# HIP: Hybrid Interrupt-Polling for the Network Interface

Constantinos Dovrolis[1] Brad Thayer[2] Parameswaran Ramanathan[3]
[1]Department of Computer and Information Sciences, University of Delaware
[2]Department of Computer Sciences, University of Wisconsin
[2]Department of Electrical and Computer Engineering, University of Wisconsin
dovrolis@cis.udel.edu, brad@cs.wisc.edu, parmesh@ece.wisc.edu

## Abstract

The standard way to notify the processor of a network event, such as the arrival or transmission of a packet, is through interrupts. Interrupts are more effective than polling, in terms of the per packet send/receive latency. Interrupts, however, incur a high overhead both during and after the interrupt handling, because modern superscalar processors use long pipelines, out-of-order and speculative execution, and multi-level memory systems, all of which tend to increase the interrupt overhead in terms of clock cycles. In this paper, we attempt to reduce the network interface overhead by introducing a hybrid scheme (HIP) that uses interrupts under low network load conditions and polling otherwise. Even though such hybrid schemes have been proposed in the past, the polling period in HIP is adjusted dynamically based on the rate of the arriving packet stream. In this way, the increase in the per packet latency, which occurs with polling, is quite low. This is quantified with trace-driven simulations, which also show that the per packet overhead with HIP is significantly reduced compared to the conventional interrupt-based mechanism. HIP would be beneficial for high bandwidth network interfacing in servers with a heavy WWW or streaming media workload.

## 1 Introduction

Personal computers and workstations will soon be used as videophones, televisions, and multiplayer game systems. These applications are network-intensive, and due to their multimedia nature they require high-throughput network interfaces. Many workstations today are connected to Fast Ethernet interfaces (100Mbps), while Gigabit Ethernet interfaces (1000Mbps) are already deployed in high-end Web servers. This dramatic bandwidth increase calls for optimizations in all key components of the network interface, including the network interface card (NIC), the protocol stacks, the operating system, the input-output unit, the memory system, and the processor. In general, the network interface has been traditionally viewed as just an I/O peripheral that causes unpredictable and infrequent events, and so not much optimization has been put into it [1, 2, 3, 4, 5].

We first have to differentiate between the concepts of latency and throughput, as they apply to the receive operation from the network interface. *Latency* is the time duration between a packet arrival at the NIC and its delivery to the application. The *throughput*, on the other hand, is the rate with which the application can read packets from the NIC. The throughput is the inverse of the receive overhead, where the latter accounts for all processing delays of the receive operation. The latency can be larger than the overhead if the received packets are queued at the NIC before they are received and processed by the application. There are systems and applications where reducing latency is the major issue. For example, in a loosely-coupled multiprocessor system the network latency is part of the computational delay and, therefore, it has to be minimized [6]. In contrast, in multimedia applications the audio and video streams that arrive from the network are queued for a playback delay of several tens or hundreds of milliseconds before they are delivered to the user [7]. Consequently, an increase of a few milliseconds in the receive latency would normally go unnoticed from that type of applications.

Current personal computers and workstations are notified of an arrival of a packet at the NIC through interrupts. This is because interrupts guarantee a minimum receive latency since the packets are not queued at the NIC for a duration more than the interrupt handling period. An interrupt, however, is an asynchronous event with high hardware and soft-

ware overhead. The hardware overhead is mainly due to the flushing of out-of-order and speculative execution state in the processor and due to the reduction in the locality of references in the instruction and data caches caused by the context switches that are necessary for interrupt processing [8]. The software overhead is due to the following reasons. When an interrupt occurs, the architecturally visible state must be saved, an appropriated interrupt handler must be dispatched, and upon completion of the handler, the system state must be restored. In addition, to handle nested interrupts, appropriate process states and priorities must be updated in each invocation of the interrupt handler, requiring time-consuming bookkeeping [9].

An alternative to interrupts is polling. In polling, the processor periodically initiates a read operation of a control NIC register. If one or more packets have arrived, they are moved to the main memory for further processing. Since several packets may be read in the same poll and since the code to perform a poll is usually much shorter than the interrupt processing code, the receive overhead is reduced. However, on the negative side, packets are not guaranteed to be present at each poll; the polls in which no packet is found in the NIC (unsuccessful polls) increase the overall overhead of the network interface. Additionally, the latency of the receive operation increases because packets are queued in the NIC until the polling event. Because of these two drawbacks, polling is not commonly used in general purpose systems. Polling is used however in systems that have a heavy network load, such as routers and bridges, firewalls, or file servers [5]. In such systems the probability of unsuccessful polls is small, and the receive latency remains quite low by using a high polling period, and/or specialized hardware (see Section 5).

In this paper, we propose an input network interface mechanism which combines the advantages of interrupts and polling, i.e., it has a receive overhead that is comparable to that of polling, and a receive latency that is not prohibitively larger than that of interrupts. This scheme is based on the following two observations about next generation multimedia applications.

1. Multimedia network traffic is stream-based with frequent packet arrivals that have some statistical predictability. By monitoring the packet interarrival times, the next packet arrival time can be roughly estimated. If this prediction is effective, we can in principle set the polling period to the estimated packet interarrival period, reducing both the unsuccessful polls as well as the receive latency. Notice that in all current implementations of polling-based network interfaces, the polling period is fixed and independent of the arriving packet stream interarrivals.

2. Multimedia-based applications can tolerate an increase in the receive latency, if this increase is much smaller than the playback delay of the application. The playback delay is the time duration between the receipt of a packet and the display of its data to the user. The playback delay is typically used for absorbing the random delays in the network. For interactive real-time applications, the playback delay is in the order of 100–200 milliseconds. Consequently, an increase in the receive latency of a few milliseconds should not be a problem for such applications. For other applications, such as Remote Procedure Calls (RPC) or Network File System (NFS) operations, the latency should not be larger than, for example, the average latency of a disk operation. Since a disk access can be several milliseconds, a network interface latency of the same magnitude should not cause noticeable performance degradations.

The basic idea of the proposed network interface scheme, called *Hybrid Interrupt Polling* (HIP), is to adaptively switch between the use of interrupts and polling based on the observed rate of packet arrivals. Specifically, if the packet arrivals are frequent and predictable, the receive mechanism operates in a polling mode and the interrupts are disabled. In this mode, the polling period is set based on the predicted packet interarrival times. However, to bound the receive latency, the polling period is not allowed to exceed a pre-determined limit (say, ten milliseconds). On the other hand, if the packet arrivals are infrequent, less predictable, or if the number of consecutive unsuccessful polls exceeds a threshold, the receive mechanism operates in the interrupt mode. In this mode, the polling operation is stopped and the interrupts are enabled.

HIP performs better than interrupts in terms of receive overhead and better than polling in terms of receive latency. Additionally, the receive latency is always upper bounded, as a safety precaution of the

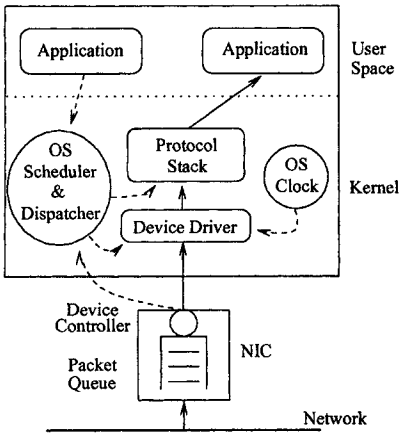



Figure 1: The basic components of the receive part of the network interface

mechanism. We demonstrate these two results by simulating HIP and other network interface mechanisms using traces from real local-area network traffic.

The rest of this paper is as follows. Section 2 presents a simple model for the network interface and it illustrates the main performance measures. Section 3 explains how polling is performed in HIP. Section 4 presents HIP and the involved algorithms in detail. Section 5 discusses some related works and their differences and similarities with HIP. Section 6 evaluates quantitatively HIP and other schemes through simulation. Finally, Section 7 concludes and identifies remaining open issues.

# 2 A Model of the Network Interface

The network interface is a complex system that consists of several hardware and software interacting units in both the computer and the NIC. We show the major components of the receive part of this system in Figure 1. As [4] points out, the receive and the transmit parts of the network interface are not completely symmetrical. In this paper we focus on the receive-part, where the interrupt overhead is more important.

We consider a typical multitasking workstation, where all the network interface functionality is performed by operating system processes running in the kernel address space, while the application processes run in the user address space. When a packet arrives at the NIC it gets temporarily stored in a local queue. In an interrupt-based system, the NIC device controller can either transfer the packet in the kernel memory buffers using a DMA engine and then interrupt the processor, or it can immediately interrupt the processor that will then copy the packet to the kernel memory. Currently, most network interfaces are DMA-capable. The activation of the hardware interrupt line preempts the running process, and the OS interrupt dispatcher is invoked to identify the nature of the interrupt and the corresponding device driver. After an additional context switch, the device driver completes the receive transaction with the device controller, and the packet processing continues with the network protocol processes (e.g., IP, UDP). Finally, the packet is copied again, this time from the kernel space to the user space, and the recipient application is notified.

Some of the above operations have a fixed overhead per packet, while some others have an overhead that is roughly proportional to the packet size in bytes. Following the model of [4], the final throughput $R$ that the application gets is

$$R = \frac{1}{V_P + L \times V_B}$$

where $L$ is the packet length in bytes, $V_P$ is the per-packet overhead, and $V_B$ is the per-byte overhead. The fixed overhead accounts for the hardware cost of the DMA transfer setup and of the I/O bus arbitration, as well as for the execution of the interrupt service routine, the device driver, the protocol stack, and for costs associated with memory management and context switches. The per-packet overhead is mainly due to data copying from the NIC to the kernel space (if there is no DMA support), and from the kernel space to the user process, as well as with error-checking processing, such as checksum calculations. In this paper we attempt to optimize the component of the fixed overhead per packet that is related with the interrupt overheads. As measured in [10], a hardware interrupt (such as the interrupt generated by the NIC) with a null interrupt handler introduces an overhead of about 4 $\mu$s in a 500Mhz Pentium III system running FreeBSD 2.2.6. This is an important overhead for a workstation that receive streams of several hundreds of Mbps from the network interface, and it is important to examine possible optimizations that can reduce it.

# 3 The Polling Process

Before describing HIP in detail, we address the important issue of how polling is performed. The constraints that HIP imposes are, first, that the polling period has to be dynamically adjustable, so that it can track the packet interarrivals, and second, that the polling period has to have as fine a time granularity as possible, so that the packet latency to be minimized.

For a multitasking operating system, one possible polling approach is to have a periodically scheduled kernel polling process. This approach, however, requires a context switch for each poll. Although the overhead of this context switch is smaller than the cost of an interrupt (no hardware overhead and no interrupt dispatching), it is still comparable to the interrupt handling overhead. Ideally, the polling operation should not introduce a context switch overhead.

The adopted solution in HIP is based on the operating system soft clock [9]. This clock causes a periodic interrupt that is used for time-slicing and other bookkeeping activities. Its period is the finest timeslice and system clock granularity that the operating system allows. In HIP, the polling period is always set to a multiple of the clock period. Specifically, the next polling event is scheduled using a counter $P$ in the clock handler. In every clock tick, the value of $P$ gets decremented. When it reaches zero, the network device driver is called to poll the NIC. Then, the value of $P$ is set to the number of clock events until the next poll, and the process repeats. This operation requires only a few additional instructions in the clock interrupt handler. The actual polling overhead is the cost of reading a status register on the NIC to check for a packet arrival, and if one is detected, to transfer the packet(s) from the NIC buffers to main memory (normally using DMA). Note that the overhead of the soft clock interrupt would be encountered in anyway, so the polling operation does not introduce an additional context switch.

Over the last decade or so, the OS clock period was commonly set to 10 milliseconds [9]. Although the polling period can be constrained to a multiple of this period, the time granularity of adjusting the polling period would be too coarse, and the maximum latency that polling could introduce would be excessive (several tens of milliseconds) for many applications. More recently however, some OS vendors have moved to a smaller clock interrupt period of 1 millisecond (e.g., Solaris 8). This reduction is well justified, given that the CPU performance has improved several orders of magnitude over the last ten years. We believe that most OS vendors will soon also switch to a 1-msec clock interrupt period, or even lower than that. As it will become clear in the next section, HIP becomes more practical and effective as the clock interrupt period decreases.

# 4 HIP

In this section we first present the important parameters in HIP, and then describe the algorithms to determine when to perform polling instead of interrupts, and how to compute the polling period.

- **Interrupt Overhead** $V_I$: The fixed overhead of receiving a packet from the network interface using interrupts.
- **Polling Overhead** $V_P$: The fixed overhead of receiving a packet from the network interface using polling. A rule of thumb is that the polling overhead is an order of magnitude less than the interrupt overhead [8, 6, 11].
- **Transfer Overhead** $V(B)$: The overhead of transferring B bytes from the network interface to main memory. This accounts for the variable part of the interrupt and polling overhead, i.e., if $B$ bytes have arrived, the total interrupt overhead is $V_I + V(B)$, and the total polling overhead is $V_P + V(B)$. $V(B)$ is practically zero for a DMA-capable network interface. For simplicity, we assume that this overhead is proportional to $B$ ($V(B) \propto B$).
- **Mode Switching Overhead** $V_S$: The overhead of switching from Interrupt-mode to Polling-mode, and vice versa. This overhead accounts for the network interface interrupt enabling and disabling operations. We assume that both operations have the same cost.
- **Minimum Polling Period** $T_P^{MIN}$: The minimum polling period that can be set. It is limited by the soft clock period.

- **Maximum Polling Period** $T_P^{MAX}$: The maximum polling period that can be set. It is limited by the maximum latency that can be tolerated by latency-sensitive applications such as NFS or RPC.
- **Polling Period** $T_P$: The scheduled polling period. The constraints are that $T_P^{MIN} \leq T_P \leq T_P^{MAX}$, and that $T_P$ must be an integral multiple of $T_P^{MIN}$.
- **Average Packet Interarrival** $\bar{I}$: A dynamic estimation of the average duration between packet arrivals.
- **Last Packet Interarrival** $D$: The duration between the last packet arrival and the one before that.
- **Packet Interarrival Variance** $\sigma_I^2$: A dynamic estimation of the packet interarrival variance. It is used as a rough statistical indication of the regularity of the packet interarrivals. The square root of $\sigma_I^2$ is the packet interarrival standard deviation $\sigma_I$.
- **Consecutive Unsuccessful Polls** $P_U$: The number of consecutive unsuccessful polls (polls where there was nothing received at the network interface). An unsuccessful poll increases the overhead without producing any useful work. A large value of $P_U$ indicates that the polling period used was poorly set, or that the stream of received packets has ended.
- **Maximum Consecutive Unsuccessful Polls** $P_U^{MAX}$: The maximum allowed value of $P_U$. After this number of unsuccessful polls, HIP switches back to Interrupt-mode. As a first guess, $P_U$ can be equal to the ratio $\frac{V_I}{V_P}$, since this many unsuccessful polls are equivalent, in terms of overhead, to an interrupt.

The average packet interarrival $\bar{I}$ is estimated after each packet arrival using a simple exponential weighted average:

$$\bar{I} = \alpha\bar{I} + (1-\alpha)D \qquad (0 \leq \alpha \leq 1) \qquad (1)$$

where $\alpha$ is the **average interarrival weight factor**. $\alpha$ controls the importance that is given to the last interarrival relative to the past history of interarrivals, as that is accumulated in $\bar{I}$.

The packet interarrival variance $\sigma_I^2$ is estimated as

$$\sigma_I^2 = \beta\sigma_I^2 + (1-\beta)(\bar{I}-D)^2 \qquad (0 \leq \beta \leq 1) \qquad (2)$$

where $\beta$ is the **interarrival variance weight factor**. Again, $\beta$ controls the memory of the predictor.

HIP switches between the interrupt and the polling modes based on the observed statistics of packet arrivals. It switches to the polling mode if the following two conditions are satisfied:

1. Packets arrive regularly enough
2. Packets arrive frequently enough

The first condition attempts to avoid the risk of many unsuccessful polls when the packet stream is too bursty. The second condition avoids polling when the arrival rate is small compared to the minimum possible polling rate. Specifically, HIP switches to polling mode if

$$\frac{\sigma_I}{\bar{I}} < \gamma \quad \text{and} \quad \bar{I} < T_P^{MAX} \qquad (3)$$

The ratio $\sigma_I/\bar{I}$ is the coefficient of variation of the packet interarrivals and it is an indication of the predictability of the packet interarrivals. The parameter $\gamma$ is the **prediction threshold**. Reducing $\gamma$ requires less predictable packet interarrivals before switching to polling mode.

HIP switches to interrupt mode when the complement of the above conditions occur, or when the number of consecutive unsuccessful polls has reached its maximum allowed value. This latter condition ensures that when a stream has finished, or when the interarrivals prediction has failed, HIP will switch back to interrupts. Specifically, HIP switches to the interrupt mode if

$$\frac{\sigma_I}{\bar{I}} \geq \gamma \quad \text{or} \quad \bar{I} \geq T_P^{MAX} \quad \text{or} \quad P_U = P_U^{MAX} \qquad (4)$$

When $P_U$ reaches its maximum value $P_U^{MAX}$, it is re-initialized to zero, and $\bar{I}$ and $\sigma_I$ get their default values.

Ideally, when in polling mode, the polling period should be set to exactly match the next packet arrival instant. This, of course, is impossible, first because we cannot exactly predict future arrivals, and second, because the polling period has to be a multiple of $T_P^{MIN}$, and $T_P^{MIN} \leq T_P \leq T_P^{MAX}$. In HIP, the polling period is set based on the following algorithm.

$T_P = \bar{I} + \phi\sigma_I$;
**If** $(T_P < T_P^{MIN})$
$T_P = T_P^{MIN}$;
**Else**
**If** $(T_P > T_P^{MAX})$
$T_P = T_P^{MAX}$;

In this algorithm, $\phi$ is the **slack parameter**, and it specifies how 'pessimistic' the estimation of the average packet interarrival is, given a certain interarrival standard deviation. The computations involved in the above algorithm are quite simple to consist a significant overhead. If, however, more efficient calculations are required, the variance estimator can be replaced with the simpler to calculate mean prediction error, as it was done in [12] for the round-trip delay estimation in TCP implementations.

It is clear that the performance of HIP can depend strongly on the the parameters $\alpha$, $\beta$, $\gamma$, $\phi$, and $P_U^{MAX}$. If they are chosen too conservatively, polling will seldom be used and the results will be similar to those of exclusive interrupts. At the other extreme of parameter selection, polling will be used too often and the overhead of unsuccessful polls will dominate, or the polling period will be too large, and the receive latencies will be increased without significant overhead reduction. In Section 6 we present some experimental results on the effect of these parameters.

A concern about the proposed algorithm is that the fractal (or self-similar) nature of data traffic [13] may deteriorate the predictability of the packet interarrivals. It is exactly because of this concern that we used real local-area network traffic traces in evaluating HIP. We observed through experimentation that frequently the simple coefficient-of-variation metric fails to accurately track the predictability of the arriving stream. This causes unsuccessful polls, or increased receive latencies. The constraint on the maximum number of unsuccessful polls serves as a protection mechanism, forcing HIP back to interrupt mode when the polling period is badly set over a sequence of packets.

# 5 Related Work

Several others have examined the trade-offs between interrupts and polling and have suggested hybrid schemes. An interesting mechanism is the *Clocked Interrupts* [14, 15]. In that scheme, a fine-granularity timer determines the rate of interrupt events. A typical frequency range for this timer is 500Hz to 4KHz. Upon the expiration of the timer the network interface is polled for the arrival of one or more packets. If a packet has arrived, the interrupt service routine for the network interface is called to move the packets into the main memory. The receive overhead is reduced because the interrupt is not caused by the asynchronous event of a packet arrival, but by the timer expiration. This makes clocked interrupts quite similar with polling. Note however that this mechanism requires an additional fine-granularity high-frequency hardware timer. A second difference is that the clocked interrupts timer has a constant period that does not depend on the arriving network traffic, while HIP determines the polling period based on the measured packet interarrivals. This adaptation allows HIP to control the overhead-latency trade-off between interrupts and polling depending on the network interface load.

Another relevant scheme was presented in [5]. The authors focused on an effect called *Receive Livelock*, that occurs under heavy network load conditions in workstations that are configured as routers, firewalls, file servers, or promiscuous network monitors. Specifically, if packets arrive very frequently, the system can practically spend all the CPU time at the interrupt handling routine, while lower priority processes (including user processes) do not get the chance to process the packets to completion /citeqie:01. This leads to extensive packet drops at intermediate buffering points, and thus, to a livelock situation. The authors of [5] implemented a mechanism where interrupts are only used at low network load conditions, while in high loads the interrupts are disabled and a polling thread is scheduled for reading the network interface. Every time a poll is executed, a certain *packet quota* is specified, i.e., the maximum number of packets that can be read in that poll. The quota is used for fairness purposes, when other tasks must also be permitted to make progress, so that to avoid the above livelock condition. If at the end of the polling some packets still remain at the NIC, the polling thread is executed again after a few milliseconds. Otherwise, the system switches back to interrupts. In principle, this mechanism is very similar to HIP. However, in HIP, the polling period is adjusted based on the observed packet interarrivals.

Aron and Druschel designed a *soft timers* OS facility that allows efficient scheduling of software events at microsecond granularity [10]. The basic idea behind soft timers is to take advantage of certain states in the execution of a system where an event handler can be invoked at low cost. Such states include the entry points of various kernel handlers, such as system calls and exception handlers. A drawback of soft timers is that they can only schedule events probabilistically. [16] has demonstrated, however, that under practical workloads it is possible to schedule events at intervals down to a few tens of microseconds, with rare delays up to a few hundred of microseconds. The soft timers facility can be combined with HIP in the following way: instead of scheduling the HIP polling thread at the granularity of the clock interrupt period, a soft timer can be scheduled to perform it instead. The execution frequency of the polling thread will then be adjusted by HIP, as described in the previous section.

In the context of message-passing parallel systems, a scheme that is similar to HIP is the *Polling Watchdog* [11]. In parallel systems the network latencies are part of the computational delays, and so it is critical to minimize them. The Polling Watchdog is a hardware extension at the NIC that limits the generation of interrupts to the cases where explicit polling fails to handle the packets quickly. The basic idea is that when a packet arrives at the NIC, a timer starts counting. If the packet is not removed from the NIC through polling within a given amount of time (the watchdog timeout period $T_{wdog}$), the watchdog interrupts the CPU. $T_{wdog}$ is set to around 50 $\mu$s, in order to strictly limit the maximum latency. In the EARTH-MANNA multiprocessor system[1] on which this scheme has been implemented, the cost of an interrupt is 4.5 $\mu$s, and the cost of a poll is 400 ns. The major difference between the Polling Watchdog and HIP is that in the former the CPU always polls the NIC in every context switch (on the average every 50 $\mu$s) and interrupts are used only to bound the maximum receive latency, while HIP switches between interrupts and variable-period polling, depending on the observed arriving network stream.

Another scheme that combines interrupts with polling in the context of message-passing parallel systems is used in the CM-5 [8]. For that system the time per poll is 1.6 $\mu$s (0.6 to poll the interface and 1.0 to check the type of message and move it to memory), the interrupt overhead is 19 $\mu$s, and the cost of enabling or disabling interrupts is about 4.3 $\mu$s. The basic idea of the CM-5 scheme is to keep polling for incoming packets while servicing an interrupt for a packet arrival. The interrupt service routine exits only if there are no more packets waiting at the NIC. This is also called *batched interrupts* and it is a common practice in current network interface drivers [5]. The virtue of this scheme is that if the average packet interarrival is much smaller than the interrupt overhead, then multiple packets can be serviced in a single interrupt. In multiprocessor systems, the network bandwidth is very high and, therefore, the average packet interarrival times are often in the range of a few microseconds. Hence, batched interrupts are quite effective. For more conventional workstations and networks, not many packets can arrive within a single interrupt service period. For example, for a 10Mbps Ethernet the minimum time between packet arrivals is about 51.2$\mu$s, and so in a 5 $\mu$s interrupt handling period at most two packets can be captured. HIP does both batched interrupts and batched polls. In addition, it attempts to reduce the receive overhead when packets arrive every several hundreds or thousands of microseconds.

# 6 Evaluation

We wrote an event-driven simulator to quantitatively evaluate the overhead and latency that results from HIP, as well as from schemes that exclusively use interrupts or polling. The main goal of this study is to examine the relative overhead and latency that results from these different interface schemes, rather than to predict system-specific absolute performance measures. Additionally, we are interested to examine the behavior of the packet interarrival predictors using real network traces, and to find appropriate values for the HIP parameters.

The events that the simulator captures (and their symbols) are:

- Packet arrival ($A$)
- End of interrupt period ($I$)
- Start of polling period ($P_s$)
- End of polling period ($P_f$)

[1] based on Intel i860 XP processors.

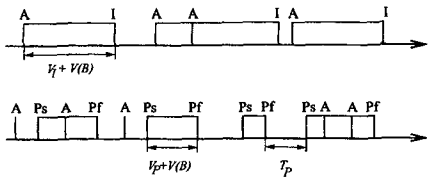



Figure 2: Two timelines with packet arrivals (A), interrupt period ends(I), polling period starts ($P_s$), and polling period ends ($P_f$).

The packet arrivals are determined from real LAN traffic traces (described later). In the interrupt mode, an interrupt period is initiated with a packet arrival, unless the packet arrival occurs inside an interrupt service period. In the latter case, we receive all the packets within the same interrupt (batched interrupts). The length of the interrupt period is equal to the interrupt overhead $V_I + V(B)$. The start of a polling event is initiated based on the polling period $T_P$, and its duration is determined based on the polling overhead $V_P + V(B)$. Batched polls are also supported. After every packet arrival, the predictions $\bar{I}$ and $\sigma_I^2$ are updated and the next polling period is determined (if in Polling-mode). Statistics regarding the average receive overhead per packet and the average latency per packet are reported at the end of the simulation run. The timelines in Figure 2 illustrate the above events, and the interrupt and polling overheads.

The characteristics of the network traffic traces that we use in this paper appear in Table 1. All traces were collected using the 'snoop' packet filter, observing the packets that are destined to a Sun Ultra-1 workstation from a 10Mbps Ethernet segment. It is important to note that on this specific Ethernet segment there are several dozens of connected workstations, and our workstation frequently received ARP, NFS, DNS, and other kinds of packets that the user does not normally see. The traces are classified in three classes, depending on the average bit-rate that arrives at our host (high, medium, and low). The first class (T-high) consists of two traces that were generated while long FTP sessions with nearby workstations were ongoing. The achieved average throughput in these sessions is close to 3 Mbps, so these streams may represent the kind of high-bandwidth multimedia streams that computers will be receiving in the near future. The second class (T-medium) consists of two traces that were generated while Web-browsing sessions were performed and correspond to an average incoming throughput of several tens of kbps. The third class (T-low) of traces were generated while the workstation was otherwise idle. In these traces the average incoming throughput is only a few kbps, mainly due to ARP, NFS, and DNS traffic.

The parameter values that we used in the simulations are shown in Table 2. The fixed parameters ($V_I$, $V_P$, $V(B)$, $V_S$) were selected based on reported values in the literature [16, 5]. Note these particular numbers can vary widely from system to system, depending on the I/O system design and implementation. The fact though that the polling overhead is almost an order of magnitude smaller than the interrupt overhead seems to be rather generally true. The per-byte overhead $V(B)$ was assumed to be quite low (500ns for 1500 byte packets), as would be the case with a DMA-capable NIC. The minimum polling period was set to 1ms, assuming a soft clock period of the same rate, while the maximum tolerable packet latency was set to 10ms. The HIP-related parameters ($\alpha$, $\beta$, $\gamma$, $\phi$, $P_U^{MAX}$) were tuned through extensive simulations with a wide range of traffic traces (not only the reported ones here).

The comparison of HIP with the interrupts-only and the polling-only network interface schemes is shown in Table 3. Note that, the simulated polling scheme uses the same polling-period adaptation algorithm as HIP, without ever switching to interrupt mode though. Conventional polling schemes, on the other hand, either busy-wait on the NIC, or use a constant polling period. The average overhead per packet and the average latency per packet are shown for each of these three network interface schemes. Several other performance measures of HIP, such as the average number of packets that are captured through interrupts and through polling, are also shown.

As expected, the interrupt overhead remains almost constant independent of the incoming traffic's rate. On the contrary, the polling overhead keeps increasing as the incoming traffic's rate decreases, due to many unsuccessful polls. HIP achieves the low overhead of polling at high input rates, while for lower input rates it consistently introduces a lower overhead than the interrupt-based and polling-based schemes. For the high rate traces, the average HIP overhead per packet is almost six times smaller than the interrupts-only overhead. On the negative side,

| | T-High-1 | T-High-2 | T-Med-1 | T-Med-2 | T-Low-1 | T-Low-2 |
|---|---|---|---|---|---|---|
| Duration (sec) | 11.5 | 16.7 | 255.0 | 341.9 | 1118.7 | 1302.3 |
| Packets Received | 4502 | 5407 | 3736 | 3997 | 3779 | 4044 |
| Average Interarrivals (ms) | 2.5 | 3.1 | 68.2 | 85.6 | 296.0 | 322.0 |
| Avg. Packet Length (bytes) | 904 | 949 | 421 | 561 | 90 | 101 |

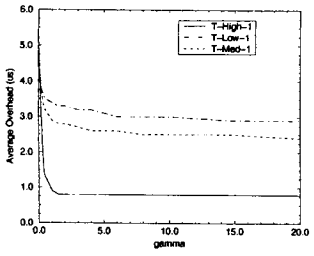
Table 1: Characteristics of the network traces that were used in the simulations.

| $V_I$ | $V_P$ | $V(B)$ | $V_S$ | $T_P^{MIN}$ | $T_P^{MAX}$ | $\alpha$ | $\beta$ | $\gamma$ | $\phi$ | $P_U^{MAX}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $5\mu$s | $0.5\mu$s | $0.5\mu$s B=1500 | $1\mu$s | 1ms | 10ms | 0.3 | 0.3 | 2 | 2.5 | 10 |

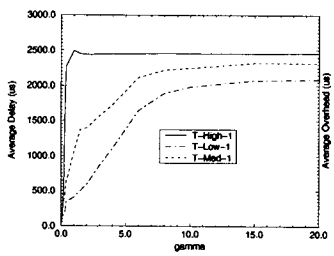
Table 2: Parameter values used in simulations.

| | T-High-1 | T-High-2 | T-Med-1 | T-Med-2 | T-Low-1 | T-Low-2 |
|---|---|---|---|---|---|---|
| Interrupts: Avg-Overhead ($\mu$s) | 5.3 | 5.3 | 5.1 | 5.2 | 5.0 | 5.0 |
| Polling: Avg-Overhead ($\mu$s) | 0.9 | 0.9 | 6.3 | 5.4 | 18.4 | 20.3 |
| HIP: Avg-Overhead ($\mu$s) | 0.9 | 0.9 | 2.6 | 4.1 | 3.3 | 3.4 |
| Interrupts: Avg-Delay ($\mu$s) | 5.3 | 5.3 | 5.1 | 5.2 | 5.0 | 5.0 |
| Polling: Avg-Delay ($\mu$s) | 1698 | 1640 | 3455 | 4598 | 3919 | 3975 |
| HIP: Avg-Delay ($\mu$s) | 1557 | 1547 | 1074 | 1321 | 393 | 367 |
| HIP: Total # of Interrupts | 103 | 111 | 1750 | 2792 | 2329 | 2531 |
| HIP: Avg. Packets per Interrupt | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| HIP: Total # of Polls | 4415 | 5458 | 2429 | 2726 | 1404 | 1504 |
| HIP: Total # of Unsucc. Polls | 1813 | 2224 | 1511 | 2093 | 825 | 869 |
| HIP: Avg. Packets per Poll | 1.0 | 1.0 | 0.8 | 0.4 | 1.0 | 1.0 |

Table 3: Comparison of HIP with the schemes that are based exclusively on interrupts and polling.

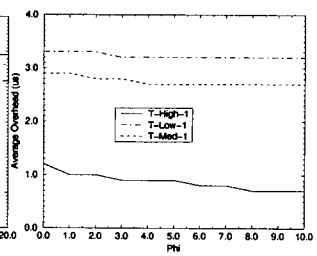



(a) The effect of $\gamma$ on the average overhead per packet

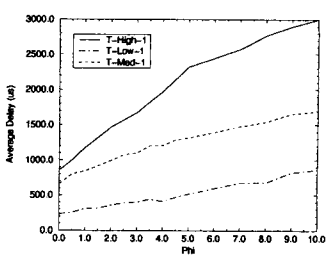



(b) The effect of $\gamma$ on the average delay per packet

Figure 3: The average per packet overhead and delay, as a function of the parameter $\gamma$.




(a) The effect of $\phi$ on the average overhead per packet




(b) The effect of $\phi$ on the average delay per packet

Figure 4: The average per packet overhead and delay, as a function of the parameter $\phi$.

HIP introduces a considerably larger average per packet latency (between $400\mu s$ and 1.5ms) compared to the interrupt-based scheme. Note that the average delay using polling-only can be much larger than that of HIP, especially for low-rate streams. The last five lines of Table 3 show the mix between interrupts and polls for HIP. The high incoming rate traces are captured mainly through polling, while the low rate traces lead to most interrupts and many unsuccessful polls. In general, the large number of unsuccessful polls relatively to the total number of polls can be attributed to interarrival prediction failures because of the traffic burstiness.

Figure 3 shows the effect of the parameter $\gamma$ on the average packet overhead and latency. It is clear that as $\gamma$ increases, we expect less regularity in the incoming stream and polling is selected more frequently. This results in decreasing overhead and increasing latency. A value of $\gamma$ between 2 and 3 seems to achieve a good trade-off between average overhead and latency.

Figure 4 shows the effect of the parameter $\phi$ on the average packet overhead and latency. As $\phi$ increases, the polling period also increases. For high-rate incoming streams this reduces the average overhead, since more packets are received in each poll and less unsuccessful polls are performed. This effect is less important in low rate streams. On the other hand, larger values of $\phi$ can increase the average latency significantly. We found that a value of $\phi$ between 2 and 3 is a good middle point.

# 7 Conclusions

We proposed a hybrid interrupt-polling (HIP) scheme for the network interface. HIP exploits the trade-off between decreased receive-overhead and increased receive-latency. The careful selection of the related parameters allows the system designer to set this trade-off to the appropriate operating point. Through trace-driven simulations, we showed that HIP is more effective when the network workload consists of high-bandwidth streams, such as multimedia traffic. In this paper we focused on the underlying ideas and the general architecture, rather than on implementation specific problems. We expect, though, that several challenges can arise at that phase. Identifying and addressing those issues is something that we plan to do in the future.

# References

[1] D. Banks and M.Prudence, "A high-performance network architecture for a PA-RISC workstation," *IEEE Journal on Selected Areas in Communications*, vol. 11, pp. 191–202, Feb. 1993.

[2] C. B. S. Traw and J.M.Smith, "Hardware-software organization of a high-performance ATM host interface," *IEEE Journal on Selected Areas in Communications*, vol. 11, pp. 240–253, Feb. 1993.

[3] B. Davie, "The architecture and implementation of a high-speed host interface," *IEEE Journal*

*on Selected Areas in Communications*, vol. 11, pp. 228–239, Feb. 1993.

[4] K. K. Ramakrishnan, "Performance considerations in designing network interfaces," *IEEE Journal on Selected Areas in Communications*, vol. 11, pp. 203–219, Feb. 1993.

[5] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Transactions on Computer Systems*, vol. 15, pp. 217–252, Aug. 1997.

[6] S. S. Mukherjee and M. D. Hill, "A survey of user-level network interfaces for system area networks," Tech. Rep. TR 1340, Computer Sciences Department, University of Wisconsin- Madison, Feb. 1997.

[7] H. Schulzrinne, S.Casner, R.Frederick, and V.Jacobson, *RTP: A Transport Protocol for Real-Time Applications*, Jan. 1996. RFC 1889.

[8] D. A. Patterson and J. L. Hennessy, *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann, 1996.

[9] U. Vahalia, *UNIX Internals, the new frontiers.* Prentice Hall, 1996.

[10] M. Aron and P. Druschel, "Soft Timers: Efficient Microsecond Software Timer Support for Network Processing," *ACM Transactions on Computer Systems*, vol. 18, pp. 197–228, Aug. 2000.

[11] O. Maquelin, G. R. Gao, H. H. J. Hum, K. B. Theobald, and X. Tian, "Polling watchdog: Combining polling and interrupts for efficient message handling," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 179–188, 1996.

[12] V. Jacobson, "Congestion Avoidance and Control," in *Proceedings of ACM SIGCOMM*, pp. 314–329, Sept. 1988.

[13] W. Willinger, M.S.Taqqu, R.Sherman, and D.V.Wilson, "Self-Similarity Through High-Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level," in *Proceedings of ACM SIGCOMM*, pp. 100–113, Sept. 1995.

[14] J. D. Chung, C. B. S. Traw, and J. M. Smith, "Event-signaling within higher performance network subsystems," in *Proceedings, High Performance Communications Subsystems*, pp. 220–225, Aug. 1995.

[15] J. M. Smith and C. B. S. Traw, "Operating systems support for end-to-end Gbps networking," *IEEE Network*, vol. 7, pp. 44–52, Feb. 1993.

[16] M. Aron and P. Druschel, "Soft Timers: Efficient Microsecond Software Timer Support for Network Processing," in *Proceedings of ACM SOSP*, Dec. 1999.