Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Review:

Resource scheduling techniques in cloud from a view of coordination: a holistic survey*

Yuzhao WANG[1], Junqing YU‡[1], Zhibin YU[2]

[1]*School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China*

[2]*Center of Heterogeneous Intelligent Computer Architecture and Systems, Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China*

E-mail: yuzhao_w@hust.edu.cn; yjqing@hust.edu.cn; zb.yu@siat.ac.cn

Received June 24, 2021; Revision accepted Nov. 2, 2021; Crosschecked Jan. 4, 2023

Abstract: Nowadays, the management of resource contention in shared cloud remains a pending problem. The evolution and deployment of new application paradigms (e.g., deep learning training and microservices) and custom hardware (e.g., graphics processing unit (GPU) and tensor processing unit (TPU)) have posed new challenges in resource management system design. Current solutions tend to trade cluster efficiency for guaranteed application performance, e.g., resource over-allocation, leaving a lot of resources underutilized. Overcoming this dilemma is not easy, because different components across the software stack are involved. Nevertheless, massive efforts have been devoted to seeking effective performance isolation and highly efficient resource scheduling. The goal of this paper is to systematically cover related aspects to deliver the techniques from the coordination perspective, and to identify the corresponding trends they indicate. Briefly, four topics are involved. First, isolation mechanisms deployed at different levels (micro-architecture, system, and virtualization levels) are reviewed, including GPU multitasking methods. Second, resource scheduling techniques within an individual machine and at the cluster level are investigated, respectively. Particularly, GPU scheduling for deep learning applications is described in detail. Third, adaptive resource management including the latest microservice-related research is thoroughly explored. Finally, future research directions are discussed in the light of advanced work. We hope that this review paper will help researchers establish a global view of the landscape of resource management techniques in shared cloud, and see technology trends more clearly.

Key words: Coordination; Co-location; Heterogeneous computing; Microservice; Resource scheduling techniques
https://doi.org/10.1631/FITEE.2100298 **CLC number:** TP39

1 Introduction

Many enterprises and individuals have been offloading their workloads to public clouds such as Amazon and Google Compute Engine, or private clouds managed by resource management systems (RMSs) such as Mesos (Hindman et al., 2011), YARN (Vavilapalli et al., 2013), and Kubernetes (https://kubernetes.io/). This trend is mainly due to the great flexibility of the cloud for end-users and its operational benefits for providers.

These advantages are directly linked to the resource scheduling efficiency (e.g., scheduling delay and quality (Delimitrou et al., 2015)) and performance isolation effectiveness provided by the RMS, and ongoing trends towards increased heterogeneity in software and hardware pose new challenges in RMS design. For example, the current shift of cloud services from monolithic architecture toward microservice architecture fundamentally changes

‡ Corresponding author
* Project supported by the National Key R&D Program, China (No. 2016YFB1000204)
ORCID: Yuzhao WANG, https://orcid.org/0000-0002-9056-4277; Zhibin YU, https://orcid.org/0000-0001-8067-9612

many assumptions under which current RMSs were designed. Current cluster schedulers often fall short of capturing the complex dependencies of microservices, resulting in suboptimal scheduling. Another increasingly popular application domain, deep learning (DL), also exhibits radically distinct characteristics in terms of resource usage (e.g., graphics processing unit (GPU) demand) and intra-job behaviors as compared with traditional data analytic applications.

On the other side, data center (DC) hardware is also becoming increasingly heterogeneous as servers are progressively replaced and upgraded over time. Particularly, domain-specific accelerators, such as GPUs, tensor processing units (TPUs), field-programmable gate arrays (FPGAs), and application-specific integrated circuits (ASICs) are widely deployed as alternatives to general-purpose central processing units (CPUs) to accelerate critical operations. The isolation in individual accelerator-integrated servers, and the communication, cooperation, and coordination among accelerators and host CPUs expose more respects to consider for resource management. Therefore, a traditional RMS on a homogeneous cluster is ill-suited for DL-like workloads, and thus calls for rethinking and redesign.

Apart from the diversity of static attributes in the application codebase and physical resources, dynamic heterogeneity at runtime (e.g., the various transition behaviors across hardware) due to resource contention among co-located tasks tends to further distort the scheduling-decision space and distracts optimization exploration (Luo et al., 2019), such as the effectiveness of a performance model based on offline characterization. As a result, the tension between resource utilization and performance guarantee becomes more severe.

Overall, several challenges remain, especially in the context of heterogeneous computing and serverless computing.

1. Inefficient isolation support

Isolation mechanisms supported by current hardware or operating systems are limited. Multiple micro-architectural resources are shared unclearly, and thus a complete understanding of the corresponding interplay is required. This is particularly common in domain-specific accelerators, such as GPUs and FPGAs, which often make task co-location or multitasking infeasible. This shortage not only undermines the efficiency of offline profile-based coordination mechanisms (Han et al., 2016; Zhu HS and Erez, 2016), but also makes online interference tracing and performance correction invalid.

2. Resource utilization vs. quality of service (QoS)

In the context of imperfect resource isolation, resource over-allocation becomes commonplace to guarantee QoS, which leaves some resources idle. In addition, the diverse resource demands across applications and dynamic intra-application behaviors at runtime induce many resource fragments, which are quite difficult to capture promptly, and quantify accurately for re-allocation. As a result, resource utilization across the industry is generally low. For example, the GPU utilization at Microsort is only 52% on average (Jeon et al., 2019), and the CPU usage at Google is around 30% (Tirmazi et al., 2020).

3. Heterogeneity-aware coordination

Compared to a homogeneous environment, heterogeneity makes the scheduling procedure more complicated: (1) Hardware performance varies across different accelerator types, contributing differently to an application's QoS; (2) The exploration space for jobs with complementary behaviors for co-location is expanded significantly; (3) Various acceleration choices require an adaptive computation offloading model that accounts for the architectural difference.

To confront the above challenges, massive academic and industrial efforts are expended. Singh and Chana (2016) performed a methodical literature analysis of cloud resource scheduling that involves abundant algorithms and scheduling policies. In contrast, targeting application performance and security, Bauman et al. (2015) reviewed different out-of-VM (virtual machine) monitoring techniques to overcome the semantic gap with in-VM application. For heterogeneous computing, Hong et al. (2017) presented an in-depth survey of GPU virtualization techniques and their scheduling methods. Weerasiri et al. (2017) performed a broad analysis of cloud resource orchestration techniques by evaluating and classifying them by proposed taxonomy.

In contrast, this paper performs a holistic survey of resource management techniques across the cloud stack in a coordinated view, and captures corresponding development trends to shed light on future system design. Briefly, techniques are investigated at

the micro-architecture, operating system (OS), virtualization, and cluster levels. Particularly, resource management with respect to heterogeneous computing and serverless computing (i.e., microservice) is detailed. Based on our survey and categorization of existing approaches at each layer, we propose several guidelines for future system research, and provide a holistic view of the technical landscape for researchers. As Table 1 shows, related approaches at each level of the RMS stack are introduced following a bottom-up exposition order.

1. Micro-architecture level and virtualization level

Isolation mechanisms within individual machines (including accelerators, like GPUs) are involved in this taxonomy. Specifically, kernel-based schemes supported by hardware architecture (e.g., last-level cache repartition) or OS kernel, and software-based schemes via thread scheduling are included. Particularly, GPU and FPGA isolation techniques are also detailed (Section 2.1.2) and virtualization schemes consisting of hypervisors and container-based approaches are investigated in Section 2.1.3.

2. System level

This category includes software-based scheduling optimizations, which rely on available isolation mechanisms per node and monitored data to guarantee the application performance. For simplicity, related content is outlined based on the targeted resource types, including simultaneous multithreading, memory subsystem resources, and accelerators (Section 2.2.1). Notably, efforts regarding computation offloading on heterogeneous machines and corresponding OS designs are presented for heterogenous multicore processors, memory, GPUs, and FPGAs (Section 2.2.2). Scheduling techniques at the virtualization level are also included in Section 2.2.3. Particularly, network-related work is presented separately due to its special role in the data center (Section 2.4).

3. Cluster level

Existing resource management architectures and their evolution are introduced in detail (Section 3.1).

In addition, scheduling algorithms and adaptive techniques for performance compensation are detailed. For scheduling algorithms, research efforts on the conventional homogeneous clusters, such as data mining, linear programming, network-driven, and bin-packing-driven approaches are discussed, and techniques regarding GPU clusters (Section 3.3) are also investigated. For adaptive performance compensation techniques, adaptive management targeting at dynamic heterogeneity at runtime is detailed, including elastic methods for co-location, best-effort (BE) job-oriented resource throttling, and latency-sensitive (LS) job-oriented resource compensation, all of which are of great significance in reconciling resource under-utilization and application QoS. Particularly, resource management regarding microservices is described thoroughly in Section 3.4.3.

Overall, this paper is organized as follows: Section 2 introduces the isolation mechanisms and resource management methods within an individual server, including micro-architecture, virtualization,

Table 1 An overview of the topics at each layer and the related categories

Stack level	Topic	Related category
Cluster level	Scheduling system architectures	Centralized; two-level; shared-state; distributed; hybrid; hierarchical
	Task-to-machine mapping	Data mining; LP-driven; graph-driven; bin-packing-driven
	Scheduling in GPU cluster	BE-oriented; LS-oriented
	Adaptive management	Elasticity for co-location; BE-oriented; LS-oriented resource management
OS level	Task scheduling within machine	Profile-based; role-based; offloading model
	Heterogeneous OS design	CPU core; memory; GPU; FPGA
	Detection and correction	Profile-based; feedback-based
Virtualization level	Isolation methods	Hypervisor-based; container-based
Micro-architecture level	Isolation methods	Kernel-based; software-based; GPU-oriented

OS: operating system; GPU: graphics processing unit; LP: linear-programming; BE: best-effort; LS: latency-sensitive; CPU: central processing unit; FPGA: field-programmable gate array

and system levels in Table 1. Section 3 investigates existing cluster management systems with respect to the architecture, algorithms, and methods related to resource elasticity and performance compensation (at the cluster level). Finally, we discuss future research directions and possible solutions for the challenges outlined in Section 4, and draw conclusions in Section 5.

2 Performance coordination within a machine

Performance interference in a shared environment is ubiquitous due to resource contention. The coordination of co-running tasks (or threads) is imperative because it greatly impacts the QoS (e.g., the cache line tends to be evicted by co-running tasks). To coordinate the performance locally, performance isolation, coordination-oriented thread scheduling, and compensation control at runtime are widely explored. This section investigates prior research.

2.1 Resource isolation mechanisms

Isolation mechanisms at the micro-architecture level (in multi-processors and GPUs) and at the virtualization level are investigated in this subsection.

2.1.1 Isolation at the micro-architecture level

Prior work has proposed two main isolation approaches which target at micro-architectural resources: kernel-based mechanisms that rely on hardware/kernel support and software-based approaches that repartition resources via an analytical model, thread scheduling, page coloring, and so forth.

1. Kernel-based mechanisms

These include available isolation schemes that are supported by the current hardware architecture (e.g., last-level cache repartition) or OS kernel.

(1) Cgroups

The best-known isolation mechanism that is supported by the Linux kernel is probably control groups (i.e., Cgroups). It provides a mechanism for task segregation by partitioning resources like CPU time, system memory, disk, and network bandwidth into groups, then assigning processes to those groups. By using Cgroups via OS interfaces, a fine-grained control over allocating, prioritizing, and monitoring system resources is available to users. However, it does not yet cover all micro-architectural resources, which tends to be the hot spot under contention, such as cache, memory bandwidth, interconnect bandwidth on chip, and peripheral component interconnect express (PCIe).

(2) Namespace

As another fundamental mechanism, kernel namespace enables applications to run in their own isolated environment with separate process IDs, network devices, and file systems. As a concrete example, the network namespace is used to virtualize a network stack, where each namespace will have its private set of IP addresses, routing table, connection tracking table, and other network-related resources. Details can be found in Linux Community (2016).

(3) Intel's cache allocation technology (CAT)

For the last-level cache (LLC), Intel's CAT helps address resource-sharing concerns by providing software control of data placement, enabling isolation and prioritization of primary applications (Intel, 2016). Similar to Cgroups, it provides OS interfaces to group applications in classes of service (CLOS) and indicates the amount of LLC available to each CLOS via configuration. Currently, the CAT feature is available on all stock keeping units (SKUs) starting with the Intel Xeon processor E5 v4 family.

(4) Dynamic voltage and frequency scaling (DVFS)

Another widely used energy mechanism is DVFS for energy management. DVFS is a technique for altering the voltage and frequency of processors based on performance and power requirements. Because of Power $\propto FV^2$ (F represents the frequency and V represents the voltage), both dynamic voltage scaling and frequency scaling can be used to conserve power. By adjusting the frequency of a microprocessor (also known as CPU throttling) on the fly, it can be adopted to throttle the pressure posed to other shared resources such as memory bandwidth (Zhang X et al., 2013; Lo et al., 2015). It is also explored to achieve fine-grained voltage/frequency boosting to rein in tail latency (Hsu et al., 2015), because the current architecture lacks corresponding support.

2. Software-based enhancement

As a complement to kernel-supported isolation, software-based schemes employ task characterization to enable safe co-location of tasks via thread

scheduling, related to resource partitioning on chip-multi-processors (CMPs), especially those that currently lack explicit control.

Utility-based cache partitioning in Qureshi and Patt (2006) provides an allocation policy which attempts to repartition LLC among multiple tasks, and relies on the reduction of cache misses due to the adjustment. By controlling the replacement process, Vantage (Sanchez and Kozyrakis, 2011) proposes a fine-grained cache-partitioning scheme derived from a statistical model, which retains the same associativity as in non-partitioned cases. Cho and Jin (2006) proposed management of shared L2 cache banks through OS-level page allocation. In contrast to the fixed caching policy, this approach enables higher flexibility without any hardware support, and involves the private caching policy, shared cache policy, and a combination of both. Zhang X et al. (2009) relied on an optimized page-coloring method for cache management; the method partitions the physical address space to cache banks, and establishes the mapping from data to cache banks. However, it is limited to coarse-grained partition sizes. As for private caches, Feliu et al. (2013) proposed to estimate the L1 bandwidth requirements based on the positive correlation between L1 bandwidth and tasks' performance, and isolation at the L1 level was achieved by informed thread scheduling.

Summary As mentioned above, a wide range of resource types has been covered by isolation mechanisms supported by the OS (Table 2), and the isolation at finer-grained resources (e.g., L1) can also be enhanced via software optimization. The pros and cons in kernel- and software-based schemes are displayed in Table 3. Overall, several aspects are worthy of a complete understanding for further improvements.

First, interplay between shared resources is still opaque to the OS, especially under the increasing trend of resource heterogeneity.

Second, the efficiency of each mechanism lacks a systematic evaluation, such as the inertia quantification, as evidenced in Kasture and Sanchez (2014).

Third, software/hardware co-design is promising for efficient resource isolation. A modest hardware support can greatly simplify characterization of micro-architecture resource usage, and further facilitate resource management locally.

2.1.2 Isolation for GPU multitasking

Traditional GPUs represent a monolithic resource that cannot be shared across users. However, along with the wide deployment of GPUs in cloud infrastructure, GPU sharing and isolation among multi-tenants become increasingly important for efficiently using GPUs. Note that the kernel in the GPU context refers to an executable program on a GPU.

To support time multiplexing on GPUs, kernel preemption allows a time-sharing scheduler to execute multiple kernels fairly (e.g., by context switching like that employed by CPUs). Tanasic et al. (2014) allowed two preemption mechanisms, context switch and streaming multi-processor (SM) draining.

Table 2 Summary of supported resource controllers

Resource	Cores	Memory	Pid, net, etc.	LLC	Disk I/O	Network I/O	CPU bandwidth	Power
Controller	Cpuset	Cgroup/ Memory	Namespace	CAT	Blkio	Net_cls/Prio	Cpufreq	RAPL* and DVFS

Pid: process ID; I/O: input/output; LLC: last level cache; CAT: cache allocation technology; RAPL: running average power limit; DVFS: dynamic voltage and frequency scaling. *https://access.redhat.com/documentation

Table 3 Strengths and weaknesses of different isolation approaches

Approach	Strength	Weakness
Kernel-based	Efficient API (e.g., via system call) Effective resource isolation	Targeting a single resource Lacking interplay characterization or co-optimization knobs regarding different resources Lacking complete support of shared resources
Software-based	Informed isolation via a better understanding of performance impact of shared resources	Profile-based and requiring expert knowledge

API: application programming interface

The former hot-swaps kernel contexts in one SM with a new kernel via off-chip memory, whereas the latter first drains the whole SM before preemption happens. The authors also implemented a token-based scheduling policy that dynamically distributes the GPU SM among concurrently running kernels, accounting for the kernel queue and SM, but this policy comes with high preemption overhead (high memory traffic or long draining delay). To reduce the swap overhead, Chimera (Park et al., 2015) introduces "flushing," which drops running thread blocks (TBs) of one SM if the kernel is idempotent. Flushing combines with context switching and draining to collaboratively minimize throughput overhead and preemption latency. To enable GPU register time-sharing to achieve high warp concurrency, RegMutex (Khorasani et al., 2018) takes register demand fluctuation within warp into consideration, and enables on-demand register allocation. Specifically, it uses compiler analysis and primitive injection to determine the locations within the kernel to conduct register scale-up or scale-down via inter-warp register sharing. However, the time-sharing strategy still leads to GPU resource under-utilization due to context switching.

GPU vendors spend much efforts in supporting spatial multiplexing. For example, NVIDIA's multi-process service (MPS) (https://docs.nvidia.com) enables concurrent execution of kernels from different virtual address spaces in a GPU. The multi-instance GPU technology in A100 GPUs (https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html) allows static cache and memory bandwidth partitioning, which cannot adapt to dynamic demand at runtime, resulting in either over- or under-provisioning. Wang ZN et al. (2016) proposed the simultaneous multikernel (SMK) method, which achieves fine-grained resource partitions on GPU. Based on a partial context switching mechanism, SMK co-schedules kernels with compensating characteristics in the same SM. Specifically, it uses the dominant resource fairness (DRF) policy to partition static resources on one SM, such as registers and active threads, and computing cycles are allocated in proportion to the amount in solo execution (i.e., executed in an isolated environment). Thus, one dedicated SM is needed for profiling for each kernel. Similarly, Warped-Slicer (Xu QM et al., 2016) uses an online profiling-based analytic method to guide dynamic intra-SM slicing across different kernels. Concerning inference application, GSLICE (Dhakal et al., 2020) provides performance isolation through dynamic adjustment and apportioning just the right number of GPU threads for inference functions (IFs), while accounting for the respective monitored latency and throughput. It adopts hot-standby IF (shadow IF) on the CPU side to reduce startup delay when regaining GPU resources, and ensures coordinated transition with the previous one.

Targeting the virtual memory mechanisms that impede multitasking in modern GPUs, MASK (Ausavarungnirun et al., 2018) provides architectural support to mitigate high contention for shared address translation structures (translation lookaside buffer, TLB) among multiple applications. Specifically, it designs a TLB miss rate based token assignment policy to regulate the number of warps of each application that can use the L2 caches, thus reducing TLB misses and the associated long-latency stall. For cases with TLB misses, MASK adopts a page-table hit rate based policy to guide address translation requests that bypass the L2 cache, reducing the queueing delay there. In contrast, Pratheek et al. (2021) designed a dynamic page walk system that partitions the shared pool of walkers among the applications to achieve spatial multiplexing. In addition, this system enables walk stealing between page walkers that belong to different applications, following several carefully designed stealing policies to coordinate the instruction throughput and fairness.

Apart from GPUs, other works aim at sharing FPGAs, such as AMORPHOS (Khawaja et al., 2018), Optimus (Ma et al., 2020), and ViTAL (Zha and Li, 2020). AMORPHOS enables applications (i.e., user FPGA logic) to scale dynamically according to load and to be remapped to the physical fabric to increase utilization. Optimus offers both spatial multiplexing (via page table slicing) and temporal multiplexing (via application preemption) for efficient sharing of each application in an FPGA. We leave corresponding investigation to interested readers.

Summary GPU multitasking involves many resources and control components. Prior efforts often focused on individual resources (as summarized in Table 4). This is critical for a certain application set, but ignoring underlying inter-resource interactions.

Thus, more integrated mechanisms are required in both hardware and software to curb performance interference due to resource contention, while maximizing its computing advantages.

2.1.3 Isolation at the virtualization level

Apart from the isolation mechanisms supported by hardware, VM- and container-based techniques are also widely deployed to encapsulate applications in a multi-tenant environment.

1. VM-based

VM-based isolation relies on the virtualization layer (e.g., hypervisor) to interact with a host operating system (i.e., hosted virtualization, see Fig. 1a) or to manage all hardware directly (i.e., bare metal virtualization, in Fig. 1b), but both modes partition physical resources in terms of virtual resources across guest operating systems (i.e., VMs).

With supervisory privileges over the entire machine, the hypervisor supervises and multiplexes multiple VMs, and seeks to enforce strong separation policies and finite boundaries so that VMs can operate separately. However, the induced overhead is so high (e.g., long VM boot time) that consolidation capacity is limited. To deal with it, unikernel-based virtual machine (UKVM) (Williams and Koller, 2016) implements a specialized unikernel monitor on top of KVM and uses MirageOS unikernels to achieve 10 ms boot time. LightVM (Manco et al., 2017) transforms the centralized control of Xen to distributed control and uses unikernels to achieve fast instantiation for specialized applications (as little as 2.3 ms), small per-instance memory footprints, and high density on a single machine. It presents the possibility of simultaneously retaining good isolation and performance on par or better than containers. To alleviate the overhead of current live update methods (e.g., kernel live patching and VM live migration), Orthus (Zhang XT et al., 2019) adopts the VM monitor live upgrade technique, which relies on VM grafting between two KVM kernel instances and device migration support (e.g., GPU passthrough) to update the whole VM monitor in real time. However, this method is ill-suited for type-1 hypervisors like Xen.

2. Container-based

Container-based isolation relies on kernel features (e.g., Cgroup, namespace) to create an isolated environment for processes, instead of a complete OS. Therefore, containers do not require installation of separate guest operating systems, but share the hardware and host OS kernel with each

Table 4 Targeted resource for partitioning in GPU and related research

Resource type	Research
Streaming multi-processor (SM)	Context switch & swap (Tanasic et al., 2014), Chimera (Park et al., 2015)
Intra-SM resources	SMK (Wang ZN et al., 2016), Warped-Slicer (Xu QM et al., 2016)
Register	RegMutex (Khorasani et al., 2018)
Translation lookaside buffer (TLB)	MASK (Ausavarungnirun et al., 2018)
Cache and memory bandwidth	MIG (https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html)
Page table walker	Walker (Pratheek et al., 2021)

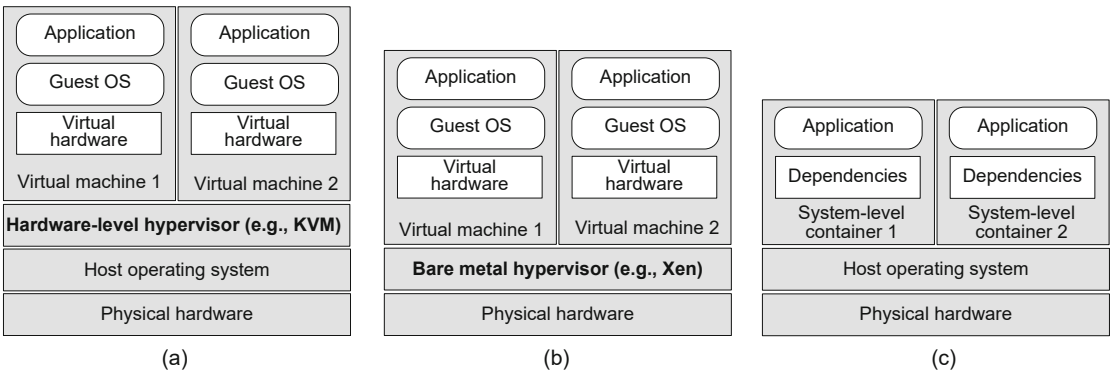

Fig. 1 Hardware-level hypervisor (a, b) vs. system-level container (c). Modified from Weerasiri et al. (2017), Copyright 2017, with permission from ACM

other, as Fig. 1c depicts. In contrast to heavyweight, hypervisor-based technologies, containers provide extremely fast instantiation, small per-container memory footprint, and consequently high density on a single machine (Manco et al., 2017). However, flexibility is accompanied by weaker isolation and security vulnerabilities compared to VMs.

To further satisfy the tight constraints in startup delay and resource usage of container-based serverless models, Boucher et al. (2018) proposed to restructure the serverless model using shared-process execution and language-based compile-time isolation guarantees. The merits of Rust language (a type-safe system-programming language) was exploited to provide security and isolation when consolidating multiple jobs into a single process. The invocation delay under language-based isolation is at least one magnitude better than that under process-based isolation, which facilitates finer-grained resource accounting and scheduling. With the same aim, SAND (Akkus et al., 2018) adopts two isolation levels, wherein different applications are isolated using containers, but concurrent invocations within the same application are isolated only by processes.

In contrast, to allow applications that require kernel customization to run in containers, X-Containers (Shen et al., 2019) provide a library OS execution environment to support concurrent multi-processing. They transform a system call to a function call without hardware-assisted virtualization, which improves both container isolation and application performance, but they take longer time to boot and have a larger memory footprint. For data-intensive serverless computing, Faaslets (Shillaker and Pietzuch, 2020) adopt a new isolation abstraction to coordinate with efficient state sharing across functions. On one hand, Faaslets leverage the software-based fault isolation provided by WebAssembly and Linux Cgroups, and offer a memory safety guarantee and resource isolation. On the other hand, Faaslets adopt a two-tier state architecture to support efficient local/global state access among functions.

Apart from container-based isolation between functions, Nightcore (Jia and Witchel, 2021) uses optimizations including a fast path for internal function calls, low-latency message channels for inter-process communication (IPC), efficient threading for input/output (I/O), and function executions with dynamic concurrency to eliminate invocation overheads (<100 μs). However, Nightcore assumes that an individual worker server can hold most function containers from a single application, which results in lower scalability for large-scale applications such as microservices.

Summary These two technologies provide system-level schemes to guarantee an execution environment and resource isolation, but both come with corresponding tradeoffs that have to be considered with respect to application-specific requirements. Table 5 displays a comparison of isolation solutions in serverless computing. In the context of serverless computing, strict requirements for function startup delay motivate the optimization and evolution of both. For example, the 100-ms startup delay is expensive for microservices, which typically have sub-millisecond demand. Beyond virtual machines and containers, other works have proposed the use of minimalistic kernels to provide lightweight virtualization, like unikernel (Madhavapeddy and Scott, 2014; Cadden et al., 2020) and exokernel (Engler et al., 1995), which are also included in Table 5.

Table 5 Isolation approaches for serverless computing (Shillaker and Pietzuch, 2020)

Target	VMs	Container	Unikernel
Resource isolation	Yes	Yes	Yes
Memory safety	Yes	Yes	Yes
Efficient state sharing	No	No	No
Shared filesystem	No	Yes	No
Startup delay	100 ms	100 ms	10 ms
Memory footprint	MBs–GBs	MBs	KBs

2.2 Resource management at the machine level

Apart from the aforementioned isolation techniques at the micro-architecture level, contention management through software-based approaches at the system level has also been widely explored. Task (or thread) scheduling optimizations targeting different resources, heterogeneous OS design, and inter-VM performance coordination are discussed.

2.2.1 Scheduling against hardware resources

Along with the evolution of modern hardware structures, such as multi/many-core processors, simultaneous multi-threading (SMT) technology, and

non-uniform memory access (NUMA), new optimization methods are proposed.

1. Simultaneous multi-threading (SMT)

To overcome the symbiosis analysis overhead, Eyerman and Eeckhout (2010) adopted an analytic model that relies on the cycle per instruction (CPI) stack of each application to predict the performance loss when co-located. A CPI stack breaks down the execution cycles into various components and quantifies how much time is spent due to each event. The model interprets the normalized CPI components as probabilities, and calculates the probabilities of events that cause interference. However, the inputs to the model are generated using complex new hardware, which is not supported by current processors. In comparison to the CPI approach, Zhang YQ et al. (2014) pointed out that the interference on different shared resources does not correlate with each other, such as private caches and memory ports. This motivated them to decouple interference management into multiple dimensions, in which the sensitivity and contentiousness are separately captured. Finally, a prediction model combining all dimensions was adopted to guide task co-location. The model involves an offline profiling phase that undermines its adaptability.

To improve the utilization and throughput per core with likely contention in mind, Snavely and Tullsen (2000) designed an informed scheduler that selects the best thread combination to co-schedule on one core based on a greedy exploration. The defect is obvious; the overhead increases exponentially with the number of cores and tasks. Similarly, Feliu et al. (2016) proposed a regression-based symbolistic scheduling algorithm that uses performance counters. Combined with the corresponding measures collected in single threaded (ST) mode, the performance of co-located tasks was inferred and corresponding scheduling was applied. Feliu et al. (2016) also designed a transform method that infers the measure in ST mode from that in SMT, which avoids the offline overhead.

2. Memory subsystem resources

Instead of methods targeting SMT, Tang LJ et al. (2011) demonstrated the impact of memory resource sharing (e.g., cache and memory bus) for DC applications. They presented a heuristics-based thread-to-core mapping technique that enables threads to run simultaneously in a constructive way, e.g., minimizing coherence traffic. As for memory bank/bus/row-buffer conflicts, Mutlu and Moscibroda (2008) proposed a parallelism-aware batch scheduling (PAR-BS) technique that combines batch dynamic random access memory (DRAM) request processing and parallelism-aware scheduling policy to provide fairness, starvation avoidance, and reduced memory-related stall-time of a given thread. In addition, PAR-BS seamlessly incorporates support for system-level thread priorities and supports opportunistic service (Mutlu and Moscibroda, 2008).

A key enabling factor of contention management of shared resources is finding a metric that can reflect their behavior consistently. To this end, Zhuravlev et al. (2010) proposed a classification-based thread-to-core mapping algorithm through a comprehensive analysis of related classification methods, including stack distance competition (SDC) (Chandra et al., 2005) and solo miss rate (Knauerhase et al., 2008). As a result, they found the solo LLC miss rate to be the most accurate contention predicator for capturing memory controller contention behaviors and prefetching-related resources. Based on this insight, they further designed an online algorithm at the user level, distributed intensity (DI), to minimize overall performance degradation.

3. Non-uniform memory access (NUMA)

The above optimizations try to maximize benefits by identifying symbiotic tasks in the context of uniform memory access (UMA) architecture. However, in the NUMA context, an NUMA-agnostic scheduler fails to eliminate contention for resources like inter-domain interconnect. Toward the goal of NUMA-awareness, Blagodurov et al. (2010) proposed the distributed intensity NUMA online (DINO) algorithm based on the work by Li JL et al. (2014). DINO maintains local memory access via memory migration when threads move across NUMA domains, and adopts a coarse-grained migration-triggering scheme to reduce unnecessary migrations. Furthermore, migration size strategies were evaluated to alleviate interconnect pressures. Similar work can be found in Goglin and Furmento (2009). Instead of single-bottleneck optimization, Popov et al. (2019) explored the combined optimization space of thread scheduling, data mapping, NUMA degree (i.e., the number of NUMA nodes used), and parallelism, based on reproduced interaction between system and application. In

comparison, Mitosis (Achermann et al., 2020) mitigates the NUMA effects on page table walks by transparently replicating and migrating page tables across sockets.

In short, NUMA-related scheduling optimizations are concerned mainly with thread-data affinity via memory migration. Prior research efforts focus on when, which, and where memory pages are to be migrated, so that NUMA-agnostic overhead is alleviated.

4. Accelerators

Here, an accelerator refers to a hardware-specialized device to accelerate the processing of certain functions, such as GPU for graph processing, and neural network processing units (NPUs), TPUs for deep neural network inference acceleration. They are often coupled with a host processor via a PCIe bus. To make best use of them, workloads are often split and offloaded to them.

To make full use of SmartNIC, UNO (Le YF et al., 2017) is proposed as a SDN-controlled (software-defined network controlled) network function (NF) offload architecture, which works independently from central control. It employs an integer linear programming formulation to locally determine the subsets of switching and NFs offloaded to SmartNIC to minimize host resource usage while accounting for PCIe bandwidth limitations. It allows NF migration by periodically re-running the placement algorithm based on updated states. If the new solution is better (i.e., with lower host resource utilization), it identifies the set of NFs by resolving a graph cut problem, and offloads them to SmartNIC. For real-time deep neural network (DNN) model inference jobs, Fowers et al. (2018) offloaded workloads to a specialized FPGA to cater to the low latency and model flexibility requirement.

Nowadays, GPU has become one of the most popular options for deep learning applications. However, on-board GPU memory can barely satisfy the increasing requirement for training. To tackle this issue, vDNN (Rhu et al., 2016) chooses the input of the convolutional layer as a swapping target to CPU memory, and expects to increase the overlap of swapping with convolutional layer computation. SuperNeurons (Wang LN et al., 2018) also avoids recomputation of the convolutional layer to reduce overhead. However, it performs layer-wise GPU memory management based on computation graph analysis prior to execution, which is coarse-grained with high operational overhead. In comparison, Capuchin (Peng X et al., 2020) performs tensor-granularity swapping to CPU memory or tensor recomputation to reduce the GPU-side memory footprint. It tracks the tensor access pattern during training to identify tensors to be swapped (e.g., after an access to itself). In addition, it decides which optimization technique to perform by estimating the computation time and swapping time.

To coordinate GPU under-utilization and application QoS, Baymax (Chen Q et al., 2016) pre-trains regression models to predict job execution time. Based on predictions, it re-orders tasks issued to the accelerator, accounting for PCIe bus contentions, thus preventing latency-sensitive jobs from missing QoS targets. Similarly, Prophet (Chen Q et al., 2017) uses offline profiling and prediction models to co-locate kernels for higher GPU utilization and QoS. However, both of these models rely on CPU-side software schedulers, which are insensitive to dynamic behaviors inside the GPU or have inevitable host-device overhead. For tasks with a smaller degree of parallelism, Pagoda (Yeh et al., 2017) relies on a MasterKernel running on the GPU side to support concurrent GPU task scheduling on an SM (streaming multi-processor) and pipelined task spawning, scheduling, and execution. It leverages TaskTable with lazy update to coordinate task spawning and scheduling between CPU and GPU with reduced handshaking overhead. Lax (Yeh et al., 2021) adopts a GPU offloading scheme for latency-sensitive jobs. This scheme designs a laxity-aware stream scheduler that takes into consideration the jobs' remaining execution time, deadlines, and per-kernel work completion rates, and relies on stream inspection to adjust job priorities dynamically for GPU processing. In contrast to other efforts, Lax achieves concurrent kernel scheduling at the microsecond timescale.

Targeting GPU servers consisting of multiple GPUs, various scheduling methods have been proposed to distribute GPU resources between applications. Rain (Sengupta et al., 2013) adopts a two-layer scheduling architecture comprising a server-level balancer and a GPU-level scheduler. The former distributes workload across GPUs based on the load value per GPU. The latter multiplexes individual GPUs among applications. Similarly, Strings (Sengupta et al., 2014) adopts a metric (least

attained service) to achieve load balancing across GPUs, but co-locates applications with complementary behaviors on the same GPU. However, the estimation of resource usage before the execution of one application is inadequate due to the varied GPU utilization. As a complement to the above approaches, DCUDA (Guo et al., 2019) supports live migration of running applications from overloaded GPUs to underloaded GPUs, while maintaining the same runtime environment and associated data.

Summary First, I/O performance tends to be a bottleneck in computation offloading models, due to factors like an agnostic PCIe network topology (corresponding characteristics) and asymmetric performance between the host and accelerators. Efficient I/O control should sidestep the host CPU, allowing accelerator access to the I/O device directly. Second, the two-layer scheduling architecture with a well-designed cross-layer coordination is promising for servers with multiple accelerators, which could guarantee resource utilization and application QoS simultaneously.

2.2.2 Operating systems for heterogeneous computers

The end of Dennard scaling and the dark silicon effect are driving the development of specialized, power-efficient processor architectures. The heterogeneous computer is becoming a new norm to support wider applications. In this subsection, we investigate research on OS designs targeting different types of hardware.

1. Multi-core processor

For managing multi-processors with heterogeneous CPU cores, Helios (Nightingale et al., 2009) establishes a uniform set of OS abstractions, with minimal hardware primitives called satellite kernels running on each CPU core, to perform resource management independently. It applies an extended message-passing scheme to achieve inter- and intra-kernel communication, and introduces an affinity value to assist in placement of applications, accounting for their dependency or interference. Helios is also well-suited for programmable devices, such as GPUs and network interface cards (NICs). Similarly, Barrelfish (Baumann et al., 2009) adopts a multi-kernel design, acting as a distributed system to support heterogeneous cores. Another microkernel-based OS, M3 (Asmussen et al., 2016), integrates cores and memories into a network-on-chip (NoC) and uses a per-core data transfer unit (DTU, a hardware component) to abstract away the heterogeneity, and thus allows uniform control of different cores. Specifically, isolation is enforced per kernel at the NoC level by controlling respective DTUs, instead of within the core. In theory, the design of M3 is general for any kind of processor, such as GPUs or FPGAs. Tangram (Pothukuchi et al., 2019) is a similar spirit. It explores integrated resource control in heterogeneous computers via inter-controller coordination in different subsystems (e.g., CPU and GPU).

2. Memory

Current systems tend to combine different memory technologies for balanced optimization of latency, bandwidth, and cost. For heterogeneous memory management (e.g., 3D-stacked DRAM and non-volatile memory) in a virtualized environment, HeteroOS (Kannan S et al., 2017) exposes memory heterogeneity to a guest OS, and combines an application's memory usage pattern to provide OS-level memory placement. In detail, it employs page type-aware replacement (e.g., I/O or heap page) and DRF-based memory type-aware resource sharing algorithm to avoid expensive migrations. Conversely, the guest OS also coordinates with the virtual machine manager (VMM) with application-specific information for page hotness tracking. Currently, HeteroOS lacks page type specific promotion/demotion policies for multi-level memory, and the software approach for hotness tracking has high overhead. Chameleon (Kotra et al., 2018) adopts OS-hardware co-design where two new processor instructions are introduced for the OS to communicate with hardware, which dynamically reconfigures regions in heterogeneous memory (i.e., stacked DRAM) between part of OS-visible memory for high memory footprint workloads and cache modes to adapt changing workload behavior. KLOCs (Kannan S et al., 2021) present a new OS abstraction to systematically group kernel objects with similar hotness, reuse, and liveness, achieving efficient tiering. Specifically, KLOCs leverage data structures extended with a list of pointers to knodes touched by each CPU to quickly ascertain kernel object page hotness/coldness without page table walking. In practice, KLOCs aim to increase direct allocations of kernel objects of an active knode to fast memory, reducing slow-to-fast memory migration.

3. GPU

To move the GPU into the direct control of the OS, Rossbach et al. (2011) claimed that modern OSs lack abstractions for supporting interactive applications that use GPUs. Because the traditional input/output control (ioctl) oriented interface hinders direct OS management of GPU, the resulting excessive data movement across the user/kernel boundary limits the full potential of GPU. Thus, Rossbach et al. (2011) proposed a new set of OS abstractions for GPUs (and other accelerators) called the PTask API. PTask promotes GPUs from I/O devices to a general-purpose, shared compute resource, managed by the OS, instead of vendor-supplied drivers and user-mode runtimes. To simplify task offloading, PTask adopts a dataflow programming model to organize PTasks in the form of a directed acyclic graph, which is coordinated by the OS scheduler with guaranteed fairness and isolation. In contrast, dCUDA (Gysi et al., 2016) enhances the fork-join offloading model in the compute unified device architecture (CUDA) with remote memory access capabilities in the massage passing interface (MPI), providing an abstraction for overlapping remote memory accesses with concurrent computation. Similar work can be found in rCUDA (Castelló et al., 2018), which provides a virtualization layer for inter-GPU operations in a cluster.

Solros (Min et al., 2018) presents a data-centric OS architecture, which delegates I/O services of accelerators (e.g., GPU) to the host processor, facilitating coordination among accelerators and I/O devices based on system-wide knowledge. Specifically, it exploits characteristics of the PCIe network to transfer data among accelerators, and employs a ring buffer to reconcile the high concurrency of accelerators. Solros designs a file system service to decide whether to perform peer-to-peer communication or host-side buffered I/O, and provides load balancing for the shared listening socket from multiple accelerators by rule-based packet forwarding. More efforts with respect to GPU virtualization can be found in Hong et al. (2017).

4. Field-programmable gate array (FPGA)

FPGAs enable tremendous improvements in performance and energy efficiency for many applications, but remain difficult to program, deploy, and securely manage due to a lack of a unified abstraction. Feniks (Zhang JS et al., 2017) provides abstracted interfaces for FPGA accelerators (i.e., functions off-loaded to FPGA) while hiding underlying hardware details. It exploits the memory-mapped PCIe configuration space address to support direct access to other devices over PCIe. Resources on other servers can also be accessed based on a remote direct memory access (RDMA) NIC through cloud networks. Cloud users can request FPGA resources based on application requirements via a centralized controller. Korolija et al. (2020) discussed the challenges for OS design on FPGAs compared with other processor architectures, like GPUs. They proposed Coyote, which combines a tight set of OS abstractions in a single unified runtime for FPGA-based applications. It provides essential functions on which other services reside, including virtual memory management, an analog of software processes or tasks for user logic, scheduling, etc. Specifically, it relies on a runtime manager on the host CPU to schedule tasks across virtual FPGAs (vFPGAs) or by reconfiguring a vFPGA if required. The kernel driver undertakes the handling of TLB misses in a vFPGA and the allocation of physical memory by creating virtual memory mappings for the user logic and application CPU code. Zha and Li (2021) presented a hybrid abstraction scheme that provides a homogeneous view of the heterogeneous cloud FPGAs to substantially reduce the resource management complexity. It is instructive for FPGA OS design.

Summary Typically, accelerators run as peripheral devices that are connected with the host CPU via PCIe, and lack the resource management support in conventional OSs. As Fig. 2 depicts, OS-level resource abstractions like process, interprocess communication (IPC), and file service are absent in a GPU, which will impede its adaptation scope. The same applies to other accelerators. Prior efforts have created and applied proposals in OSs to manage resources in accelerators, but often focus on individual resource abstraction and lack integrated solutions.

2.2.3 Virtualization level

In a virtualized environment, there is a symbiotic problem in VM scheduling. Xu YJ et al. (2013) noted that a non-complementary VM-based workload pattern on shared processors can cause long-tail latency. The key component underlying this problem is the VM scheduler.

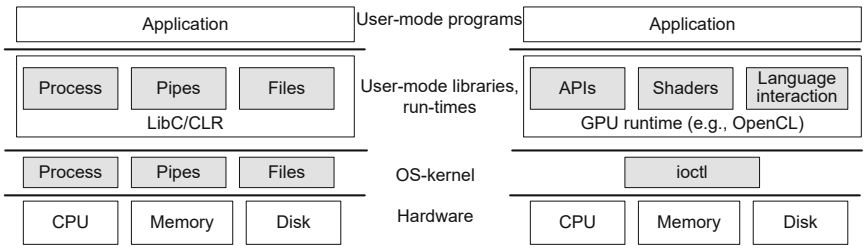

Fig. 2 Technology stacks for CPU vs. GPU programs (OS-level and user-mode runtime abstractions for GPU programs are absent). Modified from Rossbach et al. (2011), Copyright 2011, with permission from ACM

Cherkasova et al. (2007) performed an in-depth analysis and comparison of three CPU schedulers in Xen, and verified that the choices of VM scheduler and corresponding parameter configuration are closely related to application performance. As for NUMA, virtualization poses additional challenges in user-level optimization, due to the disconnect between NUMA topology and the guest OS. To deal with this issue, Rao et al. (2013) incorporated NUMA-awareness into VM-level scheduling relying on a hardware metric (called uncore penalty) to dynamically adjust the vCPU-to-core assignment. Targeting the runtime overhead, Tableau (Vanga et al., 2018) designs a table-driven VM scheduler that enables high VM-density while maintaining a predictable scheduling delay for every VM. Tableau adopts a decoupled design, wherein the table generator takes system-wide information into consideration, such as when VMs are created, torn down, or reconfigured, and feeds the generated table to the vCPU dispatcher for resource assignment. Currently, Tableau concerns merely CPU scheduling, regardless of other sources which contribute to performance variability, such as cache interference. Alternatively, Zhao M and Cabrera (2018) proposed a cross-layer scheduling architecture to support communication and coordination between host- and guest-level VM schedulers, which enables dynamic time-sensitive computing.

In short, with a higher layer, VM scheduling focuses on CPU multiplexing among VMs, and pays less attention to micro-architecture-level resources and the performance impact.

2.3 Interference detection and performance correction

In addition to the interference awareness in the task scheduling phase, interference detection and performance correction at runtime are important in the shared cloud. We introduce related research in this subsection.

1. Profile-based

Program profiling is a form of dynamic program analysis that measures the space (e.g., memory requirements) or time (e.g., execution time) complexity to provide insight into execution behaviors. Here, aggregated profiling information (e.g., performance model) guides performance compensation. Bitirgen et al. (2008) adopted an ensemble of artificial neural networks (ANNs) to model each application's performance as a function of underlying resource states. The model was employed at runtime to guide adaptive resource allocation. However, model training overhead was prohibitively high with increasingly diverse applications.

In contrast, Dirigent (Zhu HS and Erez, 2016) monitors the LS execution process by sampling the runtime status in the form of ⟨time stamp, retried instruction⟩. To predict the actual progress, data are compared against offline records, and the actual progress is predicted. Subsequently, resources are reallocated between LS and BE jobs via DVFS and Intel CAT. Subramanian et al. (2015) designed an online slowdown estimation model, which takes the cache access rate as a performance proxy and estimates corresponding measurement of the application in solo mode. This model can be further used to guide cache and memory bandwidth partitioning in a slowdown-aware manner, but it needs additional registers to store the collected data.

2. Feedback-based

Generally, feedback refers to a closed-loop control procedure, where the latest monitored signal (i.e., performance proxy) is fed back periodically, and compared with a predefined value to adjust any deviations. Thereby, enforcement can be carried

out to ensure that the task's performance varies within an acceptable range (Fig. 3). Ebrahimi et al. (2010) relied on a hardware-based mechanism that continuously monitors and estimates slowdowns of co-running applications, and accounts for multiple shared resources (cache, DRAM bus/bank/row-buffer). Given this estimation, memory request rates were adjusted collectively to restore fairness.

Following the same spirit of coordination, Lo et al. (2015) designed a feedback-based performance control scheme, Heracles, which conducts coordinated management by relying on four software/hardware mechanisms to guarantee LS performance and resource utilization simultaneously. These mechanisms include core isolation using Cgroups/cpuset, CAT and power-related DVFS, and system-on-chip running average power limit (RAPL). As for network bandwidth, flow controlling tools (e.g., TC) are adopted, as will be described in Section 2.4. Alternatively, based on the monitored cache miss rates and bus contention, Bhadauria and McKee (2010) proposed a feedback-based scheduler to enable application-to-core mapping reconfiguration. Zhang X et al. (2013) (Google) adopted the circle per instruction (CPI) as the interference proxy, but cross-node CPI collection and distribution were entailed at the cluster level.

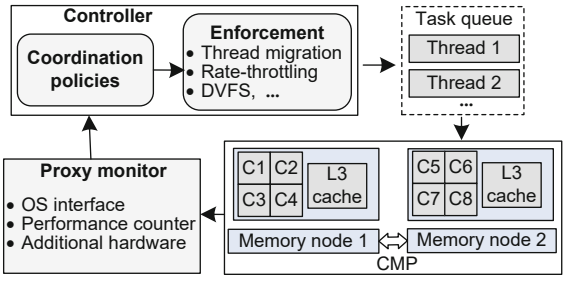

Fig. 3 Overview of feedback-based mechanisms (the monitor collects resource states and application performance data, which are fed to the control plane; the controller performs resource adjustment according to a certain policy). DVFS: dynamic voltage and frequency scaling; OS: operating system; CMP: chip-multi-processor

In contrast, Merlin (Tembey et al., 2014) attempts to manage resource shares in a more holistic manner, involving LLC, a memory bandwidth/controller, and interconnects. Briefly, it relies on a performance counter to approximate usage (e.g., approximating L3 usage as L2 misses−L3 misses). These metrics are tracked as proxies for resource reconfiguration on the fly, wherein Merlin always applies lowest cost reallocation, including CPU capping, vCPU migration across NUMA nodes, and memory page migration. To address the same problems in a virtualized environment, Zhu T et al. (2017) adopted token-bucket rate-limiting to consolidate workloads by simultaneously considering the r-b curves (r refers to the rate limit and b refers to the bucket size) for all workloads. Particularly, r and b can be dynamically reconfigured to accommodate varied interference intensity, but only network bandwidth is taken into consideration.

Summary A comparison of the above approaches is presented in Table 6. Overall, several challenges remain. First, the effectiveness of different hardware and software control varies, and is often application-dependent. For example, cache warmup usually takes tens of milliseconds to take effect, whereas DVFS costs only several microseconds (Hsu et al., 2015). Second, because contention shifting across resources exists due to workload dynamics, an integrated strategy is required to deal with it. Thus, the strategy space for different mechanism combinations must be explored to cater to application diversity. Third, hardware performance counters are proved effective in capturing the complex interaction between hardware and applications, and can drive performance enhancement.

2.4 Network-related optimization

In this subsection, we review network-related isolation methods in the data center, including networking acceleration and bandwidth isolation.

Significant effort has been devoted to achieving

Table 6 Strengths and weaknesses of performance compensation methods at the machine level

Approach	Strength	Weakness
Profile-based	High accuracy Good compensation quality	Time-consuming and low scalability
Feedback-based	Sensitivity to performance variance Good effectiveness	High overhead due to continuous monitoring

networking acceleration. One line of inquiry aims at moving networking to the user space (Kapoor et al., 2012; Jeong et al., 2014; Kalia et al., 2019), whereas other research offloads networking to specialized adapters (Kalia et al., 2016; Ibanez et al., 2019; Moon et al., 2020), or employs SmartNICs to satisfy the performance requirements of various applications by tuning network configurations (Firestone et al., 2018; Liu et al., 2019).

To highlight contributions to end-to-end application latency of different components in the network stack, Kapoor et al. (2012) thoroughly analyzed the latency sources (Table 7). It is clear that kernel latency constitutes a large portion of end-to-end application latency (86%–95%). To eliminate the latency overhead due to kernel processing, Chronos (Kapoor et al., 2012) moves request handling logic out of the kernel to the user space, and relies on zero-copy, kernel-bypass network APIs, which are supported by commodity server NICs. Jeong et al. (2014) built a user-level TCP stack that translates multiple expensive system calls into a single shared memory reference, and integrates packet- and flow-level event batching. Kalia et al. (2019) presented eRPC, which uses user space networking and polling to eliminate overhead due to interrupts and system calls from the datapath. In addition to performance improvement, the above schemes are still subject to system overhead because of the software-only and CPU-based implementation.

Co-designing distributed systems with network hardware (e.g., RDMA and FPGA) is a well-known technique to improve performance. FaSST (Kalia et al., 2016) designs an all-to-all remote procedure call (RPC) system by using RDMA's datagram transport, which bypasses the remote CPU to reduce the round-trip time. Different from FaSST's remote CPU bypassing, L-NIC (Ibanez et al., 2019) aims to bring a message into CPU as fast as possible. Thus, it co-designs NIC and CPU by directly placing packet data into a CPU register, bypassing the cache hierarchy. Instead of RDMA, AccelTCP (Moon et al., 2020) presents a dual-stack TCP design, which partially offloads stateful TCP operations (e.g., connection setup and teardown) to the NIC stack, thus saving the compute cycles and memory bandwidth.

As the network bandwidth in the data center grows continuously and new application paradigms emerge, there is increasing computational demand. Consequently, apart from the multicore system on chip based (SoC-based) NIC techniques (as discussed above), more research efforts are focusing on customized hardware (e.g., FPGA) for higher throughput and lower latency. AccelNet (Firestone et al., 2018) offloads host networking to an FPGA-based SmartNIC, which embraces both software-like programmability (e.g., supporting SDN workload over time) and hardware-like performance. Targeting microservices with sub-millisecond RPC latency, Dagger (Lazarev et al., 2021) not only offloads the entire RPC stack to the FPGA-based NIC, but also adopts a new communication interface with the host CPU via memory interconnects instead of PCIe buses, and migrates the overhead of the latter, such as memory synchronizations.

As another aspect of a DC network, network bandwidth isolation at the NIC level is critical, because traffic congestion would adversely affect shared environments and undermine DC availability. The kernel-supported traffic control mechanism, the qdisc scheduler (http://linux-ip.net/articles/Traffic-Control-HOWTO), can be

Table 7 Breakdowns of latency sources in data center application (Memcached is used)

Component	Description	Mean latency (μs)	P99 latency (μs)	Overall share
Data center Fabric	Propagation delay	<1	–	–
	Single switch	1–4	40–60	1%
	Network path	6	150	**7%**
Endhost	Network serialization	1.3	1.3	1.4%
	DMA	2.6	2.6	3%
	Kernel (including lock contention)	**76**	**1200–2500**	**86%–95%**
Application	Memcached	2	3	2%
	Total latency	88	1356–2656	100%

The best results are in bold. P99 means the 99^{th} percentile

used to enforce bandwidth limits for outgoing traffic. For example, Heracles (Lo et al., 2015) adopts the hierarchical token bucket (HTB) queueing discipline to limit the maximum traffic burst rate for BE jobs. However, ingress network bandwidth management is not considered. In contrast, EyeQ (Jeyakumar et al., 2013) provides a protocol-independent solution based on end-to-end feedback methods. It relies on the feedback messages from receiving ends to adjust the bandwidth allocation between entities (e.g., VMs) in the transmitting ends. Thus, the convergence may be delayed due to stale message or loss caused by congestion. Similarly, NUMFabric (Nagaraj et al., 2016) adopts two logical layers to maximize network utilization and achieves optimal bandwidth allocation across competing flows. Specifically, the upper layer manages information communication among sources (or application flows), and dynamically computes the weights for each flow. The bottom layer carries out network-wide rate allocation, and relies on both a weighted fair queuing (WFQ) based packet scheduling scheme at the switch and rate control at the host. It provides a unified framework that allows an operator to optimize for different service-level objectives (e.g., weighted fairness), and minimize flow completion times. We can see that the NUMFabric design follows a network-wide coordination principle that involves the switches and end hosts. In contrast, Sharma NK et al. (2020) proposed a packet scheduler based on calendar queue to support adaptive priority at the switch level, facilitating the adoption of multiple scheduling policies with improved inter-flow fairness.

Summary First, along with an increasing network speed, the network stack contributes significantly to end-to-end latency. Consequently, kernel-bypass and CPU-bypass methods are proposed. Second, to save more CPU and memory bandwidth for applications, multicore SoC- and FPGA-based NIC design techniques draw more attention, and often come with a hardware/software co-design principle that unleashes benefits of customized hardware for network acceleration. Third, network bandwidth allocation is a cluster-wide optimization problem that involves each interconnected machine. The feedback-based end-to-end bandwidth isolation schemes are often faced with long time to converge, especially for shorter tasks. Software-defined network architecture provides a promising approach, which enables global coordination with various policies (Panda et al., 2017; Zhou ZY and Benson, 2019). We leave corresponding exploration to interested readers.

In Sections 2.2 and 2.3, we investigated scheduling-related work within a single machine. In the next section, we step up to the cluster level and investigate research efforts regarding resource management coordination.

3 Cluster-level resource management

First, we focus on RMS scheduling architecture and algorithm design to deal with different issues, such as scheduling delay and scheduling quality, guaranteed QoS, and high resource utilization.

3.1 Scheduling architectures

In recent years, scheduling architectures quickly evolved, and several design models and corresponding systems have been developed (Fig. 4).

1. Centralized architecture

In a centralized architecture, the scheduler uses a single, integrated scheduling logic for all applications (Mars and Tang, 2013; Verma et al., 2015).

As the first container-management system in Google, Borg (Verma et al., 2015) relies on Borgmaster to maintain cluster-wide states, and is responsible for selecting resources to meet job constraints. It supports co-locating jobs with different priorities,

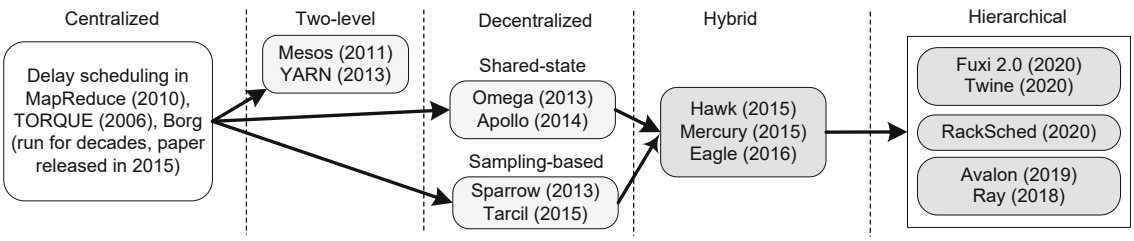

Fig. 4 Architectural shift of scheduling systems

and the performance coordination is achieved by priority-aware preemption. Additionally, it uses equivalence classes for task performance/resource requirements and node score caching to reduce scheduling delay (i.e., the waiting time before obtaining resources). Generally, the centralized mode favors the deployment of global policies. However, the drawbacks lie in the head-of-line blocking effect (which causes short tasks that are behind long ones to experience a long scheduling delay due to resource starvation), and feeble functionality for handling application diversity.

Notably, faced with this situation, Borg embraces the decentralized principle in Omega (Schwarzkopf et al., 2013), and allows the scheduler to be split into multiple replicates, with each working on a local state copy. The state updates on the local and master sides are coordinated by the Paxos algorithm. In some way, this architecture shift reflects the requirement for new scheduling techniques. Stemming from Borg and Omega, the current mainstream of "cloud-native" container platforms, Kubernetes (https://kubernetes.io/), also adopts a centralized architecture, but with a distinct design paradigm, known as microservices. In other words, multiple separate, autonomous components are combined and collaborate to achieve a desired behavior. For example, the control plane comprises multiple controllers, with each managing a particular aspect of the cluster state by interacting with the cluster API server.

For microservice applications in the cloud, centralized scheduling architecture is widely adopted to maintain a global view of the microservice graph and anticipate the impact of dependencies on end-to-end performance. Examples include the systems in Kakivaya et al. (2018), Qiu et al. (2020), and Zhang YQ et al. (2021). The same applies to heterogeneous computing platforms and GPU cluster for DNN training (Jeon et al., 2019). For example, E3 (Liu et al., 2019) employs a central cluster resource controller for traffic control and microservice placement.

2. Two-level architecture

Another alternative in architecture evolution is two-level architecture. This architecture type has become widely deployed with the popularity of Mesos (Hindman et al., 2011) and YARN (Vavilapalli et al., 2013). Taking Mesos as an example, the allocator at the cluster level adopts a centralized module to offer resources to applications. Each scheduler on the application side receives offered resources (step ② in Fig. 5) and further partitions them to different tasks. To enable cooperation between these two levels, Mesos provides a set of common interfaces, through which resources are offered upward and tasks are spawned downward. Due to the decoupled architecture, Mesos offers better scalability compared to the centralized model. However, this coarse-grained allocation is oblivious to specific resource demands and induces unavailability of surplus resources (as step ③ shows) at each allocation (Schwarzkopf et al., 2013). In addition, the low-level resource status is unknown to the allocator, which means that scheduling quality varies. In contrast, YARN adopts a request-based resource offer method, but also falls short of interference awareness.

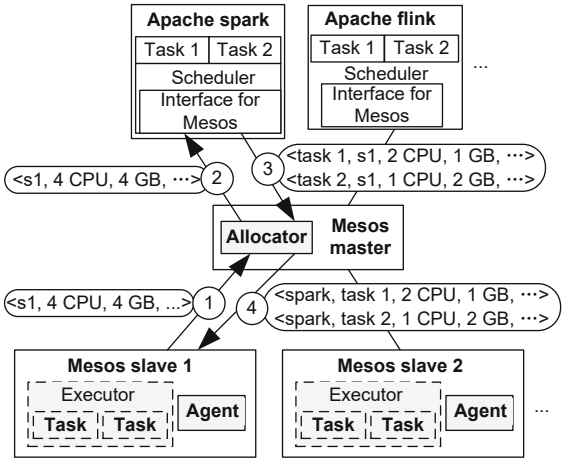

Fig. 5 The Mesos workflow

3. Shared-state architecture

The decentralized model adopted by Borg is similar in spirit to Omega (Schwarzkopf et al., 2013), the next-generation scheduling system in Google. Omega adopts a shared-state approach, which grants each scheduler full access to all cluster resources. In detail, when a scheduler makes a placement decision, it updates the local copy of the shared state in a transaction fashion to ensure data consistency, and any conflicts that occur are mediated by optimistic concurrency control. Interestingly, the Omega schedulers can be implemented in a heterogeneous way with respect to the scheduling policy, provided that common rules are enforced for coordination. In practice, performance viability is ultimately determined

by the frequency of transaction failures and corresponding costs. Microsoft's Apollo (Boutin et al., 2014) shares a common design, except that more local information is involved in scheduling (e.g., queue time estimation at each node), and the deferred correction scheme is used for inter-scheduler coordination.

4. Distributed architecture

Instead of the shared-state method, there is no data plane in the distributed scheduling model. For example, each Sparrow scheduler (Ousterhout et al., 2013) applies batch sampling to retrieve the resource states of cluster nodes. Specifically, the number of schedulers can adapt to load variation, which reduces scheduling delay effectively. However, its scheduling quality is limited due to partial visibility of cluster states, especially when faced with high load or high task diversity. Inspired by Sparrow, Tarcil (Delimitrou et al., 2015) inherits the spirit of Sparrow and Omega, but has a better tradeoff between scheduling delay and quality. Specifically, it takes resource preference and interference into account to enhance scheduling quality, and combines state replicates and sampling at each scheduler to work in parallel. One key optimization compared with Sparrow lies in the adaptive batch size regarding cluster load, which alleviates the visibility limitation. Generally, due to the lack of central control and scheduling information, the distributed scheduling model is hampered in applying global strategies, such as fairness and priority.

5. Hybrid architecture

As mentioned above, the centralized and distributed scheduling models have pros and cons in a complementary way. Naturally, as an alternative to achieve the advantages of both worlds, a hybrid scheduling scheme has come to researchers' attention. Hawk (Delgado et al., 2015) presents a hybrid model consisting of a centralized scheduler for long jobs and several distributed schedulers for short jobs (Fig. 6). To compensate for the suboptimal decision made by a distributed scheduler, a work stealing scheme is used at runtime. Based on Hawk, Delgado et al. (2016) coordinated the distributed scheduler's decision via a bit map, which is used to identify whether a long job exists or not at a cluster node.

Mercury (Karanasos et al., 2015) uses a unified approach that supports a broad range of scheduling modes, from centralized to distributed. In contrast to task classification, Mercury divides cluster resources into two types, guaranteed and queueable, which are allocated by the centralized and distributed schedulers, respectively. Each Mercury agent at each node undertakes interaction with applications and schedulers, and plays a mediator role between them. Specifically, to overcome a suboptimal decision, an individual job provides task runtime estimates for inter scheduler coordination, e.g., via dynamic load partition. Phoenix (Thinakaran et al., 2017) improves the existing hybrid schedulers by introducing the CRV_Monitor unit, which monitors the resource supply and demands from incoming constrained jobs. It can take advantage of application-specific hardware accelerations like GPUs, FPGAs, and job reordering in the worker-side queue to minimize tail latency. However, the sharing/multiplexing policies of these accelerators are unexplored.

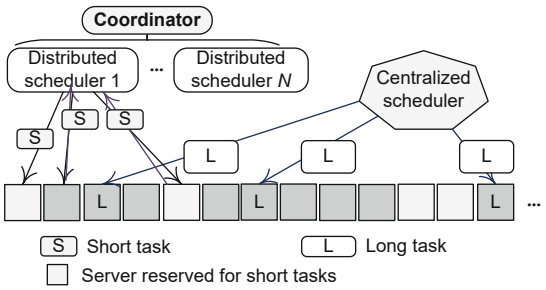

Fig. 6 Diagram of Hawk (the centralized scheduler schedules long tasks with a global view, while the distributed schedulers schedule short tasks)

6. Hierarchical architecture

In a shared cloud, performance coordination regarding co-located jobs is a pressing need. However, the conventional cluster scheduling architecture is designed with no such explicit intent, and lacks runtime knowledge at each machine, such as the processor architecture. In addition, the OS scheduler lacks the ability to bear the responsibility. This motivates the design of the local scheduler per machine, which presents a hierarchical scheduling architecture combined with the global one(s).

Radically, Monotasks (Ousterhout et al., 2017) employs a per-resource scheduler, which makes the contention control more explicit. Similarly, LegoOS (Shan et al., 2018) disseminates traditional OS functionalities into loosely coupled monitors (i.e., schedulers), each of which runs on and manages only one hardware component. The global resource manager is responsible for coarse-grained decisions, while each

monitor makes its own fine-grained resource decision. Targeting AI applications, Ray (Moritz et al., 2018) uses a bottom-up distributed scheduler, where tasks are submitted first to a local scheduler at a node, and forwarded to the global scheduler only if the task requirements cannot be satisfied locally. Instead of relying on an OS scheduler, Avalon (Chen Q et al., 2019) employs a scheduler at each node to perform fine-grained shared resource management, based on both system load and the LS job requirements; any QoS violation can be detected and corrected quickly.

Apart from the per-machine scheduler (different from the OS scheduler), RackSched (Zhu H et al., 2020) presents an inter server scheduler at the rack level (in the top-of-rack switch), which is integrated with intra-server scheduling in each server. It facilitates resource scaling out to hundreds or thousands of cores in a rack, while achieving near-optimal performance as centralized scheduling policies.

Along with the growth of the DC scale, especially the number of data centers owned by one company (e.g., Facebook and Alibaba), a unified system that manages machines across data centers is required, to expand the resource sharing scope and enable scheduling optimization with a broader view (Alibaba, 2020; Multicluster Special Interest Group, 2020; Tang CQ et al., 2020).

Fuxi 2.0 (Alibaba, 2020) adopts an additional federation layer above data centers to achieve cross-DC scheduling, including data caching, service orchestration, and job placement. Thus, cross-DC data dependency is resolved with better data locality, saving network bandwidth. Naturally, it adopts a decentralized architecture to support ultra-scale federated data centers. Similarly, Multicluster Special Interest Group (2020) achieved management of multiple clusters that rely on the cluster registry to store per-cluster metadata, which serves as a basis for developing multi-cluster controllers. In contrast, Twine (Tang CQ et al., 2020) in Facebook scales out natively without an additional federation layer. It manages millions of machines across clusters using a single management system, and adopts the abstraction "entitlement" as the unit to allocate machines from different clusters to host a job. Different from Fuxi's decentralized architecture, both Kubernetes Federation and Twine adopt the centralized architecture.

Summary The existing scheduling models are demonstrated explicitly in Fig. 7, and respective features are summarized in Table 8. We can see that the scheduling system exhibits more hierarchies as the cluster scales (out and up) over time. However, the concern about scheduling quality and delay continues. Clearly, the decoupled design principle is finding its way into both horizontal design (e.g., Sparrow and Avalon) and vertical orchestration (e.g., Mesos, Fuxi 2.0, and Twine). Overall, the following challenges require more attention: (1) how to efficiently coordinate various scheduling behaviors of different scheduler instances or layers; (2) how to design hierarchical scheduling policies to accommodate the increased system hierarchies.

3.2 Task-to-machine mapping schemes

If an architecture is the "skeleton" of a scheduling system, then the adopted algorithm or policy will be the "soul." Below we divide prior work on scheduling algorithms into four categories based on the adopted techniques.

1. Data mining driven approaches

This line of study aims to extract patterns and knowledge from previous resource management records, and transform the knowledge into a comprehensible structure to support scheduling decisions. DejaVu (Vasić et al., 2012) adopts the K-means clustering algorithm to classify workloads to reuse the cached results of previous decisions to minimize scheduling overhead. It is suitable for applications that follow a repeating pattern. However, for jobs never encountered before, the modeling process is time-consuming. Alternatively, Quasar (Delimitrou and Kozyrakis, 2014) performs collaborative filtering (CF, a data mining algorithm) to infer the scale-up/down gains, sensitivity to heterogeneity, and interference in each shared resource upon submitted applications. Measurements like interference sensitivity to shared resources are gathered via profiling and rely on several carefully designed microbenchmarks, but it was assumed that interference sensitivity to different shared resources can be characterized independently, neglecting the complicated interactions.

Apart from inferring resource demand, Morpheus (Jyothi et al., 2016) provides support for service level objective (SLO) inference for periodic jobs that rely on exploiting history data. The job resource

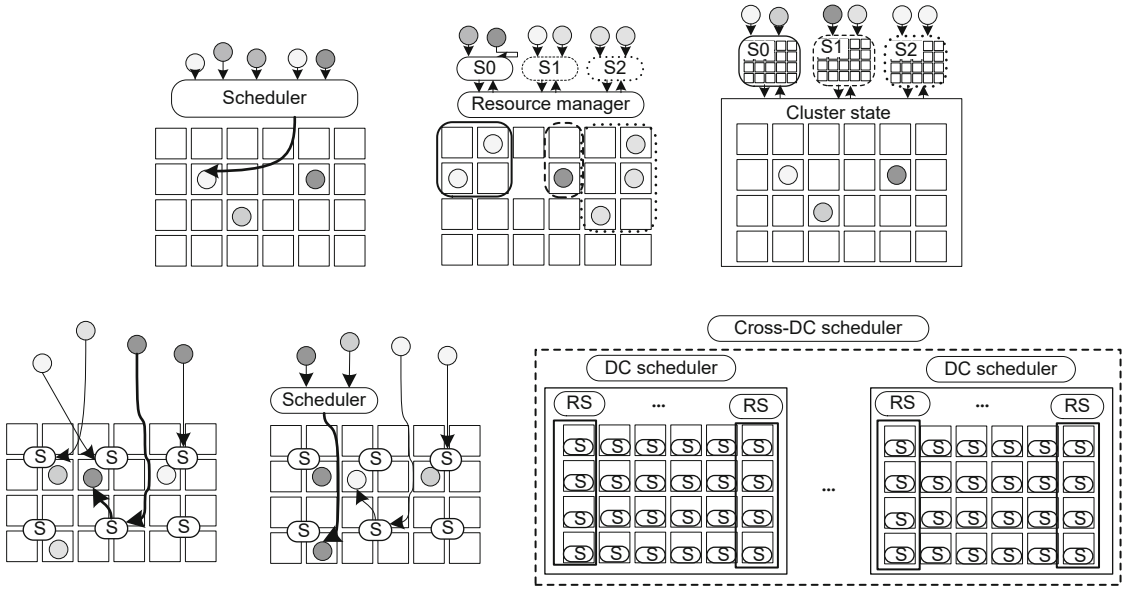

Fig. 7 Abstractions of existing scheduling architectures (□: server; S: scheduler; RS: rack-level scheduler)

Table 8 Existing architectures and the corresponding qualitative comparison

Type	System	Visitability	Cluster-wide policy	Scalability	Scheduling delay	Scheduling quality
Centralized	TORQUE (Staples, 2006), Borg (Verma et al., 2015), Kubernetes (https://kubernetes.io/)	Global	Strict priority (preemption)	Bad	High	Good
Two-level	Mesos (Hindman et al., 2011), YARN (Vavilapalli et al., 2013)	Partial (Mesos), global (YARN)	Strict fairness	Medium	Low	Medium
Shared-state	Omega (Schwarzkopf et al., 2013), Apollo (Boutin et al., 2014)	Global	Priority (preemption)	Good	Medium	Good
Distributed	Sparrow (Ousterhout et al., 2013), Tarcil (Delimitrou et al., 2015)	Partial	Poor support	Best	Low	Bad
Hybrid	Hawk (Delgado et al., 2015), Eagle (Delgado et al., 2016), Mercury (Karanasos et al., 2015)	Global (central component), partial (distributed component)	Priority, load balance	Good	Low	Medium
Hierarchical	Fuxi 2.0 (Alibaba, 2020), Twine (Tang CQ et al., 2020), RackSched (Zhu H et al., 2020)	Global & partial	Locality, priority	Good	Low–medium	Good

model relies on a cost function that combines the penalty of over/under allocation linearly. Based on the inference modules, Morpheus adopts an online packing algorithm specialized for periodic jobs to conduct task-to-machine mapping. Sinan (Zhang YQ et al., 2021) leverages a convolutional neural network (CNN) model and a boosted tree (BT) model to automatically determine the impact of per-tier resource allocations on end-to-end latency. Based on the latency predication, it will adjust resources to each tier periodically to avoid any QoS violation. Different from Morpheus, the training dataset is collected using a space exploration algorithm to cover possible resource allocations, which requires a lot of effort on each deployed application. Additionally, it needs to query resource states where each microservice runs in each decision interval, which may induce high costs.

Overall, the above approaches navigate the tradeoffs in scheduling algorithm design by adopting data mining on previous resource allocations. In short, Quasar trades scheduling delay for better execution environments (Delimitrou and Kozyrakis, 2014), and Morpheus (Jyothi et al., 2016) aims to ease the tension between high cluster utilization and a job's performance predictability. In the microservice context, the accuracy of latency prediction and resource adjustment becomes more important.

2. Linear programming driven approach

This line of work attempts to transform the resource allocation process into a linear programming (LP) problem. The LP problem encodes the pursued objective as a cost function, taking into account resource capacity constraints or requirements as linear equalities and inequalities. Curino et al. (2014) tried to theoretically formalize resource allocation as a mixed integer linear programming (MILP) problem that captures the job's parallelism and demand constraints. In practice, they relied on greedy heuristics to construct or update the allocation plan on the fly in response to new job admissions and cluster state changes. Production and best-effort jobs can be arbitrated simultaneously. Jyothi et al. (2016) relied on LP to capture resource usage patterns in periodic jobs and to derive a resource provisioning model. Particularly, over-allocation and under-allocation penalties were formalized as the cost function to improve resource efficiency.

In a similar way, Tumanov et al. (2016) translated the resource requirements of multiple jobs into an MILP problem and adopted an MILP solver to re-evaluate the plan at each scheduling cycle. Optimus (Peng YH et al., 2018) adopts an online fitting model to learn the convergence curve of one deep learning job running on the GPU server, and reflects resource allocation as a non-linear integer programming problem based on a resource-to-speed model. It quantizes the performance impact of GPU allocation by predicting the job completion time with different numbers of allocated GPUs.

In a nutshell, linear programming formulation is a useful formal tool for studying the solution space, but it is not practical for online scenarios due to the limited scalability of both the number of jobs and cluster scale.

3. Bin-packing-driven approaches

In this category, task scheduling is represented as a multi-dimensional bin-packing problem, which abstracts a single server accounting for different resource types as a bin, and tasks with various resource demands as items to be packed. Tetris (Grandl et al., 2014) adopts heuristics for resource scheduling in the multi-dimensional bin-packing problem for resource scheduling. Specifically, Tetris learns task requirements, t_{r}, and monitors available resources at machines m_{r}. The packing heuristic projects t_{r} and m_{r} into a Euclidean space to pick the ⟨task, machine⟩ pair with the highest dot product value. It can improve average job completion time by serving jobs first that have less remaining work. Specifically, Tetris uses parameter f (which lies between 0 and 1) as the knob to trade off negligible fairness for makespan.

Based on Tetris, Graphene (Grandl et al., 2016b) performs directed acyclic graph (DAG) structure analysis and task profiling to identify long-running tasks and tasks with tough-to-pack resource demands in a job with a heterogeneous DAG. Accordingly, the DAG is divided into subsets of tasks. Each subset is scheduled at a time using bin-packing following a strict resource order.

From the work on Tetris by Grandl et al. (2014), we see that the main efforts were spent on DAG scheduling, and multiple objectives were optimized by combining different policies, but the complex interplay of co-located tasks because of resource contention was not considered.

4. Network-driven approaches

Below we detail network-based solutions to conduct task-to-machine placement. Two lines of work are included, the min-cost max-flow (MCMF) optimization over a graph and neural network based deep learning techniques.

In 2009, Isard from Microsoft introduced a network flow based algorithm, Quincy, which represents task scheduling as an MCMF optimization problem on a graph. The edge weights and capacities are used to enforce scheduling policies, by encoding the application requirements of data locality and fairness (Isard et al., 2009). Firmament (Gog et al., 2016) retains the advantages of Quincy, but it achieves much lower placement latency. Instead of relying on edge weights and capacities to express scheduling policies, it uses aggregators to support network bandwidth-aware policy, load-spreading policy, and others. Fig. 8 depicts the load-spreading policy, where weights at edges denote the number of running tasks at corresponding machines. Moreover, it combines two MCMF algorithms, relaxation and incremental cost scaling, to address placement latency.

The key of MCMF-based scheduling lies in the graph updates to reflect the cluster dynamics and scheduling policies. However, it is difficult to integrate the runtime interference into consideration because its quantification is troublesome. In contrast, AlloX (Le TN et al., 2020) transforms the job

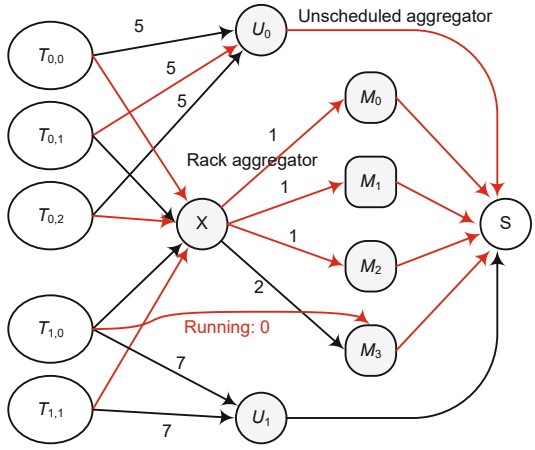

Fig. 8 Load-spreading policy in Firmament with a single-cluster aggregator (X) and costs proportional to the number of tasks per machine (M) (Gog et al., 2016)

scheduling problem in a CPU-GPU hybrid cluster to a min-cost bipartite matching problem. The central challenge is how to pick the configuration for each job and order the jobs, so that the CPU and GPU usage is coordinated to minimize the average job completion time while providing fairness.

In the context of rapid development of neural networks (NNs) and deep learning, multiple NN-based resource managers have been proposed and designed to make the scheduler more intelligent. Mao et al. (2016) claimed that reinforcement learning (RL) is well-suited for modeling resource management systems and decision-making policies. The authors translated the problem of packing tasks with multiple resource demands into a learning problem and built DeepRM, a multi-resource cluster scheduler that learns to optimize various objectives such as minimizing average job slowdown or completion time. Targeting long-running applications in containers, Metis (Wang LP et al., 2020) can automatically learn optimal placements from past logs or lightweight offline profiling using deep RL. Basically, it encodes the scheduling policy into an NN, and trains it with extensive simulations, in which the policy is refined iteratively. Metis innovatively decomposes a complex learning task into a tree-like hierarchy of subtasks to reduce the state and action space. However, it trains a dedicated model for each container group, which trades scheduling delay for high-quality placements.

Similarly, RLScheduler (Zhang D et al., 2020) uses an automated high-performance computing (HPC) batch job scheduler built on RL, which can adapt to various loads and optimization goals via continuous learning and reward-based search. To schedule data-parallel jobs with tasks in DAG, Spear (Hu et al., 2019) applies Monte Carlo tree search (MCTS) for task scheduling, and trains a deep reinforcement learning (DRL) model to guide the expansion and rollout steps in MCTS. It considers task dependencies and heterogeneous resource demands simultaneously to minimize the makespan of complex jobs. In contrast, AuTO (Chen L et al., 2018) builds a two-level DRL system for automatic traffic optimization in data centers. It can collect network information, learn from past decisions, and perform actions to achieve operator-defined goals. Bao et al. (2019) presented Harmony, a deep learning driven machine learning (ML) cluster scheduler, which employs an actor-critic algorithm, job-aware action space exploration, experience replay, and a prediction model to minimize interference or maximize training job performance. To deal with the scalability problem, Mao et al. (2019) developed novel data, scheduling action representation, and new RL training techniques to learn high-quality scheduling policies and handle stochastic job arrival sequences.

Currently, DRL techniques have become popular in solving complex online control problems. However, they are faced with long scheduling delays and scalability problems. Existing intelligent scheduling schemes have to learn policies from simulation or scheduling logs (Mao et al., 2016, 2019; Carastan-Santos and de Camargo, 2017; Wang LP et al., 2020), which may induce unpredictable convergence speeds at runtime and scalability problems regarding job diversity and cluster scale.

Summary The above methods adopt various schemes to cope with resource scheduling at the cluster level, and pursue coordination of different aspects, such as scheduling delay, quality, fairness, and makespan. One practical insight is that perpetually maintaining the algorithmic optimality is impossible, because competing goals are inevitable in a dynamic context. The pros and cons of the above four methods are summarized in Table 9. Overall, they require: (1) efficient performance-critical online information identification, collection, and propagation (e.g., contention quantification of physical resources and demand dynamicity on the application side); (2) a parallel, event-driven algorithm that can

Table 9 Strengths and weaknesses of different resource scheduling approaches

Approach	Strength	Weakness
Data mining driven	Abundant information High scheduling quality Good prediction performance	Requiring a large training set Limited scalability Significant scheduling delay (Quasar)
LP-driven	High efficiency using mature tools Guaranteed optimality	Limited scalability (NP-hard for integer LP) Likely resource over-allocation
Bin-packing-driven	Easy adoption with a good formal structure Increased cost efficiency (e.g., energy) Providing opportunities for co-optimization	Resource fragmentation Increasing complexity with a larger scale (NP-hard) Failing to capture system dynamics
Network-driven	Inspiring innovation High efficiency for assignment Capturing resource dynamics	Requiring extensive tuning High implementation complexity

LP: linear programming

limit the scheduling delays by overlapping information processing time and increasing responsiveness; (3) systematic design with introspection, which takes after-scheduling variation into consideration, instead of ignoring it; (4) an intelligent scheduling system with high learning speed and quality.

3.3 DL-oriented GPU scheduling

Deep leaning (DL) workloads have become so critical that large companies often build multi-tenant GPU clusters for them, similar to shared clusters for big-data analytics. In contrast, GPU scheduling of DL workloads requires Gang scheduling policy and strict locality preference, and GPU is often represented as a monolithic resource to avoid performance interference from co-located jobs, which impairs cluster utilization.

To train DL models, Philly, a centralized resource scheduler, employs a locality-aware policy for distributed training that considers both GPUs and network connectivity, thus reducing the time needed for parameter synchronization (Jeon et al., 2018). However, due to the training time, neither workload co-location nor migration is effectively supported. In addition, CPU cores and memory are allocated in proportion to the requested GPU count, without considering the actual demand. Therefore, resource fragments exist widely. Gandiva (Xiao WC et al., 2018) exploits the cyclic pattern among minibatch iterations to time-slice GPUs across multiple DL training jobs. Specifically, job suspension and migration are conducted at the minibatch boundary, where corresponding memory usage is the lowest, thus reducing CPU-GPU copy cost. It also supports interference-aware job packing based on online profiling to reduce queue waiting time, and inter-GPU migration to mitigate GPU fragmentation or pursue better locality for faster training.

To reduce queuing time, Tiresias (Gu et al., 2019) leverages priority discretization and assignment, and employs the attained service to determine the job scheduling order. In addition, it uses network communication monitoring to infer skew in tensor distributions across parameter servers to guide GPU consolidation. Themis (Mahajan et al., 2020) presents a two-level semi-optimistic scheduling architecture, where an AGENT that resides with an app-scheduler in a hyperparameter optimization system (e.g., HyperDriver) communicates with Themis's ARBITER about performance-critical information, such as an estimated finish-time fair metric, winning allocation. Each ML job can bid on resources offered in an auction, with the goal of fairness in GPU resource allocation. Similarly, AntMan (Xiao WX et al., 2020) co-designs the cluster scheduler and DL frameworks (e.g., TensorFlow) to enable dynamic scaling in both memory (via tensor swap) and the computation unit (via controlling GPU operator launch frequency) for GPU-sharing safety. Specifically, the execution of co-located jobs is carefully coordinated by a local coordinator using the above scaling methods. Without modifying DL training frameworks, Wang SQ et al. (2020) used an offline job profile to guide GPU scale-down for running jobs and scale-up when the GPU is idle. It performs GPU preemption via a daemon called SideCar to stop and restart the DL training process in a container, thus mediating GPU time multiplexing.

However, it cannot use idle GPUs on machines without a SideCar daemon.

Targeting long queuing time as a result of GPU Gang scheduling and fragmentation due to quota-based allocation, HiveD (Zhao HY et al., 2020) reserves GPU resources for each tenant using a new abstraction, the virtual private cluster (VC), comprising a set of GPU cells with different levels of GPU affinity in a physical cluster. Independent GPU scheduling policies can be used within an individual VC. In addition, dynamic binding of cells in a VC to physical GPUs in the cluster is supported, leaving execution opportunity for preemptible, low-priority jobs. The design principle of HiveD is universally applicable to other affinity-aware resources, like the CPU core or NUMA node. This sheds light on resource management problems involving the increasingly heterogeneous infrastructure.

The above schedulers, like Gandiva, Tiresias, and THEMIS, do not consider performance behaviors across heterogeneous GPUs. To fill this gap, Gandiva_fair (Chaudhary et al., 2020) uses the notion of tickets to allocate a proportional share of each GPU type accordingly, and allows utility-guided resource trading of faster GPUs for different DL jobs. To this end, it leverages job profiles on each GPU type to guide trading decisions, coupled with job migration. In comparison, AlloX (Le TN et al., 2020) uses a matrix to capture the sensitivity of DL jobs for different resources, and converts it to a min-cost bipartite matching problem to decide job placement. Both Gandiva_fair and AlloX do not support configurable scheduling policies according to the optimized objective, and thus lack flexibility.

Gavel (Narayanan et al., 2020) formalizes policies with different objectives in a transformed optimization problem that considers heterogeneity of both GPUs and workloads. It takes throughput (number of training iterations per second) of each job running different GPU types in the form of a matrix as input, and expresses the objective as a throughput function. The output is an allocation matrix that performs job-to-GPU mapping in a round-based manner. However, Gavel requires a massive job profile, and may create a scalability bottleneck as GPU clusters scale.

From the above discussion, we can see that the resource heterogeneity considered by current efforts is limited to GPU generations and CPUs, without including resources like TPUs and FPGAs. In other words, cluster heterogeneity is carefully manipulated to be highly application-specific, avoiding unnecessary complexity and operational overhead. Apart from scheduling techniques about DL jobs, the placement of microservice on SmartNICs, and graph processing on GPUs, FPGAs are also receiving increased research efforts. For example, E3 (Liu et al., 2019) uses network-topology-aware microservice placement to allocate SmartNIC processors. ForeGraph (Dai et al., 2017) makes careful use of on-chip memory resources and off-chip bandwidth in a multi-FPGA architecture. Scaph (Zheng et al., 2020) makes efficient use of GPU by identifying high-value subgraphs. We leave the details to interested readers.

Summary A comparison of related GPU scheduling methods is listed in Table 10. We can see that technical features vary across different approaches, which implies the complexity of GPU scheduling algorithm design. However, specialized accelerators such as TPUs, FPGAs, and custom ASICs are also increasingly deployed to train DL models or for other purposes. Corresponding scheduling system design is challenging but very much expected. Nonetheless, locality-related high latency (e.g., parameter synchronization in DL) and co-location-related low utilization (e.g., exclusive access to GPU) are common issues for contemporary accelerator clusters.

3.4 Adaptive management techniques

In this subsection, we emphasize adaptive resource management techniques, which are divided into three types according to their themes: promoting cluster utilization via co-location, inter-application (i.e., LS and BE), and intra-application (e.g., microservices) performance coordination.

3.4.1 Elastic techniques for co-location

Elasticity is the degree to which a system can adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that the available resources match the current demand as closely as possible (Herbst et al., 2013). For most applications, the actual resource usage is highly dynamic, and the average resource requirement is much lower than the peak usage (Reiss et al., 2012). This motivates the exploitation of resource slacks to

Table 10 Comparison of different GPU scheduling approaches

Approach	Architecture	Algorithm	Objective	Job's pattern-aware	Locality-aware
Gandiva (2018)	Centralized	Time slicing	Queuing time, utilization	Yes, cyclic	Yes
Optimus (2018)	Centralized	Remaining-time-driven	JCT	Yes	Yes
Tiresias (2019)	Centralized	Gittins index, LAS	JCT, utilization	Yes	Yes
Themis (2020)	Two-level	Auction, bid	Finish-time fairness	No	N/A
HiveD (2020)	Centralized	Buddy cell	Queuing time, training speed	No	Yes, multi-level affinity
AntMan (2020)	Centralized	Dynamic scaling	Queue time, memory/SM utilization	Yes	N/A
Gavel (2020)	Centralized	Linear programming	Configurable	No	Yes

Approach	Contention-aware	Heterogeneity-aware	Preemption	Job migration	Profilling
Gandiva (2018)	Yes	No	Suspend & resume	Yes	Online, iteration boundary
Optimus (2018)	No	No	Model checkpoint	No	Online, training speed
Tiresias (2019)	No	No	Model checkpoint	No	Online, tensor skew
Themis (2020)	N/A	No	N/A	N/A	Offline, JCT
HiveD (2020)	No	No	Model checkpoint	No	No
AntMan (2020)	Yes	No	Context switch	No	Yes
Gavel (2020)	Yes	Yes	Model checkpoint	Yes	Yes

LAS: least attained service; SM: streaming multi-processing; JCT: job completion time

improve resource efficiency. In a shared cloud, resource slacks can be divided into three categories (Fig. 9).

1. Reservation/Allocation slacks

Reservation slacks refer to resources that have not been reserved, namely, the resources that are available after reservations. Allocation slacks refer to unallocated resources in a given reservation. The former is managed regularly by the cluster manager. We focus on allocation slacks, which are reclaimable.

Carvalho et al. (2014) provided an approach for quantifying the availability of reclaimable resources via a time-series forecasting technique, but it focuses on unused resources over a long time (e.g., multiple months). Bistro (Goder et al., 2015) employs a hierarchical forest of resource trees to model data hosts and their resources. It seeks to run data-parallel jobs (on allocation slacks) with hierarchical resource constraints on customer-facing production systems. However, the resource capacity and constraints are configured manually, which is often error-prone. Similarly, Zhang YQ et al. (2016) proposed to harvest spare computing and storage resources (i.e., allocation slacks) of the primary tenant (i.e., LS) to place secondary tenants (i.e., BE) or a corresponding dataset, with preemption enabled. Briefly, they classified LS jobs based on historical resource utilization, and resource availability was ensured to accommodate BE tasks with the fewest preemptions.

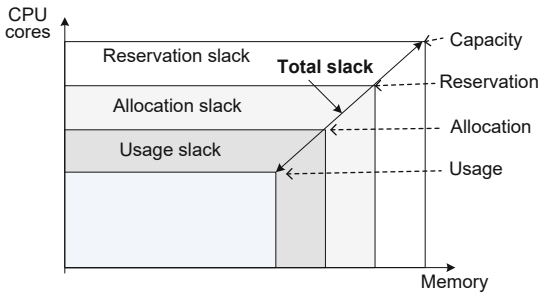

Fig. 9 Three different kinds of resource slacks available in the shared cloud (CPU and memory are considered) (Carvalho et al., 2014)

HCloud (Delimitrou and Kozyrakis, 2016) designs a hybrid resource provision strategy that strives to take advantage of on-demand resources in the public cloud (e.g., Google computing engine) with respect to an application's preference and system interference. However, how the on-demand resources are determined is not clear. In contrast, Yang Y et al. (2017) combined use-reserved resources and allocation slacks to execute data analytic jobs, which improves elasticity at the application level.

2. Usage slacks

Usage slacks refer to allocated resources with no occupancy. It is more challenging to determine the available amount due to the execution dynamics and inefficiency of resource isolation mechanisms.

Grandl et al. (2016a) proposed an altruistic approach (Carbyne) regarding usage slacks based on two general issues:

(1) How to determine how much resource slack a job should yield.

(2) How to redistribute the leftover resources among the rest of the jobs.

Specifically, Carbyne adopts dominant resource fairness (DRF) as the inter-job scheduling solution, which creates significant leftover resources (reclaimable) for each job. For intra-job scheduling, it employs Teris (Grandl et al., 2014) to calculate the reclaimable resources without impacting its performance (a guideline to the first issue). The leftover resources can be redistributed globally to maximize the efficiency by packing more unscheduled tasks or minimizing the average job completion time by adopting a shortest-remaining-time-first policy (which solves the second issue). In contrast to the exploration of cluster-wide resource slacks, Elfen (Yang X et al., 2016) exploits idle SMT resources to run batch programs for the sake of higher utilization, while respecting SLOs of LS jobs. Specifically, it identifies the short periods via high-frequency LS monitoring, such as whether an LS thread is executing on the same SMT core, or whether the budget for a batch job is exhausted. If so, a customized system call will be triggered to release batch hardware resources, without relinquishing their SMT hardware context. Otherwise, the batch thread runs for a budget period. Resource slacks in LS systems are hard to determine due to tight QoS constraints and load dynamics. Generally, the usage (especially the spikes) is captured at a coarse timescale (e.g., in order of minutes in Carvalho et al. (2014) and Goder et al. (2015)), and a finer-scale characterization will yield more opportunities (Islam et al., 2015).

Below, we pay closer attention to elastic techniques at the application level (especially data analytics) to take advantage of resource slacks with limited performance degradation. In addition to the enabling methods at the system level, Amoeba (Ananthanarayanan et al., 2012) enables data-intensive jobs (e.g., MapReduce) to dynamically adapt its resource shares to resource availability at the application level. Incurred resource contention (e.g., preemption by high-priority jobs) is resolved by ending a task via a critical boundary-aware checkpoint/restart mechanism, wherein the evicted task is enqueued as a new task. In essence, it breaks the static granularity of a task by splitting the original task safely (particularly for the remaining work) without work wastage or state migration. However, when to terminate the task and how to manage the reclaimed resources are delegated to the cluster scheduler (as steps a–d in Fig. 10), which may incur scheduling delays due to the control shifting across machines.

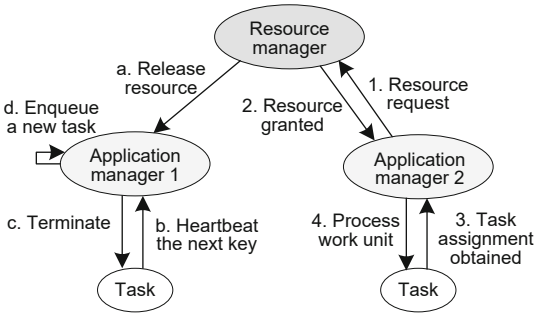

Fig. 10 System architecture for elasticity in Ameoba (the resource manager notifies application manager 1 to relinquish resources, and redistributes them to application 2)

To alleviate elasticity-incurred recomputation, TR-Spark (Yan et al., 2016) uses a task-level checkpoint scheme, coupled with an opportunistic scheduling algorithm, to ensure performance of transient resources. Briefly, it takes resource instability and task duration/output size into consideration to reduce the number of recomputation times and checkpoint cost. Sharma P et al. (2016) shared a similar spirit. Wang JJ and Balazinska (2017) designed an elastic memory allocation approach, which adaptively adjusts memory limits in a Java virtual machine for cloud data analytics. In detail, it adds a set of new customized APIs (to JVM) at the application level to probe the heap state, and combines garbage collection (GC) cost estimation to drive memory reallocation. In contrast, Chen W et al. (2017) explored container-based resource preemption schemes that rely on Linux Cgroups to adapt to CPU and memory allocation gracefully.

Targeting tension between resource utilization and end-to-end latency due to microservice-sharing in multi-tenancy, GrandSLAm (Kannan RS et al., 2019) conducts contention management at microservice granularity, which breaks the end-to-end SLAs into disaggregated partial SLAs at each microservice. Specifically, it builds a linear regression model

for each microservice to yield the estimated completion time (ECT) of a request, accounting for critical factors like the sharing degree and queuing delay. The partial deadline at each particular microservice is calculated as the proportion of application-specific SLAs. GrandSLAm exploits the variability between ECT and partial deadline (i.e., idle slack) to perform request reordering and dynamic batching to achieve balanced SLA and server throughput. Particularly, time slacks can be inherited by downstream microservices. However, GrandSLAm aims at the sequential microservice architecture, which limits its application scope.

Summary Resource elasticity enables better coordination between collocated tasks with respect to QoS and resource efficiency. However, three respects need to be considered: (1) current methods often resort to the global scheduler for reclaimed resource management, which lacks local information and often fails to differentiate the reclaimable resources (usage slacks) from reservation/allocation slacks. (2) Generally, multiple resource types are required to accommodate new tasks. Ideal elastic approaches need to determine the set of resource types and available amounts to accommodate co-scheduled tasks. (3) The core aspects of elasticity speed and precision (Herbst et al., 2013) also require quantitative analysis for fine-grained optimization. Specifically, several principles can be drawn for elastic resource scheduling on usage slacks:

(1) For high-priority jobs, QoS should be guaranteed when determining the amount of reclaimable resources.

(2) Low-priority jobs that request the leftover resources should be elastic enough with low overhead imposed on the system.

(3) Workload characterization and online monitoring in both resources and performance are of particular importance in elastic scheduling.

3.4.2 Best-effort job-oriented resource throttling

In shared environments, BE jobs are often throttled via resource isolation methods to curb the performance interference to LS jobs. Typically, anomaly detection and tracing are needed.

To this end, Zhang X et al. (2013) from Google designed a monitoring system named CPI^2 (Fig. 11), which uses the globally collected CPI as an interference proxy. For each application, it collects the CPI information for a task on each machine in a fan-in fashion (see the CPI sample aggregator in Fig. 11), and calculates the corresponding distribution (CPI_spec, including the application name, averaged CPI value, and standard deviation). Then the result is sent back to the agent on each machine for the sake of anomaly detection via comparison with the local CPI. To pinpoint the disturber, correlation analysis regarding observed CPI and CPU usages of each BE job is conducted. Consequently, the disturber's CPU resources are throttled locally via CPU hard-capping. This method is effective in CPU-intensive applications, but it can barely adapt to other types, such as I/O-intensive ones. Yang HL et al. (2013) proposed a profile-based interference measure and local QoS correction scheme, Bubble-Flux. It leverages a set of carefully designed micro-benchmarks to quantify and model resource contention. The interference measure can be used to guide global task scheduling to achieve coordinated co-running of LS and BE jobs. Specifically, the Flux mechanism periodically monitors the number of instructions per circle (IPC) of an LS job, and controls the execution of BE jobs via Linux signals (e.g., SIGSTOP) to correct any performance variation. Currently, this specifically concerns the memory subsystem, especially the last-level cache and memory bandwidth.

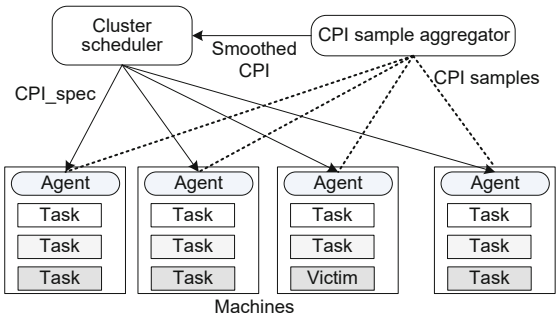

Fig. 11 Data flow in cluster for anomaly detection. Modified from Zhang X et al. (2013), Copyright 2013, with permission from ACM

Unlike local resource throttling, DeepDive (Novaković D et al., 2013) adopts a migration-based approach to eliminate local interference. Targeting a virtualized environment, it periodically combines local and global information (including low-level metrics and disk/network I/O measurements) for tasks of the same stage to detect misbehaviors. The disturber VM is pinpointed via classification. Before

migration, the low-level behaviors of the VM and corresponding impact on other VMs at the destination machine are evaluated, which may incur some delay. VM-related migration can be expensive because it consumes CPU and I/O bandwidth, which may deteriorate the already overloaded environment.

Summary The prior work explores workload profiling and continuous monitoring to guide online coordination between BE and LS jobs. Faced with increased diversity in application, corresponding characterization is essential for online monitoring, cause analysis, disturber pinpointing, and associated resource regulation. Currently, the complex interplay between underlying resources makes the determination of disturber BE jobs and hot resource difficult. As a result, resource throttling lacks accurate control in terms of resource size and warmup delay.

3.4.3 Resource compensation for latency-sensitive jobs

In contrast to the BE-related resource manipulation, another line of research concentrates on resource re-allocation directly applied to LS jobs, based on intra-application features.

For interactive applications, Haque et al. (2015) presented an incremental parallelization algorithm, Few-to-Many (FM), which uses task demand profiles and hardware parallelism to compute re-allocation policies. The policies are represented in a table that specifies when and how much software parallelism to add (according to the dynamic system load and task execution progress). Incremental parallelism has proved to be a powerful tool for reducing tail latency, but FM does not consider the diversity of task features, such as tasks with short execution time, which often do not benefit from parallelism. As a complement, Li J et al. (2016) used an adaptive work-stealing mechanism that entails instantaneous task progress, system load, and target latency to decide when to parallelize requests with stealing and when to limit parallelism of large requests. Similar work with respect to efficient parallelism was conducted in Han et al. (2016) and Jeon et al. (2016).

Targeting web search applications, Li CL et al. (2017) developed an effective caching mechanism in which a revenue-aware adaptive refresh policy and domain-specific information were employed to achieve net profit improvement. RobinHood (Berger et al., 2018) re-allocates cache space to balance tail latency across different back ends. It relies on a new metric, the request blocking count, to identify which back end requires additional resources. However, caching reduces freshness of the computation results, which may lead to potential revenue loss, especially in the search advertising context.

Targeting similar large-scale online services with multi-tiers, Dean and Barroso (2013) from Google offered an in-depth discussion of tail-tolerant techniques, such as hedged requests and tied requests. As an example, the hedged request method relies on a replicated request issued to multiple replicas to hide the tail latency caused by any straggler. This method can be further enhanced by tied requests to reduce redundant computing. Generally, this line of work does not explicitly take the interference due to resource contention into consideration, but focuses on performance-oriented adaptivity at the application level. Although tail-tolerant techniques can reduce latency hiccups regardless of root causes, approaches that do are particularly useful for joint optimization with multiple objectives, such as resource utilization and tail latency, as stated by Kasture and Sanchez (2014) and Lo et al. (2015).

In contrast to simple multi-tiered web services, microservice applications create numerous loosely coupled, specialized microservices with hundreds or thousands of unique tiers. As Gan et al. (2019a) claimed, dependencies between microservices will introduce backpressure effects and cascading QoS violations that quickly propagate through the system, making performance unpredictable. This calls for coordinated (with respect to multiple microservices) methods to curb any performance degradation.

To tackle overload-incurred performance degradation on microservices like WeChat, DAGOR (Zhou H et al., 2018) uses the average waiting time of requests in the pending queue at each microservice as an indicator of overload. Once overload is detected, a corresponding server performs priority-based local admission control (i.e., load shading). Specifically, related upstream servers coordinate with the overloaded server via admission-level notification, such that requests destined to be rejected are shed early at upstream servers and collaborative correction is achieved. To cater to diverse demand across microservices in the system and CPU architecture, μTune (Sriraman and Wenisch, 2018) uses a linear

model built offline to autotune the optimal threading models (e.g., polling vs. blocking network reception; inline vs. dispatched RPC execution) and thread pool sizes under changing service loads to reduce tail latency of mid-tier microservices. However, μTune does not analyze inter-model transition overhead, where spurious context switches and thread wakeups can dominate microservice latency distributions.

Sriraman et al. (2019) exploited coarse-grained OS and hardware configuration knobs to tune existing server CPU architectures to accommodate assigned microservice. To this end, μSKU was developed to automate search within the seven-knob soft-SKU design space using A/B testing, to identify the best microservice-specific configurations to reduce performance fluctuations. In contrast, ANT-Man (Hou et al., 2020) exploits the intra-application variability of power usage in each microservice, and uses a decision tree model to capture performance-power correlations under different load levels. To eliminate the power management latency, it directly manipulates an on-chip voltage regulator and adopts a model-specific register associated with each microservice to accelerate on-chip voltage transition. It dynamically determines differentiated power budget for each microservice, thus preventing any power-caused QoS variations and achieving performance-power coordination.

Seer (Gan et al., 2019b) employs DL and the massive amount of tracing data in a production cloud to learn spatial and temporal patterns that indicate upcoming QoS violations. Fig. 12 demonstrates its architecture. It uses a lightweight, distributed RPC-level tracing system to collect end-to-end execution traces for each user request, which are aggregated to a centralized database (i.e., TraceDB). At runtime, it takes real-time tracing such as per-microservice queue depths as input, and outputs the probability of initiating a QoS violation. Once a misbehaving microservice is pinpointed, Seer relies on low-level monitoring or tunable contentious microbenchmarks to determine the root causes (i.e., which resource is saturated). Subsequently, the cluster manager is notified to take actions, such as resizing the Docker container or last-level cache partitioning, for correction. However, Seer requires offline and online trace labeling and fine-grained tracing to track the number of outstanding requests across the system stack, which is non-trivial in practice and may induce interference. In comparison, FIRM (Qiu et al., 2020) leverages a support vector machine (SVM) to detect/localize microservices that cause SLO violations, taking the variability of each critical path and each microservice instance as input. It adopts an RL model for every microservice to mitigate the violations via dynamic reprovisioning, such as adding or reducing the amount of resources attached to a microservice, or scaling the number of replicas.

Instead of the supervised model in Seer, Sage (Gan et al., 2021) leverages unsupervised learning based on using low-frequency traces and user-level metrics to diagnose the root cause of end-to-end QoS violations. Specifically, Sage uses causal Bayesian networks to capture inter-microservice dependencies, and relies on hypothetical scenario-based inference to determine which set of microservices initiated a QoS violation. Subsequently, adjustment regarding the deployment or resource allocation is performed

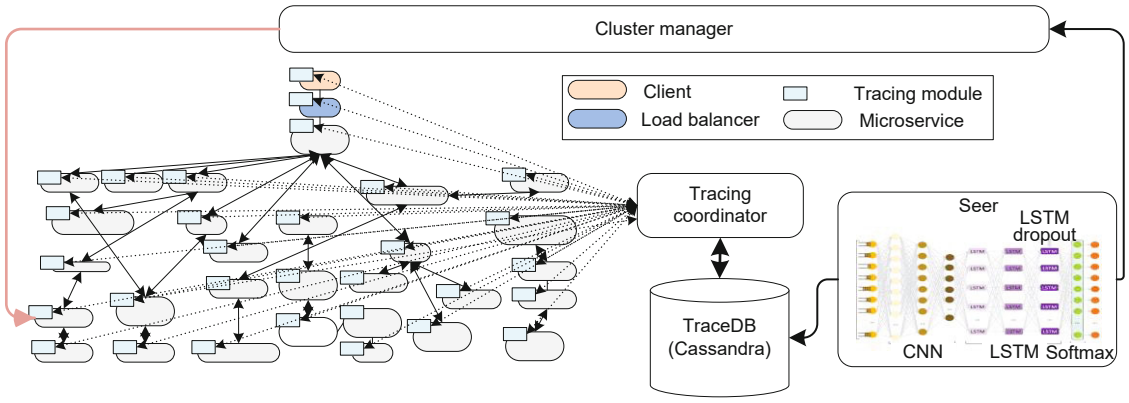

Fig. 12 Workflow of anomaly monitoring and correction for microservices (taking Seer (Gan et al., 2019b) as an example)

accordingly.

In the context of proliferation of special-purpose hardware, E3 (Liu et al., 2019) explores schemes in which microservices are offloaded to SmartNICs. Targeting the overload problem, E3 monitors the incoming/outgoing network throughput, packet queue depth, and microservice's execution time to guide microservice migration to the host, thus avoiding potential QoS violation. Furthermore, anticipating accelerator design, Accelerometer (Sriraman and Dhanotia, 2020) analyzes common operations among microservices, and identifies acceleration opportunities while accounting for associated offloading overhead when deployed at scale.

Summary Current research relies on interference detection (regarding either performance or resource usage), resource throttling/reprovisioning, or application migration to limit performance degradation. We also see that tracing systems like Dapper (Sigelman et al., 2010) and Jaeger (https://www.jaegertracing.io/) are essential for resource management in Gan et al. (2019b, 2021) and Qiu et al. (2020), which facilitates locating performance issues and resolving them. Above all, we learn the following:

1. Application-dependent resource sensitivity and the corresponding cost of reallocation should be fully understood and considered to reduce the time required for changes to take effect, especially in cloud microservices due to the delayed queuing effect (Zhang YQ et al., 2021).

2. As the application/hardware complexity and cloud scale continue to grow, data-driven systems like Seer are a promising way to offer practical solutions (Gan et al., 2019b).

3. RL is proved to be well-suited for learning resource reprovisioning policies, because it provides a tight feedback loop for exploring the action space and generating optimal policies without relying on prior knowledge or rules.

4 Future research directions

Through the investigation of the literature, we can see that a lot of progress has been made in resource scheduling techniques, and several research directions remain to be explored.

4.1 Coordination across system layers

The trend of the decoupled scheduling architecture has resulted in multiple layers in the scheduling system stack (Fig. 13), and the disconnected schedulers at different levels expose new opportunities for optimization.

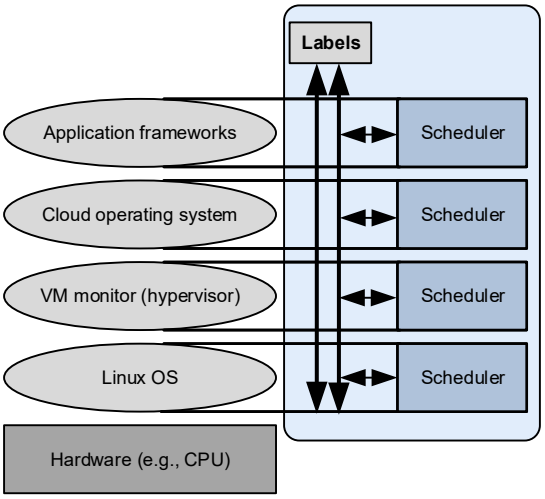

Fig. 13 Scheduling stack in the cloud

Take the two-level scheduling architecture as an example. Cooperation exists between the cluster allocator and task scheduler in both Mesos and YARN. However, in Mesos, the allocation size does not match the task demand due to the lack of cross-layer coordination, as explained in Section 3.1. As for YARN, the task scheduler is oblivious to detailed resource states, which may limit the scheduling quality. The same problem exists in virtualized systems. Song et al. (2013) tried to address the double CPU scheduling problem (i.e., most operations in a guest OS are opaque to the hypervisor) between the hypervisor and guest operating systems via a vCPU ballooning scheme. Similarly, Zhao M and Cabrera (2018) pointed out that the lack of unawareness between VM host- and guest-level schedulers hinders the timeliness guarantee, and attempts to address it via cross-layer communication (e.g., bandwidth demands and deadlines). Similar research can be found in Kannan S et al. (2017). In addition to the work related to virtual systems, Giceva (2016) proposed to narrow the knowledge gap between the OS and database engines to improve resource efficiency. Therefore, the scheduling barrier among application, hypervisor, and OS should be removed and requires

more attention.

Although coordination has already been adopted in scheduling systems, accounting for scheduling either at the same layer (Ousterhout et al., 2013; Schwarzkopf et al., 2013; Boutin et al., 2014; Delimitrou et al., 2015; Delgado et al., 2016) or across two different layers remains an open issue. However, along with the popularity of the hierarchical architecture (Alibaba, 2020) and increasing diversity of scheduling policies at each layer, multi-layer coordination and software-defined policy management are still a pressing need.

4.2 Push disaggregation to the extreme

As explained above, decoupled design is gaining popularity in system design. A similar trend is observed in hardware architecture evolution.

The current hardware architecture is mostly server-centric, which makes it ill-suited to harness increasingly heterogeneous resources and support new programming models (e.g., microservices). To overcome this obstacle, multiple aggressive prototypes of disaggregated architecture prototypes have been designed, such as Facebook's Disaggregated Rack (Facebook, 2015), memory blades (Lim et al., 2009), NUMA-related architecture (Novaković S et al., 2014), and storage blades (Klimovic et al., 2016). These solutions improve hardware scalability and provide new optimization opportunities for task scheduling. Zellweger et al. (2014) and Shan et al. (2018) explored decoupled or disaggregated design of operating systems. Specifically, Shan et al. (2018) presented a new OS with traditional functionalities decoupled into loosely coordinated monitors for each disaggregated resource type. Task scheduling and execution are achieved in a cooperative manner.

Most of these systems are in the early stages of development, and the scope of disaggregation is often limited to a single rack. Disaggregation at the cluster scale and related software design choices still need to be further explored. Furthermore, a corresponding programming model and performance impact require complete understanding. Overall, several open questions remain:

1. How can performance isolation be achieved at each independent resource component, and how can their allocation be balanced to avoid hotspots (e.g., network bandwidth)?

2. Which scheduling architecture is a good fit for a disaggregated data center, and do new design paradigms exist?

3. How can global scheduling policies be enforced and coordinated with local ones at each resource component?

4.3 Accelerator-oriented scheduling coordination

Deployment of specialized accelerators such as GPUs, TPUs, and FPGAs has been the norm in data centers. It is foreseeable that this trend will continue, and result in ubiquitous access to various accelerators. The design of a corresponding management system and scheduling strategies will become a pressing need, including combinations of different accelerators, computation offloading, and resource multiplexing. Three research areas are presented below:

1. Intra-accelerator management

Each accelerator should support a self-contained resource virtualization scheme that facilitates spatial or temporal sharing among concurrent jobs and avoids resource under-utilization. This also calls for architecture-specific isolation design that minimizes runtime performance interference, such as context switch overhead and state swapping between the accelerator and host memory. Finally, each accelerator should provide a unified platform to outside entities (other accelerators, host CPU) that hiding underlying resource details.

2. Inter-accelerator management

The communication among accelerators, host CPU, and I/O devices should be efficient, with a unified protocol, and avoid unnecessary data copying (e.g., allowing local direct access over PCIe, or remote access through cloud networks). Providing proper I/O abstraction is essential to fully drive these heterogeneous systems (Min et al., 2018).

3. Heterogeneity-aware global control

Such control requires a centralized coordinator to harness and allocate different accelerators with system-wide knowledge, such as the characteristics of each accelerator, the load on each one, and the topology of the PCIe network. Furthermore, different scheduling policies targeting multiple objectives are essential for the overall performance, such as job collocation for better resource utilization and context switching for fairness.

4.4 Edge-cloud coordination

Edge computing (EC) is an emerging technology that moves the computation and storage resources to the network edge, closer to Internet of Things (IoT) devices (e.g., mobile phones, self-driving cars) (Wang JY et al., 2019). Compared to central cloud computing (CC), EC has relatively small computation capacity, but provides advantages of short access distance and flexible geographical distribution. Thus, it is able to significantly reduce network traffic and response latency, which complements to CC.

To take advantage of both CC and EC, a collaborative processing model between the edge and the cloud is required (Kang et al., 2017; Grulich and Nawab, 2018). For NN inference used in real-time video processing, Neurosurgeon (Kang et al., 2017) splits the NN model at a specific layer into edge and cloud parts. When processing the frame, the edge node conducts frame processing first using the edge part, and transfers the output to the cloud node, which performs further processing using the cloud part. This splitting-based cooperation enables the use of the compute resources at the edge, which makes processing at the cloud more efficient. Apart from workload splitting, other efforts focus on data compression and differential communication to reduce network bandwidth cost and processing latency (Vulimiri et al., 2015).

It is expected that the increased diversity in resource demands, latency requirements, and processing capacity, and the proliferation of IoT applications and terminal devices, will encourage the design and deployment of various edge-cloud coordination solutions.

5 Conclusions

In light of the trends of high resource heterogeneity and application diversity, current resource isolation mechanisms and scheduling systems are faced with new challenges, such as the lack of efficient resource isolation schemes (especially for accelerators) and the coordination along performance and resource dimensions. Targeting the unpredictable performance and low resource utilization in shared cloud environments, in this paper we presented an extensive survey and analysis of related works. First, we showed the potential for software-based scheduling schemes as a complement to the available isolation mechanisms at the microarchitecture and system levels. Second, we examined existing OS design for heterogeneous computers, and resource management architectures at the cluster level. In addition, task-to-machine mapping algorithms including DL placement on the GPU cluster were explored. Third, we investigated the adaptive techniques to cope with performance disturbances (e.g., of microservices), and competing objectives from both the system and cluster levels. Finally, future research directions concerning the requirement for inter-/intra-layer coordination across the software stack, the practicality of disaggregated data centers, the pressing need for heterogeneity-aware coordination, and state-of-the-art edge-cloud coordination solutions were presented. To some extent, this paper provides a holistic technical view for related researchers, and explores new system design and optimization guidelines and solutions for current scheduling challenges.

Contributors

Yuzhao WANG drafted the paper. Zhibin YU helped organize the paper. Junqing YU revised and finalized the paper.

Compliance with ethics guidelines

Yuzhao WANG, Junqing YU, and Zhibin YU declare that they have no conflict of interest.

References

Achermann R, Panwar A, Bhattacharjee A, et al., 2020. Mitosis: transparently self-replicating page-tables for large-memory machines. Proc 25th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.283-300. https://doi.org/10.1145/3373376.3378468

Akkus IE, Chen RC, Rimac I, et al., 2018. SAND: towards high-performance serverless computing. Proc USENIX Annual Technical Conf, p.923-935.

Alibaba, 2020. Fuxi 2.0—The Core Dispatching System of Ali Economy Towards the Big Data and Cloud Computing Scheduling Challenge (in Chinese). https://developer.aliyun.com/article/760083 [Accessed on July 1, 2021].

Ananthanarayanan G, Douglas C, Ramakrishnan R, et al., 2012. True elasticity in multi-tenant data-intensive compute clusters. Proc 3rd ACM Symp on Cloud Computing, p.1-7.

Asmussen N, Völp M, Nöthen B, et al., 2016. M3: a hardware/operating-system co-design to tame heterogeneous manycores. Proc 21st Int Conf on Architectural Support for Programming Languages and Operating Systems, p.189-203. https://doi.org/10.1145/2872362.2872371

Ausavarungnirun R, Miller V, Landgraf J, et al., 2018. MASK: redesigning the GPU memory hierarchy to support multi-application concurrency. Proc 23rd Int Conf on Architectural Support for Programming Languages and Operating Systems, p.503-518. https://doi.org/10.1145/3173162.3173169

Bao YX, Peng YH, Wu C, 2019. Deep learning-based job placement in distributed machine learning clusters. Proc IEEE Conf on Computer Communications, p.505-513. https://doi.org/10.1109/INFOCOM.2019.8737460

Bauman E, Ayoade G, Lin ZQ, 2015. A survey on hypervisor-based monitoring: approaches, applications, and evolutions. *ACM Comput Surv*, 48(1):10. https://doi.org/10.1145/2775111

Baumann A, Barham P, Dagand PE, et al., 2009. The multikernel: a new OS architecture for scalable multicore systems. Proc ACM SIGOPS 22nd Symp on Operating Systems Principles, p.29-44. https://doi.org/10.1145/1629575.1629579

Berger DS, Berg B, Zhu T, et al., 2018. RobinHood: tail latency-aware caching—dynamically reallocating from cache-rich to cache-poor. Proc 13th USENIX Conf on Operating Systems Design and Implementation, p.195-212.

Bhadauria M, McKee SA, 2010. An approach to resource-aware co-scheduling for CMPs. Proc 24th ACM Int Conf on Supercomputing, p.189-199. https://doi.org/10.1145/1810085.1810113

Bitirgen R, Ipek E, Martinez JF, 2008. Coordinated management of multiple interacting resources in chip multiprocessors: a machine learning approach. Proc 41st IEEE/ACM Int Symp on Microarchitecture, p.318-329. https://doi.org/10.1109/MICRO.2008.4771801

Blagodurov S, Zhuravlev S, Fedorova A, et al., 2010. A case for NUMA-aware contention management on multicore systems. Proc 19th Int Conf on Parallel Architectures and Compilation Techniques, p.557-558. https://doi.org/doi.org/10.1145/1854273.1854350

Boucher S, Kalia A, Andersen DG, et al., 2018. Putting the "micro" back in microservice. Proc USENIX Annual Technical Conf, p.645-650.

Boutin E, Ekanayake J, Lin W, et al., 2014. Apollo: scalable and coordinated scheduling for cloud-scale computing. Proc 11th USENIX Symp on Operating Systems Design and Implementation, p.285-300.

Cadden J, Unger T, Awad Y, et al., 2020. SEUSS: skip redundant paths to make serverless fast. Proc 15th European Conf on Computer Systems, p.1-15. https://doi.org/10.1145/3342195.3392698

Carastan-Santos D, de Camargo RY, 2017. Obtaining dynamic scheduling policies with simulation and machine learning. Proc Int Conf for High Performance Computing, Networking, Storage and Analysis, p.1-13. https://doi.org/10.1145/3126908.3126955

Carvalho M, Cirne W, Brasileiro F, et al., 2014. Long-term SLOs for reclaimed cloud computing resources. Proc ACM Symp on Cloud Computing, p.1-13. https://doi.org/10.1145/2670979.2670999

Castelló A, Peña AJ, Mayo R, 2018. Exploring the interoperability of remote GPGPU virtualization using rCUDA and directive-based programming models. *J Supercomput*, 74(11):5628-5642. https://doi.org/10.1007/s11227-016-1791-y

Chandra D, Guo F, Kim S, et al., 2005. Predicting inter-thread cache contention on a chip multi-processor architecture. Proc 11th Int Symp on High-Performance Computer Architecture, p.340-351. https://doi.org/10.1109/HPCA.2005.27

Chaudhary S, Ramjee R, Sivathanu M, et al., 2020. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. Proc 15th European Conf on Computer Systems, p.1-16. https://doi.org/10.1145/3342195.3387555

Chen L, Lingys J, Chen K, et al., 2018. AuTO: scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. Proc Conf of the ACM Special Interest Group on Data Communication, p.191-205. https://doi.org/10.1145/3230543.3230551

Chen Q, Yang HL, Mars J, et al., 2016. Baymax: QoS awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. Proc 21st Int Conf on Architectural Support for Programming Languages and Operating Systems, p.681-696. https://doi.org/10.1145/2872362.2872368

Chen Q, Yang HL, Guo MY, et al., 2017. Prophet: precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. Proc 22nd Int Conf on Architectural Support for Programming Languages and Operating Systems, p.17-32. https://doi.org/10.1145/3037697.3037700

Chen Q, Wang ZN, Leng JW, et al., 2019. Avalon: towards QoS awareness and improved utilization through multi-resource management in datacenters. Proc ACM Int Conf on Supercomputing, p.272-283. https://doi.org/10.1145/3330345.3330370

Chen W, Rao J, Zhou XB, 2017. Preemptive, low latency datacenter scheduling via lightweight virtualization. Proc USENIX Annual Technical Conf, p.251-263.

Cherkasova L, Gupta D, Vahdat A, 2007. Comparison of the three CPU schedulers in Xen. *ACM SIGMETRICS Perform Eval Rev*, 35(2):42-51. https://doi.org/10.1145/1330555.1330556

Cho S, Jin L, 2006. Managing distributed, shared L2 caches through OS-level page allocation. Proc 39th Annual IEEE/ACM Int Symp on Microarchitecture, p.455-468. https://doi.org/10.1109/MICRO.2006.31

Curino C, Difallah DE, Douglas C, et al., 2014. Reservation-based scheduling: if you're late don't blame us! Proc ACM Symp on Cloud Computing, p.1-14. https://doi.org/10.1145/2670979.2670981

Dai GH, Huang TH, Chi YZ, et al., 2017. ForeGraph: exploring large-scale graph processing on multi-FPGA architecture. Proc ACM/SIGDA Int Symp on Field-Programmable Gate Arrays, p.217-226. https://doi.org/10.1145/3020078.3021739

Dean J, Barroso LA, 2013. The tail at scale. *Commun ACM*, 56(2):74-80. https://doi.org/10.1145/2408776.2408794

Delgado P, Dinu F, Kermarrec AM, et al., 2015. Hawk: hybrid datacenter scheduling. Proc USENIX Annual Technical Conf, p.499-510.

Delgado P, Didona D, Dinu F, et al., 2016. Job-aware scheduling in Eagle: divide and stick to your probes. Proc 7th ACM Symp on Cloud Computing, p.497-509. https://doi.org/10.1145/2987550.2987563

Delimitrou C, Kozyrakis C, 2014. Quasar: resource-efficient and QoS-aware cluster management. Proc 19th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.127-144. https://doi.org/10.1145/2541940.2541941

Delimitrou C, Kozyrakis C, 2016. HCloud: resource-efficient provisioning in shared cloud systems. Proc 21st Int Conf on Architectural Support for Programming Languages and Operating Systems, p.473-488. https://doi.org/10.1145/2872362.2872365

Delimitrou C, Sanchez D, Kozyrakis C, 2015. Tarcil: reconciling scheduling speed and quality in large shared clusters. Proc 6th ACM Symp on Cloud Computing, p.97-110. https://doi.org/10.1145/2806777.2806779

Dhakal A, Kulkarni SG, Ramakrishnan KK, 2020. GSLICE: controlled spatial sharing of GPUs for a scalable inference platform. Proc 11th ACM Symp on Cloud Computing, p.492-506. https://doi.org/10.1145/3419111.3421284

Ebrahimi E, Lee CJ, Mutlu O, et al., 2010. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. Proc 15th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.335-346. https://doi.org/10.1145/1736020.1736058

Engler DR, Kaashoek MF, O'Toole J, 1995. Exokernel: an operating system architecture for application-level resource management. Proc 15th ACM Symp on Operating Systems Principles, p.251-266. https://doi.org/10.1145/224056.224076

Eyerman S, Eeckhout L, 2010. Probabilistic job symbiosis modeling for SMT processor scheduling. Proc 15th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.91-102. https://doi.org/10.1145/1736020.1736033

Facebook, 2015. Facebook Disaggregated Rack. http://goo.gl/6h2Ut [Accessed on July 1, 2021].

Feliu J, Sahuquillo J, Petit S, et al., 2013. L1-bandwidth aware thread allocation in multicore SMT processors. Proc 22nd Int Conf on Parallel Architectures and Compilation Techniques, p.123-132. https://doi.org/10.1109/PACT.2013.6618810

Feliu J, Eyerman S, Sahuquillo J, et al., 2016. Symbiotic job scheduling on the IBM POWER8. Proc IEEE Int Symp on High Performance Computer Architecture, p.669-680. https://doi.org/10.1109/HPCA.2016.7446103

Firestone D, Putnam A, Mundkur S, et al., 2018. Azure accelerated networking: smartnics in the public cloud. Proc 15th USENIX Symp on Networked Systems Design and Implementation, p.51-66.

Fowers J, Ovtcharov K, Papamichael M, et al., 2018. A configurable cloud-scale DNN processor for real-time AI. Proc ACM/IEEE 45th Annual Int Symp on Computer Architecture, p.1-14. https://doi.org/10.1109/ISCA.2018.00012

Gan Y, Zhang YQ, Cheng DL, et al., 2019a. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. Proc 24th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.3-18. https://doi.org/10.1145/3297858.3304013

Gan Y, Zhang YQ, Hu K, et al., 2019b. Seer: leveraging big data to navigate the complexity of performance debugging in cloud microservices. Proc 24th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.19-33. https://doi.org/10.1145/3297858.3304004

Gan Y, Liang MY, Dev S, et al., 2021. Sage: practical and scalable ML-driven performance debugging in microservices. Proc 26th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems, p.135-151. https://doi.org/10.1145/3445814.3446700

Giceva J, 2016. Database/Operating System Co-design. PhD Thesis, ETH Zurich, Switzerland.

Goder A, Spiridonov A, Wang Y, 2015. Bistro: scheduling data-parallel jobs against live production systems. Proc USENIX Annual Technical Conf, p.459-471.

Gog I, Schwarzkopf M, Gleave A, et al., 2016. Firmament: fast, centralized cluster scheduling at scale. Proc 12th USENIX Conf on Operating Systems Design and Implementation, p.99-115.

Goglin B, Furmento N, 2009. Enabling high-performance memory migration for multithreaded applications on LINUX. Proc IEEE Int Symp on Parallel & Distributed Processing, p.1-9. https://doi.org/10.1109/IPDPS.2009.5161101

Grandl R, Ananthanarayanan G, Kandula S, et al., 2014. Multi-resource packing for cluster schedulers. Proc ACM Conf on SIGCOMM, p.455-466. https://doi.org/10.1145/2619239.2626334

Grandl R, Chowdhury M, Akella A, et al., 2016a. Altruistic scheduling in multi-resource clusters. Proc 12th USENIX Conf on Operating Systems Design and Implementation, p.65-80.

Grandl R, Kandula S, Rao S, et al., 2016b. Graphene: packing and dependency-aware scheduling for data-parallel clusters. Proc 12th USENIX Conf on Operating Systems Design and Implementation, p.81-97.

Grulich PM, Nawab F, 2018. Collaborative edge and cloud neural networks for real-time video processing. *Proc VLDB Endow*, 11(12):2046-2049. https://doi.org/10.14778/3229863.3236256

Gu JC, Chowdhury M, Shin KG, et al., 2019. Tiresias: a GPU cluster manager for distributed deep learning. Proc 16th USENIX Symp on Networked Systems Design and Implementation, p.485-500.

Guo F, Li YK, Lui JCS, et al., 2019. DCUDA: dynamic GPU scheduling with live migration support. Proc ACM Symp on Cloud Computing, p.114-125. https://doi.org/10.1145/3357223.3362714

Gysi T, Bär J, Hoefler T, 2016. dCUDA: hardware supported overlap of computation and communication. Proc Int Conf for High Performance Computing, Networking, Storage and Analysis, p.609-620. https://doi.org/10.1109/SC.2016.51

Han J, Jeon S, Choi YR, et al., 2016. Interference management for distributed parallel applications in consolidated clusters. Proc 21st Int Conf on Architectural Support for Programming Languages and Operating Systems, p.443-456. https://doi.org/10.1145/2872362.2872388

Haque E, Eom YH, He YX, et al., 2015. Few-to-Many: incremental parallelism for reducing tail latency in interactive services. Proc 20th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.161-175. https://doi.org/10.1145/2694344.2694384

Herbst NR, Kounev S, Reussner R, 2013. Elasticity in cloud computing: what it is, and what it is not. Proc 10th Int Conf on Autonomic Computing, p.23-27.

Hindman B, Konwinski A, Zaharia M, et al., 2011. Mesos: a platform for fine-grained resource sharing in the data center. Proc 8th USENIX Conf on Networked Systems Design and Implementation, p.295-308.

Hong CH, Spence I, Nikolopoulos DS, 2017. GPU virtualization and scheduling methods: a comprehensive survey. *ACM Comput Surv*, 50(3):35. https://doi.org/10.1145/3068281

Hou XF, Li C, Liu JC, et al., 2020. ANT-Man: towards agile power management in the microservice era. Proc Int Conf for High Performance Computing, Networking, Storage and Analysis, Article 78.

Hsu CH, Zhang YQ, Laurenzano MA, et al., 2015. Adrenaline: pinpointing and reining in tail queries with quick voltage boosting. Proc IEEE 21st Int Symp on High Performance Computer Architecture, p.271-282. https://doi.org/10.1109/HPCA.2015.7056039

Hu ZM, Tu J, Li BC, 2019. Spear: optimized dependency-aware task scheduling with deep reinforcement learning. Proc IEEE 39th Int Conf on Distributed Computing Systems, p.2037-2046. https://doi.org/10.1109/ICDCS.2019.00201

Ibanez S, Shahbaz M, McKeown N, 2019. The case for a network fast path to the CPU. Proc 18th ACM Workshop on Hot Topics in Networks, p.52-59. https://doi.org/10.1145/3365609.3365851

Intel, 2016. Intel Cache Allocation Technique. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology [Accessed on July 1, 2021].

Isard M, Prabhakaran V, Currey J, et al., 2009. Quincy: fair scheduling for distributed computing clusters. Proc ACM SIGOPS 22nd Symp on Operating Systems Principles, p.261-276. https://doi.org/10.1145/1629575.1629601

Islam S, Venugopal S, Liu AN, 2015. Evaluating the impact of fine-scale burstiness on cloud elasticity. Proc 6th ACM Symp on Cloud Computing, p.250-261. https://doi.org/10.1145/2806777.2806846

Jeon M, He YX, Kim H, et al., 2016. TPC: target-driven parallelism combining prediction and correction to reduce tail latency in interactive services. Proc 21st Int Conf on Architectural Support for Programming Languages and Operating Systems, p.129-141. https://doi.org/10.1145/2872362.2872370

Jeon M, Venkataraman S, Phanishayee A, et al., 2018. Multi-Tenant GPU Clusters for Deep Learning Workloads: Analysis and Implications. Technical Report No. MSR-TR-2018-13, Microsoft Research, USA.

Jeon M, Venkataraman S, Phanishayee A, et al., 2019. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. Proc USENIX Annual Technical Conf, p.947-960.

Jeong EY, Woo S, Jamshed M, et al., 2014. mTCP: a highly scalable user-level TCP stack for multicore systems. Proc 11th USENIX Conf on Networked Systems Design and Implementation, p.489-502.

Jeyakumar V, Alizadeh M, Mazières D, et al., 2013. EyeQ: practical network performance isolation at the edge. Proc 10th USENIX Symp on Networked Systems Design and Implementation, p.297-311.

Jia ZP, Witchel E, 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. Proc 26th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems, p.152-166. https://doi.org/10.1145/3445814.3446701

Jyothi SA, Curino C, Menache I, et al., 2016. Morpheus: towards automated SLOs for enterprise clusters. Proc 12th USENIX Conf on Operating Systems Design and Implementation, p.117-134.

Kakivaya G, Xun L, Hasha R, et al., 2018. Service fabric: a distributed platform for building microservices in the cloud. Proc 13th EuroSys Conf, Article 33. https://doi.org/10.1145/3190508.3190546

Kalia A, Kaminsky M, Andersen DG, 2016. FaSST: fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. Proc 12th USENIX Conf on Operating Systems Design and Implementation, p.185-201.

Kalia A, Kaminsky M, Andersen D, 2019. Datacenter RPCs can be general and fast. Proc 16th USENIX Symp on Networked Systems Design and Implementation, p.1-16.

Kang YP, Hauswald J, Gao C, et al., 2017. Neurosurgeon: collaborative intelligence between the cloud and mobile edge. Proc 22nd Int Conf on Architectural Support for Programming Languages and Operating Systems, p.615-629. https://doi.org/10.1145/3037697.3037698

Kannan RS, Subramanian L, Raju A, et al., 2019. GrandSLAm: guaranteeing SLAs for jobs in microservices execution frameworks. Proc 14th EuroSys Conf, Article 34. https://doi.org/10.1145/3302424.3303958

Kannan S, Gavrilovska A, Gupta V, et al., 2017. HeteroOS: OS design for heterogeneous memory management in datacenter. Proc 44th Annual Int Symp on Computer Architecture, p.521-534. https://doi.org/10.1145/3079856.3080245

Kannan S, Ren YJ, Bhattacharjee A, 2021. KLOCs: kernel-level object contexts for heterogeneous memory systems. Proc 26th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems, p.65-78. https://doi.org/10.1145/3445814.3446745

Kapoor R, Porter G, Tewari M, et al., 2012. Chronos: predictable low latency for data center applications. Proc 3rd ACM Symp on Cloud Computing, Article 9. https://doi.org/10.1145/2391229.2391238

Karanasos K, Rao S, Curino C, et al., 2015. Mercury: hybrid centralized and distributed scheduling in large shared clusters. Proc USENIX Annual Technical Conf, p.485-497.

Kasture H, Sanchez D, 2014. Ubik: efficient cache sharing with strict QoS for latency-critical workloads. Proc 19th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.729-742. https://doi.org/10.1145/2541940.2541944

Khawaja A, Landgraf J, Prakash R, et al., 2018. Sharing, protection, and compatibility for reconfigurable fabric with AMORPHOS. Proc 13th USENIX Conf on Operating Systems Design and Implementation, p.107-127.

Khorasani F, Esfeden HA, Farmahini-Farahani A, et al., 2018. RegMutex: inter-warp GPU register time-sharing. Proc ACM/IEEE 45th Annual Int Symp on Computer Architecture, p.816-828. https://doi.org/10.1109/ISCA.2018.00073

Klimovic A, Kozyrakis C, Thereska E, et al., 2016. Flash storage disaggregation. Proc 11th European Conf on Computer Systems, Article 29. https://doi.org/10.1145/2901318.2901337

Knauerhase R, Brett P, Hohlt B, et al., 2008. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54-66. https://doi.org/10.1109/MM.2008.48

Korolija D, Roscoe T, Alonso G, 2020. Do OS abstractions make sense on FPGAs? Proc 14th USENIX Symp on Operating Systems Design and Implementation, p.991-1010.

Kotra JB, Zhang HB, Alameldeen AR, et al., 2018. CHAMELEON: a dynamically reconfigurable heterogeneous memory system. Proc 51st Annual IEEE/ACM Int Symp on Microarchitecture, p.533-545. https://doi.org/10.1109/MICRO.2018.00050

Lazarev N, Xiang SJ, Adit N, et al., 2021. Dagger: efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs. Proc 26th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems, p.36-51. https://doi.org/10.1145/3445814.3446696

Le TN, Sun X, Chowdhury M, et al., 2020. AlloX: compute allocation in hybrid clusters. Proc 15th European Conf on Computer Systems, Article 31. https://doi.org/10.1145/3342195.3387547

Le YF, Chang H, Mukherjee S, et al., 2017. UNO: uniflying host and smart NIC offload for flexible packet processing. Proc Symp on Cloud Computing, p.506-519. https://doi.org/10.1145/3127479.3132252

Li CL, Andersen DG, Fu Q, et al., 2017. Workload analysis and caching strategies for search advertising systems. Proc Symp on Cloud Computing, p.170-180. https://doi.org/10.1145/3127479.3129255

Li J, Agrawal K, Elnikety S, et al., 2016. Work stealing for interactive services to meet target latency. Proc 21st ACM SIGPLAN Symp on Principles and Practice of Parallel Programming, Article 14. https://doi.org/10.1145/2851141.2851151

Li JL, Sharma NK, Ports DRK, et al., 2014. Tales of the tail: hardware, OS, and application-level sources of tail latency. Proc ACM Symp on Cloud Computing, p.1-14. https://doi.org/10.1145/2670979.2670988

Lim K, Chang JC, Mudge T, et al., 2009. Disaggregated memory for expansion and sharing in blade servers. Proc 36th Annual Int Symp on Computer Architecture, p.267-278. https://doi.org/10.1145/1555754.1555789

Linux Community, 2016. Linux Kernel Namespace. https://en.wikipedia.org/wiki/Linux_namespaces [Accessed on Feb. 23, 2021].

Liu M, Peter S, Krishnamurthy A, et al., 2019. E3: energy-efficient microservices on SmartNIC-accelerated servers. Proc USENIX Annual Technical Conf, p.363-378.

Lo D, Cheng LQ, Govindaraju R, et al., 2015. Heracles: improving resource efficiency at scale. Proc 42nd Annual Int Symp on Computer Architecture, p.450-462. https://doi.org/10.1145/2749469.2749475

Luo QY, Lin JK, Zhuo YW, et al., 2019. Hop: heterogeneity-aware decentralized training. Proc 24th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.893-907. https://doi.org/10.1145/3297858.3304009

Ma JC, Zuo GF, Loughlin K, et al., 2020. A hypervisor for shared-memory FPGA platforms. Proc 25th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.827-844. https://doi.org/10.1145/3373376.3378482

Madhavapeddy A, Scott DJ, 2014. Unikernels: the rise of the virtual library operating system. *Commun ACM*, 57(1):61-69. https://doi.org/10.1145/2541883.2541895

Mahajan K, Balasubramanian A, Singhvi A, et al., 2020. Themis: fair and efficient GPU cluster scheduling. Proc 17th USENIX Symp on Networked Systems Design and Implementation, p.289-304.

Manco F, Lupu C, Schmidt F, et al., 2017. My VM is lighter (and safer) than your container. Proc 26th Symp on Operating Systems Principles, p.218-233. https://doi.org/10.1145/3132747.3132763

Mao HZ, Alizadeh M, Menache I, et al., 2016. Resource management with deep reinforcement learning. Proc 15th ACM Workshop on Hot Topics in Networks, p.50-56. https://doi.org/10.1145/3005745.3005750

Mao HZ, Schwarzkopf M, Venkatakrishnan SB, et al., 2019. Learning scheduling algorithms for data processing clusters. Proc Special Interest Group on Data Communication, p.270-288. https://doi.org/10.1145/3341302.3342080

Mars J, Tang LJ, 2013. Whare-Map: heterogeneity in "homogeneous" warehouse-scale computers. Proc 40th Annual Int Symp on Computer Architecture, p.619-630. https://doi.org/10.1145/2485922.2485975

Min C, Kang W, Kumar M, et al., 2018. Solros: a data-centric operating system architecture for heterogeneous computing. Proc 13th EuroSys Conf, Article 36. https://doi.org/10.1145/3190508.3190523

Moon Y, Lee S, Jamshed MA, et al., 2020. AccelTCP: accelerating network applications with stateful TCP offloading. Proc 17th USENIX Symp on Networked Systems Design and Implementation, p.77-92.

Moritz P, Nishihara R, Wang S, et al., 2018. Ray: a distributed framework for emerging AI applications. Proc 13th USENIX Conf on Operating Systems Design and Implementation, p.561-577. https://doi.org/10.48550/arXiv.1712.05889

Multicluster Special Interest Group, 2020. Kubernetes Multicluster. https://github.com/kubernetes/community/tree/master/sigmulticluster [Accessed on July 1, 2021].

Mutlu O, Moscibroda T, 2008. Parallelism-aware batch scheduling: enhancing both performance and fairness of shared DRAM systems. Proc Int Symp on Computer Architecture, p.63-74. https://doi.org/10.1109/ISCA.2008.7

Nagaraj K, Bharadia D, Mao HZ, et al., 2016. NUMFabric: fast and flexible bandwidth allocation in datacenters. Proc ACM SIGCOMM Conf, p.188-201. https://doi.org/10.1145/2934872.2934890

Narayanan D, Santhanam K, Kazhamiaka F, et al., 2020. Heterogeneity-aware cluster scheduling policies for deep learning workloads. Proc 14th USENIX Symp on Operating Systems Design and Implementation, p.481-498.

Nightingale EB, Hodson O, McIlroy R, et al., 2009. Helios: heterogeneous multiprocessing with satellite kernels. Proc ACM SIGOPS 22nd Symp on Operating Systems Principles, p.221-234. https://doi.org/10.1145/1629575.1629597

Novaković D, Vasić N, Novaković S, et al., 2013. DeepDive: transparently identifying and managing performance interference in virtualized environments. Proc USENIX Annual Technical Conf, p.219-230.

Novaković S, Daglis A, Bugnion E, et al., 2014. Scale-out NUMA. Proc 19th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.3-18. https://doi.org/10.1145/2541940.2541965

Ousterhout K, Wendell P, Zaharia M, et al., 2013. Sparrow: distributed, low latency scheduling. Proc 24th ACM Symp on Operating Systems Principles, p.69-84. https://doi.org/10.1145/2517349.2522716

Ousterhout K, Canel C, Ratnasamy S, et al., 2017. Monotasks: architecting for performance clarity in data analytics frameworks. Proc 26th Symp on Operating Systems Principles, p.184-200. https://doi.org/10.1145/3132747.3132766

Panda A, Zheng WT, Hu XH, et al., 2017. SCL: simplifying distributed SDN control planes. Proc 14th USENIX Symp on Networked Systems Design and Implementation, p.329-345.

Park JJK, Park Y, Mahlke S, 2015. Chimera: collaborative preemption for multitasking on a shared GPU. Proc 20th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.593-606. https://doi.org/10.1145/2694344.2694346

Peng X, Shi XH, Dai HL, et al., 2020. Capuchin: tensor-based GPU memory management for deep learning. Proc 25th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.891-905. https://doi.org/10.1145/3373376.3378505

Peng YH, Bao YX, Chen YR, et al., 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. Proc 20th EuroSys Conf, Article 3. https://doi.org/10.1145/3190508.3190517

Popov M, Jimborean A, Black-Schaffer D, 2019. Efficient thread/page/parallelism autotuning for NUMA systems. Proc ACM Int Conf on Supercomputing, p.342-353. https://doi.org/10.1145/3330345.3330376

Pothukuchi RP, Greathouse JL, Rao K, et al., 2019. Tangram: integrated control of heterogeneous computers. Proc 52nd Annual IEEE/ACM Int Symp on Microarchitecture, p.384-398. https://doi.org/10.1145/3352460.3358285

Pratheek B, Jawalkar N, Basu A, 2021. Improving GPU multi-tenancy with page walk stealing. Proc IEEE Int Symp on High-Performance Computer Architecture, p.626-639. https://doi.org/10.1109/HPCA51647.2021.00059

Qiu HR, Banerjee SS, Jha S, et al., 2020. FIRM: an intelligent fine-grained resource management framework for SLO-oriented microservices. Proc 14th USENIX Symp on Operating Systems Design and Implementation, p.805-825.

Qureshi MK, Patt YN, 2006. Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches. Proc 39th Annual IEEE/ACM Int Symp on Microarchitecture, p.423-432. https://doi.org/10.1109/MICRO.2006.49

Rao J, Wang K, Zhou XB, et al., 2013. Optimizing virtual machine scheduling in NUMA multicore systems. Proc IEEE 19th Int Symp on High Performance Computer Architecture, p.306-317. https://doi.org/10.1109/HPCA.2013.6522328

Reiss C, Tumanov A, Ganger GR, et al., 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. Proc 3rd ACM Symp on Cloud Computing, Article 7. https://doi.org/10.1145/2391229.2391236

Rhu M, Gimelshein N, Clemons J, et al., 2016. vDNN: virtualized deep neural networks for scalable, memory-efficient neural network design. Proc 49th Annual IEEE/ACM Int Symp on Microarchitecture, p.1-13. https://doi.org/10.1109/MICRO.2016.7783721

Rossbach CJ, Currey J, Silberstein M, et al., 2011. PTask: operating system abstractions to manage GPUs as compute devices. Proc 23rd ACM Symp on Operating Systems Principles, p.233-248. https://doi.org/10.1145/2043556.2043579

Sanchez D, Kozyrakis C, 2011. Vantage: scalable and efficient fine-grain cache partitioning. Proc 38th Annual Int Symp on Computer Architecture, p.57-68. https://doi.org/10.1145/2000064.2000073

Schwarzkopf M, Konwinski A, Abd-El-Malek M, et al., 2013. Omega: flexible, scalable schedulers for large compute clusters. Proc 8th ACM European Conf on Computer Systems, p.351-364. https://doi.org/10.1145/2465351.2465386

Sengupta D, Belapure R, Schwan K, 2013. Multi-tenancy on GPGPU-based servers. Proc 7th Int Workshop on Virtualization Technologies in Distributed Computing, p.3-10. https://doi.org/10.1145/2465829.2465830

Sengupta D, Goswami A, Schwan K, et al., 2014. Scheduling multi-tenant cloud workloads on accelerator-based systems. Proc Int Conf for High Performance Computing, Networking, Storage and Analysis, p.513-524. https://doi.org/10.1109/SC.2014.47

Shan YZ, Huang YT, Chen YL, et al., 2018. LegoOS: a disseminated, distributed OS for hardware resource disaggregation. Proc 13th USENIX Conf on Operating Systems Design and Implementation, p.69-87.

Sharma NK, Zhao CXY, Liu M, et al., 2020. Programmable calendar queues for high-speed packet scheduling. Proc 17th USENIX Symp on Networked Systems Design and Implementation, p.685-699.

Sharma P, Guo T, He X, et al., 2016. Flint: batch-interactive data-intensive processing on transient servers. Proc 11th European Conf on Computer Systems, Article 6. https://doi.org/10.1145/2901318.2901319

Shen ZM, Sun Z, Sela GE, et al., 2019. X-Containers: breaking down barriers to improve performance and isolation of cloud-native containers. Proc 24th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.121-135. https://doi.org/10.1145/3297858.3304016

Shillaker S, Pietzuch P, 2020. Faasm: lightweight isolation for efficient stateful serverless computing. Proc USENIX Annual Technical Conf, p.419-433. https://doi.org/10.48550/arXiv.2002.09344

Sigelman BH, Barroso LA, Burrows M, et al., 2010. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. https://storage.googleapis.com/pub-tools-public-publication-data/pdf/36356.pdf [Accessed on July 1, 2021].

Singh S, Chana I, 2016. A survey on resource scheduling in cloud computing: issues and challenges. *J Grid Comput*, 14(2):217-264. https://doi.org/10.1007/s10723-015-9359-2

Snavely A, Tullsen DM, 2000. Symbiotic jobscheduling for a simultaneous multithreaded processor. *ACM SIGOPS Oper Syst Rev*, 34(5):234-244. https://doi.org/10.1145/378993.379244

Song X, Shi JC, Chen HB, et al., 2013. Schedule processes, not VCPUs. Proc 4th Asia-Pacific Workshop on Systems, p.1-7. https://doi.org/10.1145/2500727.2500736

Sriraman A, Dhanotia A, 2020. Accelerometer: understanding acceleration opportunities for data center overheads at hyperscale. Proc 25th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.733-750. https://doi.org/10.1145/3373376.3378450

Sriraman A, Wenisch TF, 2018. μTune: auto-tuned threading for OLDI microservices. Proc 13th USENIX Conf on Operating Systems Design and Implementation, p.177-194.

Sriraman A, Dhanotia A, Wenisch TF, 2019. SoftSKU: optimizing server architectures for microservice diversity @scale. Proc 46th Int Symp on Computer Architecture, p.513-526. https://doi.org/10.1145/3307650.3322227

Staples G, 2006. TORQUE resource manager. Proc ACM/IEEE Conf on Supercomputing. https://doi.org/10.1145/1188455.1188464

Subramanian L, Seshadri V, Ghosh A, et al., 2015. The application slowdown model: quantifying and controlling the impact of inter-application interference at shared caches and main memory. Proc 48th Int Symp on Microarchitecture, p.62-75. https://doi.org/10.1145/2830772.2830803

Tanasic I, Gelado I, Cabezas J, et al., 2014. Enabling preemptive multiprogramming on GPUs. Proc ACM/IEEE 41st Int Symp on Computer Architecture, p.193-204. https://doi.org/10.1109/ISCA.2014.6853208

Tang CQ, Yu K, Veeraraghavan K, et al., 2020. Twine: a unified cluster management system for shared infrastructure. Proc 14th USENIX Symp on Operating Systems Design and Implementation, p.787-803.

Tang LJ, Mars J, Vachharajani N, et al., 2011. The impact of memory subsystem resource sharing on datacenter applications. Proc 38th Annual Int Symp on Computer Architecture, p.283-294.

Tembey P, Gavrilovska A, Schwan K, 2014. Merlin: application- and platform-aware resource allocation in consolidated server systems. Proc ACM Symp on Cloud Computing, p.1-14. https://doi.org/10.1145/2670979.2670993

Thinakaran P, Gunasekaran JR, Sharma B, et al., 2017. Phoenix: a constraint-aware scheduler for heterogeneous datacenters. Proc IEEE 37th Int Conf on Distributed Computing Systems, p.977-987. https://doi.org/10.1109/ICDCS.2017.262

Tirmazi M, Barker A, Deng N, et al., 2020. Borg: the next generation. Proc 15th European Conf on Computer Systems, Article 30. https://doi.org/10.1145/3342195.3387517

Tumanov A, Zhu T, Park JW, et al., 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. Proc 11th European Conf on Computer Systems, Article 35. https://doi.org/10.1145/2901318.2901355

Vanga M, Gujarati A, Brandenburg BB, 2018. Tableau: a high-throughput and predictable VM scheduler for high-density workloads. Proc 13th EuroSys Conf, Article 28. https://doi.org/10.1145/3190508.3190557

Vasić N, Novaković D, Miučin S, et al., 2012. DejaVu: accelerating resource allocation in virtualized environments. Proc 17th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.423-436. https://doi.org/10.1145/2150976.2151021

Vavilapalli VK, Murthy AC, Douglas C, et al., 2013. Apache Hadoop YARN: yet another resource negotiator. Proc 4th Annual Symp on Cloud Computing, Article 5. https://doi.org/10.1145/2523616.2523633

Verma A, Pedrosa L, Korupolu M, et al., 2015. Large-scale cluster management at Google with Borg. Proc 10th European Conf on Computer Systems, Article 18. https://doi.org/10.1145/2741948.2741964

Vulimiri A, Curino C, Godfrey PB, et al., 2015. Wanalytics: geo-distributed analytics for a data intensive world. Proc ACM SIGMOD Int Conf on Management of Data, p.1087-1092. https://doi.org/10.1145/2723372.2735365

Wang JJ, Balazinska M, 2017. Elastic memory management for cloud data analytics. Proc USENIX Annual Technical Conf, p.745-758.

Wang JY, Pan JL, Esposito F, et al., 2019. Edge cloud offloading algorithms: issues, methods, and perspectives. *ACM Comput Surv*, 52(1):2. https://doi.org/10.1145/3284387

Wang LN, Ye JM, Zhao YM, et al., 2018. SuperNeurons: dynamic GPU memory management for training deep neural networks. Proc 23rd ACM SIGPLAN Symp on Principles and Practice of Parallel Programming, p.41-53. https://doi.org/10.1145/3178487.3178491

Wang LP, Weng QZ, Wang W, et al., 2020. Metis: learning to schedule long-running applications in shared container clusters at scale. Proc Int Conf for High Performance Computing, Networking, Storage and Analysis, Article 68.

Wang SQ, Gonzalez OJ, Zhou XB, et al., 2020. An efficient and non-intrusive GPU scheduling framework for deep learning training systems. Proc Int Conf for High Performance Computing, Networking, Storage and Analysis, Article 90.

Wang ZN, Yang J, Melhem R, et al., 2016. Simultaneous multikernel GPU: multi-tasking throughput processors via fine-grained sharing. Proc IEEE Int Symp on High Performance Computer Architecture, p.358-369. https://doi.org/10.1109/HPCA.2016.7446078

Weerasiri D, Barukh MC, Benatallah B, et al., 2017. A taxonomy and survey of cloud resource orchestration techniques. *ACM Comput Surv*, 50(2):26. https://doi.org/10.1145/3054177

Williams D, Koller R, 2016. Unikernel monitors: extending minimalism outside of the box. Proc 8th USENIX Workshop on Hot Topics in Cloud Computing, p.1-6.

Xiao WC, Bhardwaj R, Ramjee R, et al., 2018. Gandiva: introspective cluster scheduling for deep learning. Proc 13th USENIX Conf on Operating Systems Design and Implementation, p.595-610.

Xiao WX, Ren SR, Li Y, et al., 2020. AntMan: dynamic scaling on GPU clusters for deep learning. Proc 14th USENIX Symp on Operating Systems Design and Implementation, p.533-548.

Xu QM, Jeon H, Kim K, et al., 2016. Warped-Slicer: efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming. Proc ACM/IEEE 43rd Annual Int Symp on Computer Architecture, p.230-242. https://doi.org/10.1109/ISCA.2016.29

Xu YJ, Musgrave Z, Noble B, et al., 2013. Bobtail: avoiding long tails in the cloud. Proc 10th USENIX Symp on Networked Systems Design and Implementation, p.329-341.

Yan Y, Gao YJ, Chen Y, et al., 2016. TR-Spark: transient computing for big data analytics. Proc 7th ACM Symp on Cloud Computing, p.484-496. https://doi.org/10.1145/2987550.2987576

Yang HL, Breslow A, Mars J, et al., 2013. Bubble-Flux: precise online QoS management for increased utilization in warehouse scale computers. Proc 40th Annual Int Symp on Computer Architecture, p.607-618. https://doi.org/10.1145/2485922.2485974

Yang X, Blackburn SM, McKinley KS, 2016. Elfen scheduling: fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. Proc USENIX Annual Technical Conf, p.309-322.

Yang Y, Kim GW, Song WW, et al., 2017. Pado: a data processing engine for harnessing transient resources in datacenters. Proc 12th European Conf on Computer Systems, p.575-588. https://doi.org/10.1145/3064176.3064181

Yeh TT, Sabne A, Sakdhnagool P, et al., 2017. Pagoda: fine-grained GPU resource virtualization for narrow tasks. Proc 22nd ACM SIGPLAN Symp on Principles and Practice of Parallel Programming, p.221-234. https://doi.org/10.1145/3018743.3018754

Yeh TT, Sinclair MD, Beckmann BM, et al., 2021. Deadline-aware offloading for high-throughput accelerators. Proc IEEE Int Symp on High-Performance Computer Architecture, p.479-492. https://doi.org/10.1109/HPCA51647.2021.00048

Zellweger G, Gerber S, Kourtis K, et al., 2014. Decoupling cores, kernels, and operating systems. Proc 11th USENIX Symp on Operating Systems Design and Implementation, p.17-31.

Zha Y, Li J, 2020. Virtualizing FPGAs in the cloud. Proc 25th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.845-858. https://doi.org/10.1145/3373376.3378491

Zha Y, Li J, 2021. When application-specific ISA meets FPGAs: a multi-layer virtualization framework for heterogeneous cloud FPGAs. Proc 26th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems, p.123-134. https://doi.org/10.1145/3445814.3446699

Zhang D, Dai D, He YB, et al., 2020. RLScheduler: an automated HPC batch job scheduler using reinforcement learning. Proc Int Conf for High Performance Computing, Networking, Storage and Analysis, p.1-15. https://doi.org/10.1109/SC41405.2020.00035

Zhang JS, Xiong YQ, Xu NY, et al., 2017. The Feniks FPGA operating system for cloud computing. Proc 8th Asia-Pacific Workshop on Systems, Article 22. https://doi.org/10.1145/3124680.3124743

Zhang X, Dwarkadas S, Shen K, 2009. Towards practical page coloring-based multicore cache management. Proc 4th ACM European Conf on Computer Systems, p.89-102. https://doi.org/10.1145/1519065.1519076

Zhang X, Tune E, Hagmann R, et al., 2013. CPI^2: CPU performance isolation for shared compute clusters. Proc 8th ACM European Conf on Computer Systems, p.379-391. https://doi.org/10.1145/2465351.2465388

Zhang XT, Zheng X, Wang Z, et al., 2019. Fast and scalable VMM live upgrade in large cloud infrastructure. Proc 24th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.93-105. https://doi.org/10.1145/3297858.3304034

Zhang YQ, Laurenzano MA, Mars J, et al., 2014. SMiTe: precise QoS prediction on real-system SMT processors to improve utilization in warehouse scale computers. Proc 47th Annual IEEE/ACM Int Symp on Microarchitecture, p.406-418. https://doi.org/10.1109/MICRO.2014.53

Zhang YQ, Prekas G, Fumarola GM, et al., 2016. History-based harvesting of spare cycles and storage in large-scale datacenters. Proc 12th USENIX Conf on Operating Systems Design and Implementation, p.755-770.

Zhang YQ, Hua WZ, Zhou ZZ, et al., 2021. Sinan: ML-based and QoS-aware resource management for cloud microservices. Proc 26th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems, p.167-181. https://doi.org/10.1145/3445814.3446693

Zhao HY, Han ZH, Yang Z, et al., 2020. HiveD: sharing a GPU cluster for deep learning with guarantees. Proc 14th USENIX Symp on Operating Systems Design and Implementation, p.515-532.

Zhao M, Cabrera J, 2018. RTVirt: enabling time-sensitive computing on virtualized systems through cross-layer CPU scheduling. Proc 13th EuroSys Conf, Article 27. https://doi.org/10.1145/3190508.3190527

Zheng L, Li XL, Zheng YH, et al., 2020. Scaph: scalable GPU-accelerated graph processing with value-driven differential scheduling. Proc USENIX Annual Technical Conf, p.573-588.

Zhou H, Chen M, Lin Q, et al., 2018. Overload control for scaling WeChat microservices. Proc ACM Symp on Cloud Computing, p.149-161. https://doi.org/10.1145/3267809.3267823

Zhou ZY, Benson TA, 2019. Composing SDN controller enhancements with Mozart. Proc ACM Symp on Cloud Computing, p.351-363. https://doi.org/10.1145/3357223.3362712

Zhu H, Kaffes K, Chen ZX, et al., 2020. RackSched: a microsecond-scale scheduler for rack-scale computers. Proc 14th USENIX Symp on Operating Systems Design and Implementation, p.1225-1240.

Zhu HS, Erez M, 2016. Dirigent: enforcing QoS for latency-critical tasks on shared multicore systems. Proc 21st Int Conf on Architectural Support for Programming Languages and Operating Systems, p.33-47. https://doi.org/10.1145/2872362.2872394

Zhu T, Kozuch MA, Harchol-Balter M, 2017. WorkloadCompactor: reducing datacenter cost while providing tail latency SLO guarantees. Proc Symp on Cloud Computing, p.598-610. https://doi.org/10.1145/3127479.3132245

Zhuravlev S, Blagodurov S, Fedorova A, 2010. Addressing shared resource contention in multicore processors via scheduling. Proc 15th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.129-142. https://doi.org/10.1145/1736020.1736036