An "A+" Heuristic for Dispatching in Large-Scale Systems with Unknown Server Speeds

Cole Stephens
Department of Computer Science
Amherst College
Amherst, MA, USA

Kristen Gardner
Department of Computer Science
Amherst College
Amherst, MA, USA
kgardner@amherst.edu

***Abstract*—How to dispatch jobs to servers is a question of critical importance in today's computer systems. While there is a long history of literature on dispatching, much of this work focuses on settings where servers are homogeneous in their speeds—an unrealistic assumption in modern computer systems. Dispatching policies that are heterogeneity-aware typically assume that the dispatcher has access to detailed information about all servers' speeds; this may also be unrealistic. In this paper, we study the setting in which server speeds are heterogeneous and the dispatcher has no information about server speeds. We propose a novel heuristic, called *server accomplishment*, that can be used as a proxy for detailed server speed information in heterogeneity-aware dispatching policies. Using extensive simulation studies, we demonstrate that our accomplishment-based dispatching policies are able to bridge most of the performance gap between heterogeneity-unaware policies, such as JSQ(d), and policies that make use of detailed server speed information.**

***Index Terms*—Dispatching; Load balancing; Power-of-d; Heterogeneity**

I. Introduction

How to dispatch jobs to servers is a question of critical importance in today's computer systems. There is a long history of literature that strives to answer this question, beginning with the canonical Join-the-Shortest-Queue (JSQ) policy, under which an arriving job is dispatched to the server with the fewest jobs in its queue. JSQ is known to minimize mean response time in certain settings, and its performance has been analyzed in a variety of regimes [4], [10], [14], [15]. Unfortunately, JSQ is an impractical choice for modern systems for two key reasons. First, today's systems operate at large scale. In systems consisting of hundreds or thousands of servers, querying all servers for their queue lengths upon every job's arrival is prohibitively expensive. Second, today's systems are heterogeneous, meaning that different servers may operate at different speeds. In this regime, JSQ is no longer optimal because it does not account for differing server speeds.

To reduce the communication overhead in large-scale systems, researchers have proposed a number of alternative approaches. In the "power-of-d" approach, the dispatcher queries a small number of servers upon a job's arrival, and dispatches the job to one of the queried servers. A well-known policy in this category is Join-the-Shortest-Queue(d) (JSQ(d)), under which the job is dispatched to the queried server with the shortest queue [7], [13]. In the "pull-based" approach, the dispatcher relies on servers providing periodic updates of their statuses. One policy in this category is Join-Idle-Queue (JIQ), under which the dispatcher sends an arriving job to an idle server if there is one, and to a randomly chosen busy server otherwise [6]. Both of these approaches require far less communication between servers and dispatcher than JSQ, making them suitable for large-scale systems. However, the canonical policies within these approaches assume that server speeds are homogeneous; this assumption can lead to poor performance in heterogeneous settings [5], [9], [11], [16].

More recently, new policies have been proposed that do account for heterogeneous server speeds. Heterogeneity-aware descendants of JSQ(d) include Shortest-Expected-Delay(d), Balanced Routing [2], Hybrid SQ(d) [9], and JSQ(d_F, d_S) [3]. Pull-based policies such as JIQ also can be adapted to be heterogeneity-aware. In all cases, the heterogeneity-aware variants produce lower mean response times than their heterogeneity-unaware analogues. Yet all of the heterogeneity-aware policies require the dispatcher to know some information about the servers' speeds. The required information ranges from the coarse (e.g., heterogeneity-aware JIQ requires only an ordering of servers by their speeds) to the exceedingly detailed (e.g., JSQ(d_F, d_S) requires the exact speed of each server).

In this paper, we study the setting in which server speed information is unavailable to the dispatcher. This situation may arise, for example, in a cloud computing setting where the exact server speeds depend on the underlying physical machine and resource contention experienced by a virtual machine. Unfortunately, the heterogeneity-aware policies proposed in the literature are not feasible in this setting.

Our primary contribution is a novel heuristic, called *accomplishment*, that is used to help the dispatcher estimate server speeds. A server's accomplishment at time t is defined as the number of jobs that have been dispatched to the server by time t. Clearly, the dispatching policy affects the servers' accomplishment values: under random or round-robin dispatching all servers have the same accomplishment, whereas policies that do not treat all servers symmetrically will yield higher accomplishments for the favored servers. In the setting in which server speeds are unknown, accomplishment is a useful heuristic if faster servers tend to become more accomplished. If this is the case, the dispatching policy can favor more accomplished servers as a proxy for favoring faster servers, thereby effectively implementing a heterogeneity-aware policy

even when server speeds are unknown.

Because the dispatching policy's decisions affect the servers' accomplishment, which in turn affects later dispatching decisions, it is not at all obvious that using accomplishment as a proxy for speed will actually yield dispatching policies that perform well. We will show how four heterogeneity-aware dispatching policies, SED(d), BR, JIQ, and JSQ(d_F, d_S), can be adapted to incorporate the accomplishment heuristic, yielding four new "Accomplishment+" policies: A+SED(d), A+BR, A+JIQ, and A+JSQ(d_F, d_S). Our extensive simulation results demonstrate that, in many cases, the A+ policies are capable of closing most of the performance gap between heterogeneity-unaware policies such as JSQ(d) and JIQ, and their heterogeneity-aware descendants. As an added benefit, accomplishment is a low-communication heuristic, requiring no more communication between servers and dispatcher than the baseline heterogeneity-aware policies.

The remainder of this paper is organized as follows. In Section II we introduce our model and formally define the accomplishment heuristic. In Section III we define the four "A+" policies that we consider and empirically evaluate their performance in systems with two server speed classes and exponentially distributed service times. Sections IV-A, IV-B, and IV-C address three generalizations aimed at moving to more realistic system settings: general service times, more than two server speed classes, and server speeds that may change over time. Finally, in Section V, we conclude.

II. Model and Preliminaries

We consider a system that consists of k servers with heterogeneous speeds, where k_F of the servers are "fast" and $k_S = k \quad k_F$ of the servers are "slow" (for more than two classes of servers, see Section IV-B). Service times are exponentially distributed with rate μ_F at fast servers and rate $\mu_S < \mu_F$ at slow servers (for general service times, see Section IV-A). The overall capacity of the system is $\mu_F k_F + \mu_S k_S = 1$. Each server has its own dedicated queue and processes the jobs in its queue in first-come first-served (FCFS) order.

Jobs arrive to the system as a Poisson process with rate λk. Upon arrival, a job is dispatched to a single server according to some dispatching policy. We assume that the dispatcher has limited information, both about jobs and about servers: the dispatching decision can be based on queue length information (for which the dispatcher can query individual servers), but it cannot be based on any information about job sizes or server speeds. In particular, we consider the setting in which the dispatcher *does not have any information about server speeds*.

We consider several dispatching policies, beginning with two well-studied heterogeneity-unaware policies. Under **Join-the-Shortest-Queue(d)** (JSQ(d)), when a job arrives the dispatcher queries d servers uniformly at random. The job is dispatched to the queried server with the fewest jobs in its queue (including the job in service, if there is one). Under **Join-Idle-Queue** (JIQ), the dispatcher maintains a queue of idle servers. When a job arrives, it is dispatched to an idle server (in FIFO order) if there are any, and to a busy server chosen uniformly at random otherwise.

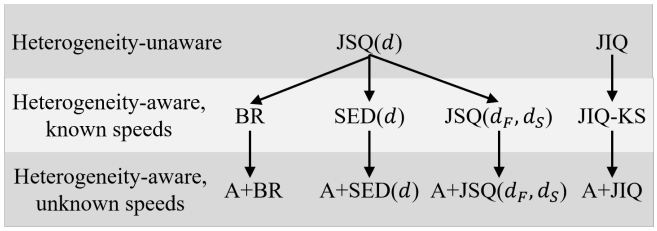


Fig. 1. Taxonomy of policies we consider.

The dispatching policies that we propose are based on a novel heuristic, called *accomplishment*, that allows the dispatcher to use server heterogeneity information without knowing server speeds *a priori* or attempting to measure them.

Definition 1: A server i's **accomplishment** at time t, denoted by $a_i(t)$, is the number of jobs that have been dispatched to server i by time t. Server i is *more accomplished* than server j at time t if $a_i(t) > a_j(t)$. When the time is clear from context, we will drop the t in our notation and write a_i.

We use the idea of server accomplishment to augment several existing heterogeneity-aware dispatching policies. In our "A+" version of each policy, we replace server speed information with server accomplishment, making the "A+" policies suitable for settings where server speeds are unknown. In the sections that follow, we will review each of these existing policies and then define the accomplishment-based policy augmentations. Figure 1 gives a hierarchy of the dispatching policies studied in this paper.

III. Performance Evaluation

In this section, we describe how to apply the accomplishment heuristic to four heterogeneity-aware dispatching policies: Join-Idle-Queue, Balanced Routing, Shortest-Expected-Delay(d), and Join-the-Shortest-Queue(d_F, d_S). We define the "A+" version of each policy and compare its mean response time, E[T], to that of the baseline heterogeneity-unaware policy and the heterogeneity-aware "parent" policy. Throughout, we present simulation results for a system with $k = 1000$ servers and we set $d = 4$ for JSQ(d) and its descendants; 95% confidence intervals were all less than 2%.

A. Accomplishment+Join-Idle-Queue

We begin by incorporating the accomplishment heuristic into a heterogeneity-aware version of JIQ. Under all versions of JIQ, the dispatcher maintains a queue of idle servers. An arriving job is dispatched to an idle server if there is one, and to a server chosen uniformly at random if not. The versions differ in how ties are broken among multiple idle servers. Under **Join-Idle-Queue-Known-Speeds** (JIQ-KS), ties are broken in favor of the *fastest* idle server. Under **Accomplishment+JIQ** (A+JIQ), ties are broken in favor of the *most accomplished* idle server, thereby approximating JIQ-KS.

Figure 2 compares E[T] under JIQ, JIQ-KS, and A+JIQ. As λ increases, E[T] under JIQ—which breaks ties among

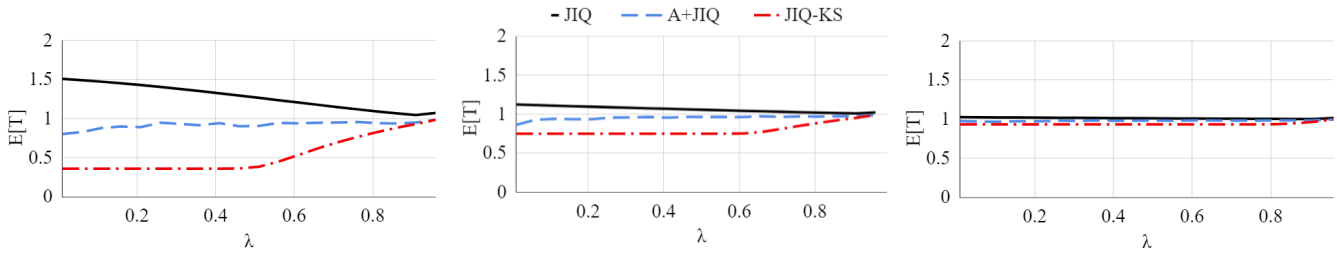


Fig. 2. E[T] as a function of under JIQ, A+JIQ, and JIQ-KS for three different settings of q_F and r.

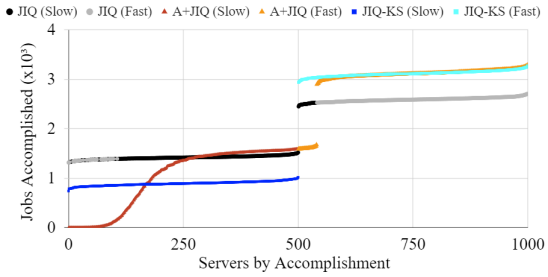


Fig. 3. Number of jobs accomplished by each server (ordered by accomplishment) under JIQ, A+JIQ, and JIQ-KS, after 2 10^6 arrivals, where q_F = 0:5, r = 2, and = 0:2.

idle servers in FIFO order—improves. When arrivals are more frequent, a busy fast server is likely to return to the idle queue before a busy slow server. This means that fast servers are more frequently at head of the idle queue, so jobs are dispatched to fast servers disproportionately often, in turn decreasing E[T]. On the other hand, E[T] under JIQ-KS remains constant as long as $k\lambda < k_F\mu_F$, i.e., if the fast servers have enough capacity to maintain stability without needing the slow servers. Once λ is high enough that the slow servers are needed to ensure stability, E[T] under JIQ-KS begins to increase. E[T] under A+JIQ falls between that under JIQ and JIQ-KS at all values of λ. By sorting idle-queues by server accomplishment, A+JIQ provides a significant improvement in E[T] compared to JIQ, even in the low λ, high r setting where JIQ performs poorly. This result demonstrates that using the accomplishment heuristic can shrink the E[T] gap between JIQ and JIQ-KS, despite the lack of server speed information.

Figure 3 compares the servers' accomplishment levels after 2 10^6 arrivals. Under all three variants of JIQ, the fast servers consistently have higher accomplishments than the slow servers. Notably, the fast servers' accomplishments are similar under JIQ-KS and A+JIQ, and are higher under these two policies than under baseline JIQ. This illustrates that the low response time achieved by heterogeneity-aware policies is driven by the proportion of jobs dispatched to fast servers.

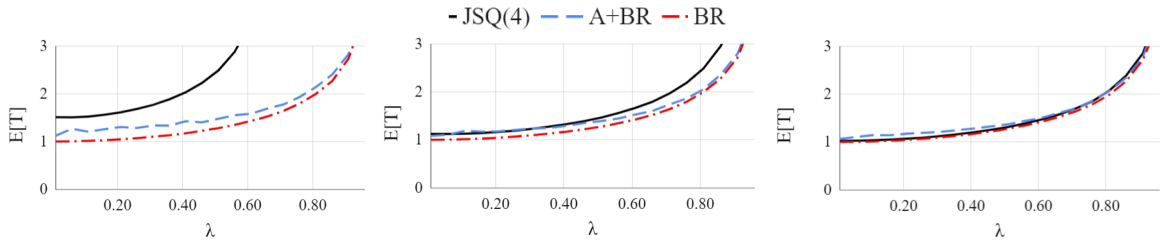
B. Accomplishment+Balanced Routing

We next consider Balanced Routing, a heterogeneity-aware "descendant" of JSQ(d). Under both the baseline and the A+ versions of Balanced Routing, when a job arrives the dispatcher queries d servers and sends the job to the queried server with the fewest jobs in its queue (including the job in service, if there is one). The difference between the baseline and A+ versions lies in the probabilities with which the d servers are selected. Under **Balanced Routing** (BR), servers are queried with probabilities proportional to their *speeds*: server i is queried with probability $\frac{\mu_i}{\sum_{j=1}^{k}\mu_j}$. For the setting in which server speeds are unknown, we use each server i's accomplishment to compute its estimated speed, μ_i^{est}. Let $a^{\text{RAND}}(t) = N(t)/k$ denote the expected number of jobs accomplished by server i by time t under random dispatching, where $N(t)$ is the total number of jobs that have arrived to the system by time t. We set $\mu_i^{\text{est}}(t)$ $a_i(t)/a^{\text{RAND}}(t)$. Under **Accomplishment+BR** (A+BR), servers are queried with probabilities proportional to their *accomplishments*: server i is queried with probability $\frac{\mu_i^{\text{est}}}{\sum_j \mu_j^{\text{est}}} = \frac{a_i}{\sum_{j=1}^{k} a_j}$.

Intuitively, under A+BR the query probabilities are determined by asking "how accomplished is this server compared to the expected accomplishment under random dispatching?" We expect faster servers to be more accomplished by time t under BR than under random dispatching, thus faster servers tend to have higher estimated speeds. As a result, fast servers are queried more often. This in turn increases their accomplishment, creating a feedback loop in which more accomplished servers become even more likely to be queried in the future.

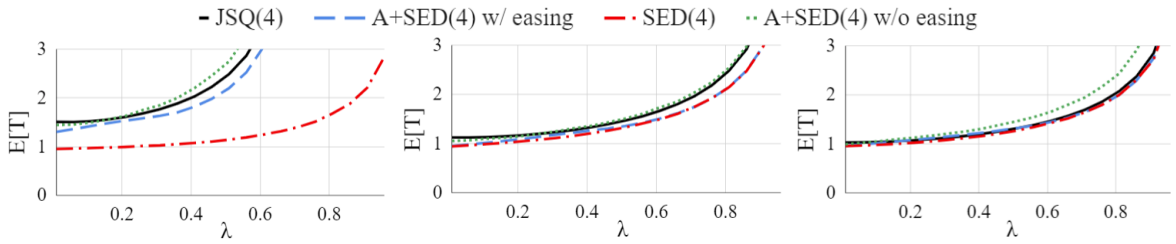
Figure 4 shows E[T] under JSQ(d), BR, and A+BR. As under the JIQ-based policies, when r is low all three policies perform similarly because the system is relatively homogeneous. At high r, JSQ(d) does not perform as well as BR and A+BR because it does not take advantage of the larger difference between fast and slow server speeds.

A+BR performs somewhat similarly to JSQ(d) when λ is low, and increasingly close to BR when λ is high. At low λ most queues tend to be idle, so joining the shortest queue among d queried servers often amounts to random routing among those d servers. In this case, all servers tend to have similar accomplishment values, and so A+BR makes similar querying and dispatching decisions to JSQ(d). As λ increases, both fast and slow servers are busy more often; because fast servers complete their assigned jobs more quickly than slow servers, they in turn are dispatched more jobs. Indeed, at high λ the fast servers are more accomplished than slow servers by approximately a factor of r: in this regime the



(a) $q_F = 0.2$, $r = 5$ (b) $q_F = 0.5$, $r = 2$ (c) $q_F = 0.8$, $r = 1.5$

Fig. 4. E[T] as a function of under BR, A+BR, and JSQ(d) for three different settings of q_F and r.



(a) $q_F = 0.2$, $r = 5$ (b) $q_F = 0.5$, $r = 2$ (c) $q_F = 0.8$, $r = 1.5$

Fig. 5. E[T] as a function of under SED(d), A+SED(d) with and without the initial easing policy, and JSQ(d) for three different settings of q_F and r.

accomplishment-based approximation of server speeds is quite accurate, and so A+BR and BR converge.

That A+BR performs as well as it does is perhaps surprising: unlike A+JIQ, which relies on the accomplishment heuristic only to estimate the *ordering* of servers by their speeds, A+BR uses accomplishment to estimate the *actual speed* of each server. The BR policy, then, is likely to be much more sensitive to errors in estimating μ. Nonetheless, our results demonstrate that A+BR typically performs very similarly to BR.

C. Accomplishment+Shortest-Expected-Delay(d)

We now turn to Shortest-Expected-Delay(d), another heterogeneity-aware "descendant" of JSQ(d). Under both the baseline and A+ versions of SED(d), when a job arrives the dispatcher queries d servers chosen uniformly at random and computes the expected delay at each queried server i, denoted D_i. The job is dispatched to the queried server i with the smallest D_i. The versions of SED(d) differ in how D_i is computed. Under **Shortest-Expected-Delay**(d), we set $D_i = \frac{N_i+1}{\mu_i}$, where N_i denotes the number of jobs at server i. In order to calculate expected delay when speeds are unknown, we must estimate server speeds. Consistent with the server speed estimation used for Balanced Routing, we set $\mu_i^{\text{est}}(t) \quad a_i(t)/a^{\text{RAND}}(t)$ for each server i. Under **Accomplishment+SED**(d) (A+SED(d)), we set $D_i = \frac{N_i+1}{\mu_i^{\text{est}}}$.

Surprisingly, as defined, A+SED(d) yields higher E[T] than JSQ(d) (see Figure 5). Figure 6 illustrates why: A+SED(d) frequently misestimates server speeds by a considerable margin. To understand why this misclassification occurs, observe that the dispatching decisions for the earliest arrivals significantly impact the dispatching decisions made in the future. Because all servers start with zero accomplishment, a server that is not assigned any jobs early on will have significantly lower

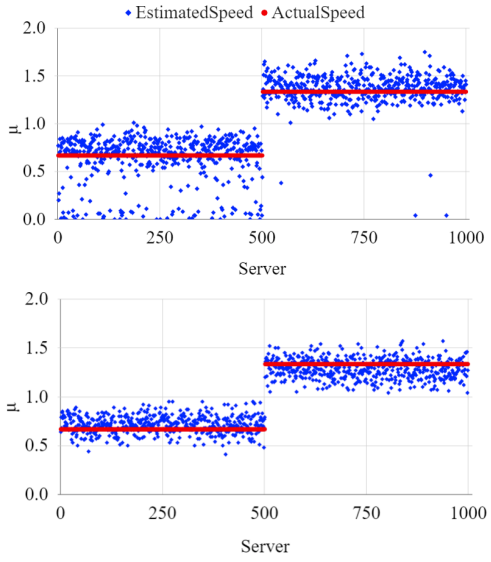
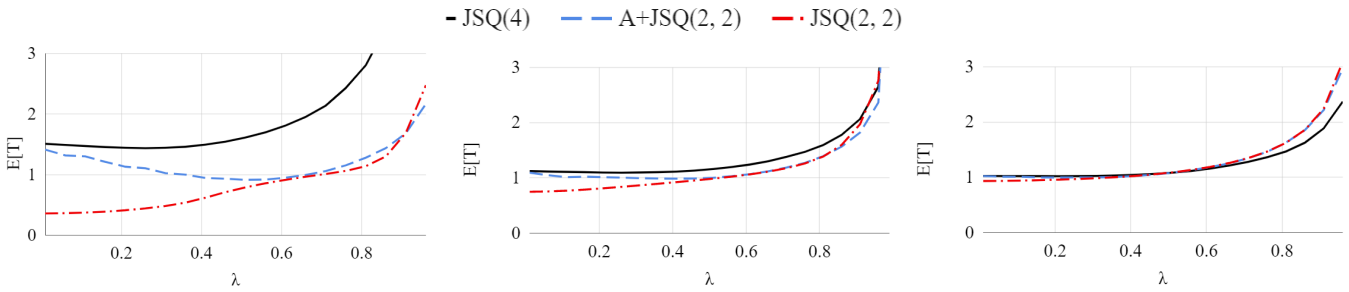


Fig. 6. Estimated speed, $\mu_i^{\text{est}}(t) \quad a_i(t)/a^{\text{RAND}}(t)$, compared to actual speed, μ_i, after 100,000 arrivals without easing (top) and with easing (bottom).

accomplishment than its expectation under random. Thus, its estimated speed will be much slower than its actual speed.

We resolve this challenge with an "easing policy" that sets $\mu^{\text{est}} = 1$ for all servers for the first 20 000 arrivals, effectively running JSQ(d) for this period. This ensures that all servers receive some jobs while also allowing the fast servers to become more accomplished than slow servers. The easing policy increases the accuracy of the μ^{est} values (see Figure 6). Figure 5 shows that, when r is low, A+SED(d) with the easing policy yields similar E[T] to SED(d). This speaks volumes about the value of the accomplishment heuristic:



(a) $q_F = 0.2$, $r = 5$ (b) $q_F = 0.5$, $r = 2$ (c) $q_F = 0.8$, $r = 1.5$

Fig. 7. E[T] as a function of under JSQ(4), JSQ(2,2), and A+JSQ(2,2) for three different settings of q_F and r.

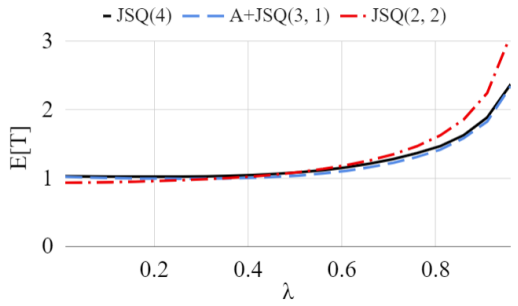


Fig. 8. Mean response time as a function of under JSQ(4), JSQ(2,2), and A+JSQ(2,2) when $d = 4$, $d_F = 3$, and $s = q_F$. Here $q_F = 0.8$ and $r = 1.5$.

even though SED(d) is sensitive to detailed server speed information, A+SED(d) often can match its performance.

The need for the easing policy demonstrates that care must be taken when augmenting heterogeneity-aware policies with the accomplishment heuristic. While a server's accomplishment can offer a useful proxy for the server's speed, the ultimate performance of an A+ policy depends on the interaction between dispatching decisions and accomplishment. With A+SED(d) without the easing policy, this interaction causes a performance degradation—in stark contrast to A+BR and A+JIQ. As such, when determining how to apply the accomplishment heuristic to a new policy, one must carefully assess whether the interaction between accomplishment and the dispatching policy will lead to a positive or negative feedback loop, and then adjust the balance of exploring a large number of servers and exploiting the information provided by the servers' accomplishment values accordingly.

D. Accomplishment+Join-the-Shortest-Queue(d_F, d_S)

Finally, we consider the JSQ(d_F, d_S) policy, a descendant of JSQ-d designed specifically for heterogeneous systems.

Under **JSQ**(d_F, d_S), when a job arrives the dispatcher queries d_F fast servers and d_S slow servers, where $d_F + d_S = d$, a constant. If any of the queried fast servers are idle, the job is dispatched to an idle fast server. If all queried fast servers are busy and any of the queried slow servers are idle, the job is dispatched to an idle slow server with probability p_S and to the queried fast server with the shortest queue with probability $1 - p_S$. If all queried servers are busy, the job is dispatched to the queried fast server (respectively, slow server) with the shortest queue with probability p_F (respectively, $1 - p_F$). The probabilities p_F and p_S are chosen optimally to minimize E[T], given system parameters μ_F, μ_S, q_F, q_S, and λ.

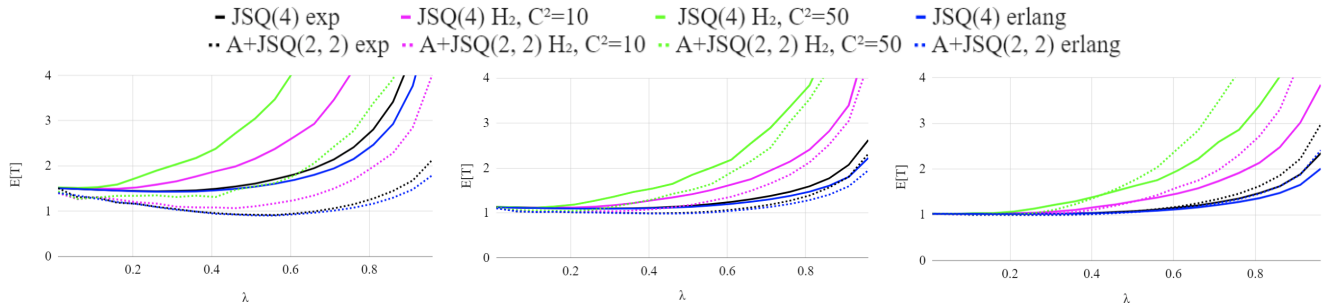
JSQ(d_F, d_S) is highly sensitive to the detailed server speed information: p_F and p_S are optimized based on μ_F, μ_S, q_F, and q_S. Absent accurate knowledge of these system parameters, it is infeasible to optimize the policy parameters. We propose the A+JSQ(d_F, d_S) policy, which uses server accomplishment to capture the spirit of JSQ(d_F, d_S) without requiring detailed server speed information.

Under **A+JSQ**(d_F, d_S), when a job arrives the dispatcher queries d_F servers from among the ks most accomplished servers and d_S servers from among the $k(1 - s)$ least accomplished servers, where s is a policy parameter capturing the dispatcher's estimate of q_F. The job is then dispatched to the queried server with the shortest queue.

We note that the performance of A+JSQ(d_F, d_S) is sensitive to the choice of the policy parameter s. In this section we set $s = q_F$, which yields the best performance, thereby demonstrating the potential of A+JSQ(d_F, d_S). Throughout this section, unless otherwise specified, we set $d_F = d_S = 2$.

Figures 7(a) and (b) compare E[T] under A+JSQ(d_F, d_S) to that under JSQ(d) and JSQ(d_F, d_S) at moderate to high r and low to moderate q_F. At low λ, A+JSQ(d_F, d_S) performs similarly to JSQ(d). As λ increases, E[T] under A+JSQ(d_F, d_S) actually *decreases*, eventually matching the performance of its fully heterogeneity-aware counterpart at higher load. The idea behind A+JSQ(d_F, d_S) is that it is beneficial to guarantee that some fast servers are always included in the query. The accomplishment heuristic can only help achieve this if fast servers do in fact receive more jobs than slow servers. When λ is very low, this does not occur because most servers are idle and thus receive similar numbers of jobs. As λ increases, queues build up at all servers; the fast servers complete their jobs more quickly than the slow servers, so they tend to have shorter queues, and, in turn, they become more accomplished. Thus, at high λ, the accomplishment heuristic has the desired effect of successfully sorting servers by their speeds.

When r is low and q_F is high (Figure 7(c)), we see the opposite results: JSQ(d) in fact outperforms JSQ(d_F, d_S) and A+JSQ(d_F, d_S) at high load. This performance reversal is



(a) $q_F = 0.2$, $r = 5$ (b) $q_F = 0.5$, $r = 2$ (c) $q_F = 0.8$, $r = 1.5$

Fig. 9. E[T] as a function of under JSQ(4) and A+JSQ(2,2), where service times are generally distributed.

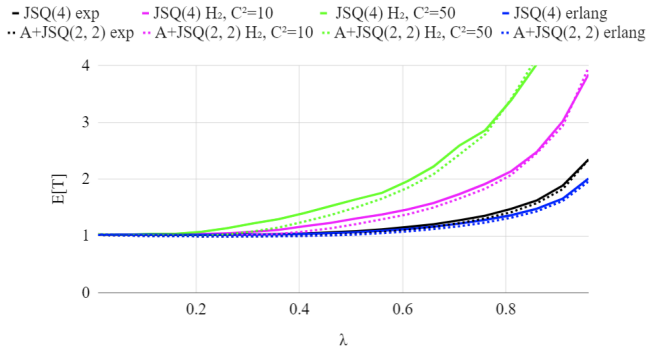


Fig. 10. Mean response time as a function of under JSQ(4) and A+JSQ(2,2), where service times are generally distributed, and $d = 4$, $d_F = 3$, and $s = q_F$. Here $q_F = 0.8$ and $r = 1.5$.

a consequence of the specific parameter settings used for A+JSQ(d_F, d_S). In particular, we have $q_F = 0.8$, $d = 4$, and $d_F = 2$, meaning that only half of the queried servers are fast even though 80% of the servers are fast: fast servers are underrepresented in the query. Figure 10 shows results for the same setting, using $d_F = 3$ instead of $d_F = 2$, i.e., the fast servers are queried with probability roughly proportional to the fraction of servers that are fast. Here we see that A+JSQ(d_F, d_S) consistently outperforms JSQ(d).

IV. Extensions and Generalizations

In this section we evaluate the performance of the accomplishment heuristic in more realistic settings. Throughout, in the interest of brevity, we focus on A+JSQ(d_F, d_S). While we do not show our results for other policies, the lessons that we learn from studying A+JSQ(d_F, d_S) in more general settings apply to all of the policies we introduce in Section III.

A. General Service Times

Thus far, we have assumed that service times are exponentially distributed. We now turn to general service times and evaluate the sensitivity of our results to service time variability.

We consider four different service time distributions. Given q_F and r, for all four distributions we let the mean service time on fast servers and slow servers be $\frac{1}{\mu_F}$ and $\frac{1}{\mu_S}$ respectively, where $\mu_F = r \mu_S$ and $\mu_F q_F + \mu_S q_S = 1$ (see Section II). The first two distributions are the exponential (see Section III) and the two-phase Erlang. The remaining two distributions are two-phase hyperexponentials; the service time on the slow servers is drawn from the distribution $H_2(p_1; \mu_1, \mu_2)$, where $\frac{p_1}{\mu_1} + \frac{1-p_1}{\mu_2} = \frac{1}{\mu_S}$ and $\frac{p_1}{\mu_1} = \frac{p_2}{\mu_2}$ (the service time distribution on the fast servers is defined analogously). We consider two hyperexponential distributions with squared coefficients of variation $C^2 = 10$ and $C^2 = 50$ respectively.

In Figures 9(a) and (b) we see that A+JSQ(d_F, d_S) outperforms JSQ(d) by a similar margin regardless of the service time variability, for the same three system configurations considered throughout Section III. The opposite trend is evident in Figure 9(c), for the same reason as in Section III-D: in this case, the fast servers are underrepresented in the query. As before, when we set $d_F = 3$, A+JSQ(d_F, d_S) consistently outperforms JSQ(d) (see Figure 10). Unsurprisingly, mean response time increases with service time variability for all system parameter settings.

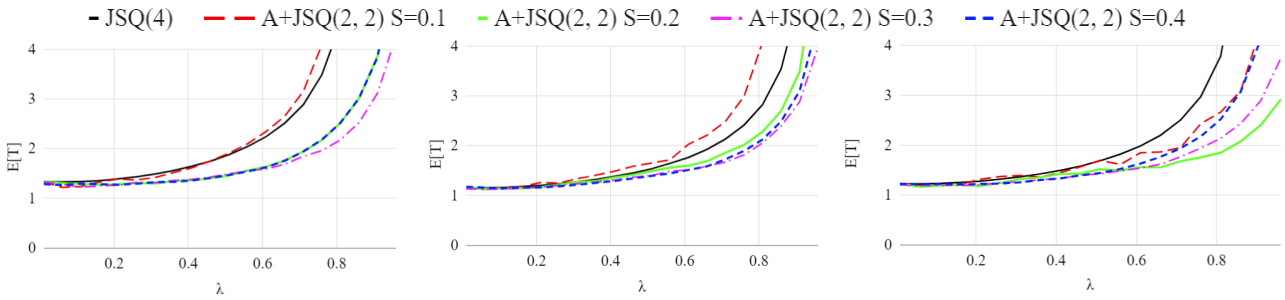
Our results indicate that the strong performance of A+JSQ(d_F, d_S) is insensitive to the service time distribution. This suggests that our A+ policies remain good candidates for achieving low mean response time when server speeds are unknown in settings with highly variable service times, as is more realistic in many practical systems.

B. More Than Two Server Classes

So far we have restricted our attention to settings with only two server speeds; this may be unrealistic in practice. For example, a system can comprise of different server hardware generations, or certain virtual machines may experience more resource contention than others. We now extend our results to this more general setting.

We consider three different systems with four classes of servers, each with a different set of server speeds and distribution of servers among the four classes. In all cases, the average service rate is 1; hence, we consider servers with rate $\mu > 1$ to be "fast" and servers with rate $\mu < 1$ to be "slow."

There are many ways in which one could generalize the A+JSQ(d_F, d_S) policy for more than two server speeds; we study one relatively simple such generalization, defined as follows. Upon each job's arrival the dispatcher queries d servers, of which d_F are chosen from among the sk most accomplished servers and d_S are chosen from among the $(1 - s)k$ least



(a) $\vec{\mu} = (0.45, 0.65, 0.9, 2)$ $\vec{q} = (0.25, 0.25, 0.25, 0.25)$

(b) $\vec{\mu} = (0.7, 1, 1.35, 2)$ $\vec{q} = (0.6, 0.133, 0.133, 0.134)$

(c) $\vec{\mu} = (0.6, 0.77, 1.1, 3)$ $\vec{q} = (0.3, 0.433, 0.167, 0.1)$

Fig. 11. Mean response time as a function of for JSQ(d) and A+JSQ(d_F, d_S) with in systems with four server classes, where ~ = (1;:::; 4) gives the speeds of all classes, and q = (q_1;:::;q_4) gives the distribution of servers across the four classes.

accomplished servers. That is, even though there are more than two server classes, the dispatcher continues to classify servers into only two groups. Choosing s is challenging in this setting, and, as noted in Section III-D, the choice of s can substantially impact the system's performance. Our results in this section are for s 2 f0.1, 0.2, 0.3, 0.4g; when s is low, there are relatively few slow servers that are classified as fast.

A+JSQ(d_F, d_S) reduces E[T] relative to JSQ(d) for most values of s that we consider (see Figure 11; we omit JSQ(d_F, d_S), which is not well defined in this setting). JSQ(d) only outperforms A+JSQ(d_F, d_S) when s = 0.1. Here, the fastest server class contains more than sk servers, so A+JSQ(d_F, d_S) quries the fastest servers *less* frequently than the uniform querying used by JSQ(d). When s is higher A+JSQ(d_F, d_S) outperforms JSQ(d) because A+JSQ(d_F, d_S) queries the fast servers more often than JSQ(d).

As defined, A+JSQ(d_F, d_S) is not optimal in the many-speed setting. Opportunities for further improvement include classifying servers into more than two speed categories, using accomplishment data (see Figure 3) to learn the appropriate cutoffs between server classes, and choosing which server classes to query probabilistically, as in [3]. Nonetheless, our results demonstrate that the accomplishment heuristic continues to be a useful tool in this more realistic setting.

C. Changing Server Speeds

In some systems, server speeds are unlikely to stay fixed over time. For example, migration of a VM could cause the VM's speed to change at certain points in time. When server speeds change, the accomplishment-based predictions of relative server speeds—upon which our A+ policies rely—could suddenly become inaccurate. Here we evaluate the robustness of the accomplishment heuristic to changes in server speeds.

Figure 12 shows, as a function of the total number of arrivals to the system, the fraction of servers that are *actually* fast,slowg and that are *classified* as ffast,slowg. We consider two scenarios: one in which each server's speed changes with probability 0.4 after relatively few arrivals, and the other in which the speeds change after many arrivals. In the former setting, only a short amount of time is required for the

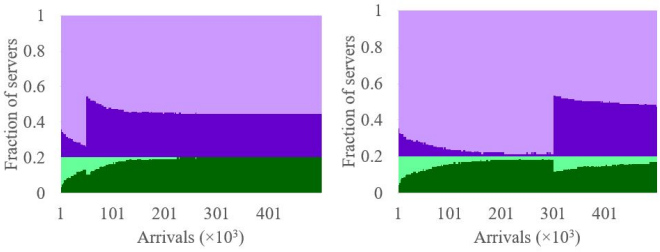


Fig. 12. Fraction of servers that are actually fast and classified fast (light purple), actually slow and classified fast (dark purple), actually fast and classified slow (light green), and actually slow and classified slow (dark green). Here r = 1:5, = 0:5, and the probability that a server changes speed is 0:4. The server speed changes occur after 50 000 arrivals (left) or 300 000 arrivals (right).

classification to return to near-perfect accuracy, whereas in the latter setting the recovery is slow. After many arrivals, the (initially) fast servers have far higher accomplishments than the slow servers (see, e.g., Figure 3); after the speeds change, many arrivals are needed before the newly-fast servers' accomplishments can catch up to those of their initially-fast counterparts. The consequence of this slow recovery time is that mean response time, in turn, will suffer, as many jobs will be dispatched to servers that are mistakenly classified as fast.

Motivated by this observation, we modify the A+ policies so that the dispatcher stores only a *partial* history of servers' accomplishments. Specifically, we track each server's accomplishment over only the most recent 40 000 arrivals, grouped in "buckets" of 10 000 arrivals; after each 10 000 arrivals to the system, we discard the information in each server's oldest "bucket." Thus, out-of-date information collected prior to server speed changes will be discarded quickly.

We evaluated the performance of the partial-history accomplishment for a wide range of parameter settings; Figure 13 shows the results for a few cases that illustrate the overall patterns. We note that the classification accuracy depends on q_F, λ, r, and the probability with which speeds change, with higher accuracy associated with higher values of both λ and r, lower probabilities of changing speeds, and values of q_F close to 0.5. Importantly, regardless of the system parameters,