Optimal Polling for Latency–Throughput Tradeoffs in Queue–Based Network Interfaces for Clusters[1]

Dmitry Ponomarev[2], Kanad Ghose[2], and Eugeny Saksonov[3]

[2] Dept. of Computer Science, State University of New York, Binghamton, NY 13902–6000
{dima, ghose}@cs.binghamton.edu
[3] Moscow Institute of Electronics and Mathematics, Moscow, Russia
saks@miem.edu.ru

Abstract. We consider a networking subsystem for message–passing clusters that uses two unidirectional queues for data transfers between the network interface card (NIC) and the lower protocol layers, with polling as the primary mechanism for reading data off these queues. We suggest that for accurate mathematical analysis of such an organization, the values of the system's states probabilities have to be taken into consideration, in addition to the well–known mean–value estimates. A single server single queue polling system with server's "vacations" is then articulated via an M/G/1 queueing model. We present a method for obtaining the values of system's states probabilities in such a scheme that can be used to identify "sweet spots" of the polling period that result in a reasonable compromise between the message passing latency and the overall system's throughput.

1 Introduction

We consider the problem of tuning the performance of message passing in clusters where low latency as well as high system throughput are desirable performance goals. These requirements dictate the use of non–traditional network subsystems for the clusters [4]. Throughput, in this context, refers to the throughput of the system as a whole, including the networking subsystem (which comprises of the protocol layers, OS services, drivers and the network interface card). Traditional network interfaces are interrupt–driven. When the network interface card (NIC) lacks bus mastering capabilities, the arrival of a packet causes the NIC to generate an interrupt. The handler for this interrupt initiates the DMA operation that moves the data from the NIC to the RAM. When the NIC is a bus master, the arrival of a packet does not generate an interrupt as the NIC can initiate the DMA transfer on its own. Interrupts are also generated after the completion of DMA transfers between the host memory and the NIC, specifically when a packet is DMA–ed from RAM–resident outgoing buffers to the NIC in the course of sending a packet and after the completion of a DMA transfer from the NIC to the RAM–resident incoming buffers in the course of receiving a message. These interrupts are needed to trigger protocol layer actions, such as message demultiplexing or actions that initiate further transmissions.

The impact of such interrupt driven interfaces on the overall system performance – that of the network subsystem, as well as that of the host – are two–fold. First, The NIC cannot sustain large data movements on its own, and arrival of data at a high rate can cause overflows from the buffers within the NIC. This is particularly true as the raw hardware data rate of the network increases and approaches that of the bus on which the

[1] supported in part by the NSF thru award Nos. EIA 991109, CDA 9700828 & an equipment donation from CISCO Systems

NIC resides (typically, the I/O bus). Such overflows from the NIC buffers result in retransmission of the dropped packets (when reliable delivery is needed), resulting in an overall performance degradation. Message transmission latency, as well as the overall message transfer rates are both affected adversely. Second, frequent interrupts generated by the NIC cause a commensurate increase in the rate of context switching (which can be relatively expensive in terms of cycles), impacting the throughput of the interrupted application adversely.

As NICs continue to evolve and begin to incorporate additional logic for DMA streaming and message demultiplexing (i.e., identifying the end–point of a packet), it is imperative to move away from an interrupt–driven interface to the NIC and seek solutions that exploit the full performance potentials of these NICs [4].

Polled unidirectional queues for incoming and outgoing messages maintained in the RAM provide the basis for implementing efficient interfaces in–between the NIC and the software of the networking subsystem. Typically, in these systems, both the NIC and the networking subsystem software operate autonomously until queue limits, as known to the software or the NIC, are reached. At that point, usually, interrupts are generated to update queue limits and take any other actions that are needed. The incoming and outgoing queues can be global (i.e., common to all connections/channels) or independent queues could be used for each open channel/connection. The latter is preferred when differentiated services (to implement connections with different quality of service, for example) are implemented, with additional logic in the NIC to perform message demultiplexing. For example, Myrinet [5], and the semi–custom field–programmable gate array based NIC described in [2]) use unidirectional queues in this manner to implement low latency communication facilities. In these systems, two unidirectional queues are used, one is written to by the lower protocol layers and read by the NIC while the other is written to by the NIC and read by the software.

Instead of interrupts, both the software and the NIC use polling to read their corresponding queues. The Virtual Interface Architecture specification [Via] [Den+ 98] represents a general form of these interfaces. The motivation behind such a design is a reduction of the frequency of interrupts. A polling agent, such as a daemon or process/thread activated by applications can poll the outgoing and incoming queues. Several incoming messages may accumulate in the queue before it is polled by the application, resulting in a single context–switch to process all these messages as opposed to a per–message context switch, as in traditional designs. Similarly, the NIC can poll the outgoing queue(s) to determine if any message has to be sent out.

When polling is performed by software, the period at which each queue is polled is critical – if polling is performed too often, the overhead for the polling process itself can be detrimental to performance, and in addition, the queue may not have useful data to process. On the other hand if polling is performed infrequently, packets may end up spending a longer time on the queues, increasing the overall latency. The goal of this paper is to analytically model queue–based networking subsystems as described above and identify optimal polling intervals. We concentrate on systems that use global queues; the analysis can be generalized to systems that use multiple sets of queues.

The rest of the paper is organized as follows. Section 2 gives an overview of a queue–based polling–driven networking subsystem, outlining its advantages and hardware modifications that have to be incorporated into a fairly simple NIC to support the design. In section 3, we describe how a single M/G/1 queue with server's "vacations"

maps to the situation encountered in a queue–based networking followed by a formal description of the queueing system and the presentation of a method for computing the system's states probabilities. It is also shown in this section how the errors in computations can be estimated over a compact domain in a recursive manner. Section 4 discusses new performance metrics and possible optimizations followed by our conclusions in section 5.

2 Queue–Based Polling–Driven Networking Subsystem

In a queue–based polling–driven networking subsystem, two unidirectional queues are used for interfacing between the device driver software and the NIC. These queues are located in the main memory and serve as the intermediate buffers between the software and the hardware. One of the queues is typically used for passing the commands from the software to the NIC. For example, when an application calls a *send ( )* routine, the message is authenticated, processed by the protocol layers and then passed to the NIC driver, which initiates the DMA indirectly by putting a *do–send* command into the queue, perhaps passing the message descriptor as a parameter. Other commands may also be placed on this queue, for example *do_fast_hardware_retransmit ()* etc. depending on the implementation. The NIC polls this queue periodically and executes commands placed by the software as it finds appropriate.

The second queue is used for the receive path. When the NIC receives a message from the network (we assume that the NIC is a bus master), it DMAs the message into the RAM and acknowledges the completion of this event by placing an appropriate information into the queue. Also, the NIC may signal the occurrence of other events through this queue, including transmission error and acknowledgement that the message has been successfully sent out to allow the driver to free up some buffer space for the upcoming messages. A separate kernel process, or even a thread, polls this queue with some frequency and calls appropriate driver or protocol stack functions to further manipulate the incoming message. No interrupts are normally generated during the course of message reception. The only time when the NIC generates an interrupt is when the queue becomes full. Even when the transmission error is encountered, it is sufficient to associate this event with the higher priority and sort all accumulated requests in the queue according to their priorities. Should such operation seem to be an expensive endeavor, a traditional interrupt can simply be generated on this rare occasion.

Such an organization de–couples the NIC from the driver software, allowing each to execute at their own speeds. The frequency of interrupts comes down, since the interrupts are only generated when the queue becomes full. Moreover, if polling is designed carefully, the number of expensive context switches should also decrease because at a polling instant, the software may very well find several requests already posted by the NIC, thus amortizing the cost of the context switch. However, if polling frequency is chosen badly, polling–driven networking may even increase the number of context switches – this is, of course, the result of polling the empty queue.

Design of such a subsystem requires quite modest modifications to the NIC hardware: a pair of registers has to be added to the NIC for addressing the queues, interrupts have to be turned off, and logic has to be added for generating an interrupt when the queue is full and performing polling on one of the queues. A few changes have to be also made to the driver code, those mainly include adding the functions for accessing the queues.

However, applications and higher–level networking software remain oblivious to these modifications.

Clearly, the performance of a system employing such queues is highly dependent on a carefully chosen frequency of polling a queue. Traditional analysis based on mean–value estimates, such as the average waiting time in the queue, may suggest the necessity of a high polling frequency, because this results in low message–passing latency. Indeed, a message descriptor placed in the queue by the NIC, corresponding perhaps to a completion of a DMA operation upon reception of a message, could receive the almost immediate attention of the polling server in this scenario. However, every attempt to poll the queue results in a performance hit for other processes currently in the system. In the least, when the polling server is scheduled, the context switch has to be performed. Unless the server can execute in the same virtual address space as the interrupted user process (as the case is in UNIX–based systems), this context switch involves saving the page mapping information of the user process and flushing the TLBs, that are quite expensive operations by all standards. In addition, if a polling process is scheduled frequently, all processes in the ready queue will experience longer delays before receiving their time slices. In this context, the polling frequency should be kept to a reasonable minimum. It is especially important to minimize the probability of a situation when the server polls the empty queue. In some ways, this is analogous to a process doing a busy–wait on a software lock issued by another process in a critical section. If the duration of a busy–wait is a short one, the exploring process spins on the lock even more rapidly making it an expensive waiting. If, on the other hand, the busy–wait is one of a longer duration, the process might find its critical section available at its next try. Polling the queue containing only one request may not be very efficient either, in the best case such a situation will not be different from a traditional interrupt–based networking subsystem where an interrupt is generated by the NIC upon the reception of every message.

Thus, to carefully analyze the effects of polling on the performance of the entire system, one should attend to such nuances as the values of system's states probabilities (system's state is defined as the number of requests in the queue), that were largely ignored in favor of more easily obtainable mean value estimates. In what follows, we show how these values can be computed for a single queue system.

3 Analytical Model

Some performance aspects of a queue–based networking subsystem can be analytically modelled using an M/G/1 queue. Hereafter, we shall refer to the process performing polling operation on the queue as the server. If the server finds any requests in the queue at the polling instant, it serves all those requests in accordance with gated service discipline, that is, only those requests that were on the queue at the moment of server's arrival there are served. Requests entering the queue while it is being served must wait until the next cycle and will receive service in the next service period. The state of such system at any moment of time is defined as the number of requests in the queue.

A single server single queue M/G/1 model with server's "vacations" is a degeneration of a multiqueue polling system where several queues are served in a cyclic order. Requests arrive at the queue according to Poisson process, that is, the interarrival time is governed by the probability distribution $1-e^{-\lambda t}$ with average interarrival time $1/\lambda$. Probability distribution of the time required to serve a request is denoted as $B(t)$ with

finite first and second moments ($0 < \beta_1 < \infty$ and $0 < \beta_2 < \infty$). After servicing all requests in the queue, the server goes for a vacation. Vacation time is generally distributed in accordance with some probability law and its probability distribution function is denoted as *G(t)* with finite first and second moments ($0 < \gamma_1 < \infty$ and $0 < \gamma_2 < \infty$). After vacation period completes, the server switches to the queue again and continues to serve the requests sitting there. We consider the system at the time epoches of the initiation of service at the queue, that is, the moments when vacation periods complete. Assuming that the system operates in the equilibrium mode (the existence of such mode is conditioned by the inequality $\lambda\beta_1 < 1$), we attempt to obtain the values of system's states probabilities at the moments of service initiation. We also derive the upper bounds for errors in the computations.

At the moments of service initiation, the system can be considered as an embedded Markov chain with a finite number of states. The stationary probabilities are connected by the following system of linear equations [6]:

$$p_i = \sum_{k=0}^{\infty} p_k \int_0^{\infty} \frac{(\lambda t)^i}{t!} e^{-\lambda t} d(B_k(t) \oplus G(t)),$$

where p_i is the stationary probability of state i, $B_k(t)$ is the service time distribution of k customers and $\oplus$ is the convolution symbol for the two distributions.

Using the generating function $\pi(x) = \sum_{i=0}^{\infty} x^i p_i$, one obtains:

$$\pi(x) = \gamma(\lambda(1-x))\pi(\beta(\lambda(1-x))), \qquad (1)$$

where $\gamma(.)$ and $\beta(.)$ are the Laplace–Stieltjese transforms of functions *G(t)* and *B(t)* respectively. This result is well–known and it had been shown in [6] how the approximate values of system states probabilities can be computed from (1). In what follows, we outline how one could compute these probabilities, in particular, p_0 and p_1 respectively, with a bounded estimation of the error.

Consider the sequence $\{x_k\}$ *k=0,1,2,...*; $0 \le x \le 1$, the elements of which are connected by the following recursive expression:

$$x_{k+1} = \beta(\lambda(1-x_k)). \qquad (2)$$

Sequence $\{x_k\}$ converges to 1 for any $0 \le x_0 \le 1$ if $k \to \infty$ due to the properties of function $\beta(s)$. Indeed, $\beta(s) \le 1$ when $s \ge 0$ and by differentiating the equality $\beta(\lambda(1-x)) = \int_0^{\infty} e^{-\lambda(1-x)t} dB(t)$ it can be shown that $\beta'(\lambda(1-x)) \le \beta_1\lambda < 1$.

3.1 Computing Probability of State 0

For computing p_0 we introduce the following sequence:

$$\pi(x_k) = \gamma(\lambda(1-x_k))\pi(x_{k+1}). \qquad (3)$$

The limit of the sequence $\{\pi(x_k)\}$ when $k \to \infty$ exists because of two reasons:
a) there exists the limit of the sequence $\{x_k\}$ and,
b) function $\pi(x)$ is continuous on the closed interval [0,1].
Since $\pi(1) = 1$, the limiting value of this sequence is 1.
From (3) one obtains:

$$\pi(x_0) = \prod_{i=0}^{k} \gamma(\lambda(1-x_i))\pi(x_{k+1}). \tag{4}$$

In the limit $k \to \infty$ and under the condition $\lambda\beta_1 < 1$, noting that $\pi(1) = 1$, this results in:

$$\pi(x_0) = \prod_{i=0}^{\infty} \gamma(\lambda(1-x_i)). \tag{5}$$

Using equality (5), the value of $\pi(x_0)$ can be approximately computed at any point $0 \le x_0 \le 1$. To do so, we introduce another sequence, the elements of which $\{\pi_{0k}(x_0)\}$ are defined as products of the first k terms from the equation (5).

$$\pi_{0k}(x_0) = \prod_{i=0}^{k} \gamma(\lambda(1-x_i)). \tag{6}$$

If $x_0 = 0$ then $\pi(x_0) = p_0$ (exact value of the probability that the queue is empty at the moment of service initiation) and $\pi_{0k}(x_0)$ gives an approximation of p_0. Clearly, when $k \to \infty$ the sequence $\{\pi_{0k}(x_0)\}$ converges to $\pi(x_0)$ and the accuracy of formula (6) improves.

3.2 Error Estimation

The major advantage of the technique shown in the previous section is the possibility to estimate the accuracy of computing $\pi(x_0)$ as a function of k using formula (6) by deriving the expression for the upper bound in computational error.

Indeed, from (2) we obtain that $x_{k+1} \le x_k + (x_k - x_{k-1})M$ and, hence, $(x_{k+1} - x_k) \le (x_k - x_{k-1})M \le (x_1 - x_0)M^k$, where $M = \max\{\beta'(\lambda(1-x))\} = \lambda\beta_1 = \varrho, 0 \le x \le 1$.

Analogously, for $\pi(x_k)$ and $\pi(x_{k+1})$: $\pi(x_{k+1}) \le \pi(x_k) + (x_{k+1} - x_k)M_1$, or $\pi(x_{k+1}) \le \pi(x_k) + (x_1 - x_0)M^k M_1$ where $M_1 = \max\{\pi'(x)\} = \pi'(1) = \dfrac{\gamma_1\lambda}{1-\varrho}$, $0 \le x \le 1$.

Using the last expression, it can be shown that for any N and k such that $N>k$

$$\pi(x_N) - \pi(x_k) \le \sum_{i=k}^{N} (x_1 - x_0)M^i M_1 .$$

In the limit $N \to \infty$ we have $\pi(x_\infty) = 1$. Thus,

$$1-\pi(x_k) \le \sum_{i=k}^{\infty}(x_1-x_0)M^iM_1 = (x_1-x_0)M_1\frac{M^k}{1-M} = \frac{(x_1-x_0)p^k\lambda\gamma_1}{(1-\varrho)^2}.$$

Consider now the difference $\pi_{0k}(x_0)-\pi(x_0)$. From the equalities obtained earlier, it follows that:

$$0 \le (\pi_{0k}(x_0)-\pi(x_0)) \le \prod_{i=0}^{k}\gamma(\lambda(1-x_i))\frac{(x_1-x_0)\lambda\varrho^k\gamma_1}{(1-\varrho)^2} \quad (7)$$

If $\varrho < 1$ and $k \to \infty$, the difference in (7) tends to the limiting value 0. Expression (7) provides an upper bound for the error in computations of p_0 using formula (6).

3.3 Computing Probability of State 1

Differentiating equality (1) with respect to x and using the sequence $\{x_k\}$, it is possible to construct another sequence $\{\pi'(x_k)\}$ in the following way:

$$\pi'(x_k) = -\lambda\gamma'(\lambda(1-x_k))\pi(x_{k+1})-\lambda\beta'(\lambda(1-x_k))\ \gamma(\lambda(1-x_k))\pi'(x_{k+1}), \quad (8)$$

where $\gamma'(\lambda(1-x_k)) = \frac{d}{ds}\gamma(s)|_{s=\lambda(1-x_k)}$, k=0,1,2....

From equations (3) and (8) we construe for any integer k :

$$\pi'(x_0) = \pi(x_0)\sum_{i=0}^{\infty}(-1)^{i+1}\frac{\lambda\gamma'(\lambda(1-x_i))}{\gamma(\lambda(1-x_i))}\prod_{j=0}^{i-1}\lambda\beta'(\lambda(1-x_j)) \quad (9)$$

The convergence of the series is guaranteed here because both $\gamma'(x)$ and $\gamma(x)$ are continuous functions in the closed interval [0,1]. As before, the approximate value of p_1 can be computed using formula (9) by taking into account the first k terms of the summation. Accuracy of such an approximation can be estimated by a method similar to that presented in section 3.2.

4. Performance analysis and optimizations

In this section, we show how the results obtained in the previous section can be applied to performance analysis. We assume that the offered network load is 150Mbytes/sec and we consider two average packet sizes: one is 1500 bytes and the other is 600 bytes. This translates into the arrival rates of 0.1 packets/microsecond and 0.25 packets/microsecond. Accepting a microsecond as a unit of time, we assume that the packets arrive at the host according to the Poisson distribution with $\lambda = 0.1$ and $\lambda = 0.25$ respectively. We further assume that the average service time of a packet by the polling server is 2 microseconds ($\alpha = 0.5$) and the service discipline is gated and context switch latency in such a system is 1 microsecond. Figure 1 shows the values of average waiting time of a packet in the queue (as defined in [7]), probability p_0 (computed using formula (6)), and probability p_1 (formula (9)) as functions of the

average vacation period. We assumed that the context switch is a part of a vacation, therefore the minimum value of a vacation period is 1 microsecond.

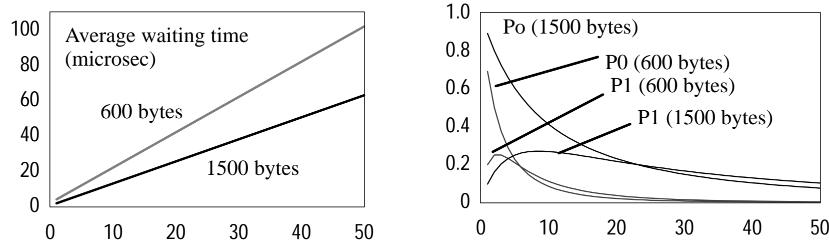

Fig. 1. The graph on the left shows the average waiting time of a packet as a function of the average vacation interval (in microsecs.) for different packet sizes. The graph on the right shows the value of P0 and P1 as functions of the average vacation interval

As can be seen from the Figure 1, short "vacation" periods indeed reduce the average waiting time in the queue, but this comes at the expense of having a higher probability of polling the empty queue (p_0), thus wasting a considerable number of cycles for such useless switchings. The value of probability p_1 also carries some importance because it measures the frequency of finding exactly one request in the queue at the polling instant. Though not resulting in a total waste of CPU time, such service does not offer any improvement compared to a traditional interrupt–based mechanism, since one context switch is required per packet. In a properly designed networking subsystem, each of these parameters has to be given a balanced attention. One simple way to achieve this is to define a cost function (F) depending on the values of p_0 and the average waiting time of a request in the system (w) along with a pair of appropriate coefficients that can be tuned up based on the needs of specific applications and timings obtained by instrumenting the OS kernel. A simple cost function, for instance, can be represented as follows:

$$F(\gamma_1) = C_1 p_0 + C_2 w \qquad (10)$$

Here, the coefficient C_1 is the cost of switching to the empty queue and the coefficient C_2 is the cost of a unit of time that an incoming message spends in the queue. If the latency of message passing is more important design goal, then higher waiting cost should be assumed and thus the value of C_2 must be set higher than the value of C_1. However, the opposite is true if the more important characteristic is the overall system's throughput – that is, the amount of work done in a unit of time, and higher latency can be tolerated. Of course, these values can be readjusted dynamically according to the changing workload. As shown in Figure 1, both p_0 and w depend on the average vacation interval which is the single argument of the function F as expressed by formula (10). Any good design should aim at minimizing the value of F by selecting the optimal vacation period.

Figure 2 shows the function F for the two networking subsystem configurations considered in this section. The first graph is for the average packet size of 1500 bytes, the second is for that of 600 bytes. The value of p_0 was multiplied by 100 to keep it in the interval [0..100] and this new value was then used in the formula for F. We computed the value of F for three different sets of cost coefficients. First, C_1 was assumed to be

equal to C_2 (for simplicity, we considered the case of both coefficients being set to 1). Second, C_1 was assumed to be greater than C_2 (we set the first coefficient to 1 and the second coefficient to 2). Third, C_2 was assumed to be greater than C_1, so the value of 2 was assigned to C_2 while keeping C_1 at 1. The results show that function F is sensitive to the values of these coefficients, and the system can be effectively controlled by careful selection of the values of C_1 and C_2. In all three cases, the function F has a minimum, and the average vacation period that provides this minimum increases along with the increasing cost of switchings to the empty queue. Another observation from the results shown in the Figure 2 is that the minimums formed by the function F are not the sharp ones. Indeed, there is a sizable range of argument values, for which the values of F deviate only slightly from the minimum value. Thus, a "sweet spot" of vacation periods can be identified, where the value of F is either a minimum or its deviation from the minimum does not exceed a predefined threshold.

Other variations of cost functions obviously exist, where the value of p_1 can be included in some form. To be practical, however, any expression for the cost function has to be fairly simple, so that the optimal vacation period could be computed quickly. This is especially important when the behavior of the applications changes frequently and the vacation period has to be adjusted accordingly to respond to the changing environment.

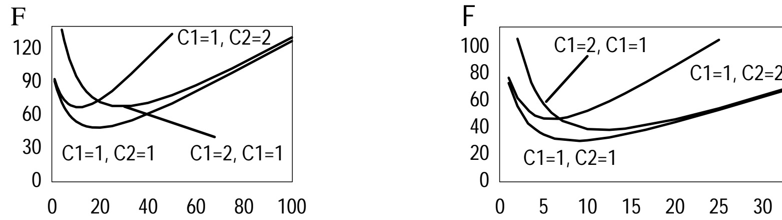

Fig. 2. Cost function F for various average vacation intervals. Graph on the left shows the results for the average packet length of 1500 bytes, and graph on the right – for 600 bytes.

Alternative ways to employ the value of p_0 in system's performance analysis also exist. With the capability of computing the system's states probabilities with high degree of accuracy, the traditional mean–value estimates no longer represent an exclusive gauge of system's performance. In fact, the new metrics, in the very least taking into account the value of p_0 , should be considered. One such simple indicator (S) is the average number of switches (vacations) for the service of one request.

Specifically, S=Q/N, where Q is the average number of consecutive switches between two service periods, and N is the average queue length at the polling instant. Q can be easily computed from p_0 in the following way:

$Q = 1 + (1-p_0)\sum_{k=0}^{\infty} k(p_0)^k = \frac{1}{(1-p_0)}$. The minimum value of Q is 1, because at least one switch between two consecutive service periods always occurs. The value of (Q–1) is the indicator of the fraction of switches that find the target queue empty. Parameter S can be controlled by varying the average vacation interval. As the vacation period increases, Q –> 1, N –> ∞ and thus, S–> ∞ which is advantageous from the perspective

of overall system's throughput. Apparently, by varying the average vacation period, one can tune the system in such a way that the value of S is minimized under the constraint that the average queue length does not fall below certain threshold, which can be defined as the level of system's tolerance to the increase in message passing latency.

5. Conclusions

This paper argues that intelligent selection of polling frequency is a key to performance of a system, where the number of interrupts within the networking subsystem is reduced through the use of two polled, unidirectional queues to provide an interface between the NIC and the networking software on the host. Such polling should not only aim at servicing the requests in the queue fast, thus effectively reducing the latency of a message passing, but also attend to the needs of other processes running in the system. In this realm, polling of an empty queue should be avoided and polling of the queue that contains only one request should be minimized, thus amortizing the cost of expensive context–switch operation. The numerical method for computing the probabilities of system's states in the single queue polling system is then presented. In contrast to previously suggested techniques, our mechanism allows to estimate the errors in computations of the state probabilities and accurately derive the probability of polling the empty queue or polling the queue that contains only one request. These values, in conjunction with the mean–value estimates like the average number of customers in the queue and the average queue length, are instrumental in the design of efficient polling in a queue–based networking subsystem. This is illustrated by constructing the cost function, which depends on the probability of switching to the empty queue and the average waiting time in the queue. This cost function can be minimized by selecting the appropriate value of average vacation period.

References :

1. Danning, D. et.al.: The Virtual Interface Architecture. IEEE Micro vol. 18 N2 (1998)
2. Ghose, K., Melnick, S., Gaska, T., Goldberg, S., Jayendran, A. and Stien, B.: The Implementation of Low Latency Communication Primitives in the SNOW Prototype. Proceedings of the 26–th Int'l. Conference on Parallel Processing (ICPP) (1997) 462–469
3. Leibowitz, M.: An Approximate Method for Treating a Class of Multiqueue Problems. IBM J. Research Dev. Vol.5 N3 (1961) 204–209
4. Mukherjee, S. and Hill, M.: The Impact of Data Transfer and Buffering Alternatives on Network Interface Design. Proceedings of the Fourth Int'l Symposium on High–Performance Computer Architecture (HPCA) (1998)
5. Myrinet on–line documentation/specs at:http://www.myri.com
6. Saati, T.: Elements of Queueing Theory with Applications. McGraw–Hill (1961)
7. Takagi, H.: Queueing Analysis of Polling Models. ACM Computer Surveys, Vol.20, N1 (1988)
8. Virtual Interface Architecture on–line documentation/specs at: http://www.viarch.org