TierProvision: Characterizing applications for a scheduling algorithm in a multi-tier cloud

Abhishek Tiwari
Department of Computer Science
University of Toronto
Email: a.tiwari@mail.utoronto.ca

Siddhartha Thota
Department of Computer Science
University of Toronto
Email: thota@cs.toronto.edu

***Abstract*—Cloud computing resources are heterogenous, mostly because they are constantly resized, dynamically allocated and grouped together from multiple resource providers. Such an architecture can have unpredictable kind of resources, often clubbed together to work as one unit. Multi-tier cloud architecture is one example of this, where multiple tiers of one architecture is built by allocating resources at varying distance from the end user. Each tier can have heterogeneous nodes, and every tier has varying resources too. In such an architecture, a dynamic policy to allocate resources to run applications is needed. This project tries to characterize applications based on it's performance in relation to the resources it is allocated, thereby creating a metric that can be used to make future decisions about the application. This measure also lets us make decisions about reprovisioning applications while having least impact on performance guarantees. The approach is based on experiments run on docker hosts, running on local machines.**

***Keywords*—multi-tier cloud, scheduling**

I. Introduction

Cloud computing resources scale elastically, and hence reduce the risk of under provisioning and wastage due to over provisioning during non-peak hours. Further, with the widespread use of mobile devices and their increasing resources, Mobile Edge Computing is gaining importance. By offloading computation to edge servers, device energy consumption and execution delays can be greatly improved. This paradigm introduces new scheduling and provisioning challenges that are yet to be solved. We seek to address the problem of dynamic resource provisioning in multi-tier cloud data centres.

II. Background

Several research efforts have been made to efficiently solve the dynamic resource allocation problem. Some algorithms use queuing models that determine which tier to re-provision, like in Urgaonkar et. al.[1]. As opposed to a fixed, pre-built model, our approach is to dynamically understand the performance characteristics of an application.

There has been work considering low level hardware based performance models for scheduling, such as a paper by Marin et. al.[2]. In our view, this work might be a good foundation for continuing from where our work leaves off, since Marin et. al. provide a way of predicting application's performance

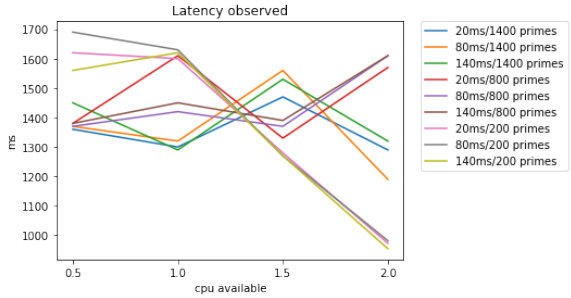

Fig. 1. Latency variation with CPU available

based on it's characteristics, independent of the architecture on which it runs. This is essentially how we translate our scheduling problem, as we show in the rest of the report. There is also some work[3] around using control theory to predict performance of an app, rather than using a predictive approach. We do not use this approach, rather, we aim for a system that learns based on historic performance and adapts online to application's statistics.

Some algorithms, such as the one proposed by Dejun et. al.[4] also factor in heterogeneity of resources in different tiers and use profiling of machine performance to select a tier, and our design is also similar to this. Instead of profiling, we continuously track the performance of the app on a particular tier, and record it's statistics along with the specification of the tier it's running on.

The following sections will be: System Desgin, where we elaborate the phases of our system, Implementation, where we show the components and how they were built, and a Results section to elaborate our results. A conclusion and a future work section will follow.

III. System Design

The application was initially designed using Nomad, an abstraction over Docker, and deployed on multiple machines running Ubuntu. As we went through our experiments, we realized we required more fine-grained control over the Docker than what Nomad offered, at which point we resorted to using Docker directly. Hence, the report describes the final

design, and offers an explanation of the failed steps in the final subsection.

A. A multi-tier environment

To best test our hypotheses about application behaviours, we needed to build a experiment setup similar to a multi-tier cloud environment. In order to best emulate this, we created two machines, each running a docker server, but each allowing only a certain amount of CPU and memory power to the docker instances running on it. We found that the performance of the two machines were quite similar, so in order to bring in a latency factor, we added soft latency to the application itself, more on which will be detailed in the respective subsection. We then wrote a few scripts that would automate the process of monitoring and scheduling tasks for us, which took substantial amount of our research time.

B. Task queueing and Monitoring

We designed an API for end user to interact with this scheduler. The scheduler has a Task API, wherein a new task can be defined. Every new task comes with it's own docker image, which is simply run to run the task. Further, the user can create a "TaskRequest", which is a request to place the task at a particular latency tier, with a minimum CPU and other resource guarantees. Further, the Node table in this API can be accessed to get information about participating nodes in the scheduler. A background task updates the TaskStatus and Node tables with the most recent statistics.

C. Algorithm and Scheduler modules

In order to write and simulate algorithms, we write an "Algorithm" module with python. This module performs two distint tasks. First task is to allocate tasks from the task queue exposed to the User. In this task, the algorithm can obtain the current status of the nodes and the tasks running. The second task is reprovisioning, which is a cleanup task that is run periodically. A cleanup (or reorganization) algorithm can be engineered, so that even without new incoming task requests, a continuous maintenance of tasks is done. This component is flexible, and can be extended to customize algorithms. Adding a new algorithm is just a matter of extending the base Algorithm class and overriding the methods for task queue consumption and reprovisioning.

Scheduler module is a multi-threaded python module that keeps the Algorithms and the monitoring scripts running. Scheduler schedules the reprovisioning function regularly, and schedules a task consumption worker periodically. Scheduler is a flexible module and can be configured to use any algorithm.

D. Simulated task

A Cards Against Humanity API was customized to be used as a sample app. However, it was found, over a period of testing various conditions of the API, that it's hard to simulate a "hard task" to partially occupy RAM and CPU. Several approaches were attempted, including repetitive simple number multiplication, sleep-work-sleep cycles etc for CPU and creating string variable of length over a million to emulate CPU consumption. Outcomes of these experiments are outlined in the results section, however, we finally settled for a prime number counting API, that essentially just counts the nth prime number, higher n for higher load. This gave a good balance between a task that takes adjustimble amount of time, occupies CPU and is predictable. A soft delay was also introduced inside this API, which was configurable on instantiation. This soft delay is used to simulate network latency. A combination of simulated network latency and the task complexity is shown in fig. 1.

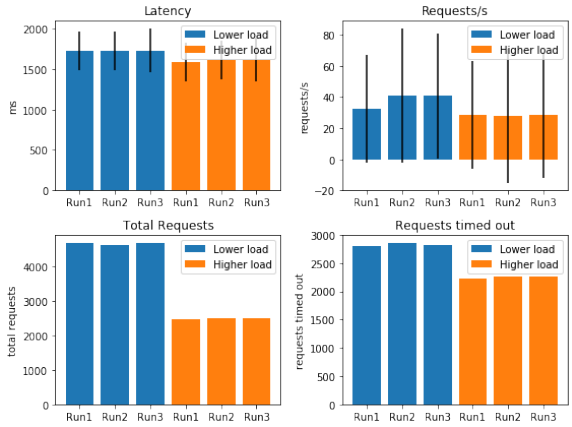

Fig. 2. Three runs of light and heavy loaded application

IV. Implementation

The scheduler and algorithm components are in python, since it is an API that talks to a database backend. For data storage, we used a simple MySQL database, directly reflecting the Task-Node components. The Cards Against Humanity app is in javascript, since it's the easiest to deploy with a docker.

A. Experiment setup

To perform an experiment, a new task is created, and a Task Request is placed. As per the task request, the app will be placed at a particular tier. On being placed at a particular tier, it is assigned a fixed amount of CPU, memory. It is also programmed to simulate a network latency. Following this, a load testing tool (Apache Wrk) is used to test the API deployed. Statistics are collected from the load testing tool, and also from the task history tables filled by the monitor. These statistics are then processed offline.

V. Results

A. Varying CPU load on application

We wrote APIs that would be loaded synthetically while returning results. Our synthetic load tests were based on counting N prime numbers. Based on our tests, shown in fig 2, the applications would load the CPU fully and take increasing amount of time to respond with an answer. This consistent testcase helped us create reproducible load on an application while simulating their running on various tiers.

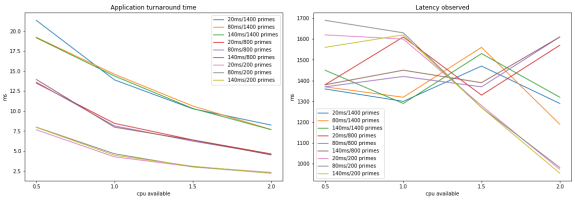

Fig. 3. Application turnaround time vs Latency Statistics

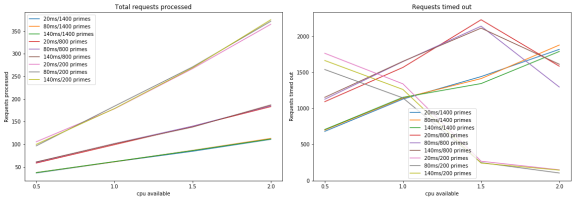

Fig. 4. Requests processed vs requests timed out

B. Tier to app relationship

Further, we chose two apps, one lighter and one heavier. We ran the two apps on two different tiers of docker hosts. In one tier, which we can assume to be closer to the user, the RTT is really less, but the processing speed and memory available are less. In the other tier, RTT is higher, but the CPU and memory availability is higher. We compared the performance average between these scenarios.

C. Turnaround time vs Latency

The results in 3 is intriguing, because while the application turnaround time (time consumed from the reception of request in the application, to when the application returns a result) continuously decreases as with the increase of resources allocated to it, which is natural. What's surprising is the latency statistics. The latency of these applications remain close to each other within the difference in their tier's RTT differences, but jump unpredictably when more resources are allocated. This is a behaviour that we have not been able to explain. In fact, this exact behaviour is why we think a general task-resource relationship model cannot predict such performance differences.

D. Heavy vs Light tasks

One interesting result observed in 4 is the fact that heavy applications benefit disproportionately more than lighter applications when switched to a higher tier (with more resources). This could be pointing out to the fact that heavier applications are worth moving to a higher tier because the RTT increase will be compensated by the performance improvement. Although this is intuitive and not ground-breaking, we felt that this result was important.

E. The case of lost requests

There is quite a bit of randomness in the way timed out requests changed when resources changed. This is inexplicable, and it seems like there's a scope for some more experimentation.

F. Future Work

As observable from our result section, in the scope of this project, we didn't actually evaluate an algorithm. This is owed to two factors: first, Nomad turned out to be a bad investment of time for this experiment, because it doesn't offer the fine grained control on Docker application the way we thought it would. It has a "CPU" and "memory" option in it's task configuration, but these are soft limits and are only realized when the entire system comes under load. Experimenting under that condition turned out to be chaotic. The second factor is, applications behaved much different than we expected, hence drawing us into more experimentation around this. We ended up synthesizing way more conditions than we anticipated we would.

The obvious next direction is to explore the behavior of applications under varying conditions, including available resources, network latencies etc. Further, we seek to come up with a measure (or a linear formula) to predict the performance of an application when placed at a particular tier, and have dynamic parameters regularly updated to account for changing behavior of the application. Past this, an actual algorithm for scheduling applications on heterogeneous tiers can be designed.

VI. Conclusion

Although we don't draw huge conclusions based on the limited scope of experiments we have already conducted, we observed that applications dont behave intuitively under varying circumstances. We also learned a great deal about designing, setting up and conducting experiments around operating systems, hardware limit simulation and varying network conditions. We also learned quite a bit about simulating load on a real-life application and load testing such applications. Further, our reading has given us quite a bit of exposure to current work, and has us excited about possibilities around our current work.

Acknowledgment

Our experiments were possible due to several open-source projects that are out there. Apache wrk, docker, python community and other have contributed in large parts to this experiment. We would also like to acknowledge the guidance of our project supervisor, Dr. Eyal de Lara, Professor, Department of Computer Science, University of Toronto.

References

[1] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An analytical model for multi-tier internet services and its applications," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 33, no. 1. ACM, 2005, pp. 291–302.

[2] G. Marin and J. Mellor-Crummey, "Cross-architecture performance predictions for scientific applications using parameterized models," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 1. ACM, 2004, pp. 2–13.

[3] E. Kalyvianaki, T. Charalambous, and S. Hand, "Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters," in *Proceedings of the 6th international conference on Autonomic computing*. ACM, 2009, pp. 117–126.

[4] J. Dejun, G. Pierre, and C.-H. Chi, "Resource provisioning of web applications in heterogeneous clouds," in *Proceedings of the 2nd USENIX conference on Web application development*. USENIX Association, 2011, pp. 5–5.