More Sharing, More Benefits? A Study of Library Sharing in Container-Based Infrastructures

José Bravo Ferreira[1(✉)], Marco Cello[2], and Jesús Omana Iglesias[2]

[1] Princeton University, Princeton, USA
josesf@princeton.edu
[2] Nokia Bell Labs, Dublin, Ireland
{marco.cello,jesus.omana_iglesias}@nokia-bell-labs.com

Abstract. Container-based infrastructures have surged in popularity, offering advantages in agility and scaling, while also presenting new challenges in resource utilization due to unnecessary library duplication. In this paper, we consider sharing libraries across containers, and study the impact of such a strategy on overall resource requirements, scheduling, and utilization. Our analysis and simulations suggest significant benefits arising from library sharing. Furthermore, a small fraction of libraries shared between any two containers, on average, is enough to reap most of the benefits, and even naïve schedulers, such as a First Fit scheduler, succeed at doing so. We also propose a score maximization, mixed-integer linear-programming scheduler for handling bulk request arrivals (such as large jobs composed of many smaller tasks), which compares favorably against state-of-the-art schedulers in these scenarios.

1 Introduction

Container-based infrastructures are gaining popularity both in Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) models. In IaaS (e.g. Pantheon and Amazon ECS) abstracting away the host operating system allows providers to make system-wide changes quickly, to provision new containers with little delay, and to scale up without worrying about architecture. This is also the case for PaaS (e.g. Heroku and Google's App Engine), where providers must automatically manage and maintain necessary infrastructure behind the scenes.

Containers can be added and removed in seconds, allowing for greater flexibility in dynamically scaling applications and in running mostly idle services. Modern container hypervisors, such as LXD[1], or container managers, such as Docker[2] or Kubernetes[3], facilitate the management of containers, with automatic scheduling, scaling and storage orchestration. This is accomplished while running seamlessly on most infrastructures using open standard containers.

[1] https://linuxcontainers.org/lxd/.
[2] https://www.docker.com/.
[3] https://kubernetes.io.

F.F. Rivera et al. (Eds.): Euro-Par 2017, LNCS 10417, pp. 358–371, 2017.
DOI: 10.1007/978-3-319-64203-1_26

The adoption of container-based infrastructures is bolstered by the increasing popularity of a microservices approach to software design, focused on scalability, agility, resilience, and developer efficiency [3]. However, breaking down an application into many smaller processes that need large-scale replication might also require loading the same set of libraries inside each container, resulting in memory duplication.

Existing schedulers, such as Tetris [5] or dominant resource fairness [4] can successfully schedule and pack tasks in an efficient way, but a large increase in the number of containers demands answers to new challenges related to volume, locality (runtime environments), and dependencies between containers [9].

As an extreme example of microservices design, recent efforts in *Serverless* solutions, such as AWS Lambda[4], incentivize the application developer to implement their services as a composition of stateless functions, often triggered by predefined events (e.g. a user request or a database change) and written in predefined programming languages. This leads the cloud provider to instantiate many containers loaded with similar runtime environments and the same language-specific libraries, so that library-sharing could be not just beneficial but an actual necessity. The analysis presented in [8, Figs. 3 and 13] shows that current Docker images contained in Docker Hub[5] already share some amount of libraries since they share common AUFS layers.

In this paper, we analyze the possible benefits of library-sharing across containers as a potential solution in current container-based infrastructures. Conceptually, we consider a scenario in which libraries are shared using a union file system across Linux containers (LXC), and study the impact on resource utilization through simulations and mathematical analysis. We:

1. Study library-sharing in single- and multi-resource scenarios analytically and through simulation, illustrating that even small levels of sharing yield significant savings in memory-bound scenarios (Sects. 4.1 and 4.2);
2. Propose a scheduling algorithm based on mixed-integer linear-programming (MILP) for handling bulk arrivals, which compares favorably against state-of-the-art schedulers and can be adapted to custom goals via a scoring function Sect. 4.2 providing improvements over naïve schedulers.

Section 2 lays down the motivation for this work in greater detail, Sect. 3 describes the methodology used in the paper, Sect. 4 presents the results of the analysis, and Sect. 5 summarizes the main conclusions.

2 Motivation

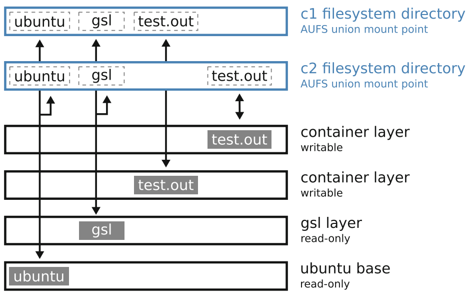
Current standard practice in container-based applications is to package all needed libraries in the container [7]. This approach has several advantages, such

[4] https://aws.amazon.com/lambda/.
[5] https://hub.docker.com/u/library/.

as ensuring that the application uses the intended version of the libraries. However, this solution suffers from redundantly loading libraries in memory that could otherwise be shared across containers.

Sharing libraries is a form of *memory deduplication* (MD), defined as a set of techniques to reduce the memory footprint of a running system by merging memory pages[6] with the same contents. A way to implement MD is through the use of union filesystem, such as AUFS[7]. AUFS takes a list of directories on a single Linux host and provides a single unified view. These directories are often referred to as *layers* and the technology used to layer them is known as a union mount. In AUFS, all the layers but the top one are read-only, and the unified view is exposed through its own directory called merged

In order to showcase the benefits of AUFS, we present results when 3 containers are running in the same host. Let us denote the three containers by `c1`, `c2` and `c3`. `c1` and `c2`'s filesystem directories are AUFS union mount points, composed of 3 layers: *ubuntu base* (Ubuntu base OS), *gsl layer* (GNU scientific libraries), and *container layer* (writeable layer containing the binary test file). In contrast, `c3`'s filesystem directory is a regular ext4 filesystem containing Ubuntu base OS, GNU scientific libraries and the binary test file. Figure 1 shows how the filesystem directories are created for the three containers. When the LXC containers are created, `c1` and `c2` share both Ubuntu base OS files and GSL scientific libraries on disk, while `c3` does not share any file with the other two.



(a) c1 and c2 filesystems as AUFS union mount point

ubuntu + gsl + test.out
c3 filesystem directory
AUFS union mount point

(b) c3 filesystem as regular filesystem

Fig. 1. Creation of testbed container filesystems.

[6] A fixed-length contiguous block of virtual memory, described by a single entry in the page table. It is the smallest unit of data for memory management in a virtual memory operating system.

[7] http://aufs.sourceforge.net/aufs.html.

A C executable, `test.out`, using GSL, is ran in all three containers. We considered different scenarios in which containers and the binary are running:

S1 Only `c1+test.out` is running on the host.
S2 Both `c1+test.out` and `c2+test.out` are running on the host.
S3 Both `c1+test.out` and `c3+test.out` are running on the host.

To show the amount of memory shared by GSL libraries (used internally by `test.out`) across containers we make use of the linux command `pmap`[8] which displays the process map of any process and its actual memory consumption.

Table 1 shows part of the output of the `pmap` command for scenarios **S1**, **S2**, and **S3**. The `Shrd` column shows the amount of memory which is shared with other processes and has not been modified, while the `Priv` column shows the amount of memory which is private to this process. Note that `libgsl` is not shared in **S1**. In **S2**, all libraries are now shared by the two containers, such that the total memory utilized is $\approx 2780 + 2 \cdot 156 = 3092$ kB. Finally, in scenario **S3**, we see the same numbers as in scenario **S1**, indicating no additional libraries are being shared. In this instance, $\approx 1288 + 2 \cdot 1472 = 4232$ kB are in use.

Table 1. Memory usage by c1 in scenarios S1, S2, and S3 [in kB].

S1		S2		S3		
`Shrd`	`Priv`	`Shrd`	`Priv`	`Shrd`	`Priv`	
0	4	0	4	0	4	`test.out`
0	56	64	0	0	56	`libgcc`
1156	0	1244	0	1156	0	`libc`
0	128	128	32	0	128	`libm`
0	752	748	28	0	752	`libstdc++`
0	44	44	0	0	44	`libgslcblas`
0	**472**	**420**	**92**	**0**	**472**	`libgsl`
128	0	128	0	128	0	`ld`
1288	**1472**	**2780**	**156**	**1288**	**1472**	**Total**

With this analysis, we verified that when two containers are sharing some of the libraries (in this case GSL libraries) on disk (using AUFS), they can actually share libraries in memory (**S2**). Moreover, from an operational point of view, AUFS is no more complex than other filesystems.

In this work, we analyze and quantify the improvements in resource utilization from sharing libraries. In particular, we highlight that even if two containers share, on average, only a small fraction of their libraries, the resulting memory deduplication is still substantial. Therefore, when memory is the limiting

[8] https://linux.die.net/man/1/pmap.

resource, as in today's data intensive applications, library sharing can result in improved server utilization. Meanwhile, resource isolation of containers is not affected, as it is handled by the linux control groups (cgroups) regardless of the union filesystem used (e.g. AUFS or OverlayFS).

3 Analysis Description

We begin by describing an abstracted model of a cloud environment for use in mathematical analysis and simulations (Subsect. 3.1). This is followed by a description of our methodology (Subsect. 3.2), and, finally, by a brief presentation of the simulator developed for this analysis (Subsect. 3.3).

3.1 Abstract Representation of a Cloud Environment

This work considers an arbitrary cloud environment, which could be a Platform-as-a-Service (PaaS) or a Serverless (Computation-as-a-Service, CaaS) platform in which the applications are designed following microservices or stateless functions principles. Users' requests for a specific application arrive at the scheduler[9] according to a well-defined distribution, and the scheduler schedules containers in an available server to complete each request. A request and a container are equivalent since container start-up and wind-down times are small relative to the duration of the container, and the containers' duration is randomly sampled, such that start-up and wind-down times can be absorbed into the distribution.

The cloud platform is composed of n_s servers, with normalized capacity of 1 in each resource (memory, cpu). Fixed, nonzero server boot-up and shut-down times are assumed throughout. The scheduler is assumed to have perfect knowledge of the state of the system (i.e. resource availability at each server). This is the case at different granularity levels in typical cloud environments [11].

3.2 Analysis Methodology

This work presents theoretical analysis and simulation results of single- and multi-resource scenarios. The two types of analysis are described in the following:

Mathematical analysis, Sect. 4.1. The memory requirements of containers are studied and parameterized by the average level of sharing across containers (f), the ratio of library to container memory requirements (r), and the memory requirements of the containers (v). The analysis considers a single-resource scenario (memory), where containers have similar memory requirements and their library sets are randomly sampled from a finite set of libraries.

Simulations, Sect. 4.2. Several simulation scenarios are considered in order to: (1) validate the mathematical analysis, (2) study the performance of the system under different scheduling algorithms and under realistic load traces. Details about the setting and traces are provided in the relevant sections.

[9] Or frontend. We refer here to the logical entity in the cloud infrastructure in charge of receiving requests and assigning them to the cloud's resources.

3.3 Simulator

In order to single-out the parameters of our abstracted cloud model in a large-scale context, we designed and implemented an event-driven simulator in Python 2.7, for arrival and scheduling across an arbitrary number of abstracted servers using four different scheduling algorithms, (First Fit, Greedy Fit, Tetris [5], and a mixed-integer linear-programming scheduler), in both single and multi-resource scenarios (memory, CPU, disk, I/O, and so on).

The simulator can generate requests on the fly according to predefined probability distributions, or use predefined container types (both described in Sect. 4) arriving according to Poisson distributions or uniformly on a time interval. Scheduling events periodically trigger the orchestrator, which queries the state of the servers and then runs the chosen algorithm to schedule containers in available servers. A hysteresis controller governs the boot-up and shut-down of servers by estimating resource utilization at future times assuming linear system dynamics [1]. The state of the simulation is logged periodically.

A pictorial representation of the inner workings of the simulator is presented in Fig. 2 below.

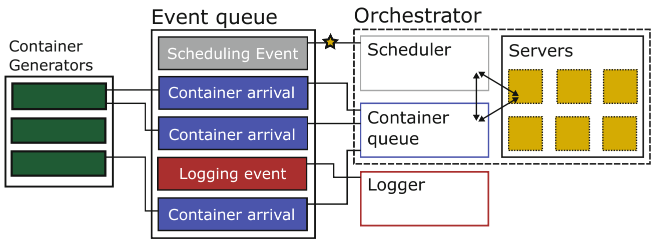


Fig. 2. Container generators (left) produce container arrival events, which contribute to form a container queue. Scheduling events trigger the orchestrator, which prompts the scheduler to query the queue and the servers and schedule containers appropriately. The status of the simulator is logged periodically. The controller (not shown) monitors the container queue and the servers, estimates resource utilization at future times assuming linear system dynamics, and triggers server boot-ups and shut-downs as needed to handle the load using a hysteresis-based approach [1].

4 Performance Evaluation

Terminology. Server capacities are normalized to 1 for each resource. The memory requirements of a container (excluding libraries) are denoted by v. The memory requirements of the library set for a container is given by $r \cdot v$, so that the total memory required by the container is $v \cdot (1 + r)$. A value of $r = 1$ thus indicates that 50% of the container's total required memory is due to the library set. If a container shares libraries with other containers on the same server, then the effective memory required by the libraries is smaller. We introduce the variable f to represent the fraction of libraries shared by any two containers, on average. Therefore, if two containers require a library set of 10 libraries, a value of $f = 0.1$ would indicate that the containers would share 1 library, on average.

The following section describes the mathematical analysis. This is followed by a discussion of the simulations.

4.1 Mathematical Analysis

The goal of this analysis is to study the relationship between memory utilization and the three variables: v, r, and f. Assume each container has the same resource requirements v, and utilizes a library set of m libraries, selected uniformly at random from a set of n libraries, such that, in expectation, $f = m/n$. The scheduler schedules each container in the first available server that can accommodate it (FirstFit).

Under the conditions defined above, the expected number of containers that fit in each server, N, is given by the following implicit equation:

$$N \approx \frac{\mathbb{E}[c_{\text{avail}}]}{v} = \frac{1 - \frac{v \cdot r}{f}(1 - (1-f)^N)}{v}. \tag{1}$$

For a derivation of this result see Appendix A.

To quantify the impact of library sharing, we introduce the concept of **relative utilization**, u, which is the ratio of the total utilized resources to the total utilized resources when libraries are not shared (the worst-case scenario).

Figure 3 shows the value u obtained from solving Eq. (1) for various values of v and r, with f varying between 0 and 1. Several aspects are worthy of note:

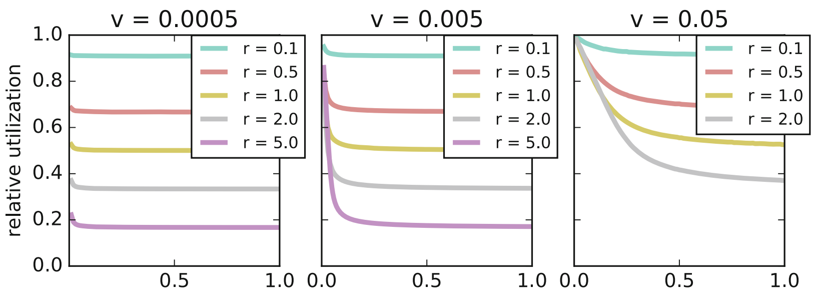


Fig. 3. Relative utilization as a function of f for various values of v and r, obtained from solving the implicit Eq. (1).

Observation 1: Higher values of f reduce the effective resource requirements of containers.

Observation 2: r governs the "steady-state" of utilization in terms of f, determining the savings that can be obtained through library sharing by regulating the relative impact of libraries on the overall resource requirements of the tasks.

Observation 3: A low f is sufficient to yield significant savings.

Observation 3 is less intuitive than the others, but of great importance, as it suggests that even small levels of library-sharing across containers suffice to achieve most of the memory savings that one could hope for. Note, for example, that for $v = 0.005$ there is little difference between $f = 0.1$ and $f = 1$.

Observation 4: Larger v (the memory requirements of a container, excluding libraries) and r (the ratio of library memory requirements relative to v) both result in a delay of the "steady-state" in terms of f (the fraction of libraries shared by any two containers, on average).

Observation 4 implies that a larger f is required to produce the same savings when resource requirements of each container are larger, suggesting that container-based approaches have more to gain from library sharing, as container memory requirements are typically small relative to the server capacities.

4.2 Simulations

Three sets of simulations were conducted, exploring different scenarios.

The first set of simulations, **A1**, studies a single-resource scenario similar to the one assumed by the analysis in Sect. 4.1. **A2** considers a multi-resource scenario with multiple container types arriving in a Poisson fashion. Finally, **A3** studies a multi-resource scenario where requests arrive in bulk (100 s of containers at the same time).

A1. This is a single-resource simulation with 50 servers, representative of a small cluster [5], each with capacity $c = 1$. Four different container types and distributions are considered, with different v, r, and Poisson rate of arrival, λ. These are specified in Table 2 below. A FirstFit algorithm that places each container in the first available server that can accommodate it was used for

Table 2. Parameters for simulation set **A1**.

	S1.a	S1.b	S1.c	S1.d
v	$U(0, 0.01)$	$U(0, 0.01)$	$U(0, 0.003)$	$U(0, 0.1)$
r	1.0	0.1	5.0	1.0
λ	50	50	50	5

scheduling. The key difference between this simulation and the setting in Sect. 4.1 is that the container sizes, v, are sampled uniformly in some interval.

Figure 4 shows the results of simulation **S1**.a for different values of f (left), where we see that $f = 0.05$ reduces utilized memory by nearly 50%, confirming the previous finding that even a small level of overlap between container library sets suffices to yield significant savings. Figure 4 also shows the agreement in relative utilization between the simulations and the analysis in Sect. 4.1 (right), validating Eq. 1 even when v is randomly sampled.

The initial bump in the active server count is a result of the finite server boot-up times, as the controller initializes a large number of servers to accommodate the requests accumulated at the beginning of the simulation.

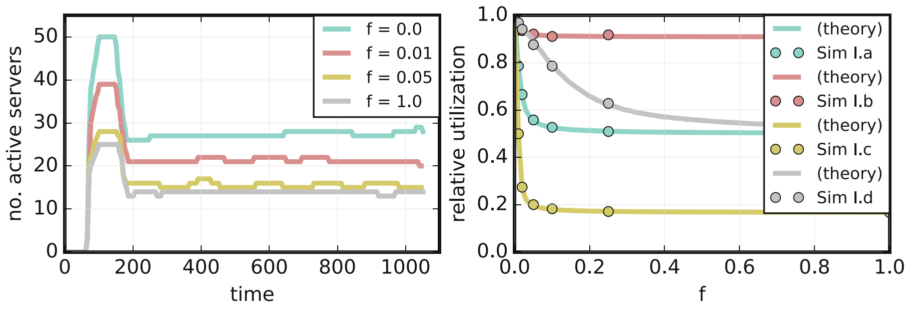


Fig. 4. Left: Number of active servers over time in a simulation with $v \sim U(0, 0.01)$, $r = 1$, and $f \in (0, 0.01, 0.05, 1.0)$. Each simulation runs until $5 \cdot 10^5$ containers are processed. Right: Comparison of relative total utilization in simulations **A1** versus theoretical results from Sect. 4.1, showing clear agreement.

A2. The second set of simulations considers a multi-resource scenario (memory and cpu) with 20 container types, each arriving according to a Poisson distribution where the arrival rate, $\lambda = 0.4$ (number of arrivals per unit of time). The container size, v, for both memory and cpu is sampled according to the cumulative distribution functions described in [10], falling in the range $(10^{-4}, 10^{-1})$. Each container type's library sets were chosen such that $r = 1$ and such that any two container types share 10% of their libraries (so that $f = 0.1$). Note that this inflates the memory requirement of the containers relative to [10], but it illustrates the point when memory is the dominant resource.

Figure 5 shows the result of the simulation when libraries are not shared (left) and when libraries are shared (right). We observe a drastic drop of about 40% in the required number of servers when libraries are being shared. We also note that by reducing the memory saturation, fewer inputs for server boot-ups and shut-downs are needed from the controller, resulting in a more stable number of active servers over time.

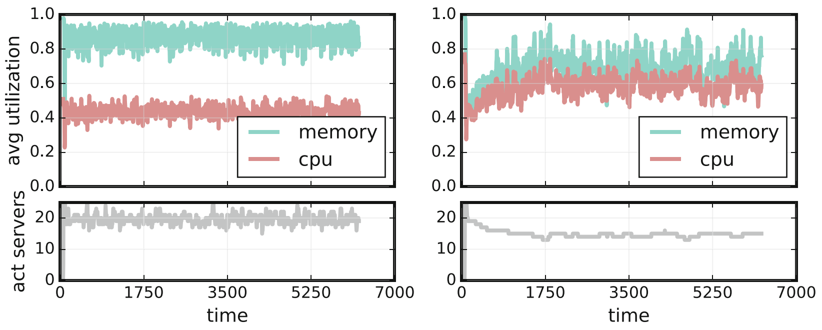


Fig. 5. Normalized memory and cpu utilization and number of active servers over time for simulation **A2**. The plots on the left show the result when libraries are not shared, while the plots on the right showcase the results when libraries are shared.

A3. This simulation set considers a multi-resource scenario where containers arrive in bulk, representing large *jobs* such as map-reduce tasks. Each job is randomly chosen as small (100 containers) or large (500 containers) with equal probability. Containers are chosen randomly from 20 container types, with randomly chosen libraries such that $f = 0.1$ across container types. Each container type is either high- or low-memory and high- or low-cpu ($v = 0.10$ or $v = 0.025$, respectively). The size of the libraries is adjusted such that $r = 1.0$. Arrival times are sampled uniformly between $t = 0$ and $t = 3600$, and there are 200 jobs in total. This simulation setting is similar to the one in [5].

Having quantified the memory savings one can achieve through library sharing, this simulation considers how to best take advantage of library set overlaps between different containers, at the scheduler's level. To do this, we propose a mixed-integer linear-programming (MILP) scheduling algorithm that exploits the cost of libraries on each server, while attempting to minimize waiting time.

MILP scheduler: Let $X_{(i,j),k}$ be the number of containers of type i from job j scheduled on server k. Let $\mathbf{t}_{(i,j),k}$ be the total resource requirements of container of type i from job j on server k and $\mathbf{c}_k$ be the vector of resource capacities for server k. Let $A_{(i,j),k}$ be the corresponding score of scheduling a container of type i from job j on server k. Each entry $A_{(i,j),k}$ is a weighted combination of scores for waiting time $(t_{\max} - t_{(i,j),k})$, the shortest time remaining to finish (STRF) metric $(s_{\max} - s_{(i,j),k})$, and a fairness score (amount of resources below the job's fair-share). The algorithm is summarized below.

This is a score maximization program with linear constraints. Note that constraint 4 ensures that at most $n_{(i,j)}$ containers of type i from job j get scheduled, while constraint 5 prevents the servers' capacities from being exceeded, thereby avoiding overallocation. Gurobi [6] was used to obtain an approximate solution to the problem at each allocation interval (<100 ms per scheduling event).

Algorithm 1. MILP scheduling algorithm

```
letN=Y
Require: containers, jobs, servers
  $A \leftarrow$ score(containers, jobs, servers)
  Solve:
```

$$\max_{X} \quad \sum_{(i,j)} \sum_{k} X_{(i,j),k} A_{(i,j),k} \tag{2}$$

$$\text{s.t.} \quad \sum_{k} X_{(i,j),k} \leq n_{(i,j)}, \forall (i,j) \tag{3}$$

$$\sum_{(i,j)} X_{(i,j),k} \mathbf{t}_{(i,j),k} \leq \mathbf{c}_k, \forall k \tag{4}$$

$$X_{(i,j),k} \in \mathbb{Z}_0^+, \forall (i,j), k \tag{5}$$

```
  for each $(i,j), k$ do
    schedule $X_{(i,j),k}$ containers of type $i$ from job $j$ on server $k$
```

Efficiency: The size of the problem is proportional to the number of servers and of distinct job/container type tuples. Scheduling separately on different sets of servers can help improve efficiency in scheduling at a small cost in the solution quality (provided that the server subsets are still large enough). Jobs and container types can also be classified into a smaller, perhaps fixed, number of job and container type classes, or divided into separate scheduling groups in order to minimize the scaling effects on the solver (approaches of this flavor have been adopted in scheduling literature before [2]). Note that this scheduler assigns many tasks to servers in a single scheduling event, such that such events can happen at less regular intervals.

Tetris. For comparison purposes, we also adapted the Tetris scheduler [5] to our problem, as it has been found to perform well in reducing job processing times. In this simulation we do not consider writes and reads over the network, which are explicitly accounted for in Tetris.

Figure 6 shows the average waiting times for the 20 different container types for all three algorithms. Waiting times using the MILP scheduler were reduced by 22% and 41% compared with Tetris and FirstFit, respectively. Job processing times (arrival to end of processing of the job's last task) were reduced by 18% and 19%. This can be attributed to the fact that the algorithm schedules all available containers in a single-pass, thus taking explicit advantage of the overlap between the containers' library sets and those already loaded in the servers.

Note that, unlike Tetris, the MILP scheduler accounts for packing in the constraints, so it does not require an alignment score in the cost function. This also contributes to the improved results. Overall, the results suggest that accounting for containers' libraries explicitly when allocating a large number of containers simultaneously is advantageous in a shared-library setting.

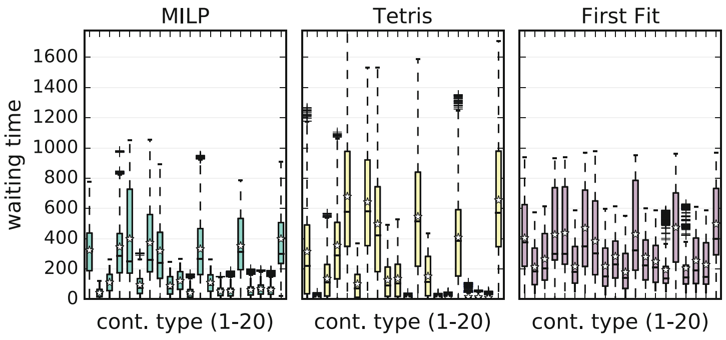


Fig. 6. Boxplot of waiting times for the 20 different container types in **A3** using three algorithms: MILP, Tetris, and First Fit.

4.3 Real Case Scenario

The analysis above covers a very large spectrum of real case scenarios. As an example, the analysis in [8] presents results about Docker images contained in Docker Hub and describes a situation in which different images share common AUFS layers. For the most downloaded docker container images, the authors show that the top layer of 90% of the images represents less than 10% of the size of the whole image [8, Fig. 13].

These scenarios fall within the parameterization in our analysis. Specifically, the results in [8] suggest a large value of $r > 5.0$ (since the topmost layer is typically a small fraction of the container), and a value of $v < 0.005$ for a server with 32 GB of RAM, in the majority of cases. The fraction of shared layers is not explicitly reported, but as the analysis in the above section has made clear, a small f of even 0.05 would drastically reduce the memory required by the containers. In particular, Fig. 3 and Eq. 1 suggest that if the images share 5% of their layers, then for $r = 5.0$ and $v = 0.005$ one could expect a relative utilization of 33.1% of that achieved when layer sharing is not used.

5 Conclusions and Future Work

Sharing libraries using filesystems such as AUFS offers a convenient yet effective solution to combat memory duplication in container-based cloud applications. Our mathematical analysis and simulations showed that the memory used by a container can be reduced by nearly 50% when the containers' library sets are as large as the container itself ($r = 1$), even if any two containers share just 10% of their libraries, on average. Our proposed MILP scheduler further improved on the results by considering the scheduling of hundreds of containers at once when requests arrive in bulk, reducing waiting times and processing times by about 22% and 18% respectively, relative to state-of-the-art schedulers. Its generality as a score-maximization algorithm also opens the door to many possible scoring functions that could include locality and architectural constraints [9].

Appendix A

Derivation of Equation 1

Recall that each container has the same resource requirements v, and a set of m libraries randomly sampled from a larger set of n total libraries, such that $f = m/n$.

The probability that a particular library is not one of the m libraries used by a particular container is simply $(n-m)/n = 1-f$. Letting N denote the number of containers in a particular server, then the probability that a particular library is loaded in that server is given by

$$p = 1 - (1-f)^N \tag{6}$$

The expected number of unique libraries in the server, n_L, is thus $\mathbb{E}[n_L] = n \cdot p$.

Using the chosen terminology, the memory required by each library is given by $s_L = v \cdot r/m$. We can therefore estimate the expected available capacity in a server after discounting all libraries loaded in memory:

$$\mathbb{E}[c_{\text{avail}}] = 1 - s_L \cdot \mathbb{E}[n_L] = 1 - \frac{v \cdot r}{m} \cdot n \cdot (1-(1-f)^N) = 1 - \frac{v \cdot r}{f}(1-(1-f)^N).$$

The remaining capacity is used for the containers themselves, each requiring v resources. As a result, we have

$$N \approx \frac{\mathbb{E}[c_{\text{avail}}]}{v} = \frac{1 - \frac{v \cdot r}{f}(1-(1-f)^N)}{v} \tag{7}$$

which implicitly defines N in terms of v, r, and f.

References

1. Bodík, P., Griffith, R., Sutton, C., Fox, A., Jordan, M., Patterson, D.: Statistical machine learning makes automatic control practical for internet datacenters. In: Proceedings of the 2009 Conference on Hot Topics in Cloud Computing, Hot-Cloud 2009. USENIX Association, Berkeley (2009). http://dl.acm.org/citation.cfm?id=1855533.1855545
2. Delimitrou, C., Kozyrakis, C.: Paragon: QoS-aware scheduling for heterogeneous datacenters. SIGPLAN Not. **48**(4), 77–88 (2013)
3. Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. CoRR abs/1606.04036 (2016). http://arxiv.org/abs/1606.04036
4. Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., Stoica, I.: Dominant resource fairness: fair allocation of multiple resource types. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, pp. 323–336. NSDI 2011. USENIX Association, Berkeley (2011). http://dl.acm.org/citation.cfm?id=1972457.1972490

5. Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., Akella, A.: Multi-resource packing for cluster schedulers. SIGCOMM Computer Communication Review, vol. 44, no. 4, August 2014
6. Gurobi Optimization Inc.: Gurobi optimizer reference manual (2015). http://www.gurobi.com
7. Haas, F.: Containers: just because everyone else is doing them wrong, doesn't mean you have to. https://www.hastexo.com/blogs/florian/2016/02/21/containers-just-because-everyone-else/. Accessed 10 Feb 2017
8. Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Slacker: fast distribution with lazy Docker containers. In: 14th USENIX Conference on File and Storage Technologies (FAST 2016), pp. 181–195. USENIX Association (2016)
9. Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Serverless computation with OpenLambda. In: 8th USENIX Workshop on Hot Topics in Cloud Computing. USENIX Association, June 2016
10. Reiss, C., Tumanov, A., Ganger, G.R., Katz, R.H., Kozuch, M.A.: Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In: Proceedings of the 3rd ACM Symposium on Cloud Computing, SoCC 2012, pp. 7:1–7:13. ACM, New York (2012). http://doi.acm.org/10.1145/2391229.2391236
11. Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., Wilkes, J.: Omega: flexible, scalable schedulers for large compute clusters. In: Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys 2013, pp. 351–364. ACM, New York (2013). http://doi.acm.org/10.1145/2465351.2465386