Exploring Efficient Microservice Level Parallelism

Xinkai Wang[1], Chao Li[1], Lu Zhang[1], Xiaofeng Hou[2], Quan Chen[1], Minyi Guo[1]
[1]*Department of Computer Science and Engineering, Shanghai Jiao Tong University*
[2]*ACCESS, Hong Kong University of Science and Technology*
Emails: {unbreakablewxk, luzhang}@sjtu.edu.cn, {lichao, chen-quan, guo-my}@cs.sjtu.edu.cn, houxiaofeng@ust.hk

***Abstract*—The microservice architecture has recently become a driving trend in the cloud by disaggregating a monolithic application into many scenario-oriented service blocks (microservices). The decomposition process results in a highly dynamic execution scenario, in which various chained microservices contend for computing resources in different ways. While parallelism has been exploited at both the instruction/thread level and the task/request level, very limited work has been done with the grain-size of a microservice. Current parallel processing solutions are sub-optimal as they neither capture the unique characteristics of microservices nor consider the uncertainty arises in the microservice environment. In this work we introduce microservice level parallelism (MLP), a technique that aims to precisely coalesce and align parallel microservice chains for better system performance and resource utilization. We identify major issues that prevent servers from effectively exploiting MLP and we define metrics that can guide MLP optimization. We propose v-MLP, a volatility-aware MLP that is able to adapt to a highly heterogeneous and dynamic microservice environment. We show that v-MLP can reduce tail latency by up to 50% and improve resource utilization by up to 15% under various scenarios.**

***Index Terms*—Microservice, Parallelism, Request Management**

I. Introduction

With an emphasis on agile development and scalable deployment, microservice has recently become a key trend in building modern cloud applications [19], [37], [45]. In general, the microservice design approach is about disaggregating conventional monolithic applications into massive scenario-oriented microservices communicating through light-weight network protocols. Each microservice is self-contained, encapsulating its own code, data, and dependencies in a separate context. Considering the ability of microservice in building and running scalable applications, many IT companies such as Microsoft [24] and Alibaba [10] are actively embracing this new software development paradigm.

Microservice applications have several important features. Firstly, the functionality of microservice application is decomposed. Unlike a monolithic application in Figure 1(a) where each request triggers the entire application, each microservice is supposed to accomplish a specific function and a request only invokes related microservices. In addition, microservices have different invocation patterns, as shown in Figure 1(b). The invoked microservices may form directed acyclic graphs (DAG), where each vertex represents a microservice and the edge represents caller-callee relationship between any two microservices. During runtime, microservice DAG generally follows topological sorting and produces a chain-structured execution sequence. Moreover, due to its fine-grained functionality and chained structure, microservices are widely reused to serve different requests. The function of microservices is independent of the function of requests, thus offering interoperability across the application boundary.

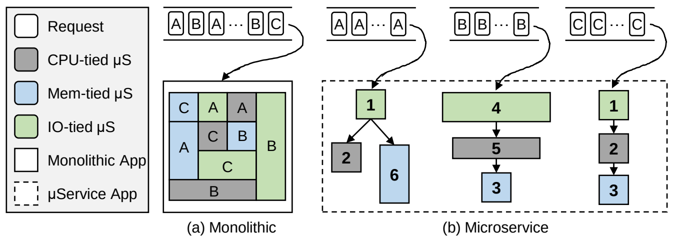


Fig. 1. Execution characteristics of microservice applications (The numbers represent different types of microservices)

In spite of various advantages, emerging microservice applications also pose new challenges for request scheduling and resource management. Today, parallelism has been exploited at various levels of the system design for better performance and efficiency. However, there is still a critical void between the existing macroscopic request scheduling (optimized for server utilization) and the microscopic instruction scheduling (optimized for chip performance). As we enter the cloud-native era, the pressure for increased performance and cost-efficiency in data centers drive us to think about more aggressive parallelism optimization at the microservice level.

In this paper we consider the problem of extracting parallelism from parallel microservice chains invoked by various requests. We call this new form of parallelism as *Microservice Level Parallelism* (MLP). There are two prerequisites that lay the foundation for MLP. Firstly, the chained structure splits each request into several independent phases, i.e., microservices, reducing the minimum execution granularity. Secondly, service alignment can be planned in advance since both invocation patterns and the resource types that each microservice demand can be foreseen. In our mind, MLP looks to harness the computing power of cloud-native infrastructures in microservice scenarios. Servers can exploit MLP by processing requests in parallel and strategically aligning multiple associated microservice chains in a fairly precise manner.

Importantly, exploiting MLP can be non-trivial due to the inherent uncertainty and dynamicity present in microservice environment. we observe that microservice applications are susceptible to the influence of different types of software and architecture issues. (detailed in Section II). First, the duration of a microservice can change greatly under different user requests. Second, microservices are differently sensitive to resource interference and contention. Worse, microservices are faced with many stochastic noises widely spread in data centers such as non-negligible communication time [27].

Existing parallel processing designs cannot adapt to the above complex microservice issues. Traditional user request scheduling schemes [11], [17], [36] use general heuristics to manage server resources in a data center while ignoring the performance-resource relationship of various microservices. Although a few microservice-aware solutions [20], [26] consider the heterogeneity of microservices, they ignore critical features such as the chained execution pattern and the uncertainty that arise in the environment. With insufficient MLP, a server node will be under-utilized due to the long time interval between the caller and callee microservices. In addition, system resources (CPU, memory, I/O, etc) are more likely to be misused as a result of increased contention and interference among co-located microservices.

To better exploit MLP, it is important to take into account the uncertainty factors and enhance the robustness of microservice scheduling. We develop a solution to analyze and tackle the uncertainty issue in microservice environment. We define *volatility of requests* (V_r), which reflects the degree of uncertainty of invoked microservices. Based on the new metric, we devise volatility-aware MLP (v-MLP) to allow for a more intelligent management of parallel microservice chains. It serves as an interface layer that bridges the high-level user request handler and the low-level server management system.

Specifically, v-MLP mainly comprises two parts: a *self-organizing* module and and a *self-healing* module. Firstly, the self-organizing module of v-MLP takes into account request characteristics to coalesce complementary microservices from different microservice chains. To ensure SLO for requests with different V_r, we use a conservative approach when estimating the execution time of a request. Secondly, the self-healing module of v-MLP aims to handle uncertain disturbances and allow the system to restore its ideal execution state. For example, we put certain microservices in a delay slot so that they can be advanced to take advantage of the idle resources.

We adopt a trace-driven evaluation approach to thoroughly analyze v-MLP. We use two open-source microservice benchmarks and three realistic workload patterns as input. We compare v-MLP with four SOTA schemes on five types of requests with different degrees of volatility. We show that v-MLP could reduce the tail latency by up to 50% and improve the resource utilization by up to 15%.

This paper makes the following contributions:

1) **Analysis:** We examine the performance implications of both program logic and resource budget in the microservice environment. We also investigate the performance uncertainty faced by microservices in the real world. We show that microservice applications present a new challenge for request scheduling and concurrency optimization in data centers.
2) **Design:** We envision the potential of Microservice Level Parallelism (MLP), a well-aligned microservice execution pattern for better resource utilization. We present MLP as an abstraction layer in the existing computing stack. We propose v-MLP, which considers the uncertainty issue of microservice applications.
3) **Evaluation:** We validate the effectiveness of our design with extensive evaluation. We build a proof-of-concept system with realistic traces as input. We show that v-MLP can greatly outperform the SOTA request scheduling schemes in terms of both efficiency and performance under various workloads and situations.

The rest of the paper is organized as follows. Section II performs workload characterization and further motivates our work. Section III proposes the concept of microservice level parallelism (MLP) and describes our solution v-MLP. Section IV introduces our evaluation methodology. Section V presents evaluation results. Section VI discusses related work and Section VII concludes this paper.

II. Workload Characterization

In this section, we conduct an in-depth analysis of microservice applications with representative benchmark suites from both academia and industry. Our characterization reveals three types of unique properties of microservice applications. We demonstrate the challenges of managing microservice requests in such a complex environment.

A. Impact of Application Heterogeneity

In this work we first investigate the variation of microservice execution time under various user invocations. The variation is often caused by the differences in actual execution logic of the microservice program.

We experiment with *TrainTicket*, an industrial open-source microservice benchmark [46]. It implements a microservice-based railway ticketing application that supports two types of requests, i.e., *Advanced Ticketing* and *Basic Search*. We deploy *TrainTicket* with *docker swarm* on a cluster and the detailed hardware configuration is presented in Section IV. We select six representative microservices, provide them with abundant resource, and invoke them respectively using different types of request. In addition, we use two sub-systems to obtain experimental data. By collecting the tracing data with Zipkin [2], we can get the response time of requests and the execution time of each microservice. In the meantime, we monitor the resource usage of each microservice with Prometheus [5] and cAdvisor [4]. In the following, we repeat each experiment 100 times and report the average results.

In Figure 2, we report the cumulative distribution function (CDF) of a microservice's execution time. We observe that the execution time distribution varies greatly. For example, the execution time of *order* almost doubles in the worst case. This reflects a wide difference in the program execution logic of a invoked microservices.

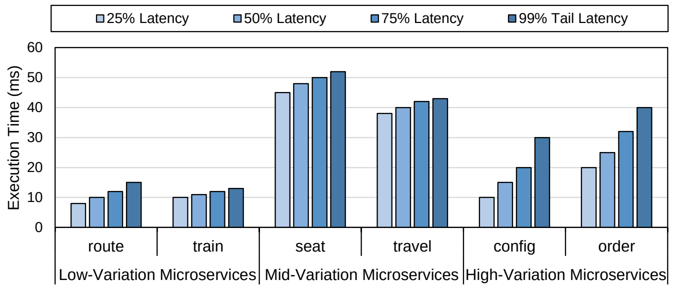


Fig. 2. Impact of application heterogeneity on execution time of microsevices

The variability of execution logic is a inherent nature of microservices. Based on the results, we can divide our microservice workload into three types:

1) *Low-variation microservices*: The largest variation in execution time is less than 15%.
2) *Mid-variation microservices*: The largest variation in execution time is between 15% and 45%.
3) *High-variation microservices*: The largest variation in execution time is larger than 45%.

B. Impact of Resource Provisioning

In addition to the inner execution logic of microservices, many outer factors such as resource constraints may affect microservice execution time as well. In this subsection, we discuss the resource-performance relationship of microservices and we present three key observations.

Observation 1: Microservices have unique resource requirements and can be co-located to improve utilization.

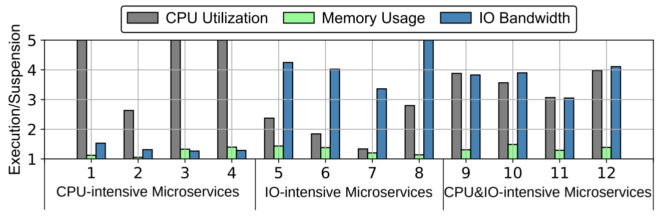
We first study the resource usage behavior/characteristics of different types of microservices. We mainly consider CPU, memory, and IO bandwidth resources and analyze their influence on application performance. Figure 3(a) shows the ratio of the resource demand in execution state to the resource demand in suspension state of 12 microservices obtained from *SocialNetwork*, an academic open-source microservice benchmark [13]. In the experiment, we provide all the evaluated microservices with abundant resources and record their resource usage using our monitoring sub-systems.

In contrast to monolithic applications that are generally bounded by mixed resource types, we observe that a microservice faces fewer resource bottlenecks upon execution. For example, memory capacity is not a bottleneck of the studied microservices. In this work, microservices can be CPU-intensive, IO-intensive, or CPU&IO-intensive.

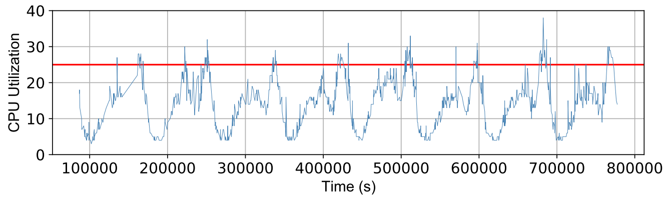
Observation 2: The resource requirements may not be always met under highly dynamic microservice traffic.

Like conventional cloud, the microservice environment also faces resource provisioning issues. Figure 3(b) shows the server utilization of a container running microservice in a real data center. The results are based on our analysis of an open-source Alibaba cluster log [3] that encompasses an eight-day trace of containers from a production cluster.

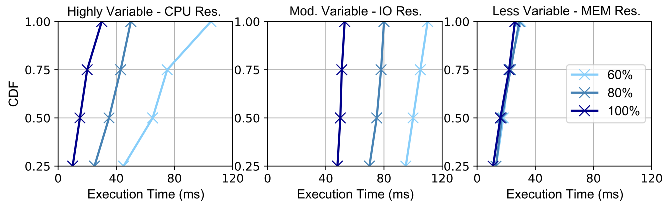
It is evident that the workload fluctuations are significant and there are many peaks caused by frequent traffic surges. Considering that resource over-subscription has become a common practice for cloud providers to improve cost-efficiency [42], the resource demand peaks may not be always satisfied. Resource over-subscription makes sense if the system utilization is low. However, when the microservice traffic increases, it would cause resource supply shortage and unpredictable performance interference.



(a) Various resource dependency of microservices



(b) Highly dynamic traffic faced by microservices



(c) Different sensitivity to system resource capping

Fig. 3. Microservice applications are susceptible to the influence of several software and hardware characteristics

Observation 3: Microservices are differently sensitive to resource shortage from the perspective of performance.

As stated above, microservices require different types of resources and resource shortage exists. Therefore we further investigate the relationship between microservices performance and resource budget using *SocialNetwork*. Figure 3(c) shows the CDF of the execution time of different types of microservices under varying resource budget. We manually lower the resource supply to create resource contention.

We find that the execution time of microservices is differently sensitive to resource restrictions, thus increasing the difficulty of predicting system performance. Microservices can be categorized into three types by looking at how the mean value and variance of execution time change with resource capping. 1) *Highly variable microservice*: Resource shortage/contention would increase both the mean value and variance of execution time. 2) *Moderately variable microservice*: Resource shortage/contention would increase the mean value while keeping the variance unchanged. 3) *Less variable microservice*: Both the mean value and the variance of execution time are not affected. It is uncommon in microservice scenarios.

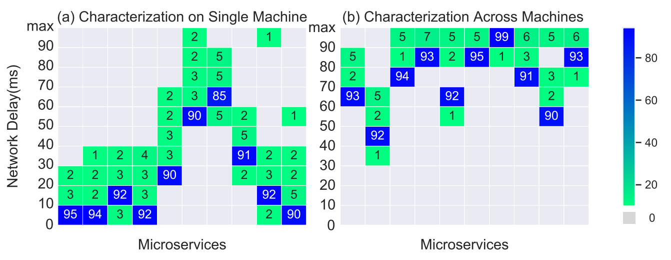
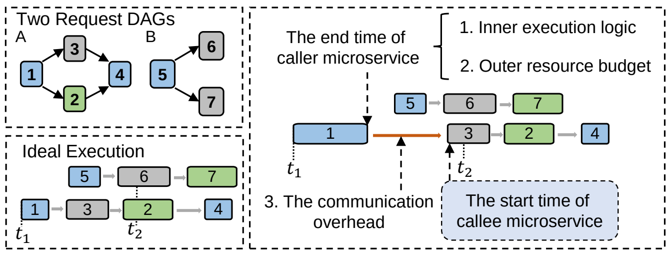


Fig. 4. Highly uncertain communication overheads


Two Request DAGs
A
B
Ideal Execution
The end time of caller microservice
1. Inner execution logic
2. Outer resource budget
3. The communication overhead
The start time of callee microservice
Fig. 5. Summary of design challenges in microservice scheduling

Summary: The three observations above reveal that the performance implication of microservice resource usage can be complex, which poses new challenges for us to accommodate scale-out microservice workload.

C. Impact of Communication Overhead

The inner execution logic and outer resource constraints are relatively predictable. Apart from these factors, there are also stochastic noises in the microservices environment. In particular, we characterize the communication time between microservices to study the uncertainty of microservices execution time. We perform two sets of experiments. Figure 4(a) deploys *TrainTicket* on a single machine with *docker-compose*. We generate 100 requests and record the communication time of 10 callee microservices from the caller microservice. Figure 4(b) deploys the studied callee microservice on one machine and the other microservices on another machine with *docker swarm* to study the communication time across machines. For each callee microservice, we respectively generate 100 requests and record the communication time of the callee microservice from its caller.

In Figure 4, we present the distribution of microservice communication time. The digits on each block represent the frequency of execution within certain latency range. We find that the average communication time between microservices on a single machine is significantly lower than that across machines. It indicates that communication time variations on a single machine are more stable than those across machines due to the longer network links. Meanwhile, there is a possibility that communication time increases (as shown in the green blocks) due to network congestion or changed routing. Without proper treatment, the above uncertain communication time issue would disturb normal system operation and cause degraded scheduling effectiveness.

D. Summary of Design Challenges

Current utilization-based schedulers are not prepared for the microservice environment for two reasons. First, they cannot accurately predict the end time of a caller microservice. Based on our analysis in Subsections II-A and II-B, the end time of the caller microservice is affected by both the actual execution logic triggered by the request and the resource budget at the time. Second, they often overlook the uncertainty of communication overhead between the caller and callee. As discussed in Section II-C, the communication issue represents a non-deterministic factor which complicates system management.

Figure 5 further illustrates the challenge of microservice scheduling. Assuming that there are two requests A (involving microservices 1-4) and B (involving microservices 5-7). The two requests can finish without any resource contention/interference in an ideal situation given by the bottom left of the figure. Nevertheless, the scheduler could estimate the end time of caller microservice 1 in a wrong way. Even worse, the communication between microservices 1 and 3 could experience unexpected latency. In this case, the start time of microservice 3 is mispredicted and the actual execution of the two requests no longer maintains the ideal resource efficiency. As the right part of the figure shows, there are direct resource contention between microservices 3 and 6, and the system incurs performance degradation at the timestamp t_2.

In sum, scheduling microservice requests and ensuring their efficient execution can be non-trivial. The crux of the problem is *two-fold*. On the one hand, we need to know how different microservices should be coalesced. On the other hand, we must ensure fairly accurate alignment throughout the process.

III. Microservice Level Parallelism

A. Definition of Microservice Level Parallelism

According to our analysis, at the microservice level, the workload often exhibits irregularity in behavior and uncertainty in demand. Being able to fine tune the system at this granularity is of great importance for emerging cloud-native platforms. Traditional task scheduling methods need to be re-examined to unleash the full potential of the system.

In this work we propose *Microservice Level Parallelism* (MLP), where various chained microservices generated by different requests are the basic units of parallelization. MLP intends to narrow the wide gap between macroscopic request scheduling and microscopic instruction scheduling. There is an opportunity of improved performance with well-aligned execution of microservice-based applications.

In Table I we compare MLP with three forms of parallelism. Overall, they correspond to different granularities of parallelism. At lower levels, contemporary computer systems support Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP). ILP targets at exploiting pipeline scheduling for efficient instruction execution, while TLP focuses on executing many threads to improve hardware utilization.

TABLE I
ILP vs TLP vs RLP vs MLP

Parallelism	**ILP**	**TLP**	**MLP**	**RLP**
Scheduling Level	Chip Level		System Level	
Granularity	Instruction	Instruction Stream	Microservice	Monolithic Application
Key Opti. Approach	Temporal	Spatial	Temporal	Spatial

At a higher level, Request Level Parallelism (RLP) plays a dominant role for data center-scale computing in the cloud era. It mainly considers the parallel execution of enormous online requests across machines.

MLP is orthogonal to existing parallelism models. Basically, it is very much like pipeline parallelism, which organizes a program as a sequence of execution stages. In each stage, we process certain amount of data and then pass the processed data to the next stage. Different from ILP and TLP, the scheduling granularity of MLP becomes much larger. Each user request involves a chain of microservices which may spread across computing nodes and a client needs to go through multiple microservices for processing. In addition, MLP is not RLP and it complements RLP perfectly. RLP can be viewed as managing gigantic service program with little need for communication or synchronization. In the microservice scenario, each larger problem (request) are divided into much smaller program logic (microservices) and microservices are more sensitive to resource interference and stochastic noises. In sum, MLP aims to enhance system efficiency by taking into account the interrelationship of microservices and their environment when making scheduling decisions.

B. Volatility of Request

Considering the volatile behavior of microservices, optimizing MLP is about giving the system more control on the workload in a highly dynamic execution environment.

Based on our characterization, we define *volatility* of requests (V_r), indicating the likelihood of the request to deviate from its ideal execution conditions. For a request that invokes n microservices, its volatility is given by:

$$V_r = \alpha \times \sum_{i=1}^{n} I_i \times S_i \times C_i / n$$

where I is the variability related to inner program execution logic (detailed in Section II-A), S is the sensitivity to resource budget (detailed in Section II-B), and C represents the communication overhead levels (detailed in Section II-C). In Table II we show the intensity scale of different design considerations.

TABLE II
SELECTION RANGE OF VOLATILITY TERMS

Abbr.	**Value Range**	**Descriptions**
I	1(low) - 3(high)	Inner Logic Variability
S	1(low) - 3(high)	Sensitivity to Resource
C	1-3: Var(RTT) from 100 to 400	Communication Overhead

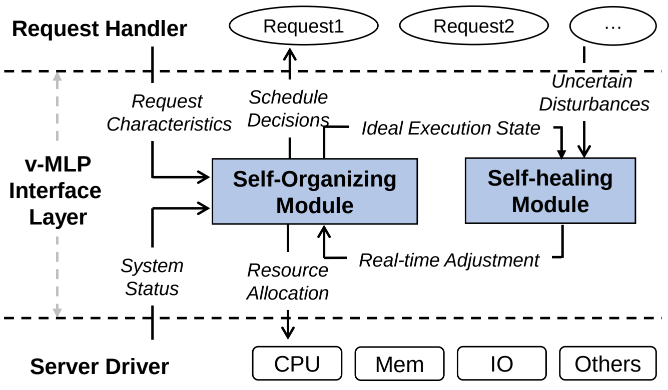


Fig. 6. Overview of volatility-aware Microservice Level Parallelism

Here, low volatility implies that the start time of the callee microservice is more predictable and less volatile. It is much easier for one to maintain efficient execution of microservice chains that have low volatility. For heterogeneous requests with different V_r, understanding the volatility of requests allows the scheduler to make prudent decisions that could enhance system efficiency. We introduce our solution for efficiently exploiting MLP in the following subsection.

C. v-MLP: Volatility-aware Microservice Level Parallelism

With an understanding of user request, servers can better accommodate microservices. We devise volatility-aware MLP **(v-MLP)** to allow for a more intelligent management of parallel microservice chains. As shown in Figure 6, v-MLP serves as an interface layer that bridges the high-level user request handler and the low-level server hardware. It could reduce the gap between the upper request scheduling and the lower instruction scheduling. Specifically, v-MLP automates the about process through two elements: a self-organizing module and and a self-healing module. The two modules cooperate with each other.

1) **Self-organizing Module:** The self-organizing module takes into account request features to coalesce microservices from different microservice chains. It examines the dependency of different microservices and analyzes their resource consumption characteristics to form a ideal execution pipeline on the server. It aims to maximize system utilization while avoiding resource contention.
2) **Self-healing Module:** The responsibility of v-MLP's self-healing module is to handle any uncertain disturbances during execution. It aims to restore the system to its ideal execution status, if necessary. The module uses a delay slot mechanism to reduce server idleness. Meanwhile, it adjusts microservice resource consumption on the fly through a resource stretch mechanism.

D. Interface Layer

As an interface layer, v-MLP abstract away the complexity of the underlying hardware. During runtime, v-MLP interact with the system to gain necessary information for scheduling.

TABLE III
RESOURCE MONITORS AND CONTROLLERS

Resource Type	Monitor	Controller
CPU	dockerstats	cgroups cpuset
Memory	dockerstats	cgroups memory.limit_in_bytes
IO Bandwidth	dockerstats	cgroups net_cls

It features a local monitor and a control toolkit on each container that runs microservices. We use several ready-to-use system interfaces and methods to obtain the execution statistics of microservices. Table III lists the related tools for monitoring and controlling the hardware resources such as CPU cores, memory usage, and IO bandwidth. We uses open-source distributed tracing systems such as *Jaeger* [1] and *Zipkin* [2] to obtain the execution time of each microservice and the response time of each request. The information collected is further fed into the self-organizing module to make decisions and is stored as historical traces for future scheduling.

v-MLP execute resource management decisions with special care. It can control the resource availability of each container through mature tuning knobs and configuration files. For example, it can manipulate the CPU, memory, and IO usage of each container by restricting the processes and threads running the containers. It can also limit the CPU upper bound of containers by writing the expected value to *resources-limits-cpu*. Since containers provide virtually isolated environments with namespace, microservices can execute independently within containers wherever there are enough resources.

E. Self-organizing Module

The self-organizing module of v-MLP takes into account request characteristics to coalesce microservices from different microservice chains. Specifically, we denote the request DAG as $r = (V_s, e)$ where V_s contains n microservice vertices and e is their dependency. For each microservice $s_i \in V_s$, we describe it using a matrix $s_i = [u_{cpu}^T, u_{mem}^T, u_{io}^T, l^T, \Delta t^T]$, where column u_x denotes the usage of resource x, column l denotes the machine load, and column Δt denotes the maximum execution time slack. Each row of s_i is a historical execution case of this microservice. In the meantime, we can extract m microservice chain choices for each request as $c_j = (s_j^1, s_j^2, \cdots, s_j^n)$ following topological sort.

We summarize the scheduling process of the self-organizing module in Algorithm 1 . Firstly, it refreshes the status of each machine in the scheduling cluster with real-time data collected, which contains future resource status and microservice execution states. Then we define a reorder ratio R to sort the waiting queue based on volatility V_r of requests. R is a comprehensive consideration of SLA requirement and two classic scheduling policies, i.e. *First Come First Serve (FCFS)* (t_{arr} is the arrival time of request) and *Shortest Job First (SJF)* (Δt_0 is the smallest element in Δt column of s_i) to assign priority for waiting requests. By normalized factor α, R is a normalized value between $(0, 1)$. Requests with higher R would be popped from the waiting queue and examined execution status earlier.

Algorithm 1: Algorithm for Self-organizing Module

```
Data: $\mu$Service Metrics & Timestamp $t$
Result: $\mu$Service Pipeline
for $r_i$ in waiting queue do
    Traverse machine status and refresh;
    if $V_r \leq 0.3$ then
        for $s_k$ in each $c_j$ of $r_i$ do
            for each machine $m_n$ and load do
                Compare $t \rightarrow t + \Delta t : l_{res} \geq u_{res}$;
                If so, assign $s_k$ to $m_n$;
            end
        end
    end
    if $0.3 \leq V_r \leq 0.7$ then
        for $s_k$ in each $c_j$ of $r_i$ do
            let $\Delta t = 50\%$ latency of $x\%$ executions;
            Repeat traversing machines;
        end
    end
    if $V_r \geq 0.7$ then
        for $s_k$ in each $c_j$ of $r_i$ do
            let $\Delta t = 99\%$ latency of $x\%$ executions;
            Repeat traversing machines;
        end
    end
    if This request is totally assigned then
        Go to next request;
    else
        Switch $r_i$ with $r_{i+1}$;
    end
end
```

$$R = \alpha \times \frac{V_r \times \text{SLA} \times t_{arr}}{\Delta t_0}$$

Our algorithm traverses the waiting queue and the microservice chains of the popped request for scheduling. In the scheduling process, microservices would be assigned to proper machines with best efforts by comparing with the calculated resource budget within execution time Δt. We divide the range of volatility into three parts: low (0 $\sim$ 0.3), medium (0.3 $\sim$ 0.7), and high (0.7 $\sim$ 1). For requests with low V_r, Δt is directly determined by historical value. Meanwhile, Δt of microservices invoked by request with medium V_r is approximated by $\Delta t = 50\%$ mean latency of $x\%$ executions and Δt of those with high V_r is approximated by $\Delta t = 99\%$ tail latency of $x\%$ executions. Here x is a metric considering SLA requirement and volatility: $x \propto \text{SLA} \times V_r$, where x is a value between 1 and 100. Then each microservice is assigned to proper machines with best efforts within the approximated execution time Δt. If the scheduling request is not able to execute without contention, the algorithm delay its execution and advances the next request in the waiting queue. The algorithm ends until the cluster is saturated.

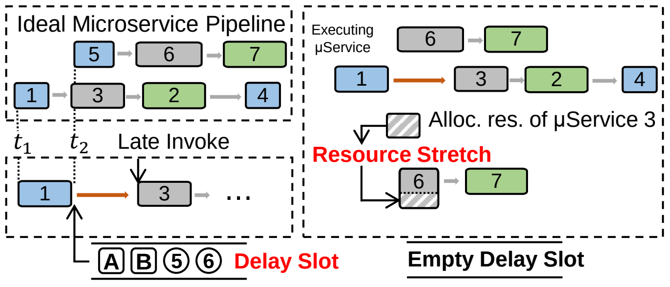


Fig. 7. Two mechanisms in self-healing module of v-MLP

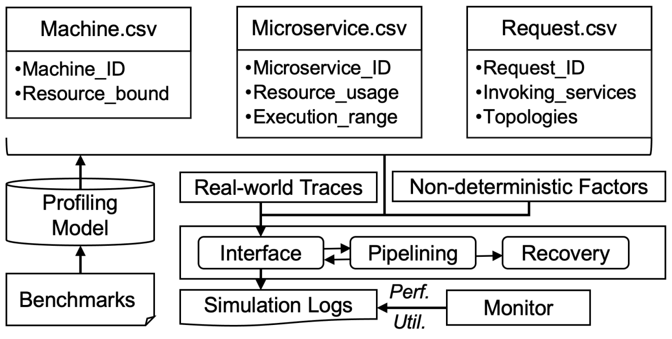


Fig. 8. The workflow of trace-driven evaluation

F. Self-healing Module

With the self-organizing module, we aim to produce an ideal microservice execution pipeline without resource contention as shown in Figure 7. However, as stated in Section II-D, late invocation of microservices would drive the scheduling results out of control. In order to handle uncertain disturbances in real-time execution and allow the system to restore its ideal execution state, the self-healing module is equipped with two mechanisms: delay slot and resource stretch.

We design a delay slot mechanism to handle the resource vacancy in the pipeline. As shown in Figure 7, due to the late invoke of microservice 3 (3 is predicted to start at t_2), the pipeline would stall for a period and resource in this period would be wasted. Delay slot contains two kinds of candidates: requests (A&B) and microservices (5&6). Requests are the latter in the waiting queue following the reorder ratio R. Microservices are the waiting ones of executing requests without dependence on executing ones and late-invoking ones. Without dependence on these, candidates in the delay slot would not conflict with executing ones or interfere with the relocation of late-invoking ones. After the relocation of the candidates in the delay slot, the self-healing module refreshes the system status and pipeline arrangement for further scheduling. Equipped with delay slot mechanism, v-MLP can fulfill the resource vacancy in the pipeline and accelerate the waiting requests.

Sometimes the delay slot is lack of candidate microservices since the system load may drop or fluctuate. Therefore, we design a resource stretch mechanism for the empty delay slot scenarios. We adjust the resource usage of executing microservices using the resource controllers in Table III. We give priority to microservices following two principles: 1) *earliest deadline first* (EDF); (2) high variability first (as depicted in Figure 3(c)). As shown in Figure 7, we monitor the idle resources allocated to late-invoking microservices and reassign them to the executing ones with higher priority.

IV. Experimental Methodologies

We adopt a trace-driven evaluation approach to analyze the large design space of v-MLP. The workflow of our trace-driven evaluation is shown in Figure 8. Our simulator takes realistic profiling data of open-source benchmarks as input. To obtain the necessary execution traces, we run microservice workload on a server cluster as described in Table IV.A. We deploy a microservice benchmark (*TrainTictket* (TT) [46]) from the industry and a microservice benchmark (*SocialNetwork* (SN) [13]) from the academia. The configurations of our simulation platform is shown in Table IV.B.

TABLE IV
Testbed Configurations

A. Workload Characterization Configurations	
Cluster	4 worker nodes(total 24 cores) + 1 manager node
Server	Dell R730, Intel(R) Xeon(R) E5-2620 @ 2.40GHz
Memory	32GB, DDR4 for each node
Network	1000 MB/s for each node
HostOS	Ubuntu 18.04.5 LTS, docker 18.06.3-ce
B. Simulation Platform Configurations	
Server	Dell R740, Intel(R) Xeon(R) Gold 5218 @ 2.30GHz
Memory	64GB × 2, DDR4
HostOS	Ubuntu 18.04.5 LTS, docker 20.10.3

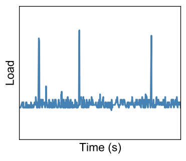
TABLE V
Evaluated requests descriptions

Category	Request	Descriptions
High V_r	compose-post	Request to compose post in SN
	getCheapest	Request of advanced search in TT
Mid V_r	basicSearch	Request of basic search in TT
Low V_r	read-home-timeline	Request to get home timeline in SN
	read-user-timeline	Requets to get user timeline in SN

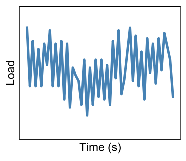
We conduct experiments with user requests of different volatility. We consider five requests from three categories. As shown in Table V, the *High* V_r category includes the `compose-post` request from *SocialNetwork* and the `getCheapest` request from *TrainTicket*. Microservices invoked by the above two requests are highly variable. The *Mid* V_r category includes the `basicSearch` request from *TrainTicket*. The *Low* V_r category, representing more stable microservices, includes the `read-home-timeline` and the `read-user-timeline` requests from *SocialNetwork*. In our experiment, we ensure that different types of requests in one category take up the same portion in a request stream.

We invoke each request stream using the workload patterns as shown in Figure 9, which is drawn from a realistic datacenter [3]. L1 is a pulse-like workload peak, L2 is fluctuating workload, and L3 is a periodic workload with wide peaks. The maximum workload rate is 1000 request/s.

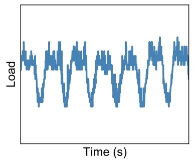
Table VI summarizes our evaluated request scheduling



(a) L1: pulse-like



(b) L2: fluctuating



(c) L3: periodic

Fig. 9. Workload patterns in realistic datacenters

TABLE VI
EVALUATED SCHEMES DESCRIPTIONS

Category	Scheme	Descriptions
Simple Scheduler	FairSched	*FCFS, Allocate equal resource*
	CurSched	*FCFS, Allocate by current load*
Advanced Scheduler	PartProfile	*Prior., Allocate by performance profile*
	FullProfile	*Prior., Allocate by overall profile*
MLP Scheme	v-MLP	*Our Proposal*

schemes. We consider two important baselines: simple scheduler and advanced scheduler. Simple scheduler is a group of trivial methods without consideration of historical data. *FCFS* represents the First Come First Serve algorithm for request waiting queue. *FairSched* represents the fair scheduling methods that give each microservice equal resources [22]. *CurSched* allocates resources according to the current load of containers. Advanced scheduler refers to a series of methods that schedule microservices based on their historical and current status. *Prior.* represents reordering of the waiting queue according to specific priorities. *PartProfile* manages microservices to machines simply depending on the partial profiling [26]. *FullProfile* represent the state-of-the-art workload-specific strategies that allocate resources based on the overall profiling of the whole applications [11].

V. EVALUATION RESULTS

This section quantifies the effectiveness, efficiency, and performance of v-MLP on a wide range of workload configurations and different scheduling schemes.

A. Effectiveness Analysis

MLP aims to achieve aligned execution of microservices. A basic requirement is to maintain the QoS requirement, which indicates the effectiveness of our design. In Figure 10, we compare the QoS violation rate of different schemes and we normalize the violation rate of other schemes to v-MLP. We study the effectiveness of scheduling with different levels of V_r requests and different workload patterns. We find that among all the schemes, v-MLP greatly outperforms simple schedulers and *FullProfile* while exceeding *PartProfile* less. Considering that *PartProfile* focuses on QoS violation, we believe v-MLP is capable of maintaining QoS in real execution.

As for volatility consideration, Figure 10 shows that different baseline schemes are relatively good at managing requests with low V_r. In contrast, v-MLP excel at keeping QoS with high V_r. As for workload consideration, we find that it is more difficult to maintain QoS under workload patterns L2 and L3. Compared to the pulse-like workload (L1), L2 and L3 are more fluctuating to the extent that normal schemes cannot manage the requests very well. Overall, we show that v-MLP is effective in maintaining QoS and we evaluate the efficiency of v-MLP in the next section.

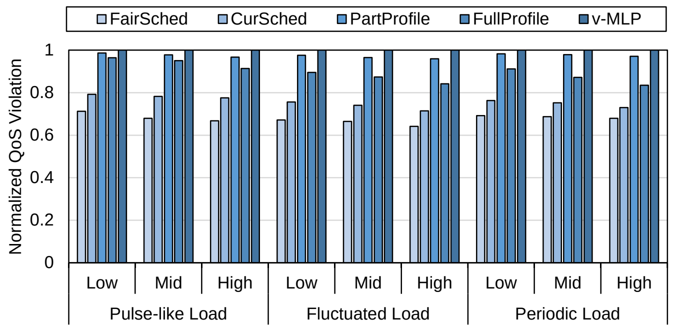


Fig. 10. Effectiveness: comparison of normalized QoS violation

B. Efficiency Analysis

We evaluate the efficiency of MLP by looking at the resource utilization of the system. The simulated cluster includes 100 machines and we consider CPU, memory, and IO bandwidth resources of each machine. We observe that all the three workload patterns contain load peaks and we examine the adjustment on resource utilization with workload peaks. In Figure 11, we present the overall resource utilization U of cluster, which is defined as:

$$U = \frac{\sum_{\#\text{nodes}}(u_{cpu} + u_{mem} + u_{io})}{\#\text{resource type} \times \#\text{nodes}}$$

u_x is the utilization rate of resource x of one node and U indicates the average efficiency of the cluster at each timestamp. Here we consider 3 types of resources and the number of nodes is 100. The whole scheduling process lasts 100 seconds and the load peak arrives at the 40th second.

As Figure 11 shows, v-MLP greatly outperforms simple schedulers and slightly outperforms advanced schedulers at the beginning. When the workload peak arrives, the utilization of all schedulers increases at once due to the increased number of scheduling entity. However, after a while, the utilization of all schedulers decreases due to the mismatch of fully-allocated resources and the complex dependency between microservices. We find that the utilization of simple schedulers decreases the most due to the simple strategies. The advanced schedulers show lower utilization mainly due to ignorance of dependency of microservices. On the contrary, v-MLP experiences the same increase of utilization at beginning. It can restore the normal utilization rate because the self-organizing module considers the inner dependency of requests and makes visionary scheduling decisions.

C. Performance Analysis

We also evaluate the performance of v-MLP by measuring the end-to-end latency of requests and the throughput of the system. The end-to-end latency is a major consideration of

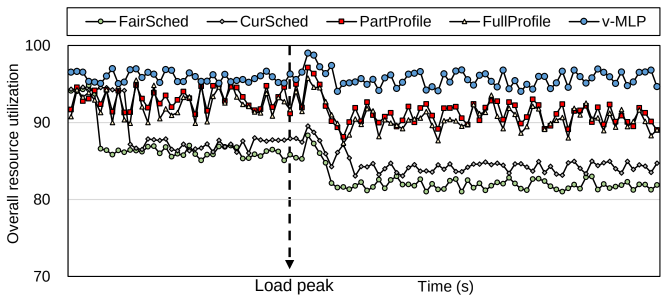


Fig. 11. Efficiency: comparison of resource utilization with workload peaks

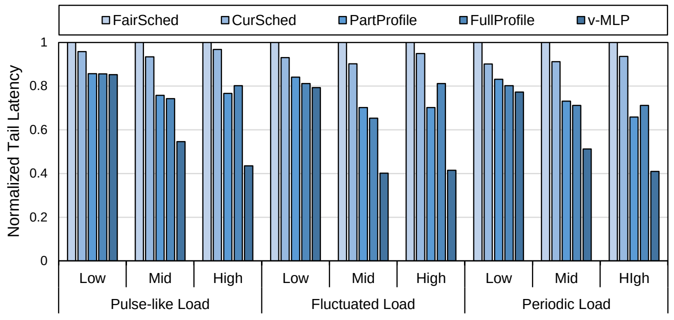


Fig. 13. Performance: comparison of normalized tail latency

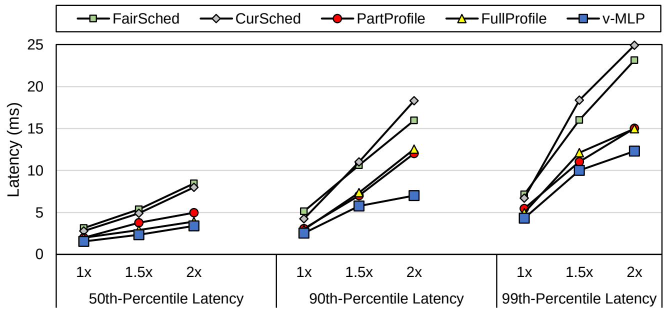


Fig. 12. Performance: comparison of latency distribution

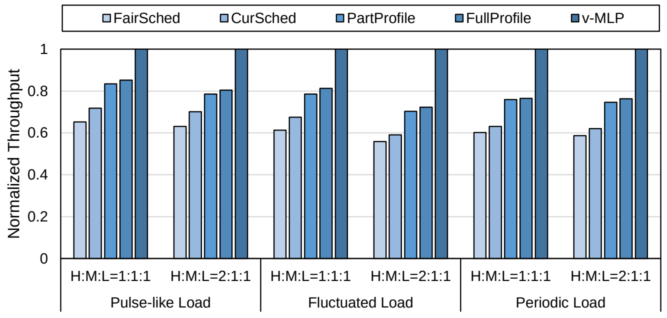


Fig. 14. Performance: comparison of normalized throughput

request scheduling [26] and we evaluate the performance of different schemes under different workload levels. We adjust the queries per second (QPS) in the simulation to proportionally scale the workload levels. We consider a mixed request stream which is composed of requests of different V_r values.

As shown in Figure 12, v-MLP greatly outperforms other schemes at each percentile point of the distribution. In addition to the 50th-percentile and the 90th-percentile latency, MLP is also good at improving the 99th-percentile tail latency. The key reason is that it considers the fairness of the request waiting queue. Under high workload levels, v-MLP shows better effectiveness since it contains the self-healing module for handling the uncertain situations.

Aside from latency distribution, we also evaluate more detailed performance with tail latency [16]. Figure 13 presents the tail latency under different workload patterns. We separate the mixed request stream into three independent streams composed of three levels of V_r requests. To be specific, we set the tail latency of FairSched as one and normalize all other schemes. It is evident that simple schedulers are almost similar, advanced schedulers are better, and v-MLP performs the best. For the request stream with low V_r, the gap between the five schemes is small since requests of low V_r are easier to schedule and the strategies of v-MLP for them are trivial. However, v-MLP outperforms much more for the request streams with mid and high V_r, where v-MLP adopts the maximum time slack and self-healing methods. It shows that v-MLP is better at managing requests in a dynamic datacenter environment rather than a static cluster.

Lastly, we evaluated the throughput improvement of v-MLP. The throughput is calculated as the number of finished requests within certain scheduling period (100s). We normalize the results of other schemes with v-MLP. We evaluate the throughput under different types of request streams by adjusting the ratio of high V_r requests. As shown in Figure 14, v-MLP greatly outperforms the other schemes, especially with a higher ratio of high V_r requests due to their tailed management strategies. Also, v-MLP performs better under fluctuating workload since the self-healing module enables fairly accurate aligned execution in a highly dynamic environment.

VI. Related Work

A. Resource Management in Datacenter

In the last decades, there are many works on datacenter resource management [9], [25], [29], [30], [32], [39], [44]. For example, at the infrastructure level, researchers propose to coordinate power supply management and computing system management to achieve high sustainability and high availability [18], [28]–[32]. At the system level, there are many papers on mitigating resource contention [12], [34], [40] and improving server utilization [8], [11], [36], [41]. Nevertheless, these works do not take into account the unique behaviors of microservices and serverless functions.

B. System Research on Microservices

The microservice architecture is proposed to solve several problems of deploying monolithic applications in data centers [6], [24], [33], [43]. Consequently, microservices are becoming an important type of data center workload [13], [46]. Several proposals have focused on improving the performance or efficiency of microservices. For example, Yu et al. focused on predicting QoS violations among massive microservices [14]. Kannan et al. use time estimation to guarantee SLAs for jobs in microservice execution frameworks [26] without considering resource management. Sriraman et al. discussed optimization approaches for microsecond-scale services [37]. Chou et al. investigated dynamic power management for microsecond-scale services. Hou et al. conducted a series of researches on power-aware microservice workload management. [19]–[21]. To our knowledge, very limited work has been done in terms of managing parallel microservice chains towards higher resource efficiency.

C. Exploration of Parallelism

In the past, parallelism on various granularity has been examined in depth [15], [23], [35]. Parallelism is widely used to solve computing problems in complex conditions. There are several representative forms of parallelism: instruction level parallelism (ILP), thread level parallelism (TLP), and request level parallelism (RLP). ILP is proposed earliest and serves at the lowest level [38]. TLP looks at instruction streams across multiple processors in parallel computing environments [7]. Meanwhile, RLP refers to concurrent processing of multiple requests in the datacenters [6]. Nevertheless, very few works have considered exploiting parallelism at the microservice granularity in datacenters. In this work we show that it is rewarding to fine-tune the microservice chain in a highly heterogeneous and dynamic execution environment.

VII. Conclusion

The microservice architecture is redefining the cloud environment today and drives scalable and agile application deployment. In this paper, we show that conventional parallel processing solutions are sub-optimal as they neither capture the unique characteristics of microservices nor consider the uncertainty arises in the microservice environment. We propose Microservice Level Parallelism (MLP), a new form of parallelism that aims to improve system efficiency by coordinating microservice chains invoked by various requests. We develop a solution to analyze and tackle the uncertainty issue in a highly dynamic microservice environment. We show that our design can yield lower end-to-end latency and higher resource utilization and throughput while keeping lower QoS violations. We expect that this work will provide valuable insights for both academic and practitioners in the design of the next-generation cloud-native infrastructure.

Acknowledgment

This work is supported in part by the National Natural Science Foundation of China (No.61832006) and Shanghai S&T Committee Rising-Star Program (No.21QA1404400). We thank all the anonymous reviewers for their valuable feedback. Corresponding author is Chao Li.

References

[1] "Jaeger," https://jaegertracing.io, 2021.

[2] "Zipkin," https://zipkin.io/, 2021.

[3] "Alibaba cluster data," https://github.com/alibaba/clusterdata, 2021.

[4] "Container advisor," https://github.com/google/cadvisor, 2021.

[5] "Prometheus," https://prometheus.io/, 2021.

[6] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[7] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner, "Evolution of thread-level parallelism in desktop applications," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 302–313.

[8] Q. Chen, S. Xue, S. Zhao, S. Chen, Y. Wu, Y. Xu, Z. Song, T. Ma, Y. Yang, and M. Guo, "Alita: comprehensive performance isolation through bias resource management for public clouds," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–13.

[9] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 681–696, 2016.

[10] W. Cui, H. Zhao, Q. Chen, N. Zheng, J. Leng, J. Zhao, Z. Song, T. Ma, Y. Yang, C. Li *et al.*, "Enable simultaneous dnn services based on deterministic operator overlap and precise latency prediction," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.

[11] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 77–88, 2013.

[12] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating interference at microsecond timescales," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 281–297.

[13] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 3–18.

[14] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, 2019, pp. 19–33.

[15] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," *ACM SIGPLAN Notices*, vol. 41, no. 11, pp. 151–162, 2006.

[16] M. E. Haque, Y. H. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley, "Few-to-many: Incremental parallelism for reducing tail latency in interactive services," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 161–175, 2015.

[17] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, no. 2011, 2011, pp. 22–22.

[18] X. Hou, L. Hao, C. Li, Q. Chen, W. Zheng, and M. Guo, "Power grab in aggressively provisioned data centers: What is the risk and what can be done about it," in *Proceedings of the 36th IEEE International Conference on Computer Design (ICCD)*, 2018.

[19] X. Hou, C. Li, J. Liu, L. Zhang, Y. Hu, and M. Guo, "Ant-man: towards agile power management in the microservice era," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.

[20] X. Hou, C. Li, J. Liu, L. Zhang, S. Ren, J. Leng, Q. Chen, and M. Guo, "Alphar: learning-powered resource management for irregular, dynamic microservice graph," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 797–806.

[21] X. Hou, J. Liu, C. Li, and M. Guo, "Unleashing the scalability potential of power-constrained data center in the microservice era," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[22] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 261–276.

[23] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," *ACM SIGARCH Computer Architecture News*, vol. 17, no. 2, pp. 272–282, 1989.

[24] G. Kakivaya, L. Xun, R. Hasha, S. B. Ahsan, T. Pfleiger, R. Sinha, A. Gupta, M. Tarta, M. Fussell, V. Modi *et al.*, "Service fabric: a distributed platform for building microservices in the cloud," in *Proceedings of the thirteenth EuroSys conference*, 2018, pp. 1–15.

[25] R. S. Kannan, A. Jain, M. A. Laurenzano, L. Tang, and J. Mars, "Proctor: Detecting and investigating interference in shared datacenters," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2018, pp. 76–86.

[26] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "Grandslam: Guaranteeing slas for jobs in microservices execution frameworks," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–16.

[27] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou, "Dagger: efficient and fast rpcs in cloud microservices with near-memory reconfigurable nics," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 36–51.

[28] C. Li, Y. Hu, L. Liu, J. Gu, M. Song, X. Liang, J. Yuan, and T. Li, "Towards sustainable in-situ server systems in the big data era," *Acm Sigarch Computer Architecture News*, vol. 43, no. 3S, pp. 14–26, 2015.

[29] C. Li, Y. Hu, R. Zhou, M. Liu, L. Liu, J. Yuan, and T. Li, "Enabling datacenter servers to scale out economically and sustainably," in *Proceedings of the 46th annual IEEE/ACM international symposium on microarchitecture*, 2013, pp. 322–333.

[30] C. Li, A. Qouneh, and T. Li, "iswitch: Coordinating and optimizing renewable energy powered server clusters," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 512–523, 2012.

[31] C. Li, Z. Wang, X. Hou, H. Chen, X. Liang, and M. Guo, "Power attack defense: Securing battery-backed data centers," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 493–505, 2016.

[32] C. Li, R. Zhou, and T. Li, "Enabling distributed generation powered sustainable high-performance data center," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2013, pp. 35–46.

[33] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 412–426.

[34] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 248–259.

[35] S. Rul, H. Vandierendonck, and K. De Bosschere, "Function level parallelism driven by data dependencies," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 1, pp. 55–62, 2007.

[36] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 351–364.

[37] A. Sriraman and T. F. Wenisch, "μtune: Auto-tuned threading for {OLDI} microservices," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 177–194.

[38] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, 1991, pp. 176–188.

[39] J. Wang, C. Li, T. Wang, l. Zhang, P. Wang, J. Mei, and M. Guo, "Excavating the potential of graph workload on rdma-based far memory architecture," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022.

[40] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 607–618, 2013.

[41] H. Yang, Q. Chen, M. Riaz, Z. Luan, L. Tang, and J. Mars, "Powerchief: Intelligent power allocation for multi-stage applications to improve responsiveness on power constrained cmp," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 133–146.

[42] R. Yang, C. Hu, X. Sun, P. Garraghan, T. Wo, Z. Wen, H. Peng, J. Xu, and C. Li, "Performance-aware speculative resource oversubscription for large-scale clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 7, pp. 1499–1517, 2020.

[43] R. Zeng, X. Hou, L. Zhang, C. Li, W. Zheng, and M. Guo, "Performance optimization for cloud computing systems in the microservice era: state-of-the-art and research opportunities," *Frontiers of Computer Science*, vol. 16, no. 6, pp. 1–13, 2022.

[44] L. Zhang, W. Feng, C. Li, X. Hou, P. Wang, J. Wang, and M. Guo, "Tapping into nfv environment for opportunistic serverless edge function deployment," *IEEE Transactions on Computers*, 2021.

[45] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: Ml-based and qos-aware resource management for cloud microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 167–181.

[46] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Poster: Benchmarking microservice systems for software engineering research," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 2018, pp. 323–324.