Handling Request Variability for QoS-max Measures

Pedro Furtado
University of Coimbra
pnf@dei.uc.pt

Abstract. We denote as QoS-max the control of a request processing system to try to maximize QoS qualities and we focus on external, non-intrusive approaches with statistics on readily measurable quantities. In order to do this, the controller characterizes requests in terms of response times (or resource use) and uses that characterization to try to achieve QoS-max. However, measures vary both between different requests and for different runs of the same request. In this paper we show how we incorporated these for robust statistical QoS-max control. We use a simulator and requests with varied arrival and duration distributions to show the effectiveness of the variability handling approach.

Keywords: Performance, Transactional Systems, Autonomic.

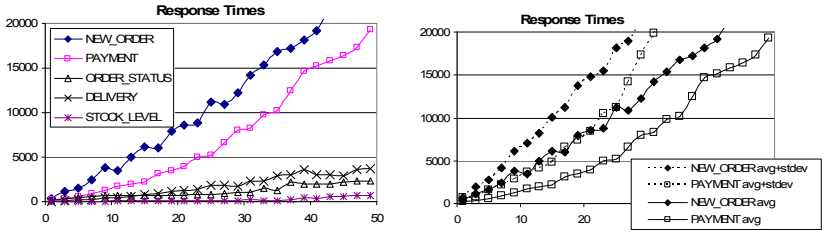
1 Introduction

As we move from statically controlled systems to systems offering autonomic functionality and from standalone systems to virtualized environments where services can run in any platform, it becomes necessary to devise mechanisms that can learn from runtime conditions and provide QoS features. In traditional systems, control was enforced by static configuration parameters such as a limit on the number of sessions or transactions that could be accepted simultaneously. With the recent trend towards embedding autonomic functionality into systems, adaptability enables the system to adjust more tightly to working conditions, which may also change with time. Expressive constraints can be used for QoS control, like for instance, requests should be answered in less than 5 seconds, the miss rate on that deadline should not be above 15% and the throughput should be as large as possible. We emphasize a special requirement for the QoS control system: that it should be readily deployable in any request processing system without any change. Autonomic QoS Control requires a model to predict what would be the response to possible parameter adaptations. Prediction has been proposed using tools that include analytic models, resource capacities and request demands for those resources [2, 14] or response time statistics [8]. Whatever the approach that is used, variability is a practical fact, which is not due to the types of measures taken, but rather to characteristics of requests, systems and running conditions, so that total precision is not possible in most practical environments and approaches should account for variability. Some causes of variability in response time or resource demands by requests include system properties such as caches (e.g. part of indexes or data may be in the cache of a DBMS or not), locks, system overheads that are external to the request processing system, request properties (e.g. a transaction that submits an order may include a single or

twenty order lines), or multiple request run patterns (combinations of multiple requests running simultaneously, each at a different execution point). The characteristics of variability are such that no practical model with reasonable overhead may account for each (e.g. it would not be practical to account for which items are cached or the resource usage patterns at the precise running point of each running request).
Execution statistics from TPCC [19] are used to show the variability within and among requests. After presenting a basic control design, we show how statistics can be collected and processed to have effective control with highly variable workload characteristics. A simulation is used to test the robustness of the approach against varied workload conditions, with different arrival and duration distributions.
The paper is structured as follows: in section 2 we analyze the response time variability within a TPCC workload. After presenting the QoS control approach in section 3, we discuss how the collected statistics are used in a robust, high-variation resilient QoS controller in section 4. In section 5 we present experimental results which show, using multiple distributions, that the robust approach is able to deal with variability satisfactorily. In section 6 we review related work section 7 concludes the paper.

2. Workload Variability

One of the most important control knobs for performance and QoS in a transactional system is the number of requests running simultaneously, which we denote as Execution Cardinality (EC), because the response times of requests are very influenced by that variable, as shown in Figure 1a for the TPC-C workload [19]. From the point-of-view of response time of a request arriving and executing alone, the best EC would be 1. However, because requests arriving have to wait in a queue until they can execute, a much higher value of EC than 1 typically allows the system to explore its parallelism and resource utilization features better (e.g. while one thread waits for I/O the other one computes or does I/O on another disk). The best EC value actually varies with conditions (requests being submitted, database size, system utilization, among others). With QoS objectives (execution of all requests should take less than 2 seconds) and statistic information we can have a dynamic control of EC, but then the statistic information becomes very relevant.
Our objective in this section is to analyze experimentally the variability of the TPC-C [19] workload in a test case. We used a system running on a Pentium 4 with 1GB of memory and PostgreSQL version 8.1. The experiment consisted in logging response times for several runs of the transaction mix with incrementing EC values, from 1 to 50. Figure 1a shows the average response time versus EC and Figure 1b shows the standard deviation of response times for the two slowest transactions. From Figure 1a we can see that average response times increase steadily and in some cases (e.g. NEW_ORDER) almost linearly as EC increases. Looking at Figure 1b for two of the transactions and in Figure 2 for all transactions (with EC=9), we can see that the standard deviation is quite large.



(a) Average Response Times (b) Average+ Stdev

Figure 1: Response Times versus EC

In Figure 2 the standard deviation was actually larger than the average for all transactions except NEW_ORDER.

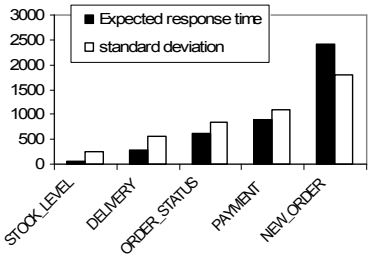


Figure 2: Response Time Statistics EC=9

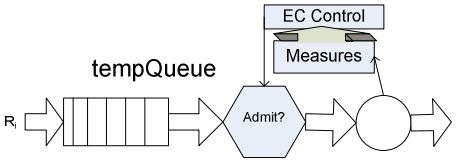


Figure 3: Controller Architecture

The possibility of very disparate response times between different requests is obvious and expected, as some requests have to process a lot of information while others implement simple and fast tasks. This depends on the application scenario. But the observed disparate response times for different runs of the same request are also to be expected, because there are many physical details that determine the response time. For instance, the backend DBMS may have the required data already in its cache or not (for performance reasons, DBMSs have large caches and both table and index data maybe or not be in the cache), an index may be used or not (selectivity issues), there may be more or less contention for resources or locks over data. The transaction itself may process more or less data (e.g. a new order may process one or 100 order lines). Given the objectives of having an external, non-intrusive and readily deployable control that does not go into system or application details, what would be a robust prediction procedure? Due to the high variability of the measured item, the model must account for that variability using conservative statistical parameters.

3. Basic QoS_{max} Controller

Figure 3 shows the controller architecture, which includes a temporary queue, an admission procedure, a monitor collecting measures and a control element. In this system, requests can be at any of the following states:

Queued: the request is waiting to be served. Each request has a wait deadline (e.g. 100 msecs or the largest of 100 msecs and the statistical response time measure for the request). If it is not served within that deadline, the request is not admitted.

Rejected: the request may not be accepted into execution due to excessive congestion of the system;

Executing: the request is executing. It may have missed its deadline or not;

Missed: the request is no longer executing and missed its deadline. The deadline may be "soft", meaning that the request still runs to completion, or "hard", meaning that it should be dropped if it reaches the deadline (and possibly rollbacked). We consider mostly soft deadlines, but both alternatives can be used;

Ended Within Deadline (EWD): requests that executed to completion and did not miss their deadline.

We now describe two alternatives to control EC by finding a maximum utility value for it (EC_{max}). The first alternative chooses EC as the value obtaining maximum throughput (this is comparable to the strategy used in [17] and [8]). Then we propose the more dynamic alternative seeking for a QoS_{max} utility function.

EC(T) Strategy: one important quantity is the EC that maximizes the throughput without considering deadlines, which we denote as EC_T. A strategy that tries to maximize that throughput can simply fix EC_{max}= EC_T, which is obtained using EC-increment test runs such as the one we ran for the results of section 2. Figure 4 shows the throughput curve using TPC-C running on a Pentium 4 with 1GB of memory, 4 disks and PostgreSQL version 8.1. In this case EC_T=20.

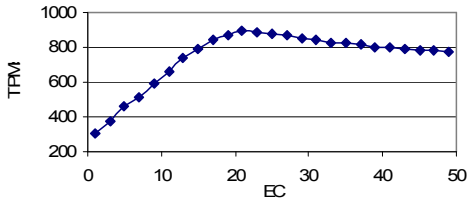


Figure 4: Throughput Curve to determine EC_T

The EC(T) strategy is adequate when we do not consider that requests should not take more than a certain time to run (deadlines).

E(ES) Strategy: this approach searches the EC value that allows more requests to end within their time constraint (deadline). It tests the utility function for each candidate EC within a pre-defined interval [EC_{low}, EC_{high}] and chooses the one that maximizes the utility function. We used default values EC_{low} = 4 (as it should not be too low to avoid rejecting too many requests) and EC_{high}= 50. The initial EC_{max} value ($EC_{default}$) is set to EC_T.

The approach assumes statistics are collected in the form $RT(r_i,EC_i)$: request r_i has response time statistic value $RT(r_i,EC_i)$ when the execution cardinality is EC_i.

Consider the current (within a recent interval) workload $\{r_i,f_i\}$, where r_i is a request and f_i is its frequency in the workload. Consider also that each request r_i should

execute within time t_i. For an EC_{max} candidate EC_j we compute the utility function U_j value as $EC_j \times \sum ft_i$, where ft_i is 0 if the statistic response time for request r_i is above t_i when EC_j is considered or the request frequency f_i otherwise.

Algorithm:

For each candidate EC_j
 U_j=0;
 For each Request r_i in workload W,
  If ($RT(r_i,EC_j)<t_i$) U_j+= $EC_j \times f_i$
Choose max { U_j}

The test runs that obtain $RT(r_i,EC_j)$ are to happen offline (e.g. during the night at weekends) (some EC_j values are tested, the remaining interpolated). If a request does not have offline test values yet, it counts as if its deadline is just above its response time with $EC=EC_T$ (so that EC_T would be a good choice for it).

Queue handling: the time waiting in the queue adds to the total response time of requests, even though the request is not being served while in the queue. The primary purpose of the queue is to hold requests temporarily during request arrival bursts, but the maximum amount of time waiting should be constrained. The queue may implement any policy, with FIFO often being the policy of choice, but there needs to be an additional control on the size of the queue to reject requests based on the waiting time. In our proposals this is implemented by means of an Admittance Decision Time limit (ADT) that is contracted for all requests, for an individual application or for a specific request. For instance, ADT<=100ms or ADT<=max{100ms,10%RT}, where RT is the expected runtime of the request. Given this parameter, the request waits in the queue for at most ADT and if it is not accepted until then, it is immediately rejected.

Control rate: the frequency of control (number of times the algorithm runs) can be set as desired, but in our experimental work we observed that the overhead is relatively modest if the workload is not too large, so the control can run for instance every 100 msecs (this is the value we used in the experiments) or every second. The size of the workload window depends on this choice. If the frequency of control is large, the system will be able to tune itself to current workload conditions.

4. Accounting for Variability

Intra-request variability refers to the response time variability among different runs of a request. It may lead to inadequate setting of EC_{max} and consequently to much higher miss rates than desired. We propose a simple conservative estimation to deal with this problem. Inter-request variability also poses a relevant issue concerning response time and throughput when EC is constrained: fast executing requests may be blocked by slower ones executing. In this case we propose a simple scheme to have a balanced execution between slow and fast requests.

Intra-request Variability: consider a miss rate (default=0%). Deadlines are specified for requests and the QoS_{max} Control functionality should try to achieve those while

maximizing the throughput. This means that the EC_{max} value should not be such that, due to request variability, many more requests miss their deadlines than desired. Given that very precise prediction of actual running times is impractical for reasons we discussed before, the solution is for the control approach to be sufficiently conservative to avoid excessive miss rates. In control algorithm, the response time was taken from values $RT(r_i,EC_i)$, which is a statistic response time value. While it could be the expected response time (average of observed values), the more conservative estimate requires other measures to be used instead. If $E(RT(r_i,EC_i))$ is the expected response time and the standard deviation is $S(RT(r_i,EC_i))$, a simple more conservative estimate is to use $E(RT(r_i,EC_i)) + S(RT(r_i,EC_i))$. In this case the offline test runs must collect these two quantites, as depicted in Figure 1b; A much more conservative statistic would be obtained by using the maximum observed response time $\max(RT(r_i,EC_i))$. The most flexible solution is for a control configuration parameter to specify a desired percentile in the observed response times distribution. That solution requires much more information, a histogram of response times for each pair $<r_i,EC_i>$.

Inter-request Heterogeneity: given strict deadlines, the QoS control may constrain EC to a small value (e.g. $EC_{max}<10$) to achieve QoS_{max}. This in turn may lead to a blocking behavior, as slow running requests may "block" fast running ones. For instance, if EC=4, fast requests enter and leave execution fast, while slow requests enter execution and are left executing for a much longer time. As a consequence, at certain moments there will be too few or no execution slots available to maintain the high throughput of fast requests. We show this effect in the experimental section.
We wanted to provide a counterbalance mechanism to avoid blocking. For that we formulate the objective that the runtime-to-arrival_rate ratio be similar for all types of request. Requests are divided into request groups RG_i, defined by expected response time intervals (this can be done automatically) and seeing that RG_is have similar runtime-to-arrival rates. Given arrival rate λ_i and runtime β_i, the ratio is β_i/λ_i. When a request arrives and is classified into a group *i*, we compare the runtime-to-arrival rate of that group β_i/λ_i to the one of the remaining groups β_r/λ_r,

```
If(EC<ECmax)
        if(θh × βr/λr < βi/λi)
                reject request
        else
                admit request
```

Where θ_h >1 is a threshold. For instance, if β_i/λ_r=30%, β_i/λ_r= 65% and θ_h =1.5, the request group has ran more than 1.5 times the remaining groups, so the request is rejected. As a practical implementation for this work we defined two groups based on the response times being above or below μ-σ (overall average and standard deviation of response times) and θ_h =1.5. As part of our future work we intend to design automated strategies to determine these parameters for best performance.

5. Experimental Results

Our experimental objective was to analyze the effects of variance and the robustness of the approach in responding to variable conditions (represented here by distribution

variance), and trying to provide the QoS_{max} objective of maximizing the number of requests that End-Within-the-Deadline under variable workloads and congestion.
In order to test this, we have built an event-driven simulator that uses the throughput curve of Figure 4 (obtained from tests with TPCC) to model the effects of multiple simultaneous executions. We have set request deadlines to 5 seconds and varied either duration (request duration when running alone, between 100 ms and 1.5 seconds) or arrival rates (between 10 ms and 500 ms), fixing the other parameters (the duration at 1 second, the inter-arrival time at 100ms). In every case, these were always mean values for each of the distributions depicted in Figure 5 (Uniform, Exponential, Gaussian and Poisson, with varied rates). We included low variance and high variance distributions on purpose, so that we could test the effects of variability.

DISTR.	avg	stdev	DISTR.	avg	stdev
Poisson	1013	92	Gaussian	1000	33
Exponential	1038	976	Uniform	1024	578

Figure 5: Characteristics of distributions used in the experiments

We have engaged in extensive experimentation with the arrival and duration distributions from Figure 5, then focused the analysis to reach conclusions on the variability issue. The results were consistent across distributions, with the variability being the major factor. Two major experiments are shown: The first one tests the response of the system to different variances, by successively increasing that parameter in a Gaussian request duration distribution with mean 1000 msecs. The second experiment tests the system response while varying the mean duration of requests in an exponential request duration distribution. In both cases a uniform arrival distribution was used. Each simulation run submitted 10000 transactions according to the arrival rate distribution and statistics.
Our results concern the following control alternatives:

ACTUAL: this is the control that would result if the controller could guess the exact duration of the request (which it cannot);

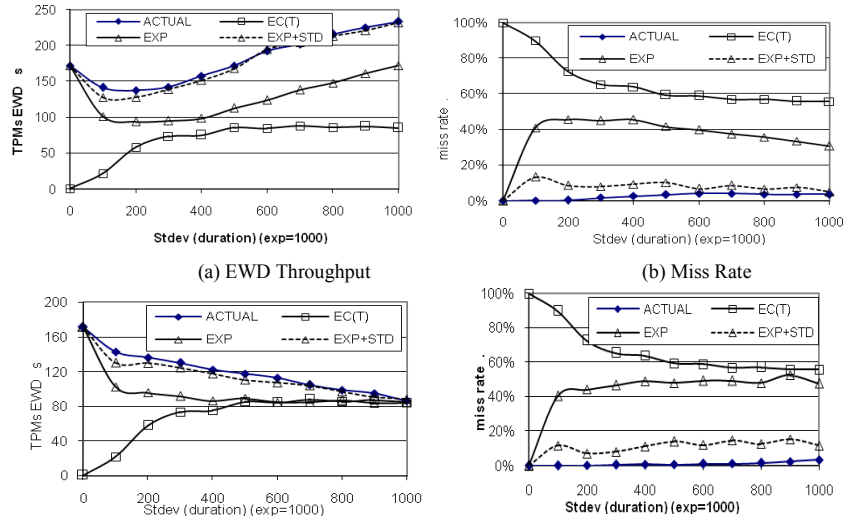
EC(T): the Maximum Throughput strategy, that is, EC_{max} is fixed at the maximum throughput value obtained from the test runs (Figure 4);

EXP: QoS_{max} control using the average of previous request run durations;

EXP+STD: QoS_{max} control using the average+stdev quantity from previous request run durations;

5.1. Variance, with and without Inter-Request Handling

This experiment tests the behavior of the system with different variances and the capability of the approach to handle those different variances. A Gaussian request duration distribution is submitted with expected duration 1000 msecs and successively larger variance (from 100^2 to 800^2 msecs). Figure 6 shows the EWD throughput (a) and the Miss Rate (b) against variability, and for comparison purposes it also shows the results when inter-request blocking avoidance is off (c and d). It is clear from Figure 6a that EXP+STD achieves much better T_{EWD} than EXP as soon as variability appears and also that T_{EWD} of EXP+STD is only slightly less than T_{EWD} if the actual request duration values were known by the controller (ACTUAL). Both EXP+STD and EXP are much better than E(T). This can also be seen by looking at the miss rates of Figure 6b.



(a) EWD Throughput

(b) Miss Rate

(c) EWD Throughput, no inter-request handling

(d) Miss Rate, no inter-request handling

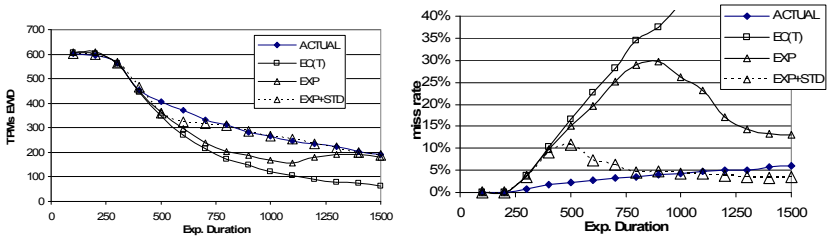
Figure 6: T_{EWD} and Miss Rate vs Variability

The results in Figure 6c and 6d prove that, as we expected, inter-request variability handling to avoid fast request blocking is very important, because without that strategy the T_{EWD} throughput of ACTUAL, EXP and EXP+STD converge to that of EC(T) when variability is high (Figure 6c), even though miss rates are similar to those of Figure 6b.

5.2. Exponential with Varied Durations

In this experiment we tested varying expected durations over an Exponential distribution for durations and uniform arrival distribution. Figure 7 shows T_{EWD} (7a) and the Miss Rates (7b). When the durations are small (e.g. below 500 msecs) EC(T), EXP and EXP+STD have the similar behavior. There is no congestion, expected duration is still relatively small and therefore these strategies fix the same EC_{max} at the maximum throughput value of EC(T). However, as the expected duration increases, the miss rate also starts increasing, as a fraction of requests has much larger durations and miss the deadlines. As a result, the throughput of ACTUAL is larger than the one of EXP or EXP+STD for this part of the Figure spectrum. From 500 msecs on, EXP+STD senses the larger expected durations and decreases EC_{max}, the throughput catches up. A similar adjustment also happens for EXP but it reacts much later. The miss rate of EC(T) increases significantly as we increase the expected duration, so its T_{EWD} is much worse than the other alternatives.

This result shows that QoS_{max} adapts to varied durations, but while EXP exhibits a large miss rate in some cases (up to 30%) and not-so-good throughput, EXP+STD miss rate was always less than 10% and its throughput was almost always as good as the one QoS_{max} would get if it could know actual durations in advance (ACTUAL).



(a) T_{EWD} (b) Miss Rate

Figure 7: T_{EWD} and Miss Rate vs Duration

6. Related Work

There are several related works on Quality-of-service in Web-servers or computer-systems in general. The works in [9, 10, 11] focus on QoS for request processing in real-time databases, especially aimed at applications with strict time requirements such as military ones [1, 10]. They propose strategies to meet deadlines for individual requests, but the authors consider an all-in-memory context, as this renders the system completely predictable and therefore controllable with precision. In [13, 18] the authors propose feedback control to adapt the admission procedure to guarantee deadlines in real time systems, using a control theory approach. In [2, 3, 14] control is based in a specification of QoS targets. There, prediction is based on an analytic model using Markov Chains and Queuing Networks, taking resource demands and capacities as inputs. Quality-of-Service was also studied for Web Servers. In [7] the authors propose session-based Admission Control (SBAC), noting that longer sessions may result in purchases and therefore should not be discriminated in overloaded conditions. They propose self-tunable admission control based on hybrid or predictive strategies. [6] uses a rather complex analytical model to perform admission control. There are also approaches proposing some kind of service differentiation: [4] proposes architecture for Web servers with differentiated services; [5] defines two priority classes for requests, with corresponding queues and admission control over those queues. [16] proposes an approach for Web Servers to adapt automatically to changing workload characteristics and [8] proposes a strategy that improves the service to requests using statistical characterization of those requests and services. Comparing to our own work, all the works referenced above except [8] are either intrusive, require extensive modifications to systems, or use analytical models that may not provide guarantees against real system. The work in [8] does not consider deadlines and does not adapt constantly as ours does, instead it fixes the maximum throughput capacity when it runs the test runs.

7. Conclusions

In this paper we proposed variability-solving solutions for robust QoS control. We have discussed the fact that variability is inherent to systems and no practical acceptable overhead control can account for all the variability factors. We presented the control strategy and then the variability-handling approaches, with the objective of

turning the strategy into a robust one. Our extensive experimental results have shown the advantage of our proposals.

8. References

1. Abbott R. and H. Garcia-Molina, "Scheduling Real-Time Requests: A Performance Evaluation," ACM Trans. Database System, vol. 17, pp. 513-560, 1992.
2. Bennani, "Autonomic Computing Through Analytic Performance Models", Ph.D in CS dissertation, George Mason University, May 2006.
3. Bennani, Menasce, "Assessing the Robustness of Self-Managing Computer Systems under Highly Variable Workloads", Proc. International Conf. Autonomic Computing (ICAC-04), New York, NY, May 17-18, 2004.
4. Bhatti N. and R. Friedrich. Web server support for tiered services. *IEEE Network*, 13(5):64–71, September 1999.
5. Bhoj, Rmanathan, and Singhal. Web2K: Bringing QoS to Web servers. Tech. Rep. HPL-2000-61, HP Labs, May 2000.
6. Chen X., P. Mohapatra, and H. Chen. An admission control scheme for predictable server response time forWeb accesses. In *Proceedings of the 10th World Wide Web Conference*, Hong Kong, May 2001.
7. Cherkasova and Phaal. Session-based admission control: A mechanism for peak load management of commercial Web sites. *IEEE Req. on Computers*, 51(6), Jun 2002.
8. Elnikety S., Nahum E., Tracey J. ; Zwaenepoel W., "A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites", WWW2004: The Thirteenth International World Wide Web Conference, New York City, NY, USA, May 2004.
9. Kang K. D., S. H. Son, J. A. Stankovic, "Managing Deadline Miss Ratio and Sensor Data Freshness in Real-Time Databases", IEEE Trans. on Knowledge and Data Engineering, Vol 16, Nr 10, pages 1200-1216. October 2004.
10. Kang K., "QoS-Aware Real-Time Data Management," PhD thesis, U. Virginia, May 2003.
11. Kang K.D., S.H. Son, J.A. Stankovic, and T.F. Abdelzaher, "A QoSSensitive Approach for Timeliness and Freshness Guarantees in Real-Time Databases," Proc. 14th Euromicro Conf. Real-Time Systems, June 2002.
12. Kanodia V. and E. W. Knightly. Ensuring latency targets in multiclass Web servers. *IEEE Requests on Parallel and Distributed Systems*, 13(10), October 2002.
13. Lu C., J. Stankovic, G. Tao and S. Son, Feedback Control Real-Time Scheduling: Framework, Modeling and Algorithms, special issue of Real-Time Systems Journal on Control-Theoretic Approaches to Real-Time Computing, Vol. 23, No. 1/2 July/September, 2002, pp. 85-126.
14. Menasce, Bennani, "On the Use of Performance Models to Design Self-Managing Computer Systems", Proc. 2003 Comp. Measur. Group Conf., Dallas, TX, Dec. 7-12, 2003.
15. Mogul J. C. and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Requests on Computer Systems*, 15(3):217–252, 1997.
16. Pradhan P., R. Tewari, S. Sahu, A. Chandra, and P. Shenoy. An observation-based approach towards self managing Web servers. In *International Workshop on Quality of Service*, Miami Beach, FL, May 2002.
17. Schroeder, Mor Harchol-Balter, Iyengar, Nahum and Wierman. "How to determine a good multi-programming level for external scheduling." . 22nd International Conference on Data Engineering (ICDE 2006).
18. Stankovic J., C. Lu, S. Son, and G. Tao, The Case for Feedback Control Real-Time Scheduling, EuroMicro Conference on Real-Time Systems, June 1999.
19. TPCC: www.tpc.org.