Regulating Traffic in a Crowded Cache: Overcoming the Container Explosion Problem

Kevin Gao
University of California, Berkeley

Tim Shaffer (Advisor)
University of Notre Dame

Kyle Chard (Advisor)
University of Chicago

***Abstract*—Multi-user interactive computing services, such as Binder, dynamically create and deploy software containers to provide customized execution environments with required system and language dependencies. Unfortunately, container creation can be slow, making services unresponsive to users, while caching leads to combinatorial explosion in the number of containers due to the infinite possible combinations, versions, code changes, and users. We analyze 13,946,918 Binder launches and explore caching strategies that consider various features, including package popularity, version stability, recent use, and install time and size. We show that our methods can reduce total storage consumption by 1–3% and creation time by 6–11% when compared with a least recently used strategy.**

I. Introduction

Binder [1] is a multi-tenant service that enables users to *execute* a Git repository by providing an interactive interface to the Jupyter notebooks contained in that repository. Binder builds a custom container for every execution using specifications in the repository and the *repo2docker* tool. Unfortunately, building containers can be time-consuming, with a lower bound of several minutes to build with few dependencies. As a result, services like Binder cache containers for short periods of time to rapidly serve repeated invocations. However, in practice, the Binder workload is so diverse (e.g., Binder logs show 112,793 distinct repositories) that effective caching is difficult.

In this paper we explore Binder workload traces that provision Python environments. We augment these traces with additional features (e.g., container specifications, package installation times, and popularity) and explore various strategies for reusing and caching containers under different constraints.

II. Workload Traces

Binder published a workload with $\sim$14 million launch records, over the period of 3 years. Each launch record includes the time and target Git repository. We crawled the target repositories and obtained all repositories that included Python configuration files (i.e., pip and Conda). This resulted in 33,987 repositories and 2,011,412 associated launches [2].

We subsequently augmented this dataset with the following features for each Python package:

1) Install time and size. We developed a profiler system that deploys an empty virtual environment, installs the package, and records time and size. We repeated experiments for each package ten times and take the median.
2) Popularity statistics including GitHub stars/forks and downloads, obtained from PyPI, libraries.io, and Github.
3) Version lifetime by obtaining all releases from Conda or PyPI and computing the average time between releases.

Fig. 1 shows the correlation between our collected features.

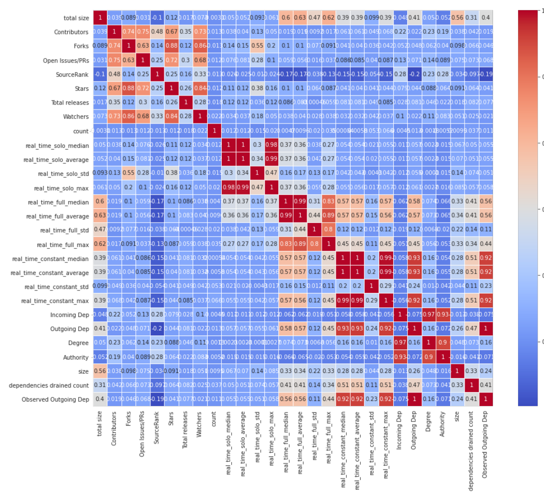

Fig. 1: Pairwise Relationships For Collected Factors

III. Simulation

We developed a simulator to explore different caching models. The simulator “plays” the binder workload and records performance metrics (e.g., time to create containers, responsiveness to requests, cache space) under configurable cache constraints (e.g., number of containers and cache size).

IV. Container Sharing

One way to reduce the number of containers and improve cache performance is to share containers between invocations for different repositories. We explored three models.

1) Baseline: No containers are shared. Requests must use a unique container for each repository.
2) Identical: Repositories with the same packages may share containers.
3) Contained: Repositories with a subset of packages from another repository may share that container.

We further constrain container sharing to consider package versions: containers may be incompatible when package versions are miss-matched. While this reduces opportunities for sharing, it is needed to meet user expectations.

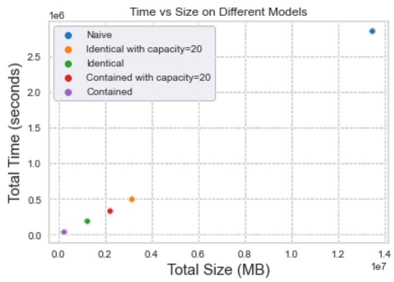

Fig. 2: Performance of different sharing models

Fig. 2 shows the time to create containers and their size with these policies over the multi-year trace. The figure shows that reuse can improve performance over the baseline by ~5x. If we impose a constraint on cache capacity (in this case 20 containers) we see improvement of ~2x.

V. Caching Metrics

We explore five metrics for each package. **Versions:** Average time between versions. **Popularity**: Average number of stars and forks. **Size**: Total size on disk. **Time**: Total time to install. **Dynamic Count**: An online approach that counts the number of package invocations within a sliding window.

We evaluate caching strategies that order containers based on each metric. Table I shows the average performance of each strategy using our simulator and Binder trace. We compare against a baseline least recently used (LRU) implementation. Our results suggest that individually, LRU is best.

Metric	Size (TB)	Time (Hrs)	Hit rate (%)
LRU	2.71	115.39	84.61
Dynamic	5.11	128.05	83.39
Size	4.90	123.25	83.47
Popularity	5.68	130.11	83.11
Time	4.75	119.37	83.84
Versions	5.89	134.26	82.75

TABLE I: Average cache strategy performance with cache size limits of [2000, 4000, 6000, 8000, 10000] megabytes.

VI. Weighted Combination: CacheRank

Without an obvious "best" metric, we combined metrics into a single ranking metric (CacheRank), implemented as follows.

```
for met in metrics:
  rk = Rank(met)
  score += met_coeff * (rk - mean(rk)) / std(rk)
```

This approach aims to normalize the different metrics' scores, efficiently account for overlap, and utilizes input weightings for the factors. We use random values from a Gaussian distribution as parameters for our simulations. Fig. 3 shows the best performance at every cache size limit and the improvement over LRU in terms of size, time, and hit rate.

VII. MRU Protection

CacheRank weights the ranking of each container uniformly throughout the cache. However, this is likely to be flawed as the containers at the front of the cache (most recently

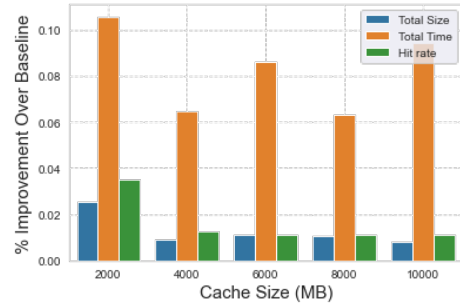

Fig. 3: Improvement over LRU for different cache sizes.

used, MRU) are more impactful proportionally to the least recently used containers. To integrate this observation we explored protecting the MRU containers in the cache. In our simulations, we set the MRU protection parameter *cache_safe* to be the 20^{th} percentile for the cache limit being evaluated. We can see in Fig. 4 that this strategy provides the best performance as the most optimal value of *cache_safe* lies somewhere between 20% and 60% of the capacity.

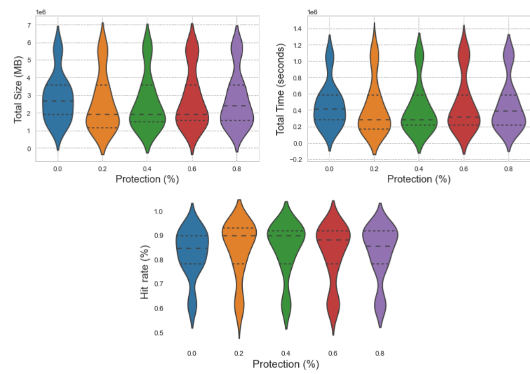

Fig. 4: Distribution and quartiles for protection fractions

With the unprotected sector of the cache, we tested different heuristics to prioritize the removal of containers, using: the metric score as a sole basis for decision, the smallest total size based on a minimum Knapsack algorithm, and a linear combination of both score and size. We found the linear combination outperformed the other methods.

VIII. Summary

We explored container sharing and caching strategies to improve the performance of multi-user computing services. We collected features such as installed package size, installation time, popularity metrics, and version history to create metrics that could be combined with an LRU cache. Our strategies, CacheRank and MRU protection, use these metrics to improve cache performance under space constraints.

References

[1] Binder, https://mybinder.org, 2021
[2] T. Shaffer, K. Chard, D. Thain, "An Empirical Study of Package Dependencies and Lifetimes in Binder Python Containers". eScience, 2021.
[3] T. Shaffer et al., "Solving the Container Explosion Problem for Distributed High Throughput Computing," IPDPS, 2020.