B-Optimal: Resource Allocation Optimization for High Workload Applications

Rahul Balakrishnan, Amog Kamsetty
CS 262 Final Paper
Email: rahulb@berkeley.edu, amogkamsetty@berkeley.edu

Abstract—We propose B-Optimal, a Bayesian Optimization based resource allocator for large-scale commercial or personal applications. Our optimizer automatically determines the optimal resource allocation of CPU Quota, Memory, Disk Bandwidth, and Network Bandwidth for different microservices by measuring the impact of configuration changes on application performance. We treat microservices as black-boxes and our end-to-end optimizer requires very minimal human intervention. While prior resource allocation systems function well for laptop applications, such as those running the ELK or MEAN Stack, we design a system that supports high traffic web-applications, adding features that enable horizontal scaling within the optimized application. We also provide an analysis on the feasibility, advantages, and shortcomings of Bayesian Optimization relative to other resource allocation techniques. B-Optimal outperforms prior work within the short term producing response latencies that are 25% and 66% lower than a gradient descent based optimizer across two different applications, respectively, after ten minutes of running both optimizers. Finally, horizontal scaling on B-Optimal produces a 52% improvement from a balanced replica configuration, where every service has an equal number of replicas to fill the machine, and limits latency increase to a factor of 2.5x when workload is increased by 4x.

I. INTRODUCTION

A. Motivation

Many companies today like Uber and Netflix deploy large-scale web applications that consist of up to 100s or 1000s of different microservices [1]. These different microservices each run in separate containers and communicate with each other over a virtual network. While this approach allows for microservices to be reused across multiple applications, managing the communication among these microservices is a very complex effort. To deal with this complexity, companies rely on microservice orchestration services like Kubernetes that take in an application and cluster specification and handles the launching of these microservices [2]. Although Kubernetes handles the management of these microservices, it does not optimize their performance. With a large number of microservices it is increasingly difficult to analytically determine optimal values for these variables. As a result, users typically overprovision resources and use heuristic rules to determine placement, leading to inefficient use of resources [2, 3]. B-Optimal alleviates this issue by using Bayesian Optimization to automatically find optimal resource allocation of CPU Quota, Disk Bandwidth, Memory, and Network Bandwidth for different microservices. B-Optimal first randomly samples the input space and then considers neighbors around the best performing points, choosing the point with the highest expected improvement in application latency according to a Gaussian regression of the points evaluated so far. This automated end-to-end approach allows microservices to optimally utilize resources and makes the lives of system operators much easier.

B. Evaluation Metrics

We use two main applications and sample workloads for these applications to evaluate our optimizer. The goal of our optimizer is to minimize the latency, given a fixed or variable workload, for the following two applications in a timely manner.

1) MEAN Stack: The first application that we evaluate against is a sample ToDo application leveraging the MEAN Stack. The MEAN stack is a commonly used software bundle consisting of three services with distinct functionality: a NodeJS web server, a MongoDB cluster as the database service, and a HAProxy load balancer that routes traffic to different node application containers. This service stack aims to imitate the low volume web development applications that an individual may use to run his or her blog, personal website, or in our case, a ToDo list. The workload that we used for this application consists of requests to the frontend that traverse all services in the application.

2) Apartment Application: The second application more closely resembles corporate use, serving as an apartment finding tool, or a database where multiple users can query apartments that fit their specifications. The application exposes 4 endpoints, two relying on the base web page and sign-in portal, which keeps a database entry on MySQL for every user account. The third endpoint uses the PostgreSQL database to keep track of the apartments and their specifications. The other features a LogStash pipeline that connects the Postgres
database to an Elasticsearch cluster, which performs a computationally expensive search on the user’s query and pipes the data to running Kibana instances. We use this app for scaling and high workload benchmarks within our paper. Figure 1 shows a diagram of the apartment application and all of its various microservices. The workload that we used for this app consists of a mix of reads and writes.

II. PRIOR WORK

We consider various baselines and prior work that we compare B-Optimal against.

A. Static provisioning

The most naive baseline to compare against is a static provisioning of resources to each service. As mentioned previously this basic method is what is most commonly used by operators today. They simply provision a service pipeline such that each service shares a machine with a minimal number of other services, and all services on each machine take up an equal portion of all resource type [4]. For the rest of the paper, we will refer to this configuration as an “equally provisioned” configuration. Any further transferring of resources is done in an ad-hoc approach where operators manually investigate resource utilization and use that information to inform any allocation transfer decisions. This strategy results in the necessity for a large machine deployment, involving additional nodes being added to the cluster, which in turn, incurs additional costs. We simulate this approach by placing three services per machine and providing approximately thirty percent of the machine’s resources to each service (Fig 2.)

B. Greedy Transfer

In prior work, Yang et al. [5] discuss a heuristic based resource transfer approach, stemming from a bulk transfer of resources from one service to another based on service need. We generalize their scheme to a greedy algorithm of resource transfers between services. This greedy approach comprises of large transfers of resources from less utilized services to more utilized services, where utilization is measured during the administration of workload on a service pipeline’s endpoints. Put simply, the algorithm starts at a configuration with each service having an equal provision of resources. The algorithm then runs such that for each resource, the least utilized service donates half of its resource allocation to the most utilized service, the second least utilized service donates half of its resources to the second most utilized service, and so on. One iteration consists of all service pairs performing one transfer. Iterations are run until performance no longer improves, at which point the algorithm is terminated and the best configuration is returned. We poll the performance over time of this greedy algorithm, displaying the results in Figure 2.

The approach of utilization-based resource transfers indicates significant performance improvements over the base baseline. However, allocating resources to services with higher utilization bears several dangers, specifically, those concerning service bottlenecks. For example, one service that continually receives requests but is blocked in piping the results to another service may experience more utilization than the second service. To improve performance, however, one must allocate more resources to the service with lower utilization, to remove the cause of the blocking. As shown in [4], merely looking at utilization metrics to decide which service to allocate more resources to does not necessarily result in better application performance. Utilization does not capture the complex dependencies between the different services, and thus a better approach would be to make allocation decisions by measuring overall application performance as a metric instead of just utilization statistics.

C. Gradient Descent

Concerning more recent prior works in the field of resource allocation optimization, we consider Throttlebot, built by Chang et. al in 2017 [4]. Throttlebot provides a more restrictive heuristic that defines the transfer it makes between resources motivated by the principles of gradient descent. It estimates the impact certain resources has on the overall application performance by iteratively “throttling” each resource. It then uses those gradient estimates to guide resource transfer decisions for the different services. After many iterations of this gradient descent algorithm, a steady-state will be reached that has the optimal resource allocation for all the services. A more detailed description of one iteration of Throttlebot is described below:

1) For each service $s$, and for each resource type $r$, calculate sensitivity of application performance to changes of $r$ for $s$. This estimates a gradient for each $(s, r)$ tuple.
2) Sort the $(s, r)$ tuples by sensitivity in descending order.
3) For each pair of tuples: $(s_i, r_k)$ and $(s_m, r_n)$, if $s_i$ and $s_m$ are located on the same machine and $r_k$ and $r_n$ are the same resource, transfer $\delta$ resources from the lower sensitivity tuple to the higher sensitivity one.
4) Ensure that the same $(s, r)$ tuple does not receive new resources from more than one service in the same
iteration (this prevents resource starvation in minimally sensitive tuples).

5) Reduce step size $\delta$ by factor of 1.2

In the above steps, sensitivity of each service-resource tuple refers to the ratio of application performance decrease to resource decrease, the latter of which is a fixed stress amount determined within the pre-run configuration of the Throttlebot. The sensitivity calculations consists of the most time-consuming portions of Throttlebot. The large duration of these calculations is due to the necessity to run $S \times T$ iterations where $S$ represents the number of services within the pipeline and $T$ represents the number of resource types that the optimizer modifies. Per each of these $S \times T$ iterations, a performance benchmark must be run to determine how changing one resource type, belonging to one service, affects that application performance. For an application such as the MEAN Stack, this results in 12 sensitivity calculations, as the application contains 3 different services and optimizes over 4 different resource types. If each workload run takes an average of 10 seconds, the runtime of sensitivity calculations for just a single iteration is 120 seconds.

The $\delta$ reduction by 1.2 per iteration accounts for a normal practice within gradient descent, which underplays later transfers such that instead of jumping between various local minima, the algorithm converges smoothly into one minimum and is able to terminate after a certain amount of iterations. The termination condition comprises when an iteration no longer produces a significant performance gain when there are no feasible transfer conditions.

Overall, Throttlebot performs an iterative gradient descent after building a sensitivity prior using performance tests for each service-resource tuple. While Throttlebot eventually converges to an optimal resource configuration, the time it takes to reach this optimal point is the main drawback of this approach.

D. Bayesian Optimization

The primary prior work regarding Bayesian Optimization for resource configuration is CherryPick by Alipourfard et al. in 2017 [6]. Unlike Throttlebot, CherryPick is used to improve the performance of an application consisting of a single service, and thus is determining optimal resource allocations for that single microservice only, leading to a low-dimensional search space. Our work in B-Optimal is inspired by the Bayesian Optimizer used in Cherrypick, but makes some key improvements in the algorithm to deal with the high-dimensional search space that Throttlebot optimizes over. In the following section we describe how the Bayesian Optimizer works, and the improvements that B-Optimal makes over Cherrypick.

III. DESIGN OF B-OPTIMAL

Our primary motivation in developing our own resource allocation optimizer is to provide a speedup on the exhaustive search that Throttlebot performs within its gradient descent algorithm, developing B-Optimal to achieve lower short term and steady state latency than its gradient descent counterpart. As mentioned before, $S \times T$ iterations are performed per iteration of gradient descent, producing a situation where a majority of the runtime is spent towards calculating a prior, or a distribution from we can infer future evaluated points, for the gradient descent to perform. Thus, we use Bayesian Optimization to generate an accumulative prior such that all inputs combinations need not be exhaustively run before optimization occurs.

A. Bayesian Optimization

Bayesian Optimization is a type of iterative optimization and like gradient descent, is useful for when knowledge of the internals of the objective function is limited or unknown [7]. However, unlike gradient estimation, which is the technique employed by Throttlebot, Bayesian Optimization does not require a large number of samples as it imposes a prior on the objective function. Bayesian Optimization starts off with an initial prior of the true objective function $f$ and updates this prior through samples of $f(x)$ to incrementally get a better approximation of $f$. Thus, there are two primary components of Bayesian optimization: a surrogate model $f'$ which serves as a model of the true object function $f$, and an acquisition function that is computed from $f'$ and is used to determine which point to evaluate next.

1) Surrogate Model: One of the most popular choices for a surrogate model is a Gaussian Process, an example of a non-parametric model. Cherrypick uses Gaussian Processes as a prior, and we use the same for B-Optimal. By imposing a Gaussian prior, this makes the assumption that the final function is a sample from the Gaussian Process. Thus the objective function is determined by a mean function $u(x)$ and a covariance kernel function $k(x_1, x_2)$. The optimization first starts by defining a prior over the functions by initially randomly sampling points. Then on each iteration of the algorithm, more points are observed and used to update the Gaussian Process. Over time, the Gaussian Process becomes a better approximation of the objective function, and in conjunction with the acquisition function, can start proposing more optimal sampling points.

The authors of Cherrypick decide to go with a Gaussian Process because its inherent expression of uncertainty allows Bayesian Optimization to perform more quickly and organically than other optimization forms, where uncertainty must be artificially construed [8]. Linear and logistic regression, for example, take on a backward-oriented approach, where a function is fit according to minimizing a loss from the points calculated so far. Preventing overfitting is largely a task of art and intuition, where human judgment must come into play to determine how representative a sample is of future points that the model may encounter. This post-facto means of representing uncertainty also creates issues within deep learning or reinforcement learning models, which are often only able to express uncertainty through pre-determined regression patterns, producing issues that largely line up with those of regression techniques.
2) Acquisition Function: The acquisition function uses the surrogate model to propose future sampling points. In our case, these sampling points will be better resource configurations over time. Given an acquisition function $u(x)$, the point to be sampled at time $t$ is $x_t = \arg\max(u(x_t | D_{t-1}))$ where $D_{t-1} = \{(x_1, y_1), \ldots, (x_{t-1}, y_{t-1})\}$ are previously drawn samples from $f$ so far [7]. The acquisition function that CherryPick and B-optimal use is Expected Improvement [8]:

$$EI(x) = E(max(f(x) - f(x'), 0)$$

where $f(x')$ is the value of the best sample so far. Using the Gaussian Process as $f(x)$, the analytical expression of Expected Improvement is

$$EI(x) = (m - \mu(x))\Phi(Z) + \sigma(x)\phi(Z)$$

where $Z = \frac{m - \mu(x)}{\sigma(x)}$ and $\Phi$ and $\phi$ are the normal cumulative distribution function the standard normal probability density function respectively. Notice how expected improvement calculations are directly tied to the normal probability distribution function, suggesting that these calculations are affected cumulatively by uncertainty recalculations when new points are run. All in all, this contributes to the idea of a progressive prior function, one in which all new points calculated represent the prior so far.

B. Improvements over Prior Work

Here we describe two improvements made to the traditional Bayesian Optimization algorithm employed by CherryPick. Mainly, we increase the random exploration at the beginning, and we increase the number of points that we sample on each iteration.

1) Initial Random Exploration: Recall from the previous section that the initial Gaussian prior needs to be initialized by a starting point. CherryPick started from a balanced starting state, such as an equally provisioned configuration. While this provides a constant starting state for all applications, producing consistent results for multiple runs of the optimizer, such a starting state produces a uni-modal Gaussian Process, where the first point is perceived as the mean upon which the Normal Probability Distribution is built. Such a low dimensional process leads to a consistent but myopic outcome, where points often fixate in one local minimum. Compounded with an acquisition function that finds neighboring points in close proximity to previously calculated samples, the chosen samples rarely stray from a small fixed range. Such a uni-modal Gaussian Process often features high uncertainty estimates, as the process attempts to fit the entire range space based on samples that are located very close to one another.

We aim to improve this approach by a procedure of random search. We add a feature where the algorithm evaluates several quasi-random points before beginning the Gaussian-regression based optimization. We use the Sobol sequence as the basis of our random generation; the algorithm uses binary bit manipulation and a hole-finding heuristic to cover all the combinations over the entire binary space repeats. The purpose of performing random search before the regression phase is to produce samples in different portions of the Gaussian space, such that even if samples are not necessarily the best performing inputs, they offer future samples that may converge to various local minima. More specifically, these additional samples reduce the variance of each Gaussian curve produced in the range space when compared to a singular encapsulating Gaussian curve, producing a Gaussian Process dimensionality that more closely matches the true dimensionality of a multi-service interdependent environment, whilst also increasing accuracy of expected improvement estimates.

To reinforce this point, the random search period is likely more integral for the optimization of complex environments than for the optimization of simple environments due to the fact that such environments bear high dimensionality performance functions that require assessment at various local minima. As such, increasing the number of points sampled during the random search procedure will generally improve the performance at which B-Optimal converges, perhaps more so for complex environments than for simple environments. We perform experiments regarding the number of samples run for random search preceding the optimization and discuss the implications for both the MEAN App and the Apartment App in future sections.

2) Acquisition Function: CherryPick [6] uses an acquisition function that compares points uniformly sampled from a fixed range from either side of the mean. A narrow neighbor finding algorithm in combination with a fixed starting configuration with no pre-sampling produces a very small range of evaluated points, as well as a high uncertainty estimate of expected improvement. We instead develop a system that randomly samples from a range that is a percentage of the point around which we are finding neighbors. This is motivated by two primary principles. The first involves domain. Given the noise present within system performance calculations on large-scale clusters, we desire to find distinct input samples that differ from previously evaluated samples by more than a hundredth of a CPU Quota Percentage. As such, we base the range in which we can find new points upon the given inputs, allowing us to achieve notable changes between consecutive evaluation inputs. The second principle largely involves the limited coverage of the Gaussian prior. Even with adding a random search period before the optimization, we still desire a performant optimization application. As such, we only extend the random search iteration count to, at most, a multiple of ten. Uncertainty estimates for such a low number of samples will be high, since the number of samples is often the same level of magnitude as the count of the optimizing variables, and in turn, the dimensionality of the regression. As such, our choice of randomly sampling from a given range for the acquisition function prevents an over reliance on the high uncertainty gaussian curve produced within the regression phase; this contrasts with CherryPick’s uniformly sampled neighbor points which adhere strongly to the Gaussian curve.

On the spectrum between Bayesian Optimization and Random Search, our choice of randomly sampling over a large interval shifts the B-Optimal slightly towards the latter. By
adding this feature, we are in part extending the preliminary random search process such that it occurs in a smaller form between every iteration of optimization.

While we can also combat Gaussian uncertainty by increasing the count of neighbor points to assess before choosing the next evaluation point, we choose to maintain a neighbor count similar to that of CherryPick, which uses around 10 points, to limit the runtime overhead to the optimizer.

We benchmark various range intervals within the results section of our paper.

C. B-Optimal Algorithm

Putting all of the above information together, the algorithm for B-Optimal is as follows:

1) Randomly sample \( m \) starting resource configurations. These points initialize the starting Gaussian prior, GP.

2) For \( t = 1, 2, \ldots \) repeat
   a) For the top \( n \) best points seen so far, \( \{x_1, \ldots, x_n\} \), find the next sampling point \( x_t = \text{argmax}_x(u(x|D_{t-1})) \) where \( u \) is the acquisition function.
   b) Obtain a possibly noisy sample \( y_t = f(x_t) + \epsilon_t \)
   c) Add the sample to previous samples \( D_{t-1} = \{D_{t-1}, (x_t, y_t)\} \) and update the GP.

D. Comparison to Gradient Descent

As mentioned in a previous section, the main bottleneck to Throttlebot is the large number of performance evaluations (\( O(S \times T) \)) that need to run before the gradient descent transfers occur. Every Resource-Service sensitivity value is not even guaranteed to be used, since certain transfers don’t occur due to small sensitivity differences or due to the pruning stage which removes repeated receiver tuples. In future versions of Throttlebot, we are working with the NetSys lab to subet the combinations to evaluate both randomly and heuristically; however, the current working version of Throttlebot evaluates all Resource-Service Tuples exhaustively, and this search is very time-consuming.

Our goal for the design of B-Optimal was to limit the prior calculation overhead while simultaneously achieving a high coverage search of the input space. To fulfill the former point, we elected to proceed with Gaussian Regression, which features an adjustable prior dependent on new curve centers and iteratively changing variance values. To fulfill our goals of a high coverage search space, we designed a period of random exploration of the input space as well as an increased and randomized search for neighbors of the best performing points so far.

While our coverage is less than the coverage of a pure iterative gradient descent algorithm, which can continually transfer a resource in a certain direction without limitations from Gaussian restraints and neighbor finding, our algorithm resembles gradient descent in that the beginning of the algorithm comprises of big transfers, through blind random sampling of the input space, and then narrows down to smaller transfers, through a neighbor finding algorithm that limits evaluation to a closed range. However, B-Optimal exceeds Throttlebot in its ability to limit overhead per iteration. With a prior that is calculated only once per iteration and the use of regression instead of full application performance evaluation to gauge the expected improvement of neighboring points, B-Optimal is able to optimize the application for more iterations than Throttlebot does. With the Mean application for example, B-Optimal achieves about 50 iterations in an hour, compared to 7 iterations by Throttlebot.

We display and discuss the differences in B-Optimal’s runtime and performance within the results section of the paper.

IV. B-Optimal and Horizontal Scaling

Both the CherryPick \([6]\) and Throttlebot \([4]\) papers discuss applications designed for use by individuals or small companies who expect low amounts of traffic within their websites. Throttlebot sends out 350 requests across 50 threads to benchmark its MEAN Stack application. While this provides a significant workload for the type of application that Throttlebot aims to benchmark, we look to optimize for workloads on a level of magnitude higher, more in line with the reported 40,000 concurrent search requests received by Google Web Servers.

To increase the workload-incurring abilities of a base application, we look to horizontal scaling. The unpublished AutoTune Paper by UC Berkeley’s NetSys Lab \([9]\) discusses the idea of horizontal scaling as a means of combating highly variable workloads, specifically ones that momentarily scale by a factor of ten. We helped develop this architecture with NetSys and made various observations concerning the choice of deployment tool. The paper uses a Kubernetes cluster to host its running services, using Kubernetes inbuilt scaling functionality to increase a replica count for a service if that service exceeded a certain utilization threshold. When workload increased, the Kubernetes deployment more or less maintained performance, if the utilization threshold for scaling was set sufficiently low, resulting in a larger amount of running service replicas to adjust for the higher workload.

We align with the AutoTune paper in our goals of adapting low workload applications to higher workloads, using horizontal scaling to limit performance drops through workload increases. However, AutoTune’s scaling tests also demonstrate the dangers of unchecked horizontal scaling. When setting the scaling utilization threshold to 50%, such that replicas would be added if any service exceeded 50 percent utilization, within the Kubernetes scaling logic, pod count jumped to 30 replicas across all running services within the Mean stack. The authors discuss using a machine autoscaler, that scaled up machine count, to keep up with the demands of a quickly replicating application, increasing machine count from two to ten m4.large nodes by the time scaling had reached a steady state. As mentioned with respect to our baseline benchmarks, utilization often paints a warped photo about the application in question and can often produce excessive scaling when it is
not necessary, as applications that are scaled may still be bottlenecked by the next service within the pipeline. Within our horizontal scaling application, we desire to pair performance oriented resource allocation principles with horizontal scaling, such that replicas are only added to the application when they improve performance.

In addition, we align with more recent trends in cost management within AWS and Kubernetes, where a cost maximum is allowed for a running application, based on the total costs that all machines within that application require. Instead of providing a strict cost maximum for pod scaling, we instead add a system of a pod penalty, or an additive amount applied to application performance per pod, such that the resulting replica configuration balances performance for replica count with an adjustable parameter.

The ease of use of B-Optimal renders the addition of optimization parameters a simple task. We simply add variables representing the replica count of each service to be scaled. Notably, we do not scale the load balancer, as the load balancer features throughout in the order of Gb/s while requests are normally less than 1 Kb. With the number of concurrent requests we are sending, usually a multiple of \(10^3\) multiplied by the number of endpoints, we do not exceed the throughput limit of the HAProxy balancer used within the tested applications. B-Optimal proceeds to optimize over replica counts in addition to resource configurations and a penalty for pod count is applied at the end to the evaluated performance.

We provide benchmarks for our scaled version of B-Optimal optimization within the results section of our paper.

V. IMPLEMENTATION

A. Application Runner

The application resides on a Quilt cluster, a third party tool that serves as a Kubernetes-like deployment environment where each container runs a Docker image. We chose Quilt over Kubernetes as the application is optimized for low overhead management of smaller applications, such as the applications we were testing within this study. The application runner first performs an SSH action into the machines on which containers reside, setting each container provision to match with the provided resource configuration.

We elected to use four resource types for our optimization: CPU-Quota, Memory Allocation, Disk bandwidth, and Network Bandwidth. CPU-Quota signified the percentage of CPU cycles that were allocated to a service. The first three resource types were set using the linux cgroups function, which applied changes to the entire container running a service. The Network resource type was set using linux’s “tc” application.

The workloads described in the previous section were administered using Apache’s ab benchmarking tool, which doubled as a performance measuring tool for the application. A certain amount of requests were sent out to all exposed endpoints of an application, with a configurable concurrency measure. We used 500 requests per endpoint for apartment-app and 350 for Mean Stack, using request numbers from default Throttlebot configurations. The performance results were written to output files within the machines that hosted each container. Latency information was collected from these files and added up amongst all endpoints. The mapping of resource configuration to performance result is returned to the database, concluding the iteration.

For performance measures, we simply run a process that polls a file that contains the best results so far every 30 seconds.

B. Scaling Implementation

As mentioned in the previous section, we add variables within our configuration to represent replica scaling for services. We limit replica scaling to 10 pods, such that the finalized replica count more likely fits into the starting machine configuration. Per iteration, we redeploy our container deployment tool with the necessary amount of replicas and run performance tests with the provided resource configuration.

VI. RESULTS AND EVALUATION

A. Baseline and Optimizer Results

In this section, we provide comparisons between baselines and the optimizer results. We also compare optimizer techniques, displaying performance over time graphs of Gradient Descent and Bayesian Optimization techniques.

Generally, the baselines in Figure 3 align with our motivations to shift from a utilization based model to a performance based model for resource allocation; shifting heavily across lines of utilization, as was performed within the Transfer Baseline test, demonstrates only two improvements in performance despite five transfers having occurred within the duration of the graph. On the other hand, even the BayOpt baseline,
Throttlebot’s gradient descent performance is compared to the Greedy Transfer Approach to resource optimization for the Apartment App. Here Throttlebot begins at an equally provisioned starting configuration. Our recreation of the CherryPick optimizer, which consists of performance based heuristics for allocation shifts, shows potential for improvement, as no optimizations are performed after steady state is reached at approximately the 15% mark of the time axis of the graph. While the baseline chart does not sufficiently isolate variables to explain BayOpt Baseline’s ceased optimization, the BayOpt baseline shows signs of a conservative acquisition function, where the range of points evaluated is very low, differing by one or two percentage points from the best point evaluated so far.

Within Figure 4, we compare Throttlebot to the Transfer Benchmark, as both are predicated on resource transfers between services. In our benchmark, Throttlebot performs 5 transfers using its gradient descent algorithm, 4 of which bring improvements to the lowest latency witnessed so far. Through benchmarked performance gains, Throttlebot demonstrates its decaying transfer quota; while the first drop is the most significant, the subsequent drops demonstrate smaller resource allocation shifts that produce smaller, albeit still significant, performance gains. Throttlebot’s performance shifts largely occur in the same direction, as sensitivity rankings are mostly preserved even as the range of these rankings grow closer. This matches up with the Transfer baseline, where transfer directions are fixed into static allocation calculations at the beginning of the experiment. Despite fixed transfer directions in both algorithms, the performance heuristic of Throttlebot shows more consistent performance improvement during the duration of the graph, as well as a steady state point 45% lower than that of the greedy algorithm.

Within Figure 5, the Bayesian Optimization Baseline keeps up with B-Optimal for the first half of the experiment, indicating strong performance of input points located around the initial equal provisioning. At 500 seconds, B-Optimal is still within its random search and finds an input configuration that improves performance to the level of the Baseline, which is undergoing optimization at that point. At that point, however, the two algorithms are in different portions of the input space, with B-Optimal concerning high CPU-Quota values, high memory, and low disk, while the Baseline considers middling CPU-Quota values, high memory, and high disk values for all its services. After about 2000 seconds, B-Optimal begins optimization, entering the space of high CPU-Quota, high memory, middling disk, and low network, an input space based on the previous best run within the random search period. Here, B-Optimal outperforms the Bayesian Optimization Baseline, settling at a steady state value approximately 35% lower than the Baseline.

Within Figure 6, equally provisioned configurations provide a relatively slow start for both optimizers, and both start at latencies at approximately twice of their steady state. For the apartment app, during the period of random exploration before 500 seconds, B-Optimal enters an input space of high CPU Quota and high memory just by random sampling, reaching a latency value of approximately 30,000 more quickly than Throttlebot, which must perform consecutive transfers to get to that portion of the input space (Fig 7). In fact, Throttlebot does not reach that latency mark until 2500 seconds. Eventually, both algorithms reach a similar steady state around 2600
Fig. 7. B-Optimal: B-Optimal resource allocation percentages at 1400 seconds captured during the optimization phase; Values already lie close to the steady state configuration, with high CPU Quota within frontend applications and high Memory for database applications.

Throttlebot: Throttlebot resource allocation at 1400 seconds after 2 transfer phases; Notice that values still lie close the equally provisioned configuration, where most service-resource tuples begin at 33%.

Fig. 8. Steady state configurations for B-Optimal and Throttlebot. Configurations are relatively similar, with the highest variance occurring within network allocations.

Fig. 9. Evaluation of B-Optimal on the MEAN application with varying random exploration iterations/samples. In varying the random exploration iteration count, we initially expected that increasing random iteration count for higher dimensional applications, such as the Apartment application, would bring larger proportional improvements than doing the same for lower dimensional applications due to the increased importance of covering large input spaces. While the range of steady state values seems to align with this claim, as the Apartment app sports a range of approximately 18% of the highest value across all steady values compared to Mean-App’s 8%, the ranking of iteration count at steady state is unexpected. For the Apartment app, at the point of 800 seconds, “2 iteration” and “10 iteration” have begun optimizing while “15 iteration” is still in the phase of random exploration. “2 iteration” outperforms “15 iteration” by this point, with “10 iteration” shown in Figures 9 and 10 respectively. In varying the random exploration iteration count, we initially expected that increasing random iteration count for higher dimensional applications, such as the Apartment application, would bring larger proportional improvements than doing the same for lower dimensional applications due to the increased importance of covering large input spaces. While the range of steady state values seems to align with this claim, as the Apartment app sports a range of approximately 18% of the highest value across all steady values compared to Mean-App’s 8%, the ranking of iteration count at steady state is unexpected. For the Apartment app, at the point of 800 seconds, “2 iteration” and “10 iteration” have begun optimizing while “15 iteration” is still in the phase of random exploration. “2 iteration” outperforms “15 iteration” by this point, with “10 iteration”...
settling at the lowest steady state latency value. The graph demonstrates the possibility of the occurrence that a long exploration period produces worse performance values in the short term, even with a high dimension application with a large input space. The possibility of high performance with low random samples pushes the point that choosing a middling random sample count and running the count over multiple seeds may be an approach that produces the best steady state within a short-term span. For long term experiments, a higher random iteration count is still recommended, since as long as the optimizer reaches the optimization phase and concludes the random exploration phase, the optimizer will perform as well, if not better, than a run with fewer random iterations.

The second parameter that we vary is the range of neighbors that we consider during the bayesian optimization phase. The results for this experiment are shown in Figure 11. The largest interval from which neighbors can be found produces a steady state value that is 30% better than the smallest interval. This largely manifests as the “1x interval” demonstrating the largest improvements once the optimization phase starts at 400 seconds. Neighbors can be found further from the best performing point, producing points with potentially higher expected and actual improvement from the perceived mean of the Gaussian distribution. The Gaussian Process regression often inaccurately assumes an isolated best performing point to be at a local minima or mean value of a Gaussian distribution, which prevents small-interval neighbor-finding heuristics to escape the previously evaluated point and find the actual local minimum around that portion of the input space.

C. Bootstrapping

In this section, we evaluate a hybrid optimizer, that bootstraps gradient descent with the best configuration from B-Optimal’s random search. We compare this performance to a full run of just B-Optimal. The results are show in Figure 12.

The primary runtime shortcoming of Throttlebot is its necessity to start from an equally provisioned resource configuration and approach the optimal configuration using small, iterative steps. Here, we do away with this short coming by placing Throttlebot in the portion of the input space in which B-Optimal enters its optimization phase. Then, gradient descent outperforms the random neighbor finding acquisition function in its convergence time due to the heuristic, uni-directional nature of resource transfers to more sensitive resources. The bootstrapped gradient search consists of the fastest convergence out of all the configurations tried in this paper and it is worth exploring this system under additional workloads and seeds in future work.

D. Horizontal Scaling

Finally, in this section, we compare convergence values of our horizontal scaling implementation, and compare per-
Fig. 13. We display non-optimized static vs scaled performance under different workload settings. Static refers to fixed and singular replica counts, where only one replica per service is deployed. Scaled refers to service deployments that are replicated. “Non-Opt” refers to unoptimized replica and resource configurations, in which equally provisioned configurations are used, while “Opt” refers to those produced by B-Optimal. The default workload administered is 250 requests, and all increased workloads are relative to that baseline. Notably, the Non-opt static deployment did not produce a measurement for 20x workload, as the ab benchmarking tool consistently timed out due to missed responses.

Figure 13 shows the results of the optimizer with various configurations. Static refers to running the workload using a singular replica per service, with the exception of Node, and without increasing the number of replicas. The replica count for static is 3 for the Node service and 1 for everything else. In the scaled configuration, we allow more pods/replicas to be created. Opt uses configuration information from B-Optimal, while Non-Opt refers to unoptimized replica and resource allocations, in which we used equally provisioned configurations.

Scaled and static performances are similar at the 1x workload, as overhead to distribute the workload amongst various replicas cancels out any concurrency increases from the replica count. When workload scales up to 5x and 20x, the replicated deployments outperform the static deployments as expected.

Under higher workloads, static latency grows significantly worse, scaling linearly with the workload increase from 5x to 20x even with optimization. In fact, the non-optimized static version under 20x workload times out, due to missing responses to requests.

Within Figure 13, B-Optimal improves the 5x Workload latency of the scaled deployment by 52% from the 5x Workload Non-Optimized setting. In addition, an optimized scaled configuration under a 20x workload produces a latency that only exceeds that of an optimized scaled 5x configuration by a factor of 2.5.

Overall, the scaled optimizer shows promise in curbing performance drops with increased workloads, as well as improving performance over an equally provisioned replica configuration.

<table>
<thead>
<tr>
<th>Service</th>
<th>Replica Count (5x)</th>
<th>Replica Count (20x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>ElasticSearch</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>Logstash</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Mysql</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Postgres</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Fig. 14. Replica count for optimized configurations under 5x and 20x Workloads

The optimized experiments are run where a fixed penalty is applied per pod deployed, so the replica configuration of these data points reflects this cost function. The total replica counts for 5x workload and 20x workload were 24 and 32, respectively, and the total replica capacity for the cluster was 40, indicating that the cluster was not blindly overprovisioned with replicas and that the final configuration adhered to the replica penalty.

VII. DISCUSSION

The baseline performance suggests the intrinsic benefits of using Bayesian Optimization as a resource allocation optimizer. Our mock of CherryPick’s optimizer, which serves as the Bayesian baseline, outperforms the static provisioning configuration as well as a greedy transfer approach. However the Bayesian Optimization baseline falls short in the design principles we aimed to enact within B-Optimal. Specifically, the optimizer does not sufficiently prune the input space with a period of random sampling. Instead, it immediately narrows its efforts close to an equally provisioned input space. Points near this space are sampled during optimization, but are done so in a manner that conforms strictly to an under-defined Bayesian regression and within a range that limits discovery of input points that deviate significantly from the mean. This produces poor long-term performance, where within our experiments the baseline Bayesian Optimization ceased optimizing after 450 seconds and produced worse performance than Throttlebot from approximately 2500 seconds onwards.

In designing B-Optimal, we looked to experiments varying the duration of the random sampling period and experiments varying neighbor-finding intervals, or the range considered by the acquisition function. We found that short term steady state was seed oriented, based on how quickly optimal random samples were given to the optimizer during the random exploration, and how quickly it subsequently began the optimization phase. While a middling range of 10 iterations produced the best results in the short term, we increased the iteration count to 15 for our longer experiments, given that we knew that the optimizer would perform as well as 10 iterations if not better within the long term. For neighbor intervals, we saw that larger neighbor intervals produced better steady states even within the short term, since input points were discovered that were closer to local maxima than points that were already evaluated. We tuned B-Optimal according to these findings.
Comparing a tuned B-Optimal to Throttlebot, we saw that B-Optimal outperformed the latter by 25% and 66% in the Apartment Application and MEAN stack, respectively, after 1000 seconds and converged to a steady state of 3-10% better than Throttlebot after 45 minutes.

Our Bayesian Optimization implementation and tuning maintained better short term performance than Throttlebot, whilst not sacrificing long term steady state, as was witnessed within CherryPick’s high latency steady state. While we fulfilled our goals of better short term and long term performance than Throttlebot, we must still confirm that Bayesian Optimization makes a practical choice over Throttlebot despite its seeding variability. In other words, Bayesian Optimization’s short term performance, where it most strongly outperforms Throttlebot, is dependent on the random seed, and therefore, the random sample count. In future examination, a means of dynamically determining the optimal random exploration count, based on an inferred steady state, or a means of sampling from a normal distribution where samples gather around optimal inputs can be tested and implemented. We will revisit this idea in the future work section. These developments may also improve the performance of the bootstrapped gradient descent, which constitutes our fastest converging algorithm with a steady state value below 30,000 ms.

Concerning the horizontal scaling version of B-Optimal, the optimizer’s scaling mode produces better performance with replicas than without replicas, as expected for higher workloads. The optimizer also improves scaled performance compared to an equal-provisioning configuration and limits scaled performance drops when workload increases as was also seen within AutoTune’s horizontal scaling experiments.

VIII. Future Work

In revisiting the B-Optimal and Throttlebot resource allocations in the baseline benchmark, we see that B-Optimal finds the optimal portion of the input space earlier in the experiment than Throttlebot. However, using a random seed for the random search phase renders B-Optimal’s short term advantage over Throttlebot highly variable. To combat this, we return to our principle of tuning our selection of inputs based on performance-based heuristics. Two potential approaches can further narrow down the inputs B-Optimal considers before the optimization phase. Firstly, Throttlebot’s gradient descent sensitivity data provides a performance based estimate of the gradient of the entire application, elucidating the most-impacted resource-service tuples. Therefore, we can consider inputs where the most-impacted tuples hold more resources than less-impacted tuples that share their machines within the random exploration phase. The initial allocation can begin as a proportion equivalent to the ratio of the sensitivity estimates within one machine. To avoid over defining this heuristic, we can sample from a normal distribution, where values closest to the mean, or the proportional provisions, are most common but not omnipresent. Secondly, we can perform a similar task by performing Principal Component Analysis on the input space after various random samples and sampling from a normal distribution based around an initial provision proportional to the rankings of the dimensions. Here, we would sample randomly from the input space, perform the PCA algorithm, and then refine the remaining random samples from the created distribution.

Overall, results suggest that a mix of random exploration and heuristic based optimization reaches a competitive steady state in the quickest manner. Tuning the random exploration using a mix of random sampling and dimensional analysis could further speed up the convergence of such an optimizer, whilst limiting hits to steady state latency.

IX. Acknowledgements

We would like to thank the UC Berkeley NetSys Lab for the inspiration for this project.

REFERENCES