Heterogeneous Resource Allocation for Efficient Cluster Management

Zachary Ankenman (zankenma@andrew.cmu.edu), Benjamin Berg (bsberg@cs.cmu.edu)

Motivation

Recent trends in datacenter design towards the use of warehouse-scale computers [5] have forced computer architects to reconsider many classical architecture questions. These questions center around how to allocate resources such as CPU cores, cache space, memory bandwidth, and network bandwidth to various jobs in order to complete each job quickly. The rollout of warehouse-scale computers has required companies to invest in custom built solutions to handle the challenges of operating at this new scale and processing new kinds of workloads[5, 8]. The central challenge of resource allocation in a warehouse-scale computer is that the system must run several vastly different *types* of jobs, each of which can respond differently to being allocated additional resources. Thus, it is difficult to devise a single resource allocation policy which will work well for all jobs.

Early warehouse-scale cluster management systems tried to solve this problem by relying on *users* to submit *reservations* for system resources, thus specifying what they consider to be the ideal allocation for their job. However, analysis has shown these reservation-centric systems to be highly inefficient [5]. In practice, these systems run at utilization levels below 20% [4]. In response to this inefficieny, systems researchers have focused on developing *performance-centric* systems which aim to improve job and system performance by automating resource allocation decisions [6, 4]. These systems aim to reduce job latencies by building detailed performance profiles for each job, and then using these profiles to make good allocation decisions for each job. This work has shown how to effectively profile jobs, but does not make any theoretical guarantees about the algorithms used to actually allocate resources to jobs. In fact, recent theoretical work from the performance modeling community suggests that, when jobs follow different performance profiles, allocating resources to minimize average job latency is difficult [2]. Hence, this paper aims to improve warehouse-scale computer performance by describing *resource allocation policies* which allocate resources to jobs in order to minimize job latency.

The Problem

We consider the problem of minimizing job latency in a warehouse-scale computer tasked with completing a stream of incoming jobs. While many system designers claim to optimize for job throughput, we know warehouse-scale systems are never fully utilized. Thus, these systems actually function as *open systems* where throughput is simply defined by the arrival rate of jobs. Instead, what designers actually want is low mean job latency, which we refer to as *mean response time*. Designers also want to understand mean response time as a function of system load in order to enable them to run at higher load. This is often

phrased as the goal of "increasing utilization". A good resource allocation policy will not only lower mean response times under current system conditions, but produce a flat curve for mean response time as a function of system load, allowing the system to be run at higher utilization.

There are two main tradeoffs which make this problem difficult.

Resource Use is Inefficient

Allocating additional resources to a job will generally decrease a job's running time. For example, allocating additional CPU cores to a job will increase the number of operations that job can do in parallel. However, it is a well-known property that will recieve a *sublinear* speedup when running on additonal cores – a job will run less than k times faster on k cores than it would on a single core. This phenomenon of sublinear performance increases is known to apply to several system resources (e.g. cache space [1]). Hence, any resource allocation policy must balance the tradeoff between running an individual job quickly and using system resources inefficiently. We want each job to run quickly, but if system efficiency is too low the system could become unstable.

Maximizing Throughput is Insufficient

Previously, [2] showed that, if all jobs receive the same sublinear speedup from resource allocation, the resource allocation policy which maximizes throughput at all times will also minimize mean response time. However, we quickly see that this is not the case when jobs receive different speedups. Consider the problem of determining how many CPU cores to allocate to two different jobs, one of which is highly parallelizable and one of which is not. Throughput will be maximized by allocating nearly all the cores to the highly parallelizable job. However, this makes it very likely that the highly parallelizable job will be completed first, leaving just the less parallelizable job in the system. This less parallelizable job will then make highly inefficient use of the cores available to it. A better strategy may have been to allocate fewer cores to the parallelizable job initially, increasing the chance that the less parallelizable job is finished first and allowing the highly parallelizable job to recieve an even larger speedup in the future.

Hence, an allocation policy must not only consider how to allocate resources to jobs at the current moment in time, but how resources will be allocated to jobs in the future. In this paper, we consider an allocation policy which must allocate a single resource to a stream of incoming jobs. To address this problem, we first develop a theoretical model of resource allocation for *heterogeneous jobs* which can receive different speedups from the same resource allocation. We provide a bound on the performance of a myopic *Greedy* allocation policy versus the optimal policy, and then we evaluate this bound via simulation.

Our Model

Recent research [2] has described a model for job performance as a function of CPU core allocation and analyzed various core allocation policies. Hence, we will assume, without loss of generality, that our allocation policies are tasked with allocating CPU cores.

We assume that jobs arrive into a system composed of n cores according to a Poisson Process with rate Λ jobs/second where

$$\Lambda \equiv \lambda n$$

for some λ. The random variable X denotes the *size* of a job. We think of X as the inherent work associated with a job. Any job can be run on any subset of cores. If a job of size X is allocated k cores, the job will run in time X_k, where

$$X_k = \frac{X}{s(k)}$$

and $s(k) \leq k$ is called the *speedup factor*. The function $s : \mathbb{Z} \to \mathbb{R}_{\geq 1}$ is assumed to be Amdahl's Law with some parallelizable fraction p, and is thus *sublinear*, *non-decreasing* and *concave* [7].

As jobs arrive, each job is allocated some subset of the cores, and this allocation can be changed at any time. At any moment in time, there may be several jobs running on a particular core. We assume each core schedules jobs according to the *Processor Sharing* (PS) scheduling discipline, which is common for processors in multicore machines.

All cores are allocated according to an *allocation policy*. An allocation policy defines, at every moment in time, which particular cores each job runs on. Cores are assumed to be homogenous, and thus any job is capable of running on any core.

We define the random variable T to be the *response time* of a job – the time between when a job arrives and when it is completed. Our goal is to find and analyze allocation policies that minimize the mean response time, $\mathbb{E}[T]$, across all jobs. Clearly, $\mathbb{E}[T]$ depends on the allocation policy, the speedup function, s, the arrival rate of jobs into the system, Λ, the mean job size, $\mathbb{E}[X]$, and the number of cores, n. We define the average system load, ρ, to be

$$\rho \equiv \frac{\Lambda \mathbb{E}[X]}{n}.$$

We assume both ρ and s are constant over time.

To model the heterogeneity of jobs with respect to speedup function, we allow jobs to belong to one of two *classes*. Class I jobs arrive with rate $\Lambda_1 = \lambda_1 n$ and all follow the speedup function $s_1(k)$, which is Amdahl's law with parameter p_1. Class II jobs arrive with rate $\Lambda_2 = \lambda_2 n$ and follow the speedup function $s_2(k)$, which is Amdahl's law with parameter p_2. Without loss of generality, we assume that $p_1 \geq p_2$. Note that, via Poisson splitting and merging,

$$\Lambda = \Lambda_1 + \Lambda_2 = \lambda n = (\lambda_1 + \lambda_2) n.$$

We will assume that jobs from both classes follow a single size distribution, X.

For the purposes of tractability, we will assume that X follows an exponential distribution with rate μ. For simplicity, we assume $\mu = 1$ in this paper. Hence, since jobs are exponential and arrive according to a Poisson process, we can model our system via a markov chain, where each state (i, j) denotes that there are i class I jobs and j class II jobs in the system. The exact rate of transitions due to jobs leaving the system will vary with the particular allocation policy being used.

We will only consider cases where the system is *stable*, that is, where $\rho < 1$. Hence, since the mean job size is fixed, we will restict the total arrival rate of jobs to be $\Lambda < n$.

Theoretical Results

Our goal in this section is to evaluate the performance of a myopic allocation policy in comparison to the optimal policy. We first define a myopic policy, Greedy, which is similar to the allocation policies used in practice. We then provide a bound on the difference in mean response time between Greedy and the optimal policy, emphOPT. Finally, we outline a path towards improving this bound via simulation.

The Greedy Policy

We define the myopic policy, *Greedy*, to be the policy which maximizes throughput in every state (i, j), similar to the allocation strategy used in [4]. As stated in [2], when jobs follow a single speedup function, throughput is maximized by dividing the cores evenly across all jobs. Hence, throughput is can be maximized in any state (i, j) by finding $k^*(i, j)$ such that

$$k^*(i,j) = \arg\max_k \quad i \cdot s_1\left(\frac{k}{i}\right) + j \cdot s_2\left(\frac{n-k}{j}\right).$$

That is, given that k cores will be allocated to class I jobs and $n-k$ cores will be allocated to class II jobs, throughput is maximized by dividing each set of cores evenly amongst its respective set of jobs. To maximize throughput, the Greedy policy must simply choose the correct value of k. If multiple throughput maximizing values of k exist, we define $k^*(i, j)$ to be the lowest such value. This was shown to produce the best throughput maximizing policy in [2].

A Bound on Greedy vs. OPT

We now wish to bound the difference between Greedy and the optimal policy, OPT. To do this, we first define an allocation policy which provides an upper bound on the mean response time under Greedy. We then imagine an idealized allocation policy which performs

even better than OPT. By bounding the difference between these two bounding policies, we obtain a bound on the difference between Greedy and OPT.

All-Class-II: An Upper Bound on Greedy

To bound the performance of Greedy, we consider a policy which assumes that all jobs are of Class II. In every state (i, j), the policy *All-Class-II* chooses $k_2(i, j)$ such that

$$i \cdot s_1\left(\frac{k_2(i,j)}{i}\right) + j \cdot s_2\left(\frac{n - k_2(i,j)}{j}\right) = (i+j) \cdot s_2\left(\frac{n}{i+j}\right). \tag{1}$$

That is, All-Class-II obtains the same throughput as that of the throughput maximizing policy *if all jobs were of Class II*. Note that $k_2(i, j)$ must exist because $(i + j) \cdot s_2\left(\frac{n}{i+j}\right)$ is less than the maximal throughput in state (i, j), $j \cdot s_2\left(\frac{n}{j}\right) \leq (i + j) \cdot s_2\left(\frac{n}{i+j}\right)$, and the left hand side of (1) is continuous.

Finally we will make use of the following lemma:

Lemma 1. *For any n, Λ_1, and Λ_2, the mean response time under Greedy, $\mathbb{E}[T]_{Greedy}$ is less than the mean response time under All-Class-II, $\mathbb{E}[T]_2$.*

Proof. This follows from a straightforward application of precedence relations and aggregation on Markov chains as outlined in [3]. First, we note that the Markov chain unser the All-Class-II can be aggregated into a 1-dimensional chain that tracks the total number of jobs in the system. Then, we examine the 2-D Markov chain under Greedy. By transforming some transitions to the states $(i-1, j)$ and $(i, j-1)$ into self loops, the total throughput in any state (i, j) can be made to match that of All-Class-II. The resulting chain can then also be aggregated into the same 1-dimensional chain as the All-Class-II chain, and it will have the same exact transitions. Hence, by making the Greedy policy worse, we can attain the same performance as All-Class-II and $\mathbb{E}[T]_{Greedy} \leq \mathbb{E}[T]_2$ as desired. □

All-Class-I: A Lower Bound on OPT

Similarly, we wish to obtain a chain which provides a lower bound on the performance of OPT. To describe such a policy, we will relax the rules of our model. Specifically, we imagine an *idealized* policy that can run every job as if it was a Class I job. In this case, we know from [2] that the best idealized policy evenly divides all n cores among the $(i+j)$ jobs, maximizing throughput in every state. We refer to this idealized policy as the *All-Class-I* policy.

We now prove that the mean response time under this idealized policy is a lower bound on the mean response time under OPT.

Lemma 2. *For any n, Λ_1, and Λ_2, the mean response time under All-Class-I, $\mathbb{E}[T]_1$ is less than the mean response time under OPT, $\mathbb{E}[T]_{OPT}$.*

Proof. We prove this lemma by contradiction. Consider the behavior of OPT and the All-Class-I policy in any state (i, j). Because All-Class-I assumes every job is of class I *and* maximizes throughput, its throughput in this state is higher that that of OPT, which must deal with having j "slower" class II jobs. Hence, the All-Class-I policy could be transformed to achieve the same throughput as OPT in every state, and the resulting allocation policy would be a valid way to process a stream of all class I jobs. However, we know that the All-Class-I policy was optimal in the case where all jobs were of class I, so this transformation must have *increased* mean response time under All-Class-I. Hence, the mean response time under the idealized All-Class-I policy is lower than that of OPT. □

Finishing the Bound

We have now established that

$$\mathbb{E}[T]_1 \leq \mathbb{E}[T]_{OPT} \leq \mathbb{E}[T]_{Greedy} \leq \mathbb{E}[T]_2 \tag{2}$$

Hence, one way to bound the difference between Greedy and OPT is to bound the difference between All-Class-I and All-Class-II. This leads us to Theorem 1, which will provide a bound for Greedy vs. OPT by an easy corollary.

First, we must prove a Lemma used in obtaining our bound. Let

$$d_x(i) := i \cdot s_x\left(\frac{n}{i}\right)$$

denote the throughput of the All-Class-x policy when there are i jobs in the system.

Lemma 3. *For any $c \geq 1$, if*

$$\frac{s_1(k)}{s_2(k)} \leq c \qquad \forall k$$

then

$$\frac{d_2^{-1}(k)}{d_1^{-1}(k)} \leq c \qquad \forall k \leq n.$$

Proof. Recall that s_1 and s_2 are both instances of Amdahl's law with parallel fractions p_1 and p_2 respectively. Since $\frac{s_1(k)}{s_2(k)} \leq c \qquad \forall k$, we know that

$$p_1 \leq \frac{c - 1 + p_2}{c}$$

and hence

$$\begin{aligned}\frac{d_2^{-1}(k)}{d_1^{-1}(k)} &\leq \frac{cn + k - (c + p_2)k}{n - p_2 k} \\ &= c + \frac{k(1 - p_2)(1 - c)}{n - p_2 k} \\ &\leq c\end{aligned}$$

where the last step holds because $c \geq 1$ and $k \leq n$. Thus, the claim holds.

□

Theorem 1. *For any, λ_1 and λ_2 such that $\Lambda \leq n$*

$$\frac{s_1(k)}{s_2(k)} \leq c \qquad \forall k$$

then

$$\lim_{n\to\infty} \frac{\mathbb{E}[T]_2}{\mathbb{E}[T]_1} \leq c$$

Proof. To prove this statement, we make use of the analysis in Theorem 4.2 of [2]. This tells us that

$$\lim_{n\to\infty} \mathbb{E}[T]_2 \leq \lim_{n\to\infty} \frac{d_2^{-1}(\Lambda)}{\Lambda}$$

and

$$\lim_{n\to\infty} \mathbb{E}[T]_1 \geq \lim_{n\to\infty} \frac{d_1^{-1}(\Lambda)}{\Lambda}.$$

Furthermore, we know that these limits both exist, so we can conclude that

$$\lim_{n\to\infty} \frac{\mathbb{E}[T]_2}{\mathbb{E}[T]_1} \leq \frac{d_2^{-1}(\Lambda)}{d_1^{-1}(\Lambda)} \leq c$$

by Lemma 3, as desired. □

We have thus proven that, given two classes of jobs whose speedup functions are within a factor of c, All-Class-II becomes a c-approximation of All-Class-I as the size of the system becomes large. This yield the following corollary.

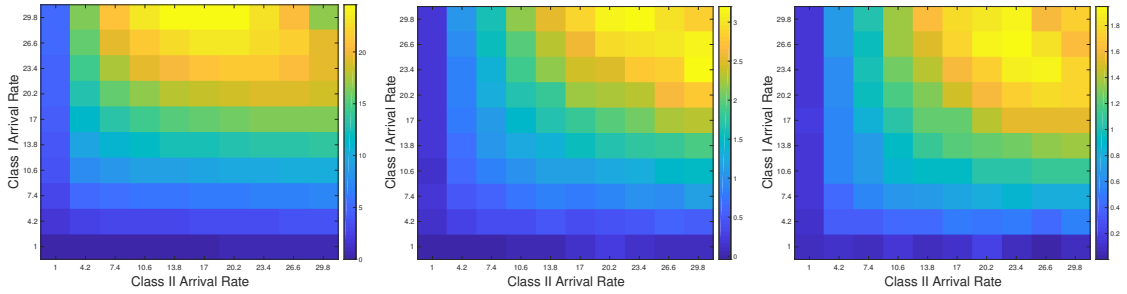
Corollary 1. *Given two classes of jobs whose speedup functions are within a factor of c,*

$$\lim_{n\to\infty} \frac{\mathbb{E}[T]_{Greedy}}{\mathbb{E}[T]_{OPT}} \leq c$$

Proof. This follows directly from Theorem 1 and Equation (2). □

Discussion

That may appear to be a lot of work to only get a decent bound. An astute reader will note that our proof did not rely on the properties of Greedy at all. In fact, this bound holds for any two policies which are better than All-Class-II and worse than All-Class-I. We describe the set of such policies as the set of *admissible policies*. Theorem 1 implies that *any two* admissible policies perform within a factor of c given that their speedup functions are within a factor of c.



(a) [0.9, 0.1] (b) [0.75, 0.5] (c) [0.5, 0.25]

Figure 1: Greedy vs EQUI

This bound can be viewed as more of a jumping off point. For example, if one could show that the Greedy policy alway outperforms another admissible policy by a factor of d, then this bound becomes a $\frac{c}{d}$ bound. We did not have time to specifically investigate such a bound theoretically, and thus addressed this question via simulation in the next section. We compared Greedy to several idealized policies as well as *EQUI*, the policy which divides cores evenly amongst all jobs regardless of class. If we can show the Greedy is either close to OPT or far from another admissible policy at all times, we can stregnthen the above bound.

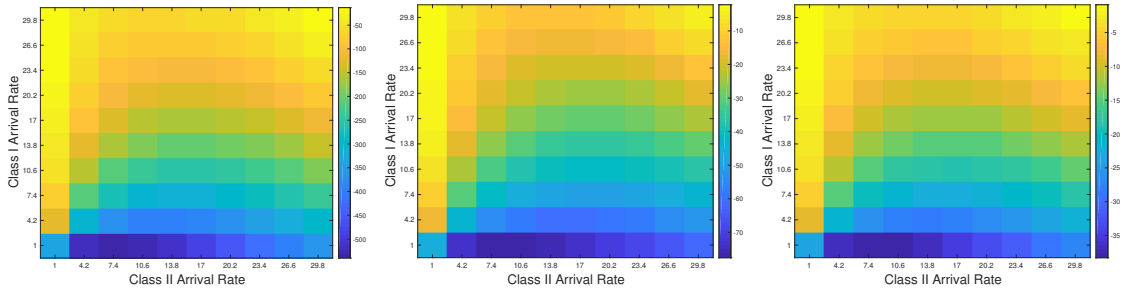
Simulation Results

Our simulation setup attempted to implement our Greedy policy and compare it to *EQUI*, and to then create our idealized circumstances to emulate *OPT* and our upper- and lower-bounds. We chose to fix the number of cores at 64 because that matches a common setup for 2P, 2U commodity servers available. We simulated using three different sets of parallel fractions for the Class I and Class II jobs: [0.9, 0.1], [0.75, 0.5], [0.5, 0.25]. The first pair was chosen to represent heterogeneous jobs, and the latter two represent jobs which have a smaller-yet-still-significant difference in parallel work.

Greedy vs EQUI

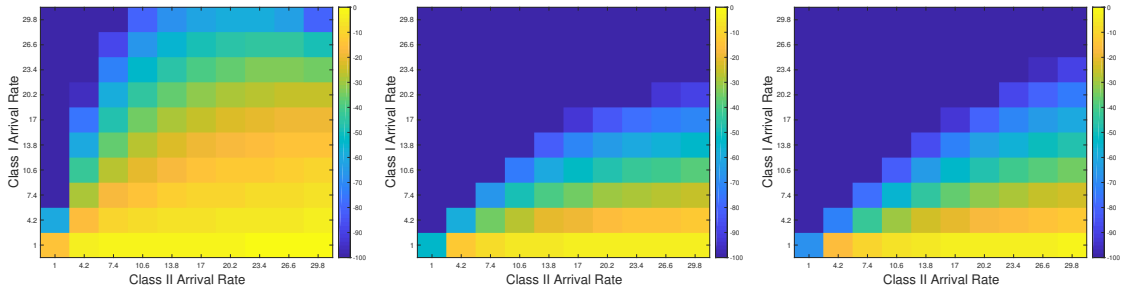
There were two circumstances in which Greedy and *EQUI* were approximately the same: under light loads (arrival rates less than 10), and when the arrival rate was skewed towards Class I or Class II jobs. The former was expected because there were not enough jobs to overwhelm the system. The latter was expected because the system is generally working on only one class of job and is therefore approximating *EQUI*. We were pleased to see that, for all arrival rates, Greedy had a mean response time at or below *EQUI*.

These results are the most significant to our proposal and can be seen in Figure 1. For



(a) [0.9, 0.1] (b) [0.75, 0.5] (c) [0.5, 0.25]

Figure 2: Greedy vs All-Class-I



(a) [0.9, 0.1] (b) [0.75, 0.5] (c) [0.5, 0.25]

Figure 3: Greedy vs No-Class-I

the two pairs of job types with similar parallel fractions ([0.75, 0.5] and [0.5, 0.25]), the best Greedy could do for mean response time was only 3% lower than *EQUI*; however, Greedy was able to stretch its legs and showed up to a 30% advantage under higher loads.

Greedy vs All-Class-I

To emulate a lower bound for *OPT*, we chose to run the same simulation with all of the less-parallel Class II jobs replaced with more-parallel Class I jobs as discussed earlier. As expected, Greedy always had a higher mean response time than the All-Class-I policy, but the disadvantage diminished as the Class I load was increased. This result helps to show that when the mix of jobs is heavily skewed towards Class I, Greedy is close to *OPT* (the top-left corner of the plots in Figure 2), within about 3%.

Greedy vs No-Class-I

Another *OPT* emulation was replacing all Class I jobs with jobs that had zero size (e.g., the jobs would have a mean response time of 0). Again, as expected, this circumstance

led to Greedy having a longer mean response time in all cases–sometimes by 1500% to 2000%, and these results were elided in the plots seen in Figure 3 (the top left) in order to emphasize the dynamic range of the interesting portion (the lower right). This result helped to show that when the the job mix is skewed towards Class II, Greedy is close to *OPT*, within about 2%.

Discussion

These simulation results suggest, surprisingly, that Greedy does *not* outperform EQUI by a significant margin, even when the class I and class II speedup functions are far apart. For example, when simulating the case where $p_1 = .9$ and $p_2 = .1$, we would hope to see at least some arrival rates for what Greedy was close to 10 times better than EQUI. However, the largest percentage difference we see is about a 30% improvement by Greedy.

While the other two comparisons discussed above suggest that Greedy is indeed close to OPT when the arrival rates are heavily skewed in one direction or another, we still cannot say whether Greedy is close to OPT when the arrival rates of jobs from both classes are roughly balanced. For example, it could be that both Greedy and EQUI are far from OPT in this case. Prior work [2] has suggested that when arrivals are balanced, Greedy is close to OPT in practice. Thus, we think it is more likely that EQUI and Greedy are *both close* to OPT in many cases. This is surprising, since it would seem that EQUI drastically overallocates cores to class II jobs. However, even when the speedup functions are a factor of 10 apart, this does not seem to have a deleterious effect on mean response time.

Future Work

Future work should focus on trying to prove a bound between EQUI and OPT. The EQUI policy is easily defined regardless of the specific speedup function used, and may be easier to work with than Greedy. It is still not known, in general, whether Greedy dominates EQUI with respect to mean response time. If this could be proven, any bound between EQUI and OPT would also provide a bound on Greedy vs. OPT. However, it seems unlikely that Greedy is *always* superior to EQUI, since we know that the optimal policy, like EQUI, allocates more cores to class II jobs than Greedy does.

At a higher level, future work in this area will also have to address how to allocate resources to several classes of jobs as opposed to just two classes. While the case of two classes is theoretically interesting and difficult, real systems handle very diverse workloads. It may be even more realistic to consider every job following a speedup function drawn from some distribution rather than restricting the problem to consider discrete classes of jobs. Even if an exact analysis of this situation is intractable, it would be interesting to know what happens to mean response times as job's speedup functions become more diverse. For example, warehouse scale computers are often required to make decisions about which jobs can be effectively colocated on the same machine. Future work could

address this question by suggesting which jobs can be colocated and still recieve a near optimal resource allocation. This work could also quantify the cost of this colocation as opposed to running the jobs in isolation.

Goals

Our biggest change during the project was to shift away from focusing on multiple types of resources (e.g. CPU and cache space) to focusing on one resource with multiple job classes. Our goals are summarized below, with slight updates to reflect this change.

Our 75% goal is to present a model and extend the theoretical results from [2] to handle multiple job classes. We motivate this model by examining existing public data from warehouse-scale computers[9] and will leverage similar datasets available within CMU if possible. We combine these results via trace-driven simulation to validate our theoretical results.

As a 100% goal, we extend our model and simulations to also incorporate two classes of jobs, each with its own speedup function. We describe the performance of the kind of greedy policies described in the literature[4, 6] and make a statement about how good these policies are compared to an optimal resource allocation policy.

As a 125% goal, we will begin running trace driven experiments on a small scale system. The main goal here is to begin providing open source access to software which implements both the ideas of this project and those of the existing literature, since open source implementations of this work are largely unavailable.

We mostly met our 100% goal. The main discrepancy between this paper and the above goals was our plan to incorporate more trace data. We did evaluate the public dataset cited in the 75% goal, but it did not yield interesting results. Mainly, this seems to be due to the obfuscation used in making the trace, which made it hard to pull meaningful performance profiles of jobs out of the data. Instead, we drove our simulations using the assumptions made in theoretical model. Additionally, our backup plan to use another dataset from CMU fell through because the dataset was not made public in time. Still, we did learn about the format and shortcomings of the popular Google WSC dataset, and were able verify some trends identified in prior work such as [4].

References

[1] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 64–75. IEEE, 2015.

[2] Benjamin Berg, Jan-Pieter Dorsman, and Mor Harchol-Balter. Towards optimality in parallel scheduling. *To Appear in SIGMETRICS 2018*, 2017. Preprint at: https://arxiv.org/pdf/1707.07097.pdf.

[3] A. Bušić, I. Vliegen, and A. Scheller-Wolf. Comparing Markov chains: aggregation and precedence relations applied to sets of states, with applications to assemble-to-order systems. *Mathematics of Operations Research*, 37:259–287, 2012.

[4] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[5] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 158–169. IEEE, 2015.

[6] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259. ACM, 2011.

[7] J. McCool, M. Robison, and A. Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier, 2012.

[8] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[9] John Wilkes. More Google cluster data. Google research blog, November 2011. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.