Available online at www.sciencedirect.com
ScienceDirect
Computer Communications 30 (2007) 3425–3441
computer communications
www.elsevier.com/locate/comcom

Performance analysis and comparison of interrupt-handling schemes in gigabit networks

K. Salah *, K. El-Badawi, F. Haidari

Department of Information and Computer Science, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia

Received 29 December 2006; received in revised form 20 June 2007; accepted 21 June 2007
Available online 27 June 2007

Abstract

Interrupt processing can be a major bottleneck in the end-to-end performance of Gigabit networks. The performance of Gigabit network end hosts or servers can be severely degraded due to interrupt overhead caused by heavy incoming traffic. In particular, excessive latency and significant degradation in system throughput can be encountered. Also, user applications may livelock as the CPU power gets mostly consumed by interrupt handling and protocol processing. A number of interrupt-handling schemes has been proposed and employed to mitigate the interrupt overhead and improve OS performance. Among the most popular interrupt-handling schemes are normal interruption, polling, interrupt coalescing, and disabling and enabling of interrupts. In previous work, we presented a preliminary analytical study and models of normal interruption and interrupt coalescing. In this article, we extend our analysis and modeling to include polling and the scheme of interrupt disabling and enabling. For polling, we study both pure (or FreeBSD-style) polling and Linux NAPI polling. The performances for all these schemes are compared using both mathematical analysis and discrete–event simulation. The performance is studied in terms of three key performance indicators: throughput, system latency, and the residual CPU bandwidth available for user applications. As opposed to our previous work, we consider not only Poisson traffic, but also bursty traffic with empirical packet size distribution. Our analysis and simulation work gives insight into predicting the system performance and behavior when employing a certain interrupt-handling scheme. It is concluded that no single interrupt-handling scheme outperforms all other schemes under all traffic conditions. Based on obtained results, we propose and discuss a novel hybrid scheme of interrupt disabling–enabling and pure polling in order to attain peak performance under low and heavy traffic loads.
© 2007 Elsevier B.V. All rights reserved.

Keywords: High-speed networks; Operating systems; Interrupts; Interrupt coalescing; Polling; Modeling and analysis; Simulation; Performance evaluation

1. Introduction

1.1. Background

In hosts with Gigabit Ethernet links, the arrival rate of incoming packets can surpass the kernel's packet processing rate of network protocol stack processing and interrupt cost. In fact even with today's powerful multi gigahertz processors, the cost of per-packet interrupt *alone* surpasses the inter-arrival time of packets. With Gigabit Ethernet and the highest possible rate of 1.23 million interrupts per second for a minimum sized packet of 64 bytes, the CPU must process a packet in less than 0.82 μs in order to keep up with such a rate. According to reported measurements in [1], an incoming-packet interrupt cost, on a 450 MHz Pentium-III machine running Linux 2.2.10, was 13.23 μs. With the presence of more powerful multi gigahertz processors these days, it is expected the interrupt cost will not be decreased linearly by the speed frequency of the processor, as I/O and memory speed limits dominate [2]. In [2] it was concluded that the performance of 2.4 GHz processor only scales to approximately 60% of the performance of an 800 MHz processor.

* Corresponding author. Tel.: +966 38604493.
E-mail addresses: salah@kfupm.edu.sa (K. Salah), elbadawi@kfupm.edu.sa (K. El-Badawi), fahd@kfupm.edu.sa (F. Haidari).

0140-3664/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.comcom.2007.06.013

Under heavy traffic load such as that of Gigabit networks, the performance of interrupt-driven systems can be degraded significantly, and thus resulting in a poor host performance perceived by the user. For one thing, every hardware interrupt, for every incoming packet, is associated with context switching of saving and restoring processor's state as well as potential cache/TLB pollution. More importantly, interrupt-level handling, by definition, has absolute priority over all other tasks. If the interrupt rate is high enough, the system will spend all of its time responding to interrupts, and nothing else will be performed; and hence, the system throughput will drop to zero. This situation is called *receive livelock* [3]. In this situation, the system is not deadlocked, but it makes no progress on any of its tasks, causing any task scheduled at a lower priority to starve or not have a chance to run.

A number of interrupt-handling schemes and solutions have been proposed in the literature [4–22] to mitigate interrupt overhead and improve OS performance. Some of these solutions include interrupt coalescing, polling, disabling interrupts, OS-bypass protocol, zero-copying, jumbo frames, pushing some or all protocol processing to hardware, etc. A comprehensive summary that comprises many of these solutions can also be found in [4]. The most popular interrupt-handling schemes include primarily normal interruption, interrupt coalescing, polling, and interrupt disabling and enabling. In this paper we present analytical models for a number of schemes that includes ideal system, normal interruption, pure polling, and interrupt disabling and enabling. The analytical models presented in this paper are based on queueing theory and Markov process. Discrete–event simulation (DES) is utilized to verify analysis. Also we utilized DES to study the performance of *all* schemes when hosts are subjected to bursty traffic.

In previous work [23–25], we presented a preliminary analytical study of normal interruption. The performance was studied primarily in terms of throughput [23,24]. The performance in terms of latency for only normal interruption was briefly discussed in [23]. A detailed simulation models for hosts with PIO and DMA were given in [25]. In [26], we presented a complete analytical model to study the performance of interrupt coalescing scheme.

1.2. New contributions

In sharp contrast to our previous work presented in [23–26], this paper is different in significant ways. First, the paper presents novel analytical study and models for polling and for the scheme of interrupt disabling and enabling. For polling, we study both pure (or FreeBSD-style) polling and Linux NAPI polling. Second, the paper summarizes and extends the analytical work of normal interruption to model the system when there is a per-packet processing overhead. Third, a comprehensive performance comparison of all known interrupt-handling schemes is presented. The host performance is studied and compared in terms of three key performance indicators which include system throughput, system latency, and CPU availability for other processing including user applications. Fourth, we utilize DES simulation to model and examine the performance when host is subjected to not only to Poisson traffic but also to bursty traffic and with variable packet sizes. Fifth and as opposed to our previous work, we consider more realistic values for system parameters that suit modern Gigabit network environment and hosts. In previous work we used system parameters of 400 MHz Pentium III machines. In this article we consider system parameters of today's CPU cores such as the 2.53 GHz Pentium-IV machines. Sixth, the paper investigates the influence on performance due to the selection of different parameter values for a certain interrupt-handling scheme. One may argue that the selection of parameter values may favor one scheme over the other. Seventh, the paper proposes and discusses a hybrid scheme that combines interrupt disabling–enabling and pure polling in order to attain peak performance under low and heavy traffic loads. Lastly, the paper discusses in detail a typical DMA-based architecture model of transferring packets between the Network Interface Card (NIC) and host memory.

The rest of the paper is organized as follows. Section 2 illustrates a common and typical architecture model of transferring packets between the NIC and host memory. The architecture model is based on employing DMA. Section 3 explains briefly the different interrupt-handling schemes and presents analytical models that capture the system behavior and study their performance. Section 4 gives numerical examples showing both analysis and simulation results under Poisson traffic. Section 5 examines performance impact when hosts are subjected to bursty traffic with empirical variable-size packets. The impact is studied using simulation. Also Section 5 discusses the impact of selecting different parameter values for each scheme on the host's performance. A hybrid interrupt-handling scheme is also proposed in Section 5. Finally, Section 6 concludes the study and identifies future work.

2. DMA-based design

For our hosts, we assume that the NIC is equipped with DMA engines. Today's high-speed NICs are equipped with DMA engines in order to save CPU cycles consumed in copying packets. NICs are typically equipped with a receive Rx DMA engine and a transmit Tx DMA engine. A Rx DMA engine handles transparently the movement of packets from the NIC internal buffer to the host's kernel memory. A Tx DMA engine handles transparently the movement of packets from the host memory to the NIC internal buffer. It is worth noting that the transfer rate of incoming traffic into the kernel memory across the PCI bus is not limited by the throughput of the DMA channel. These days a typical DMA engine can sustain over 1 Gbps of throughput for PCI 32/33 MHz bus and over 4 Gbps for PCI 64/66 MHz bus [27–30].

Albeit there are numerous variants of how packets get transferred from the NIC and then to protocol buffer [1,9,16,19,27–35], we discuss here one of the common architecture models that best suits the Gigabit networking environment and currently implemented in FreeBSD and Linux latest releases of 2.4 and beyond [9,19,32,33,35,36]. Fig. 1 shows such an architecture model of DMA-based design. The figure shows the flow path of an incoming packet involving the NIC, host memory, and application. The packets are DMA'd from the NIC Rx buffer, through the bus interface such as the PCI, to a system shared ring Rx buffer or Rx DMA ring, and subsequently consumed by user application or routed elsewhere.

Note that the DMA ring is shared between the NIC and kernel's protocol processing, with the NIC being the producer and protocol processing being the consumer. A producer–consumer implementation must be carried out and the implementation varies depending on the machine architecture and NIC features. Typically, the DMA engines implement scatter–gather DMA logic and operate in a bus-master fashion. At initialization, the kernel allocates the Rx circular buffer. Rx circular buffer is typically a FIFO of memory block pointers. The blocks are scattered in memory and are pre-allocated to store incoming packets. The block pointers are linked in a circular fashion and can be read by the NIC DMA engine. Usually, the NIC is configured to have a write-pointer register. If the NIC's write-pointer register has a null value, no DMAing will take place, and incoming packets will get dropped. To start DMAing of incoming packets, the kernel writes, at initialization, the address of the first block pointer into the NIC's write-pointer register. After a successful DMAing of an incoming packet, an interrupt is generated by the NIC. The NIC DMA engine will read and update automatically its write pointer to next address in the chain. The NIC will keep DMAing incoming packets as long as the buffer is not full, i.e., the next address in the chain does not have a null value. The last block in the chain will have a null pointer. It is to be noted that the kernel has a read pointer to the circular buffer. Every time a packet is processed by the kernel, the kernel refills the ring with a new block and updates its read pointer. Note that kernel's protocol processing for a single packet will finish when the packet is placed in an upper layer queue such as that of the application or is placed in an outgoing output queue of an interface in case of routing.

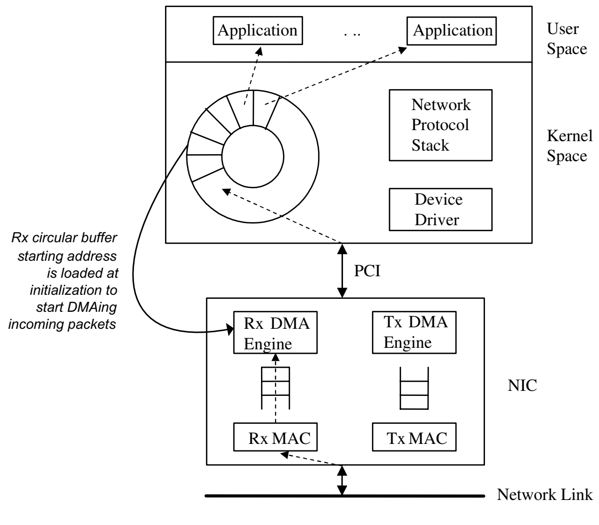

Fig. 1. Architecture model of DMA-based design.

Under normal-interruption mode, it is important to emphasize again that the NIC is typically configured such that an interrupt is generated after the incoming packet has been completely DMA'd into the host's kernel memory. In order to minimize the time for ISR execution, ISR handling mainly sets a software interrupt to trigger the protocol processing for the incoming packet. Practically, it is prudent to limit the function of ISR handling to only notifying the kernel to start protocol processing of the received packet. In [1,16,33,36] and Linux releases prior to 2.4.20 [34,35], ISR handling included appending or chaining DMA'd incoming packets from DMA ring to protocol incoming buffer. Additionally ISR handling included refilling or replenishing DMA ring with new buffer space to make up for the space consumed by the appended packets. If any packet handling (such that of appending or replenishment) is done during ISR, the interrupt overhead can considerably stretch as multiple packets can be received during ISR handling, and subsequently causing considerable cache pollution as well as starving protocol processing. As opposed to [1,16,33–36], Linux 2.4.20 and thereafter [19,32,35] implements a NAPI (New API) architecture in which the DMA ring is the same as the protocol buffer, and therefore there is no extra chaining or appending required. Also the replenishment of DMA ring is carried out as part of the protocol processing. Therefore, the only remaining function of ISR handling becomes the notification of the kernel to start protocol processing, and therefore minimizing interrupt cost and causing minimal cache pollution.

After the notification of the arrival of a new packet, the kernel will process the packet by first examining the type of frame being received and then invoking immediately the proper handling stack function or protocol, e.g., ARP, IP, TCP, UDP, etc. Note that TCP or UDP processing includes IP processing. The packet will remain in the kernel or system memory until it is discarded or delivered to the user application. The network protocol processing for packets carried out by the kernel will continue as long as there are packets available in the system memory buffer. However, this protocol processing of packets can be disrupted by interrupts as a result of new packet arrivals. This is so because packet processing by the kernel runs at a lower priority than interrupts of incoming packets. There are two possible system delivery options of packet to user applications. The first option is to perform an extra copy of packet from kernel space to user space. This is done as part of the OS protection and isolation of user space and kernel space. This option will stretch the time of protocol processing for each incoming packet. A second option eliminates this extra copy using different techniques described in [6,7,12,36–39]. The kernel is written such that the packet is delivered to the application using block pointer manipulations (known also as zero copy) whereby the data is not copied but moved from kernel to user space by changing pointers.

3. Analysis

In this section, we present analytical models to examine the impact of interrupt overhead on OS performance. First we define the system parameters. Let λ be the mean incoming packet arrival rate and μ be the mean protocol processing rate carried out by the kernel. Note that $1/\mu$ is the average time the kernel takes to process one DMA'd packet and deliver it to user application. This time includes primarily the network protocol stack processing carried out by the kernel. $1/\mu$ includes the average cost of OS context switching and scheduling overhead, cache bouncing and pollution, protocol code execution, any checksum computation, as well as packet copying and buffering [4,40,41]. However, $1/\mu$ excludes any time disruption due to interrupts. Examples of protocol processing can be IP processing, TCP processing, and UDP processing. For TCP and UDP processing, IP processing would be included. Let $1/r$ be the mean time for handling an incoming packet interrupt. This mean time is essentially the overall interrupt cost which includes interrupt overhead and handling. We will refer to this overall cost $1/r$ simply by "*ISR handling*".

Throughout our analysis, we assume the following:

(i) It is intuitive to assume the times for protocol processing or ISR handling to be not deterministic. These times change due to various system loads and conditions, I/O activities, kernel activities, as well as cache pollution/bouncing. For example ISR handling for incoming packets can be interrupted by other interrupts of higher priority, e.g., timer interrupts. Also, protocol processing can be interrupted by higher priority kernel tasks, e.g., scheduler. For our analysis, we assume these service times to be exponential.

(ii) The network traffic follows a Poisson process, i.e., the packet inter-arrival times are exponentially distributed. In many situations, assuming Poisson arrivals is adequate. In [42], it was concluded that modeling the voice traffic as Poisson gives adequate approximation, especially if the voice traffic is high.

(iii) The packet sizes are fixed. This assumption is true for Constant Bit Rate (CBR) traffic such as uncompressed interactive voice and video conferencing.

Our analytical models assume Poisson traffic with fixed-size packets. In practice, network traffic is not always Poisson and packets are not always fixed in size. In [43–45], it was shown that the aggregated Ethernet traffic (resulting from network applications and services) is bursty and characterized as self-similar with long-range dependence. An analytical solution becomes intractable when considering variable-size packets and non-Poisson arrivals. In Section 5.1, we use simulation to study and compare the impact of bursty traffic with empirical variable packet sizes on system performance.

3.1. Ideal mode

In the ideal system, we assume the overhead involved in generating interrupts is totally ignored. The ideal system gives the best performance that can possibly be obtained when employing interrupts, thus serving as a reference or a benchmark to compare with. Under our stated assumptions, we can simply model such a system as an $M/M/1/B$ queue with a Poisson packet arrival rate λ and a protocol processing time that has an exponential distribution with a mean of $1/\mu$. B is the maximum size the system buffer can hold. $M/M/1/B$ queueing model is chosen as opposed to $M/M/1$ for two important reasons. First, in $M/M/1/B$, the arrival rate can go beyond the service rate, i.e., $\lambda > \mu$. This assumption is a must for Gigabit environment where under heavy load λ can be very high compared to μ. Second, hosts practically and realistically has a finite amount of buffer space reserved for protocol processing.

3.2. Normal interruption

In normal interruption, every incoming packet causes an interrupt. Modeling normal interruption mode is based on first determining the CPU utilization for ISR handling, next finding the mean effective or disrupted protocol processing rate, and then modeling the protocol processing as $M/G/1/B$ queueing system with this mean effective rate. More details on this model can be found in [25]. In [25], the system performance was *only* studied in terms of throughput. In this paper we extend the analysis work to examine more performance metrics. In particular we study system latency and CPU availability for user applications.

The CPU utilization for ISR handling, U_{ISR}, was derived in [25] and expressed as

$$U_{ISR} = \left(\frac{\lambda}{\lambda + r}\right). \tag{1}$$

The mean effective service rate μ' for protocol processing was computed in terms of CPU percentage availability for protocol processing. The mean effective service rate was expressed as

$$\mu' = \mu \times (\%\text{CPU availability for protocol processing}),$$
$$\mu' = \mu \times (1 - U_{ISR}),$$
$$\mu' = \mu \times \frac{r}{\lambda + r}. \tag{2}$$

The term $\frac{r}{\lambda+r}$ is the percentage of CPU bandwidth available for protocol processing, and is equal to $1-\frac{\lambda}{\lambda+r}$.

3.2.1. CPU availability

For such a model, the percentage of CPU power or bandwidth available for other processing, including user applications, is basically the probability when there is no ISR handling and there are no packets being processed by the protocol stack. It is to be noted from Eq. (2) that the effective service time is exponential. Therefore, the protocol processing can be modeled as an $M/M/1/B$ queue with a mean service rate of μ'. Hence, the CPU availability for other processing can be expressed as

$$V=\left(\frac{r}{\lambda+r}\right)\cdot p_0, \tag{3}$$

where p_0 is the probability of not queueing, i.e. finding zero packets, in the $M/M/1/B$ queueing system of the kernel's protocol processing. p_0 is known as

$$p_0=\frac{1-\rho_{IP}}{1-\rho_{IP}^{B+1}}, \tag{4}$$

where $\rho_{IP}=\left(\frac{\lambda}{\mu'}\right)$. Note that ρ_{IP} is the network load, or traffic intensity, being encountered due to kernel's protocol processing.

3.2.2. Mean system throughput

System throughput, in this context, refers to the achievable throughput of protocol processing of the OS networking subsystem. Consequently, the mean system throughput γ is basically the departure rate due to protocol processing. γ can be derived multiple ways, which are all mathematically equivalent. One way is to express γ as

$$\gamma=\mu'(1-p_0), \tag{5}$$

where p_0 is expressed by Eq. (4).

3.2.3. Mean system latency

System latency, in this context, refers to the delay encountered from the arrival of the packet into host memory (i.e., after being completely DMA'd into protocol processing buffer) until the completion of protocol processing of the OS networking subsystem. The mean system latency per packet is affected by both ISR handling and protocol processing. An incoming packet experiences a delay due to interrupt handling and due to the delay of protocol processing. Accordingly, the mean system delay is therefore decomposed to be the sum of the mean delay of interrupt handling plus the mean delay of protocol processing. Hence the total mean system delay, $E[T]$, can be expressed as

$$E[T]=E_{ISR}[T]+E_{IP}[T],$$

where $E_{ISR}[T]$ is the mean delay due to ISR and $E_{IP}[T]$ is mean delay due to protocol processing. $E_{ISR}[T]$ is simply $1/r$. This is so due to the nature of servicing packets during ISR handling. The mean ISR handling time for one packet or many packets is practically the same, i.e. $1/r$. As for the mean delay caused by protocol processing, $E_{IP}[T]$, is simply the mean delay encountered in the $M/M/1/B$ queueing system with $\rho_{IP}=\left(\frac{\lambda}{\mu'}\right)$. Therefore, such a delay can be expressed using Little's theorem as $E_{IP}[T]=\frac{E_{IP}[N]}{\lambda'}$, where $E_{IP}[N]=\frac{\rho_{IP}}{1-\rho_{IP}}-\frac{(B+1)\rho_{IP}^{B+1}}{1-\rho_{IP}^{B+1}}$ (which is the average packets in the system) and λ' is the mean effective arrival rate. λ' is basically the same as γ and is expressed of Eq. (5). Therefore, the mean system delay is expressed as

$$E[T]=\frac{1}{r}+\frac{E_{IP}[N]}{\lambda'}. \tag{6}$$

3.2.4. Special case

There is a special case of interest that can be used to verify our analysis and mathematical derivation. The case is when interrupt handling is ignored, i.e., the case of the ideal system when $1/r=0$. The mean effective protocol processing rate of Eq. (2) becomes just μ as follows: $\mu'=\lim_{r\to\infty}\mu\cdot\left(\frac{r}{\lambda+r}\right)=\lim_{r\to\infty}\mu\cdot\left(\frac{1}{\lambda/r+1}\right)=\mu$.

3.3. Interrupt disabling and enabling

The key idea behind interrupt disable–enable handling scheme is inspired by [5]. This scheme is used by some OSes such as the case of Linux NAPI [19]. In short, the idea of pure interrupt disable–enable scheme is to have the interrupts of incoming packets turned off or disabled as long as there are packets to be processed by kernel's protocol stack, i.e., the protocol buffer is not empty. When the buffer is empty, the interrupts are turned on again or re-enabled. Any incoming packets (while the interrupts are disabled) are DMA'd quietly to protocol buffer without incurring any interrupt overhead.

Fig. 2 exhibits a Markov chain model for the behavior of the interrupt disable–enable scheme with finite buffer. This model is more realistic than a simpler model which was presented in [20] considering *infinite* buffer. The state space of Fig. 2 is defined as $S=\{(n,\ 0),1\leqslant n\leqslant B\}\cup\{(n,1),0\leqslant n\leqslant B\}$ where n denote the number of packets in the buffer, and B denote the buffer size. States $(n,1)$ define the states in which the interrupts are enabled. States $(n,0)$ define the states in which the interrupts are disabled. State $(0,1)$ represents the state where the system is idle (with no packets) and the interrupts are enabled. We let $1/v$ denote the mean processing service time when the interrupts are enabled. We assume $1/v$ is exponentially distributed and it includes: the time to disable the interrupts, the time to handle interrupt (with a mean of $1/r$), and the time to service one packet by the kernel's protocol stack (with a mean of $1/\mu$). For simplicity, we ignore the time for re-enabling the interrupt. This delay is very small (typically one or two write instruction to the NIC's control register). It was shown that this delay has negligible impact on simulation results shown in Section 4.

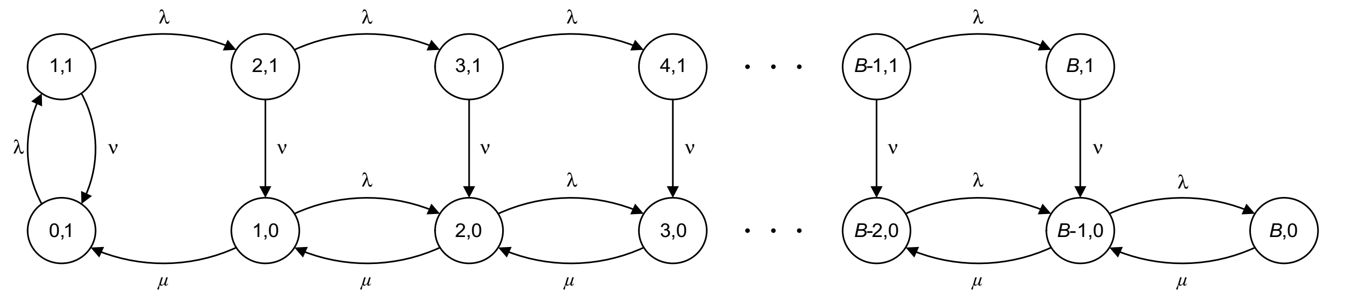

Fig. 2. Markov state transition diagram for modeling interrupt disable–enable scheme.

Let $p_{n,m}$ be the steady-state probability at state (n,m). The stationary equations of the Markov chain in Fig. 2 are:

$$0 = -\lambda\, p_{0,1} + v p_{1,1} + \mu p_{1,0}$$
$$0 = -(\lambda + v)p_{1,1} + \lambda\, p_{0,1}$$
$$0 = -(\lambda + \mu)p_{1,0} + v p_{2,1} + \mu p_{2,0} = 0$$
$$0 = -(\lambda + v)p_{n,1} + \lambda p_{n-1,1} \quad 1 \leqslant n < B$$
$$0 = -(\lambda + \mu)p_{n,0} + \lambda p_{n-1,0} + v p_{n+1,1} + \mu p_{n+1,0} \quad 1 \leqslant n < B$$
$$0 = -v p_{B,1} + \lambda p_{B-1,1}$$
$$0 = -\mu p_{B,0} + \lambda p_{B-1,0}$$

Let $a = \lambda/\mu$ and $b = \lambda/(\lambda + v)$. By applying mathematical induction, the solution of the above system is given as follows:

$$p_{n,0} = \frac{\left(ab^{n+1} - a^{n+1}b\right)}{b-a}p_{0,1}, \quad 1 \leqslant n < B$$
$$p_{n,1} = b^n p_{0,1}, \quad 1 \leqslant n < B$$
$$p_{B,1} = \frac{b^B}{1-b}p_{0,1}, \text{ and}$$
$$p_{B,0} = \frac{a\left(ab^B - a^B b\right)}{b-a}p_{0,1}.$$

The probability $p_{0,1}$ can be obtained by the fact that the sum of all probabilities is equal to 1. This will yield

$$\begin{aligned}1 &= p_{0,1} + \sum_{n=1}^{B-1} p_{n,0} + p_{B,0} + \sum_{n=1}^{B-1} p_{n,1} + p_{B,1}\\
&= p_{0,1} + \frac{ab}{b-a}p_{0,1}\cdot\sum_{n=1}^{B-1}(b^n - a^n) + \frac{a^2b}{b-a}(b^{B-1} - a^{B-1})p_{0,1}\\
&\quad + p_{0,1}\cdot\sum_{n=1}^{B-1} b^n + \frac{b^B}{1-b}p_{0,1}\\
&= p_{0,1}\left[1 + \frac{ab}{b-a}\cdot\sum_{n=1}^{B-1}(b^n - a^n)\frac{a^2b}{b-a}(b^{B-1} - a^{B-1})\right.\\
&\quad \left. + \sum_{n=1}^{B-1} b^n + \frac{b^B}{1-b}\right]\end{aligned}$$

Therefore,

$$p_{0,1} = \frac{1}{1 + \frac{ab}{b-a}\cdot\sum_{n=1}^{B-1}(b^n - a^n) + \frac{a^2b}{b-a}\left(b^{B-1} - a^{B-1}\right) + \sum_{n=1}^{B-1} b^n + \frac{b^B}{1-b}}.$$

Simplifying, we obtain

$$p_{0,1} = \frac{(b-a)(1-a)(1-b)}{(b-a)(1-a+ab-ab^B) - a^2(1-b)(ab^B - a^B b)}. \tag{7}$$

3.3.1. CPU availability

CPU availability for user applications is basically the idleness state which can be given by $p_{0,1}$.

3.3.2. Mean system throughput

The mean system throughput γ can be expressed as

$$\begin{aligned}\gamma &= \sum_i \mu_i p_i = \sum_{n=1}^{B} v p_{n,1} + \sum_{n=1}^{B} \mu p_{n,0}\\
&= v\sum_{n=1}^{B-1} b^n \cdot p_{0,0} + v\frac{b^N}{1-b}\cdot p_{0,0} + \mu\sum_{n=1}^{B-1}\sum_{i=1}^{n} a^i b^{n-i+1}\cdot p_{0,0}\\
&\quad + \mu\sum_{n=1}^{B-1} a^i b^{B-i}\cdot p_{0,0},\end{aligned}$$

which can be subsequently simplified to

$$\gamma = \lambda \times \frac{(b-a)(1-a+ab-b^B) - a(1-b)(ab^B - a^B b)}{(b-a)(1-a+ab-ab^B) - a^2(1-b)(ab^B - a^B b)}. \tag{8}$$

3.3.3. Mean system latency

The mean system latency $E[T]$, can be expressed straightforward as $E[T] = \frac{E[N]}{\lambda'}$, where λ' is the mean effective arrival rate, and is expressed by

$$\lambda' = \lambda\,(1 - p_B)$$

where

$$\begin{aligned}p_B = p_{B,0} + p_{B,1} &= \frac{a^2b^B - a^{B+1}b}{b-a}\cdot p_{1,0} + \frac{b^B}{1-b}\cdot p_{1,0}\\
&= \left(\frac{(1-a)(ab+b-a)}{(b-a)(1-b)}\cdot b^B - \frac{ab}{b-a}\cdot a^B\right) \times p_{1,0}\end{aligned}$$

and $E[N]$ is the average number of packets encountered due to protocol processing and can be expressed as

$$
\begin{aligned}
E[N] &= \sum_{n=1}^{B} n(p_{n,0}+p_{n,1}) \\
&= \sum_{n=1}^{B-1} n\cdot\left(\frac{ab^{n+1}-a^{n+1}b}{b-a}+b^n\right)p_{1,0} \\
&\quad + B\cdot\left(\frac{(1-a)(ab+b-a)b^B}{(b-a)(1-b)}-\frac{a^{B+1}b}{b-a}\right)p_{1,0} \\
&= p_{1,0}\times\left[\left(\frac{ab+b-a}{b-a}\right)\sum_{n=1}^{B-1} nb^n-\left(\frac{ab}{b-a}\right)\sum_{n=1}^{B-1} na^n\right. \\
&\quad \left. +B\cdot\left(\frac{(1-a)(ab+b-a)b^B}{(b-a)(1-b)}-\frac{a^{B+1}b}{b-a}\right)\right]
\end{aligned}
$$

3.3.4. Special case

Let us consider the special case when $v=\mu$. It can be easily verified that all the equations derived for the throughput, latency and CPU availability will be exactly reduced to those of a pure M/M/1/B queueing system with an arrival rate of λ and service rate of μ.

3.4. Polling

The idea of polling is to disable interrupts of incoming packets altogether and thus eliminating interrupt overhead completely. In polling, the OS periodically polls its host system memory (i.e., protocol processing buffer or DMA Rx Ring) to find packets to process. In general, exhaustive polling is rarely implemented. Polling with quota is usually the case whereby only a maximum number of packets is processed in each poll in order to leave some CPU power for application processing. There are primarily two drawbacks for polling. First, unsuccessful polls can be encountered as packets are not guaranteed to be present at all times in the host memory, and thus CPU power is wasted. Second, processing of incoming packets is not performed immediately as the packets get queued until they are polled. Selecting the polling period is crucial. Very frequent polling can be detrimental to performance as significant overhead can be encountered at each poll. On the other hand, if polling is performed infrequently, packets may encounter long delays.

3.4.1. Pure polling vs. NAPI polling

Polling was proposed in [3,17–19,21,22] to mitigate completely the interrupt overhead generated by packet arrivals. Releases of FreeBSD 4.6 and Linux 2.6 and thereafter can be configured for polling mode. Both of these releases use polling with quota, however, there is a difference. In FreeBSD polling, the interrupt is completely disabled for incoming packets. The algorithm for pure polling (or basic FreeBSD-style) is illustrated in Fig. 3. During the polling period, a limited number of packets, say Q, are processed by the protocol stack. In the situation where polling is triggered while in the midst of a polling cycle (i.e., servicing packets by the protocol stack), the trigger is ignored, but polling is turned on again and overhead is incurred which is purely a waste of CPU cycles.

In Linux NAPI polling [19,28], a combination of the scheme of interrupt disabling–enabling and polling is used. This is achieved by disabling the interrupts of incoming packets once a packet is received and triggering polling *immediately* (as illustrated in the NAPI algorithm of Fig. 4). After processing Q packets, if the protocol processing buffer is not empty, polling is triggered again on the next polling cycle; otherwise, polling is turned off and the interrupts of incoming packets are re-enabled. The key idea behind Linux NAPI polling is to combine the mitigation of interrupt overhead at high load while improving the responsiveness at low load.

The polling period in the latest versions of FreeBSD and Linux is not deterministic. Linux polls occur with softirqs. As all softirqs, they get typically executed at end of hardware interrupt and just before returning from kernel to user mode. FreeBSD polls occur at end of clock interrupt and system calls, and within idle loops. With these techniques, context switching overhead and cache pollution are

```
Poll_Int (start of polling cycle $T_{Poll}$)
    1. Set Poll_Mode ON
    2. Trigger IP Processing

IP Processing
    1. Set Poll_Mode OFF
    2. Process up to $Q$ packets or until no
       more packets in buffer
    3. If Poll_Mode is ON goto Step 1
```

Fig. 3. Pure polling algorithm.

```
Rcvd_Pkt_INT
    1. Disable Rcvd_Pkt_INT
    2. Enable Poll_INT and
       trigger the handling
       of Poll_INT

Poll_INT (start of polling cycle $T_{Poll}$)
    1. Set Poll_Mode ON
    2. Trigger IP Processing

IP Processing
    1. Set Poll_Mode OFF
    2. Process up to Q packets or
       until no more packets in
       buffer
    3. If Poll_Mode is ON goto
       Step 1
    4. If buffer is empty then
       disable Poll_INT and
       enable Rcvd_Pkt_INT
```

Fig. 4. NAPI polling algorithm.

decreased. It is also possible to avoid more context switching overhead and cache pollution in polling by utilizing soft timers [18]. With soft timers, the OS can choose to poll the protocol buffer at more "convenient" or "trigger" points. OS convenient points can occur when the system is already in the right context and has already suffered cache pollution [18]. As opposed to Linux, the quota during polling in FreeBSD is dynamically adjusted. In FreeBSD, the quota depends on a number of configured parameters such as system load, CPU speed, remaining CPU fraction for polling process, and maximum quota. In this paper and for the sake of performance comparison, we will use a fixed polling period and a fixed quota. Pure polling is studied using mathematical analysis and simulation, while NAPI polling is studied using simulation.

Let us assume T_{Poll} is the polling period, that is, a hardware timer is configured to periodically generate an interrupt every T_{Poll}. Obviously, there is a polling overhead associated with each polling period that involves context switching overhead and notifying the protocol processing. We denote T_{PollOH} for this polling overhead. In this case where T_{Poll} is generated by an interrupt, the polling overhead is close to the cost of interrupt overhead. Note that the cost of polling overhead would be an order of magnitude less than the interrupt overhead when using soft timers to trigger polling [17,18,22]. For our analysis in this paper, we use hardware timers as opposed to software timers for the generation of T_{Poll}, and hence T_{Poll} is deterministic. The analysis of using soft timers for polling is left for future work.

Let us assume for now a deterministic distribution for both T_{PollOH} and $1/\mu$. Hence, for our analysis the time is divided into a sequence of polling slots lasting exactly T_{Poll} time units. Fig. 5 illustrates the time division of a T_{Poll} which is comprised of an initial polling overhead, lasting T_{PollOH}, a time span used for processing up to Q packets, lasting up to Q/μ, and possible an idle (or leftover) period lasting until the end of T_{Poll}. Obviously protocol processing may not run at the full rate of μduring T_{Poll} (in the situation when Q/μ ends before the end of T_{Poll}). Running at full rate means that protocol processing for packets will continue back-to-back without giving away the CPU during the polling period.

Therefore, the effective mean protocol processing rate $\mu^{'}$ can be expressed as

$$\mu' = \begin{cases} Q/T_v & (Q/\mu < T_v), \\ \mu(1 - U_{PollOH}) & (Q/\mu \geqslant T_v), \end{cases} \tag{9}$$

where $T_v = T_{Poll} - T_{PollOH}$ and $U_{PollOH} = T_{PollOH}/T_{Poll}$. U_{PollOH} is the CPU utilization due to polling overhead.

An issue related to our analysis of pure polling above is that it assumes deterministic times for protocol processing

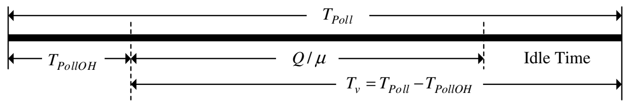

Fig. 5. Polling cycle components.

and polling overhead. It is important to recognize that the analysis for polling is not a trivial task, and it is best studied using simulation. Our polling analysis under these assumptions gave adequate approximation to those of reported results (shown in Section 4) of simulation with exponential times for protocol processing and polling overhead.

3.4.2. CPU availability

The CPU availability for user applications can be expressed as $V = 1 - (U_{PollOH} + U_{IP})$. U_{IP} is the CPU utilization due to protocol processing. To find U_{IP}, we let N denote the average number of packets arrived during T_{Poll} and M denote the average number of packets that can be served during T_v. That is, $N = \lambda \times T_{poll}$ and $M = \mu \times T_v$. Therefore, U_{IP} can be expressed as

$$U_{IP} = \begin{cases} \frac{\lambda}{\mu} & \text{if } N < M \text{ and } N < Q, \\ \frac{Q}{\mu \times T_{poll}} & \text{else if } Q < M, \\ \frac{T_v}{T_{poll}} & \text{otherwise.} \end{cases} \tag{10}$$

Note that V becomes zero when $Q/\mu \geqslant T_v$.

3.4.3. Mean system throughput

Under our assumptions, the mean effective protocol processing rate $\mu^{'}$ expressed in Eq. (9) is constant. With a Poisson traffic rate arrival λ, an $M/D/1/B$ queueing model can be utilized to obtain the mean system throughput. The mean system throughput γ can be expressed as $\gamma = \mu^{'}(1 - p_0)$, where $\mu^{'}$ is given by Eq. (9) and p_0 is the probability for not queueing in the $M/D/1/B$ queueing system. p_0 was derived by both [46,47] and was expressed as

$$p_0 = \frac{1}{1 + \alpha\rho},$$
$$\text{where } \alpha = \sum_{k=0}^{B-1} \frac{(-1)^k}{k!}(B-k-1)^k e^{(B-k-1)\rho}\rho^k \text{ and } \rho = \frac{\lambda}{\mu'}. \tag{11}$$

3.4.4. Mean system latency

The mean system latency for polling is comprised of mainly two types of delay:

(i) a delay due to packet processing with a mean effective service rate of $\mu^{'}$ expressed in Eq. (9), and
(ii) a delay between the initial arrival of a packet and the start of the protocol processing.

The average delay of (i) can be expressed using M/D/1/B queueing model as follows

$$E_{IP}[T] = \frac{E_{IP}[N]}{\lambda(1 - p_B)} = \frac{1}{\lambda} \cdot \frac{1 + \rho b_{B-1}}{b_{B-1}} \cdot \frac{B + B\rho b_{B-1} - \sum_{k=0}^{B-1} b_k}{1 + \rho b_{B-1}},$$

where $b_0 = 1, b_n = \sum_{k=0}^{n} \frac{(-1)^k}{k!}(n-k)^k e^{(n-k)\rho}\rho^k, \quad n \geqslant 1$, and $\rho = \lambda/\mu^{'}$.

The delay of (ii) is encountered only when the packet first arrives (right after it gets DMA'd and before the start of the protocol processing cycle). It is more obvious to visualize this delay when the system has no packets. A packet may encounter an upper bound delay as high as T_{poll}. This happens when the system is empty and the packet arrives just after protocol processing has finished examining that there are no packets to process. In this case, this packet will be missed (by protocol processing) and will have to wait (a time T_{poll}) until the start of the next protocol processing cycle. On the other hand, a packet may encounter a lower bound delay of almost zero if it arrives during protocol processing cycle. As a conservative approximation for the delay of (ii), we will use the upper bound delay. Therefore, the delay of (ii) can be approximated as $p_0 \times T_{poll}$, where p_0 is the probability of not queueing which was expressed in Eq. (11).

Accordingly, the mean system delay can be approximated as

$$E[T] = E_{IP}[T] + p_0 \times T_{poll}. \tag{12}$$

3.5. Interrupt-coalescing (IC)

One of the most popular solutions to mitigate interrupt overhead for Gigabit network hosts is interrupt coalescing or IC. In recent years most network adapters or NICs are manufactured to have interrupt coalescing. Additionally, many operating systems, including Windows and Linux, support IC. IC is a mode or a feature in which the NIC generates a single interrupt for a group of incoming packets. This is opposed to normal interruption mode in which the NIC generates an interrupt for every incoming packet. In interrupt-coalescing (IC) mode, there are two schemes to mitigate the rate of interrupts: count-based IC and time-based IC. In count-based IC mode, the NIC generates an interrupt when a predefined number of packets (denoted by τ) has been received. In time-based IC mode, the NIC waits a predefined time period (denoted by T) before it generates an interrupt. During this time period multiple packets can be received. The coalescing parameters of τ and T are tunable and configurable parameters which are set by the device driver. Analytical models and closed-form solutions for the key performance metrics under study (which include system throughput, latency, and CPU availability) were given in [26]. The same underlying assumptions and notations used in this paper were used in [26].

3.6. Simulation

In order to verify and validate our analytical models, a discrete–event simulation model was developed and written in C language. A detailed description and flowcharts of the simulation model for normal interruption can be found in [25]. The simulation model reported in [25] was extended for the schemes of pure and NAPI polling, count-based and time-based IC, and disabling and enabling of interrupts. The assumptions of analysis were used. The simulation followed closely and carefully the guidelines given in [48]. We used the PMMLCG as our random number generator [48]. The simulation was automated to produce independent replications with different initial seeds that were ten million apart. During the simulation run, we checked for overlapping in the random number streams and ascertain that such a condition did not exist. The simulation was terminated when achieving a precision of no more than 10% of the mean with a confidence of 95%. We employed and implemented dynamically the *replication/deletion* approach for means discussed in [48]. We computed the length of the initial transient period using the MCR (Marginal Confidence Rule) heuristic developed by White [49]. Each replication run lasts for five times of the length of the initial transient period. Analytical and simulation results, as will be demonstrated in Section 4, were very much in line.

4. Numerical examples

In this section, we report and compare results of analysis and simulation. Numerical results are given for key performance indicators. These indicators include mean system throughput, CPU utilization, and latency. We plot and compare the performance for the all interrupt schemes that include the ideal system, normal interruption, pure and NAPI polling, interrupt disable–enable, and interrupt coalescing of time-based and count-based. For our numerical examples, realistic values for system parameters, that suit modern Gigabit network environment and hosts, must be used. Experimental study is the best approach to give accurate measurements, as well as the underlying probability distributions. Such experimental is beyond the scope of this paper and left for future work. However, for the sole purpose of comparison, we base our values on modern credible experimental measurements reported in the literature.

The overall interrupt cost $1/r$ includes both interrupt overhead and handling. In [18], the interrupt overhead for an off-chip timer interrupt with a null event handler was measured to be in the vicinity of 4.36 µs on a 500 MHz Pentium-III machine running for FreeBSD-2.2.6. A similar result of 7.7 µs was reported by [1] on a 450 MHz Pentium-III machine running Linux 2.2.10. For a modern 2.53 GHz Pentium-IV machine, it is expected this overhead will not be decreased linearly by the speed frequency of the processor, as I/O and memory speed limits dominate [2]. In [2] it was concluded that the performance of 2.4 GHz processor only scales to approximately 60% of the performance of an 800 MHz processor. Consequently the NIC interrupt overhead with null handler for a modern 2.53 GHz Pentium-IV machine can be roughly 60% of 4.36, which is 2.62 µs. In [1] the interrupt handling was measured to be 5.53 µs on a 450 MHz Pentium-III machine running Linux 2.2.10. The measurement of interrupt handling included substantial work and a major cache pollution. The handling included appending the packet from DMA ring to protocol buffer, replenishment of the DMA ring, and finally notifying the protocol processing. In our case, considering the speed of the processor and limited work for the interrupt handling, which primarily includes notification of protocol processing with minimal cache pollution, we assume the handling cost is 20% of 5.53, or 1.11 μs. Hence for a modern 2.53 GHz Pentium-IV machine, the overall interrupt cost $1/r = 2.62 + 1.11 = 3.73$ μs.

For protocol processing, we use the TCP processing values measured by *lmbench* [50] on a 2.53 GHz Pentium-IV running Linux 2.4.18. The results are reported in [51]. Also results for different machines are reported in [52]. From the results in [51], it is reported that the average local *loopback* latency for one TCP token (i.e., 4-byte data packet) is 10.5 μs. This time, of course, includes OS overhead as well as TCP actual processing. Ideally, the TCP latency of the receiving path would be approximately half of 10.5, that is 5.25 μs. TCP processing also includes copying of packet payload to user application. [51] reports that the average TCP bandwidth (buffering and copying) is 748 Mbytes/s. Therefore, for a minimum packet size of 64 bytes, the cost of copying and buffering is $64/748 = 0.086$ μs. Hence, the mean TCP processing time $1/\mu$ (for a fixed size packet of 64 bytes) can be summed up to approximately 5.34 μs. In all of our examples, we fix the kernel's protocol processing buffer B to a size of 1024 packets, which occupies about 1.5 Mbytes of host memory when assuming a maximum of 1538 bytes per packet in accordance to IEEE802.3 standards. This buffer size is a configurable parameter [34].

For interrupt disabling–enabling scheme, we assume the same overall interrupt cost of 3.73 μs. However, there is also the incurred cost of writing to the NIC control registers to disable and enable interrupts of incoming packets. We assume the cost of writing to the NIC register is 0.5 μs. As a consequence, the parameter $1/v$ in analysis, which denote the mean protocol processing service time when interrupts are enabled, is approximately equal to 9.57 μs, that is $0.5 + 3.73 + 5.34$ μs. For polling, we use a quota of 3 packets per poll, a polling period of 20 μs, and a polling overhead of 1.59 μs. As measured in [18] on machines with a 500 MHz Pentium-III CPU running for FreeBSD-2.2.6, the average polling period was in the range between 12 μs and 32 μs. For our study, we chose a mean polling period of 20 μs. It is to be noted that in reality (and in current Linux and FreeBSD implementations) the polling period is not deterministic [18] and is generated by a soft timer with a deadline. The deadline is generated by a hardware timer. Lastly for count-based IC, we use the coalescing parameters of $\tau = 1$ and $\tau = 8$. For time-based coalescing we use the coalescing parameters of $T = 0$ and $T = 50$ μs.

Fig. 6 plots the mean system throughput, CPU availability for user applications, mean system latency at low load, and mean system latency at high load, respectively, as a function of the system load represented by packet arrival rate. The load and throughput are both expressed in pps (packets per second). Both of these measures can easily be expressed in bits per second by multiplying the packet rate by the packet size. For pure polling, analysis results with constant times for protocol processing and polling overhead gave adequate approximation to those of simulation results with exponential times for protocol processing and polling overhead. In order to compare easily and clearly the relatively close performance of pure and NAPI polling, the analysis results for pure polling are left out, and only simulation results (with dashed lines with circles) are reported in the figures. For other schemes, the solid curves represent analysis results and the circles are those of simulation. The figures exhibit a very close agreement between discrete–event simulation results and analysis results.

From the figures, it is observed that the maximum throughput occurs at 187 Kpps. For normal interruption, it can be noted that the saturation or cliff point for the system occurs at 127 Kpps. At this point, the corresponding CPU utilization (for ISR handling plus protocol processing) is at 100%, and thus resulting in a CPU availability of zero. Therefore, user applications will starve and livelock at this point. Fig. 6(a) shows that as the arrival rate increases after the cliff point the system throughput starts to decline. Fig. 6(d) shows the mean system delay also continues to increase after reaching the saturation point. Theoretically, the latency should flatten off at $B/\mu^{'}$, but rather we find it slowly shoots to infinity. The decline in the throughput and the sharp increase in latency is due to the fact that the mean effective service rate $\mu^{'}$ decreases as the arrival rate increases right after the saturation point. See Eq. (2). Intuitively, CPU availability for protocol processing decreases as CPU becomes more utilized handling ISR.

One observation can be made about IC schemes with a parameter of $\tau = 1$ in case of count-based coalescing and $T = 0$ in case of time-based coalescing. It is observed that in such cases, both coalescing scheme resort exactly, as expected, to normal interruption. Also from the figure, it is depicted that the analysis curves for time-based coalescing (more noticeable in Figs. 6(b) and (c) at very low rate) are not smooth. As illustrated in [26], the analysis for time-based coalescing is performed based on the analysis of count-based coalescing with the coalescing parameter τ being an integer and approximated to $\lceil \lambda T \rceil$. Thus, τ takes on discrete values and remains unchanged until a different value is produced as λ changes in $\lceil \lambda T \rceil$.

There are also a number of important observations and conclusions to be made when examining and comparing the performance of all interrupt handling schemes. It can be concluded that no single scheme gives the best performance. For example, the scheme of interrupt disabling and enabling outperforms all other schemes in terms of throughput and latency. However, in terms of CPU availability, the interrupt disabling and enabling gives the worst performance second to normal interruption. Also at extre-

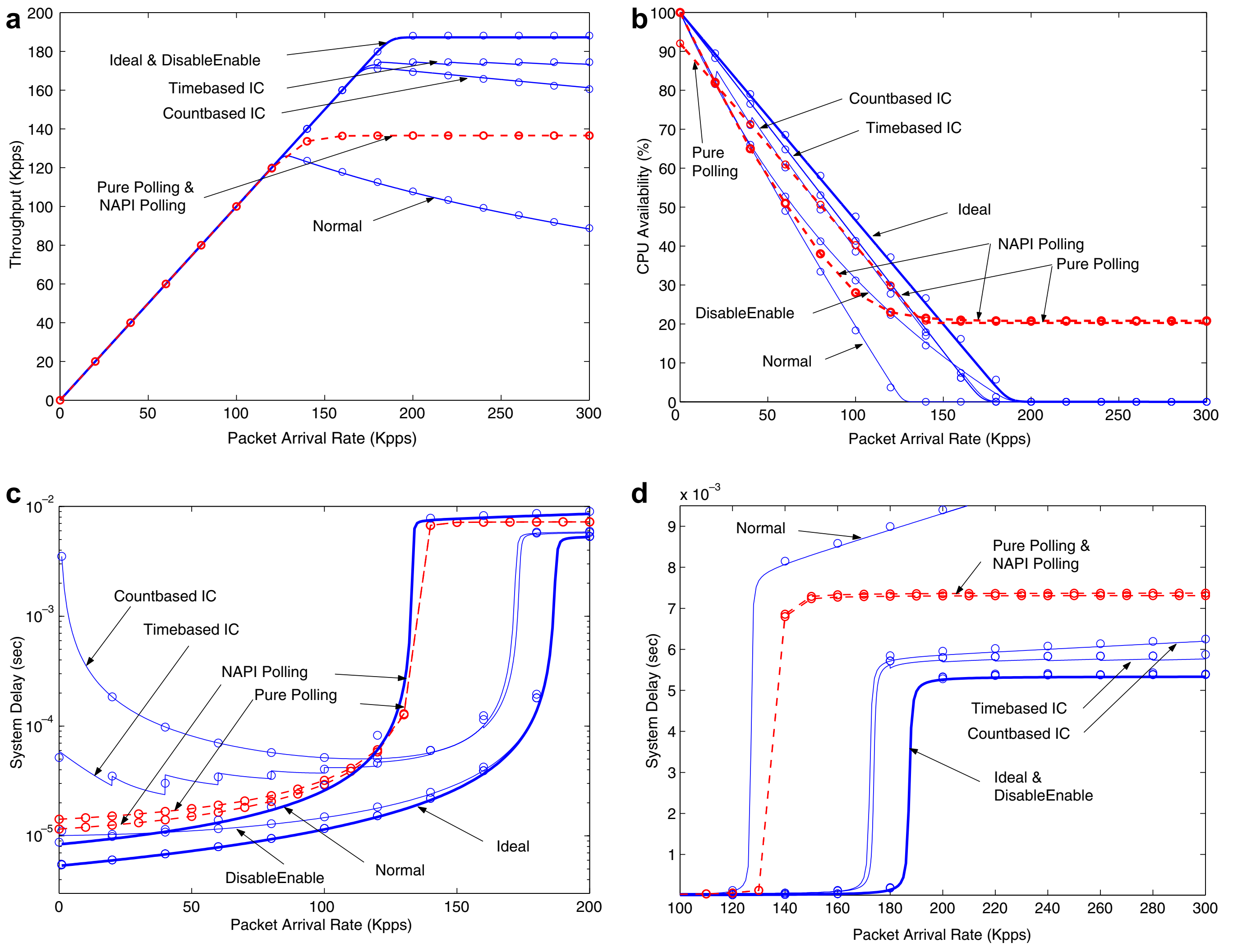

Fig. 6. Key performance indicators in relation to a Poisson arrival rate.

mely low rate interrupt disabling and enabling gives worse latency than normal interruption. When comparing pure polling with NAPI polling, it is obvious from the plots that both give comparable results in terms of throughput and latency at high load. However, NAPI polling outperforms pure polling in terms of CPU availability and latency at low load. Comparing interrupt disabling and enabling to polling, it is shown that the latency of polling at light load is larger than interrupt-disable scheme, and also at high load the system throughput of polling is smaller. However, for CPU availability, polling outperforms all other schemes at high load, as depicted in Fig. 6(b). As expected polling can sustain a certain degree of CPU availability regardless of the presented load. When comparing polling to IC, it can be noted that IC can give similar throughput and CPU availability (and latency at very high load) with large coalescing values of τ and T. However, with large coalescing values, the latency at low load degrades considerably. We will address in Section 5.2 the impact of selecting different values for scheme parameters on the overall performance.

5. Discussion

Thus far we presented analysis and simulation work to study and compare the performance of various interrupt-handling schemes considering fixed packet sizes and Poisson arrival. In this section, we address two related important questions that may affect the performance: (1) subjecting the host to bursty traffic, and (2) the selection of different parameter values for each scheme. Finally in this section and based on obtained results and observations, we propose and discuss briefly a novel hybrid interrupt-handling scheme that yields the best performance.

5.1. Impact of bursty traffic

Assuming Poisson arrival for network traffic can be valid for modeling real-time voice and video traffic [42]. However, such a Poisson traffic fails for modeling Ethernet traffic. It was shown that Ethernet traffic is bursty and characterized as self-similar with long-range dependence [44,45]. A comprehensive summary and review of the topic of self-similar network traffic can be found in [53]. For examining the impact of bursty traffic on the performance of the various interrupt-handling schemes, we modified our discrete–event simulation accordingly. To generate such a bursty traffic, we implemented the method described in [54]. This method follows fractional Gaussian noise such as the resulting self-similar traffic is obtained by aggregating multiple streams (one stream per source) each consisting of alternating Pareto-distributed *ON*/*OFF* periods. In previous preliminary work, we used a total of 100 streams to generate bursty traffic [55]. In this paper, only 8 streams are used. As few as 8 streams were reported in [56] to give as good results and significantly reduced the simulation run time to almost 20%. Fig. 7 illustrates graphically the aggregation of multiple streams.

Pareto distribution is a heavy-tailed distribution with PDF of $f(x) = ab^{\alpha}/x^{\alpha+1}$, $x \geqslant b$, where α is a shape parameter and b is a location parameter. We use this distribution to generate both the *ON* and *OFF* periods with shape parameters of $\alpha_{ON} = 1.3$ and $\alpha_{OFF} = 1.5$, respectively. The choice of the values of these shape parameters are commonly used and promoted by measurements on actual Ethernet traffic performed in [45]. The location parameter of b_{ON} is the minimum *ON* period and depends on the minimum Ethernet frame size of 64 bytes. This is fixed to 64×8 bit times or 512 ns. The calculation of b_{OFF} can be computed from the desired total load for all sources, i.e., $\rho_{Total} = \sum_i \rho_i$. We assume equal loads for all sources. The individual load of a single source is measured as $\rho_i = E[ON]/(E[ON] + E[OFF])$. Note that the load ρ_{Total} takes on values between 0 and 1. In Pareto, $E[ON] = (\alpha_{ON} b_{ON})/(\alpha_{ON} - 1)$ and $E[OFF] = (\alpha_{OFF} b_{OFF})/(\alpha_{OFF} - 1)$. Solving these simple formulas, b_{OFF} can be expressed (in terms of the known parameters of ρ_{Total}, b_{ON}, α_{ON}, and α_{OFF}) as

$$b_{OFF} = \left(\frac{\alpha_{ON} b_{ON}}{\alpha_{OFF}}\right)\left(\frac{\alpha_{OFF} - 1}{\alpha_{ON} - 1}\right)\left(\frac{1 - \rho_i}{\rho_i}\right).$$

During the *ON* period, packets are generated back-to-back with a rate of 1 Gbps. The number of packets generated in the *ON* period depends on the *ON* period, the packet size, and the inter-packet size. The inter-packet size is 20 bytes which comprises of the standard minimum Ethernet IFG of 12 bytes plus 8 bytes for the preamble. The packet sizes are not fixed and follow an empirical distribution, which are real measurements of packet sizes from MCI backbone. The measurements are reported in [57] and available online at http://www.caida.org. In [57], the reported packet size distribution represents IP datagram sizes. To obtain Ethernet frame size distribution, the packet sizes were modified to include 18-byte header (12 bytes for destination and source addresses, 2 bytes for length/type, and 4 bytes for FCS). In addition all bytes shorter than 46 bytes were padded to 46 bytes, so that the minimum Ethernet frame size is equal to 64 bytes. The histogram and CDF of the resulting Ethernet frame sizes are shown in Fig. 8. The figure shows dominating frame sizes of 64, 570, 594, and 1518 bytes.

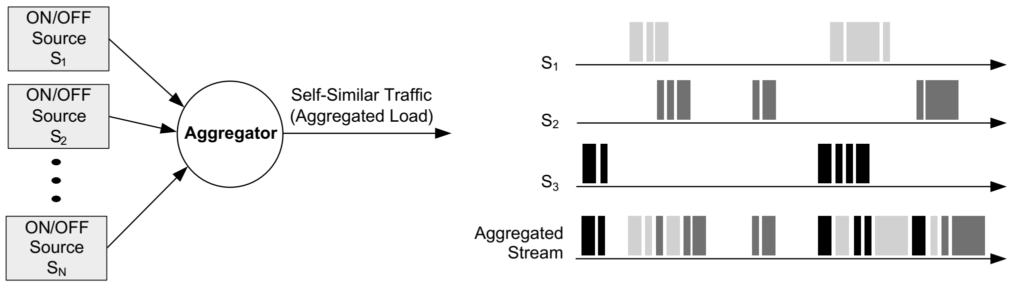

Fig. 7. Self-similar traffic generation model.

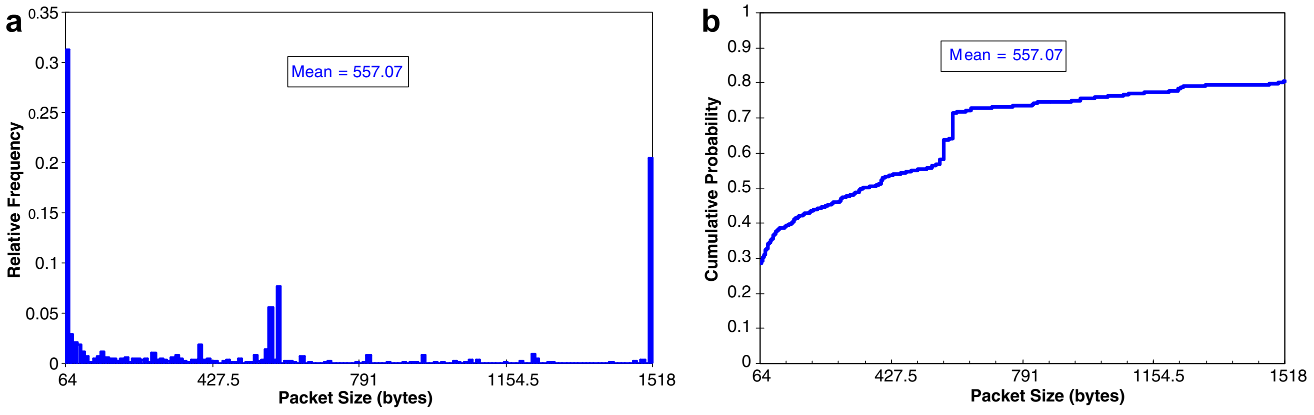

Fig. 8. Histogram of empirical packet sizes and their corresponding CDF.

It is to be noted, and as described in Section 4, when packet sizes are variable, the service time for protocol processing is strongly correlated with the packet size, primarily due to CRC checksum calculation and copying to application layer. In this case, the service time for protocol reprocessing is comprised of packet overhead which is exponentially distributed with a mean of 5.25 µs plus a fixed per-byte overhead of PacketSize/(748 Mbytes/s).

The simulation results shown in Fig. 9 represent the estimated mean of 10 simulation replications. We had to fix the number of replications when generating bursty traffic. As opposed to the simulation carried out with Poisson traffic, a simulation run with bursty traffic can not be automated to stop when achieving a desired precision for the estimated mean. This is because of the presence of irregular incoming traffic. The irregularity of traffic is due to the use of empirical variable packet sizes and the superposition of multiple streams with each stream producing *ON* and *OFF* periods (with huge variance) modeled by heavy-tailed distribution such as that of Pareto. The problem is exacerbated when the Pareto's shape parameter α is close to 1 (as is the case for the shape parameters for our *ON* and *OFF* periods). Therefore, simulation with such traffic will be very slow to converge to steady state and thus the CI (Confidence Interval) can be very long [58]. In order to obtain relatively acceptable accuracy and precision, simulation has to produce a huge number of samples [58]. For our simulation, each replication generated for each source at least 10 million samples for its *ON* period and another 10 millions for its *OFF* period. Care was taken to make sure that there is no overlapping in the random number streams of simulation. Table 1 shows the estimated mean with 95% CI for system latency of time-based coalescing when $T = 50$ µs for 10 simulation runs. The precision is defined as the percentage error in the estimated mean which is equal to CI half length divided by the estimated mean. It is noted that very low loads and high loads contribute to a longer CI, as the simulation becomes less stable and the estimators converge very slowly to their true values, especially when traffic is irregular [59]. The length of the CI of the other performance metrics exhibited the same characteristics.

Fig. 9 plots the three key performance metrics of mean system throughput, CPU availability, and latency as a

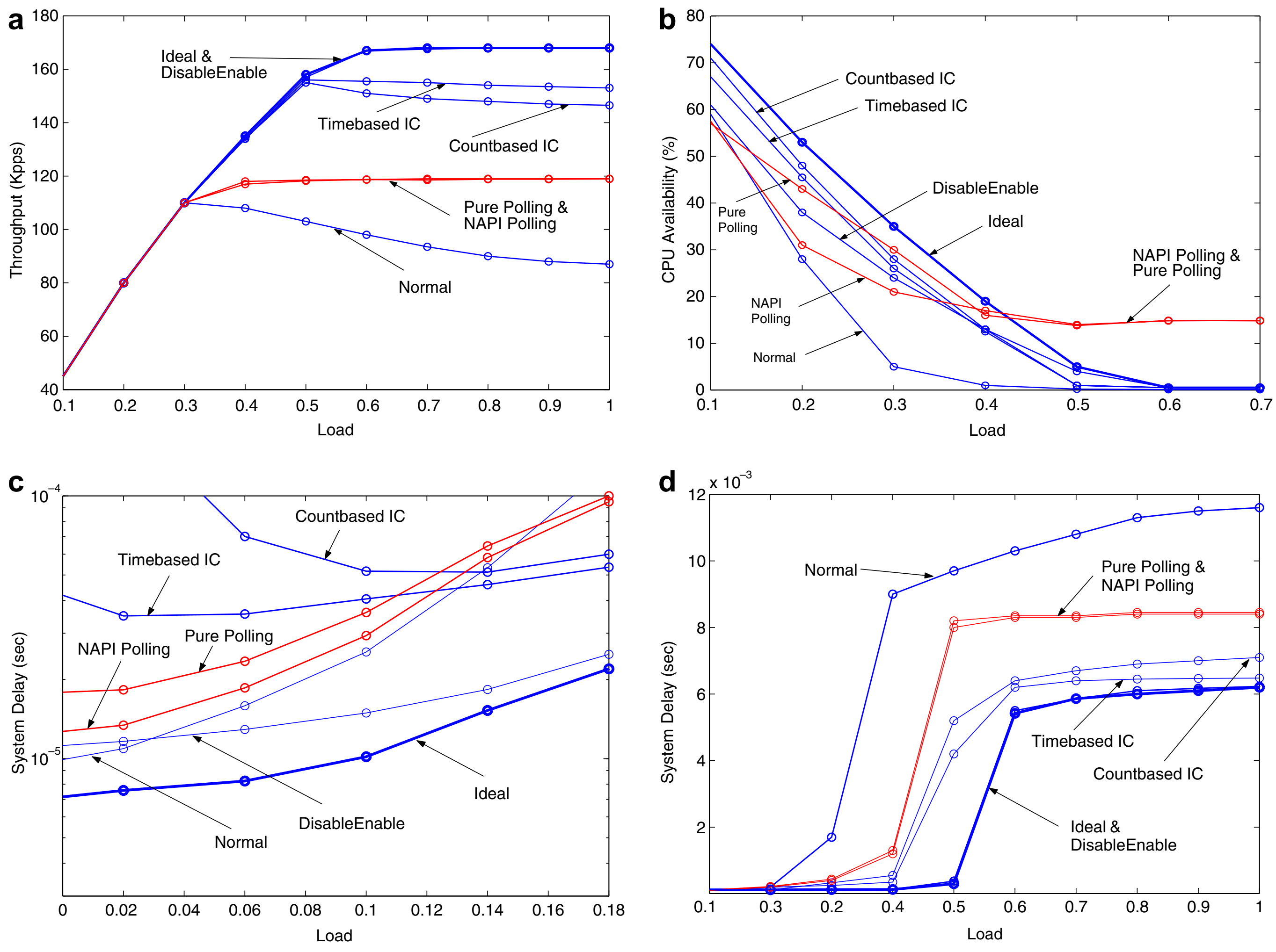

Fig. 9. Key performance indicators in relation to system load of bursty traffic.

Table 1
Simulation estimates for system latency (in ms) from 10 replications

Load	Lower 95%	Upper 95%	Estimated mean	CI half length	Precision (%)
0.0004	0.0302	0.0612	0.046	0.015	34
0.02	0.0259	0.0508	0.038	0.012	32
0.1	0.0299	0.0513	0.041	0.011	26
0.2	0.1411	0.2312	0.186	0.045	24
0.4	0.2885	0.4221	0.355	0.067	19
0.6	5.2582	7.1252	6.192	0.934	15
0.8	4.9112	7.5692	6.240	1.329	21
1.0	3.9312	8.6321	6.282	2.350	37

function of the aggregated self-similar traffic load ρ_{Total}. By subjectively eyeballing the performance curves and comparing their shapes to the performance curves of Poisson traffic, we find the shapes of the curves are very similar for the most part. Fig. 9(a) shows at low load a curved shape for the achievable system throughput as opposed to straight one. This is because the units for the throughput and the load are not the same. Remember that the load ρ_{Total} takes a value between 0 and 1. Also note that the highest average system throughput is around 167 Kpps. This is expected as the average empirical packet size, based on the distribution presented in Fig. 6, is 557 bytes. The average protocol processing service time is the sum of fixed packet overhead of 5.25 µs plus a fixed per-byte overhead of PacketSize/(748 Mbytes/s). This yields an average service time of 5.99 µs, or a throughput of 167 Kpps.

5.2. Effect of selecting different parameter values

A question related to the performance of a certain interrupt-handling scheme is the selection of its parameter values. One may argue that the parameter values selected may favor one scheme than the other. In this section, we study how the selection of parameter values impacts the performance of a particular scheme. Fig. 10 shows the impact of selecting different values for the parameters of time-based and count-based coalescing as well as pure and NAPI polling. The units of the time-based coalescing parameter T and polling period T_{Poll} are in µs.

For coalescing, Figs. 10(a–c) shows that selecting large values for the count-based coalescing parameter τ can be very detrimental to performance in terms of latency at low load. At very low arrival rate, packets have to wait longer time to be coalesced. The larger the value of τ, the larger the coalescing time. In time-based coalescing, there is a bound on the waiting time, which is the value of T.

For polling, Figs. 10(d–f) shows the impact of selecting the polling quota Q on performance. For these examples, T_{Poll} is fixed to 20 µs. When varying Q, it is noted when Q is large ($Q = 4$), protocol processing will be done at full speed, and thus resulting in a CPU availability to drop to 0 at high load. However, when Q is small ($Q = 2$), CPU availability is best, but resulting in considerable latency and degraded throughput. Selecting the polling period is crucial. Fig. 10g–i study such an effect. For these figures, we fix Q to 4. The figures are in line with intuition and show that small values of T_{Poll} (i.e., frequent polling) will result in more overhead and thus degraded performance in terms of throughput and CPU availability. However, when T_{Poll} takes on large values (i.e., infrequent polls), more latency will be encountered. NAPI polling outperforms pure polling in terms of latency, especially at low rate. However, pure polling outperforms NAPI polling in terms of CPU availability at moderate load as depicted in Fig. 10e. At very low load, pure polling has much smaller CPU availability. This is expected as pure polling has always fixed overhead of only the timer interrupt and polling overhead. However, NAPI has the additional overhead of enabling and disabling the interrupts for incoming packets, and thus acts similar to the scheme of enabling and disabling interrupts at low to moderate load.

5.3. Hybrid scheme

It was concluded from numerical examples given in Section 4 and the discussion in Sections 5.1 and 5.2 that no particular interrupt-handling scheme gives the best performance under all load conditions. Selection of the appropriate scheme depends primarily on the system performance requirements, most important performance metric of interest, and traffic load. It was shown by giving numerical examples that the scheme of disabling and enabling interrupts outperforms, for the most part, all other schemes in terms of throughput and latency. However, when it comes to CPU availability, pure polling is the most appropriate scheme to use. Based on these important observations and in order to compensate for the disadvantage of interrupt disable–enable scheme of poor CPU availability, we propose a novel hybrid scheme of interrupt disable–enable and pure polling. This hybrid scheme would make up for the CPU availability drawback of interrupt disable–enable scheme when the host is under heavy load. In short, the scheme would operate in interrupt disable–enable until reaching a heavy load region at which the system must switch to pure polling. The selection of parameters for pure polling period and quota at high load rate depends largely on what the most important performance metric is. For example, if latency has a more dominant weight than throughput and CPU availability, a small value for the polling period and a larger quota are desirable. In [5,17,22], a hybrid scheme of normal interruption and polling was proposed. As demonstrated in this paper, normal interruption performs very poorly at light and moderately light loads in terms of throughput, CPU availability, and latency. Also in [5,17,22], the switching between normal interruption and polling was done somewhat arbitrarily.

Identifying the switching point is critical. The switching point should be in the vicinity of or just before the cliff point occurs. Under Poisson traffic, our analytical work provided equations to identify where cliff point occurs. The cliff point can be simply identified as the saturation

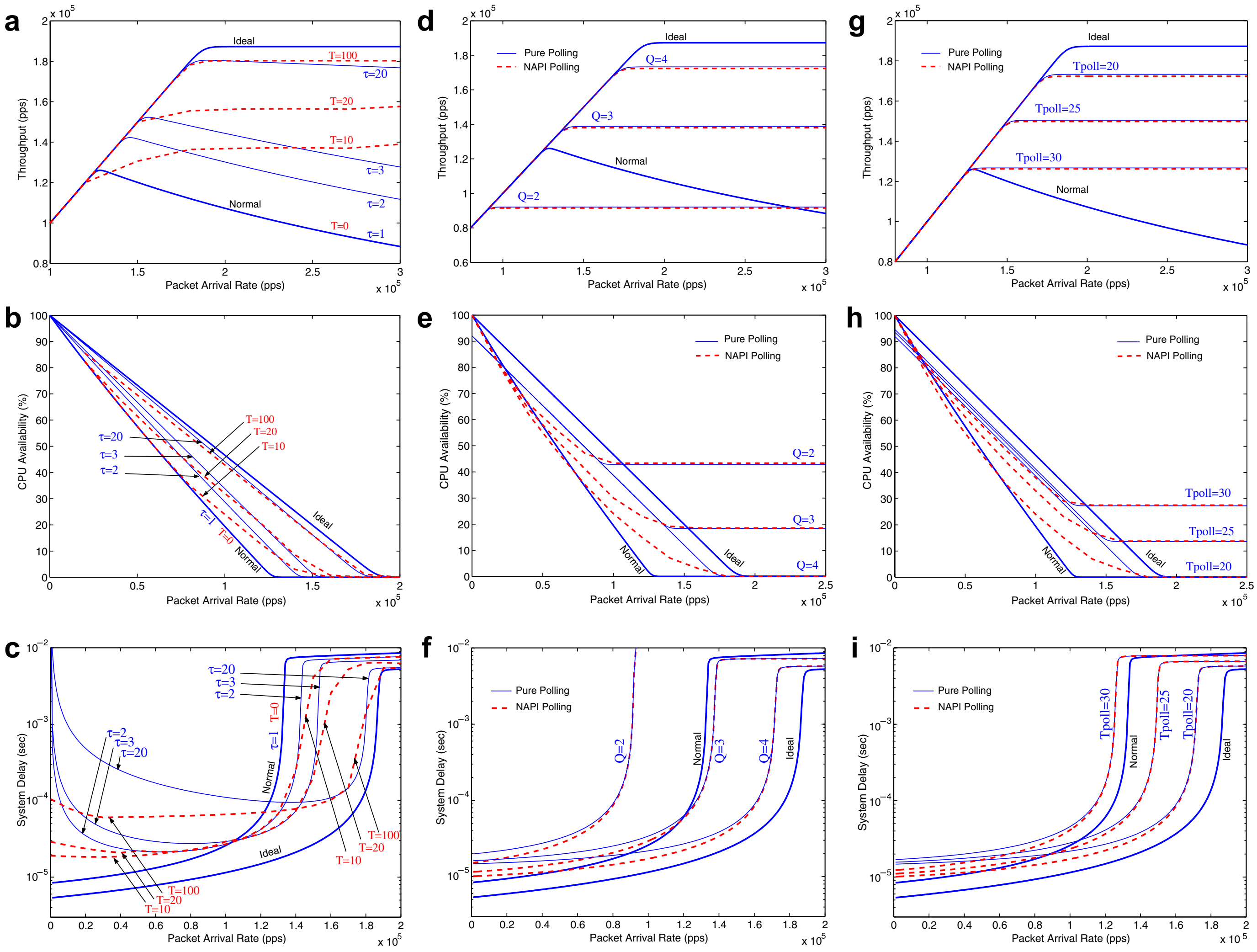

Fig. 10. Effect of parameter values of coalescing and polling on performance.

point of normal interruption. The cliff point occurs when $\rho_{IP} < 1$ or $\lambda < \mu \cdot \frac{r}{\lambda + r}$. Solving for λ, the saturation point or cliff point can be expressed as

$$\lambda_{cliff} = \frac{r}{2}\left(\sqrt{1 + 4\frac{\mu}{r}} - 1\right). \quad (13)$$

For Poisson traffic, Eq. (13) gives a cliff point at 127 Kpps, and thus a proper switching point can be selected to be at 120 Kpps. For bursty traffic, Eq. (13) surprisingly gives an adequate approximation for the cliff point. The cliff point can be computed if the average packet size is measured. Based on the empirical packet size distribution, the average packet size is 557 bytes. As discussed in Section 5.1, this yields an average protocol service time of 5.99 μs, or a rate of 167 Kpps. Consequently and applying Eq. (13), the cliff point λ_{cliff} for bursty traffic is 116 Kpps, which is very much in line with the simulation results of Fig. 9a. And therefore, a proper switching point can be at a system throughput of 110 Kpps.

From implementation point view, the OS can be modified to measure the average packet size and the average system throughput at the protocol level, and accordingly switch point pure polling and interrupt disable–enable scheme. This implementation is simple and under the control of the OS and it does not require any load measurement or functionality at the NIC end. As a further study, we are currently in the process of implementing our proposed hybrid scheme that combines interrupt disable–enable and pure polling in Linux 2.6.

6. Conclusion

We developed analytical models to analyze the performance of Gigabit-network hosts when employing different interrupt-handling schemes that included ideal system, normal interruption, interrupt disabling and enabling, count-based coalescing, time-based coalescing, pure polling, and NAPI polling. The analytical models were verified and validated by simulation and by considering special cases. Our analysis provided equations to give insight into predicting the system performance and behavior when employing a certain interrupt-handling scheme. The performance was studied in terms of system throughput, CPU availability, and latency. We also studied using simulation the impact on performance of subjecting the host to bursty traffic that is self-similar with long-range dependence. In addition we studied the impact of selecting different values for scheme parameters.

It was concluded that the relative performance of interrupt-handling schemes under bursty traffic was similar to that of Poisson traffic. Also it was concluded that no particular interrupt-handling scheme gives the best performance under all load conditions. Selection of the most appropriate scheme to employ depends primarily on system performance requirements, most important performance metric, and present traffic load. It was shown by giving numerical examples that the scheme of disabling and enabling interrupts outperforms, in general, all other schemes in terms of throughput and latency. However, when it comes to CPU availability, pure polling is the most appropriate scheme to use. Based on these important observations, we proposed and discussed briefly the implementation of a hybrid scheme of interrupt disable–enable and pure polling. Such a hybrid scheme would be able to attain peak performance under low and heavy traffic loads. As a further study, we are in the process of implementing our proposed hybrid scheme in Linux 2.6. Details, results and comparisons of such implementation are to be published in the near future.

Acknowledgements

We acknowledge the support of King Fahd University of Petroleum and Minerals in completion of this work. This work has been funded under Project #FT-2005/17.

References

[1] R. Morris, E. Kohler, J. Jannotti, M. Kaashoek, The click modular router, ACM Transactions on Computer Systems 8 (3) (2000) 263–297.

[2] A. Foong, T. Huff, H. Hum, J. Patwardhan, G. Regnier, TCP Performance re-visited, in: IEEE Symposium on Performance of Systems and Software (2003) 70–79.

[3] K. Ramakrishnan, Performance Consideration in Designing Network Interfaces, IEEE Journal on Selected Areas in Communications 11 (2) (1993) 203–219.

[4] J. Chase, A. Gallatin, K. Yocum, End System Optimizations for High-Speed TCP, IEEE Communication Magazine 39 (4) (2001) 68–74.

[5] J. Mogul, K. Ramakrishnan, Eliminating Receive Livelock in an Interrupt-Driven Kernel, ACM Transanction Computer Systems 15 (3) (1997) 217–252.

[6] P. Druschel, Operating System Support for High-Speed Communication, Communications of the ACM 39 (9) (1996) 41–51.

[7] P. Druschel, G. Banga, Lazy Receive Processing (LRP): A network subsystem architecture for server systems, in: Proceedings Second USENIX Symposium on Operating Systems Design and Implementation (1996) 261–276.

[8] Alteon WebSystems Inc., Jumbo Frames, http://www.alteonwebsystems.com/products/white_papers/jumbo.htm.

[9] A. Gallatin, J. Chase, K. Yocum, Trapeze/IP: TCP/IP at Near-gigabit speeds, Annual USENIX Technical Conference, Monterey, Canada, 1999.

[10] C. Traw, J. Smith, Hardware/software Organization of a High Performance ATM Host Interface, IEEE JSAC 11 (2) (1993) 240–253.

[11] C. Traw, J. Smith, Giving Applications Access to Gb/s Networking, IEEE Network 7 (4) (1993) 44–52.

[12] P. Shivan, P. Wyckoff, D. Panda, EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing, in: Proceedings of SC2001, Denver, Colorado, USA, 2001.

[13] A. Indiresan, A. Mehra, K.G. Shin, Receive livelock elimination via intelligent interface backoff, TCL Technical Report, University of Michigan, 1998.

[14] R. Prasad, M. Jain, C. Dovrolis, Effects of interrupt coalescence on network measurements, in: Proceedings of Passive and Active Measurement (PAM) Workshop, France, 2004.

[15] M. Zec, M. Mikuc, M. Zagar, Estimating the impact of interrupt coalescing delays on steady state TCP throughput, in: Proceedings of the 10th SoftCOM, 2002.

[16] I. Kim, J. Moon, H. Y. Yeom, Timer-based interrupt mitigation for high performance packet processing, in: Proceedings of 5th International Conference on High-Performance Computing in the Asia-Pacific Region, Gold Coast, Australia, 2001.

[17] C. Dovrolis, B. Thayer, P. Ramanathan, HIP: Hybrid interrupt-polling for the network interface, ACM Operating Systems Reviews 35 (2001) 50–60.

[18] M. Aron, P. Druschel, Soft Timers: Efficient Microsecond Software Timer Support for Network Processing, ACM Transactions on Computer Systems 18 (3) (2000) 197–228.

[19] J. H. Salim, Beyond softnet, in: Proceedings of the 5th Annual Linux Showcase and Conference (2001) 165–172.

[20] K. Salah, K. El-Badawi, Modeling and analysis of interrupt disable–enable scheme, in: Proceedings of the IEEE 21st International Conference on Advanced Information Networking and Applications (AINA-07), Niagara Falls, Canada, May 21–23 (2007) 1000–1005.

[21] L. Deri, Improving passive packet capture: Beyond device polling, in: Proceedings of the 4th International System Administration and Network Engineering Conference, Amsterdam, 2004.

[22] O. Maquelin, G.R. Gao, H.J. Hum, K.G. Theobalk, X. Tian, Polling watchdog: Combining polling and interrupts for efficient message handling, in: Proceedings of the 23rd Annual International Symposium on Computer Architecture, Philadelphia, PA (1996) 178–188.

[23] K. Salah, K. El-Badawi, Evaluating system performance in gigabit networks, The 28th IEEE Local Computer Networks (LCN), Bonn/Königswinter, Germany, October 20–24 (2003) 498–505.

[24] K. Salah, K. El-Badawi, Performance evaluation of interrupt-driven kernels in gigabit networks, in: The IEEE Global Telecommunications Conference, 2003, GLOBECOM'03, December 1–5 (2003) 3953–3957.

[25] K. Salah, K. El-Badawi, Analysis and simulation of interrupt overhead impact on os throughput in high-speed networks, International Journal of Communication Systems, 18(5) Wiley (2005) 501–526.

[26] K. Salah, To Coalesce or not to coalesce, International Journal of Electronics and Communications (AEU) 61(4) (2007) 215–225.

[27] W. Feng, Is TCP an adequate protocol for high-performance computing needs? in: Proceedings of SC2000, Dallas, Texas, USA, 2000.

[28] A. Bianco, J.M. Finochietto, G. Galante, M. Mellia, F. Neri, Open-source pc-based software routers: A viable approach to high-performance packet switching in: Proceedings of QoS-IP 2005, Catania, Italy, February 2–4 (2005) 353–366.

[29] K. Kochetkov, Intel PRO/1000 T Desktop Adapter Review, http://www.digit-life.com/articles/intelpro1000t.

[30] 3Com Corporation, Gigabit Server Network Interface Cards 7100xx Family, http://www.costcentral.com/pdf/DS/3COMBC/DS3COMBC109285.PDF.

[31] R. Bhoedjang, T. Ruhl, H. Bal, User-level network interface protocols, IEEE Computer Magazine 31 (11) (1998) 53–60.

[32] A. Sinhra, S. Sarat, J. Shapiro, Network subsystems reloaded: A high-performance, defensible network subsystem, in: Proceedings

of the USENIX Technical Conference, Boston, MA (2004) 213–226.
[33] A. Dunkels, Design and Implementation of the lwIP TCP/IP Stack, February 2001, http://www.sics.se/~adam/lwip/doc/lwip.pdf.
[34] A. Rubini, J. Corbet, The Linux Device Drivers, O'Reilly, 2001.
[35] V. Guffens, Path of a Packet in the Linux Kernel, Submitted for publications, www.auto.ucl.ac.be/~guffens/doc/path_packet.pdf, 2003.
[36] M. McKusick, K. Bostic, M. Karels, J. Quarterman, The Design and Implementation of the 4.4BSD Unix Operating System, Addison Wesley, Reading, MA, 1996.
[37] J. Brustoloni P. Steenkiste, Effects of buffering semantics on I/O performance, in: Proceedings Second USENIX Symposium. on Operating Systems Design and Implementation (1996) 277–291.
[38] Z. Ditta, G. Parulkar, J. Cox, The APIC approach to high performance network interface design: protected DMA and other techniques, in: Proceeding of IEEE INFOCOM 1997, Kobe, Japan (1997) 179–187.
[39] H. Keng, J. Chu, Zero-copy TCP in solaris, in: Proceedings of the USENIX 1996 Annual Technical Conference, 1996.
[40] D. Clark, V. Jacobson, J. Romkey, H. Salwn, An analysis of TCP processing overhead, IEEE Communication Magazine 40 (5) (2002) 94–101.
[41] A. Brown, M. Seltzer, Operating system benchmarking in the wake of lmbench: A case study of the performance of NetBSD on the intel x86 architecture, in: Proceedings of the 1997 ACM SIGMETRICS international conference on measurement and modeling of computer systems, Seattle, WA (1997) 214–224.
[42] M. Karam, F. Tobagi, Analysis of delay and delay jitter of voice traffic in the internet, Computer Networks Magazine 40 (6) (2002) 711–726.
[43] W. Leland, M. Taqqu, W. Willinger, D. Wilson, On the self-similar nature of ethernet traffic, IEEE/ACM Transaction on Networking 2 (1) (1994) 1–15.
[44] V. Paxson, S. Floyd, Wide-area traffic: The failure of poisson modeling, IEEE/ACM Transactions on Networking 3 (3) (1995) 226–244.
[45] W. Willinger, M. Taqqu, R. Sherman, D. Wilson, Self-similarity through high-variability: Statistical Analysis of ethernet LAN traffic at the source level, in: Proceedings of ACM SIGCOMM, Cambridge, Massachusetts (1995) 100–113.
[46] O. Brun, J.M. Garcia, Analytical solution of finite capacity M/D/1 queues, Journal of Applied Probability 37 (4) (2000) 1092–1098.
[47] S. Alouf, P. Nain, D. Towsley, Inferring network characteristics via moment-based estimators, in: Proceeding of IEEE INFOCOM 2001, Anchorage, Alaska (2001) 1045–1054.
[48] A. Law, W. Kelton, Simulation Modeling and Analysis, 2nd ed., McGraw-Hill, 1991.
[49] J. White, An effective truncation heuristic for bias reduction in simulation output, Simulation Journal 69 (6) (1997) 323–334.
[50] L. McVoy and C. Staelin, lmbench: Portable tools for performance analysis, in: Proceedings of the 1996 USENIX Technical Conference, San Diego, CA (1996) 279–295.
[51] Ashford computer consulting service, GigaBit Ethernet to the Desktop – Client1 system benchmarks, 2004, http://www.accs.com/p_and_p/GigaBit/Client1.html.
[52] T. Dunigan, ORNL opteron evaluation – lmbench 2.0 Summary, http://www.csm.ornl.gov/~dunigan/opteron-1.5/lm.rpt.
[53] K. Park, W. Willinger, Self-Similar Network Traffic and Performance Evaluation, John Wiley & Sons, Inc, 2002.
[54] M. S. Taqqu, W. Willinger, R. Sherman, Proof of a fundamental result in self-similar traffic modeling ACM/SIGCOMM Computer Communication Review 24 (2) (1997) 5–23.
[55] K. Salah, K. Elbadawi, Throughput and delay analysis of interrupt-driver Kernels under Poisson and bursty traffic, International Journal of Computer Systems Science and Engineering 22 (1-2) (2007).
[56] S. Malik, U. Killat, How many traffic sources are enough? in: Proceedings of Performance Modeling and Evaluation of Heterogeneous Networks (HET-NETS), Ilkely, UK, 2004.
[57] K. C. Claffy, G. Miller, K. Thompson, The Nature of the beast: Recent traffic measurements from an internet backbone, in: Proceedings of INET 1998, Geneva, Switzerland, 1998.
[58] M. Crovella, L. Lipsky, Long-lasting transient conditions in simulations with heavy-tailed workloads, in: Proceedings of the 1997 Winter Simulation Conference, Atlanta, GA (1997) 1005–1012.
[59] C. Wang, D. Wolff, Efficient simulation of queues in heavy traffic, ACM Transactions on Modeling and Computer Simulation 13 (1) (2003) 62–81.

Khaled Salah is an associate professor of Computer Science. He received the B.S. degree in Computer Engineering with a minor in Computer Science from Iowa State University, USA, in 1990, the M.S. degree in Computer Systems Engineering from Illinois Institute of Technology, USA, in 1994, and the Ph.D. degree in Computer Science from the same institution in 2000. He has over ten years of industrial experience in embedded systems and software and firmware development of network protocol stacks and device drivers for ATM and Ethernet. He joined King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, in September 2000. Dr. Salah is currently with the department of Information and Computer Science, teaching graduate and undergraduate courses in the area of high-speed networks, operating systems, and computer and network security. His research interests are in the performance analysis and design of computer systems and networks using queueing theory and simulation.

Khalid El-Badawi is a Lecturer with the department of Information and Computer Science at King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia. He received his M.S. degree in Computer Science from the same department in April 2003. His research interests include performance evaluation, operating systems, and computer networks. Mr. El-Badawi received his B.S. degree in Mathematical and Computer Science from University of Khartoum, Sudan, in 1994.

Fahd Haidari is currently a Ph.D. candidate with the department of Information and Computer Science at King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia. He received his M.S. degree in Computer Science from the same department in May 2007. His research interests include algorithms and simulation. Mr. Haidari received his B.S. degree in Computer Science from University of Mousel, Iraq, in 1999.