Scheduling Microservice Containers on Large Core Machines through Placement and Coalescing

Vishal Rao[1], Vishnu Singh[1], Goutham K S[1], Bharani Ujjaini Kempaiah[1], Ruben John Mampilli[1], Subramaniam Kalambur[1], and Dinkar Sitaram[2]

[1] Department of Computer Science and Engineering
PES University, Bangalore, India
{vishal.s.rao11, vishnuvijaysingh, ksgoutham1998@gmail.com}@gmail.com,
{ukbharani@gmail.com, rubenjohn1999@gmail.com}@gmail.com,
subramaniamkv@pes.edu
[2] Cloud Computing Innovation Council of India (CCICI), India
dinkar.sitaram@gmail.com

Abstract. Current applications deployed on the cloud use the microservice programming model to enable rapid deployment. Due to the loosely coupled nature of the interactions between microservices, they are well suited for the distributed nature of cloud systems. However, we are seeing a trend of increasing core counts on newer server hardware and scheduling microservices agnostic to the local organization of cores and memory on these systems leads to sub-optimal performance. In this paper, we propose a placement scheme to map containers of a microservice to various cores on such machines to maximize performance. We further study the impact of various parameters such as packet sizes and database sizes on the placement scheme and demonstrate that our placement scheme increases throughput by 22% while simultaneously lowering tail latency. Finally, we propose a mechanism to dynamically coalesce services on commonly called paths into a single container and demonstrate a further 7.5% improvement in throughput.

Keywords: Microservices · Container-Scheduling · Performance · NUMA

1 Introduction

A microservice has been defined as a cohesive, independent process interacting via messages [20] and a microservice-based architecture is a method of designing software where all the modules are microservices. This is in contrast to a monolithic application whose modules cannot be executed independently [20] and as a result, monolithic applications scale poorly. Microservice based deployments using containers are becoming more popular in today's world as they offer the following advantages over monolithic deployments-

– Each loosely coupled service scales independently.

– The codebase is more modular and debugging is easier.
– Decoupled services are easier to reconfigure to serve different purposes.

Due to the many advantages, the most popular internet-based services such as Netflix [39] and Amazon [34] have adopted the microservice-based architecture. These microservice-based applications are deployed on a large number of servers to take advantage of the resources provided by a cluster of hosts. This has led to research on how optimally place the containers on a cluster of systems to maximize application performance [26–30, 35, 36, 45, 52, 53]. These efforts consider different factors such as hardware resource requirements, communication affinities, and resource contentions between containers to find the best placement of the containers on the set of available hosts.

Recent development in server hardware to deliver performance scalability has seen the introduction of high core-count server CPUs are based on multi-chip module designs with multiple non-uniform memory access (NUMA) domains per socket. As a result, these CPUs behave like a mini-distributed system. On such systems, the performance impact of NUMA is more pronounced and a lack of knowledge of underlying topology leads to a loss in performance [14]. This performance impact due to NUMA is significant for large scale web-service applications on modern multicore servers [48]. Previous work on process scheduling lays emphasis on minimizing resource contentions. Resource contentions cause concurrently executing workloads interference leading to performance degradation as well as inefficient utilization of system resources [12, 19, 37, 38, 40, 41, 50]. In addition to minimizing resource contentions, minimizing communication overhead between tasks has been shown to benefit performance as well [13, 17, 23].

Similarly, in the case of container scheduling, minimizing resource contentions as well as the communication overhead between them is very important. In a microservice-based architecture, large amounts of data are exchanged between services, and when deployed using Docker containers, there is a non-negligible overhead due to network virtualization [11, 21, 51]. Hence co-locating heavily communicating containers helps reduce the communication latency and benefits performance. In the case of high core-count systems, communication between containers co-located on the same core complex (CCX), i.e a group of cores sharing an L3 cache (as illustrated in Fig. 1), have mean latencies that are 34% lower than that between containers located on adjacent CCXs. In Section 7, we show that co-located containers take advantage of the L3 cache to communicate and hence, communicate faster. In this research, we introduce the **'Topology and Resource Aware Container Placement and Deployment'** scheduler (TRACPAD). TRACPAD is a local container scheduler that strategically schedules multiple containers on a single system while reducing resource contentions and communication latencies between containers.

These latencies become very relevant in the case of end-to-end user-facing applications [44] such as social networks, search engines, and streaming services where overall application latency needs to be minimized to ensure smooth user experience. Hence, such applications have strict quality of service (QoS) constraints specified in the form of throughput and latency. As the number of users

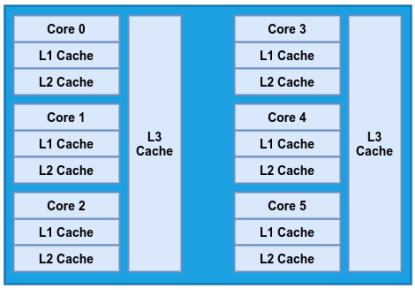


Fig. 1: AMD EPYC 7401 Architecture with 2 CCXs

increases, tail latency becomes an extremely important factor in characterizing performance [18]. Hence, in our work we use throughput, mean latency, 95^{th} percentile tail latency, and 99^{th} percentile tail latency to measure application performance.

Most of the research on container scheduling is focused on how to schedule containers on a set of hosts and does not consider the topology of each server as a factor while placing a group of containers on a server. As today's servers act like mini-distributed systems, it becomes important to schedule containers in accordance with each system's underlying architecture. Recent research demonstrates that strategically placing containers on a large core count system improves application performance [14], but to the best of our knowledge, there is no automated scheduler that does the same. Our contributions are summarized as follows -

- A workload independent local scheduler – TRACPAD, that uses historical runtime data to find a CCX-container mapping that improves application performance and a detailed evaluation of our scheduler across 4 representative microservice applications. TRACPAD boosts application throughput by up to 22% and decreases mean latency by up to 80%.
- An understanding of how TRACPAD improves application performance and the factors that impact the scheduling policies generated by TRACPAD.
- An algorithm to strategically coalesce containers in order to eliminate the communication overhead and the performance benefits of using such a method. Coalescing containers significantly reduces application latencies. We show how our method reduces mean latency by up to 40% and 99^{th} percentile tail latency by up to 60%.

The remainder of this paper is organized as follows. Section 2 goes over previous research and related work. Section 3 explains the implementation specifics of the TRACPAD scheduler and Section 4 outlines the experimental setup used for this study. Section 5 briefly introduces the microservice applications used in our study. Section 6 presents the evaluation of the scheduler and Section 7 explains why TRACPAD improves performance. Section 8 goes over two major factors that affect the scheduling policies generated by TRACPAD. Section 9 introduces the methodology to coalesce containers and goes over the performance impact of

using this method. Finally, Section 10 concludes the paper and discusses further work.

2 Related Work

Virtual machines (VMs) have paved the way for containers in today's server ecosystem. Research on VM scheduling algorithms gives useful insights regarding what factors to consider while scheduling containers. Novaković *et al.* [41] and Nathuji *et al.* [40] emphasise on minimizing performance interference while co-locating VMs. Starling [46] and AGGA [15] show how performance can be improved by placing co-communicating VMs on the same hosts as this reduces the cost of network communication. This trade-off between scheduling heavily communicating processes, and trying to minimize resource contentions is also extremely valid in the case of container scheduling. Hu *et al.* [27] introduce a multi-objective container scheduler that generates deployment schemes that distribute the workload across servers to reduce resource contentions with the primary goal of providing multi-resource guarantees. They model the problem as a vector bin packing problem with heterogeneous bins. This scheduler also uses dependency awareness between containers to co-locate containers that heavily communicate. Many such schedulers that take into account container resource utilization affinity and resource contentions have been implemented [26, 35, 45].

Containerized microservice environments are sensitive to many operating system parameters [25] and hence the problem of container scheduling has also been tackled in many other ways using a large set of heuristics and a larger set of algorithms. Zhang *et al.* [53] propose a container scheduler that tries to minimize host energy consumption as one of its objectives. They use a linear programming based algorithm to schedule containers. ECSched [28] highlights the advantages of concurrent container scheduling over queue-based container scheduling. They model the scheduling problem as a minimum cost flow problem and incorporate multi-resource constraints into this model. Fig. 2 illustrates a simple classification of the different types of schedulers surveyed as part of this research.

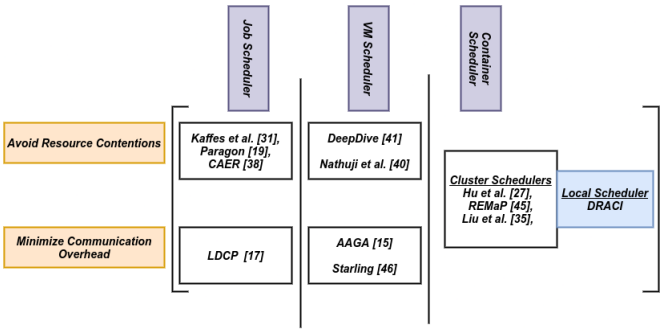


Fig. 2: Simple Classification of Schedulers

Kubernetes [4] and Docker Swarm [2] are the most commonly used container orchestration tools. These technologies do not factor in historical run-time data to analyze the characteristics of the application while scheduling containers on servers. Resource requirements of microservices vary with the workload they are servicing and historical runtime data is very useful while characterizing these dynamic workloads. Kubernetes placement algorithms schedule containers on nodes in a cluster based on the resource limits and tags preset by the developers. They do not consider the relationship between microservices while formulating a placement scheme. Kubernetes also recently added the topology manager [5] which tries to place containers on the same NUMA node based on different heuristics and user-specified criteria. However, this method does not consider previous runtime information to make an informed choice on which containers to co-locate on the same NUMA node whereas, the TRACPAD scheduler uses historical runtime data to make a more informed choice on which containers to co-locate.

REMaP [45] is another container scheduler that tries to solve some of the shortcomings of the common container management tools used today by using runtime adaptation. Each pair of microservices are assigned affinity scores based on the amount of data they exchange. REMaP tries to group microservices with high affinities on the same server to minimize the number of hosts while maximizing the affinity score of each host. The placement scheme also takes into account the amount of resources available on the server and approximates the amount of resources the microservice needs by using historical runtime data. It only co-locates a pair of microservices if the host has enough free resources to handle them. All these contributions present methods to schedule containers on a cluster of systems and do not consider the underlying topology of each server. This work presents a method that finds a placement scheme for containers that would benefit the application performance after taking into consideration the architecture of a server with a large core count. Previous works also shed light on how granular resource allocation can benefit performance. PARTIES [16] and CLITE [42] use fine-grained resource allocation to co-locate latency-critical jobs without violating their respective QoS targets. Sriraman [47] presents a method that uses A/B testing to tune configurable server parameters, such as core frequency and transparent hugepages, specifically for a microservice, so that it performs better on that server. Kaffes *et al.* [31] outline the benefits of a centralized core granular scheduler such as reducing interference. Octopus-Man [43] uses a core granular scheduling method to bind processes to cores in servers that have both brawny and wimpy cores in such a way that QoS constraints are not violated, throughput is improved and the energy consumption is reduced. The TRACPAD scheduler uses granular resource allocation to avoid resource contentions between containers while reducing the Docker network communication overhead.

3 Implementation of TRACPAD

This section explains how our method arrives at a partition and then allocates resources to containers dynamically based on the partition. This section is divided into 5 modules which describe how the whole scheduling process is automated.

3.1 The Container Resource Utilization model

Fig. 3 depicts a sample container resource utilization model for the 'Get Products' workload of the TeaStore application. The edge weights quantify the communication between two services while the node weights represent the CPU utilization of the service. In TRACPAD we use a bi-criteria approximation graph partitioning algorithm to [32] to partition the container resource utilization model based on two criteria -

- Minimization of the edgecut, i.e the sum of the edge weights between partitions is minimized.
- Even distribution of node weights, i.e the node weights are balanced across all partitions.

The edge weights in the model represent the amount of data transferred between the containers. The partitioning will, therefore, reduce network communication between the partitions. The node weights represent the CPU utilization of each container. Hence, each partition will have a balanced CPU utilization, thereby, reducing the CPU contentions in each partition. TRACPAD uses the 'Container Resource Utilization Model' to generate different partitioning schemes.

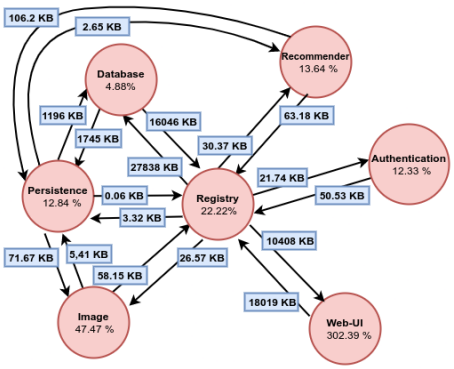


Fig. 3: Container Resource Utilization Model - Get Products Workload

3.2 Collection of Container Specific Data

Every container in an application requires adequate server resources for smooth functioning. This module identifies the CPU-intensive containers for a particular workload. Docker Stats is used to obtain container resource usage statistics for the duration of the workload. This is how TRACPAD uses historical runtime data to generate an informed placement scheme. TRACPAD uses this data to calculate average CPU utilization for every container. The CPU utilization metrics act as the node weights in the container resource utilization model.

3.3 Monitoring Network Traffic

This module establishes which sets of containers are communicating and the amount of data being transferred between them during any given workload. First, the ID of the Docker-bridge being used for communication is obtained. All network packets used to communicate between containers are sent on the Docker bridge. Tshark [10] is used to monitor the Docker bridge and records every packet that is transmitted. The packet headers contain the source container's IP address, destination container's IP address, and the payload size. TRACPAD uses the data in the packet headers to characterize container resource utilization patterns during the course of the workload.

A single request from the client may result in several packets being sent between containers. Different network protocols of the application layer and transport layer in the TCP/IP stack may be used to communicate between different containers and we need to capture all these packets. For example, in the social network application, Apache Thrift RPCs are used to communicate between the microservices only. To communicate with their respective databases, they use TCP, UDP, or Memcache protocols. The networkX library [7] is used to generate a container resource utilization model as illustrated in Fig. 3. TRACPAD aggregates all the data communicated between two containers in one direction irrespective of the protocol used for data transfer and assigns the sum of the payload sizes to the weight of that edge in the model.

3.4 Generation of the TRACPAD Partitioning Scheme

Algorithm 1 details the TRACPAD generates container placement schemes that map containers to CCXs. Note that this mapping is workload-specific. Thus, different workloads will have different placement schemes. Once the container resource utilization model has been partitioned, a Container-CCX mapping is generated. This mapping is invariant, i.e it remains the same for the corresponding configuration of the application on that system and respective workload as long as the communication patterns for the workload do not change. Hence, it can be used to partition the application on the system multiple times in case the application needs to restart or if the server goes down. The next step is to affinitize the containers to the CPUs based on the mapping.

Algorithm 1: TRACPAD Partition Generator

Result: List of Tuples that map CCXs to Containers.
Require: Set of all Containers, CCXList
/* *List of CCXs allocated to application* */
Initialize CRM ← [] // *Container Resource Utilization Model*
Initialize PartitionList ← []
Initialize ContainerCCXMapping ← []
Initialize NoOfPartitions ← Length(CCXList)

NodeWeights$_i$ ← CPU-Util(C_i) ∀ C_i ∈ ContainerSet
Edges $_{i,j}$ ← BytesSent(C_i, C_j) ∀ C_i, C_j ∈ ContainerSet
CRM ← [NodeWeights, Edges]

PartitionList ← generatePartitions(CRM, NoOfPartitions)
/* *PartitionList is a list of sets, All containers in a set belong to the same partition as shown in Fig. 9* */

while !empty(PartitionList) **do**
  ContainerSet ← pop(PartitionList)
  CCX ← pop(CCXList)
  for all Container ∈ ContainerSet **do**
    push(ContainerCCXMapping, (Container, CCX))
  end for
end while

3.5 Dynamic Provisioning of Resources

Once the Container-CCX mapping has been generated, TRACPAD redistributes server resources, in this case, CPUs, across all the containers according to the partitioning scheme. This is done on the fly, i.e, there is no need to stop and restart the containers. Since this redistribution of resources happens on the fly, the transition in performance is seamless. The cost of redistribution is system dependent. In our experiments, it took approximately 1.7 seconds to redistribute resources among the 30 containers spawned by the social-network application.

4 Experimental Setup

All our studies were conducted on a dual-socket AMD EPYC 7401 24-Core processor server with 128GB of NUMA RAM and 2 TB Storage Capacity. The server consists of 16 core-complexes with each core-complex consisting of 3 cores (with 2-way SMT providing 6 logical CPUs) sharing an 8 MB L3 cache as illustrated in Fig. 1. Each physical core has a private 64 KB L1 instruction cache, 32KB L1 data cache, and a 512 KB L2 cache.

The microservice-based applications were deployed on socket-0 and the HTTP workload generator on socket-1. We deployed both the client and the application on the same system to avoid any performance impact of network latency across

multiple runs. The applications were allocated 24 Logical Cores across 4 CCXs and the TRACPAD algorithm was configured to generate four partitions, one per CCX. In the case of the Social Network Application and Media Service Application, the Nginx container was given access to all 24 logical cores in case of all scheduling policies so that the number of requests serviced by the application was not limited by the number of cores given to the Nginx server.

5 Microservice Benchmarks and Workloads

This section briefly introduces a variety of representative microservice benchmarks and workloads that have been used to evaluate the TRACPAD scheduler.

The DeathStar Bench Suite [22] provides two applications – a social network and a media service application. Workloads provided by both these applications are used to conduct our evaluations. These applications are heterogeneous, using C/C++, Java, Go, Scala, JavaScript, and many others. This makes the applications more representative as different functionalities are developed using different languages. The third application used is TeaStore [33] which is an end-to-end e-commerce application that is used to sell different types of tea and tea products. The last application used in this study is Sock Shop [9]. This is another end-to-end e-commerce application that sells socks to customers. All these applications have been used in previous microservices-based performance studies as suitable proxies for real-world applications [14,22,45,51]. Section 5.1 explains the social network application in detail to provide an example of the complexity and functioning of all these applications.

Table 1 summarizes the applications and lists the workloads that are used in this study.

Table 1: Summary of the Applications and Workloads

Application	**Workload**	**No. of Containers**	**Storage Backends**
Social Network *(DeathStarBench)* [22]	Compose Post Read Home Timeline Read User Timeline	30	MongoDB, Redis, Memcached
Media Service *(DeathStarBench)* [22]	Compose Review	33	MongoDB, Redis, Memcached
TeaStore [33]	Get Products Add to Cart	7	MariaDB
Sock Shop [9]	Get Catalogue	15	MongoDB MySQL

5.1 The Social Network Application

It is an end-to-end application that implements a broadcast-style social network with unidirectional follow relationships. This application uses 14 microservices spawned across 30 containers. The microservices communicate with each other using Apache Thrift RPCs. This application uses Memcached and Redis for caching data and MongoDB for persistent storage of posts, profiles, and media. Each storage backend is implemented using an individual container. The Jaeger tracing application uses a Prometheus backend for storage and is a monitoring service implemented to trace packets. The architecture of the application is shown in Fig. 4.

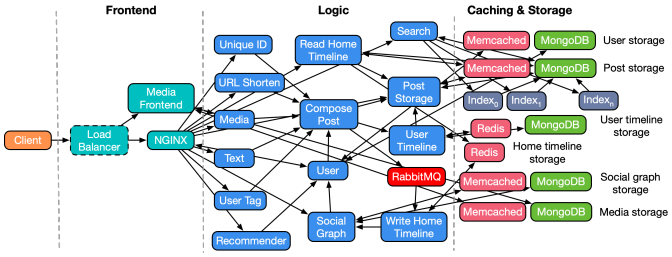


Fig. 4: Social Network Application Architecture

Source: https://github.com/delimitrou/DeathStarBench

To test the impact of database load on the scheduling policy, we used two social network databases [3] to conduct our tests on the social network application.

- Reed 98 Social Network - 962 Users and 18812 Edges
- Penn Social Network - 43,000 Users and 1.3M Edges

6 Evaluation of the TRACPAD Scheduler

To evaluate the TRACPAD scheduler, 6 workloads from four end-to-end microservice applications were used. Application performance was evaluated using four metrics -

- Application throughput
- Mean latency
- 95^{th} percentile tail latency
- 99^{th} percentile tail latency

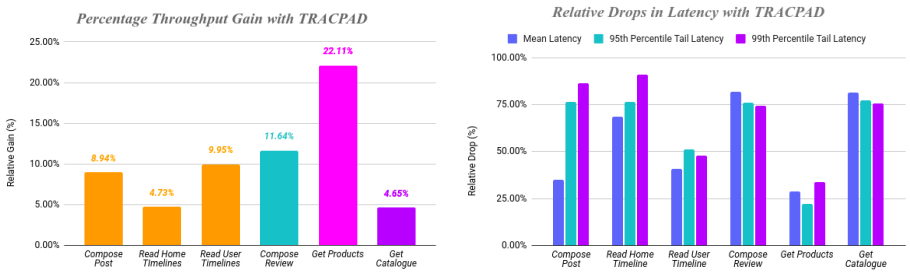
These are the four important metrics that are commonly used to measure the performance of user-facing applications. The wrk HTTP workload generator

[24] was used to measure application throughput and application latency was measured using the wrk2 HTTP workload generator [49].

First, all the partitions were generated after running the tool with each workload for 3-5 minutes. The tool collected all the required metrics and generated the partition. As the underlying graph partitioning algorithm is extremely parallelized, TRACPAD can generate partitions within a few milliseconds once the workload-specific runtime data points are collected.

As a baseline, we first measured these metrics for all the applications without any affinitization, i.e all application containers could migrate between any of the 24 logical cores allocated to them. Fig. 5a illustrates the relative gain in throughput observed for each workload.

The TRACPAD scheduler boosts the throughput by an average of 10.34% across all 6 workloads. Fig. 5b illustrates the percentage reduction in latencies when the TRACPAD scheduler is used as compared to the naive policy of allocating all cores to all containers that Docker uses by default. Across all the workloads, on average, there is a 56.11% drop in mean latency, a 63.11% drop in 95^{th} percentile tail latency, and a 68.15% drop in 99^{th} percentile tail latency. As an example, Fig. 6 illustrates a comparison of the cumulative latency distributions between the naive policy and the policy generated by TRACPAD for the 'Get Products' workload of the TeaStore application. There is a drop in latencies throughout the latency percentile distribution demonstrating the ability of the TRACPAD scheduler to improve latencies by the strategic allocation of resources.



(a) Comparison of Baseline Throughput with TRACPAD Throughput

(b) Comparison of Baseline Latencies with TRACPAD Latencies

Fig. 5: Performance Comparison between TRACPAD and Baseline

These improvements in performance can be explained by the observations in Section 7. They are a result of lower communication latency between co-located containers, reduced resource contentions due to the graph partitioning, and reduced CPU migrations and context switches. There may be cases wherein a container needs more logical cores than provided by 1 CCX, like the Nginx

Fig. 6: Comparison of Cumulative Latency Distribution of the Get Products Workload

container in our case, and in such cases, it is necessary to allocate more cores to the container. The TRACPAD scheduler has been developed with the aim of scaling up performance and can be implemented in tandem with schedulers that improve performance by using scale-out methodologies.

7 Analysis of TRACPAD

This section helps explain the factors that are responsible for the improvement in application performance. TRACPAD affinitizes containers to a set of cores belonging to the same CCX. We observed that this led to a reduction in both CPU thread migrations and context switches. CPU thread migrations can be very expensive on multi-socket systems with multiple NUMA domains in cases where threads migrate between NUMA domains as this either results in remote memory accesses, which have higher latencies than local memory accesses, or memory migration to the new node. Linux perf [8] was used to measure both metrics for the workloads listed in Table 1. Fig. 7 illustrates the relative drop in these two metrics when the TRACPAD scheduler was used as compared to the naive case where containers have access to all cores allocated to the application.

TRACPAD also co-locates communicating containers on the same CCX. This helps reduces the communication latency between the containers and hence, improves performance. To illustrate this we used Netperf [6], which measures the network latency by sending packets between a client and a server. The server and client were containerized and the network latency was measured in 3 cases -

– Case 1: Server and client are placed on the same CCX.

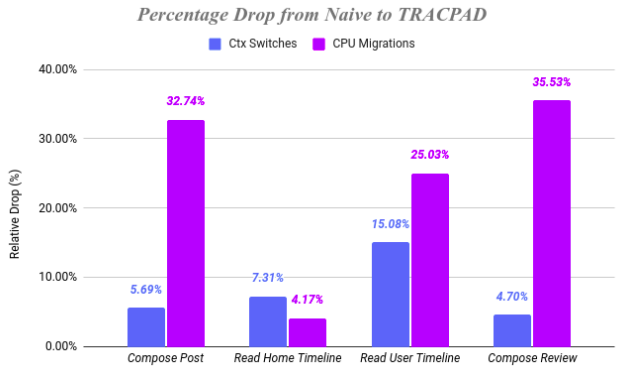


Fig. 7: Drop in Context Switches and CPU Migrations by using TRACPAD

– Case 2: Server and client are placed on adjacent CCXs on the same NUMA node.
– Case 3: Server and client are placed on adjacent NUMA nodes on the same socket.

Fig. 8 compares the mean latencies, 90^{th} percentile and 99^{th} percentile tail latencies. It illustrates that there is a significant increase in all these metrics when the server and client are not placed on the same CCX. This happens because they can no longer use the L3 cache to communicate. To validate this, AMD μProf [1] was used to measure L3 miss rates for the CCX that the server and client were placed on for two packet sizes - 5KB and 2MB.

– A packet size of 5KB is much smaller than the 8MB capacity of the L3 cache and in this case, the L3 miss rates, as well as the network latencies, were extremely low.
– A packet size of 2MB will exhaust the capacity of the L3 cache quickly and in this case, the L3 miss rate increased by 4 times as compared to the 5KB case and the network latencies shot up as well.

To explain the impact of cross CCX communication highlighted in Fig. 8, L3 miss rates were measured when the client and server were placed on the same CCX and when they were placed on different CCXs. AMD μProf showed that the L3 miss rate increased by 10 times when the server and client were placed on different CCXs. This points to the explanation that co-locating communicating containers on the same CCX reduces the communication latency between them as they use the shared L3 cache to communicate. TRACPAD takes this into account and creates placement schemes that co-locate heavily communicating containers on the same CCX to reduce the impact of the communication overhead induced by Docker. The placement schemes can be sensitive to factors like request packet sizes, database sizes, and data access patterns. Section 8 explores

the impact of the request packet sizes and database sizes on the partitioning schemes.

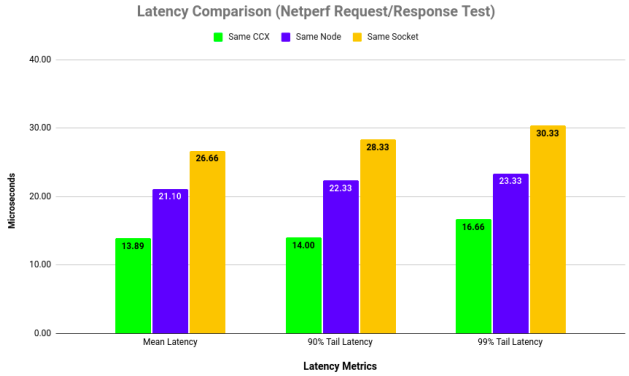


Fig. 8: Latency Comparison using Netperf

8 Factors Affecting the Scheduling Policy

This section investigates the impact of request packet size and database size on the scheduling policies generated by TRACPAD. Fig. 9a and Fig. 9b depict the TRACPAD scheduling policies for different Request Packet Sizes for the Compose Post Workload on the same underlying database, i.e, The Reed98 Social Network. Fig. 9c, which depicts the TRACPAD policy for a packet size of 256 bytes with the Penn Social Network as the underlying DB. The different TRACPAD policies improved application performance in each case after factoring in the changes in request packet size and database size.

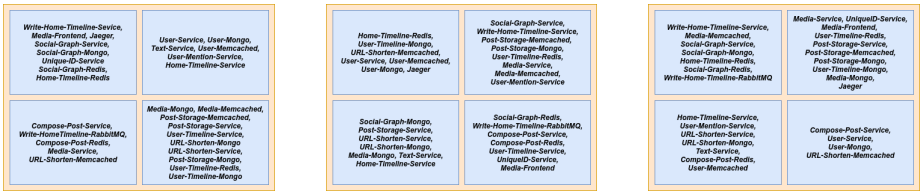
8.1 Effect of Database Size on Scheduling Policy

TRACPAD scheduling policies can change with the size of the underlying database as evidenced by Fig. 9c and 9b. This may be because, as the database sizes increase, database operations like querying and sorting start utilizing more CPU. Therefore, the containers housing these databases also utilize resources differently. This alters the node weights in the container resource utilization model and hence, affects the scheduling policy.

8.2 Effect of Request Packet Size on Scheduling Policy

TRACPAD scheduling policies can change with the request packet size as evidenced by Fig. 9a and 9b. One of the reasons for this change is that a change

in the request packet size can lead to a change in the amount of data processing done by the containers. If a container is performing compute-intensive operations (Ex. String Search) on the payload, then as the payload size increases, the container's CPU utilization will increase. This alters the node weights in the container resource utilization model and affects the scheduling policy. A change in the request packet size can also alter the edge weights of the container resource utilization model as this change can change the amount of data exchanged between application containers. This will also modify the scheduling policy. In the next section, we introduce a novel method to merge communicating containers so that the data exchanged between them is not routed through the Docker bridge and discuss how this method can be used along with TRACPAD to further improve performance.



(a) Request Packet Size 2 kB, Reed98 Database

(b) Request Packet Size 256 B, Reed98 Database

(c) Request Packet Size 256 B, Penn Database

Fig. 9: TRACPAD policies for different configurations of the Compose Post Workload

9 Container Coalescing

In microservice environments, as a result of loose coupling between different modules, huge amounts of data are exchanged between different microservices. The communication overhead induced by Docker can negatively impact performance while huge amounts of data are being exchanged over the network. TRACPAD scheduler improves performance because it tries to minimize the communication overhead. Container coalescing further reduces the impact of communication overhead by eliminating all communication overhead by merging frequently communicating containers. This section outlines the factors to consider before merging containers and shows that strategic coalescing of containers can improve application performance.

9.1 Design Considerations

Containers that functioned as storage backends were not coalesced to ensure business continuity and prevent loss of data on failure. A greedy method was

Algorithm 2: Coalescer

```
Result: One New Coalesced Image.
Require: ContainerCommunicationTuples
  //  Tuple - (SrcContainer, DstContainer, PayloadSize)
  Initialize CCTuples ← ContainerCommunicationTuples();

  sort(CCTuples) // Descending order of payload size
  IsCoalesced ← False

  for all  T ∈ CCTuples do
    SrcBase ← getBaseImage(T.SrcContainer)
    DstBase ← getBaseImage(T.DstContainer)
    if  !IsCoalesced ∧ Coalescable(SrcBase, DstBase) then
      CoalescedImage ← CreateNewImage(SrcBase, DstBase)
      CreateNewEntryPoint(CoalescedImage)
      IsCoalesced ← True
    end if
  end for
  /* Edit the Docker-compose files and any other
  application files to support the new image */
```

used to coalesce the containers, i.e the two containers that communicated the most during the course of a workload were coalesced if they were compatible. Determining whether a pair of containers were compatible, comprised of checking whether there were any dependency conflicts and if the two containers were using the same set of internal ports to expose their functionality. In either case, the pair of containers cannot be coalesced.

9.2 Methodology

Algorithm 2 outlines the method we have used to coalesce a pair of containers. The 'Coalescable' function in the algorithm checks if the two containers can be coalesced and whether they violate the design considerations outlined in the previous section. While creating a new base image, a new Docker entry point is required. The new entry point should execute the entry point commands of the images being combined in the appropriate order.

9.3 Experimental Setup

To evaluate the impact of the container coalescing methodology, we used two single-socket AMD EPYC 7301 16-Core processors with 64GB of NUMA RAM on each server. Each server consists of 8 CCXs with each CCX consisting of 2 cores (with 2-way SMT providing 4 logical CPUs) sharing an 8MB L3 cache. Each physical core has a private 64 KB L1 instruction cache, 32KB L1 data cache, and a 512 KB L2 cache. The client and the application were deployed on different servers.

9.4 Results

The impact of coalescing was evaluated using three simple workloads from Table 1. The most heavily communicating containers were coalesced in the case of each of these workloads as listed in Table 2.

Table 2: Workloads and Coalesced Containers

Workload	Containers Coalesced
Read Home Timeline	Home Timeline Service Post Storage Service
Read User Timeline	User Timeline Service Post Storage Service
Add to Cart	Web UI Service Persistence Service

On coalescing, the amount of data transmitted over the network reduced by over 10%. Fig. 10 illustrates the improvement in performance for the 'Get Products' workload. After the containers were coalesced, the TRACPAD scheduler was applied to further improve performance. Fig. 11 illustrates the combined performance gain of coalescing and scheduling for the workloads. TRACPAD schedules the new set of containers, both original and coalesced, to reduce resource contentions and further reduce the impact of the network communication overhead.

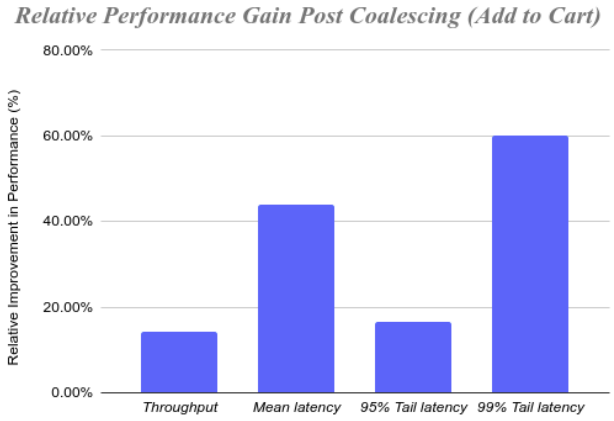


Fig. 10: Relative gain in Performance after Coalescing.

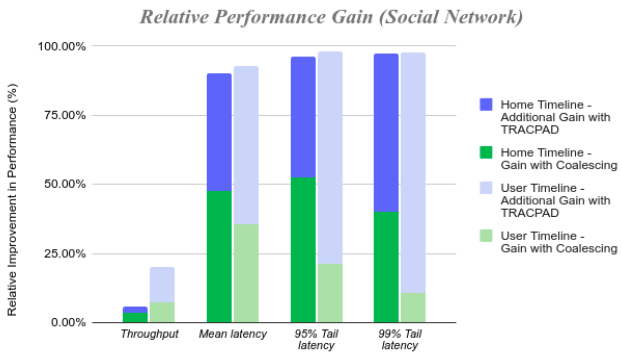


Fig. 11: Combined Performance Gain of Coalescing and TRACPAD

10 Conclusion and Further Work

This paper presents a local container scheduler that creates placement schemes based on the underlying server architecture. The scheduler is evaluated using 4 representative microservice applications and using the scheduler improves the application performance. A detailed analysis of improvement in application performance is also studied. The scheduler tries to minimize resource contentions as well as minimize the communication overhead induced by Docker. To eliminate this overhead between a pair of communicating containers, a container coalescing methodology is outlined in the latter sections of this paper. This method also improves performance and combined with the TRACPAD scheduler, can offer significant reductions in user-perceived latencies.

As part of future work, we plan to implement NUMA node based memory scheduling by factoring in the memory utilization of each container. This will help reduce the memory contentions between the containers scheduled on the same NUMA node and as a result, improve application performance. We also plan to investigate the security implications of coalescing containers and develop a coalescing strategy that does to impact the security provided by containers adversely. Our final goal is to automate the process of container coalescing and integrate it with the TRACPAD scheduler so that their combined performance gains can benefit application performance.

Acknowledgement

We would like to thank AMD India Private Limited for their funding of this project.

References

1. Amd μprof, "https://developer.amd.com/amd-uprof"
2. Docker swarm overview, "https://docs.docker.com/engine/swarm/"
3. Facebook social networks, "http://networkrepository.com/socfb"
4. Kubernetes: Production-grade container orchestration, "https://kubernetes.io"
5. Kubernetes topology manager, "https://kubernetes.io/docs/tasks/administer-cluster/topology-manager/"
6. Netperf: A network performance benchmark, "https://linux.die.net/man/1/netperf"
7. Networkx: Network analysis in python, "https://networkx.org"
8. perf: Linux profiling with performance counters, "https://perf.wiki.kernel.org/index.php/Main$_Page$"
9. Sock shop microservice application, " https://microservices-demo.github.io"
10. Tshark: Terminal based wireshark, "https://www.wireshark.org/docs/man-pages/tshark.html"
11. Alles, G.R., Carissimi, A., Schnorr, L.M.: Assessing the computation and communication overhead of linux containers for hpc applications. In: 2018 Symposium on High Performance Computing Systems (WSCAD). pp. 116–123. IEEE (2018)
12. Athlur, S., Sondhi, N., Batra, S., Kalambur, S., Sitaram, D.: Cache characterization of workloads in a microservice environment. In: 2019 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM). pp. 45–50. IEEE (2019)
13. Buzato, F.H., Goldman, A., Batista, D.: Efficient resources utilization by different microservices deployment models. In: 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA). pp. 1–4. IEEE (2018)
14. Caculo, S., Lahiri, K., Kalambur, S.: Characterizing the scale-up performance of microservices using teastore. In: 2020 IEEE International Symposium on Workload Characterization (IISWC). pp. 48–59. IEEE (2020)
15. Chen, J., Chiew, K., Ye, D., Zhu, L., Chen, W.: Aaga: Affinity-aware grouping for allocation of virtual machines. In: 2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA). pp. 235–242. IEEE (2013)
16. Chen, S., Delimitrou, C., Martínez, J.F.: Parties: Qos-aware resource partitioning for multiple interactive services. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 107–120 (2019)
17. Daoud, M.I., Kharma, N.: A high performance algorithm for static task scheduling in heterogeneous distributed computing systems. Journal of Parallel and distributed computing **68**(4), 399–409 (2008)
18. Dean, J., Barroso, L.A.: The tail at scale. Communications of the ACM **56**(2), 74–80 (2013)
19. Delimitrou, C., Kozyrakis, C.: Paragon: Qos-aware scheduling for heterogeneous datacenters. In: ACM SIGPLAN Notices. vol. 48, pp. 77–88. ACM (2013)
20. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. In: Present and ulterior software engineering, pp. 195–216. Springer (2017)
21. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and linux containers. In: 2015 IEEE international symposium on performance analysis of systems and software (ISPASS). pp. 171–172. IEEE (2015)
22. Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., 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. pp. 3–18. ACM (2019)
23. Georgiou, Y., Jeannot, E., Mercier, G., Villiermet, A.: Topology-aware job mapping. The International Journal of High Performance Computing Applications **32**(1), 14–27 (2018)
24. Glozer, W.: wrk - a http benchmarking tool, "https://github.com/wg/wrk"
25. Goutham, K.S., Ujjaini Kempaiah, B., John Mampilli, R., Kalambur, S.: Performance sesitivity of operating system parameters in microservice environments (in press)
26. Guo, Y., Yao, W.: A container scheduling strategy based on neighborhood division in micro service. In: NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium. pp. 1–6. IEEE (2018)
27. Hu, Y., De Laat, C., Zhao, Z.: Multi-objective container deployment on heterogeneous clusters. In: 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). pp. 592–599. IEEE (2019)
28. Hu, Y., Zhou, H., de Laat, C., Zhao, Z.: Ecsched: Efficient container scheduling on heterogeneous clusters. In: European Conference on Parallel Processing. pp. 365–377. Springer (2018)
29. Hu, Y., Zhou, H., de Laat, C., Zhao, Z.: Concurrent container scheduling on heterogeneous clusters with multi-resource constraints. Future Generation Computer Systems **102**, 562–573 (2020)
30. Kaewkasi, C., Chuenmuneewong, K.: Improvement of container scheduling for docker using ant colony optimization. In: 2017 9th international conference on knowledge and smart technology (KST). pp. 254–259. IEEE (2017)
31. Kaffes, K., Yadwadkar, N.J., Kozyrakis, C.: Centralized core-granular scheduling for serverless functions. In: Proceedings of the ACM Symposium on Cloud Computing. pp. 158–164 (2019)
32. Karypis, G., Schloegel, K., Kumar, V.: Parmetis. Parallel graph partitioning and sparse matrix ordering library. Version **2** (2003)
33. von Kistowski, J., Eismann, S., Schmitt, N., Bauer, A., Grohmann, J., Kounev, S.: Teastore: A micro-service reference application for benchmarking, modeling and resource management research. In: 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). pp. 223–236. IEEE (2018)
34. Kramer, S.: Gigaom—the biggest thing amazon got right: The platform (2011)
35. Liu, B., Li, P., Lin, W., Shu, N., Li, Y., Chang, V.: A new container scheduling algorithm based on multi-objective optimization. Soft Computing **22**(23), 7741–7752 (2018)
36. Mao, Y., Oak, J., Pompili, A., Beer, D., Han, T., Hu, P.: Draps: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster. In: 2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC). pp. 1–8. IEEE (2017)
37. Mars, J., Tang, L., Hundt, R., Skadron, K., Soffa, M.L.: 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. pp. 248–259 (2011)
38. Mars, J., Vachharajani, N., Hundt, R., Soffa, M.L.: Contention aware execution: online contention detection and response. In: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization. pp. 257–265 (2010)
39. Mauro, T.: Adopting microservices at netflix: Lessons for architectural design. Recuperado de https://www/. nginx. com/blog/microservices-at-netflix-architectural-best-practices (2015)

40. Nathuji, R., Kansal, A., Ghaffarkhah, A.: Q-clouds: managing performance interference effects for qos-aware clouds. In: Proceedings of the 5th European conference on Computer systems. pp. 237–250 (2010)
41. Novaković, D., Vasić, N., Novaković, S., Kostić, D., Bianchini, R.: Deepdive: Transparently identifying and managing performance interference in virtualized environments. In: Presented as part of the 2013 USENIX Annual Technical Conference ATC 2013). pp. 219–230 (2013)
42. Patel, T., Tiwari, D.: Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In: 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 193–206. IEEE (2020)
43. Petrucci, V., Laurenzano, M.A., Doherty, J., Zhang, Y., Mosse, D., Mars, J., Tang, L.: Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA). pp. 246–258. IEEE (2015)
44. Rahman, J., Lama, P.: Predicting the end-to-end tail latency of containerized microservices in the cloud. In: 2019 IEEE International Conference on Cloud Engineering (IC2E). pp. 200–210. IEEE (2019)
45. Sampaio, A.R., Rubin, J., Beschastnikh, I., Rosa, N.S.: Improving microservice-based applications with runtime placement adaptation. Journal of Internet Services and Applications **10**(1), 1–30 (2019)
46. Sonnek, J., Greensky, J., Reutiman, R., Chandra, A.: Starling: Minimizing communication overhead in virtualized computing platforms using decentralized affinity-aware migration. In: 2010 39th International Conference on Parallel Processing. pp. 228–237. IEEE (2010)
47. Sriraman, A., Wenisch, T.F.: μtune: Auto-tuned threading for {OLDI} microservices. In: 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18). pp. 177–194 (2018)
48. Tang, L., Mars, J., Zhang, X., Hagmann, R., Hundt, R., Tune, E.: Optimizing google's warehouse scale computers: The numa experience. In: 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA). pp. 188–197. IEEE (2013)
49. Tene, G.: wrk2 - a constant throughput, correct latency recording variant of wrk, "https://github.com/giltene/wrk2"
50. Thiyyakat, M., Kalambur, S., Sitaram, D.: Improving resource isolation of critical tasks in a workload. In: Workshop on Job Scheduling Strategies for Parallel Processing. pp. 45–67. Springer (2020)
51. Ueda, T., Nakaike, T., Ohara, M.: Workload characterization for microservices. In: 2016 IEEE international symposium on workload characterization (IISWC). pp. 1–10. IEEE (2016)
52. Xu, X., Yu, H., Pei, X.: A novel resource scheduling approach in container based clouds. In: 2014 IEEE 17th International Conference on Computational Science and Engineering. pp. 257–264. IEEE (2014)
53. Zhang, D., Yan, B.H., Feng, Z., Zhang, C., Wang, Y.X.: Container oriented job scheduling using linear programming model. In: 2017 3rd International Conference on Information Management (ICIM). pp. 174–180. IEEE (2017)