Fast and Robust Signaling Overload Control

Sneha Kasera, José Pinheiro, Catherine Loader, Mehmet Karaul, Adiseshu Hari and Tom LaPorta
Bell Laboratories, Lucent Technologies
{kasera,jcp,cathl,karaul,hari,tlp}@lucent.com

Abstract

Telecommunication switches implement overload controls to maintain call throughput and delay at acceptable levels during periods of high load. Existing work has mostly focused on controls under sustained overload–they do not meet the demands of modern telecommunication systems where the increased number of services and mobile subscribers often creates fast changing hot spots. We introduce new algorithms that are designed to be highly reactive to sudden bursts of load.

One algorithm is a modified version of RED for signaling traffic that measures the queue size. The second algorithm uses two measures: call acceptance rate and processor occupancy. Using simulations of realistic system models, we compare these new algorithms with each other and an existing algorithm that uses processor occupancy only. Our simulation results and qualitative arguments show that the combination of acceptance rate and processor occupancy results in a highly reactive and robust signaling overload control.

In our performance study, we consider two scenarios, one where no processing costs are associated with call requests that are throttled and another where finite processing overhead is incurred even for calls that are eventually throttled. In the second scenario, due to the processing overhead, the system throughput approaches zero under heavy overload because most or all of the processing resources are expended in throttling calls. In order to significantly reduce the processing overhead associated with throttled calls, we propose a simple two-layer throttling scheme where a fraction of calls are throttled at a lower protocol layer.

1 Introduction

Telecommunication networks are made up of various switches interconnected by communication links. These links comprise two separate logical networks. One network is used for carrying voice and user data, and the second for signaling and control information. To establish and release connections, and to access services and databases, switches communicate with each other over the signaling network. These networks and switches are engineered to carry a certain number of active calls, and process requests for calls and services at a certain rate. Occasionally, they might experience more traffic than the engineered capacity. Overload controls are required to maintain the throughput and the quality of service at acceptable levels. Using properly designed overload controls is much cheaper than over-provisioning resources and this may also be used for load balancing and traffic differentiation.

In this paper, we address processor overload controls due to signaling traffic. These controls execute inside switch controllers to react to conditions of high signaling requests that overwhelm internal switch processing resources. Switches may experience overload conditions even at times during which the voice and signaling networks are not congested. Ideally, the performance of the switch should degrade gracefully, according to performance profiles acceptable to a service provider. Also, the performance of the switch should return to normal as soon as the overload condition declines.

There are two types of response to overload. First, a switch may invoke *remote* congestion controls by signaling its neighbors of its state. In practice, although recommended procedures exist to deal with inter-switch congestion control, many deployed switches do not implement these algorithms. Therefore, it is critical that switch overload algorithms make no assumptions about the behavior of neighboring switches. The second response to overload is for the switch to *locally* protect its own processing resources by selectively throttling signaling messages. Each request to a switch, either to establish or release a connection, or to request a different type of service, usually results in a sequence of several messages, each of which must be processed in order for the request to be filled. Therefore, the process of throttling messages may be done intelligently by throttling

messages that initiate service requests. This practice reduces overload by eliminating future signaling messages.

Earlier work on overload control has focused on preserving performance under sustained overload. In modern telecommunication networks, due to the increased number of service types and usage, and introduction of mobile networks that often result in fast changing network hot spots, the reactiveness of overload algorithms has become critical. This requires new algorithms and models. In this paper, we introduce new *local* overload control algorithms that are designed to be highly reactive to sudden bursts of load. Our first algorithm is based on detecting and controlling overload by measuring queue-lengths. Here we look at a modified version of the well-known Random Early Discard (RED) algorithm [6]. The RED algorithm has been proposed for active queue management in routers. We call our RED variant signaling RED, or SRED. The second, called Acceptance-Rate Occupancy (ARO), uses system measures of both call arrival rates and switch processor occupancy. Using simulations of realistic system models, we compare these new algorithms with each other and an existing algorithm that uses only processor occupancy [4].

We find that under sudden load ramp up, ARO and SRED reduce the response time by orders of magnitude in comparison to the algorithm that uses processor occupancy only. In comparison to SRED, ARO has a slightly higher response time under sudden load ramp up but exhibits higher throughput under heavy overload. SRED and other variants of RED require tuning of several parameters, some of which change with processor speeds and different software releases. ARO requires specification of only one parameter, the target processor occupancy, which is dimensionless, making this approach more portable and robust to system upgrades.

The choice of a particular overload control algorithm depends upon several factors. Based on our performance study and noting that it could be portably implemented, we recommend the use of the ARO algorithm for processor overload control due to signaling traffic.

In our performance study, we consider two scenarios, one where no processing costs are associated with calls that are throttled and another where finite processing overhead is incurred even for calls that are eventually throttled. In the second scenario, due to this processing overhead, the system throughput approaches zero under heavy overload. This is because most or all of the processing resources are expended in throttling calls. In order to reduce this processing overhead, we propose a simple two-layer throttling scheme where a fraction of calls are throttled at a lower protocol layer. This two-layer throttling scheme could potentially reduce the processing overhead by a significant amount and thereby increase the system's prowess of handling higher overload.

The rest of the paper is structured as follows. Section 2 describes related work in overload control algorithms. A high level description of a network switch and the system model used is presented in Section 3. The overload algorithms considered in the paper are described in Section 4 and compared via simulations in Section 5. The two-layer throttling scheme to reduce processing overhead is described in Section 6. Our conclusions and suggestions for further research are included in Section 7.

2 Related Work

Overload and congestion control (OLC) has been a topic of active study both in circuit switched (Telecom) and packet switched (Internet) networks. Telecom signaling network overload control schemes, including the ones discussed in this paper, are designed to manage *processor utilization*. The goal is to allow as high a processor occupancy and, hence, signaling throughput as possible, without overloading the processor.

Some of the earlier work [10, 5] (see [9] for a summary) on overload control in telecom switches focused on delaying dial tones (useful only in the case of a local access switch), or throttling new calls when the number of calls being processed in the switch exceeded a certain predetermined number. Such *drop tail* schemes suffer from synchronization effects in which remote switches might reduce, or increase, sending traffic at the same time, causing degraded performance. A later proposal by [4] suggested gradually throttling a fraction of new calls depending on the measured processor occupancy, which is an indication of the system load. As shown in the performance study in Section 5, one of the problems with this approach is that it reacts slowly to sudden ramp up in input traffic. Earlier work on modeling and analyses of overload control [9] has concentrated on average value analysis with Poisson assumptions.

Our work differs from the existing work on overload control in telecom networks in the following significant ways. First, we propose a new algorithm, which uses both the processor occupancy and the acceptance rate of the system. Under steady overload, the performance of our new algorithm is similar to the algorithm based only on processor occupancy. However, our new algorithm drastically reduces the response time under sudden load ramp up. Second, our approach applies to both access and toll (or tandem) switches. Third, we do not restrict our performance study to theoretical analyses; we use simulations to consider a variety of arrival patterns and

study a variety of overload control schemes. Most importantly, we study the reactiveness of the overload control approaches.

Very recently, Pillai [8] has examined two overload control schemes using an M/M/1 queuing model. Both the schemes, one that uses arrival rate threshold and another that uses buffer size threshold, work in conjunction with delay thresholds. Delay thresholds are hard to budget across multiple switching hops and even across components in the same switch. The problem becomes harder when different end-to-end paths have varied number of switching hops. Hence we do not consider delay-based approaches in our work. Our belief is corroborated by Pillai's observation that average end-to-end delays are very different from the end-to-end delay bounds leaving ample room for further tuning of individual thresholds. Our work differs from Pillai's in many additional aspects. Unlike our overload control schemes, Pillai's buffer size threshold scheme uses drop tail throttling. His arrival rate threshold scheme could use random throttling but the computation of drop probability and the random throttling alogorithm are not specified in [8]. As mentioned before, we do not restrict ourselves to Poisson assumptions and also consider bursty call arrivals. Our algorithm design emphasizes robustness of the overload control. We use a richer system model that treats calls as made up of several subtasks with variable delays in between their execution. We also propose a two-layer throttling scheme to reduce the processing overheads associated with calls that are eventually throttled.

In the Internet, overload control has focused on the *active queue management* (AQM) of the output links of routers. Most of the proposed approaches to AQM are variants of the *Random Early Discard* (RED) algorithm [6]. RED uses a probabilistic approach to dropping packets, in which the drop probability depends on the average queue size. In AQM schemes, processor utilization is not an issue. It is assumed that the link, rather than the processor, is the bottleneck and there is always enough processing power to handle packet forwarding at full link rate while implementing the AQM scheme. To study the applicability of RED-like approaches in the context of controlling processor overload due to signaling traffic, we modify RED. Our modified version, called signaling RED, or SRED, uses the basic concept of measuring queue lengths like RED, but differs from it in many other ways as indicated in Section 4. The performance of RED has been mainly evaluated in the presence of TCP traffic with TCP's end-to-end congestion control. In our evaluation of SRED, we do not assume any remote overload controls.

3 System Model

A typical network switch is shown in Figure 1. It consists of several components including those dealing with signaling, call processing and administrative functions. The switch also has resources for bearer (voice or data) traffic. The signaling component processes signaling traffic associated with call setup and call teardown from/to neighboring switches and is responsible for any signaling protocol processing. It also processes messages from/to the call processing and administrative components. The call requests from the signaling card are sent to a call processing component which maintains call states and implements call processing functions and any other upper layer protocol processing.

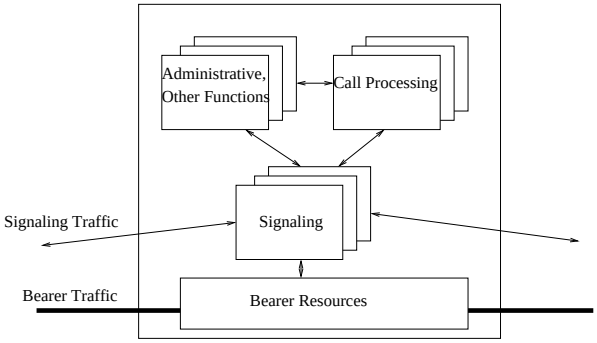

Figure 1. A Typical Network Switch.

We are concerned with signaling overload control at the signaling and the call processing cards. When a signaling message arrives at a signaling card, the signaling card must first perform lower layer protocol processing (e.g., Message Transfer Part, MTP, layers 1-3 processing [1]) and any other processing to determine if the signaling message is a new call request. Hence even if a new call is throttled due to overload in the signaling card, it imposes some processing cost on the signaling card. This processing cost is called the throttling cost. A call processing card may avoid this cost by communicating its load (or the rate at which it would accept calls) to the signaling card(s) which identifies and throttles new calls appropriately. The division of functionality across cards varies in different switch implementations. In our work, we study processor overload control in a single card of a network switch under two general scenarios representing the two throttling situations describe above, one in which throttling comes for free (where throttling cost is incurred by a neighboring card), and another where the card incurs throttling cost [1]. We

[1] A hybrid of the two scenarios is also possible but will be ignored in this paper.

assume that there is enough memory available so that the card does not run out of buffers even during overload.

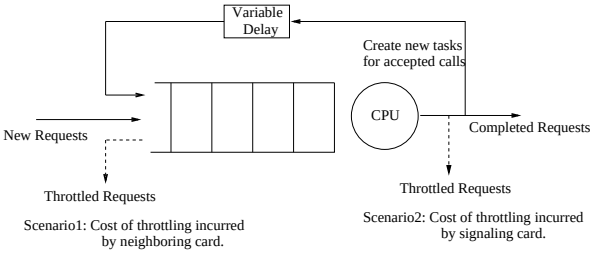

Figure 2. Single processor queue model for a card.

We model a card, also referred to as *system*, by a single processor queue as shown in Figure 2. In this model, new call requests are assumed to arrive at the queue according to some arrival process. Depending on the current measure of load of the system and the overload control algorithm, a new request could be accepted or throttled. If the new request is accepted, it is processed by the CPU and, depending on the nature of the request, an additional task is generated and fed back into the system after a variable delay. This additional task, when processed, may generate more tasks which are fed back into the system. The processor schedules tasks using a first-in-first-out policy. Eventually, when all the tasks associated with an accepted request are executed, the request is considered completed and removed from the system. The generation of new tasks models the multiple tasks and messages associated with a request. The variable delay applied to a task, before it is fed back into the system, models the variable delay between arrival of different tasks associated with a request. The two throttling scenarios are also shown in Figure 2. In the first scenario, a new call is throttled in a neighboring card and hence is shown to be throttled even before it is queued. In the second scenario a new call is throttled after incurring some processing cost.

4 Overload Control Algorithms

In this paper, we describe algorithms to locally control processor overload in a system (a card of a network switch as described in Section 3) due to excessive signaling traffic.These algorithms are based on gradual throttling of input traffic and use a time varying variable f, which is interpreted as the *fraction allowed*, or the probability that a new request will be accepted. Under normal operation, $f = 1$, meaning that all traffic will be let in. When the switch enters in overload, f is reduced to control the load on the switch by throttling fewer or more new calls. It is assumed that ongoing calls can not be terminated to reduce overload.

All overload control algorithms considered in this paper are *time driven*: the system performance is measured at fixed *probe intervals*. At each k^{th} probe time, denoted the *assessment time*, overload is assessed and f possibly modified. Different system measures may be used to evaluate overload. These system measures, described later in this section, are evaluated at each probe time and summarized (usually by averaging) at each assessment time. Let $\boldsymbol{x}_n$ represent the vector of summarized system measures observed at the n^{th} assessment time. The general form of the overload control algorithms considered in this paper is $f_{n+1} = g(f_n, \boldsymbol{x}_n, \boldsymbol{\theta})$, where g is the feedback control function, assumed non-increasing in each element of $\boldsymbol{x}_n$, and $\boldsymbol{\theta}$ is a vector of control parameters (*e.g.*, target values of system measures). In this paper, we only consider algorithms for which g is continuous; discrete feedback control algorithms, which use discontinuous g, tend to be less stable and have worse overall performance than continuous feedback control algorithms. We present three overload control algorithms in the following three subsections.

4.1 Occupancy Algorithm

We first present the processor occupancy-based algorithm proposed by [4] and discuss its applicability under different types of input traffic. Processor occupancy, ρ, is defined as the percentage of time, within a given probe interval, that the processor is busy processing tasks. Processor occupancy is a dimensionless quantity, which makes it relatively system independent. An occupancy close to 100% indicates that the switch is fully loaded and corrective action is needed.

The algorithm proposed by [4] specifies a target processor occupancy ρ_{targ}, such that, if the estimated processor occupancy $\widehat{\rho}_n$ at assessment time n (given by the average of the last k probed processor occupancies) is below ρ_{targ}, the fraction allowed f is increased, or unchanged (if already at 100%), otherwise, if $\widehat{\rho}_n > \rho_{\text{targ}}$, f is decreased. The feedback control function for this algorithm is defined below.

$$f_{n+1} = \begin{cases} f_{\min}, & \phi_n f_n < f_{\min} \\ 1, & \phi_n f_n > 1 \\ \phi_n f_n, & \text{otherwise.} \end{cases},$$

where $\phi_n = \min(\rho_{\text{targ}}/\widehat{\rho}_n, \phi_{\max})$. A minimum fraction allowed $f_{\min}$ is used to prevent the system from throttling all incoming calls.

We observe that, because ρ cannot exceed 100%, f can decrease by at most $(1 - \rho_{\text{targ}})100\%$ between successive assessment times, which may lead to a slow

reaction to overload conditions. For example., using $\rho_{\text{targ}} = 0.9$ allows f to decrease by at most 10% between assessment times. This problem is worse under overload caused by a sudden increase in call load, because of the reaction delay associated with ρ. However, the fraction allowed can increase by up to $\phi_{\max}100\%$ between successive assessment times, so that the algorithm reacts faster to ceasing overload conditions.

4.2 Signaling RED Algorithm

The RED algorithm [6] has been proposed for managing router queues in IP networks. We describe an adaption of RED for our signaling application, which we call Signaling RED (SRED) algorithm. Like RED, the SRED algorithm uses as a measure of overload an Exponentially Weighted Moving Average (EWMA) estimate Q_n of the average queue length, based on the measured lengths q_n of the new calls queue, defined as $Q_{n+1} = (1-w)Q_n + wq_n$. The updating weight w needs to be specified. The simulation results in Section 5 indicate that values of w in the range 0.01–0.1 (considerably larger than the ones proposed for RED) are appropriate. The feedback control function for the SRED algorithm is

$$f_{n+1} = \begin{cases} f_{\min}, & Q_n \geq Q_{\max} \\ 1, & Q_n \leq Q_{min} \\ \max\left(f_{\min}, \frac{Q_{\max}-Q_n}{Q_{\max}-Q_{\min}}\right), & \text{otherwise} \end{cases}.$$

To keep the algorithm close in spirit to the original RED algorithm, the fraction allowed is updated at each probe interval, when a new q_n is measured.

The basic differences between SRED and RED are:

- SRED is *time-driven*, while RED is *event-driven*: in SRED, q_n is measured and Q_n and f are updated at each probe interval, while RED does this is at every new call. SRED has smaller overhead.
- Because SRED is time-driven, no special treatment is required for the case when the queue is empty ($q_n = 0$); this is naturally handled by the EWMA estimate Q_n.
- SRED uses a deterministic throttling scheme, described in the Section 4.4, which produces more uniform throttling sequences and requires less processing than the probabilistic throttling scheme used in RED.

The main advantage of this algorithm is that it is based on a measure that reacts very fast to the onset of overload, as evidenced by the simulation results in Section 5. The new calls queue length is directly associated to the processor overload and it appears to have little latency in reacting to changes in overload conditions. The basic drawback of SRED is that it requires the specification of parameters ($w, Q_{\min}$, and $Q_{\max}$) which heavily depend on the processor capacity and traffic characteristics, making it less portable and non-robust to system upgrades.

4.3 Algorithms Based on Acceptance Rate

We now propose a new algorithm that uses two system measures, the system acceptance rate in conjunction with processor occupancy. This algorithm has the portability and robustness of the Occupancy algorithm and also reacts fast to a sudden onset of overload.

Acceptance rate is defined as the number of calls accepted by the system in a given time interval. An algorithm based on acceptance rate measure requires the specification of a target acceptance rate, which may be either fixed by design, or dynamically estimated using system measurements, as described below.

Let α_{targ} represent the target call acceptance rate for the system and $\widehat{\alpha}_n$ the estimated call acceptance rate at assessment time n (given by the average of the acceptance rates in the previous k probe intervals). A simple overload control algorithm based on acceptance rate (AR) uses the feedback control function below.

$$f_{n+1} = \begin{cases} f_{\min}, & \phi_n f_n < f_{\min} \\ 1, & \phi_n f_n > 1 \\ \phi_n f_n, & \text{otherwise} \end{cases}, \qquad (1)$$

where $\phi_n = \alpha_{\text{targ}}/\widehat{\alpha}_n$. With this feedback control function, if a large number of calls are accepted in an assessment period, then a large fraction of calls are throttled in the next assessment period. For example, if the acceptance rate in an assessment period is three times the target acceptance rate, two thirds of the calls will be throttled in the next assessment period. Thus, the acceptance rate based feedback control function reacts very fast to sudden traffic ramp up. The problem with using an algorithm based only on acceptance rate is that this measure does not indicate overload induced by internal changes in the system. An increase in the service rate for certain calls or consumption of processing resources by background tasks cannot be captured by acceptance rate. Therefore, it is necessary to combine acceptance rate with another system measure that represents the system's processed load and not just offered load. We propose to use acceptance rate in conjunction with processor occupancy. The acceptance rate–occupancy based algorithm (ARO) produces a fraction allowed $f = \min(f_A, f_O)$, where f_A and f_O are the fraction allowed using acceptance rate and processor occupancy feedback control functions respectively.

Even though ARO requires measurement of two system measures, in comparison to one in the case of SRED, it requires specification of only one parameter, the target processor occupancy. Processor occupancy is dimensionless and does not change with processor speed. Hence ARO is very robust against system upgrades.

4.3.1 Determining the Target Acceptance Rate

The target acceptance rate α_{targ} is a crucial parameter in the ARO algorithm described in (1). It may either be determined based on the engineered capacity of the system, or it may be estimated dynamically. The methodology used to estimate α_{targ} depends on whether or not there is a processing cost associated with the throttling of new calls, as described below.

No Throttling Cost

When there is no processing cost associated with throttling of new calls (Scenario 1 in Figure 2), α_{targ} can be dynamically estimated by determining the maximum system throughput $\mu_{\max} = \widehat{\alpha}/\widehat{\rho}$, where $\widehat{\alpha}$ is the current estimate of the call processing rate and $\widehat{\rho}$ is the current estimate of the processor occupancy. $\mu_{\max}$ is also the maximum acceptance rate, $\alpha_{\max}$, that can be allowed into the system. We use $\alpha_{\text{targ}} = \rho_{\text{targ}}\alpha_{\max}$. Because α_{targ} is constant under the no-throttling cost scenario, it is updated less frequently and more slowly than the fraction allowed f: α_{targ} is updated at every $K \gg k$ probe interval, according to an exponentially weighted moving average (EWMA) scheme with small updating weight. The EWMA helps in coverging to the true α_{targ} starting from some initial value.

With Throttling Cost

If throttling costs are present, under heavy overload a substantial part of the processor occupancy may be related only to the throttling of new calls, and not to the processing of existing calls. In this case, α_{targ} is no longer constant, as changes in the call load will reflect in changes in target acceptance rate, and needs to be dynamically estimated.

A first modification needed in this case is to correct the formula to calculate $\alpha_{\max}$ to take into account only the portion of $\widehat{\rho}$ that is associated with the processing of existing calls. Letting c denote the relative cost of throttling (that is, the fraction of the total service time of a call that is used, on average, just to throttle it) and f the current fraction allowed of calls, the fraction of the total processor occupancy associated with throttling is $p = c(1-f)/(f + c(1-f))$. Therefore, the total processor occupancy available for processing accepted calls is $1 - p\widehat{\rho}$, of which $(1-p)\widehat{\rho}$ is effectively being used. Hence, the equivalent processor occupancy $\widehat{\rho}_{\text{eq}}$ when throttling costs are present is

$$\widehat{\rho}_{\text{eq}} = \frac{(1-p)\widehat{\rho}}{1-p\widehat{\rho}} = \frac{f\widehat{\rho}}{f + c(1-f)(1-\widehat{\rho})}.$$

Note that, when $c = 0$, $\widehat{\rho}_{\text{eq}} = \widehat{\rho}$, as expected. The maximum acceptance rate is then estimated as $\alpha_{\max} = \widehat{\alpha}/\widehat{\rho}_{\text{eq}}$.

When the call load is in steady state, α_{targ} should remain approximately constant, so we don't want to update it too frequently or too fast. However, if the call load changes fast and significantly, then α_{targ} should be updated accordingly. In this case, the total processing cost that can be handled by the system should be kept approximately constant, so that

$$c\,(1-f_n)\,\widehat{\lambda}_n + \alpha_{\max,n} = c\,(1-f_{n+1})\,\widehat{\lambda}_{n+1} + \alpha_{\max,n+1},$$

where $f_n, \widehat{\lambda}_n$, and $\alpha_{\max,n}$ denote respectively the fraction allowed, the arrival rate, and the maximum acceptance rate at time n. This suggests the following updating scheme

$$\alpha_{\max,n+1} = \alpha_{\max,n} + c\left[(1-f_n)\,\widehat{\lambda}_n - (1-f_{n+1})\,\widehat{\lambda}_{n+1}\right], \quad (2)$$

so that abrupt increases in $\widehat{\lambda}$ will cause $\alpha_{\max,n}$ to decrease and abrupt decreases in $\widehat{\lambda}$ will have the opposite effect on $\alpha_{\max,n}$. The updating scheme for $\alpha_{\max,n}$ must serve a dual purpose: to react fast to sudden changes in $\widehat{\lambda}_n$ and to vary slowly when $\widehat{\lambda}_n$ is approximately constant. In order to achieve this, we use a hybrid scheme, applying (2) to update $\alpha_{\max,n}$ at each assessment time (thus allowing fast reaction to sudden changes in $\alpha_{\max,n}$) and using an EWMA updating scheme like the one described for the no-throttling cost case (but with $\widehat{\rho}_{\text{eq}}$ used in place of $\widehat{\rho}$) at every K^{th} interval (the slowly changing part of the updating algorithm). As before, we set $\alpha_{\text{targ}} = \rho_{\text{targ}}\alpha_{\max,n}$.

4.4 Throttling

Given the fraction allowed f_n to be used during the n^{th} interval, a throttling scheme determines which new calls to drop. We describe and compare two probabilistic and one deterministic throttling algorithm that can be used for this purpose.

For simplicity of notation, let $\tau = 1 - f_n$. Possibly the simplest probabilistic throttling algorithm is the one that uses a pseudo-random number r uniformly distributed in $(0, 1)$, as described next.

Generate $r \sim U(0,1)$.
If $r \leq \tau$, reject call
else accept call.

This algorithm, which we will denote by *Prob.*, as discussed in [6], tends to produce clusters of throttled calls, and may lead to synchronization problems. An alternative probabilistic algorithm, proposed for RED in [6] produces more uniform throttling, but results in a larger throttling fraction than the desired τ. To obtain the right τ, we corrected the RED throttling scheme by using a fraction $\tau' = \tau/(2-\tau)$. The corrected algorithm is

$c := c + 1$
Generate $r \sim U(0,1)$.
If $r \leq \tau'/(1 - c\tau')$
    reject call
    $c := 0$.
else accept call.

The variable c, representing the number of new calls that were accepted since the last throttled call, is initialized to zero.

Both the Prob. and the RED throttling schemes require pseudo-random number generation, which introduces additional variation from the desired τ, and processing overhead. In order to reduce random variation and the overhead of random number generation, we consider a deterministic throttling scheme first proposed by [7], which we denote *Determ.*. In this algorithm, a variable r is first initialized to 0, then the accept/reject decision procedure described below is used.

$r := r + (1 - \tau)$.
If $r \geq 1$
    $r := r - 1$
    accept call
else reject call.

To compare the performance of the different throttling schemes, we simulated data according to a Poisson process with an arrival rate of 100 calls/s and used the three algorithms to throttle calls using $\tau = 0.3$. Figure 3 shows the observed fraction of throttled calls per 200 ms interval for each of the throttling schemes, during the first 15 seconds of traffic.

The smallest variation associated with the Determ. algorithm is evident from Figure 3: the observed throttled fractions corresponding to this algorithm remain much closer to the target of 30% than for either of the other two algorithms. The RED algorithm represents an improvement over the Prob. algorithm, but still presents considerable variability in the observed throttling fraction. The Determ. algorithm has better performance than the other two probabilistic algorithms and does not have the overhead associated with the pseudo-random number generation. Therefore, we propose to use Determ. for throttling in our overload control schemes.

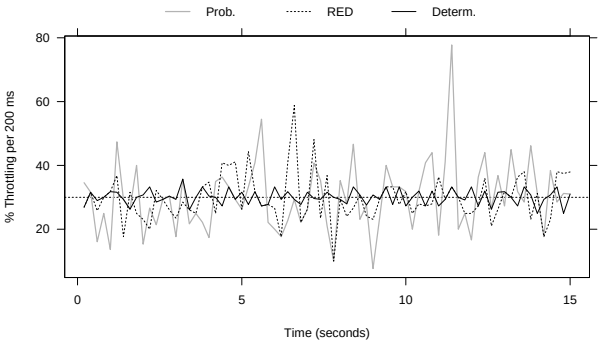

Figure 3. Throttling fractions per 200 ms for Poisson traffic with rate of 100 calls/s.

5 Performance of Algorithms

We compare the performance of the overload algorithms described in Section 4 by simulating[2] Scenario 1 and 2 of the system model described in Section 3. The simulator used to obtain the results presented in this section is custom written.

We make the simplifying assumption that the process of detecting overload is *free*. The call model used for the simulations treats each request as consisting of two task segments: call setup and call termination (see Figure 4). The call setup and call release segments comprise of several tasks which are generated after a random delay. For simplicity, we combine subtasks occurring with negligible delay of each other into one subtask, which results in the call setup being subdivided into three subtasks and the call termination being composed of a single subtask.

Figure 4. Call task and event structure.

The system represented in the simulation is designed to operate under approximately 95% processor occu-

[2]We do not attempt to derive theoretical results for the performance of the different overload control algorithms, in order to avoid making any assumptions that would limit the scope of the conclusions presented here. In particular, we study overload scenarios with non-steady call loads (see Section 5.2), which would be too difficult to analyze theoretically.

pancy under a load of 1.9 million busy hour calls attempts (BHCA), corresponding to an average of about 528 call attempts per second. The probability distributions used for the delays until the next subtask and the subtask processing times are listed in Table 1, with $\exp(\lambda)$ denoting the exponential distribution with parameter λ and $\Gamma(\alpha, \beta)$ denoting the Gamma distribution with parameters α and β.

	Subtasks			
Task	Number	Delay	Proc.	Mean
Setup	1	$\exp(250)$	$\Gamma(3, \frac{30}{11})$	1.1
	2	$\exp(7500)$	$\Gamma(2, 10)$	.2
	3	$\exp(90000)$	$\Gamma(2, 10)$	.2
Release	1	—	$\Gamma(2, \frac{20}{3})$	.3

Table 1. Probability distributions for subtask processing time and delay between subtasks.

The choice of distributions and parameter values in Table 1 is based on recommendations and measurements from telecommunications traffic engineering for wireline switches. Under these assumptions, the total average processing time per call is 1.8 ms. Thus, 100% capacity for the simulated system would be about 556 calls/s, or about two million BHCA. The holding time for a call, that is, the time between the end of the call setup and the start of the call termination, is assumed to be exponentially distributed with mean 90 seconds. We assume that new calls arrive according to a Poisson process[3]. In order to study reactiveness, sudden ramp up in new call arrival is caused by ramping up the mean arrival rate in a short time interval.

All overload algorithms considered in the simulation use the same probe intervals, 100 ms, and the same minimum fraction allowed, $f_{\min} = 0.005$. Table 2 lists the parameter values used in the simulations for the different algorithms.

The parameter values for the SRED algorithms were chosen to produce an average processor occupancy of about 95% under mild to moderate overload conditions (call rate between 550 and 1500 calls/s), for Scenario 1 of no throttling costs. As discussed in Section 5.1, due to the sensitivity of queue lengths to changing load and due to smaller assessment interval, the performance of SRED deteriorates under high overload (call rates above 2000 calls/s) and it is not possible to choose parameter values that give steady 95% occupancy for the range of call rates considered in the simulation.

[3]Other arrival processes can be easily used in our simulator.

	Algorithm		
Parameter	Occupancy	SRED	ARO
ρ_{targ}	0.95	–	0.95
$\phi_{\max}$	20	–	20
k	10	1	10
w	–	0.05	0.02
K	–	–	300
$Q_{\min}$	–	3	–
$Q_{\max}$	–	8	–

Table 2. Parameter values used in the simulations for the different overload control algorithms.

The performance metrics used to compare the algorithms are task delay (time in queue until start of processing), call throughput, and fraction of calls allowed into the system. In the rest of this section, we first describe our simulation study for the scenario where no processing costs are associated with the calls that are throttled (Scenario 1 of Figure 1). The experiments and results for the scenario where throttled calls incur processing costs (Scenario 2 of Figure 1) are described later in Section 5.3.

5.1 Performance Under Steady Load

We investigate the performance of the overload control algorithms, when call arrivals are Poisson under steady mean call attempt rates varying between 1.5 million BHCA (417 calls/s) and 9 million BHCA (2500 calls/s), covering the range from non-overload to severe overload with respect to the nominal load of 2 million BHCA assumed for the simulated system. For each mean call attempt rate, calls were simulated over a 30-minute period, with the performance metrics measured at each probe time and averaged over the whole period.

Figure 5 shows the averages of the performance metrics versus call attempt rate, for each overload control algorithm. The ARO and Occupancy algorithms perform well and similarly with respect to the performance metrics considered. The SRED algorithm is not capable of maintaining a stable performance with increasing call attempt rate. It has comparable performance to the other two algorithms up to a call attempt rate of 1500 call/s (with larger average task delays), but its performance deteriorates for higher call attempt rates (at 2500 call/s, its average process occupancy is 82% and its average throughput is 457 calls/s – about 50 call/s smaller than the average throughput for ARO and Occupancy). It should be noted that the original RED algorithm is de-

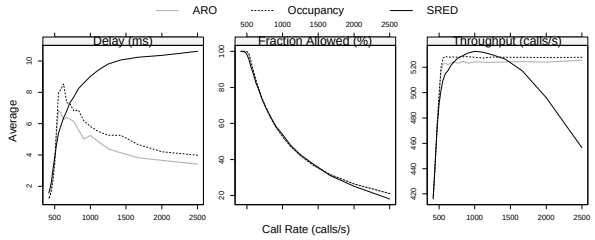

Figure 5. Average performance metrics versus call attempt rate by overload control algorithm.

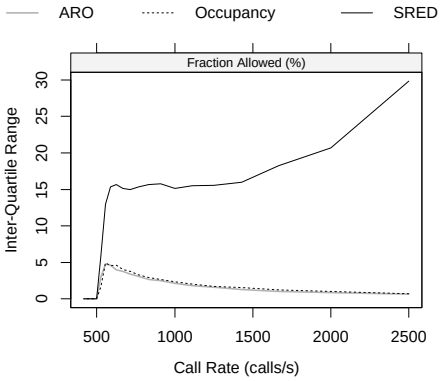

Figure 6. Inter-quartile ranges of fraction allowed versus call attempt rate by overload control algorithm.

signed to work with IP networks, where it can be safely assumed that remote overload control mechanisms will prevent the system from reaching the level of overload we consider in the simulation. In the context of signaling networks, however, such an assumption can not be made and one needs to study the behavior of overload control algorithms under severe overload conditions.

The reason for the loss in performance for SRED under high call attempt rates seems to be the instability observed for the feedback mechanism of this algorithm under these overload conditions. This is because queue length is very sensitive to changing load and the fact that SRED uses smaller assessment interval. This translates into a highly variable fraction allowed of calls. Figure 6 presents the inter-quartile ranges (*i.e.*, the difference between the third and the first quartiles, which provides a measure of variation for the variable under consideration) of the fractions allowed f_n observed over time during the simulation, for the different call attempt rates. The SRED fraction allowed variation is about ten times larger under heavy overload than the variation corresponding to the other two algorithms. In order to further explore this issue we study the behavior of the algorithms over time.

We consider the case of a steady state call rate of 7.2 million BHCA (2000 calls/s), considerably above the nominal capacity of 2 million BHCA (556 calls/s). Figure 7 shows the behavior of the performance metrics (given as averages per second), after the third minute of operation, for the three overload control algorithms.

The ARO and Occupancy algorithms have similar, good performances with respect to all three metrics, with Occupancy showing more variability with respect to task delay. The SRED algorithm shows consistently higher (but less variable) task delays, but substantially more variation in fraction allowed. As mentioned in Section 5.1, the high fraction allowed variability leads to a decrease in average throughput (under 2 million BHCA,

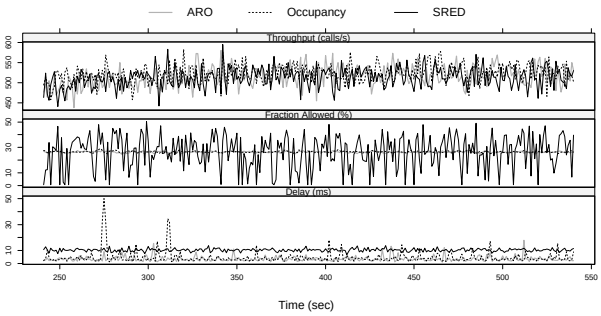

Figure 7. Evolution of performance metrics under a steady load of 7.2 million BHCA.

the average throughput is about 525 calls/s for ARO and Occupancy and about 500 call/s for SRED). At this level of overload, the performance of SRED begins to show clear signs of deterioration, as indicated in Figures 5, 6, and 7.

5.2 Performance Under Non-Steady Load

In the scenario investigated here, the call process operates at a mean call rate of 1.8 million BHCA (500 calls/s) up to 300 seconds, at which point the mean call rate increases to 7.2 million BHCA (2000 calls/s) in 1.5 seconds, stays at that level for two minutes, and then drops back to 1.8 million BHCA in 1.5 seconds. The arrival process for each mean call rate is Poisson but the overall arrival process is not Poisson anymore. The objective of this simulation is to study how fast the overload algorithms are to react to a sudden onset of overload and to a sudden cessation of overload. Figure 8 presents

the evolution of the performance metrics for the three algorithms.

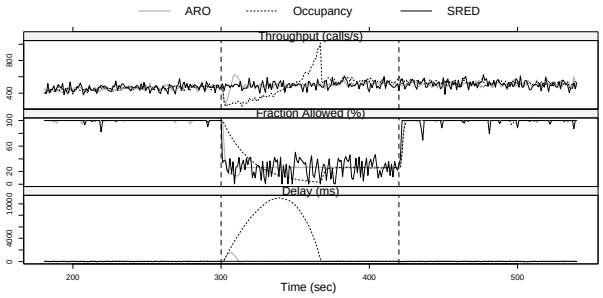

Figure 8. Evolution of performance metrics under non-steady overload.

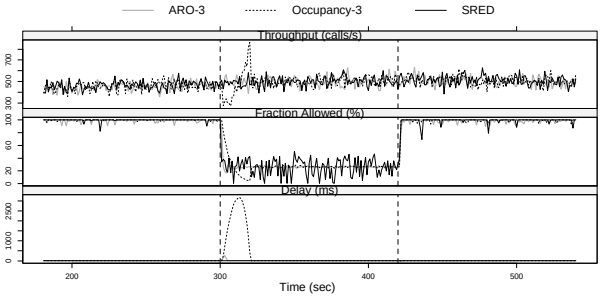

Figure 9. Performance metrics under non-steady overload with modified ARO an Occupancy algorithms.

The SRED algorithm has the best overall performance with respect to task delay under this overload scenario. The Occupancy algorithm has the worst task-delay performance, taking more than one minute to recover from the call rate ramp up and experiencing delays of up to 11 seconds. The ARO algorithm has a much better performance than the Occupancy algorithm, but not quite as good as SRED: about 12 seconds to recover and maximum delay of 1.5 seconds. Once again, the SRED algorithm displays a marked increase in variability of fraction allowed under overload, which is not observed for the other two algorithms. All three algorithms show almost immediate recovery when the system goes from overload to non-overload.

There are two main reasons for the better task-delay performance of SRED: the greater sensitivity to overload conditions of queue length, the system measurement used in SRED, and also the fact that $k = 1$ (*i.e.*, the fraction allowed changes at every probe interval) for this algorithm. To get a fairer comparison of the algorithms under this scenario, while keeping reasonable variation in f, we consider versions of ARO and Occupancy with $k = 3$, which we denote ARO-3 and Occupancy-3, respectively. Figure 9 shows the performance of these algorithms and SRED.

The task-delay performance of the ARO-3 algorithm is now nearly identical to that of SRED: 3 seconds of recovery time and maximum task delay of 245 ms. The Occupancy-3 has much better task-delay performance than the original Occupancy algorithm (recovery time of 20 seconds, maximum delay of 3.1 seconds), but is still substantially worse than the other two algorithms. The basic reason for this is the greater latency of processor occupancy as a measure of overload, compared to either queue length or call acceptance rate. As expected, the use of $k = 3$ for ARO and Occupancy also resulted in greater feedback variability, but this variability is similar to that of SRED under non-overload conditions and much smaller in the case of fraction allowed during overload. Alternative values of k may be investigated to strike a better balance between speed of reaction and feedback variability for ARO and Occupancy, but this is left for future research.

In summary, under sudden load ramp up, ARO and SRED reduce the response time by orders of magnitude in comparison to the algorithm that uses processor occupancy only. In comparison to SRED, ARO experiences a slightly higher response time under sudden load ramp up but exhibits higher throughput under heavy overload.

5.3 Throttling Cost Case

We now describe the experiments and results of simulations of the system when finite processing costs are incurred even for calls that are eventually throttled. A new call request that is eventually throttled may undergo several layers of protocol processing and result in generation of a release message which also goes through several layers of protocol processing. Based on processing cost measurements on experimental prototypes in our simulation experiments, the relative cost of throttling (defined in Section 4), c, is set to 1/3. This means that the processing cost incurred by a throttled call request is about 33% of the processing cost incurred by a call request that is accepted.

Figure 10 shows how the average performance of the three algorithms changes with increasing call attempt rates under steady overload. The experimental setup for this figure is same as that of Figure 5 with the exception that now we also consider processing costs associated with throttled calls. As expected, the average throughput starts decreasing close to the maximum system capacity (556 calls/s) and it approaches zero when the call at-

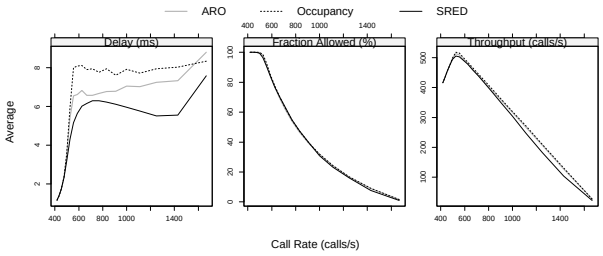

Figure 10. Average performance metrics vs call attempt rate under steady overload.

tempt rate is about three times the system capacity. This is because the processing associated with throttled calls effectively reduces the processing prowess of the system and when the call attempt rate is three times the system capacity, almost all the processing power is used in throttling calls and no new calls are accepted.

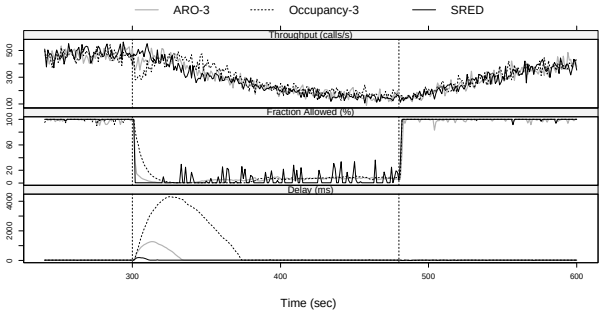

Figure 11. Performance metrics vs time under non-steady overload.

Figure 11 shows how the performance metrics of the three algorithms change with time under non-steady overload conditions. The input traffic pattern for this figure is same as that for Figure 11. Interestingly, in comparison to the delay curve in Figure 11, the delay curve in Figure 11 shows that it takes longer to recover from sudden overload when processing costs are associated with throttled calls. Another observation is that due to smaller probe intervals, SRED reacts much faster than ARO. This is because in the throttling cost scenario the sensitivity to system capacity estimation increases. SRED uses a smaller assessment interval (100ms) in comparison to ARO (300ms) and is therefore quicker in estimating the reduced capacity. When we reduce the assessment time for ARO from 300ms to 100ms we find that the difference in response to sudden overload between SRED and ARO reduces considerably. This behavior is shown in Figure 12.

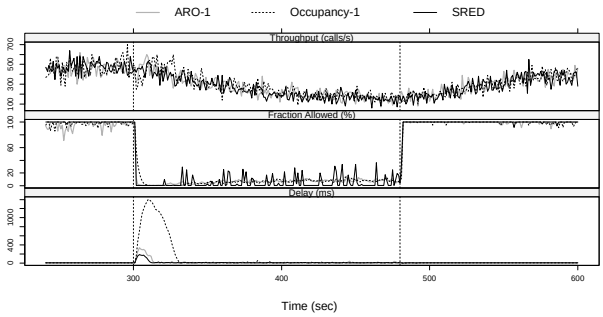

Figure 12. Performance metrics vs time under non-steady overload with assessment period = 100ms.

In summary, inclusion of throttling costs results in reduced throughput for all the three algorithms. It also results in reduced responsiveness under sudden overload. The responsiveness could be improved by reducing the assessment period. The relative throttling cost, c, plays an important role in determining the system performance especially the load at which the system throughput approaches zero. In the next section we propose a new approach to dynamically reduce the cost of throttling, taking the amount of overload in the system into account.

6 Two–Layer Throttling

The cost of throttling depends on the complexity of the throttle mechanism. Traditionally, in telephone networks that implement the Signaling System 7 (SS7) protocol stack, calls are throttled at the ISUP (Integrated Services Digital Network User Part [2]) layer by sending a call release, REL, message to the adjacent switch. The throttling cost is significant in this case due to the protocol processing at the protocol layers up to ISUP and the cost of generating and sending the REL message in response to each incoming call that is throttled. We refer to this throttling mechanism as the *release* mechanism. A simpler form of throttling is to simply discard incoming calls. This can be done at the MTP3 (Message Transfer Part 3 [1]) layer, that is just below the ISUP layer in the SS7 stack, before the call is handed over to the ISUP layer for processing. This avoids the complexity and cost of ISUP level processing including generation of REL messages. We refer to this mechanism as the *discard* mechanism. In this case, the throttling cost is simply the cost of parsing the signaling request in order to identify whether it is a candidate for throttling or not,

which is significantly less than that of the release mechanism.

REL messages could be used to carry the level of congestion at an overloaded switch[4]. The neighboring switches could use this congestion level information in the REL messages to reduce the call requests toward the overloaded switch provided they implement remote overload control. When calls are simply discarded at the MTP3 layers the congestion level information is not available at the neighboring switches. In order to reduce processing overhead and at the same time be able to send some REL messages to facilitate remote overload control when it is implemented, we are interested in throttling solutions which incorporate both the *release* and *discard* mechanisms — we refer to this as *two–layer throttling*.

If d is the fraction of incoming calls that are discarded, and r is the fraction of calls that are released, and if f is the fraction of calls allowed into the system, then we have $(1 - f) = d + r$.

Two layer throttling raises the question as to the relative proportion of d and r. The possible solutions for d and r can be visualized using Figure 13 . In this figure, the X-axis shows the fraction of throttled calls $(1 - f)$, while the Y-axis shows the fraction of calls released (r), discarded (d) and not allowed into the system $(r + d)$. The diagonal line represents $r + d$, since by definition, $(1 - f) = r + d$. A sample d curve is plotted in Figure 13. As can be seen, the fraction of released calls (r) is the amount bounded by the d curve and the diagonal. Figure 13 also shows a specific point d_o on the d curve and the corresponding release fraction value r_o. The set of possible d curves is constrained to lie in the region bounded by the X-axis and the diagonal, and the corresponding r curve is simply the difference between the d curve and the diagonal.

Even though several choices for d and r are possible, a good solution should have the following properties:

- as the load increases (and f decreases), d should increase,
- sufficient REL messages should be transmitted to facilitate remote overload control, i.e., we should not simply discard all traffic, and
- oscillations should be avoided.

We propose a simple solution that, based on preliminary analytical analysis, seems to exhibit all the desirable properties. This is based on the observation that $\frac{r}{r+d}$ represents the fraction of released calls amongst the throttled calls. When f is small, implying that the system is heavily overloaded and most calls are being throttled, we desire this fraction to be small, since the cost of r is higher than the cost of d. Conversely, as f becomes larger, we want this percentage to increase. The simplest solution is to let $\frac{r}{(r+d)} = f$. In this case, $d = (1 - f)(1 - f)$ and $r = (f)(1 - f)$. Figure 14 illustrates our initial solution. As can be seen, at low loads, the fraction of calls discarded is less, but as the load increases, the fraction of calls discarded increases. However, even as the load increases, the fraction of released calls does not go to 0 until the state of complete overload. We believe that this implementation of two–layer throttling provides an acceptable compromise between lowered cost of throttling at high loads and sufficient RELs being sent at lower levels of overload. The impact of the number of release messages could be appropriately evaluated only in conjunction with a remote overload control scheme which is beyond the scope of this paper.

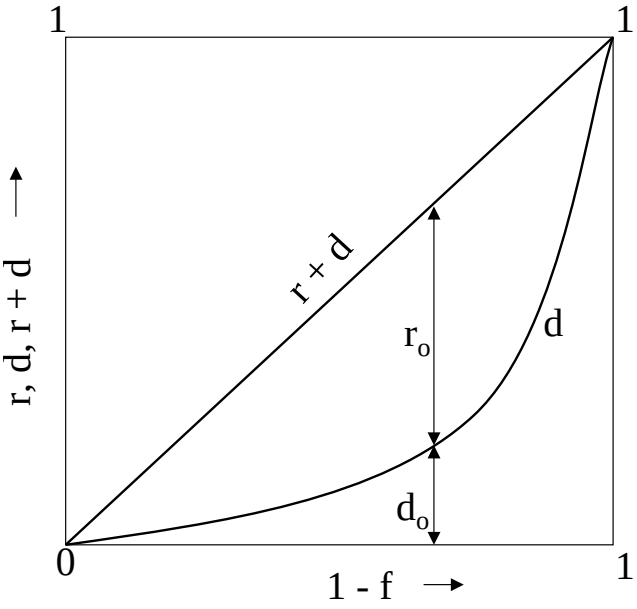

Figure 13. Two Layer Throttling - Solution Space

We end this section by demonstrating the benefit of two–layer throttling with a simple numerical example. The throttling cost parameter, c, described earlier, could be adjusted by finding the average processing cost associated with a throttled call as $(c_r r + c_d d)/(r + d)$. Here, c_r and c_d denote the processing costs of releasing and dropping a call, respectively. Typically, c_d is much less than c_r, so that the overall cost will decrease as d dominates r. Based on the values used in Section 5, we find that the average processing cost, c_r, of receiving a new

[4]More details on the use of congestion levels for remote overload control could be found in [3].

call request at ISUP and immediately releasing it is 0.6 ms. The cost of discarding a new call request at MTP3, c_d, is 0.1 ms. When the fraction of call allowed into the system, f, is equal to 30%, then using the discard and release functions proposed above, $d = 0.49$ and $r = 0.21$. With these parameters, the average processing cost associated with throttling a call reduces to 0.25 ms from 0.6 ms. Therefore the relative throttling cost, c, with two–layer throttling, reduces to 0.14 from 0.33. This reduction in c suggests that the system throughput would approach zero only when the offered load is about seven times more than the engineered capacity of the system.

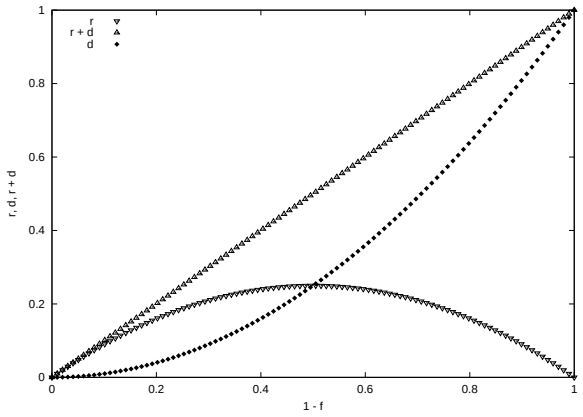

Figure 14. Two Layer Throttling - Proposed Solution

7 Conclusions

We proposed approaches for controlling processor overload due to excessive signaling traffic in a network switch. Our algorithms are designed to be highly reactive to sudden bursts of load. One algorithm, called Signaling Random Early Discard (SRED), is an extended version of RED for signaling protocols. The second, called Acceptance-Rate Occupancy (ARO), uses system measures of both call arrival rates and switch processor occupancy. Using simulations of realistic system models, we compared these new algorithms with each other and an existing algorithm that used only processor occupancy. Based on our performance study and noting that ARO is more portable and robust to system upgrades, we recommend the use of ARO for processor overload control. We also proposed and demonstrated the benefit of a two-layer throttling scheme to reduce the processing overhead associated with calls that are eventually throttled.

Future work can proceed in several directions. First, we have considered overload control in a single card of a switch. A switch typically contains several cards. We believe that our single queue model could be easily extended to a network of queues. Second, there is a need to study the local overload control algorithms examined in this paper in conjunction with remote overload control, especilly in the presence of two–layer throttling. We also need to model call retries. Third, we have considered only one class of calls in this paper. A network switch might offer different classes of calls, some more important than the others. For example, in wireless switches, different traffic types such as location updates and short message services might be given different priorities with respect to regular voice calls. It would be important to extend our work to consider multiple classes of input traffic.

References

[1] Specifications of Signaling System no. 7 - Message Transfer Part. ITU Recommendations Q.701–Q.709, 1993.

[2] Specifications of Signaling System no. 7 - ISDN User Part functional description. ITU Recommendations Q.761, 1999.

[3] Specifications of Signaling System no. 7 - ISDN User Part signaling procedures. ITU Recommendation Q.764, 1999.

[4] B. L. Cyr, J. S. Kaufman, and P. T. Lee. Load balancing and overload control in a distributed processing telecommunications system. United States Patent No. 4,974,256, 1990.

[5] B. T. Doshi and H. Heffes. Analysis of overload control schemes for a class of distributed switching machines. In *Proceedings of ITC-10*, Montreal, June 1983. Section 5.2, paper 2.

[6] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[7] B. Hajek. External splitting of point processes. *Mathematics of Operations Research*, 10:543–556, 1985.

[8] R. Pillai. A distributed overload control algorithm for delay-bounded call setup. *IEEE/ACM Transactions on Networking*, 9:780–789, 2001.

[9] M. Schwarz. *Telecommunications Networks: Protocols, Modeling and Analysis*. Addison-Wesley, 1988.

[10] B. Wallstrom. A feedback queue with overload control. In *Proceedings of ITC-10*, Montreal, June 1983. Section 1.3, paper 4.