Synergy via Redundancy: Boosting Service Capacity with Adaptive Replication

Gauri Joshi
IBM T. J. Watson Research Center
Yorktown Heights NY 10598
Email: gauri.joshi@ibm.com

ABSTRACT

The maximum possible throughput (rate of task completion) of a multi-server system is typically the sum of the service rates of individual servers. Recent works show that task replication can boost the throughput, in particular if the service time has high variability ($C_v > 1$). Thus, redundancy can be used to create synergy among servers such that their overall throughput is greater than sum of individual servers. This paper seeks to find the fundamental limit of this capacity boost achieved by task replication. The optimal adaptive replication policy can be found using a Markov Decision Process (MDP) framework, but the MDP is hard to solve in general. We propose two myopic policies, MaxRate and AdaRep that gradually add replicas only when needed. To quantify the optimality gap of these policies, we also derive an a upper bound on the service capacity.

1. INTRODUCTION

The large-scale sharing of resources in today's cloud systems provides scalability and flexibility. An adverse effect of resource sharing is that the response time of individual servers in the cloud can be large and unpredictable. This inherent variability in response time is the norm and not an exception [1]. A solution is to replicate computing tasks at multiple servers and wait for any one copy to finish. This idea was first used at a large-scale in MapReduce and further developed in several other systems works.

Although used in practical systems, only a few theoretical works provide an understanding of when redundancy is most beneficial in reducing latency. Works such as [2–7] study a multi-server queueing system where incoming tasks can be replicated at multiple queues, and as soon as any one replica is served, its copies are canceled immediately. The objective is to minimize the latency, which includes the service time of a task and its waiting time in queue. Task replication affects the latency in two opposite ways:

- **Queue Diversity**: Replicas provide diversity by help finding the shortest among the queues that they join, thus reducing the overall waiting time.
- **Load due to Redundant Service**: Multiple replicas of a task may enter service at different servers, potentially adding load to the system and increasing the waiting time for subsequent tasks.

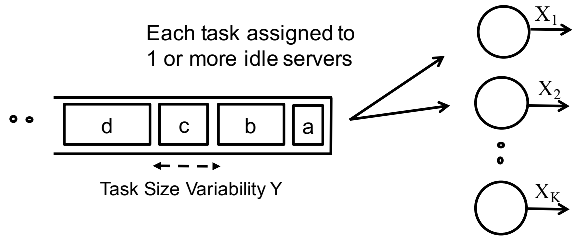

Fig. 1. System of K servers where a task replicated at two idle servers 1 and 2 takes time $Y \cdot \min(X_1, X_2)$ to finish, where the random variable Y captures the task-size variability and X_i captures the service time variability.

The effect of the redundant service of replicas is not well-understood yet. Works like [2, 4, 7] identify surprising scenarios where replication in fact results in the rate of task completion being higher than the sum of service rates of individual servers.

This paper seeks to find the maximum possible throughput or the *service capacity* of a multi-server system, and the replication policy that achieves it. To the best of our knowledge, this is the first paper to attempt finding the service capacity with replication. Our system model accounts for server heterogeneity, task size variability as well as delays in cancellation of replicas. In Section 3 we analyze 'upfront' replication policies that launch all replicas at the same time. An alternative is to add replicas *adaptively*, only if the original task does not finish in reasonable time. Finding the optimal adaptive policy involves solving a Markov Decision Process (MDP). We formulate this MDP in Section 4. This MDP can have a large state space and we need to resort to myopic policies. We propose two replication policies, MaxRate and AdaRep, that adaptively add replicas and perform better than upfront replication policies. To quantify the gap from optimality we give an upper bound on the service capacity for the two-server case. All proofs are deferred to the Appendix, which can be found in the extended version [8].

2. Problem Formulation

Consider a system of K servers with a central queue containing tasks, as illustrated in Fig. 1. We do not explicitly define a task arrival process and instead assume that the central queue is never idle. This flooded central queue model obviates the effect of queue diversity provided by replicas in a distributed queueing system, and helps us focus on the effect of redundant service of replicas.

2.1 Task Service Times

Server i takes time $S = Y \cdot X_i$ to finish a task assigned to it. The random variable X_i captures the variability in task service time due to server slowdown, assumed to be i.i.d. across tasks assigned to that server. The dependence of the service time on the size of the task is captured by Y, which is independent of X_i for all i. This method of multiplying the randomness from the two sources of variability was introduced in [9]. The value of Y is same across replicas of a task. Thus, if a task is replicated at two idle servers i and j, the time taken to complete any one replica is $Y \cdot \min(X_i, X_j)$. We also consider that when a task is replicated, each server running it reserves a cancellation window of length Δ. As soon as one replica is served, the scheduler sends a cancellation signal to the other replicas, triggering their cancellation. All these events occur in time Δ, after which the servers are available to serve subsequent tasks.

2.2 Scheduling Policy

The policy π used to schedule replicas can be based on the distributions of Y, X_1, $\dots$, X_K. The scheduler only knows these distributions, but does not know their realizations for currently running tasks. As soon as a server becomes idle, the scheduler can take one of two possible actions:

- ***new***: assign a new task to that server
- ***rep***: launch a replica of a task currently running on one of the other servers.

The space of scheduling policies with these actions is denoted by $\Pi_{n,r}$ and we aim to find the policy $\pi^*_{n,r}$ that maximizes the throughput. This space of policies can be expanded by allowing additional actions such as pausing a currently running task, or killing and relaunching it to another server. We consider ***new*** and ***rep*** as the only feasible actions in this paper, except in Section 5 where we use task pausing to find an upper bound on service capacity.

Note that all policies in $\Pi_{n,r}$ are work-conserving, that is, they do not allow any server to be idle for a non-zero time interval. We can show that there is no loss of generality in restricting our attention to work-conserving policies.

CLAIM 1. *The throughput-optimal scheduling policy π^* is work-conserving, that is, it does not allow any server to be idle for a non-zero time interval.*

2.3 Performance Metrics

DEFINITION 1 (THROUGHPUT R). *Let $T_1(\pi) \le T_2(\pi) \le \dots \le T_n(\pi)$ be the departure times of tasks $1, 2, \dots n$ from the system, when the scheduler follows a policy π. The throughput of the system is defined as*

$$R(\pi) \triangleq \lim_{n\to\infty} \frac{n}{T_n(\pi)}. \tag{1}$$

DEFINITION 2 (SERVICE CAPACITY $R^*_{n,r}$). *The service capacity $R^*_{n,r} = \max_{\pi \in \Pi_{n,r}} R(\pi)$, the maximum achievable throughput over all policies in $\Pi_{n,r}$. The policy $\pi^*_{n,r}$ that achieves $R^*_{n,r}$ is called the throughput-optimal policy.*

Next we define another performance metric, the computing time C per task.

DEFINITION 3 (COMPUTING TIME C). *The total time spent by the servers on a task is called the computing time C.*

The expected computing time $\mathbb{E}[C]$ is proportional to the cost of running a task on a system of servers. In our system model, if a task is assigned to only to server i then $\mathbb{E}[C] = \mathbb{E}[Y]\,\mathbb{E}[X_i]$. Instead if it is assigned to two servers i and j, and the replica is canceled when any one copy finishes then $\mathbb{E}[C] = 2(\mathbb{E}[Y]\,\mathbb{E}[\min(X_i, X_j)] + \Delta)$ where Δ is the cancellation window at each of the servers. Depending upon X_i, Y and Δ, $\mathbb{E}[C]$ with replication may be greater or less than that without replication.

The throughput R can be expressed in terms of the expected computing time $\mathbb{E}[C]$ as given below.

CLAIM 2. *For any work-conserving scheduling policy,*

$$R = \frac{K}{\mathbb{E}[C]}. \tag{2}$$

Thus, minimizing $\mathbb{E}[C]$ is equivalent to maximizing R.

3. UPFRONT REPLICATION

In this section we explore 'upfront' replication policies that simultaneously launch a task and its replicas. The number of replicas and the servers where they are launched governs the overall throughput.

3.1 No Replication and Full Replication

First let us compare the throughput achieved by two extreme policies: no replication and full replication. This analysis demonstrates how replication can create synergy and boost the throughput of a server cluster.

LEMMA 1 (NO REPLICATION). *If each task is assigned to the first available idle server in a system of K servers, the throughput is,*

$$R_{NoRep} = \sum_{i=1}^{K} \frac{1}{\mathbb{E}[Y]\,\mathbb{E}[X_i]} \tag{3}$$

LEMMA 2 (FULL REPLICATION). *Suppose each task is assigned to all servers, and as soon as one replica finishes, the others are canceled. The throughput achieved by this full replication policy is,*

$$R_{FullRep} = \frac{1}{\Delta + \mathbb{E}[Y]\,\mathbb{E}[\min(X_1, X_2, \dots X_K)]} \tag{4}$$

Using Lemma 1 and Lemma 2 we can compare the two policies for any given distributions $X_1, \dots, X_K, Y$ and cancellation delay Δ. In Fig. 2 we show a comparison for the two server case, with $\Delta = 0$ and $Y = 1$. In both subplots, the service time $X_1 \sim 0.5 + \mathit{Exp}(1)$, a shifted exponential. We observe that full replication gives higher throughput when X_2 has higher variability. In the left subplot, $X_2 \sim \mathit{Pareto}(0.5, \alpha)$ and replication is better for smaller α (heavier tail). In the right subplot, X_2 is a hyper-exponential $\mathit{HyperExp}(\mu_1, \mu_2, p_2)$, that is, it is an exponential with rate μ_2 with probability p_2 and otherwise it is exponential with rate μ_1. In this case, replication is better for intermediate p_2 where X_2 has higher variability.

3.2 General Upfront Replication

Instead of replicating task at all servers, or not replicating at all we can replicate tasks at a subset of the servers. We refer to this class of policies as upfront replication policies, defined formally below.

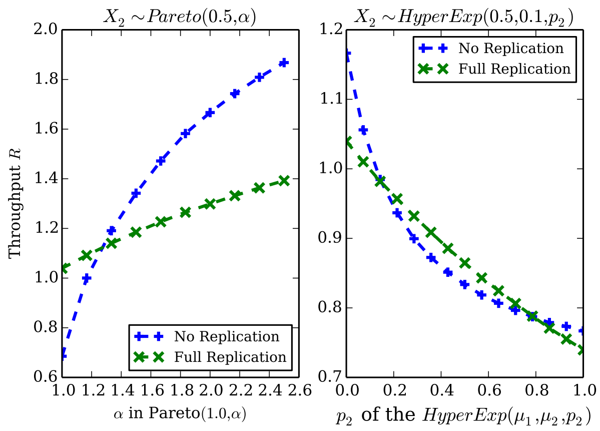

Fig. 2. Comparison of the no replication and full replication policies for $X_1 \sim 0.5 + Exp(1)$ and different X_2. When $X_2 \sim Pareto(0.5, \alpha)$, full replication is better for smaller α (heavier tail). When $X_2 \sim HyperExp(\mu_1 = 0.5, \mu_2 = 0.1, p_2)$, full replication is better for intermediate p_2.

Definition 4 (Upfront Replication). *For $h \in \mathcal{N}$, we partition of the set $[K] = \{1, 2, 3, \dots K\}$ of server indices into non-empty subsets $\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_h$, such that $\mathcal{S}_i \cap \mathcal{S}_j = 0$, and $\cup_j S_j = [K]$. When the servers in a set $\mathcal{S}_j$ become idle (they will always become idle simultaneously), replicate a task at these servers.*

The no replication policy is a special case with $\mathcal{S}_j = j$ for all $j \in [K]$. Similarly, full replication corresponds to $\mathcal{S}_1 = [K]$.

Theorem 1 (Upfront Replication). *The throughput R_{UpFr} with upfront replication at server sets $\mathcal{S}_1, \dots, \mathcal{S}_h$ is*

$$R_{UpFr}(\mathcal{S}_1, \dots, \mathcal{S}_h) = \sum_{j=1}^{h} \frac{1}{\mathbb{E}[Y]\,\mathbb{E}\left[X_{\mathcal{S}_j}\right] + \Delta}, \tag{5}$$

$$\text{where } X_{\mathcal{S}_j} = \min_{l \in \mathcal{S}_j} X_l \tag{6}$$

To maximize the throughput, we need to find the partition $\{\mathcal{S}_1, \dots, \mathcal{S}_h\}$ that maximizes (33). The number of possible partitions of a set of size K is the Bell number B_K, which is given by the recursion $B_K = \sum_{i=0}^{K-1} \binom{K-1}{i} B_i$, with $B_0 = 1$. Since B_K is exponential in K, finding the best partition can be computationally intractable for large K. However, most practical systems have limited heterogeneity, for which the problem can be tractable. For example, if the K servers are homogeneous with $X \sim F_X$, the throughput of the optimal upfront replication policy is given by the following result.

Theorem 2 (Homogeneous Servers). *For K servers with i.i.d. service times $X \sim F_X$, let r^* be the positive integer that minimizes $r(\mathbb{E}[Y]\,\mathbb{E}[X_{1:r}] + \Delta)$. The throughput achieved with upfront replication of tasks satisfies*

$$R_{UpFr} \leq \frac{K}{r^*(\mathbb{E}[Y]\,\mathbb{E}[X_{1:r}] + \Delta)}. \tag{7}$$

Equality holds in (7) if r^* divides the number of servers K.

For $Y = 1$ (no task size variability) and $\Delta = 0$, r^* is the r that minimizes $r\mathbb{E}[X_{1:r}]$. Fig. 3 illustrates the normalized expected cost per task, $r\mathbb{E}[X_{1:r}]/\mathbb{E}[X]$ versus r for

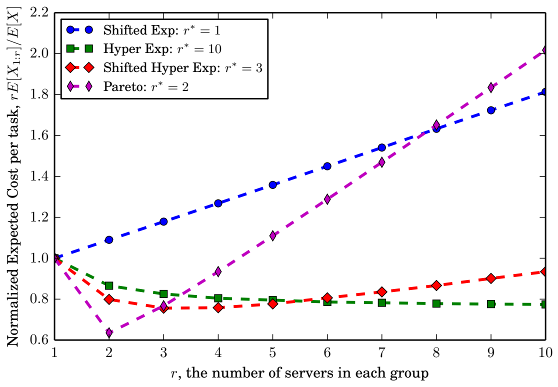

Fig. 3. Plot of $r\mathbb{E}[X_{1:r}]/\mathbb{E}[X]$ versus r for different service distributions for $K = 10$. The r^* that minimizes $r\mathbb{E}[X_{1:r}]$ is the optimal group size when $\Delta = 0$ and $Y = 1$.

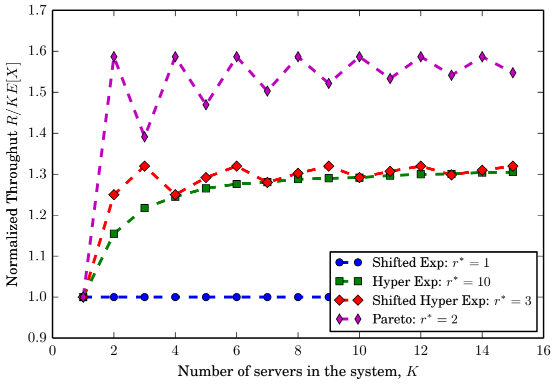

Fig. 4. Using the optimal r^* that minimizes $r\mathbb{E}[X_{1:r}]$, we partition the servers as given in Definition 5. The normalized throughput of this policy is shown here.

four different service distributions: shifted exponential $0.1 + Exp(1.0)$, hyper-exponential $HyperExp(0.6, 0.2, 0.4)$, shifted hyper-exponential $0.1 + HyperExp(1.0, 0.2, 0.4)$, and Pareto $Pareto(0.5, 1.2)$. When the tail distribution $\Pr(X > x)$ of X is log-concave (for example shifted-exponential), the optimal r is $r = 1$, whereas for log-convex X (for example hyper-exponential), $r^* = K$ is optimal. This property of log-concave (log-convex) distributions was proved in [7]. For other distributions such as shifted hyperexponential or Pareto, intermediate r can be optimal.

If r^* does not divide K, then the bound in (7) is not tight. We now propose server partitioning scheme for this case.

Definition 5 (Homogeneous Server Partitioning). *Divide K homogeneous servers into $\lfloor K/r^* \rfloor - 1$ groups of r^* each. The remaining $r^* + K \mod r^*$ servers are divided into two groups of sizes z and $r^* + K \mod r^* - z$ respectively, where*

$$z = \arg \max_{1,2,\dots,r^*} \frac{1}{\mathbb{E}[Y]\,\mathbb{E}[X_{1:z}] + \Delta} + \frac{1}{\mathbb{E}[Y]\,\mathbb{E}[X_{1:r^* + K \bmod r^* - z}] + \Delta}.$$

Applying the upfront policy (Definition 4) with this partitioning scheme yields the throughput shown in Fig. 4, for

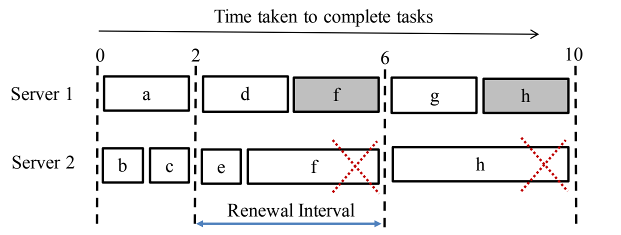

Fig. 5. Illustration of renewal instants of the system of 2 servers with the adaptive replication policy described in Example 1.

$Y = 1$ and $\Delta = 0$. For K that are divisible by r^* we achieve optimal throughput. We observe that the optimality gap of the throughput for other K is small for moderately large K. Proving the optimality of this partitioning scheme remains an open problem.

4. ADAPTIVE REPLICATION

Instead of launching replicas upfront, they could be added conditionally, only if the original task does not finish in some given time. Such policies can significantly increase the throughput, as illustrated by the example below.

Example 1. Consider a system with two servers, and assume that the task size variability $Y = 1$ and the cancellation delay $\Delta = 0$. The service times of the two servers are

$$X_1 = 2 \tag{8}$$

$$X_2 = \begin{cases} 1 & \text{w.p.} \quad (1-p) = 0.9 \\ 20 & \text{w.p.} \quad p = 0.1 \end{cases} \tag{9}$$

The throughput with full replication and no replication are

$$R_{NoRep} = \frac{1}{\mathbb{E}[X_1]} + \frac{1}{\mathbb{E}[X_2]} = 0.8448 \tag{10}$$

$$R_{FullRep} = \frac{1}{\mathbb{E}[\min(X_1, X_2)]} = 0.909 \tag{11}$$

Now consider an adaptive policy that launches a replica of a task assigned to server 2 only if it has spent more than 1 second in service. To evaluate the throughput of this policy, we consider time instants called renewals when both servers become idle. There are three types of intervals between successive renewal instants as illustrated in Fig. 5. The throughput is the expected number of tasks completed in an interval, divided by the expected interval length.

$$R = \frac{0.9 \times 0.9 \times 3 + 0.9 \times 0.1 \times 3 + 0.1 \times 2}{0.9 \times 0.9 \times 2 + 0.9 \times 0.1 \times 4 + 0.1 \times 4} \tag{12}$$

$$\approx 1.2185, \tag{13}$$

which outperforms the two extreme policies.

4.1 MDP Framework to Find the Throughput-Optimal Policy

Inspired by the motivating example above, we propose a Markov Decision Process (MDP) framework to search for the throughput-optimal policy the achieves service capacity. The state-space and actions described below satisfy the Markov property, that is, the transitions from state s to s' are depend on action $\pi(s)$, and are conditionally independent of all previous states and actions.

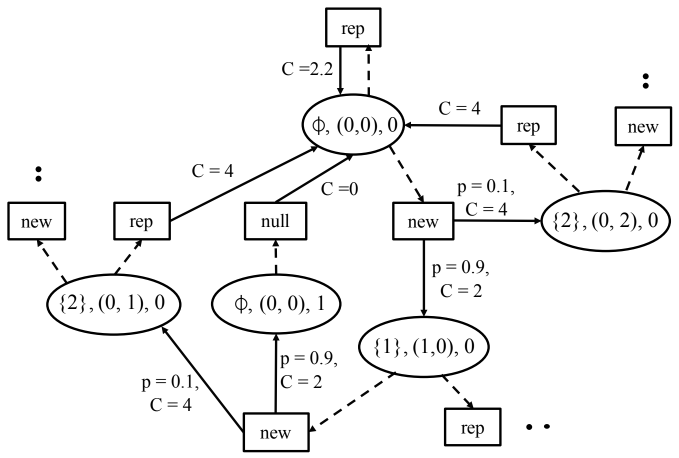

Fig. 6. Illustration of the MDP for the service distributions in Example 1. Dotted arrows correspond to the actions taken from a state and solid arrows lead to the new state resulting from the action. Parts of the MDP resulting from sub-optimal actions are omitted in this figure.

4.1.1 State-space

We denote the state evolution by $s_0, s_1, \dots s_i, \dots$ such that the system transitions to state s_i as soon as the i^{th} task departs. The state-space can be collapsed into states $[\mathcal{B}, \mathbf{t}, D_r]$ where $\mathcal{B}$ contains disjoint sets of server indices that are running the unfinished tasks in the system. For example, if $\mathcal{B} = \{\{1\}, \{2, 3\}\}$ there are two unfinished tasks in the system, one running on server 1 and another on servers 2 and 3. The vector $\mathbf{t} = (t_1, t_2, \dots t_K)$ where t_k is the time spent by server k on its current task. Since we observe the system immediately after a task departure, at least one of the elapsed times $t_1, t_2, \dots t_K$ is zero. The purpose of the D_r term is to ensure that each state transition corresponds to a single task departure. It is the number of tasks that have finished, but are still to depart. If $h > 1$ tasks exit the system simultaneously and result in the task assignment set $\mathcal{B}$ and elapsed-time vector $\mathbf{t}$, then the system goes through states $[\mathcal{B}, \mathbf{t}, h-1] \rightarrow [\mathcal{B}, \mathbf{t}, h-2] \rightarrow \dots \rightarrow [\mathcal{B}, \mathbf{t}, 0]$.

4.1.2 Actions

In each state s, denote the set of possible actions is $\mathcal{A}_s$. The scheduling policy π determines the action $a = \pi(s)$ that is taken from state s.

First note that no tasks are assigned in the exit states $s = [\mathcal{B}, \mathbf{t}, D_r]$ with $D_r > 0$. Thus, for these states, the action space $\mathcal{A}_s$ contains a single placeholder ***null*** action. The system directly transitions to $[\mathcal{B}, \mathbf{t}, D_r - 1]$.

In states $s = [\mathcal{B}, \mathbf{t}, 0]$, the scheduler can assign new tasks to idle servers (***new***), or replicate existing tasks (***rep***). For example, consider a system of 2 servers (illustrated in Fig. 6 for the service time distributions in Example 1). In states $[\{2\}, (0, t), 0]$ or $[\{1\}, (t, 0), 0]$ with $t > 0$, one server is idle while the other has spent t seconds on its current task. From the state $s = [\emptyset, (0, 0), 0]$ where both servers are idle, the ***new*** action assigns two new tasks, one to each server, and the ***rep*** action replicates a new task at both servers.

4.1.3 Cost

The cost $C(s, s', a)$ associated with a transition from state s to s' when action a is taken in state s is defined as the total time spent by the servers in that interval. Thus, the throughput-optimal policy $\pi^*_{n,r}$ is the solution to the following cost minimization problem,

$$\pi^*_{n,r} = \arg\min_{\pi \in \Pi_{n,r}} \sum_{j=0}^{\infty} C(s_j, s_{j+1}, \pi(s_j)). \tag{14}$$

For the service distributions in Example 1, we can solve the MDP. The optimal policy (illustrated in Fig. 7) is to replicate a server 2's task at server 1 only if it does not finish in 1 second. However in general, the MDP can have a large state-space even for simple service distributions. And if X_i for any i or Y is a continuous random variable for which the MDP will have a continuous state-space, which becomes even harder to solve.

4.2 The MaxRate Myopic Policy

As an alternative to solving the MDP, we propose a myopic policy called the MaxRate policy.

Definition 6 (MaxRate Policy). *From state s, the MaxRate policy chooses the action a^* that maximizes the instantaneous service rate $\hat{R}(a)$ which is defined as,*

$$\hat{R}(a) \triangleq \sum_{m=1}^{M(a)} \frac{1}{\mathbb{E}[D_m(a)]}. \tag{15}$$

where $M(a)$ is the number of unfinished tasks after taking action a, and $\mathbb{E}[D_m]$ is the expected remaining time until the departure of task j, assuming it is not replicated further.

Corollary 1. *Consider a two server system, with $Y = 1$ and $\Delta = 0$. Suppose server 1 becomes idle, and the task assigned to server 2 has spent time $t_2 > 0$ in service. Let $X_2^{rs} = (X_2 - t_2)|X_2 > t_2$ be the residual computing time. The MaxRate policy launches a replica at server 1 if*

$$\frac{1}{\mathbb{E}[\min(X_1, X_2^{rs})]} > \frac{1}{\mathbb{E}[X_1]} + \frac{1}{\mathbb{E}[X_2^{rs}]}. \tag{16}$$

and otherwise it assigns a new task to server 1.

Fig. 7 illustrates the MaxRate policy, in comparison with the FullRep and NoRep policies for the service distributions in Example 1. Observe that the throughput of the MaxRate policy is the maximum of the throughputs of the NoRep and FullRep policies. We also observe that MaxRate is suboptimal, which is not surprising because it is greedy, and oblivious to the system state resulting the action.

4.3 The AdaRep Policy

With the MaxRate policy, we dynamically find replication thresholds $t_{i\to j}$ such that a task running on server i is replicated at server j if it does not finish in $t_{i\to j}$ seconds. Based on this idea we propose another class of policies called AdaRep($\mathbf{t}$), which is directly parametrized by a replication threshold vector $\mathbf{t}$.

Definition 7 (AdaRep Policy). *Consider a vector of server indices $\mathbf{u} = (j_1, j_2, \ldots j_k)$ for $k < K$ such that a task first launched on server j_1 was later replicated on j_2, j_3 and so on. This task is replicated at an idle server i if the last server j_k has spent at least $t_{\mathbf{u}\to i}$ time on it. Otherwise it assigns a new task to the idle server. If more than one tasks satisfy the replication condition, we choose the task whose elapsed time is closest to its replication threshold $t_{\mathbf{u}\to i}$.*

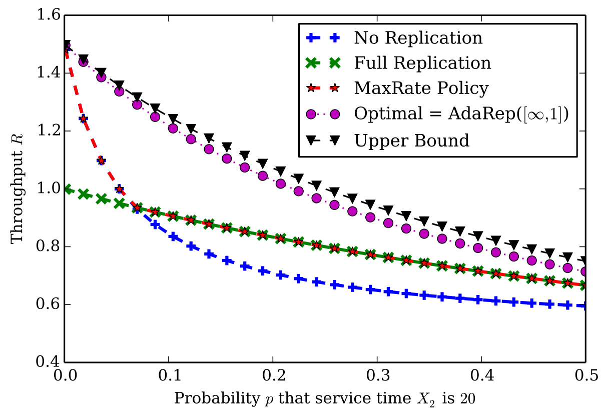

Fig. 7. Illustration of the upper bound on $R^*_{n,r}$, along with the throughputs of different replication policies. The service distributions are as defined in Example 1.

For example for $K = 2$ servers, the vector $\mathbf{t} = [t_{1\to 2},\ t_{2\to 1}]$. The optimal policy shown in Fig. 7 obtained by solving the MDP is AdaRep($[\infty, 1]$). In the next section we propose a method to choose the replication threshold vector $\mathbf{t}$.

5. BOUND ON THE SERVICE CAPACITY

To quantify the optimality gap of a policy without solving the MDP, we need an upper bound on $R^*_{n,r}$. In this section we propose such a upper bound on $R^*_{n,r}$. Drawing insights from this bound, we also propose a method to choose the replication thresholds of the AdaRep policy.

5.1 The Pause-and-Replicate System

Recall that in our problem formulation, tasks can be replicated only at time instants when one or more servers become idle. To find the upper bound on $R^*_{n,r}$, we consider a system where the scheduler is also allowed to pause ongoing tasks.

Definition 8 (Pause-and-Replicate System). *In this system, a task can be replicated at any server where it is not already running by pausing the ongoing task on that server. The paused task is resumed when the replica is served or canceled.*

For the example shown in Fig. 5, the pause-and-replicate system can pause task g at time 7 to run a replica of task h, and resume task g afterwards. Both g and h will then finish at time 9, which is 1 second faster than with the AdaRep policy without task pausing.

Claim 3. *The service capacity or maximum achievable throughput $R^*_{p,r}$ in the pause-and-replicate system is an upper bound on the service capacity $R^*_{n,r}$ of the original system.*

5.2 Evaluating the Upper Bound

In the pause-and-replicate framework, the AdaRep($\mathbf{t}$) policy can replicate a task exactly after $t_{\mathbf{u}\to i}$, instead of waiting for server i to become idle. In Theorem 3 below, we obtain a closed-form expression for the throughput $R_{p,r}(\mathbf{t})$ of the AdaRep policy for $K = 2$ servers and $Y = 1$.

Theorem 3. *In the pause-and-replicate framework, the throughput $R_{p,r}(\mathbf{t})$ of AdaRep($\mathbf{t} = [t_{1\to 2},\ t_{2\to 1}]$), with deterministic task size ($Y = 1$) can be expressed as follows. For*

$t_{1\to2} > 0$ and $t_{2\to1} > 0$,

$$R_{p,r}(\mathbf{t}) = \frac{\mathbb{E}\left[X_1^{tr}(t_{1\to2})\right] + \mathbb{E}\left[X_2^{tr}(t_{2\to1})\right]}{\mathbb{E}\left[X_1^{tr}(t_{1\to2})\right]\mathbb{E}\left[X_2^{tr}(t_{2\to1})\right](1+\gamma_{1\to2}+\gamma_{2\to1})} \tag{17}$$

where,

$$\gamma_{t_{1\to2}} \triangleq \frac{\Pr(X_1 > t_{1\to2})(\Delta + \mathbb{E}\left[\min(X_1^{rs}(t_{1\to2}), X_2)\right])}{\mathbb{E}\left[X_1^{tr}(t_{1\to2})\right]} \tag{18}$$

$$\gamma_{t_{2\to1}} \triangleq \frac{\Pr(X_2 > t_{2\to1})(\Delta + \mathbb{E}\left[\min(X_1, X_2^{rs}(t_{2\to1})\right])}{\mathbb{E}\left[X_2^{tr}(t_{2\to1})\right]}, \tag{19}$$

and $X_i^{tr}(\tau) = \min(X_i, \tau)$, the truncated part of X_i, and $X_i^{rs}(\tau) = (X_i|(X_i > \tau) - \tau)$, the residual service time after τ seconds of service.

If $t_{1\to2} = 0$ or $t_{2\to1} = 0$,

$$R_{p,r}(\mathbf{t}) = \frac{1}{\Delta + \mathbb{E}\left[\min(X_1, X_2)\right]}. \tag{20}$$

In Corollary 2 below we give the throughput expression for the special case where $t_{1\to2}$ set to infinity.

Corollary 2. *The throughput $R_{p,r}(\mathbf{t} = [\infty, t_{2\to1}])$ of the two-server pause-and-replicate system with $Y = 1$ is*

$$R_{p,r}(t_{2\to1}) = \frac{\mathbb{E}\left[X_2^{tr}\right]}{\mathbb{E}\left[X_2^{ac}\right]}\left(\frac{1}{\mathbb{E}\left[X_1\right]}\right) + \frac{1}{\mathbb{E}\left[X_2^{ac}\right]} \tag{21}$$

where, $\mathbb{E}\left[X_2^{tr}\right] = \min(X_2, t_{2\to1})$, is the truncated part of X_2, and $\mathbb{E}\left[X_2^{ac}\right]$ is the effective service time of server 2,

$$\mathbb{E}\left[X_2^{ac}\right] = \mathbb{E}\left[X_2^{tr}\right] + \Pr(X_2 > t_{2\to1})(\Delta + \mathbb{E}\left[\min(X_1, X_2^{rs}\right]), \tag{22}$$

where $X_2^{rs} = (X_2|(X_2 > t_{2\to1}) - t_{2\to1})$, the residual service time after time $t_{2\to1}$ of service.

Here is an intuitive explanation of the throughput in (21). Since server 2 is never paused, its throughput of server 2 is $1/\mathbb{E}\left[X_2^{ac}\right]$, where $\mathbb{E}\left[X_2^{ac}\right]$ accounts for the reduction in service time due to replication of tasks. For server 1, the throughput is $\zeta/\mathbb{E}\left[X_1\right]$, where $\zeta = \mathbb{E}\left[X_2^{tr}\right]/\mathbb{E}\left[X_2^{ac}\right]$, the fraction of time server 1 is not paused.

To find the optimal AdaRep policy in the pause-and-replicate framework, we find $\mathbf{t}$ that maximizes the throughput in Theorem 3. Lemma 3 below shows that for two servers, $R_{p,r}(\mathbf{t}^*)$ is in fact the service capacity $R^*_{p,r}$.

Lemma 3. *For $K = 2$ servers with $Y = 1$, there is no loss of generality in focusing on AdaRep policies to find the optimal throughput $R^*_{p,r}$ in the pause-and-replicate framework. That is, $R^*_{p,r} = \max_{\mathbf{t}} R_{p,r}(\mathbf{t})$.*

For the service distributions in Example 1, $[t^*_{1\to2}, t^*_{2\to}] = [\infty, 1]$. Thus, if a task does not finish in 1 seconds on server 2, the optimal AdaRep policy launches a replica on server 1 by pausing its ongoing task. The upper bound obtained by substituting $t^*_{2\to1} = 1$ in (21) is shown in Fig. 7.

5.3 Choosing AdaRep Replication Thresholds

We propose using the optimal $\mathbf{t}^*$ that maximizes $R_{p,r}(\mathbf{t})$ as the replication threshold vector for the AdaRep policy in the original system. This policy tries to emulate the optimal pause-and-replicate policy, under the limitation that it cannot pause ongoing tasks. In Fig. 7 we plot the throughput of AdaRep($\mathbf{t}^* = [\infty, 1]$), alongwise the upper bound. For this example, AdaRep($\mathbf{t}^*$) matches the solution of the MDP and thus it is indeed throughput-optimal. In general, we conjecture that it will give close-to-optimal throughput.

6. CONCLUDING REMARKS

Task replication is generally thought to add load to the system and reduce its service capacity. Recent works show that task replication can in fact boost the throughput of server cluster. This paper is the first attempt to find the service capacity of a multi-server system with task replication. It demonstrates how replication can not only cope with service variability, but also make more efficient use of computing resources. The search for the throughput-optimal policy involves solving an MDP, which can be hard in general. We propose two myopic replication policies: MaxRate and AdaRep that adaptively launch replicas of tasks. To quantify their gap from optimality we also obtain an upper bound on the service capacity.

Future directions include considering more actions such as killing and relaunching replicas. We also want to better understand the scaling of the proposed policies to systems with $K > 2$ servers. For systems with unknown or changing service time distributions, we plan to develop an online algorithm to dynamically adapt the replication policy.

7. REFERENCES

[1] J. Dean and L. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[2] G. Koole and R. Righter, "Resource allocation in grid computing," *Journal of Scheduling*, vol. 11, pp. 163–173, June 2008.

[3] G. Joshi, Y. Liu, and E. Soljanin, "On the Delay-storage Trade-off in Content Download from Coded Distributed Storage," *IEEE Journal on Selected Areas on Communications*, May 2014.

[4] N. Shah, K. Lee, and K. Ramchandran, "When do redundant requests reduce latency?," in *Proceedings of the Allerton Conference*, Oct. 2013.

[5] Y. Sun, Z. Zheng, C. E. Koksal, K. Kim, and N. B. Shroff, "Provably delay efficient data retrieving in storage clouds," in *Proceedings of IEEE INFOCOM*, Apr. 2015.

[6] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, E. Hyytiä, and A. Scheller-Wolf, "Reducing latency via redundant requests: Exact analysis," in *Proceedings of the ACM SIGMETRICS*, Jun. 2015.

[7] G. Joshi, E. Soljanin, and G. Wornell, "Efficient replication of queued tasks for latency reduction in cloud systems," in *Proceedings of the Allerton Conference*, Oct. 2015.

[8] G. Joshi, "Synergy via Redundancy: Boosting Service Capacity with Adaptive Replication." `https://goo.gl/549f8G`, July 2017.

[9] K. Gardner, M. Harchol-Balter, and A. Scheller-Wolf, "A better model for job redundancy: Decoupling server slowdown and job size," in *Proceedings of IEEE MASCOTS*, Sept. 2016.

APPENDIX

PROOF OF CLAIM 1. Consider a non-work-conserving scheduling policy π_{nwc} which results in task departure times $T_1(\pi_{nwc}) \leq \cdots \leq T_n(\pi_{nwc})$. Construct a work-conserving π_{wc} that follows all the actions of π_{nwc}, except the idling of servers. For example, consider a set of $r \geq 1$ servers that become idle at times $h_1, h_2, \dots, h_r$ respectively. If π_{nwc} launches replicas of a task i on these servers at times $h_1 + \epsilon_1, h_2 + \epsilon_2, \dots, h_r + \epsilon_r$, where $\epsilon_j \geq 0$ are the idle times, then π_{wc} starts the replicas at times $h_1, h_2, \dots, h_r$ instead.

We use induction to prove that $T_i(\pi_{nwc}) \geq T_i(\pi_{wc})$ for all $1 \leq i \leq n$. In both policies, all servers are available for task assignment at time 0. The departure time of the first task is

$$\begin{aligned} T_1(\pi_{nwc}) &= Y \cdot \min(X_1 + \epsilon_1, X_2 + \epsilon_2, \dots X_r + \epsilon_r) && (23)\\ &\geq Y \cdot \min(X_1, X_2, \dots X_r) && (24)\\ &= T_1(\pi_{wc}). && (25)\end{aligned}$$

This is the induction base case. For the induction hypothesis, assume that for all $i \leq n-1$, $T_i(\pi_{nwc}) \geq T_i(\pi_{wc})$. We now prove that $T_n(\pi_{nwc}) \geq T_n(\pi_{wc})$. Suppose π_{nwc} assigns task n to $r \geq 1$ servers. The times $h_1, h_2, \dots, h_r$ when these servers become idle belong to the set $\{0, T_1(\pi_{nwc}), \dots T_{n-1}(\pi_{nwc})\}$, the departure times of previous tasks. By the induction hypothesis, with π_{wc} the servers become idle earlier at times $g_1, g_2, \dots, g_r$ where $g_j \leq h_j$ for all $1 \leq j \leq r$. Thus,

$$\begin{aligned} T_n(\pi_{nwc}) &= Y \cdot \min(X_1 + h_1 + \epsilon_1, X_2 + h_2 + \epsilon_2, \dots,\\ &\qquad X_r + h_r + \epsilon_r) && (26)\\ &\geq Y \cdot \min(X_1 + h_1, X_2 + h_2, \dots X_r + h_r) && (27)\\ &\geq Y \cdot \min(X_1 + g_1, X_2 + g_2, \dots X_r + g_r) && (28)\\ &= T_{i+1}(\pi_{wc}) && (29)\end{aligned}$$

Thus, by induction, $T_n(\pi_{nwc}) \geq T_n(\pi_{wc})$ for any $n \in \mathcal{N}$. Hence by (1), $R(\pi_{nwc}) < R(\pi_{wc})$. $\square$

PROOF OF CLAIM 2. Consider tasks 1, 2, $\dots n$ run on the system of servers. If the scheduling policy is work-conserving, the total busy time of each server is exactly equal to T_n, the departure time of the last task. Since $\mathbb{E}[C]$ is defined as the total expected time spent at servers per task, by law of large numbers we have

$$\mathbb{E}[C] = \lim_{n \to \infty} \frac{KT_n}{n} = \frac{K}{R}, \tag{30}$$

where the second equality follows from Definition 1. $\square$

PROOF OF LEMMA 1. This policy is work-conserving and thus keeps all servers busy all the time. Thus, if we look at server i, the departure time of the n^{th} task assigned to that server is $T_n^{(i)}$ is the sum of n i.i.d. realizations of the service time $Y \cdot X_i$. Thus, the rate of departure of tasks from server i is,

$$R_i = \lim_{n \to \infty} \frac{n}{T_n^{(i)}} = \frac{1}{\mathbb{E}[Y]\,\mathbb{E}[X_i]}. \tag{31}$$

Adding the rates of departure from all the servers yields overall throughput as given by (3). $\square$

PROOF OF LEMMA 2. With the full replication policy, all K servers are working on the same task at any time instant. The total time spent by them on each task is,

$$\mathbb{E}[C] = K(\Delta + \mathbb{E}[Y]\,\mathbb{E}[\min(X_1, X_2, \dots X_K)]) \tag{32}$$

Then (4) follows from the result in Claim 2. $\square$

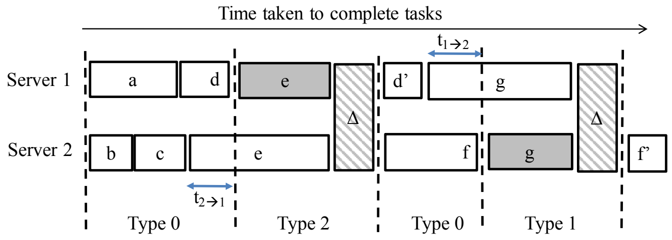

Fig. 8. Illustration of different types of intervals used to evaluate the throughput in Theorem 3. Tasks d and f are paused to launch the replicas of e and g respectively, and they are resumed when the replicas are served or canceled.

PROOF OF THEOREM 1. Incoming tasks are replicated at any one super-server, and the replicas are canceled as soon as one copy is served. Thus, the total time spent by each server in super-server $\mathcal{S}_j$ on a task is $Y \min_{l \in \mathcal{S}_j} X_l + \Delta$. The throughput of that super-server is

$$R_{\mathcal{S}_j} = \frac{1}{\mathbb{E}[Y]\,\mathbb{E}\left[\min_{l \in \mathcal{S}_j} X_l\right] + \Delta}. \tag{33}$$

The overall throughput is the sum of the throughputs of the super-servers $\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_h$, and is given by (33). $\square$

PROOF OF THEOREM 2. Let the number of servers in server $i's$ group be denoted by r_i. For example if $K = 5$ are divided into two groups of 3 and 2, then $r_1 = r_2 = r_3 = 3$ and $r_4 = r_5 = 2$. The throughput of a group with r_i servers is $1/(\mathbb{E}[Y]\,\mathbb{E}[X_{1:r_i}] + \Delta)$. If we normalize by the number of servers, the throughput per server is $1/r_i(\mathbb{E}[Y]\,\mathbb{E}[X_{1:r_i}] + \Delta)$. Summing this over all servers we have,

$$\begin{aligned} R_{UpFr} &= \sum_{i=1}^{K} \frac{1}{r_i(\mathbb{E}[Y]\,\mathbb{E}[X_{1:r_i}] + \Delta)} && (34)\\ &\leq \sum_{i=1}^{K} \frac{1}{r^*(\mathbb{E}[Y]\,\mathbb{E}[X_{1:r^*}] + \Delta)} && (35)\\ &= \frac{K}{r^*(\mathbb{E}[Y]\,\mathbb{E}[X_{1:r^*}] + \Delta)} && (36)\end{aligned}$$

If r^* divides K, then dividing servers into groups of r^* servers each gives equality in (35) above. $\square$

PROOF OF CLAIM 3. The set of feasible policies $\Pi_{n,r}$ is a subset of $\Pi_{p,r}$, the set of policies in the pause-and-replicate framework. Thus,

$$R^*_{p,r} = \max_{\pi \in \Pi_{p,r}} R(\pi) \geq \max_{\pi \in \Pi_{n,r}} R(\pi) = R^*_{n,r}. \tag{37}$$

$\square$

PROOF OF THEOREM 3. When $t_{1\to2} = 0$ or $t_{2\to1} = 0$, all tasks are replicated at both servers. Thus by Lemma 2 we get (20).

Now consider the case where $t_{1\to2} > 0$ and $t_{2\to1} > 0$. Time can be divided into three types of intervals as illustrated in Fig. 8. In Type 0 intervals, no tasks are replicated. In a Type 1 interval, both servers are serving a task that was originally launched on server 1. As soon as any one copy finishes, its replica is canceled. Then server 2 can resume its paused task, and we go back to a Type 0 interval. Similarly, in a Type 2 interval, both servers are serving a task that was originally run on server 2.

One task departs the system at the end of each Type 1 or Type 2 interval. Consider that this departure time is shifted to the end of the Type 0 preceding this Type 1 or Type 2 interval. This shift does not affect the overall throughput. Further, we rearrange the intervals to concatenate all Type 0 intervals together at the beginning of the time horizon, followed by all Type 1 and Type 2 intervals. Now the concatenated Type 0 interval can be viewed as a system of two servers running tasks according to the no replication policy, with service times $X_1^{tr}(t_{1\to 2}) = \min(X_1, t_{1\to 2})$ and $X_2^{tr}(t_{2\to 1}) = \min(X_2, t_{2\to 1})$, which are truncated versions of the original service times. Thus the rate of task completion in the concatenated Type 0 interval is

$$R_0 = \frac{1}{\mathbb{E}\left[X_1^{tr}(t_{1\to 2})\right]} + \frac{1}{\mathbb{E}\left[X_2^{tr}(t_{2\to 1})\right]}. \tag{38}$$

Since all task departures are shifted to the end of Type 0 intervals, the rate of task completion in Type 1 and Type 2 intervals is zero, that is, $R_1 = R_2 = 0$. The overall throughput can be expressed as

$$\begin{aligned} R_{p,r} &= \mu_0 R_0 + \mu_1 R_1 + \mu_2 R_2 && (39)\\ &= \mu_0 R_0, && (40)\end{aligned}$$

where R_i is the rate of task completion in concatenated interval of Type i. The weight μ_i is the fraction of total time spent in a Type i interval. The ratios μ_1/μ_0 and μ_2/μ_0 can be expressed in terms of $t_{1\to 2}$ and $t_{2\to 1}$ as follows.

$$\begin{aligned} \frac{\mu_1}{\mu_0} &= \frac{\Pr(X_1 > t_{1\to 2})(\Delta + \mathbb{E}\left[\min(X_1^{rs}(t_{1\to 2}), X_2)\right])}{\mathbb{E}\left[X_1^{tr}(t_{1\to 2})\right]} && (41)\\ \frac{\mu_2}{\mu_0} &= \frac{\Pr(X_2 > t_{2\to 1})(\Delta + \mathbb{E}\left[\min(X_1, X_2^{rs}(t_{2\to 1}))\right])}{\mathbb{E}\left[X_2^{tr}(t_{2\to 1})\right]} && (42)\end{aligned}$$

Every task originally run on server 1 spends $\mathbb{E}\left[X_1^{tr}(t_{1\to 2})\right]$ expected time in a Type 0 interval, and $\Pr(X_1 > t_{1\to 2})(\Delta + \mathbb{E}\left[\min(X_1^{rs}(t_{1\to 2}), X_2)\right])$ expected time in a Type 1 interval. Thus, the ratio μ_1/μ_0 is given by (41). Similarly we get (42).

Using (41) and (42) along with the fact that $\mu_0+\mu_1+\mu_2 = 1$, we can solve for μ_i. Substituting μ_0 in (40), we get the result in (21). □

Proof of Lemma 3. AdaRep policies replicate a task run on server 1 (or server 2) after a fixed elapsed time $t_{1\to 2}$ (or respectively $t_{2\to 1}$). Instead of fixed $\mathbf{t}$, the replication thresholds could be chosen randomly such that the threshold vector $\mathbf{t}^{(i)}$ for some $i \in [1, 2, \dots I]$ is chosen with probability $Pr(\mathbf{t} = \mathbf{t}^{(i)})$. First let us show that this does not improve the throughput.

We can divide time into I types of intervals, such that in the Type i interval, replicas are launched according to the threshold vector $\mathbf{t}^{(i)}$. We can concatenate all intervals of Type i together. Each type i interval can be further divided into three types sub-intervals as given in the proof of Theorem 3 to compute the rate of task completion in that interval. The overall throughput can be expressed as a linear combination of rates of task completion in each of these interval types,

$$\begin{aligned} R_{p,r} &= \sum_{i=0}^{I} Pr(\mathbf{t} = \mathbf{t}^{(i)}) R_{p,r}(\mathbf{t}_{(i)}) && (43)\\ &\leq \sum_{i=0}^{I} Pr(\mathbf{t} = \mathbf{t}^{(i)}) \max_{\mathbf{t}} R_{p,r}(\mathbf{t}) && (44)\\ &= \max_{\mathbf{t}} R_{p,r}(\mathbf{t}) && (45)\end{aligned}$$

where $Pr(\mathbf{t} = \mathbf{t}^{(i)})$ is the fraction of time spent in the Type i interval. The throughput of the best fixed threshold policy upper bounds each term in (43).

At any time instant the scheduler has two elapsed times available to it. AdaRep policies only consider the elapsed time of the task to be replicated. We now show that considering the elapsed time of the task that will be paused does not improve the throughput. To prove this we show that the throughput of any scheduling policy is independent of the elapsed time of the paused task. For any scheduling policy, the time horizon can be divided into three types of intervals as shown Fig. 8. Consider that the departures at the end of Type 1 and 2 are shifted to the end of the preceding Type 0 intervals. From the throughput analysis in the proof of Theorem 3 we can see that the rate of task completion in the concatenated Type 0 interval, and the fraction of time μ_0 only depend on the elapsed times $t_{1\to 2}$ and $t_{2\to 1}$. Thus, considering the elapsed times of the task to be paused does not improve the throughput. □