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

Kevin Gao
University of California, Berkeley

Kyle Chard
University of Chicago
Argonne National Laboratory

Tim Shaffer
University of Notre Dame

***Abstract*—We used a publicly available dataset containing binder launch records in the form of two tables. The launches table contains 14 million launch records, with columns that show the timestamp of the launch and index to link to an image in the specifications table. In this second table, a recipe is given for every unique launch, with raw text specifying the required packages and optionally, their versions, when needed to build a Binder container.**

I. Introduction

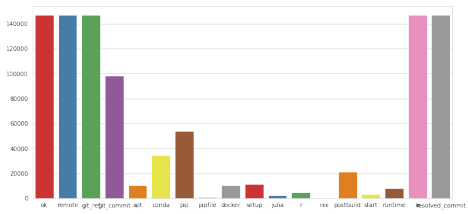
Due to the increasing popularity in modern applications relying on a large number of package systems managed through pip, conda, apt, etc., there is a wider range and ease of access to tools for the user, at the cost of an urgent need to manage a messier and more modular dependency system. In the worst case, a user would have to prepare full software environments and continually manage them as updates roll in, which can get complicated when free floating packages each have their own release schedule and unknown cross-compatibility issues with other dependencies. Containers have emerged as a solution to ease much of the burden from the user when it comes to assembling and distributing applications. At a high level, some of its main benefits include greater control and flexibility over dependencies and reproducibility across different systems [1].Today, services such as Binder, JupyterHub, Whole Tale, and Code Ocean provide dynamic research environments for users, and in this project our input data relies on a dataset of Binder launch records that were collected through crawling a large number of repositories on Github and other sources.

Binder is a an online application that builds software environments based on what the user specifies. In the "naive" case, the user must prepare that the right packages are installed, and that each is set to the right version. For example, running a rather outdated script that requires an older version of a library might require the user to downgrade their versions for the time being, and the added hassle of needing to also have the compiler, package manager, and other framework pieces can be a straining process. When using a tool such as Binder, the user specifies the environment that they want to run, such as a Github repository. Deployed on remote servers, Binder uses the specifications from the repository or source and builds a containerized version of that environment, which is then passed to the user as a ready-to-go notebook that can be modified or run without using up the user's personal resources. Consequently, this also allows for easier sharing and collaboration [2].

The introduction of containers and Binder leads way to the container explosion problem, defined as: given a large (and probably growing) number of jobs that require many overlapping software dependencies, simply creating a container to fulfill each job's requirements will lead to a combinatorial explosion in the number of images stored [2]. In the case where no containers are shared, a new containerized environment is created for every launch, leading to an obviously inefficient outcome with the worst possible bounds in terms of size and runtime. On the other side of a spectrum, a single large container may seem optimal in covering a large number of launches at once, but fails due to constraints on the size and transferability of a container. When we consider a system that must deal with comparing and sharing containers, one important observation is how much we can keep in memory at any time. Knowing that there are constraints on a container size and the amount of containers we can "remember" at any given time, it makes sense to build a cache holding the most recently used containers. We first create a least recently used cache (LRUCache) and compare its results to the naive model with no sharing. Then, we collect additional data that can be used as factors to create new metrics that can be assigned to containers in the cache. Using our LRUCache as a baseline, we integrate and assess the quality of our new metrics and any weighted combination of them given constraints on container size or the capacity of our cache.

II. Describe data

Data Collection We used a publicly available dataset containing binder launch records in the form of two tables, launches and specs. The launches table contains 14 million launch records, with each row representing an instance of a Binder launch. The columns show the timestamp of the launch and index to link to an image in the specifications table. In this second table, a recipe is given for every unique launch, with raw text specifying the required packages and optionally, their versions, when needed to build a Binder container. In our analysis and utilization of Binder launch records, we chose to work exclusively with entries requiring pip, which had the highest frequencies as shown in Figure 1, parsing the input text into a dataframe format with easy access to the versions and packages required for a particular specification.



Name	Null values	Null %
ok	0	0.0
remote	0	0.0
git_ref	0	0.0
git_commit	0	0.0
apt	155452	0.94
conda	131613	0.79
pip	111902	0.67
pipfile	165166	0.99
docker	155892	0.94
setup	155002	0.93
julia	163536	0.98
r	161233	0.97
nix	166044	1.0
postbuild	144966	0.87
start	162859	0.98
runtime	157931	0.95
ls	9825	0.06
resolved_commit	19334	0.12

Fig. 1. Frequency of Specification Calls By Specs Columns

Beyond the data presented in the launches and specifications datasets, there were a considerable amount of additional factors to explore when expanding the capabilities of our model, namely the size, installation time, list of dependencies, version history and various statistics for a given package. The most central of these would be the construction of a dependency tree, such that we are able to know the full dependencies for any Python package in pip. The importance of having a quick lookup to the full dependencies for a package extend to the ability to find the individual size and installation times for a package, and fixing the incomplete generation of the dependency table from the specification dataset. The size and installation times for a package were determined for each package through benchmarking and collecting sizes in a virtual environment. Finally, the version history and statistics such as Github stars or forks were pulled from pypi and libraries.io.

In order to determine the sizes, times and dependencies needed to install each Python package, we set up a new virtual environment for every package with only Python 3.6 and pip installed. After activating the environment, the package was installed using pip, with a function to benchmark the process. Among many other outputs, the size of the package and its list of dependencies could be determined through a command line argument. After a package was finished, the virtual environment would be destroyed and cache for the environment stored in the main instance would be removed, thus setting the stage for the next package and virtual environment. This process was repeated using a bash script, and every package was timed ten times to account for variation in installation times and connectivity to the network or instance. The benchmarking of 1000 packages took roughly 5 hours to complete.

A glaring issue that must be addressed is finding the *individual* installation time and size for a package. When a package is installed in an empty environment, it installs all of its dependencies as well, and this is factored in both the installation time and the size when the data is collected. Although this could be left as is for the time being, the problem would have to be solved somewhere down the line, so some strategies were used to utilize the dependencies list in resolving these issues at this stage. The first idea would be to run an algorithm that would, given the total size, time, and list of dependencies for each package, be able to separate them so that an individual size and time would be calculated for each package. The algorithm would repeatedly run iterations until a natural stopping point, where on each iteration, it would pop the packages with empty dependency lists, take their sizes and times, and ”subtract” them from all other packages containing them, while removing all instances of that package in the process. This method proved to be successful for sorting out sizes, but as it proved to be flawed for finding times (with an abundance of negative values at the end), a different strategy was applied directly in the bash script. Hinging on the fact that only the package itself (and not its dependencies) are uninstalled when attempting to uninstall the package, an effective add-on would simply be to uninstall the package and then benchmark the second installation of the package (which will only install the package, and not its dependencies).

A myriad of statistics can be found on websites that display Python package information and its history of versions, and often the package will have its own webpage. Given that we are crawling through every Python package, we needed a reliable way to locate every package, so we used pages that host all Python packages, such as pypi and libraries.io. From each package’s page, factors being collected included stars, forks, watchers, contributers (Github), repository size, release date, and SourceRank, a score calculated by libraries.io’s algorithm based on a number of metrics. Pulling from two sources prevented a considerable amount of NaN values, as it was often the case that one site had information on a factor that another did not. For the package’s version, each version number and its timestamp for release time was scraped and placed into a list for the package.

Data Augmentation The agglomeration of raw data prompted a closer look into new factors taken from weighted combinations of existing data. Once again, the dependencies of packages would play a large role in influencing much of the operations during this work. First, although we managed to find 5190 unique packages from the Binder launches, much of these packages were unfit to be used for analysis. Some were not installed properly when running the automated bash scripts, and some produced results that were not logically feasible, having extremely large sizes or installation times. The lack of information for some packages when crawling webpages, particularly on the package’s version history, also prompted the removal of these from the overall dataset. After factoring in dependencies into a package’s size and installation

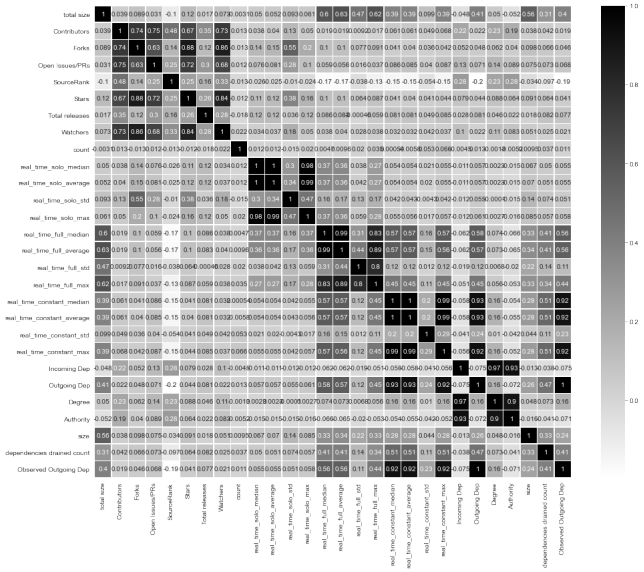


Fig. 2. Pairwise Relationships For Collected Factors

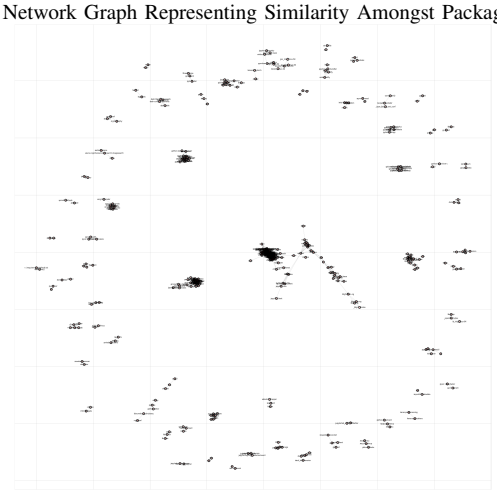
time requirements, such that each package's size and time correlates only to that single package (and not its dependencies as well), unneeded packages were filtered out, leaving a list of 4575 packages to work with. A list of full dependencies was connected to each package, such that a quick lookup could be made to find a package's dependencies and lookup for any input package. The updated package list was also applied to the specifications dataset.

A directed NetworkX graph with the libraries as the source node, the dependencies as the target nodes, and an arbitrary value as the weight was instantiated, as shown in Figure 2. In order to generate a meaningful graph, a threshold was supplied against the cosine similarity between any two nodes in the graph. As the threshold increased, the packages would have to be more similar in order for the edge to be kept in the graph. Using this data structure, additional factors were created, most notably the number and sum of the size of dependencies that filtered out of the overall data. Additionally, centrality measures were explored, and the authority value was chosen for analysis, representing the degree of incoming edges for a node.

Figure 3 shows exploratory visualizations characterizing the relationships between factors in the dataset. In the heatmap, there are 12 columns/rows corresponding to different time metrics. Columns tagged with 'full' correspond to the installation time of a package and all its dependencies, while 'solo' denotes the time for just the installation of the package itself. 'Constant', which was not used later on, is the time to install the package when it was already installed (hence nothing being installed).

Our model also considers the compatibility between versions from two separate launches. The timestamp for each launch and the individual versions of each package were converted into a UNIX time format for numerical comparisons.

Fig. 3. Network Graph Representing Similarity Amongst Package Nodes



In any recipe's specification, the packages required were optionally listed with a specific version. In the majority case where nothing was listed, the implication was that they wanted the most recent version, and this value was tagged onto the package for that launch by finding the highest UNIX time less than the time of the launch call.

Data Integration Having discussed the methods for obtaining and augmenting the data, we now have the right data for building a model. The first is derived from the launches dataset, with the timestamp for the invocation of the Binder launch, the index needed to connect it to its specification, and a new column with a dictionary structure, each key being a package in the recipe and value being the version number required for that package. The second, which connects to the launches dataset, is the dependency table that was generated from the specifications dataset, where the rows and columns are the repositories (indices of the launches) and unique packages, and the value being the version number requested (which can easily be converted into binary values, where 1 represents existence). Finally, the sizes, installation times, dependency lists, version histories, and (Github) statistics are factors in a final table where the rows are all the unique packages.

III. Simulator

Building a Simulator
Constraints
Versions
Heuristics
Weighted Combination
LRU Protection

IV. Evaluation

V. Conclusion

The conclusion goes here.

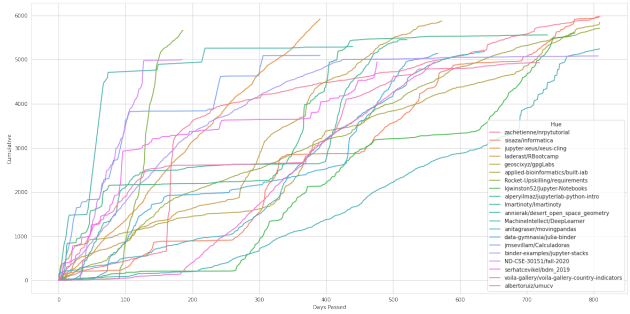


Fig. 4. Launch Invocations Over Time

	timestamp	combined_ref	Package Versions
1200000	2020-06-05T04:22:00+00:00	82543a814cdd4cb125d03112a193dd4137d8751c	{'matplotlib': '3.2.1', 'numpy': '1.18.5', 'pa...
1200001	2020-06-05T04:23:00+00:00	24b9fb3cf756b3c765579decd624132efe7be374	{'matplotlib': '3.2.1', 'numpy': '1.18.5', 'sc...
1200002	2020-06-05T04:23:00+00:00	8cd7cea22ba4863907308805af82e1ca44e57ea4	{'ipywidgets': '8.0.0a0', 'ipython': '7.15.0',...
1200003	2020-06-05T04:25:00+00:00	24b9fb3cf756b3c765579decd624132efe7be374	{'matplotlib': '3.2.1', 'numpy': '1.18.5', 'sc...
1200004	2020-06-05T04:25:00+00:00	4f1bf6107b6755066a9f7d6bfcc9d9e3f86aa12b	{'matplotlib': '3.2.1', 'numpy': '1.18.5', 'pi...
...	...	...	...
2011407	2021-06-06T23:52:00+00:00	99088cfb17d47b6244dadd40ef338dbe64a8102f	{'matplotlib': '3.4.2', 'numpy': '1.21.0rc1', ...
2011408	2021-06-06T23:53:00+00:00	adc0d7a7939696ca6e6415c1865a5a7ad716ea90	{'numpy': '1.21.0rc1', 'pandas': '1.2.4', 'ipy...

Fig. 5. Input Data: launches.db

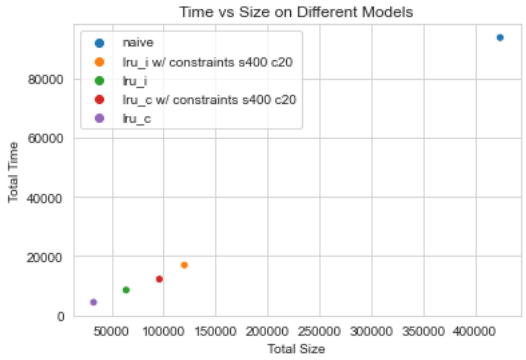


Fig. 6. Frequency of Specification Calls By Specs Columns

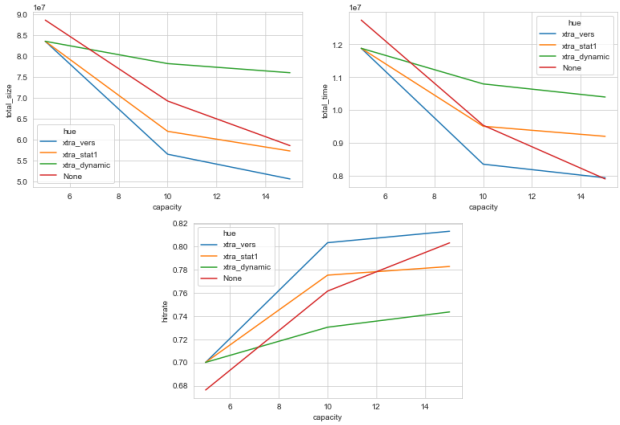


Fig. 7. Frequency of Specification Calls By Specs Columns

ACKNOWLEDGMENT

This work was supported in part by NSF grants OAC-1931348, OAC-1550588, and OAC-2004894, and by the U.S. Department of Energy under Contract DE-AC02-06CH11357. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Workforce Development for Teachers and Scientists, Office of Science Graduate Student Research (SCGSR) program. The SCGSR program is administered by the Oak Ridge Institute for Science and Education (ORISE) for the DOE. ORISE is managed by ORAU under contract number DE-SC0014664. All opinions expressed in this paper are the author's and do not necessarily reflect the policies and views of DOE, ORAU, or ORISE.

REFERENCES

[1] T. Shaffer, K. Chard, D. Thain, "An Empirical Study of Package Dependencies and Lifetimes in Binder Python Containers".

[2] T. Shaffer, N. Hazekamp, J. Blomer, and D. Thain, "Solving the Container Explosion Problem for Distributed High Throughput Computing," in International Parallel and Distributed Processing Symposium, 2020.