can be the same as the proportion of cores assigned to each service. Since each core maintains a register with the ID of the service it is assigned to execute and passes it as an argument to the Dequeue instruction, all that is needed is to augment the Dequeue instruction to check the RQ_Map first. This additional hardware would eliminate contention of different-service cores for the same RQ, likely reducing the tail latency further. We do not consider this design in the evaluation.

4.4 Hardware Support for Context Switching

State-of-the-art schemes for efficient scheduling of microservice workloads [7, 36, 61, 63, 67] use a "run-to-completion" model: a

core is assigned to execute a request until it completes. At best, the process is pre-empted if it runs for very long, to prevent head-of-line blocking [36]. In practice, as shown in Section 3.3, the process executing a request is blocked most of the time, due to issuing storage accesses or calling other services. In the meantime, the cores context switch and execute other requests. The frequent context switching induces overhead and expands tail latency.

To assess this overhead, we run the highly-optimized Shinjuku software scheduler [36] in our simulated 1024-core ScaleOut manycore (Section 5). Shinjuku needs to (1) run on a dedicated core, (2) detect when a process on a core blocks, (3) save the context of the blocked process, (4) find a ready request by checking the RQ, and (5) restore the context of the ready request from memory. We find that this centralized software easily becomes a bottleneck and limits the overall throughput. It consumes time and, as shown in Figure 6, results in a high tail latency.

To address this problem, *μManycore* adds hardware support to reduce the overhead of context switching. The idea is that, when the execution of a request blocks, special hardware in the core saves the process state to memory. Then, the core is ready to access the RQ to get a new request. Also, when a core obtains from the RQ a request that had partially executed in the past, the hardware restores from memory the state of the request. The state saved and restored includes general-purpose and special-purpose registers; interrupt, exception, debugging, and privilege level information; and cached storage descriptors [47, 48]. The state is a few hundreds of bytes.

To support this design, the entry for a request in the Request Context Memory (Figure 13) is expanded to include space for the saved process state. Further, when a process executing on a core issues an RPC and is about to get blocked, the core executes a new *ContextSwitch* instruction. In hardware, this instruction saves the process state in the corresponding entry of the Request Context Memory, and sets the Status field in the RQ to blocked. The core is now free to spin on the *Work* flag to see if there is work to do.

When the NIC receives the RPC response, it puts the response in the Request Context Memory entry of the corresponding request, and then changes the Status field of the RQ entry from blocked to ready to run. At this point, an idle core will see a set *Work* flag and execute a Dequeue instruction. *μManycore* augments the Dequeue instruction to also upload the state of the selected request from the Request Context Memory to the core registers. The other functionality of Dequeue is unchanged.

Overall, with this support, cores keep context-switching overheads to a minimum, and can perform useful work practically all the time—effectively reducing tail latency.

5 METHODOLOGY

We model a *μManycore* package with 1024 cores organized into 32 clusters. Each cluster has 4 villages of 8 cores each, one memory pool, and a network hub (NH). Each village has a 64-entry request queue (RQ). *μManycore* uses a leaf-spine interconnect with a three-level hierarchy. There are 32 leaf-level NHs organized into 4 chiplets. For the second level, there are 4 groups of 4 NHs, organized into 4 chiplets. Each second-level NH in a group connects to all 8 NHs of the first level. For the third level, there are 8 NHs in two chiplets, where each third-level NH is connected to all 16 second-level NHs. The longest communication path is only 4 hops. Overall, a *μManycore* package has 32 clusters, 128 villages, 32 memory pools, and 56 NHs, for a total of 74 chiplets.

μManycore has simple, energy-efficient cores similar to ARM A15 [3]. They are 4-issue and run at 2GHz. They have a small ROB (64 entries) and LSQ (64 entries), private L1 caches, a single-level TLB, and a shared L2 cache.

We model two baseline hardware-coherent processors: the *ServerClass* multicore and the *ScaleOut* manycore. *ServerClass* is a beefy server-class processor, similar to Intel's IceLake [34]. Its cores are 6-issue and run at 3GHz. They have a large ROB (352 entries) and LSQ (256 entries), private L1 and L2 caches, two levels of TLBs, and a shared L3 cache. For comparison to *μManycore*, we evaluate two sizes of *ServerClass* processors: one with 40 cores that consumes the same power as *μManycore*, and one with 128 cores that has the same area as *μManycore*. The former is like a current high-end IceLake; the latter is an unrealistically power-hungry multicore.

ScaleOut is a 1024-core manycore organized into 32 clusters. *ScaleOut* uses the same cores and cache hierarchy as *μManycore*, including L2 caches shared by 8 cores. *ScaleOut* does not include the *μManycore* novelties: no global cache coherence, leaf-spine ICN, hardware support for request queuing and scheduling, and hardware support for context switching.

ServerClass and *ScaleOut* use conventional ICNs, namely a mesh and a fat-tree, respectively. For comparison to *μManycore*, the fat-tree topology has 63 NHs and its longest path is 10 hops. Both baselines use a highly-optimized state-of-the-art software-based context-switching scheme [36] and techniques that reduce NIC-to-core communication overheads [32, 77].

We model 10-server machines with each of the three types of processors. Table 2 shows the parameters of the architectures. To model these machines, we use the SST architectural simulator [65] connected to the DRAMSim2 memory simulator [66]. We use Pin [49] to collect traces and feed them to the SST simulator.

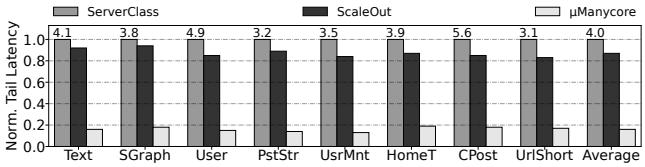
To compute the area and power consumed by each of the processors, we use CACTI [5] for the memory structures and McPAT [46] for the cores. We use the 32nm technology available with the tools, and then scale to 10nm technology [76]. The combined dynamic and static power consumed by one core and its portion of the cache hierarchy is: 10.225W for *ServerClass*, 0.396W for *ScaleOut*, and 0.408W for *μManycore*.

Applications. We use applications from the open-source DeathStarBench [23] microservice benchmark suite with commit ID c86920a. Due to space limitations, we show the results for only the 8 Social Network applications. The results are similar for the other applications of the benchmark suite. We use Poisson distributions for the request inter-arrival time. We use average loads of 5K, 10K and 15K requests per second (RPS) per server, which correspond to average CPU utilizations of <30%, 30-60%, and >60%, respectively. We collect the tail and average response time and throughput for each application.

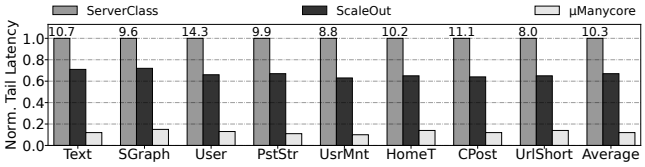
In addition, like prior work [36], we also use synthetic benchmarks with three service time distributions (exponential, lognormal, and bimodal) and 2–6 blocking calls during the execution.

Table 2: Architectural parameters used in the evaluation.

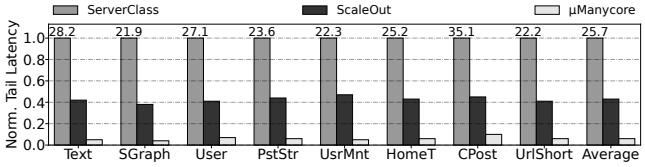
ServerClass Multicore	
Multicore	40 (or 128) 6-issue cores, 352-entry ROB, 256-entry LSQ, 3GHz
L1 cache	64KB, 8-way, 2 cycles round trip (RT), 64B line
L2 cache	2MB, 16-way, 16 cycles RT, 20 MSHRs
L3 cache	2MB/core, 16-way, 40 cycles RT, 20 MSHRs
L1 DTLB	256 entries, 4-way, 2 cycles RT
L2 DTLB	2048 entries, 12-way, 12 cycles RT
Network	2D mesh
µManycore and *ScaleOut* Manycores	
Manycore	1024 4-issue cores, 64-entry ROB, 64-entry LSQ, 2GHz
L1 cache	64KB, 8-way, 2 cycles RT, 64B line
L2 cache	256KB, 16-way, 24 cycles RT, 20 MSHRs
L1 DTLB	128 entries, 4-way, 2 cycles RT
Network	Fat tree (*ScaleOut*), leaf-spine (*µManycore*)
Network	
Intra server	5 cycles/hop (4 router delay + 1 wire delay) [9]
Inter server	1μs RT; 200GB/s
Main-memory per Server	
Capacity	80GB
Channels; Banks	4; 8
Frequency; Rate	1GHz; DDR
Mem bandwidth	8 memory controllers; 102.4GB/s per controller



(a) Load of 5K RPS.



(b) Load of 10K RPS.



(c) Load of 15K RPS.

Figure 14: Tail latency in *ServerClass*, *ScaleOut*, and *µManycore* normalized to *ServerClass*. The numbers on top of the *ServerClass* bars are the absolute latency values in ms.

6 EVALUATION

In this evaluation, response time (i.e., latency) is measured end-to-end, from when the client sends a request to when it receives the result. We give both the average and the P99 (i.e., 99^{th} percentile) values. Unless otherwise indicated, *ServerClass* has 40 cores and, therefore, consumes approximately the same power as *µManycore*.

6.1 End-to-End Tail Latency

Figure 14 shows the tail latency in the three architectures when running the DeathStarBench applications, normalized to *ServerClass*. On top of the *ServerClass* bars, we show the absolute latency values in ms. The systems are tested with three load levels: (a) 5K, (b) 10K, and (c) 15K RPS. We see that *µManycore* significantly reduces the tail latency for all applications across all loads. On average, *µManycore* reduces the tail latency over *ServerClass* by 6.3×, 8.3×, and 16.7× for loads of 5K, 10K and 15K RPS, respectively, and over *ScaleOut* by 5.4×, 6.5×, and 7.4× for the same loads.

µManycore achieves greater reductions with higher system loads, especially for the applications that are blocked more frequently, i.e., the ones that invoke a larger number of downstream services such as SocialGraph service (SGraph). In these cases, the impact of the *µManycore* techniques is more notable.

6.2 Tail-Latency Reduction Breakdown

Figure 15 shows the contributions of the four main *µManycore* techniques to the reduction of tail latency for 15K RPS. Latency reductions are normalized to the latency of *ScaleOut*. We apply the four techniques one by one in order: villages (Section 4.1), leaf-spine topology (Section 4.2), hardware scheduling (Section 4.3), and hardware context switching (Section 4.4). On average, the cumulative application of these techniques reduces the tail latency by 1.1×, 2.3×, 3.9×, and 7.4×, respectively. All techniques deliver major reductions except for the village organization, which reduces the tail latency by a modest 10%. The reason for this modest reduction is that we have favored the *ScaleOut* baseline. Specifically, while *ScaleOut* uses global cache coherence, it has one queue per 32-core cluster, and only allows processes to migrate between the 32 cores of a cluster. If we allowed processes to migrate between all 1024 cores, *ScaleOut* would perform worse. In any case, the main attractive of villages is not higher performance, but a reduction in manycore area, power, and complexity by eliminating global cache coherence—potentially allowing hardware resources to be used for other goals.

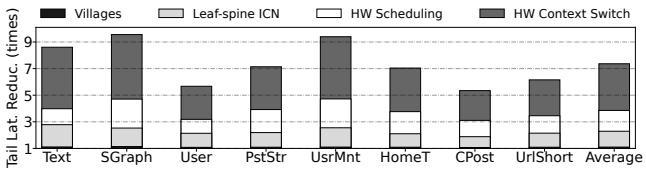
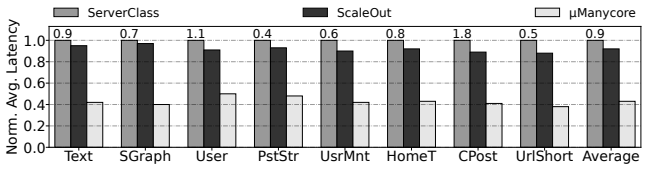


Figure 15: Contributions of the four main *µManycore* techniques to the reduction of tail latency for 15K RPS. Latency reductions are normalized to the tail latency of *ScaleOut*.

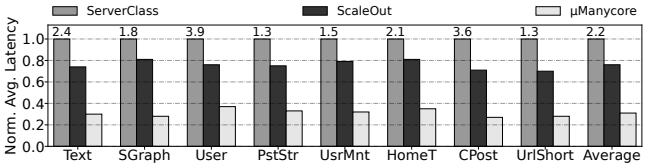
In applications that frequently use the ICN, such as Text and SGraph, the leaf-spine ICN is very effective. These same applications also substantially benefit from the hardware scheduling and hardware context switching techniques of *µManycore*. They often have requests stalled, waiting for remote storage accesses or service calls, and *µManycore* mitigates these overheads.

6.3 End-to-End Average Latency

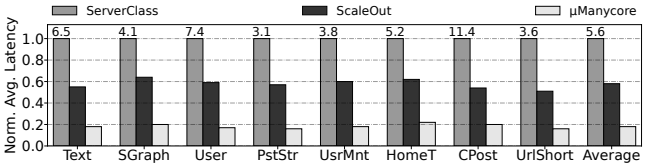
Figure 16 shows the normalized *average* latency in the three designs and for the three load levels. The numbers on top of the *ServerClass* bars are the absolute latency values in ms. *µManycore* reduces the average latency for all applications across all loads. The average latency reductions are smaller than the tail latency reductions in Figure 14. This is because the *µManycore* design is more tailored



(a) Load of 5K RPS.



(b) Load of 10K RPS.



(c) Load of 15K RPS.

Figure 16: Average latency in *ServerClass*, *ScaleOut*, and *μManycore* normalized to *ServerClass*. The numbers on top of the *ServerClass* bars are the absolute latency values in ms.

to minimizing the tail latency, by removing the major sources of contention and interference. On average, *μManycore* reduces the average latency over *ServerClass* by 2.3×, 3.2×, and 5.6× for loads of 5K, 10K, and 15K RPS, respectively, and over *ScaleOut* by 2.1×, 2.5×, and 3.2× for the same loads.

6.4 Reduction in Tail-to-Average Ratio

The goal of *μManycore* is to minimize the tail latency, but also to bring it closer to the average. In this way, the response time becomes more predictable, and more users can be served within the guaranteed QoS. Figure 17 shows the normalized tail-to-average latency ratio per application for the three designs averaged across all the loads. The numbers on top of the *ServerClass* bars are the absolute ratios.

In *μManycore*, the tail to average latency ratio is significantly smaller than in the other architectures. On average, it is 2.7× and 2.3× lower than in the *ServerClass* and *ScaleOut* baselines, respectively. In *UsrMnt*, the tail to average latency ratio in *μManycore* is 3.3× lower than in *ServerClass*.

In the baselines, the ratio between the tail and the average is especially notable under high loads. Some requests are served quickly, but the slowest ones are substantially slowed down due to queuing and contention. In such environments, *μManycore* reduces the tail latency while keeping the average latency low, thus shrinking the ratio between tail and average.

6.5 End-to-End Throughput Improvements

We measure the number of requests that can be served by the system (i.e, the throughput) without violating QoS guarantees. We say that a QoS violation occurs if the request execution time is higher than 5 times the contention-free average request execution time.

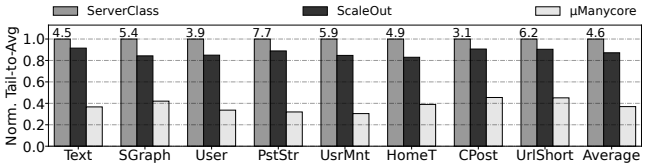


Figure 17: Tail-to-average latency ratio of *ServerClass*, *ScaleOut*, and *μManycore* normalized to *ServerClass*. The numbers on top of the *ServerClass* bars are the absolute ratios.

Figure 18 shows the normalized maximum throughput that each of the three designs can achieve without violating QoS guarantees. The numbers on top of the *μManycore* bars are the absolute throughput values that *μManycore* achieves in KRPS. *μManycore* reaches a throughput that is 13.9–17.1× higher than the *ServerClass*. On average, *μManycore* improves the throughput by 15.5× and 4.3× over the *ServerClass* and *ScaleOut* baselines, respectively.

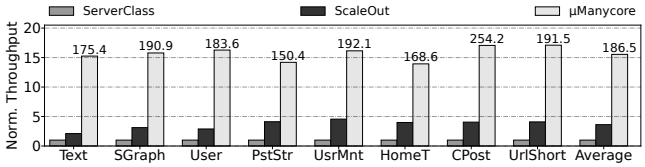


Figure 18: Normalized maximum throughput a system can achieve without violating QoS guarantees. The numbers on top of the *μManycore* bars are the absolute throughput values that *μManycore* achieves.

6.6 Sensitivity Analysis

μManycore can be organized with different configurations of number of cores per village, number of villages per cluster, and number of clusters. Figure 19 shows the tail latency in four configurations. The bars are normalized to the tail latency of the first configuration, which is the default configuration used in all the previous experiments.

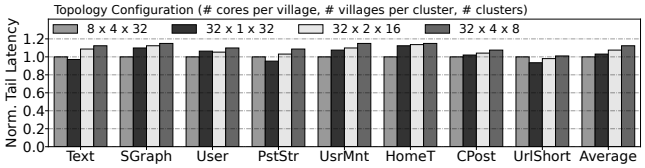


Figure 19: Normalized tail latency with different *μManycore* configurations.

All configurations are within 15% of each other's tail latency. Interestingly, different services are best suited to different configurations. Specifically, services that do not call other services such as *UrlShort* perform better in larger villages (32x1x32). On the other hand, services that frequently invoke other services such as *HomeT* and *SGraph* have better performance with many smaller villages (8x4x32). Overall, our default configuration has the overall lowest tail latency.

6.7 Tail Latency with Synthetic Benchmarks

We now consider the effect of the service-time distributions of the requests on the tail latency. We consider synthetic requests whose

service time is distributed exponential, lognormal and bimodal. Figure 20 shows the resulting tail latency for the three systems (*ServerClass*, *ScaleOut*, and *µManycore*) and for the three load levels (5K, 10K, and 15K RPS). The bars are normalized to *ServerClass*, and the numbers on top of the *ServerClass* bars are the absolute latency values in μs.

From the figure, we see that the previous trends hold for all these service distributions. *µManycore* substantially outperforms both baselines for all loads and service-time distributions. On average across all loads and service-time distributions, *µManycore* reduces the tail latency by 9.1× and 7.2× over *ServerClass* and *ScaleOut*, respectively. With increased system load, the gains of *µManycore* become larger.

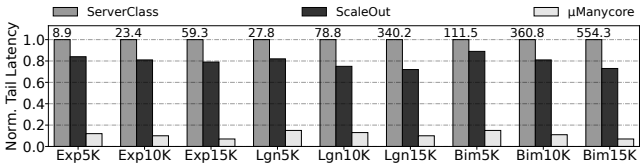


Figure 20: Tail latency of *ServerClass*, *ScaleOut*, and *µManycore* normalized to *ServerClass* with synthetic benchmarks. The numbers on top of the *ServerClass* bars are the absolute latency values in μs.

6.8 Comparison to an Iso-Area ServerClass CPU

The evaluation so far has used iso-power configurations for *ServerClass*, *ScaleOut*, and *µManycore*. In this section we compare iso-area configurations. As indicated in Section 5, we use CACTI [5] and McPAT [46] for our computations. In the iso-power configurations, *µManycore* has 2.9% more area than *ScaleOut* and 3.1× more area than the 40-core *ServerClass* (i.e., 547.2mm^2 for *µManycore* versus 176.1mm^2 for *ServerClass*). Hence, for an iso-area comparison, we keep *µManycore* and *ScaleOut* unchanged and we scale *ServerClass* to 128 cores, while leaving all the other parameters unmodified. The new *ServerClass* processor improves the performance significantly, matching and sometimes slightly outperforming the tail latency of *ScaleOut*. However, *ServerClass* still has a tail latency that is on average 7.3× higher than the *µManycore* one across all loads and applications. Also, the 128-core *ServerClass* processor uses an unacceptably large amount of power, namely 3.2× more than *µManycore*.

7 RELATED WORK

Software schedulers. Recently, researchers have explored various software solutions for μ-second scale scheduling and context switching [7, 22, 31, 36, 37, 52, 53, 61, 63, 91, 97]. IX [7] schedules a batch of requests at high throughput, but it degrades the tail latency of heavy-tailed service time distributions due to using a per-core distributed scheduler. ZygOS [63] minimizes head-of-line blocking and load imbalance by allowing cores to steal requests from other cores via expensive software operations. It can potentially degrade the tail latency of short requests. Shinjuku [36] uses a centralized scheduler with request preemption to tolerate different service time distributions. It cannot scale to a large number of cores, and may degrade the tail latency due to the cost of software context switches. Shenango [61] dedicates a core to perform scheduling, possibly limiting its throughput and scalability. *µManycore* performs scheduling and context switching in hardware, thus reducing the overheads and increasing the throughput over such software schemes.

Message passing operating systems (OSes). Fos [87] and Barrelfish [6] are distributed OSes: a core runs a local OS and communicates with the OS of other cores only via message passing. There is no cache coherence. However, as per Section 4.1, this architecture is not well-suited for microservices. It is possible to use some ideas from fos and Barrelfish to support the communication between the shared-memory OSes running on each village.

RPC accelerators. Researchers proposed hardware accelerators to improve the efficiency of the RPC-based communication [15, 32, 39, 44, 62, 67, 77, 95]. RPCValet [15] uses the on-chip NICs to monitor per-core load and to steer RPCs to lightly-loaded cores. Nebula [77] provides hardware support for efficient in-LLC network buffer management, and sends incoming RPCs into the CPU cores' L1 caches. The nanoPU [32] bypasses the cache and memory hierarchy and places the arriving messages directly into the CPU register file. Cerebros [62] executes all RPC layers in hardware without involving the CPU. While *µManycore* has been inspired by these systems, none of them considers services that invoke other services and are waiting idly for long durations. Therefore, they execute in the run-to-completion manner and do not focus on efficient support for context switching, as in *µManycore*.

Duplexity [56] is a processor architecture that, when there are not enough latency-critical jobs (e.g., microservices) to run, it reconfigures and uses the idle resources to run batch workloads. *µManycore* could incorporate this approach to increase core utilization in low-load periods.

Packet processing's execution model, such as supported by the Event Machine [58] can in principle be applied to process microservice invocations. In packet processing, arriving packets are queued up in multiple queues. Then, the system dequeues packets with some notion of priority, and sends them to execute on available cores. In *µManycore*, service invocations are queued and dequeued in hardware. A key characteristic of microservice processing is that service invocations frequently stall on I/O. In addition, some individual service invocations may execute in a multithreaded manner.

Hardware queuing. Hardware queues [43, 45, 68] have been proposed to support low latency communication between producer and consumer threads running on different cores. Existing proposals target traditional task-parallel systems and bind the queue state to an application's context. In *µManycore*, queues are not per application, but contain entries for different service requests. Hence, entries are not saved and restored on a context switch. The producer is a NIC and consumers are cores in the village. ALTOCUMULUS [95] proposes a scheme for RPC scheduling in hardware. It does not consider that requests are idle for most of the time due to blocking calls. The scheduler in *µManycore* takes into account blocked requests and only schedules those that are runnable.

Chiplet-based processor designs. Recently, both industry and academia have shown great interest in chiplet-based processor designs [4, 19, 57, 69, 86, 89, 90]. These designs improve yield, allow

the integration of heterogeneous components, and simplify processor design. In some designs, processors are grouped in clusters or core complexes [57]. One way in which *μManycore* goes beyond these designs is that, in *μManycore*, cache coherence is only supported inside these clusters (called villages), not across them.

8 FUTURE WORK

μManycore can be enhanced in a variety of ways to improve the performance of microservice environments. These enhancements add additional costs.

In *μManycore*, when different service instances are co-located in the same village, *μManycore* apportions the cores to the different service instances based on the expected load. It is possible that, as requests arrive, the distribution of load across services is different than expected—e.g., the cores of one of the instances are mostly idle while those of the other are unable to keep up with the requests. In this case, an enhancement to *μManycore* would be to allow an instance to temporarily steal cores assigned to another instance.

In *μManycore*, all villages have the same hardware. It is, therefore, a homogeneous architecture. A possible enhancement is to have different hardware in different villages. For example, some villages might have bigger cores. This approach would enable the assignment of different types of services to different types of villages—hence tailoring the hardware to the needs of the service instances. However, it is unclear what different types of villages and how many of each are needed. Moreover, services would likely have to be instrumented to declare what type of village they would prefer.

9 CONCLUSION

To address the imbalance between emerging microservice environments and current processors, this paper proposed *μManycore*, an architecture optimized for microservice environments. Based on a characterization of microservice applications, *μManycore* is designed to minimize unnecessary microarchitecture and reduce tail latency. Rather than supporting manycore-wide hardware cache coherence, *μManycore* has multiple hardware cache-coherent smaller domains called villages. Clusters of villages are interconnected with a leaf-spine network, which has many redundant, low-hop-count paths between clusters. To minimize overheads, *μManycore* schedules and queues service requests in hardware, and includes support to save and restore process state in a context-switch in hardware. Our simulation-based results showed that *μManycore* delivers high performance for microservice workloads. A cluster of 10 servers with a 1024-core *μManycore* in each server delivered 3.7× lower average latency, 15.5× higher throughput, and 10.4× lower tail latency than a cluster with iso-power conventional server-class multicores. Similar good results were attained compared to a cluster with power-hungry iso-area conventional server-class multicores.

ACKNOWLEDGMENTS

This work was supported in part by NSF under grants CNS 1956007 and CCF 2107470, and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*.
[2] Amazon AWS. 2023. AWS Lambda. https://aws.amazon.com/lambda/.
[3] ARM. 2023. ARM Cortex A15. https://developer.arm.com/Processors/Cortex-A15.
[4] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-chip-module GPUs for continued performance scalability. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA '17)*.
[5] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization (TACO '17)* (2017).
[6] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*.
[7] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*.
[8] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. 2021. Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning.
[9] 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 '20)*.
[10] Milind Chabbi and Murali Krishna Ramanathan. 2022. A Study of Real-World Data Races in Golang. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*.
[11] Shenghsun Cho, Amoghavarsha Suresh, Tapti Palit, Michael Ferdman, and Nima Honarmand. 2018. Taming the Killer Microsecond. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '18)*.
[12] Cisco. 2023. Cisco Spine and Leaf Architecture. https://ciscolicense.com/blog/cisco-spine-and-leaf-architecture/.
[13] Clang. 2023. A C language family frontend for LLVM. https://clang.llvm.org.
[14] Google Cloud. 2023. What is Microservices Architecture? https://cloud.google.com/learn/what-is-microservices-architecture.
[15] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. 2019. RPCValet: NI-Driven Tail-Aware Balancing of µs-Scale RPCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*.
[16] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56 (2013), 74–80.
[17] Docker. 2023. Docker Compose. https://docs.docker.com/compose/.
[18] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*.
[19] Pete Ehrett, Todd Austin, and Valeria Bertacco. 2021. Chopin: Composing Cost-Effective Custom Chips with Algorithmic Chiplets. In *2021 IEEE 39th International Conference on Computer Design (ICCD '21)*.
[20] Engineering at Meta. 2023. Introducing data center fabric, the next-generation Facebook data center network. https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/.
[21] B. Flachs, S. Asano, S.H. Dhong, H.P. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H.-J. Oh, S.M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yano, D.A. Brokenshire, M. Peyravian, V. To, and E. Iwata. 2006. The microarchitecture of the Synergistic Processor for a Cell Processor. *IEEE Journal of Solid-State Circuits* (2006).
[22] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*.
[23] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural*

Support for Programming Languages and Operating Systems (ASPLOS '19).
[24] Golang. 2023. Http Package. https://pkg.go.dev/net/http.
[25] Google. 2023. Google Cloud Functions. https://cloud.google.com/functions.
[26] Google. 2023. gVisor: Container Runtime Sandbox. https://gvisor.dev/docs/.
[27] gRPC. 2023. An RPC library and framework. https://github.com/grpc/grpc.
[28] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *IEEE International Symposium on High Performance Computer Architecture (HPCA '18).*
[29] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, and Joaquin Quiñonero Candela. 2014. Practical Lessons from Predicting Clicks on Ads at Facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising.*
[30] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabrice Pailet, Shailendra Jain, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Apel, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob Van Der Wijngaart, and Timothy Mattson. 2010. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *2010 IEEE International Solid-State Circuits Conference - (ISSCC '10).*
[31] 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 (SOSP '21).*
[32] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. 2021. The nanoPU: A Nanosecond Network Stack for Datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21).*
[33] IBM. 2023. IBM Cloud Functions. https://cloud.ibm.com/functions/.
[34] Intel. 2023. Intel Xeon Platinum 8380 Processor. https://ark.intel.com/content/www/us/en/ark/products/212287/intel-xeon-platinum-8380-processor-60m-cache-2-30-ghz.html.
[35] Daniel Jimenez and Calvin. Lin. 2001. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture (HPCA '01).*
[36] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for µsecond-scale Tail Latency. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19).*
[37] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2019. Centralized Core-Granular Scheduling for Serverless Functions. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '19).*
[38] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19).*
[39] Mahmoud Khairy, Ahmad Alawneh, Aaron Barnes, and Timothy G. Rogers. 2022. SIMR: Single Instruction Multiple Request Processing for Energy-Efficient Data Center Microservices. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO '22).*
[40] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2020. I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '20).*
[41] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2021. Ripple: Profile-Guided Instruction Cache Replacement for Data Center Applications. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA '21).*
[42] Kubernetes. 2023. Production-Grade Container Orchestration. https://kubernetes.io/.
[43] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. 2007. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07).*
[44] Nikita Lazarev, Neil Adit, Shaojie Xiang, Zhiru Zhang, and Christina Delimitrou. 2020. Dagger: Towards Efficient RPCs in Cloud Microservices With Near-Memory Reconfigurable NICs. *IEEE Computer Architecture Letters* (2020).
[45] Sanghoon Lee, Devesh Tiwari, Yan Solihin, and James Tuck. 2011. HAQu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11).*
[46] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '09).*
[47] Linux. 2023. Pt Regs. https://elixir.bootlin.com/linux/v5.17/source/arch/ 86/include/asm/ptrace.h#L59.
[48] Linux. 2023. Thread Struct. https://elixir.bootlin.com/linux/v5.17/source/arch/86/include/asm/processor.h#L467.
[49] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05).*
[50] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21).*
[51] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. 2022. The Power of Prediction: Microservice Auto Scaling via Workload Learning. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '22).*
[52] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19).*
[53] 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).*
[54] Microsoft. 2023. Microsoft Azure Functions. https://azure.microsoft.com/en-gb/services/functions/.
[55] Amirhossein Mirhosseini, Brendan L. West, Geoffrey W. Blake, and Thomas F. Wenisch. 2019. Express-Lane Scheduling and Multithreading to Minimize the Tail Latency of Microservices. In *2019 IEEE International Conference on Autonomic Computing (ICAC '19).*
[56] Amirhossein Mirhosseini, Akshitha Sriraman, and Thomas F. Wenisch. 2019. Enhancing Server Efficiency in the Face of Killer Microseconds. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA '19).*
[57] Samuel Naffziger, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H. Loh, Mahesh Subramony, and Sean White. 2021. Pioneering Chiplet Technology and Design for the AMD EPYC™ and Ryzen™ Processor Families : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA '21).*
[58] Nokia Networks. 2023. Event Machine on ODP. https://openeventmachine.github.io/em-odp/.
[59] Old GigaOm. 2011. The biggest thing Amazon got right: The platform. https://old.gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/.
[60] Oracle. 2023. MySQL. https://www.mysql.com.
[61] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19).*
[62] Arash Pourhabibi, Mark Sutherland, Alexandros Daglis, and Babak Falsafi. 2021. Cerebros: Evading the RPC Tax in Datacenters. In *2021 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21).*
[63] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17).*
[64] Chris Richardson. 2023. What are microservices? https://microservices.io/.
[65] Arun F. Rodrigues, Jeanine Cook, Elliott Cooper-Balis, K. Scott Hemmert, Chad Kersey, Rolf Riesen, Paul Rosenfeld, Ron Oldfield, and Marlow Weston. 2006. The Structural Simulation Toolkit. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '10).*
[66] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* (2011).
[67] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. 2019. Elastic RSS: Co-Scheduling Packets and Cores Using Programmable NICs. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019 (APNet '19).*
[68] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. 2010. Flexible Architectural Support for Fine-Grain Scheduling. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10).*
[69] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. 2019. Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '19).*

[70] Shixin Song, Tanvir Ahmed Khan, Sara Mahdizadeh Shahri, Akshitha Sriraman, Niranjan K Soundararajan, Sreenivas Subramoney, Daniel A. Jiménez, Heiner Litz, and Baris Kasikci. 2022. Thermometer: Profile-Guided BTB Replacement for Data Center Applications. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22)*.

[71] Spring Framework. 2023. RestController. https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html.

[72] Akshitha Sriraman and Abhishek Dhanotia. 2020. Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*.

[73] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F. Wenisch. 2019. SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale. In *Proceedings of the 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA '19)*.

[74] Akshitha Sriraman and Thomas F. Wenisch. 2018. μSuite: A Benchmark Suite for Microservices. In *IEEE International Symposium on Workload Characterization (IISWC '18)*.

[75] Akshitha Sriraman and Thomas F. Wenisch. 2018. μTune: Auto-Tuned Threading for OLDI Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*.

[76] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm. *Integration the VLSI journal* (2017).

[77] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandros Daglis. 2020. The NEBULA RPC-Optimized Architecture. In *Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA '20)*.

[78] The Apache Software Foundation. 2023. Apache Cassandra. https://cassandra.apache.org/.

[79] The Apache Software Foundation. 2023. Apache Kafka. https://kafka.apache.org/.

[80] The Apache Software Foundation. 2023. Apache Thrift. https://thrift.apache.org/.

[81] Think Software. 2021. Microservices Architecture of Twitter Service. https://thinksoftware.medium.com/design-twitter-microservices-architecture-of-twitter-service-996ddd68e1ca.

[82] Uber. 2020. Introducing Domain-Oriented Microservice Architecture. https://www.uber.com/blog/microservice-architecture/.

[83] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. 2011. Light-Weight Communications on Intel's Single-Chip Cloud Computer Processor. *SIGOPS Operating Systems Review* (2011).

[84] Ketan Varshneya. 2021. Understanding design of microservices architecture at Netflix. https://www.techaheadcorp.com/blog/design-of-microservices-architecture-at-netflix/.

[85] Kangjin Wang, Cheng Wang, Tong Jia, Kingsum Chow, Yang Wen, Yaoyong Dou, Guoyao Xu, Chuanjia Hou Hou, Jie Yao, Liping Zhang Zhang, and Ying Li Li. 2022. Characterizing Job Microarchitectural Profiles at Scale: Dataset and Analysis. In *51st International Conference on Parallel Processing (ICPP '22)*.

[86] Tianqi Wang, Fan Feng, Shaolin Xiang, Qi Li, and Jing Xia. 2022. Application Defined On-chip Networks for Heterogeneous Chiplets: An Implementation Perspective. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA '22)*.

[87] David Wentzlaff and Anant Agarwal. 2009. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Oper. Syst. Rev.* (2009).

[88] Wordpress. 2023. Blog Tool, Publishing Platform, and CMS. https://wordpress.org/.

[89] Yibo Wu, Liang Wang, Xiaohang Wang, Jie Han, Jianfeng Zhu, Honglan Jiang, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2022. Upward Packet Popup for Deadlock Freedom in Modular Chiplet-Based Systems. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA '22)*.

[90] 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 '18)*.

[91] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*.

[92] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. 2020. High-Density Multi-Tenant Bare-Metal Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*.

[93] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. 2016. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA '16)*.

[94] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *USENIX Annual Technical Conference (USENIX ATC '22)*.

[95] Jiechen Zhao, Iris Uwizeyimana, Karthik Ganesan, Mark C. Jeffrey, and Natalie Enright Jerger. 2022. ALTOCUMULUS: Scalable Scheduling for Nanosecond-Scale Remote Procedure Calls. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO '22)*.

[96] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. Benchmarking Microservice Systems for Software Engineering Research. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)* (Gothenburg, Sweden).

[97] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. 2020. RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*.