

QDSL: A Queuing Model for Systems with Differential Service Levels

Shiva Chaitanya
Computer Science and
Engineering
Pennsylvania State University
University Park, PA 16802
chaitany@cse.psu.edu

Bhuvan Uргаonkar
Computer Science and
Engineering
Pennsylvania State University
University Park, PA 16802
bhuvan@cse.psu.edu

Anand Sivasubramaniam
Computer Science and
Engineering
Pennsylvania State University
University Park, PA 16802
anand@cse.psu.edu

Abstract — A feature exhibited by many modern computing systems is their ability to improve the quality of output they generate for a given input by spending more computing resources on processing it. Often this improvement comes at the price of degraded performance in the form of reduced throughput or increased response time. We formulate QDSL, a class of constrained optimization problems defined in the context of a queueing server equipped with multiple levels of service. Solutions to QDSL provide rules for dynamically varying the service level to achieve desired trade-offs between output quality and performance. Our approach involves reducing restricted versions of such systems to Markov Decision Processes. We find two variants of such systems worth studying: (i) VarSL, in which a single request may be serviced using a combination of multiple levels during its lifetime and (ii) FixSL, in which the service level may not change during the lifetime of a request. Our modeling indicates that optimal service level selection policies in these systems correspond to very simple rules that can be implemented very efficiently in realistic, online systems. We find our policies to be useful in two response-time-sensitive real-world systems: (i) qSecStore, an iSCSI-based secure storage system that has access to multiple encryption functions, and (ii) qPowServer, a server with DVFS-capable processor. As a representative result, in an instance of qSecStore serving disk requests derived from the well-regarded TPC-H traces, we are able to improve the fraction of requests using more reliable encryption functions by 40-60%, while meeting performance targets. In a simulation of qPowServer employing realistic DVFS parameters, we are able to improve response times significantly while only violating specified server-wide power budgets by less than 5W.

Categories and Subject Descriptors

D.4.8 [Operating Systems]: Performance—*Queueing theory, Stochastic analysis, Simulation*

General Terms

Performance, Algorithms, Experimentation, Measurement

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMETRICS'08, June 2–6, 2008, Annapolis, Maryland, USA.
Copyright 2008 ACM 978-1-60558-005-0/08/06 ...\$5.00.

Keywords

Differential Service Levels, Markov Decision Process, Secure Storage, Dynamic Voltage Frequency Scaling

1. INTRODUCTION AND MOTIVATION

1.1 Motivation

Many computing systems can improve the quality of output they generate for a given input by expending more computational resources on it. That is, for the same input, they can operate at multiple *service levels*, each with a different computational requirement and providing a different output quality. We will use the term *Goodness of Service* (GoS) as a measure of the output quality provided by a service level. Trade-offs between GoS and performance may be seen in a variety of domains. For example, certain sensor devices can reduce the amount of data they need to transmit by running a more time-consuming compression algorithm [33]. Producing a higher quality solution (i.e., better compression, which results in lower communication costs) comes at the price of introducing higher delays (due to solving the more complex compression algorithm) before the data may be transmitted. Similarly, a system encrypting data items before storing them on a hard disk drive may be able to choose from a variety of encryption algorithms, each with a different computational requirement and offering a different “degree of security” [30, 41]. As a final example, a Web server running on a machine with Dynamic Voltage/Frequency Scaling (DVFS) capable CPUs may be able to trade-off power for response time of HTTP requests (performance) [38, 26].

These trade-offs introduce at least two fundamental challenges in the design and operation of such systems. The first challenge concerns the need for precise and quantifiable characterizations of these trade-offs. As an example, *how much degradation in response time would a Web server running on a DVFS-capable machine be willing to tolerate to save power?* Assuming the existence of a precise specification, the second challenge concerns the design of policies for scheduling and service-level selection to ensure that the system operates in desirable regimes. The service level selection policy of a system dictates how the service level at which it operates evolves with time. As an example, *how can a Web server running on a DVFS-capable machine minimize power consumption while providing a guaranteed upper bound on average response time?*

In systems with differential service levels, the problems of scheduling and service-level selection are complementary and intimately tied with one another. In this research, we assume the underlying scheduling policy to be fixed and study the problem of dynamically varying the service levels the system has to offer.

1.2 Research Contributions

Our research makes a number of contributions. First, we formulate a class of problems, QDSL, that capture trade-offs between GoS and performance found in a variety of computing domains. Specifically, we consider systems that can be represented as servers processing requests issued by clients and capable of operating at multiple service levels. The server is assumed to be equipped with multiple service levels and the switching mechanisms. Intuitively, a higher service level improves GoS but results in a slower response by requiring more computation. These trade-offs are captured via constrained optimization problems in which one of GoS and response time forms the optimization criterion whereas the other presents the constraints.

Second, we classify such systems into two categories that correspond to two different kinds of server operation: (i) VarSL, systems in which the service level used to process a request may be varied during its lifetime, and (ii) FixSL, systems in which the semantics of request processing do not allow such variations of service level. Under certain restrictions on the input, we derive reductions of VarSL and FixSL to MDPs and utilize well-established average reward maximization algorithms known for MDPs to derive optimal service-level selection policies. As a consequence of these reductions, our service level selection policies have the interesting property of being 1-stationary. Roughly speaking, our policies base their decision of selecting service levels *solely and deterministically* on the number of requests in the system (except for one value of the number of requests for which they decide probabilistically.) As a result, our policies are efficiently implementable in on-line systems. Furthermore, they are easily integrated with scheduling policies found in most real-world systems with multiple service levels.

Finally, we evaluate the utility of this research in the operation of two real-world systems that offer multiple levels of service. We use FixSL to model our first application domain, qSecStore. This is a iSCSI-based storage system that has access to multiple functions for encrypting disk requests. These encryption functions have varying degrees of computational requirements and desirability w.r.t security guarantees. The second domain considered in our study is qPowServer a DVFS-capable machine running the Web-based ECommerce benchmark TPC-W. We wish to dynamically vary the DVFS states to minimize average client response time while staying within specified server-wide power budgets. Our evaluation provides us insights into operating regimes where the policies provided by our modeling are useful in these systems. As a representative result, in an instance of qSecStore serving disk requests derived from the well-regarded TPC-H traces, we are able to improve the fraction of requests using more reliable encryption functions by up to 40-60%, while meeting performance targets. In a simulation of qPowServer employing realistic DVFS parameters, we are able to improve response times significantly while introducing minor violations of specified server-wide power budgets (less than 5W).

1.3 Roadmap

The rest of this paper is organized as follows. In Section 2, we formalize the service level selection problem and present a restricted version of this problem called rQDSL. In Section 3, we present optimal selection policies for two key variants of rQDSL. In Section 4, we empirically evaluate the suitability of these policies in two different domains. We discuss related work in Section 5 and present concluding remarks in Section 6.

2. QDSL: MODELING SYSTEMS WITH DIFFERENTIAL SERVICE LEVELS

A useful classification of systems with multiple service levels is based on the temporal granularity at which they permit these service levels to be changed. Certain systems require that the service level associated with a request be fixed during its processing. The level chosen, when the request starts receiving service, may not be changed during its lifetime. An example of such a system is our qSecStore system. All the data bits within a single storage request should be cryptographically encoded using only one type of algorithm. We call such systems, where the service level for a request is fixed at its initiation, as *Fixed Service-level* or FixSL.

In other systems, service level may be changed at any time, independent of the scheduling events. qPowServer is an example of such a system. DVFS states in a qPowServer system may be changed at any time without affecting the correctness of the processing of workloads running on the server. Whereas service level in qSecStore is tied with each request in terms of the guarantee provided to it by using a certain crypto function, in qPowServer, a service level is a DVFS state and is semantically tied with the server/CPU which can instantaneously switch amongst different power states. As a result, a single CPU task (request) can experience multiple service levels during its lifetime. We refer to these systems as *Variable Service-level* or VarSL.

In this section, we formulate the QDSL problem. We then consider a restricted version of this problem called rQDSL that we find amenable to theoretical investigation. Finally, we discuss the implications of these restrictions on the applicability of QDSL.

2.1 Basic Queuing Model

We model the system of interest as a single queuing server. The server is capable of operating at K different *service levels* chosen from the set $\mathcal{L} = \{l_1, \dots, l_K\}$. We denote by the random variable s_i ($1 \leq i \leq K$) the service time of a request when the server operates at service level l_i . Let \bar{s}_i denote the average of s_i . We assume the average service times for the service levels l_1, \dots, l_K to be monotonically increasing, i.e., $0 < \bar{s}_1 < \dots < \bar{s}_K$.

We wish to model systems with the following characteristic: by doing *more "work"* on a request (that is, spending more time processing a request), the server *improves the GoS* for the request. As an example, a server may be able to provide an improved security guarantee (GoS) to a data item by applying a more compute-intensive encryption algorithm to it. Toward this end, we make the service levels l_1, \dots, l_K correspond to monotonically improving GoS. We achieve this by assuming that the processing of requests generates *revenue* in the following manner. We assume the existence of a revenue function $\$: \mathcal{L} \rightarrow \mathbb{R}^+$ such that a request processed at service level l_i generates revenue at the rate $\$(l_i)$. Additionally, we assume that $\forall i, j : i < j \Leftrightarrow \$(l_i) < \$(l_j)$. Intuitively, processing a request at a higher service level generates higher revenue (which captures the improved GoS for the request) *and* requires more processing at the server (which captures the higher amount of work done by the system in processing the request.)

Our server employs a *service-level-selection algorithm* that complements its scheduling algorithm in the following manner. It partitions the duration between any successive invocations of the scheduling algorithm (that is, any *contiguous* duration for which a request is processed by our server), into non-overlapping and adjacent time intervals, in each of which, one service level is employed. Note that FixSL systems allow service level changes only at scheduling instants. Figure 1 illustrates the operation of a service-level-selection algorithm and its relationship with invocations of the scheduling algorithm for VarSL and FixSL, respectively.

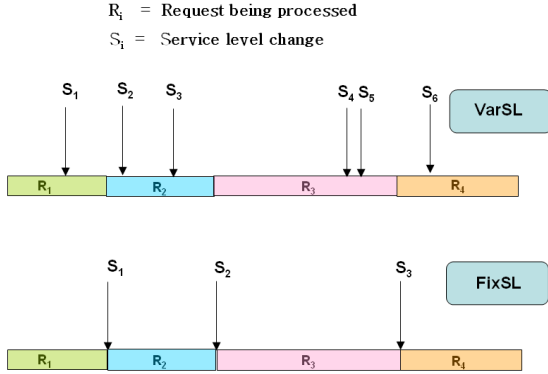


Figure 1: Illustration of the difference in service-level selection between the VarSL and FixSL systems. In VarSL, the service level can be changed at any time, including during the processing of a request. In FixSL, a fixed service level is associated with each request and is not changed throughout the request’s processing lifetime.

We are interested in exploring the trade-offs between two quantities: (i) average response time per request and (ii) revenue generated by the server per unit time. QDSL is a set of service-level-selection problems that require us to optimize one of these quantities while satisfying some constraint on the other. It can be shown that QDSL is NP-hard. Finding an optimal solution to QDSL is, therefore, unlikely to be tractable even in the presence of complete information about the arrival times and service requirements of the requests. Instead, we focus on developing stochastically optimal solutions.

2.2 rQDSL: A Restricted QDSL Problem

We impose certain restrictions on the general QDSL problem and study the resulting problem, rQDSL, theoretically. These restrictions concern the nature of the requests and the scope of the optimization problems we wish to solve. In Section 4, we attempt to empirically study the implications of some of these restrictions.

Workload characteristics and scheduling policies. Requests are assumed to arrive according to a Poisson process with rate λ . The service time variable s_i ($1 \leq i \leq K$) (defined in Section 2.1) is assumed to follow an exponential distribution with mean \bar{s}_i . We denote the corresponding service rates by μ_1, \dots, μ_K , ($\mu_i = \frac{1}{\bar{s}_i}$). In addition we assume that the scheduling policy is FIFO (requests are serviced in their arrival order).

Optimization criteria. Let \bar{R} denote the average response time per request and \mathbb{S} denote the revenue generated by the server per unit time. We are interested in the following two formulations. Our evaluation in Section 4 uses both these formulations.

Response Time Minimization: Minimize \bar{R} s.t. $\mathbb{S} \geq \mathbb{S}^{min}$, where \mathbb{S}^{min} is a specified lower bound on revenue generated per unit time.

Revenue Maximization: Maximize \mathbb{S} s.t. $\bar{R} \leq R^{max}$, where R^{max} is a specified upper bound on average response time.

We only concern ourselves with operating regimes where these constraints are realizable. In Section 4, we evaluate these problem formulations with easy-to-derive revenue or response time bounds. Determining tight bounds on the revenue or response is orthogonal to our work.

2.3 Discussion on rQDSL

Since the motivation for this work stems from the two real-world systems problems qSecStore and qPowServer, we are particularly interested in understanding the implications of various simplifying assumptions made in rQDSL. Three key assumptions in our model are related to (i) the nature of the arrival process, (ii) service time distribution, and (iii) the scheduling policy. As we will discuss in Section 4, well-regarded workloads in both these domains are known to deviate from Poisson arrivals and exponential service times. We attempt to systematically study the impact of these deviations on the utility of our model in these systems. Our assumptions about the scheduling policy appear to be less removed from reality than (i) and (ii) when modeling qSecStore. We will find in Section 4.1 (Table 1), that the encryption times for the individual disk requests are in the order of tens or hundreds of microseconds, meaning these requests would be unlikely to be pre-empted by the underlying CPU scheduler. As a result, their scheduling is well-captured by our assumption of FIFO policy.

Reductions to MDPs. Since problems captured by QDSL arise in many domains (recall examples from Section 1.1), a variety of proposals for addressing them exist. A feature common to several of these approaches is to impose restrictions on the nature of request arrivals and their service demands to allow analytical solutions. Problems of the same form as VarSL have been addressed in the Operations Research literature [18] by restricting the arrival process to be Poisson and service times to follow exponential distributions. Under these assumptions, VarSL-like problems are easily reduced to Markov Decision Processes (MDPs.) Well-established algorithms for reward maximization in MDPs are then employed to derive stochastically optimal solutions for these problems. We build upon this research in Section 3. Whereas we are able to easily adapt this existing work to VarSL systems, developing a solution for FixSL systems is less straightforward. We develop a reduction of FixSL to a semi-MDP. This reduction yields us the desired optimal algorithm for FixSL. We consider this reduction to be a key contribution of our work. Our other significant contribution is an empirical evaluation of the utility of our techniques for solving VarSL and FixSL in two real-world systems (Section 4.)

Quick background on MDPs. A Markov Decision Process consists of the following components: (i) the state space \mathcal{S} , (ii) the decision space \mathcal{D} , (iii) a transition function f associated with each (state, decision) pair - $f(S, D)$ describes the probability of transitioning to the next state $S' \in \mathcal{S}$ from state S upon making the decision D , and (iv) a reward function w associated with each (state, decision) pair - $w(S, D)$ describes the rate at which the MDP accumulates reward till the next change in state/decision. MDPs are accompanied with an optimization criterion based on the rewards. Algorithms for determining a stochastically optimal behavior of the MDP with respect to this criterion are well-known [34]. We survey other relevant literature in Section 5.

3. AN OPTIMAL MDP-BASED ALGORITHM FOR rQDSL

We reduce rQDSL to a continuous-time MDP. We find that VarSL systems lend themselves to reductions to MDPs that are structurally different from those for FixSL systems. We consider these two reductions in turn and point out their similarities and differences. We then employ existing research on MDPs to develop an optimal algorithm for rQDSL. Finally, we present our thoughts on the limitations of rQDSL as well as our expectations about its utility.

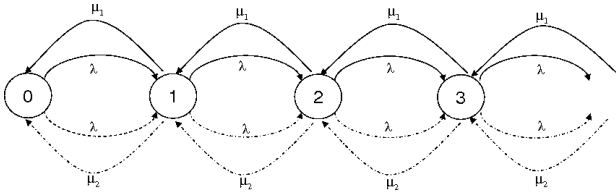


Figure 2: Continuous-time Markov Decision Process for a VarSL system with two service levels. The transition edges in thick lines correspond to decision of service rate μ_1 and the transition edges in dotted lines correspond to service rate μ_2 .

3.1 Reduction for VarSL Systems

We represent a VarSL server as a continuous-time MDP. This reduction is adopted from existing research [8, 18], hence we present it briefly.

States and decision sets. The states of our continuous-time MDP correspond to the number of requests in the system. We denote these states as $\{0, 1, 2, \dots\}$. Associated with each state of our MDP is a finite set of decisions. The decision to be made in a state concerns the choice of the service level to be picked for the request currently being processed by the server. The feasible decision set for any state is $\mathcal{L} = \{l_1, \dots, l_K\}$.

Transitions between states of the MDP. Transitions occur between states due to an arrival or a departure of a request to/from the queuing system. Figure 2 shows a representation of our MDP when the set of available service levels is $\{l_1, l_2\}$. The solid transition edge from a state corresponds to the decision of picking the service level l_1 which has a processing rate μ_1 ; the dotted edge corresponds to the decision of picking service level l_2 which has a processing rate μ_2 . For every decision made, the value on top of each transitional edge is the rate of transition to the next state.

We refer the reader to existing research for detailed proof of the correctness of this reduction [40, 34]. Briefly, the assumptions of Poisson arrival process and exponential service times allow the above reduction. The memory-less property of transitions between MDP states is satisfied by the above two assumptions on arrival and service time distributions. At any time instant t , if the (state, decision) pair is $(S(t), D(t))$ ($S(t) > 0$), the set of subsequent states $\{S(t) - 1, S(t) + 1\}$ depends only on $S(t)$. Furthermore, the infinitesimal rate of transitioning into the next state is dependent only on the current decision. VarSL allows any feasible decision to be made (i.e., service level to be changed) at any time instant.

Reward functions. The only aspect of our MDP that remains to be specified are the rewards associated with each (state, decision) pair. Our proposals for choosing these reward functions in VarSL and FixSL systems are essentially similar, so we describe them together in Section 3.3.

3.2 Reduction for FixSL Systems

Next, we represent the FixSL service level selection variant of the rQDSL as a semi-MDP. As we will show, this requires a significantly different reduction from the one developed above.

Obstacles in reducing to a regular MDP. To appreciate the difficulty in designing a reduction for a FixSL system, consider the following differences from a VarSL system. In a VarSL system, the service level selection algorithm was allowed to choose

from the entire set of possible decisions (i.e., any possible service level) at *any time*. In a FixSL system, however, changes to service level may only be made when a new request starts processing. In any work-conserving, non-preemptive server, these decision instants correspond to times when requests depart the system. At any other time instant when a request is being processed, the service level chosen for this request cannot be changed.

Based on the above discussion, we would like to capture the following: changes to the service level occur only at the discrete time instants that correspond to departures of requests. A semi-Markov decision process is an abstraction similar to an MDP, that allows decisions to be made at discrete time instants, namely, only at state transition epochs [19]. That is, decisions are made in a state only at the instant of entry into the state. We would like to exploit this characteristic of semi-MDPs to develop a reduction for FixSL systems.

Given that a semi-MDP allows decisions at *state entry instants*, while in rQDSL, we would like decisions only at *request departure instants*, we create two classes of states in our semi-MDP. Transitions to the first class of states capture departure events in the FixSL system. These transitions are accompanied with all possible decisions representing changes to service levels. Our second class of states captures all other events (request arrival or departure) in the FixSL system, where we do not allow any new decisions.

States and decision sets. We now describe our semi-MDP construction in detail. Each state of the semi-MDP belongs to one of two classes. The first class of states captures situations when the queuing system's last event was either (i) a departure of a request or (ii) an arrival of a request to an empty system. A state belonging to this class is simply represented by the number of requests in the system. State 0 means that there are no requests in the system. A state $i > 0$ means that the current number of requests in the system is i and the previous state had $(i + 1)$ requests in the system. The second class captures situations when the queuing system has at least two requests in the system and the previous event was an arrival. We denote this state by the two-tuple (i, l_j) , implying that the current number of requests in the system is $i > 1$, current service rate is μ_j and the previous system event was an arrival.

The feasible decision sets for our states are defined as follows. For the first class of states represented by i , the decision set is $\mathcal{L} = \{l_1, \dots, l_K\}$, i.e., all service levels. This captures the requirement that any decision be allowed when a request departs a non-empty system or when a request arrives to an empty system. For the second class of states represented by (i, l_j) , the feasible decision set consists of only one member and is l_j . This is because the service level may not be changed upon entering these states (which can only happen upon arrivals.) It should now be clear, why we incorporate the current service level into the two-tuple that defines our second class of states. This allows us to remember the decision made at the latest departure, which is the only decision available for the remaining lifetime of the current request.

Transitions between states of MDP. Recall that new decisions are possible only upon entry to the first class of states. Based on this, let us consider transitions from these two classes of states, in turn.

Transitions from the first class of states: Consider transitions due to departures and arrivals separately. Corresponding to a *departure* from a state $i > 0$, upon entry to which a decision j ($l_j \in \mathcal{L}$) was made, we create a transition to state $(i - 1)$ with a rate μ_j . Corresponding to an *arrival* during state $i > 0$, upon entry to which a decision j ($l_j \in \mathcal{L}$) was made, we create a transition to state

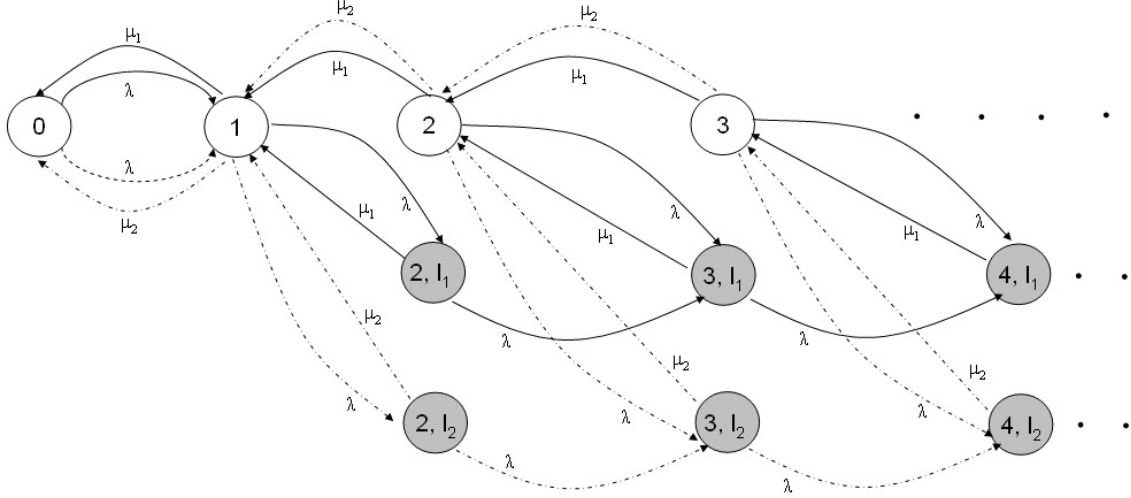


Figure 3: Semi-Markov decision process for a `FixSL` system with two service levels. The transition edges in thick lines correspond to decision of service rate μ_1 and the transition edges in dotted lines correspond to service rate μ_2 .

$(i + 1, l_j)$ with a rate of λ . Transitions from the state 0 occur at rates λ to state 1, one for each possible decision.

Transitions from the second class of states: Transitions from the second class of states are easily derived as follows. From a state (i, l_j) ($i > 0$), we create transitions to $(i - 1)$ (corresponding to a departure) and $(i + 1, l_j)$ (corresponding to an arrival), with rates of μ_j and λ , respectively.

Taken together, the above definitions of states, transitions, and decision sets conclude our reduction. Figure 3 represents the semi-MDP formulation for a `FixSL` system with two service levels.

3.3 Reward Functions for Our MDPs

Recall from Section 2.3 that an MDP has a reward function $w(\cdot)$ associated with each (state, decision) pair. This means that the MDP, when in state S and with the latest decision D , accumulates a reward at the rate $w(S, D)$ till the next change in state and/or decision. In general, there may be multiple reward functions associated with an MDP. Average reward-constrained MDPs [16] consider the following problem of maximizing one reward function subject to constraints on other reward functions. Let us denote by W_0, \dots, W_m these reward functions; W_0 is the reward function to be maximized while the rest are used to specify constraints. Formally, an average reward-constrained MDP solves the following,

$$\text{maximize } \int_{t=0}^{\text{inf}} W_0(S(t), D(t)) dt$$

subject to the constraints,

$$\int_{t=0}^{\text{inf}} W_j(S(t), D(t)) dt \leq C_j \quad j = 1, \dots, m.$$

This formulation suits our purposes since we wish to trade-off response time with revenue. We now show how to choose the reward functions for the MDPs constructed above to let us use known algorithms for average reward-constrained MDPs for solving `rQDSL`. We choose two reward functions, one corresponding to the average response time and the other to the revenue. Recall the two variants of `rQDSL` that were defined in Section 2.2. It is easily seen that

treating the reward function corresponding to the average response time as W_0 results in a reduction of the Response Time Minimization problem to an average reward-constrained MDP; treating the reward function corresponding to revenue as W_0 results in a reduction of the Revenue Maximization problem.

For a (state, decision) pair $(S, D = l)$, the reward function corresponding to revenue is simply $\$(l)$. The appropriateness of this reward function is straightforward from our definition of revenue (see Section 2). The other reward function is less easy to define. Recall that we wish the average of the reward accumulated according to this function to correspond to the average response time. We know from Little's Law that the average response time in any queuing system is N/λ , where N is the average number of requests in the system and λ is the throughput (steady-state assumption.) If we define the reward function for each state to be directly proportional to the number of requests in that state, we would achieve the above property. Based on this intuition, for a (state, decision) pair $(S, D = l)$, we choose the reward function to be $\frac{i}{\lambda}$ where i is the number of requests in the system corresponding to the state S . Notice that, for the semi-MDP (Figure 3), this reward function is the same ($\frac{i}{\lambda}$) in all states i and $(i, -)$.

3.4 Solving `rQDSL`

We use an existing Linear Programming (LP) based approach for solving an average reward-constrained MDP [16]. This approach optimally solves both the continuous-time MDP and semi-MDP under the assumptions that the number of states in the MDP is finite and that the model is *uni-chain*. An MDP is *uni-chain* if the underlying Markov chain has no *absorbing* subset of states for any stationary policy. A stationary policy is one where the decision to be made at time t is independent of the history of MDP states and depends only on the current state $S(t)$. An absorbing set of states are those from which no outgoing transitions exist. It is easily seen that the MDPs that we have formulated for both `VarSL` and `FixSL` are *uni-chain*.

It is known that for an average reward-constrained MDP, there exists an optimal policy that is M -randomized stationary where M

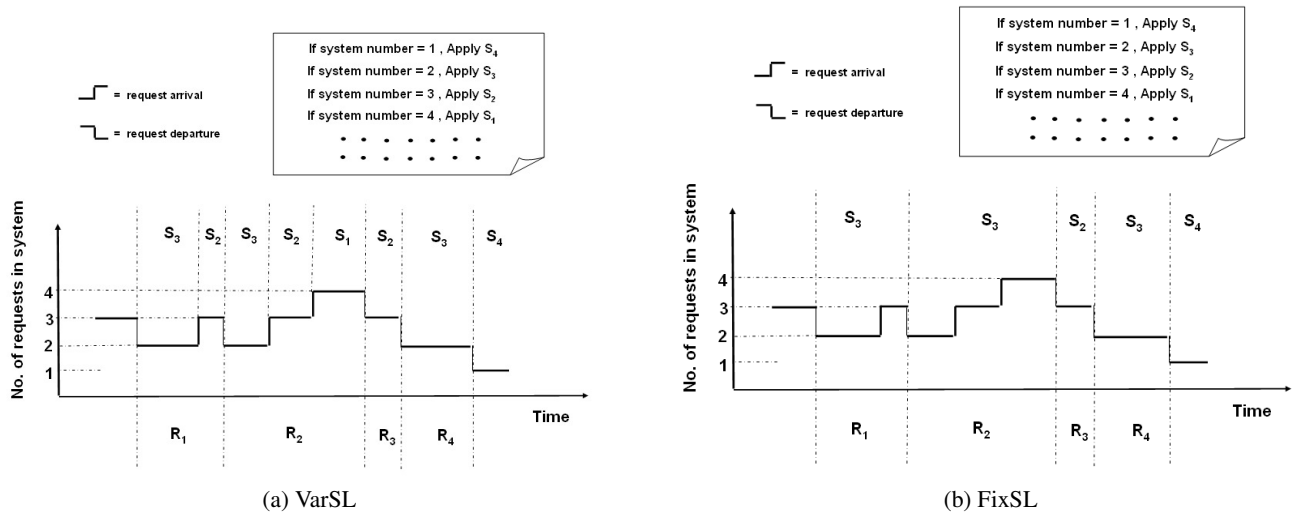


Figure 4: Illustration of the optimal policy structures for the VarSL and FixSL variants. In VarSL, the decision to choose the next service level is made whenever the system number changes whereas FixSL chooses the decision at the beginning of a request’s processing. Both the policies determine service levels based on the number of requests in system.

is the number of reward constraints [16]. A non-randomized stationary policy f always chooses the same decision in a given state i ; the decision made in state i is denoted by $f(i)$. A 1-randomized stationary policy is one that is non-randomized in all states except for one state where the decision is randomized between two actions. Note that such a policy is still stationary, in the sense that the decision policy depends only on the current state. This definition extends to a M -randomized stationary policy. The optimal policy for $rQDSL$ is a 1-randomized stationary policy because there is exactly one reward constraint. Therefore, the optimal policies for VarSL and FixSL systems have the same structure in terms of being 1-stationary.

Such an optimal policy can be represented by a parameter set $(P(\cdot), m, l_1, l_2, p_1, p_2)$, $P : \mathcal{I} \rightarrow \mathcal{L}$, $(m \in \{\mathcal{I}^+ \cup \{0\}\})$, which should be interpreted as follows.

Pick service levels to satisfy the following conditions:

if the number of requests k in system is m , apply service level l_1 or l_2 with probabilities p_1, p_2 , respectively.

else apply service level $P(k)$

The two systems VarSL and FixSL, however, differ in the manner of policy invocations for deciding the next service level. In the case of VarSL, the policy is invoked every time a request enters or leaves the system (corresponding to an entry into a new state in figure 2.) In FixSL, the policy is invoked at the start of each request’s processing. Figures 4(a) & (b) illustrate the optimal policy structures and their invocation instants for VarSL and FixSL, respectively.

The algorithm that $rQDSL$ employs [16] formulates an LP with one variable for every possible (state, decision) pair. Since any real system has an upper limit U on the number of requests simultaneously in the queue, we truncate all states corresponding to more than U requests in the system. If there are K service levels, the LP for a VarSL system has $U \cdot K$ variables, whereas that for a FixSL system has twice as many variables, $2 \cdot U \cdot K$.

Clearly, to be implementable in real-world systems, it should be possible to limit the frequency of invoking $rQDSL$ such that the

computational overheads it poses do not interfere with the activities of the system. Systems in which workload patterns repeat and are amenable to prediction can cache and reuse the policies computed by $rQDSL$. The utility of $rQDSL$ would, therefore, depend intimately on these properties of a system’s workload.

Finally, once the workload conditions for an upcoming time interval have been identified and the optimal policy during that interval computed using $rQDSL$, (or retrieved from the policy cache described above), the service level selection is extremely simple and fast—all it involves is comparing the current number of requests in the system with a threshold computed by $rQDSL$.

4. APPLICATIONS AND EMPIRICAL EVALUATION OF RQDSL

We conduct simulation studies to evaluate the utility of $rQDSL$ policy in two different application domains. Henceforth, we will refer to the policy as simply $rQDSL$. We simulate a generic QDSL system using the CSIM19 library [9]. This library provides useful routines to build an event-driven simulator.

We compare $rQDSL$ with the following policies.

- *Best static policy:* A static policy employs a fixed service level. So there are K static policies for a QDSL system with K service levels. The best static policy amongst these is the one that results in the best optimization criterion (largest revenue or smallest average response time) while satisfying the constraint (response time or revenue bound, respectively.)
- *M/GI/1-based dynamic policy:* This policy dynamically varies the service levels as an independent distribution as follows. Modeling the system as a M/GI/1 queuing system, it determines probabilities Pr_1, \dots, Pr_K , one for choosing each service level, that correspond to solving the relevant optimization problem. We omit the details of the calculation due to space constraints. The following comparison of the behavior of this policy with our policies is worth pointing out. This policy has the same stochastic behavior regardless of the number of requests in the system. This is a useful policy to compare against since it tightly meets average response

Service level	Crypto functions	Avg. latency time (microsec)	Service Quality	Revenue per request
L_1	AES128	110	128 bit (key length) Encryption	20
L_2	AES128 + HMAC-MD5	170	128 bit Encryption, 128 bit Authentication	40
L_3	AES256 + HMAC-MD5	210	256 bit Encryption, 128 bit Authentication	50
L_4	AES256 + HMAC-SHA1	440	256 bit Encryption, 160 bit Authentication	100

Table 1: Details of the four service levels in $qSecStore$. The security guarantees progressively increase from L_1 to L_4 and are captured by the revenue-per-request values. The average latency per 16KB data block was measured on a 3.06 GHz Intel Xeon processor

time bounds for workloads with Poisson arrivals ($rQDSL$ requires the additional assumption of exponential service times for each service level to do so.)

Henceforth, we refer to these policies simply as *static* and *dynamic* policies.

4.1 $qSecStore$: A Performance-sensitive Secure Storage System

4.1.1 Problem Context

In our first case study, we consider a secure storage server that can provide differential levels of security guarantees to each request it handles. Specifically, we consider an iSCSI server that services disk block requests from a remote client across a TCP/IP network. We have modified the iSCSI protocol to incorporate security at the iSCSI layer [5]. This helps in providing better performance/scalability compared to techniques where iSCSI leverages transport security mechanisms like IPsec/SSL. In addition, it provides the flexibility of being able to dynamically switch amongst different security primitives at the granularity of a disk request.

There exists a wide variety of crypto algorithms that differ in the security guarantees and per-block computation latencies. For example, authentication algorithms such as HMAC-MD5, HMAC-SHA1 ensure integrity of data by one-way hashing of data. By transmitting the data along with its hash, one can detect any malicious tampering. Encryption algorithms such as DES, AES provide data confidentiality where the original pattern of data bits is obscured. Individually, each of the algorithms can improve their crypto strength by considering different key lengths.

4.1.2 Adapting $rQDSL$ to $qSecStore$

Consider the example of a iSCSI READ operation at the storage server. After the disk blocks corresponding to the iSCSI request are fetched from the storage device, they are encapsulated in a iSCSI PDU, sent through a crypto module before the TCP/IP layer. In the crypto unit, each iSCSI PDU can be transformed using one of several available crypto functions. A combination of these functions can be used as well. We model this crypto processing unit using $QDSL$. Each iSCSI PDU corresponds to a request in $QDSL$. The response time of each request is the time spent in the crypto module. The CPU processing latency for a given crypto function is independent of the data bit pattern. For a given system configuration, this computation duration depends solely on the size of the data block and increases linearly with it. Similar arguments hold for the crypto processing at the client for a iSCSI WRITE operation. We note that the choice of crypto function made per iSCSI PDU at the server for a READ operation is enforced upon the client when it checks the hash or decrypts the data. We focus on the $QDSL$ system at one end of the critical path.

This is a $FixSL$ variant of the $QDSL$ system because each iSCSI PDU can be transformed using only one crypto function. We consider the revenue maximization problem while bounding the av-

erage response time per request. The revenue rate for each service level will be used to capture the desirability of the associated crypto function. The assignment of revenue rates is beyond the scope of this paper and we refer the reader to research that has attempted to come up with practical revenue values for security primitives [20, 39]. Our proposed mechanisms are generic and can accommodate any type of revenue function. The response time bound is assumed to be set by an external entity (such as a system administrator). Varying the response time bounds on the $QDSL$ system (crypto processing unit) has a direct effect on the end-to-end average response time as seen by the client. We consider the problem of setting such a bound to also be beyond the scope of our work. In our evaluation, we present results for a range of response time bounds to accommodate a variety of latency needs.

We pick four different crypto levels that vary significantly from each other both in computation time complexity and security guarantees. Table 1 shows the four crypto levels (service levels) and their average latencies (service times) measured for 16KB data blocks. We find the variance in CPU computation time for a given crypto function and a fixed data size to be very small. Therefore, in our simulation experiments we assume a fixed service time per service level when the requests are uniform in size.

We intend to capture a system where individual requests have an average response time bound and each of them needs to be necessarily encrypted with a 128 bit key. Beyond this requirement $QDSL$ formulation strives to utilize any system slack by deploying stronger encryption combined with authentication techniques. The desirability of using a crypto function increases from top to bottom as shown in table 1. We find it convenient to conduct our discussion in terms of *revenue-per-request*. Since this is a $FixSL$ system, these revenue-per-request values are easily transformed into revenue rates. This is achieved by normalizing the revenue-per-request values by the corresponding average service times to obtain the revenue rates. We consider a simple linear growth in the revenue-per-request obtained from the service levels. If an iSCSI PDU gets serviced at level L_1 , the system earns a revenue of 20 units, and similarly 40, 50, and 100 for the next three service levels L_2 , L_3 , and L_4 , respectively (the system does not earn anything when it is idle.) The discontinuity in the revenue growth at second and fourth crypto level was needed to ensure that the corresponding revenue rates increasing in a convex fashion (a requirement to ensure that all the service levels will be picked for some (arrival rate, response time bound) configuration of the $QDSL$ system).

4.1.3 Poisson Workloads

We begin our evaluation by considering synthetic workloads. It is assumed that the arrival pattern of the I/O requests is poisson in nature and that the requests are of fixed size (16KB blocks). We consider each workload with a large number (5 million) of requests. Therefore, an invocation of offline policies such as $rQDSL$ for such long run workloads provides a steady state average revenue and response time. For a variety of response time bounds, we simu-

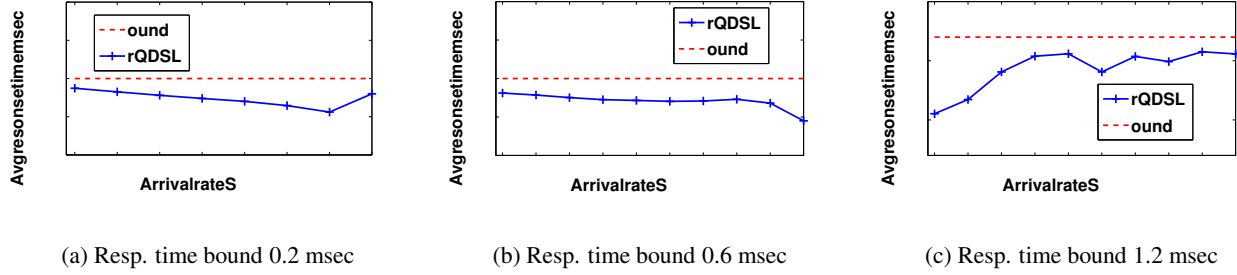


Figure 5: rQDSL policy meeting the three response time bounds for a range of arrival rates

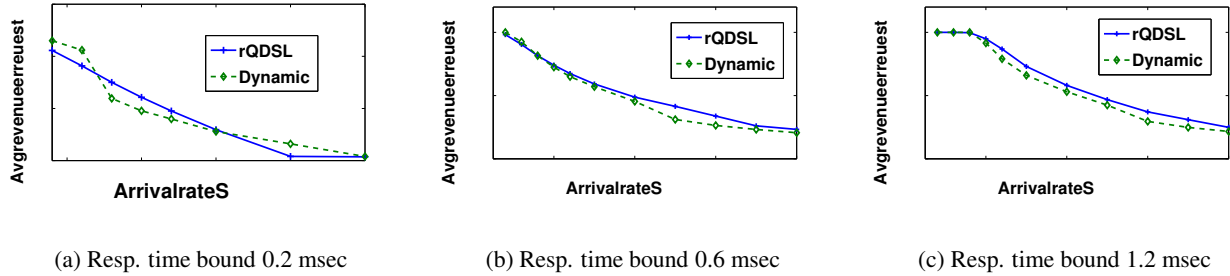


Figure 6: Comparison of the revenue yielded by rQDSL with the baseline policies for poisson workloads. A variety of response time bounds and arrival intensities were considered. The service times distribution for each level was assumed to be a fixed distribution

late the rQDSL and *dynamic* policy with different workloads that vary in their poisson arrival rates (within feasible ranges so as to be able to meet the response time bound with the lowest service level assigned to each request). We do not consider the *static* policies here because they render the system as a M/GI/1 for poisson workloads. The *dynamic* policy is optimal amongst all M/GI/1 instances of QDSL.

In each simulation, we measure the average response time and average revenue-per-request yielded by each policy. In figure 5, we show the average response times achieved by rQDSL for poisson workloads of varying intensity. We present the results for three response time bounds: 0.2 msec, 0.6 msec and 1.2 msec. For each bound, simulations were conducted within the feasible range of arrival rates. We observe that our rQDSL was able to meet all feasible response time bounds. This shows that the assumption that service times of each level have fixed distribution (as opposed to the assumption of exponential distribution in rQDSL problem formulation) does not result in a violation of the response time bound. We attribute this to the nature of the policies computed by rQDSL. The policies were found to have a threshold nature where, if the queue length keeps increasing, rQDSL picks lower service levels, thereby adjusting to short term variations in load.

In figure 6, we compare the average revenue per request obtained from the rQDSL and *dynamic* policy. We observe that the rQDSL outperforms *dynamic* policy in most of the feasible regions except for a few cases. When the response time bound is strict (0.2 msec), rQDSL is inferior to *dynamic* policy at very low and very high arrival rates (Figure 6(a)). This behavior can be explained by observing the Figure 7(a). It shows the percentage of requests in the workload that were assigned to the different service levels during the complete run. We notice that, when the response time bound

is strict, rQDSL tends towards *dynamic* policy at the two extreme ranges of the arrival rates. In addition, rQDSL is able to pick from only two service levels to meet the stringent response time bound. The *dynamic* policy switches between two service levels with an IID distribution. This is the nature of the optimal solution to a M/GI/1 assumption made by *dynamic* policy (proof omitted). A combination of these factors make the rQDSL policy conservative at these regions and cause it to be inferior to *dynamic* policy. When the response time bound becomes less stringent, rQDSL explores more possibilities in terms of the number of service levels that can be used in a workload run. In addition the system queue length varies across a larger range and this results in a further deviation from a M/GI/1 system. Queue lengths vary in a self-similar manner and this combined with the fact that rQDSL picks threshold based policies result in service level selections across requests to be less independent of each other. The consequence of that is shown in figures 6(b) and (c). rQDSL outperforms *dynamic* policy in the entire feasible region of arrival rates and the improvement in average revenue over *dynamic* policy tends to increase with more relaxed response time bounds. An interesting property is that rQDSL provides better revenue than the *dynamic* policy even though the latter tightly meets the response time bounds. This shows that rQDSL is able to utilize the server resources in a more efficient manner.

4.1.4 TPC-H Traces

In this section, we empirically evaluate the efficacy of rQDSL for real workloads. We would like to study how rQDSL performs for real workloads with arbitrary arrival patterns. Specifically, we are interested in knowing whether rQDSL violates the specified response time bounds and if so, by how much. Then, we compare the average revenue over the baseline policies. In the current work, we evaluate workloads that exhibit reasonably steady arrival rates over

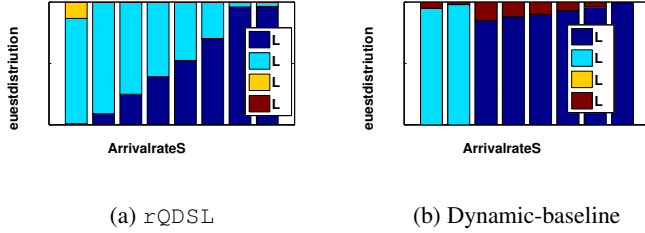


Figure 7: Comparison of request distribution across different service classes resulting in a $rQDSL$ system with that in one using the *dynamic* policy. Response time bound is 0.2 msec in all these simulations.

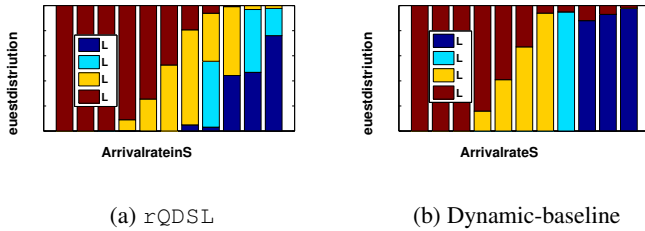


Figure 8: Comparison of request distribution across different service classes resulting in a $rQDSL$ system with that in one using the *dynamic* policy. Response time bound is 1.2 msec in all these simulations.

reasonably large time windows. For workloads that vary their average arrival rates over time, we propose $rQDSL$ separately invoked for each time window that has a steady arrival rate. Trade-offs in choosing the time window scales are not considered in this work.

Our set of simulations employs well-regarded storage traces to construct realistic arrival processes to our $qSecStore$ system. We employ the TPC-H storage benchmark [37]. TPC-H illustrates decision support systems that examine large volumes of data, execute database queries with a high degree of complexity, and give answers to critical business questions. The TPC-H benchmark provides data request traces for 22 different types of queries. We pick traces for two queries (Q12 and a subset of Q5) with similar arrival rates but with different degrees of burstiness in the arrival process (as captured by the Hurst parameter [31].) Due to their identical arrival rates, choosing this pair of queries provides allows us to isolate and compare the impact of burstiness (self-similarity is a commonly observed phenomena in real workloads) in the arrival process on the performance of our policies. In these experiments, since the arrival characteristics are fixed by the traces, we vary the response time bounds.

The well-known Hurst parameter lies in the range [0.5, 1.0]; higher values of the Hurst parameter capture larger burstiness in arrivals. Q12 has a mean inter-arrival time of 0.5 msec and a Hurst parameter of 0.55—it exhibits little variation in inter-arrival times (i.e., is not very bursty.) The significantly more bursty Q5 trace has a mean inter-arrival time of 0.5 msec and a Hurst parameter of 0.85. The service time of a disk request is proportional to its size and was computed per service level from the table 1. We found that more than 95% of the requests were of 32KB size. Each trace, in fact, has a stream of requests for size 256KB that are split up into 32KB requests across 12 different disks. Because of this, the

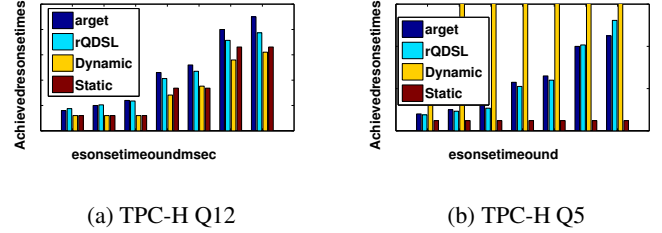


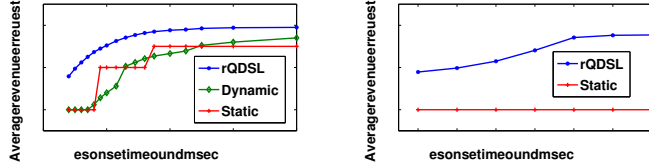
Figure 9: Comparison of the techniques in meeting a range of response time bounds when running the two TPC-H queries

condition that the inter-arrival time between request i and $j, j > i$ very often does not depend on the service time of request i . As a result, we replay these traces without considering any dependencies between requests, and expect this to not cause significant inaccuracies. We evaluate the performance of $rQDSL$ with the baseline policies for the following cases. We employ a modified *dynamic* policy here that models the system as a GI/GI/1 queue (for which a closed-form expression exists for an upper bound on the response time [22].) An implicit assumption by the *dynamic* policy is that the input workload has independent inter-arrival times.

In figure 9, we show the achieved response times by $rQDSL$ and the baseline policies for both the queries when the specified bounds were varied. We first note that $rQDSL$ achieves average response times closer to the specified bound for both the queries. We note however that for the uniform (less bursty) Q12 trace, $rQDSL$ violates the response time bound when it is very stringent. As the bounds are relaxed, it meets them with increasing slack from the bound. However, on the other hand, for the more bursty trace Q5, we noticed that $rQDSL$ violated the bounds at high (very relaxed) response time bounds. We observe that the *dynamic* policy meets the response time bounds in all cases for the Q12 trace. This is because the trace for Q12 does exhibit lack of correlations between the inter-arrival times. However, for the bursty workload Q5, the approximation of GI/GI/1 fails and *dynamic* policy does not meet any of the response time bounds.

In figure 10, we compare the average revenue per request obtained from the policies. We note that $rQDSL$ outperforms both *static* and *dynamic* policies in all the cases. *dynamic* policy is not shown in the figure 10(b) because it fails to meet the response time bounds for the trace Q5. The difference in revenue between $rQDSL$ and the other approaches increases from a less bursty to a more bursty trace. This shows the need for a policy that can dynamically adapt to variations in the instantaneous load of the system. $rQDSL$ achieves that for all response time bounds by picking service levels according to the system queue length. Finally, in figure 11, we show the distribution of service levels that were selected during the runs of the traces. The key difference in the distributions between the two types of queries is that for the bursty trace Q5, the two extreme levels L_1 and L_4 occur more frequently than for a corresponding experiment for a less bursty trace Q12. This happens due to the drastic variations in system load seen in bursty workloads. When the response time bound is relaxed, $rQDSL$ policy tends to oscillate between the extreme service levels which are picked at the extreme queue lengths. For a more uniform trace like $rQDSL$, on the other hand, the queue length variations are small and therefore $rQDSL$ spends more time in a narrow range of service levels.

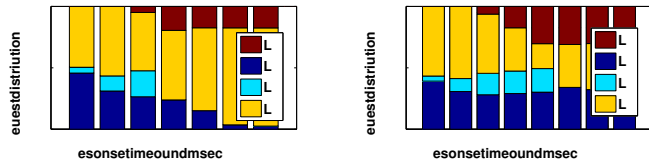
We conclude the experimental section on $qSecStore$ by noting that when the assumptions about exponential service times and



(a) TPC-H Q12

(b) TPC-H Q5

Figure 10: Comparison of the techniques in terms of average revenue obtained per I/O request when running the two TPC-H queries. A range of response time bounds were considered as shown.



(a) TPC-H Q12

(b) TPC-H Q5

Figure 11: Distribution of requests that were serviced at the four crypto levels when using $rQDSL$ for the two TPC-H queries

poisson behavior in I/O workloads are relaxed, the $rQDSL$ policy still performs satisfactorily by being able to meet the response time bounds closely (indicating high system utilization) and outperforming the baseline policies in the average revenue per request. We considered workloads that were uniform over long runs and showed that $rQDSL$ can utilize the short-term variations in the system load to exploit any system slack available at a fine granularity. The intrinsic nature of $rQDSL$ that it operates between service levels based on the instantaneous queue length enables it to adapt well to real workloads. The assumptions about poisson and exponential service times behavior in the estimation of the optimal policy does not seem to cause big violations. In fact it serves as a very good approximation in order to be able to determine the actual threshold values in a real world implementation.

4.2 $qPowServer$: A Power-aware DVFS-capable Server

4.2.1 Problem Context

In our second case study, we consider a power management problem in CPUs that are equipped with multiple frequency/voltage levels (DVFS states). DVFS serves as a mechanism to provide trade-offs between performance and power consumption. By choosing a higher DVFS state, the CPU can provide faster execution to a CPU-intensive task at the expense of higher power consumption. We are interested in designing a server that will employ DVFS modulation to ensure that its operation does not result in the power consumption exceeding a specified “power budget.” We desire that the server provide the best performance under this budget constraint. This corresponds to the “Response Time Minimization” variant of QDSL.

4.2.2 Adapting $rQDSL$ to $qPowServer$

An application execution can be viewed as consisting of CPU bursts interspersed with intervals where the CPU is idle. Each CPU burst is a request in our mapping. Note that $qPowServer$ is well-captured as a $VarSL$ system since the CPU can change its DVFS state any time independent of scheduling of individual CPU bursts. Changing the DVFS state requires a switching to kernel mode and back. It is known that these operations pose minimal overhead and can be invoked at millisecond granularity [2]. Our policy will suggest DVFS states based on the number of CPU bursts pending in the system.

Capturing $qPowServer$ using QDSL poses difficulty when there is only a single thread of execution in the system. In such an environment, a CPU burst does not arrive until the previous burst finishes processing. Similarly, for applications that saturate the CPU, the power-performance trade-off is trivially resolved. Therefore, we focus on a server running multiple applications/threads that alternate between using the CPU and blocking on I/O operations. Many important applications likely to run in energy-conscious environments behave in this manner. Web-based ECommerce applications and data-intensive scientific applications are two good examples.

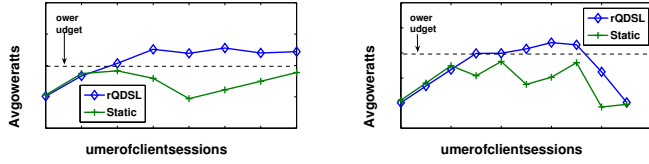
DVFS states (GHz)	Avg. power (W)
3.4	Idle Power + 60
3.2	Idle Power + 45
3.0	Idle Power + 35
2.8	Idle Power + 20

Table 2: Details of the four service levels in $qPowServer$. Idle power is constantly expended in the system at a rate of 160W.

We assume a server with the DVFS-capable Intel Xeon dual processor. We consider four DVFS states, a subset of the states provided by this processor. Table 2 presents these states and the average peak-power consumption of TPC-W for each. We measured these power consumptions by running an instance of TPC-W on a Dell PowerEdge SC1450 server in our lab that contains the above-mentioned processor. We choose the TPC-W benchmark for our evaluation [35]. TPC-W specifies an ECommerce workload that simulates the activities of a retail Web site that produces heavy load on a back-end database. Multiple instances of TPC-W applications are assumed to be hosted on our server. Each instance handles 10 simultaneous client browsing sessions. The load on the server is controlled by varying the number of instances hosted on the server. We map these four DVFS states to service levels and the corresponding power measurements to their revenue rates in $VarSL$. An interesting aspect of this mapping concerns the treatment of idle power that is expended at all times by the CPU. The idle power component (160 W for our processor) is added to the revenue rates in all states, including that representing an empty (idle) system. Measurements of the TPC-W application with 10 sessions revealed a mean inter-arrival time between CPU bursts of 350 msec and a burst parameter of 0.66. Also, the mean CPU burst length was around 20 msec when running at DVFS 3.4 GHz. This corresponds to the average service time for the service level corresponding to DVFS 3.4 GHz. We calculated these values from an offline analysis of the power time series sampled once every 2 msec over a period of 2 hours. The service time values for the other three levels were found to be 21.25, 22.67, and 24.28 msec, respectively.

4.2.3 Salient Results

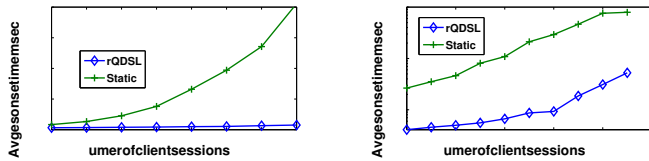
We compare the performance of DVFS modulation policy yielded



(a) Power budget = 175 watts

(b) Power budget = 190 watts

Figure 12: The average power consumption with TPC-W subjected to a wide range of workload.



(a) Power budget = 175 watts

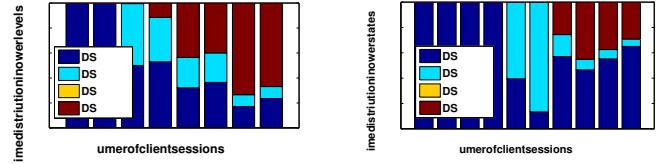
(b) Power budget = 190 watts

Figure 13: The average response time of CPU bursts with TPC-W subjected to a wide range of workload.

by our adaptation of VarSL with the *static* policy. We omit the results for *dynamic* policy as it failed to meet the power budget in most cases because of the bursty characteristics of the TPC-W application. There are two aspects to this comparison: success in meeting the power budget and average response time. We conduct this comparison for two power budgets: (i) a stringent budget of 175W (only 15W more than the idle power of 160W) and (ii) a less stringent budget of 190W. We assume the I/O power consumption to be negligible compared to CPU power—a realistic assumption [27].

We simulate both the policies within a wide workload range (30 sessions to 100 sessions) and present numbers for average power consumption and response time. As shown in Figure 12(a), for the stringent power budget of 175W, our policy causes violations beyond a load of 50 sessions. However, these violations were very small (not more than 2-3W). We attribute these violations to the fact that the inter-arrival process of the CPU bursts is far from Poisson (Hurst parameter of 0.66.) These small violations, however, are accompanied by substantial improvements in response time over the baseline policy as shown in Figure 13(a). For the highest workload intensity we experimented with, our policy improved average response time considerably while consuming not more than 5W additional power. Similar comparative trends are seen for the more relaxed power budget of 190W. While our policy still improves response time, it causes negligible violation of power budget.

Finally, Figures 14(a) and (b) show the relative percentages of different DVFS states employed by our policy for the two power budgets of 175W and 190W, respectively. One interesting observation is that DVFS 3.0 GHz was never used. Our policy chooses DVFS states in such a way that the system either operates at the two most power-expending states or drops to the lowest DVFS 2.8 state. We explain this by noticing that DVFS 3.0 state does not achieve sufficient reduction in power when compared with the improvement in the service time it provides to a CPU burst. More generally, this



(a) Power budget = 175 watts

(b) Power budget = 190 watts

Figure 14: Percentage break-down of number of CPU bursts served using different DVFS states for the two chosen power budgets.

resonates with the intuition that a large number of DVFS states very close to each other in terms of power/service times do not provide significant improvement in response time over a smaller number of more widely-spaced states.

5. RELATED WORK

Systems provide differential levels of service at multiple time scales. They range from diverse domains such as data center resource management [13], Web servers [3, 7, 24], differentiated IP networks [14, 21], sensor networks [6], streaming media servers [12] etc. In the context of Internet Data Centers (IDC), decisions are continuously made to effectively utilize the resources. Typically these decisions are semi-static and are made at a coarse granularity. For example, existing research has addressed the issue of job assignments to servers [32], controlling the number of active servers [11, 36] etc. A majority of these techniques employ some sort of a pricing/optimization model to determine the amount of resources that need to be allocated to each application to meet the specified guarantees on the application performance. In the context of web servers, there are proposed techniques to provide differential selection among heterogeneous servers based on their network locations [7], capabilities [17, 4] etc. These decisions are made at a finer time granularity of individual sessions or requests. For example, the work [42] addresses the issue of service selection algorithms for web services used to meet end-to-end bandwidth and latency constraints.

At very fine granularities such as disk requests, one way to provide differential service levels to individual requests is via scheduling. For example, scheduling algorithms such as WFQ [1], SFQ [28] can dynamically divide up the server bandwidth according to the demands of the individual request patterns. These techniques are complementary to our proposed approach. We consider a single class of requests which can be serviced independently at multiple service levels to utilize the system resources more effectively. Exploring the trade-offs between security and performance for storage data has been studied in research such as [30]. We exploit these trade-offs at a very fine granularity. Existing research on dynamic power management in systems ranging from embedded devices to high-end servers have focussed on techniques that switch amongst power states at coarse time scales [25, 10, 29]. Other specific domains where QDSL can be mapped are applications that perform dynamic data compression techniques for providing trade-offs between the transmission delay and computation. Existing work on these domains [23, 15] examine such trade-offs by analyzing possible combinations and providing dynamic solutions based on the current system behaviour (e.g network congestion).

Anytime algorithms [43], developed in the AI/Robotics commu-

nities, are concerned with problems of some similarity to QDSL. These are used in domains with real-time constraints where a satisfying answer, which falls within the range of tolerance of error and is available within acceptable time, is preferred to the best-possible correct answer requiring a large amount of time. It would be interesting future work to explore the applicability of our techniques to problems in this domain.

6. CONCLUSION

Our work was motivated by the observation that many modern computing systems are faced with a trade-off between the output quality they generate for a given input and the amount of computing resources they spend processing it. Often an improvement in the Goodness of Service (GoS) comes at the price of degraded performance in the form of reduced throughput or increased response time. Examples of systems where such trade-offs between GoS and performance arise are numerous and span diverse domains.

We formulated and studied QDSL, a class of constrained optimization problems, that captured such trade-offs. We found two variants of such systems worth studying: (i) VARSL in which a single request might be serviced at multiple levels during its lifetime and (ii) FIXSL in which the service level might not change during the lifetime of a request. Our approach involved reducing restricted versions of such systems to MDPs. Our modeling revealed that optimal service level selection policies in these systems corresponded to very simple rules that can be implemented very efficiently. We evaluated the efficacy of the service level selection policies yielded by these reductions in solving two real world problems (i) qSecStore, an iSCSI-based secure storage system that has access to multiple encryption functions, and (ii) qPowServer, a server with DVFS-capable processor.

Acknowledgements

We thank our shepherd Mor Harchol-Balter for her generous help that substantially improved this paper. We also thank our anonymous reviewers for their insightful comments. This work has been supported by NSF grants 0621429 and 0615097. Bhuvan Urgaonkar's research was supported in part by NSF grant CNS-0720456 and a research grant from Cisco Systems.

7. REFERENCES

- [1] S. Keshav A. Demers and S. Shenker. Analysis and simulation of a fair queuing algorithm. In *Journal of Internetworking Research and Experience*, 1990.
- [2] F. Bellosa. Process cruise control: Throttling memory access in a soft real-time environment. In *Technical Report TR-14-97-02, Univ of Erlangen-Nuernberg, IMMD IV*, 1997.
- [3] N. Bhatti and R. Friedrich. Web Server Support for Tiered Services. In *IEEE Network*, 1999.
- [4] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The eclipse operating system: Providing quality of service via reservation domains. In *Proceedings of the USENIX Annual Technical Conference*, 1998.
- [5] S. Chaitanya, K. Butler, A. Sivasubramaniam, P. McDaniel, and M. Vilayannur. Design, implementation and evaluation of security in iscsi based network storage systems. In *Proceedings of the Workshop on Storage Security and Survivability*, 2006.
- [6] D. Chen and P. K. Varshney. QoS support in wireless sensor networks: A survey. In *Proceedings of the International Conference on Wireless Networks*, 2004.
- [7] J. Chuang and M. Sirbu. Distributed Network Storage with Quality-Of-Service Guarantees. In *Journal of Network and Computer Applications*, 2000.
- [8] Thomas B. Crabill. Optimal Control of a Service Facility with Variable Exponential Service Times and Constant Arrival Rate. In *Management Science*, May 1972.
- [9] CSIM. <http://www.mesquite.com/>.
- [10] P. Lewis D. Grunwald and K. I Farkas. Policies for dynamic clock scheduling. In *Proceedings of the Usenix Symposium on Operating Systems Design and Implementation*, 2000.
- [11] Ejasent. Utility Computing White Paper. November 2001.
- [12] D. McNamee et al. Control challenges in multi-level adaptive video streaming. In *Proceedings of 39th IEEE Conference on Decision and Control*, 2000.
- [13] J. Chase et al. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the ACM Symposium on Operating System Principles*, 2001.
- [14] Y. Bernet et al. A Framework for differentiated services. In *Internet draft: draft-ietf-diffserv-framework-02.txt*, 1999.
- [15] Zhang et al. Dynamic selection and effective compression of key frames for video abstraction. In *Pattern Recognition Letters*, 2003.
- [16] E.A Feinberg. Optimal control of average reward constrained continuous-time finite markov decision processes. In *Proceedings of the 41st IEEE Conference on Decision and Control*, 2002.
- [17] P. Drushel G. Banga and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Usenix Symposium on Operating Systems Design and Implementation*, 1999.
- [18] Jennifer M. George and J. Michael Harrison. Dynamic Control of a Queue with Adjustable Service Rate. In *Operations Research*, 2001.
- [19] R A Howard. Dynamic probabilistic systems. vol. ii: Semi-markov and decision processes. In *Series in Decision and Control*, 1971.
- [20] C. Irvine and T. Levin. Toward a Taxonomy and Costing Method for Security Services. In *Proceedings of the Annual Computer Security Applications Conference*, 1999.
- [21] F. P. Kelly. Charging and Rate Control for Elastic Traffic. In *European Transactions on Telecommunications*, 1997.
- [22] L. Kleinrock. *Queueing Systems, Volume 1: Theory*. 1975.
- [23] Chandra Krintz and Brad Calder. Reducing delay with dynamic selection of compression formats. In *HPDC*, 2001.
- [24] K. Li and S. Jamin. A Measurement-based Admission Controlled Web Server. In *Proceedings of IEEE INFOCOM*, 2000.
- [25] J. R Lorch and A. J Smith. Improving dynamic voltage scheduling algorithms with pace. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2001.
- [26] M. Kistler M. Elnozahy and R. Rajamony. Energy conservation policies for web servers. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, 2003.
- [27] E. Elnozahy T. Keller M. Kistler C. Lefurgy R. Rajamony F. Rawson P. Bohrer, D. Cohn and E. Hensbergen. Energy conservation for servers. In *Proceedings of the IEEE Workshop on Power Management for Real-Time and Embedded Systems*, 2001.
- [28] H. Vin P. Goyal and H. Cheng. Start-time fair queuing: a scheduling algorithm for integrated services packet switching networks. In *IEEE/ACM Transactions on Networks*, 1997.
- [29] Q. Qiu and M. Pedram. Dynamic power management based on continuous-time markov decision processes. In *Proceedings of the Design Automation Conference*, 1999.
- [30] Erik Riedel, Mahesh Kallahala, and Ram Swaminathan. A framework for evaluating storage system security. In *Proceedings of the FAST 2002 conference on File and Storage Technology*, 2002.
- [31] O Rose. Estimation of the hurst parameter of long-range dependent time series. 1996.
- [32] V. Rykov and D. Efronin. Numerical Analysis of Optimal Control Policies for Queueing Systems with Heterogeneous Servers. In *Kalashnikov Memorial Seminar*, 2002.
- [33] Christopher M. Sadler and Margaret Martonosi. Data compression algorithms for energy-constrained devices in delay tolerant networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, 2006.
- [34] Linn I. Sennott. Average Cost Optimal Stationary Policies in Infinite State Markov Decision Processes with Unbounded Costs. In *Operations Research*, volume 37, 1989.
- [35] W. Smith. Tpc-w: Benchmarking an ecommerce solution.
- [36] Synchron. Synchron Enterprise Manager. 2001.
- [37] TPC-H. <http://www.tpc.org/tpch>.
- [38] T. Abdelzaher K. Skadron V. Sharma, A. Thomas and Zhijian Lu. Power-aware qos management in web servers. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, 2003.
- [39] C. Wang and W. A. Wulf. Towards a framework for security measurement. In *Proceedings of the National Information Systems Security Conference*, 1997.
- [40] Richard Weber and Shaler Stidham. Optimal Control of Service Rates in Networks of Queues. In *Advances in Applied Probability*, volume 19, 1987.
- [41] Charles P. Wright, Michael C. Martino, and Erez Zadok. Ncryptfs: A secure and convenient cryptographic file system. In *Proceedings of the USENIX 2003 Annual Technical Conference*, 2003.
- [42] T. Yu and K. J. Lin. Service Selection Algorithms for Web Services with End-to-End QoS Constraints. In *Proceedings of the IEEE International conference on E-Commerce Technology*, 2004.
- [43] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. In *American Association for Artificial Intelligence*, 1996.