QACO: Exploiting Partial Execution in Web Servers

Jinhan Kim
Pohang University of Science and Technology

Sameh Elnikety
Microsoft Research

Yuxiong He
Microsoft Research

Seung-won Hwang
Pohang University of Science and Technology

Shaolei Ren
Florida International University

ABSTRACT

Web servers provide content to users, with the requirement of providing high response quality within a short response time. Meeting these requirements is challenging, especially in the event of load spikes. Meanwhile, we observe that a response to a request can be adapted or partially executed depending on current resource availability at the server. For example, a web server can choose to send a low or medium resolution image instead of sending the original high resolution image under resource contention.

In this paper, we exploit partial execution to expose a trade off between resource consumption and service quality. We show how to manage server resources to improve service quality and responsiveness. Specifically, we develop a framework, called Quota-based Control Optimization (*QACO*). The quota represents the total amount of resources available for all pending requests. *QACO* consists of two modules: (1) A control module adjusts the quota to meet the response time target. (2) An optimization module exploits partial execution and allocates the quota to pending requests in a manner that improves total response quality. We evaluate the framework using a system implementation in the Apache Web server, and using a simulation study of a Video-on-Demand server. The results show that under a response time target, *QACO* achieves a higher response quality than traditional techniques that admit or reject requests without exploiting partial execution.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Scheduling

General Terms

Algorithms, Experimentation, Performance

Keywords

Feedback control, Optimization, Partial execution, Quality, Quality profile, Response time, Scheduling, Web server

1. INTRODUCTION

Content servers such as web servers and Video-on-Demand (VoD) servers are an important part of our web infrastructure. Such servers operate under stringent response time requirements in order to attract and keep users, as large response times cause user dissatisfaction and revenue loss [14]. In addition, servers need to provide high quality responses. For example, users may stop using a VoD service if the video quality is unsatisfactory.

Prior research has focused on satisfying response time requirements in web servers. For example, a common approach to meet a response time target is to set a limit on the length of the incoming request queue: when the queue is full, new requests are dropped upon arrival. However, choosing a static queue length limit in admission-control mechanisms [33] leads to several undesirable situations, such as dropping too many requests and causing inconsistent user experiences. Moreover, although well-designed web servers can effectively avoid being persistently overloaded (e.g., by dynamically scaling server provisioning), turning servers back to normal states takes time [22] and hence, transient periods of overloads are inevitable due to unexpected bursty load spikes [29].

These factors suggest the need for self-managed systems that can adapt to the workload changes for timely responses. Feedback control has been successfully applied for satisfying response time requirements in [1,17]. For example, prior work [6,8,24,25,31] adjusts the queue length limit dynamically (or request drop rate) according to the feedback on response time: decrease the queue length limit for admission control when the measured response time exceeds the target, and vice versa. While this type of dynamic admission control is effective for the classic "binary" request model (i.e., the server either processes the request by returning a complete response or drops it with a null response), it fails to exploit *partial execution* that many web servers support.

With partial execution, a request may have several partial results with different qualities depending on the amount of the received resources. For example, a web server can employ content adaptation (e.g., text-only, low-resolution image, and high-resolution image) to cope with varying workloads. A VoD server can also support partial execution by streaming videos of different qualities to users. Partial execution provides the flexibility of trading more processing resources for a better quality, but it presents new challenges in managing server resources: The resource manager needs to dynamically decide the amount of processing resources for each request, unlike simply "accepting" or "rejecting" a request.

To exploit partial execution, we propose a quota-based control optimization (*QACO*) framework to improve response quality while satisfying response time requirements for web servers. The quota is defined as the total execution time (which is a proxy of the total amount of processing resources) for all pending requests, and it is dynamically adjusted based on the workload. To quantify the response quality, we use a monotonically-non-decreasing discrete quality function that maps the allocated processing resources to a quality value indicating how "well" the request is served (see Section 2.1 for more

CAC'13, August 5–9, 2013, Miami, FL, USA.

details). The *QACO* model consists of two components: A control module to adapt the quota so the system can meet its desired response time, and an optimization module that decides the execution time of each request (subject to the given quota) to improve the total response quality.

For the control module, we use a feedback controller to dynamically adjust the quota. We employ an integral controller to have zero steady-state error with low runtime overhead. When the measured response time is larger than the desired target, we decrease the quota so requests would get shorter processing times, and thus their waiting time and response time decrease. Similarly, when the measured response time is smaller than the desired target, we increase the quota so that requests get more processing time and improve their response quality.

Designing an optimization module in this environment is challenging: The discreteness of quality functions requires combinatorial optimization to maximize the total response quality given a quota. We develop two optimization modules for two different types of scheduling scenarios. (1) When a scheduler does not know the service demand of all waiting and running requests, we develop a heuristic algorithm (*QACO-U*) that exploits the shape of the quality function, reserving time for all requests to complete at least an initial version that yields the highest quality gain per unit processing time. (2) When a scheduler knows the service demand of all pending requests, the optimal scheduling problem is NP-hard, and we present an efficient algorithm (*QACO-K*) that achieves a total quality arbitrarily close to the optimal.

We demonstrate the practical feasibility of *QACO* by implementing and evaluating it in the Apache web server. Here, the Apache web server does not know the service demand of all pending requests, therefore we apply *QACO-U*. Our experimental results show significant benefits of using quota-based control over queue-length-based control. In particular, *QACO* meets the desired response time target and achieves the highest service quality.

Furthermore, we perform a simulation study for a VoD server to evaluate *QACO-K* as service demand of all video requests is known in advance, and the server can choose at which bit-rate of the response on a per request basis. In our simulation study, we observe consistent findings: *QACO* outperforms the queue-based approach with improved service quality. Moreover, we show that, besides mean response time, *QACO* can also effectively bound high-percentile response time to meet the target. The high-percentile response time guarantees consistently fast response, and many commercial services specify their Service Level Agreement (SLA) using both the mean and high-percentile response time [9]. In addition, we compare *QACO-K* and *QACO-U* to demonstrate how the scheduler exploits the knowledge of service demands to make better decisions.

The contributions of this work are the following: (1) We propose *QACO*, a quota-based control optimization framework for web servers with partial execution. (2) We introduce a control module to meet the response time requirements, and two optimization modules to improve response quality with known and unknown service demands. (3) We implement *QACO* in a web server and we show the benefits experimentally. (4) We conduct a simulation study to evaluate *QACO* in a VoD server.

The paper is organized as follows. Section 2 discusses workload characteristics. Section 3 describes the *QACO* framework and its two components: the control and the optimization modules. Section 4 describes the implementation and experimental evaluation in a web server. Section 5 presents the results of a simulation study using a VoD server. We discuss related work in Section 6 and present our conclusions in Section 7.

2. WORKLOAD CHARACTERIZATION

In this section, we present a key feature of web servers, i.e., partial execution, and then formulate the scheduling problem.

2.1 Partial Execution

Partial execution enables the flexibility of trading more processing resources for better results. Many applications, such as web search, webpage browsing, and video streaming, support partial execution. For example, a web search engine receives queries from clients and returns the matched documents within a short deadline. A search query has multiple acceptable answers. With more processing time, the search engine will match and rank more web pages, producing a progressively better response [15]. Content adaptation (or content fidelity) provides another form of partial execution by providing multiple service quality options that can be adaptively selected (either by the servers or by clients) [2]. For example, YouTube provides different video streaming qualities such as 360p, 480p and 720p that can be chosen based on congestion level at the server side and/or Internet speeds (where "p" stands for "progressive").

With partial execution, the service quality of a request is quantified by a function, referred to as *quality function*, in terms of the amount of processing resources used to process the request. While various types of system resources (e.g., memory, CPU) are required to process a request, we use *processing time* as a proxy of the consolidated processing resources. Thus, mathematically, a quality function $f : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ maps the used processing time to a quality value gained by executing the request. Although in general different applications have different quality functions, a typical quality function satisfies the following three properties.

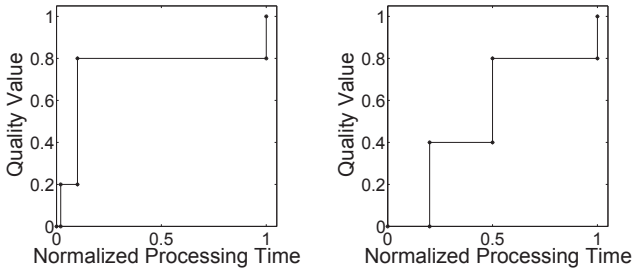
• **Monotonicity:** It is natural that quality function is monotonically non-decreasing: service quality of a request always improves or stays the same with more processing resources.

• **Discreteness:** Quality functions associated with many best-effort web applications are discrete (e.g., a staircase shape), since a request can only be processed by exploring a finite number of algorithms or data sets, thereby leading to discrete quality values. For example, a VoD server typically streams videos with a small number of different resolutions; a web server may return web pages with a limited number of choices (such as text only, low-resolution image and high-resolution image).

• **Diminishing returns:** In many applications, quality function exhibits diminishing returns: allocating more processing time to a request leads to a decreasing quality improvement. In other words, the quality improvement is significant when we start to process a request, while it gradually becomes diminishing during the processing, as can be explained by considering the example of VoD servers using scalable video coding (a popular video encoding technique used in H.264/MPEG-4 AVC standard): sending higher-rate bitstreams increases the video quality but the quality increment gradually diminishes [34]. Diminishing returns are also common in a variety of alternative domains, notably in game theory for defining continuously concave utility functions [28].[1] In fact, with continuous extension, quality function in our study also becomes continuous and concave, and for this reason, we say that the quality function is a *concave-type* discrete function.

We illustrate two examples of quality functions in Figure 1(a) and (b) where the y-axis is the quality value (or reward value) that represents the amount of satisfaction of users and the x-axis is the normalized processing time (i.e., the ratio of actual processing time to the maximum processing time required for returning the full-quality webpage). In Figure 1(a), users fully satisfy (quality value 1) when

[1]Discrete utility functions that exhibit diminishing returns are called *submodular* functions under some mild conditions [19, 28]



(a) Web Server. (b) VoD Server.

Figure 1: Example of Response Quality Function.

they receive a full-quality webpage with high-quality images from the web sever, while they only get some satisfaction (e.g., quality value 0.2) when they receive text-only reply.

Now, we briefly explain the benefits of partial execution while the detailed algorithm is presented in the next section. Without partial execution, a scheduler can either fully process a request or reject it. For example, YouTube servers may not handle all the requests for high-resolution videos in the event of overloading and, as a consequence, user requests are either buffered for a long time prior to being processed or rejected. By contrast, with partial execution, it could be desirable to return lower-resolution videos but to satisfy all the users under overloading such that total quality of responses is higher and on average, users receive better service (because partially processing two requests results in a higher total quality than fully processing one request and dropping the other, due to the concave-type quality functions). Therefore, given limited resources and response time constraint, partial execution opens new opportunities to improve response quality.

2.2 Problem Formulation

We first present the scheduling problem formulation as follows. We denote the set of pending requests (or jobs) by $\mathcal{J} = \{J_i | i = 1, 2, \ldots, N\}$. Each job $J_i \in \mathcal{J}$ is characterized by its *service demand* (a.k.a. total work) w_i and supported set of processing times $\mathcal{A}_i = \{a_{i,m} | m = 1, 2, \ldots, M_i\}$, where $0 = a_{i,1} < a_{i,2} < \ldots < a_{i,M_i} = w_i$ and 0 represents zero processing time (i.e., rejecting a request). We consider the mean response time constraint. Thus, the request does not have a deadline, and the arrival time for each request is not required when making scheduling decisions, although it is needed for monitoring the response time.

Let $p_i \in \mathcal{A}_i$ denote the *processing time* that job J_i receives before being returned to the user. For notational convenience, we also use the vectorial expression $\mathbf{p} = (p_1, p_2, \ldots, p_N)$. Without loss of generality, job J_i has a *quality function* $f_i : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ that maps the job completion ratio (i.e., p_i/w_i) to a quality value gained by processing the job. The objective of the scheduler is to maximize the total quality of all the jobs, i.e., $\max_{\mathbf{p}} \sum_{i=1}^{N} f_i(p_i/w_i)$, by optimally choosing processing times for the jobs subject to the *mean* response time constraint. This is a combinatorial optimization problem even though the service demand of each job is known, since each job only supports a discrete set of processing times.

3. QUOTA-BASED CONTROL OPTIMIZATION FRAMEWORK

This section presents the quota-based control optimization framework (*QACO* in short) for delay-sensitive web services with partial execution. *QACO* consists of two integrated modules: (1) *control* module, which applies feedback control to adjust the quota (defined as the total processing time) to meet the mean response time constraint, and (2) *optimization* module, which is a scheduling algorithm exploiting partial execution to maximize total response quality for a given quota. The quota-based framework facilitates the system design by decomposing the two major goals of a web server system (i.e., satisfying response time constraint and achieving high service quality) into the control and optimization module, such that each module can be managed separately given the output of the other module. Next, we provide details of designing the control and optimization modules.

3.1 Control Module

In this subsection, we describe the control module that controls the quota, i.e., the total processing time (or total amount of processing resources) that we plan to allocate to all the pending requests.

The control module takes the observed mean response time as its feedback and then adjusts the quota value accordingly in order to meet the response time target. While there are various types of controllers (e.g., model predictive control), we consider in this paper an effective and well-known control mechanism — integral control, which adjusts the value of control variable based on the difference between the observed output and the reference output. The control function is expressed as follows:

$$u(k) = u(k-1) + K_I e(k),$$

where $k \in \mathbb{N}$ represents the time step at which we update the quota (e.g., after processing each request); $u(k)$ is the output of the integral controller (which is also the control variable of the system) at time step k. The tracking error $e(k)$ is the difference between the observed output and the reference output, i.e., $e(k) = y_{ref} - y_(k-1)$, and the controller parameter K_I defines the control value adjustment relative to to the tracking error. In our system, the control variable u is the quota, and the output y is the mean or high-percentile response time depending on our control objective.

We apply integral control because of its two key advantages. First, it has a zero steady-state error, which allows the system to meet its desired SLA when the system becomes stabilized. Second, it is computationally efficient, which is an important property in delay-sensitive web systems, as the integral controller incurs a negligible overhead allowing recalculation of the quota upon the arrival or departure of each request. While in theory the convergence rate of integral control may be slow for certain initial values, we observe in almost all of our experiments that the integral control yields a stabilized quota within a few seconds, which is quick enough to adapt to traffic spikes in practical systems.

3.2 Optimization Module

The optimization module is a scheduling algorithm that takes as inputs a set of pending requests as well as the quota determined by the control module, and then assigns a processing time to each request with the objective of maximizing total response quality. The design of the optimization module depends on the request quality function and other application specific constraints. We do not intend to enumerate all optimization modules to cover all scenarios. Instead, we present two optimization modules for jobs with *concave-type* discrete quality functions (as described in Section 2.1), because they are popular in practice and used by many applications such as web servers and VoD servers. In particular, the two optimization modules consider the cases where the service demand of a job is unknown and known, respectively.

3.2.1 Unknown service demand

We first present an optimization algorithm with discrete concave-type quality functions and with unknown service demand in Algo-

rithm 1 (called *QACO-U*). Web server is an example application with discrete concave-type quality function (Figure 1). Moreover, web server does not know the service demand of all waiting requests in advance.[2] *QACO-U* takes the following data as inputs: (1) quota, defined as the amount of processing time available for all ready requests in the queue, (2) queue length, (3) expected service demand of requests – although precise service demand of a request is unknown before job execution, we can compute expected service demand of requests based on execution history, and (4) the supported set of the processing times for the running request.

QACO-U processes requests in the FIFO order and decides the assigned processing time of the first job in the FIFO queue based on the load and the quota. To improve total response quality, the scheduler prefers running the part of requests with a higher quality improvement. Given a concave-type quality function, processing the early portion of a request has a higher quality gain than its later portion. Therefore, the key idea of *QACO-U* is to prevent jobs at the beginning of the queue from consuming the entire quota and starving later requests, such that each request has a fair opportunity to be processed (at least for the high-return part). To achieve this goal, *QACO-U* applies two techniques. (1) Equi-Partitioning (EQ): When the system is heavily loaded, *QACO-U* performs EQ to reserve a fair share of processing time for waiting requests (in Line 2). With a concave-type quality function, giving each job a fair chance to complete at least its high-return part improves the overall quality. (2) Reservation (RESV): In a lightly loaded case, *QACO-U* performs RESV to reserve the expected service demand for the queuing requests and allocates the remaining time to the current running job (in Line 3). RESV gives the long requests a chance to finish if they will not impact short ones. Note that *QACO-U* does not use precise information on the quality function of all requests to make a scheduling decision, because, when request service demand is unknown, its quality function is unknown in advance either. *QACO-U* only requires the quality function to be concave-type as it exploits the benefits of concavity. Nonetheless, *QACO-U* implicitly assumes that all the jobs share the same concave-type quality function, because it is not feasible to differentiate jobs without knowing their service demand.

QACO-U does not need a load threshold to decide if it should use the result from EQ or RESV. During light loads, we want to estimate the processing time using RESV, and its processing time is larger than the one produced by EQ. During heavy loads, we want to use EQ, and its processing time is larger than the one produced by RESV. Therefore, selecting the larger between these two gives the assigned processing time (in Line 4). Based on the assigned processing time, *QACO-U* returns the size of the request with the highest quality while requiring processing time less than or equal to the assigned processing time (in Line 4 - Line 8).

In summary, *QACO-U* considers partial execution and concave-type quality function by seamlessly combining EQ and RESV to improve response quality. Moreover, it possesses three desirable properties. It does not require precise information on the service demand and quality function of requests. It does not incur preemption overhead. And, it is computationally efficient with the computational complexity $O(1)$, independent of the number of ready jobs. We further evaluate its performance in Apache web server in Section 4.

3.2.2 Known service demand

We now develop an efficient algorithm to maximize the total response quality with known service demand in Algorithm 3 (called *QACO-K*). Leveraging branch-and-bound techniques [4], *QACO-K* is computationally-efficient and can assign a close-to-optimal processing time to each pending request subject to the quota constraint.

[2]Although service demand of a request is known once the server starts to process a request, a web server does not know the service demand of the waiting requests.

Algorithm 1 *QACO-U*: optimization algorithm with unknown service demand

Inputs:
$\mathcal{J} = \{J_i | i = 1, \ldots, N\}$: set of pending jobs
T: quota (total processing time for pending jobs)
$\bar{w}$: mean service demand of jobs
J_1: first job in the queue
$\mathcal{A}_1$: the supported set of possible processing times for J_1 **Pseudo code:**
1: $qLen = |\mathcal{J}|$ {queue length}
2: $EQ = T/qLen$ {Equi-partitioning}
3: $RS = T - (qLen - 1) \times \bar{D}$ {Reservation}
{assign processing time for the first job J_1 at ready queue}
4: $p_1 = \max(EQ, RS)$
{discretize the processing time p_1 of job J_1}
5: **for all** $w \in \mathcal{W}$ **do**
6: $p_1 = \max\{a_{1,m} | a_{1,m} \leq p_1, a_{1,m} \in \mathcal{A}_1\}$
7: **end for**
8: **return** p_1

Algorithm 2 Greedy

Inputs:
$\mathcal{J} = \{J_i | i = 1, \ldots, N\}$: set of pending jobs
T: quota (total processing time for pending jobs)
w_i: service demand of job J_i
f_i: quality function of job J_i
$\mathcal{A}_i$: supported set of partial processing times of J_i **Pseudo code:**
1: Initialize $p_i = w_i$, for $J_i \in \mathcal{J}$
2: **while** $\mathbf{p} \neq \mathbf{0}$ **and** Constraint (2) is not satisfied **do**
3: $\Omega \leftarrow \{J_i \mid J_i \in \mathcal{J}, p_i > 0\}$
4: $\Delta p_i \leftarrow \arg\max_{p \in \mathcal{A}_i} (p < p_i), \forall J_i \in \Omega$
5: $i = \arg\min_{i \in \Omega} \{f(p_i/w_i) - f(\Delta p_i/w_i)\}$
6: $p_i \leftarrow \Delta p_i$
7: **end while**
8: **return** $\mathbf{p}^* = \mathbf{p}$

Given a quota T determined by the control module, we first formulate the problem of optimally assigning processing time for each request as follows.

$$\text{OPT-K:} \quad \max_{\mathbf{p}} \sum_{J_i \in \mathcal{J}} f_i(p_i/w_i) \tag{1}$$

$$s.t., \quad \sum_{J_i \in \mathcal{J}} p_i \leq T, \tag{2}$$

$$p_i \in \mathcal{A}_i, \forall J_i \in \mathcal{J}, \tag{3}$$

where (2) is the quota constraint, and $\mathcal{A}_i$ is the supported set of (discrete) partial processing times for job J_i.

Because of the discrete processing time constraint (3), OPT-K falls into combinatorial optimization, for which the computational complexity increases exponentially with the number of requests [5]. A simple approach to solving OPT-K is: (1) by replacing "$p_i \in \mathcal{A}_i$" with $p_i \in [0, w_i]$ (where w_i is the service demand for job J_i) and continuously extending the quality functions for all $J_i \in \mathcal{J}$ to reformulate OPT-K as a convex problem, we apply standard techniques (e.g., interior methods [5]) to solve the relaxed problem, denoted by OPT-RLX; and (2) we round the obtained continuous processing time $p_i^* \in [0, w_i]$ to the closest value in $\mathcal{A}_i$ that is no greater than p_i^*, for $J_i \in \mathcal{J}$. While this approach automatically satisfies the quota constraint, the response quality may be far from the global optimum. Below, we propose an efficient branch-and-bound algorithm in the following four steps to yield an arbitrarily close-to-optimal solution. With the flexibility of adjusting the accuracy and complexity,

the algorithm can be easily embedded in various systems (e.g., web or VoD servers) while incurring a small overhead.

1) Decomposition. We first decompose OPT-K into $M_1 = |\mathcal{A}_1|$ sub-problems, indexed by OPT-K$_1$, OPT-K$_2$, $\cdots$, OPT-K$_{M_1}$. Each sub-problem OPT-K$_m$ is expressed as follows:

$$\text{OPT-K}_m: \quad \max_{\mathbf{p}} \sum_{J_i \in \mathcal{J}} f_i(p_i/w_i) \tag{4}$$

$$s.t., \quad \sum_{J_i \in \mathcal{J}} p_i \leq T, \tag{5}$$

$$p_i \in \mathcal{A}_i, \forall J_i \in \mathcal{J} \text{ and } i \neq 1, \tag{6}$$

$$p_1 = a_{1,m}, \tag{7}$$

where we fix $p_1 = a_{1,m}$ as the m-th supported processing time for job i and maximize the total response quality over $\mathbf{p}\backslash\{p_1\}$.[3] After solving all the M_1 sub-problems, we can select one sub-problem (say, OPT-K$_m$) that yields the maximum response quality and then, combined with $p_1 = a_{1,m}$, we obtain the optimal processing times $\mathbf{p}^*$. Each sub-problem itself is an combinatorial problem and can be further decomposed into multiple smaller problems by fixing the processing time for another request. Thus, the original problem can be solved recursively, which serves as the basis for applying the branch-and-bound technique.

2) Lower and upper bounds. By relaxing the processing time constraint "$p_i \in \mathcal{A}_i$" with $p_i \in [0, w_i]$ for all $J_i \in \mathcal{J}$ and solving the relaxed problem OPT-RLX using convex standard optimization techniques [5], the resulting response quality is an upper bound on that of problem OPT-K (i.e., the maximum response quality in OPT-K is no greater than that obtained by solving the relaxed problem with "$p_i \in [0, w_i]$ for all $J_i \in \mathcal{J}$" as the processing time constraint).

To find a lower bound on the maximum response quality in OPT-K, we propose a greedy algorithm, as described in Algorithm 2, which never outperforms the optimal solution to OPT-K in terms of the total response quality. In the greedy algorithm, all the processing times are initially chosen to be their maximum values (i.e., $p_i = w_i$, for $J_i \in \mathcal{J}$). If the quota constraint $\sum_{J_i \in \mathcal{J}} p_i \leq T$ is not satisfied, we greedily decrease the processing time such that the total quality decrease is minimum (i.e., Line 3–6 in Algorithm 2). Repeat this process until all the processing times decrease to zero or the quota constraint is satisfied.

Next, we define the following notations that facilitate the description of our branch-and-bound algorithm.

DEFINITION 1. *$UB(\mathcal{P})$ is the maximum quality obtained by solving OPT-RLX with the constraint $\mathcal{P}$ as its additional input. $LB(\mathcal{X})$ is the total quality obtained by using the proposed greedy algorithm (i.e., Algorithm 2) with the additional constraint $\mathcal{P}$ as its additional input.*

We explain Definition 1 using an example. If $\mathcal{P} = \{p_1 = a_{1,m}\}$ where $a_{1,m} \in \mathcal{A}_1$, we compute $UB(\mathcal{X})$ by solving OPT-RLX with an additional constraint of $p_1 = a_{1,m}$. The variation of OPT-RLX with additional constraints specified by $\mathcal{P}$ is still convex and can be efficiently solved using standard convex optimization techniques [5]. Similarly, we compute $LB(\mathcal{X})$ using the proposed greedy algorithm with an additional constraint of $p_1 = a_{1,m}$. We use $UB(\varnothing)$ and $LB(\varnothing)$ to represent the total quality obtained by solving the original problem OPT-RLX and by using the greedy algorithm without additional constraints, respectively.

3) Fixing rule. A core component of branch-and-bound algorithms is the "fixing" rule, which determines the next decision variable to be fixed. We define the "fixing" rule as follows.

[3]We can also fix the processing time for any job other than J_1.

Algorithm 3 *QACO-K*: optimization algorithm with known service demand

Inputs:
$\mathcal{J} = \{J_i | i = 1, \ldots, N\}$: set of pending jobs
T: quota (total processing time for pending jobs)
w_i: service demand of job J_i
f_i: quality function of job J_i
$\mathcal{A}_i$: supported set of partial processing times of J_i
Pseudo code:
1: Initialize: $t \leftarrow 0$, $\mathcal{P}_0 \leftarrow \varnothing$, set of leaves of a single-node tree $\mathcal{A} \leftarrow \mathcal{P}_0$
2: Compute lower and upper bounds: $L_0 = LB(\mathcal{P}_0)$ and $U_0 = UB(\mathcal{P}_0)$
3: **while** $U_t - L_t > \epsilon$ **or** $t < IterateMax$ **do**
4: Choose the splitting node: $\mathcal{P}^* = \arg\max_{\mathcal{P} \in \mathcal{A}} UB(\mathcal{P})$
5: Choose the request index to fix: $i = next(\mathcal{P}^*)$
6: Generate M_i new constraint sets:
$\mathcal{P}_{t+1}^1 = \mathcal{P}^* \cup \{p_i = a_{i,1}\}, \cdots, \mathcal{P}_{t+1}^{M_i} = \mathcal{P}^* \cup \{p_i = a_{i,M_i}\}$
7: Update the set of leaves:
$\mathcal{A} \leftarrow (\mathcal{A}\backslash\{\mathcal{P}^*\}) \cup \{\mathcal{P}_{t+1}^1\} \cup \cdots \cup \{\mathcal{P}_{t+1}^{M_i}\}$
8: Compute upper and lower bounds for M_i new constraint sets:
$LB(\mathcal{P}_{t+1}^1), \cdots, LB(\mathcal{P}_{t+1}^{M_i}), UB(\mathcal{P}_{t+1}^1), \cdots, UB(\mathcal{P}_{t+1}^{M_i})$
9: Update global upper and lower bounds:
$L_{t+1} = \min_{\mathcal{P} \in \mathcal{A}} LB(\mathcal{P})$ and $U_{t+1} = \max_{\mathcal{P} \in \mathcal{A}} UB(\mathcal{P})$
10: $t \leftarrow t + 1$
11: **end while**
12: Choose the best constraint set thus far:
$\bar{\mathcal{P}} = \max_{\mathcal{P} \in \mathcal{A}} LB(\mathcal{P})$
13: **return** $\mathbf{p}^*$ achieved by the greedy algorithm (i.e., Algorithm 2) with $\bar{\mathcal{P}}$ as the constraint

DEFINITION 2. *$next(\mathcal{P})$ is the index of the request that the proposed greedy algorithm selects next to update the processing time given the constraint set $\mathcal{P}$ as the input.*

In essence, we select and fix the processing time for a request which, if decreased to the next smaller value out of the supported partial processing times, results in the minimum quality decrease.

4) Algorithm. We describe our branch-and-bound algorithm in Algorithm 3. The parameter $IterateMax$ is the maximum number of iterations selected based on the desired accuracy and the problem scale. The algorithm generates a tree, where each node represents a constraint set and all the leaf nodes are stored in the set $\mathcal{A}$. The algorithm begins with an empty constraint set $\mathcal{P}_0 = \varnothing$ as the parent node of the tree. In each iteration, we choose a leaf node and split it into new leaf nodes, each of which represents a new constraint set with an additional processing time p_i fixed to be one of the permissible values in $\mathcal{A}_i$. In the splitting process (i.e., Line 4–7), we split the node that corresponds to the constraint set resulting in the maximum response quality (obtained by solving OPT-RLX with an additional constraint specified by the node to be split). The reason behind our splitting process is that after splitting this node, the global upper bound will likely be decreased, while splitting any other node keeps the global upper bound unchanged and hence the algorithm does not shrink the gap between the global upper and lower bounds. Besides the maximum number of iterations, another stopping criterion is the difference between the global upper and lower bounds. Specifically, if $U_t - L_t$ is no greater than a sufficiently small positive number ϵ, it is guaranteed that the solution obtained using the greedy algorithm with an appropriate constraint set is close-to-optimal. Therefore, by increasing $IterateMax$ and using a sufficiently small positive number ϵ, *QACO-K* yields an arbitrarily close-to-optimal solution, while the global optimality is achieved at the expense of increasing the computation cost.

Complexity. Although finding the global optimal solution to OPT-K is NP-hard due to the constraints of discrete processing times, the beauty of branch-and-bound algorithm is that it typically converges much faster, especially by setting "$U_i - L_i \leq \epsilon$" rather than strictly

enforcing "$U_i - L_i = 0$". In fact, with "$U_i - L_i \leq \epsilon$" as the stopping criterion, the number of iterations required for convergence is upper bounded, and in practice, the actual number of iterations is typically even much smaller than the upper bound (which is also observed in our calculations). The analysis of convergence rate is beyond the scope of our paper, and interested readers are referred to [4] for a detailed treatment. In practice, we can also group the processing times for the same type of requests that share a common quality function into one decision to speed up *QACO-K*. Even if the computational time is strictly constrained, we can set a looser stopping criterion such that *QACO-K* can return a reasonably *good* solution within a few iterations. In the extreme case (with a sufficiently large ϵ and $IterateMax = 1$), *QACO-K* reduces to the greedy algorithm described in Algorithm 2, whose complexity is comparable to *QACO-U*. Therefore, *QACO-K* can be easily adapted to practical systems with various computational capabilities.

4. APACHE EXPERIMENTAL EVALUATION

This section presents our implementation and evaluation using the Apache web server. We use *QACO-U* because Apache web server does not know the service demand of the waiting requests in kernel queues. The servers know the request details once it starts processing it. Our results show that *QACO-U* outperforms the existing Apache scheduling as well as and traditional queue-based scheduling: *QACO-U* meets the response time target and achieves higher result quality.

4.1 Implementation

We use an Apache server, servicing HTTP requests for webpages with varying sizes. To obtain realistic distributions of webpages, we obtain the size distribution of 100 randomly crawled webpages from CNN.com and generate a workload following the same distribution as shown in Figure 2. We implement our techniques using Apache version 2.2.19 on Windows. Specifically, we modify the function ap_parse_uri() to choose the right version of the content depending on the feedback from the controller and the scheduler when *QACO-U* is enabled.

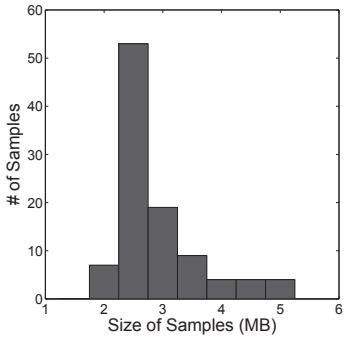


Figure 2: Size distribution of 100 CNN webpages. The mean is 2.9 MB and the standard deviation is 0.68.

For each webpage, our algorithm supports partial results, e.g., text-only or with low-resolution images. We consider three versions of varying size and quality in this evaluation–original, medium, and small (O, M, and S respectively). We point out that our approach is not dependent on the number of available response versions. We set the size of M as $\frac{1}{10} * O$ and the size of S as $\frac{1}{50} * O$, and its quality to be $\frac{8}{10}$ and $\frac{2}{10}$ of original requests. As the purpose of this study is to show the benefit of the algorithms for a given quality function, we leave a more sophisticated modeling of user satisfaction as future work. We also demonstrate that our algorithm is applicable to different quality functions in Section 5.4.

To generate the server workload, we use Httperf to send HTTP requests for webpages. Our implementation uses release version 0.9.0. We use an *open-loop* workload generator for measuring the server performance, which generates a stream of requests to the server. We choose to use this over a *closed-loop* workload generator to avoid having a fixed population of users [30].

4.2 Experimental Setup

We compare *QACO-U* to four baseline algorithms: three algorithms based on existing Apache approaches and one queue-based algorithm, as follows:

- *QACO-U* estimates possible quota of the server and decides appropriate partial results without prior knowledge of service demands of requests.
- *Apache* models Apache without support for partial results.
- *Apache*$_M$ models Apache always returning medium size of partial results.
- *Apache*$_S$ models Apache always returning small size of partial results.
- *Queue*$_O$, which applies feedback control to drop requests (or send zero replies) when the server is busy. To decide whether the server is busy or not, we use a moving mean value of request response time. When the moving mean response time exceeds the response time target T, *Queue*$_O$ sends zero reply to reduce the response time of requests.

All experiments are conducted on a server with Intel Core i7-2600K 3.40GHz CPU and 16GB RAM running the Apache Web Server on the Windows Server operating system. For the client we use a computer with an Intel Core i7 930 2.80GHz CPU/24GB RAM and Debian GNU/Linux 6.0 running Httperf. The client machine is never the bottleneck in our experiments.

4.3 Results

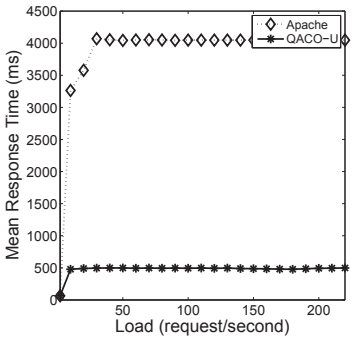
4.3.1 Comparison to Apache scheduling

We first compare *QACO-U* to *Apache*, which is Apache without support for partial execution. We first warm up the server for 10 seconds, and then client sends a fixed number of requests. We vary the load from 2 r/s to 220 r/s, and run each load for 2400 seconds. We compute the quality value and the mean response time at each load.

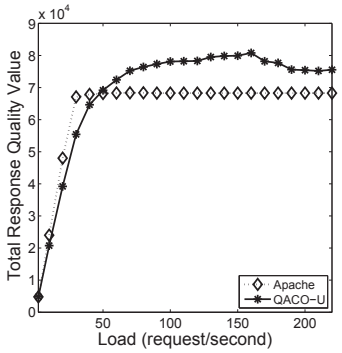
First, we observe the response time of *QACO-U*, to verify that it can control the response time to meet the response time target $T = 500$ms over varying load levels.

Figure 3(a) shows the mean response time of *Apache* and *QACO-U*. As the Apache web server does not have a control module for response time, the response time of *Apache* starts to increase from 10 r/s. After 10 r/s, the server is overloaded and a new request waits until *Apache* finishes processing earlier requests. As a result, the response time increases up to 4000ms, which significantly exceeds the target response time. In clear contrast, *QACO-U* effectively controls the response time to meet the target $T = 500ms$ by servicing partial requests of varying sizes depending on the load. We can observe constant response time around the target in Figure 3(a) for all load levels.

Next, we observe the goal of ensuring high result quality, balancing with the goal of controlling response time. We measure the response quality for a give load, which is the sum of response qualities

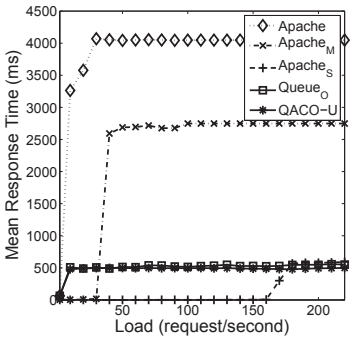


(a) Mean Response Time.

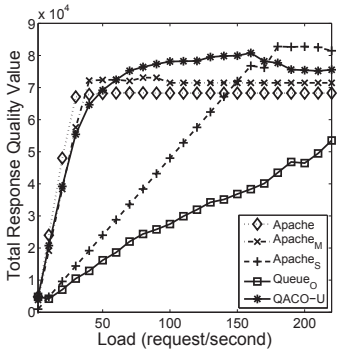


(b) Total Response Quality Value.

Figure 3: Apache Web server mean response time and total response quality for *QACO-U* and *Apache*.



(a) Mean Response Time.



(b) Total Response Quality Value.

Figure 4: Apache Web server mean response time and total response quality for systems under all approaches.

regardless of response time. We point out that this metric tends to under estimate the benefits of our techniques, since a very late response should have a lower quality.

Figure 3(b) shows the response quality of *Apache* and *QACO-U* at varying loads. At very low load (<10 rps), *Apache* and *QACO-U* have the same quality as they always return the webpages with the original size. At loads 10-50 rps, *Apache* achieves slightly higher quality with a cost of much higher response time than the target. At high-load, the quality value of *Apache* plateaus at 70000 (after 30 r/s), as the server becomes overloaded and therefore requests start to get dropped by the server. In contrast, the quality value of *QACO-U* continues to increase as the load level increases, while meeting the response time targets shown in Figure 3(a). With around 50 rps and up, *QACO-U* achieves both higher quality and much lower response time: The benefits of *QACO-U* come from exploiting the partial results and performing better scheduling that exploits the quality function. When requests are competing for resources and the server is overloaded, *QACO-U* maximizes the total result quality by serving the portion (or versions) of requests with higher quality gain given a limited amount of processing time. In contrast *Apache* either serves a request in full or drops a request, while *QACO-U* satisfies more requests (a smaller version if needed) to increase the total result quality.

4.3.2 Comparison to other approaches

We next compare *QACO-U* to other three approaches, $Apache_M$, $Apache_S$, and $Queue_O$.

First, we focus on response time; Figure 4(a) shows the mean response time of each approach at varying loads. We observe that the response time of $Apache_M$ exceeds our target of 500ms after 35 r/s, while *Apache*'s response time increases rapidly after 10 r/s. $Apache_M$ and *Apache* fail to meet response time target because they do not have admission control or other mechanism to meet a response time target. Other systems, $Apache_S$ and $Queue_O$, meet the response time target, though they suffer from a lower quality value than *QACO-U* as we show next.

We compare the quality of *QACO-U* to other approaches. Figure 4(b) shows the quality of each approach at varying loads. Overall, *QACO-U* performs comparably to the winners over all loads. *Apache* and $Apache_M$ have a short period of time offering higher quality than *QACO-U*, but at the cost of much longer response time than the desired target. The quality value of $Apache_S$ and $Queue_O$ is consistently lower than *QACO-U*. $Apache_S$'s quality is low because it always sends S (the small version of a webpage) even when the system is lightly loaded and has ample resources to send higher quality results. $Queue_O$ effectively meets the response time target by using feedback control, however, *QACO-U* outperforms $Queue_O$ on quality because *QACO-U* exploits partial execution. At high load, $Queue_O$ may serve one request in O but reject other requests, while *QACO-U* decides a version for each request based on their quality values, their processing demand and the total load, which increases total response quality given the available resource constraints.

In summary, *QACO-U* is the only approach that shows adaptive behaviors that balance the dual goals of controlling response time and achieving high result quality. It successfully exploits partial execution and meets response time targets for a wide range of load.

5. SIMULATION OF VOD SERVER

This section reports the results of a simulation study on a VoD server which receives requests for video segments, and services them with low response time and high quality under a limited server bandwidth. Since the service demand of each request is known in a VoD server storing pre-coded videos, we apply *QACO-K*. We also include

the results of *QACO-U* for comparison. The simulation results show the following: (1) *QACO-K* and *QACO-U* consistently achieve a higher quality than queue-based approaches while meeting the response time target more accurately. (2) *QACO-K* achieves an even higher quality than *QACO-U* by exploiting the additional information on service demand. (3) *QACO* can also effectively bound the high-percentile response time (rather than the mean response time) while achieving a high response quality. (4) *QACO-K* and *QACO-U* perform consistently well with different quality functions.

5.1 Simulation Setup

VoD server services requests with the given limited bandwidth. As requests compete for the server bandwidth, the scheduling algorithms can exploit partial execution by sending a lower-definition videos to trade quality for response time. The simulation parameters, include server bandwidth, video bit rates and target response time are listed in Table 1, following the convention of similar simulation studies [15].

Table 1: Model Parameters for VOD Server.

Parameter	Value
Server Bandwidth	1GBps
Video rate	10MBps (O), 5MBps (M), 2MBps (S)
Target Response Time	5secs

We use an open-loop workload generator to drive the VoD server, sending requests from 2 r/s to 1500 r/s. For each load, we run a simulation for 600 seconds to obtain the results of a single experiment. We run the simulation three times for each point and compute average values. Each video supports three different quality levels, e.g., high- , medium- and low-definition, which we denote as original O, medium M, and small S, respectively. We set the bit rate of M as $\frac{1}{2} * O$ and the bit rate of S as $\frac{1}{5} * O$. We use two sets of quality values for full and partial results: (set 1) 1 for O, 0.7 for M, and 0.3 for S; and (set 2) 1 for O, 0.8 for M, and 0.4 for S.

We compare *QACO-K* and *QACO-U* to the following three queue-based approaches: (1) $Queue_O$ sends high-definition videos, and it uses an integral controller to control the queue-length and rejects requests when queue is full. (2) $Queue_M$ and (3) $Queue_S$ apply the same feedback control mechanism, while they send medium- and low- definition videos, respectively.

5.2 Mean Response Time Results

We consider both known and unknown service demands – *QACO-K* knows the service demands of pending requests in the queue and their quality function, while *QACO-U* has no such knowledge.

The goal of this evaluation is to verify whether different approaches can control the response time to meet the mean response time target of 5s over varying load levels. Figure 5(a) shows the response time of each approach over varying loads. All five approaches return replies to meet the desired response time target. However, the queue-base approaches, by executing either in full or none, show a drastic increase in response time at certain load points: In Figure 5(a), we observe that, at 200 r/s, $Queue_O$ increases to 5s, and also similarly at 400 r/s for $Queue_M$ and at 1100 r/s for $Queue_S$. In contrast, the mean response times of *QACO-K* and *QACO-U* increase gracefully, as their optimization space is expanded to include partial executions.

Figure 5(b) shows the corresponding quality value. At low loads, the total quality value of all approaches increases as load increases. However, when the load is high the mean response time reaches the target and the queue-based approaches start to drop requests and suffer a decrease in quality. In contrast, both *QACO-K* and *QACO-U* are able to find an appropriate partial execution by exploiting concave-type quality functions, and as a result, achieve a higher quality value than the three queue-based approaches, for all loads that we used.

We also observe that *QACO-K* consistently outperforms *QACO-U* in terms of the quality value because *QACO-U* may under- or over-estimate service demands of requests in the queue. *QACO-U* does not have the precise information and can only estimate using an mean/expected value. In contrast, *QACO-K*, knowing the service demands of all pending requests, estimates the possible processing times of given requests more precisely. *QACO-K* successfully exploits the knowledge of the service demands and offers better results than *QACO-U*. Both algorithms outperform the queue-based approach by exploiting concave-type quality functions even when the service demands of pending request are unknown.

5.3 99-Percentile Response Time Results

To offer a desired user experience, VoD servers often need to guarantee stringent responsiveness, expressed as high-percentile response time (which is widely considered in the literature [10]). A commonly-used metric is 99-percentile response time, i.e., 99% of the requests need to satisfy that response time requirement. We show that *QACO* can also bound the high-percentile response time while improving the total quality. Figures 5 (c) and (d) show the 99-percentile response time and quality results of the five algorithms with varying loads, given a 99-percentile response time requirement of 6 seconds. The results show that, while both quota-based and queue-based approaches effectively bound the 99-percentile response time to the desired value, both *QACO-U* and *QACO-K* consistently offer a higher quality than the traditional queue-based approaches, demonstrating the effectiveness of exploiting partial execution and concave-type quality function for quality improvement.

5.4 Sensitivity Analysis

Figures 5 (e) and (f) report the same set of results on controlling the mean response time for a different quality function: Set 2 with quality 1 for O, 0.8 for M, and 0.4 for S. We observe that the response time (Figure 5 (a) and (e)) and quality graphs (Figure 5 (b) and (f)) are similar. We also did additional sensitivity analyses using different quality functions: Set 3 with quality 1 for O, 0.9 for M, and 0.5 for S, and Set 4 with quality 1 for O, 0.8 for M, and 0.5 for S. The results are consistent with the results in Figures 5 (a) and (b). This suggests that our approach is applicable to various concave quality functions.

6. RELATED WORK

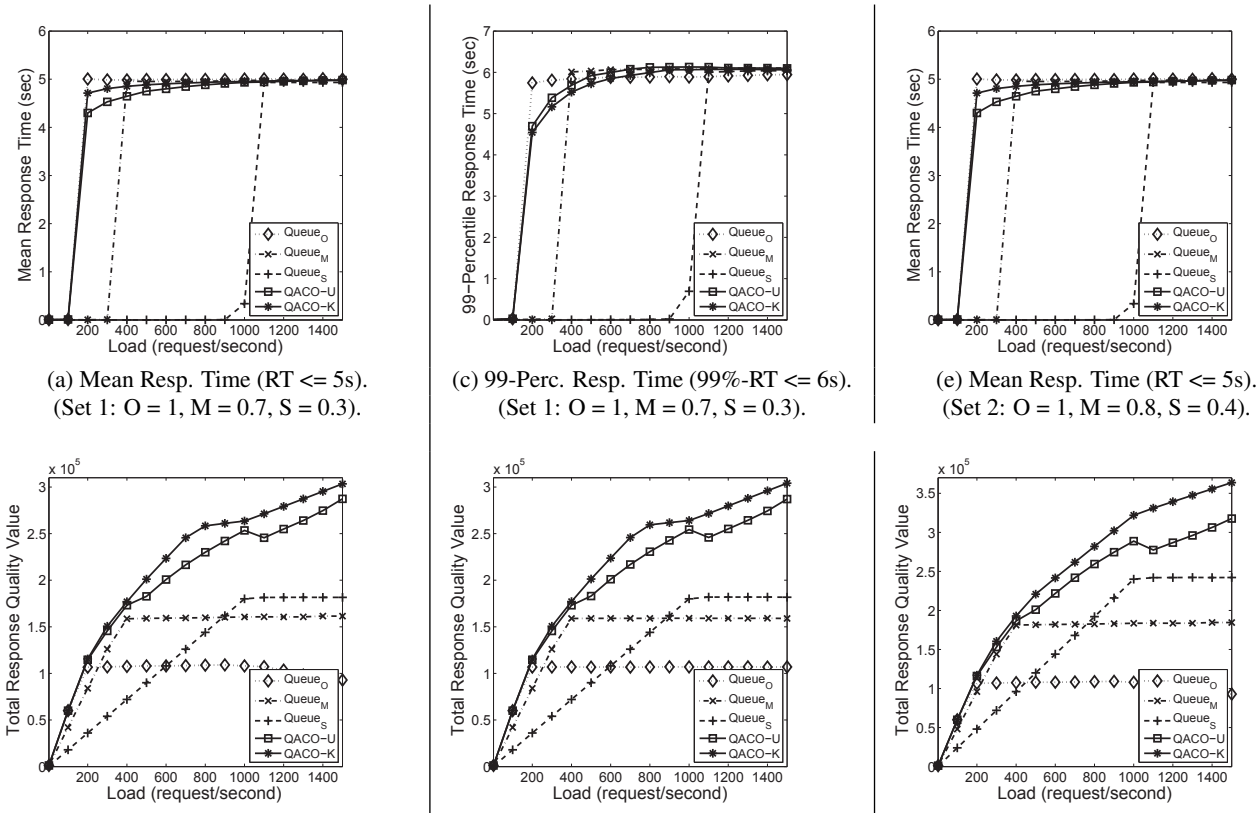
Feedback control theory has been widely used to achieve performance guarantees in computer systems with many applications such as multimedia streaming, real-time computing, transaction processing, and many others [1, 17]. In this section, we focus on server systems using feedback control to meet response time guarantees, and applications that use partial execution.

6.1 Controlling Server Systems for Response Time

The prior work along this line focuses on three main scenarios:

(1) Control for relative response time. For example, feedback control loop for an Apache web server has been used [26] to enforce desired relative response time among different service classes via connection scheduling and process reallocation.

(2) Control elastic resources. In these prior works [20, 21, 25, 31, 35], systems acquire and release resources in response to dynamic workload to meet response time target. There are various types of resources to adapt: For example, adding or removing a storage node [21], altering CPU allocation [35], changing processing speed through dynamic voltage and frequency scaling [20].



(a) Mean Resp. Time (RT <= 5s). (Set 1: O = 1, M = 0.7, S = 0.3).

(c) 99-Perc. Resp. Time (99%-RT <= 6s). (Set 1: O = 1, M = 0.7, S = 0.3).

(e) Mean Resp. Time (RT <= 5s). (Set 2: O = 1, M = 0.8, S = 0.4).

(b) Total Resp. Quality (RT <= 5s). (Set 1: O = 1, M = 0.7, S = 0.3).

(d) Total Resp. Quality (99%-RT <= 6s). (Set 1: O = 1, M = 0.7, S = 0.3).

(f) Total Resp. Quality (RT <= 5s). (Set 2: O = 1, M = 0.8, S = 0.4).

Figure 5: VoD server mean and 99-percentile response time, and total response quality using simulation.

(3) Control to prevent overloading. While a well-designed system should not be persistently overloaded, transient periods of overload are often inevitable, since the load is external to the server system and requests arrive according to a stochastic process, leading to transient overload and underload periods at the server. Such transient periods are inevitable and difficult to predict [29]. Many prior works [6, 8, 24] apply feedback control to cope with transient overload, deciding when to drop requests in order to meet response time target.

The above prior works [6,8,20,21,24–26,31,35] use control theory to achieve response time guarantees, however, none of them consider partial execution of requests. As in many prior works [13, 36] on admission control, they either serve a request in full or reject a request completely. By contrast, *QACO* is designed for servers that support partial evaluation and it optimizes the scheduling based on request quality function. We show that exploiting partial evaluation offers better response function.

6.2 Partial Execution

Employing partial execution and approximate/imprecise computations is an active area of research. For example, web content adaptation [7, 11] offers different versions of the content for the same request. Loop perforation [18] offers compiler and runtime support for partial execution and has been applied to audio and video codecs. Baek and Chilimbi [3] develop a general framework to support approximated computation of different applications to trade quality for lower energy. Montez and Fraga [27] similarly differentiate the requests into a discrete number of classes, e.g., depending on fees paid. However, they cannot support multiple levels of partial execution and evaluate with only one level, costing 20% of response time and 0.2 of the quality. These prior works [3,7,11,18,27] offer important insights on how to adapt execution for different applications. They focus on partial execution mechanism that enables individual requests to produce partial results. They do not, however, consider server environments where servers *proactively* enable partial execution to resolve resource competition among multiple requests with response time and quality targets. Imprecise computation techniques [23] also explore partial execution to address overloads and share some similarity with our proposal, but they are intended for *hard* real-time systems with deadline-constrained tasks and do not apply to our considered web servers in which average or high-percentile response time is of importance.

In video streaming applications, rate adaptation (or equivalently partial execution in our terminology) enabled by scalable video coding [34] uses mechanisms that exploit network conditions: sending lower-quality videos (e.g., only base layer in scalable video coding) to users with worse network connections or worse channel conditions in wireless networks [12, 32]. In contrast, *QACO* proposes to perform partial execution based on the system load at the server side, which is a complementary technique to the existing rate adaptation mechanisms [34]. Moreover, *QACO* can also be extended to incorporate the network condition for each request by imposing a constraint that requests with worse network conditions can only support a smaller subset of supported processing times. The simulation results in Section 5 demonstrate the significant quality improvement by *solely* using *QACO* at the server for partial execution, pointing to a potential research direction that combines both load-based and network-based partial execution mechanisms for further quality improvement in video streaming, which we leave as future work.

6.3 Systems with Content Adaptation

Prior work [15, 16] exploits partial execution in a different environment or for different objective. [15] uses partial execution to

improve total response quality of jobs while meeting job deadline. Their work to bound deadline is different from this work that bounds mean/percentile response time: average/percentile response time is accumulative statistics of multiple jobs, where feedback control is often applied to effectively monitor the performance metric and offer the desired guarantees, while deadline is imposed on each job, thus meeting deadline constraints rarely uses feedback control. Moreover, both [15] and [16] focus on the applications with continuous smooth quality functions instead of servers with discrete quality functions. Discrete quality functions increases the complexity of optimization modules: when request quality functions are continuous and concave, optimal solution can be obtained efficiently using convex optimization with known service demands; however, when quality functions are discrete, it becomes an NP-hard combinatorial optimization problem, which is notably difficult to solve. In this work, we develop an optimization module using a branch-and-bound technique to achieve an efficient solution and validate its effectiveness in both simulation and experimental studies.

The closest prior work to ours is controlling web servers that support partial execution. Abdelzaher and Bhatti [2] propose to resolve the overloading problem of web servers by adapting web content to load conditions. To meet the desired server utilization, they control the ratio between the requests offering degraded content versus all the requests. Although that work uses partial execution to meet their control target, it significantly differs from our work: It does not consider maximizing overall response quality for all requests as a goal, nor does it consider request quality function as an analytical guidance to improve the scheduling decision.

7. CONCLUSION

Web servers can exploit partial execution to meet response time requirements while providing higher quality responses. To exploit partial execution, we develop a scheduling framework that consists of two components. First, a control module dynamically determines a quota of resources of all pending requests, with the objective of meeting the response time requirements. Second, an optimization module assigns processing to each request in a manner that improves overall response quality. We implement the framework in the Apache Web server and evaluate its benefits empirically. We also conduct a simulation study and show how the proposed techniques improve the quality of a Video-on-Demand server. Our experimental and simulation results show that exploiting partial results improves response quality while meeting response time requirements.

8. REFERENCES

[1] T. Abdelzaher, Y. Diao, J. L. Hellerstein, C. Lu, and X. Zhu. Introduction to control theory and its application to computing systems, 2008.

[2] T. F. Abdelzaher and N. Bhatti. Web content adaptation to improve server overload behavior. In *WWW '99*, 1999.

[3] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI '10*, 2010.

[4] S. Boyd, A. Ghosh, and A. Magnani. Branch and Bound Methods, `http://www.stanford.edu/class/ee392o/bb.pdf`, 2003.

[5] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[6] H. Chen and P. Mohapatra. Session-based overload control in QoS-aware Web servers. In *INFOCOM '02*, 2002.

[7] Y. Chen, W.-Y. Ma, and H.-J. Zhang. Detecting web page structure for adaptive viewing on small form factor devices. In *WWW '03*, 2003.

[8] L. Cherkasova and P. Phaal. Session-based admission control: A mechanism for peak load management of commercial web sites. *IEEE Trans. Comput.*, 2002.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07*, 2007.

[10] Q. Du and X. Zhang. Statistical qos provisionings for wireless unicast/multicast of multi-layer video streams. *IEEE J.Sel. A. Commun.*, 28(3):420–433, Apr. 2010.

[11] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer. Adapting to network and client variation using infrastructural proxies: lessons and perspectives. *Personal Communications, IEEE*, 1998.

[12] F. Fu and M. van der Schaar. Decomposition principles and online learning in cross-layer optimization for delay-sensitive applications. *Trans. Sig. Proc.*, 58(3):1401–1415, Mar. 2010.

[13] R. K. Gullapalli, C. Muthusamy, and V. Babu. Control systems application in java based enterprise and cloud environments - a survey. *Journal of ACSA*, 2011.

[14] J. Hamilton. Blog article at `http://perspectives.mvdirona.com/2009/10/31/thecostoflatency.aspx`, 2009.

[15] Y. He, S. Elnikety, and H. Sun. Tians scheduling: Using partial processing in best-effort applications. In *ICDCS'11*, 2011.

[16] Y. He, Z. Ye, Q. Fu, and S. Elnikety. Budget-based control for interactive services with adaptive execution. In *ICAC '12*, 2012.

[17] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE Press, 2004.

[18] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *ASPLOS XVI*, 2011.

[19] J. Lee. *A First Course in Combinatorial Optimization*. Cambridge University Press, 2004.

[20] J. C. Leite, D. M. Kusic, D. Mossé, and L. Bertini. Stochastic approximation control of power and tardiness in a three-tier web-hosting cluster. In *ICAC '10*, 2010.

[21] H. C. Lim, S. Babu, and J. S. Chase. Automated control for elastic storage. In *ICAC '10*, 2010.

[22] M. Lin, A. Wierman, L. L. H. Andrew, and E. Thereska. Dynamic right-sizing for power-proportional data centers. In *IEEE Infocom*, 2011.

[23] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. *Proc. of IEEE*, 2004.

[24] X. Liu, J. Heo, L. Sha, and X. Zhu. Queueing-model-based adaptive control of multi-tiered web applications. *IEEE Trans. on Netw. and Serv. Manag.*, 2008.

[25] X. Liu, R. Zheng, J. Heo, Q. Wang, and L. Sha. Timing performance control in web server systems utilizing server internal state information. In *ICAS-ICNS '05*, 2005.

[26] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son. A feedback control approach for guaranteeing relative delays in web servers. In *RTAS '01*, 2001.

[27] C. Montez and J. da Silva Fraga. Implementing quality of service in web servers. In *SRDS '02*, 2002.

[28] M. J. Osborne and A. Rubinstein. *A Course in Game Thoeory*. The MIT Press, 1994.

[29] B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Trans. Internet Technol.*, 2006.

[30] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: a cautionary tale. In *NSDI'06*, 2006.

[31] L. Sha, X. Liu, Y. Lu, and T. Abdelzaher. Queueing model based network server performance control. In *RTSS '02*, 2002.

[32] C. Shen and M. van der Schaar. Optimal resource allocation for multimedia applications over multiaccess fading channels. *Trans. Wireless. Comm.*, 7(9):3546–3557, Sept. 2008.

[33] W. Szpankowski. Bounds for queue lengths in a contention packet broadcast system. *Communications, IEEE Transactions on*, 1986.

[34] M. van der Schaar and P. Chou. *Multimedia over IP and Wireless Networks: Compression, Networking, and Systems*. Academic Press, 2007.

[35] R. Wang, D. M. Kusic, and N. Kandasamy. A distributed control framework for performance management of virtualized computing environments. In *ICAC '10*, 2010.

[36] C. A. Yfoulis and A. Gounaris. Honoring slas on cloud computing services: a control perspective, 2009.