JCSE Journal of Computing Science and Engineering

Regular Paper
Journal of Computing Science and Engineering,
Vol. 9, No. 2, June 2015, pp. 108-117

Priority-Based Network Interrupt Scheduling for Predictable Real-Time Support

Minsub Lee, Hyosu Kim, and Insik Shin*
School of Computing, Korea Advanced Institute of Science and Technology, Daejeon, Korea
mslee@cps.kaist.ac.kr, hskim@cps.kaist.ac.kr, insik.shin@cps.kaist.ac.kr

Abstract
Interrupt handling is generally separated from process scheduling. This can lead to a scheduling anomaly and priority inversion. The processor can interrupt a higher priority process that is currently executing, in order to handle a network packet reception interruption on behalf of its intended lower priority receiver process. We propose a new network interrupt handling scheme that combines interrupt handling with process scheduling and the priority of the process. The proposed scheme employs techniques to identify the intended receiver process of an incoming packet at an earlier phase. We implement a prototype system of the proposed scheme on Linux 2.6, and our experiment results show that the prototype system supports the predictable real-time behavior of higher priority processes even when excessive traffic is sent to lower priority processes.

Category: Embedded computing; Smart and intelligent computing

Keywords: Real-time scheduling; Interrupt handling; Network I/O management; Process-aware resource scheduling

I. INTRODUCTION

Interrupt handling is critical to predictable real-time services. Hardware interrupts make it possible for external devices to notify asynchronous events to the processor efficiently. When the processor receives an interrupt signal, it invokes an interrupt handler, blocking the currently executing process. In most operating systems, interrupt handling is independent of process scheduling, even though many interrupts are generated and processed on behalf of user processes. This could lead to scheduling anomalies, making it more complicated to achieve the predictable behavior of real-time processes.

Most operating systems prioritize the processing of interrupts. This can lead to *priority inversion* [1]. As an example, let us consider a network interface card (NIC) signaling an interrupt to notify an incoming network packet. In response to this network interrupt, the processor triggers an interrupt handler. This interrupt handler preempts the process that is currently executing and which is the intended receiver of the packet. This can block the execution of a higher priority task until the processing of the incoming packet is complete for its lower priority receiver process.

In this paper, we aim to address such a priority inversion problem when handling interrupts for incoming network packets. We introduce an interrupt handling scheme, based on *network interrupt scheduling with process priority* (NISP), which combines network interrupt handling and priority-based process scheduling. The proposed NISP scheme identifies the intended receiver process of an incoming packet at an earlier phase, and schedules a network interrupt handler according to the receiver process' priority. This allows network interrupt handling to

Open Access http://dx.doi.org/10.5626/JCSE.2015.9.2.108 http://jcse.kiise.org

Received 21 May 2015, **Accepted** 01 Jun 2015
*Corresponding Author

pISSN: 1976-4677 eISSN: 2093-8020

avoid the priority inversion problem.

Experiments show that a prototype system based on NISP supports the predictable behavior of higher priority processes even when faced with excessive traffic on lower priority processes. In comparison, a conventional Linux system exhibits the unstable behavior of higher priority processes under high network loads on lower priority processes.

This paper presents the design and implementation of process priority-based network interrupt scheduling. This work makes the following contributions:

- It shows that the NISP scheme provides a more suitable environment for real-time tasks, such as periodic and sporadic tasks, in terms of response time, deadline miss ratio, and jitter (the time variation of two consecutive outputs) with small network throughput degradation.
- It demonstrates that our NISP scheme works on existing Linux 2.6 with minimal modifications.

The remainder of this paper is organized as follows: Section II describes the interrupt handling mechanism on Linux. Section III introduces the proposed NISP scheme. Section IV presents details of implementing the NISP, which is evaluated experimentally in Section V. Related work is discussed in Section VI. Finally, Section VII contains conclusions and recommendations for future work.

II. INTERRUPT HANDLING ON LINUX

Hardware devices issue an interrupt signal to the processor to indicate the need for attention. For example, when a network card receives an incoming packet off the network, it signals an interrupt to alert the kernel to its availability. When the processor detects the signal, it preempts its current execution in order to handle the interrupt immediately. The kernel runs *interrupt handler* or an *interrupt service routine*. Each network card has an associated interrupt handler, which is part of its device driver. In general, the interrupt handler of a network device has a large amount of work to perform. In addition to the network device acknowledging receipt of the interrupt, the interrupt handler needs to copy networking packets from the network device into memory, process them, and push the packets down to the appropriate protocol stack or application.

Interrupt handling is generally associated with two seemingly conflicting goals. It is imperative that an interrupt handler processes interrupt requests immediately in order to optimize hardware performance. The interrupt handler disables interrupts in order to avoid potential race conditions and data corruption due to multiple interrupt handling. It is important that the interrupt handler finishes up quickly, to resume execution of the interrupted code as soon as possible and not to keep interrupts blocked for long. Because of these conflicting goals, the processing of interrupts is split into two parts. The *top half* is the interrupt handler that executes immediately upon receipt of the interrupt. It performs time-critical work, such as acknowledging receipt of the interrupt or resetting the hardware (packet moving). The top half executes with its corresponding interrupts disabled. In order to facilitate small and fast interrupt handlers, the *bottom half* is used to defer as much of the interrupt processing work as possible away from the top half. The bottom half handles all of the remaining work that has not been carried out by the top half. This enables other interrupts. This dichotomy in interrupt handling reduces the time to disable interrupts, as well as the time to preempt the code that is currently being executed.

For example, in response to an interrupt from a network card, the kernel executes the network card's registered device driver. The device driver uses the top half interrupt handler to perform time-sensitive, hardware specific operations. It acknowledges the network card, copies new networking packets into the main memory, resets a device register, and makes the network card ready for more packets. The interrupt handler then defers the rest of the work to the bottom half. It marks the bottom half, instructing the kernel to run it later, and subsequently it exits.

Most work for packet processing takes place in the bottom-half interrupt handler. It performs network protocol stack processing, which corresponds to the network and transport layers. After addressing protocol-specific issues and performing demuxing, the bottom half places the packet into the buffer in the appropriate socket structure.

III. NETWORK INTERRUPT HANDLING

This section describes the design of our proposed interrupt handling scheme, based on *network interrupt scheduling with process priority* (NISP). The NISP scheme aims to avoid the priority inversion problem involved in network interrupt handling. Suppose a higher priority (HP) process is currently executing, while a lower priority (LP) process is waiting for incoming packets. When a new incoming packet arrives at NIC destined for the LP process, it issues an interrupt. The processor can then preempt the HP process immediately, in order to execute the top half interrupt handler, on behalf of the LP process. After performing time-critical operations, the handler defers the remaining work for packet processing to a later time. When the top half handler finishes, the operating system resumes the execution of the preempted HP process. When a bottom half handler starts its operation on the incoming packet on behalf of the LP process, the HP process can be preempted again. Carrying out the bottom half operations, the bottom half handler finishes, and the HP process can resume its execution.

Here, the HP process could be blocked by the top half and the bottom half interrupt handlers on behalf of the LP

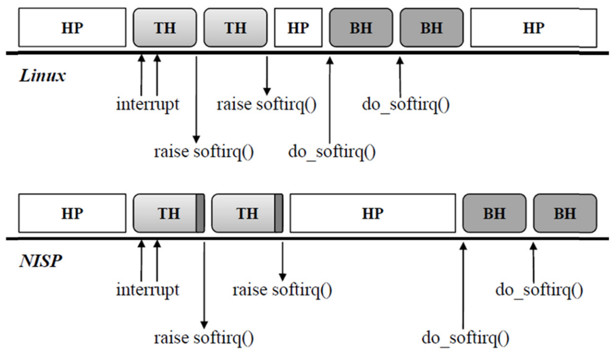

Fig. 1. Priority inversion with interrupt handling in the original version of Linux and in the proposed NISP scheme. HP: higher priority, TH: top half interrupt handler, BH: bottom half interrupt handler, NISP: network interrupt scheduling with process priority.

process, incurring priority inversion (Fig. 1(a)).

An intuitive solution to this priority inversion problem would be to disable low priority interrupts during the execution of a HP process so that the HP process can execute free from the interference of such interrupt handling. However, interrupts and processes have separate priority spaces in most operating systems, including Linux. That is, interrupt priorities characterize the relative importance between various interrupts, while process priorities specify that importance between different processes. However, Linux cannot determine which interrupt is more important than which process.

The NISP scheme proposes a systematic approach to address the priority inversion problem that the bottom half network interrupt handling can yield. In terms of network interrupts for incoming packets, the NISP scheme executes the top half handler in the same way as it takes care of the bottom half differently from the original Linux; it could defer the execution of the bottom half even further, when a priority inversion problem takes place. The key idea of NISP is that some incoming packets have their intended receiver processes. Under NISP, such intended processes (and their priorities) are identified through early demux in the top half phase. More specifically, we can find out a destination port number of an incoming packet through early demux, and we map the port number to the corresponding receiver process. NISP then uses such process priorities in the bottom half scheduling, in particular, to avoid the priority inversion problem. Therefore, NISP consists of three major components: early demux in the top half, port number-to-priority mapping, and the bottom half scheduling, which are described in Section III-A, III-B, and III-C, respectively. Section III-D discusses protocol issues involved in early demux.

A. Early Demuxing

One of the essential operations of NISP is to determine which process is connected to an incoming network

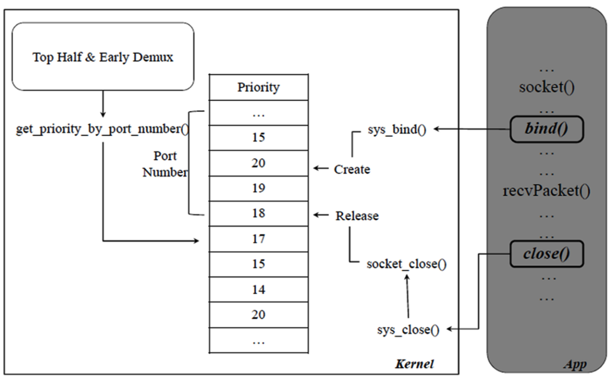

Fig. 2. Mapping of the port number to the process priority.

packet, and associate the corresponding process' priority with the packet such that its packet processing can be handled according to the process priority. In general, such a connection is revealed out at the last phase of the packet processing. Hence, early demuxing techniques [2, 3] are used to acquire such connection information earlier than the packet processing.

Early demux techniques can be implemented in hardware or by software. Network interface cards can come with an additional special co-processor that can extract some protocol information (e.g., a destination port number) from incoming packets, and pass such information to operating systems. As this hardware-based approach imposes an extra cost with the additional co-processor, we consider a software early demux in the NISP scheme.

Under NISP, the top half handler fetches a network packet from the NIC's buffer to the main memory, and it extracts a destination port number out of the packet. This can be easily achieved with two lookup operations to the packet header. The handler first looks up a type of the packet. If the packet is a Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) packet, its destination port number is stored in a fixed location in the packet header. The handler is then able to find it easily according to its packet header length.

B. Port Number to Process Priority Mapping

We are now subject to how to map a destination port number to a corresponding process (and its priority). In Linux 2.6, the operating system scans all available sockets in order to find a socket corresponding to a given destination port number, and upon the discovery of such a socket, it sends a signal to the socket in order to notify an associated process of packet availability. This approach, however, is inappropriate for NISP, as NISP focuses on finding only the priority of a corresponding process without having to know a corresponding socket; in fact, it does not have to figure out the corresponding process, as long as it is able to find its priority directly. In order to establish fast mapping from a port number directly to a corresponding process priority (not the corresponding process), we maintain our own table named port number to process priority mapping (PPM) (see Fig. 2) that indexes the port number. Such a table contains only the priority of a process corresponding to a port number. Each entry of the table is inserted when a process binds a socket, and becomes invalid when the process closes the socket. This PPM table allows its users to obtain a process priority efficiently, without scanning all sockets.

Hence, at the end of top half under NISP, the handler extracts a port number through early demux, translates it into a corresponding process priority from the PPM table, and stores it into a packet data structure.

C. Process-Aware Bottom Half Scheduling

The main purpose of NISP is to delay the bottom half of network packet processing when the executing process is subject to priority inversion. In Linux, *ksoftirqd*, a kernel thread for handling the bottom half, periodically checks availability of new incoming network packets. If available, the kernel thread starts packet handling one by one until its budget is exhausted.

In NISP, we assign a corresponding process priority to each available network packet, so the bottom half scheduler is able to compare the priority of the executing process and the priority assigned to the packet (i.e., the priority of its destination process). When the priority of the packet is lower than that of the current process, the NISP bottom half scheduler does not handle the packet, and delays it to the next bottom half daemon (ksoftirqd) period. This way, we can avoid the priority inversion problem.

Fig. 1 shows the effect of the NISP bottom half scheduling, in comparison to the original Linux case. We can estimate such an effect as follows. Let T_{TH} and T_{BH} denote the computation times of the top half and the bottom half handlers, respectively. Suppose n network interrupts happen on behalf of LP processes, while a HP process is currently executing. Then, in the original Linux, the current process could be interfered by the top half and the bottom half handlers n times each. That is, it could be interfered by $n \cdot (T_{TH} + T_{BH})$. However, in the NISP scheme, the current process could be interfered by only the top half handler, never by the bottom half handler. Thereby, it could be interfered by $n \cdot (T_{TH} + T_D)$, where T_D denotes the computation time to perform early demux in the top half in NISP. Considering T_{BH} is much larger than T_D, the NISP scheme can effectively support the predictable real-time execution of a HP process by reducing the interference from semantically lower priority interrupt handling.

D. Other Protocols

There are many other protocols than TCP and UDP packets, such as ARP, RARP, ICMP, and IP packet forwarding. In Linux, those kinds of packets are handled by the kernel itself, and do not notify any user processes of such packets. We consider them as associated with the kernel, so we assign the highest priority to those packets during the early demux procedure. For instance, network switches or routers create ARP control packets to maintain routing tables. Improper handling of such control packets may result in switch/router failure.

IV. IMPLEMENTATION

To implement the NISP prototype, we make a minimal modification to the Linux kernel and network device driver. We patched Linux kernel 2.6.24 as follows.

As shown in Fig. 3, our NISP scheme takes advantage of the process-related information of networking packets to integrate network interrupt handling and process scheduling. In order to extract the process-related information from packets, we add early demux functionality into the network device driver. When a network interrupt occurs, the Linux kernel runs an interrupt service routine immediately, executing a registered interrupt handler according to its interrupt request number. Most tasks performed by the interrupt handler are to fetch an incoming network packet into the main memory, reset the NIC's status register, and invoke a system call, named `netif_rx()` to insert the packet into an input packet queue, named `input_pkt_queue` in Linux.

Here, just before inserting the network packet into the input packet queue, we add our routine that finds a port number of the packet. Our routine first looks up the type of the packet. If the packet is of interest, then our routine extracts the port number of the packet.

In order to map a port number to a process priority, we make some modifications to socket-related system calls. In Linux, user processes invoke a system call, named `bind()`, to associate a local address and a port number with a socket, and invoke another system call, named `close()`, to close the socket. We wrap up those two system calls in order to manage a process priority table indexed by a port number. When a process invokes the `bind()` system call with a port number, we create an entry that contains the process priority with an index of the port number. When the process invokes `close()` later, a corresponding entry becomes invalid. This table makes it easy to look up a corresponding process priority associated with a port number. Once we find a process priority associated with a packet, we store the priority into the packet's data structure, named *priority*.

In Linux, every network packet coming from an NIC is added into the input packet queue in a first-in first-out (FIFO) fashion. In the proposed NISP scheme, each packet is equipped with a corresponding process priority through early demux. Considering that the bottom half scheduler finds the packets to handle according to their

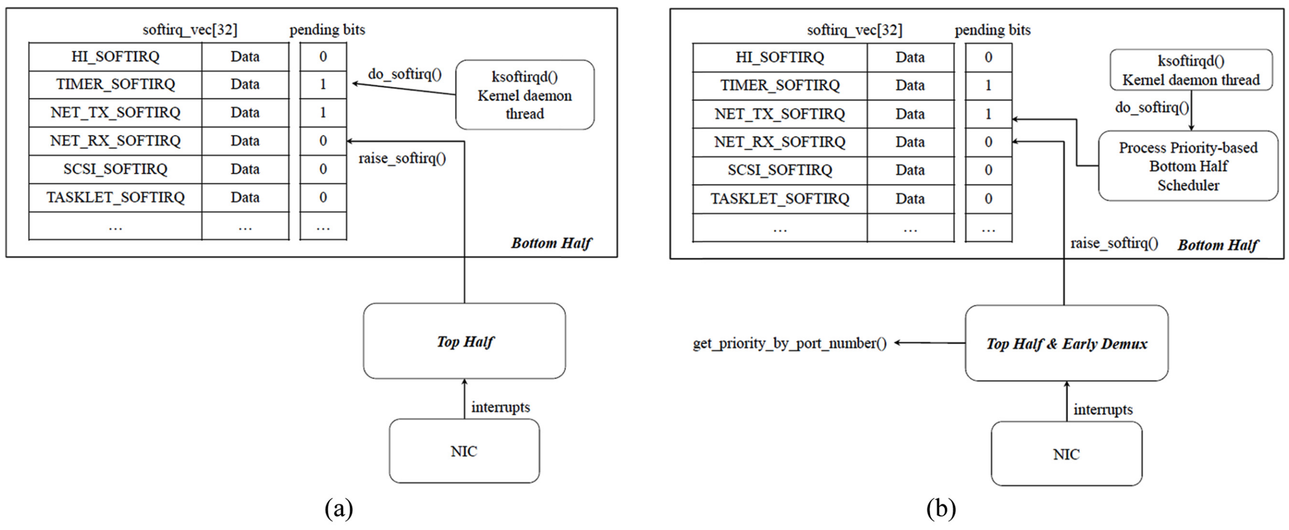

Fig. 3. (a) Linux vs. (b) the network interrupt scheduling with process priority (NISP) scheme. NIC: network interface card.

priorities, it is preferable to make the input packet queue as a priority queue, rather than a FIFO queue. In order to support an insertion sort scheme to the input packet queue, we add a new insertion sort function, named `__skb_queue_insert_sorted()`, to the existing linked list structure in Linux. Then, we replace the existing FIFO packet insertion function, named `__skb_queue_tail()`, with our new priority insertion function.

Adding a packet into the input packet queue according to its associated priority, the top half behaves in the same way as the original Linux does. Specifically, it raises a software interrupt (*softirq*) by setting a pending bit to 1 in a softirq vector array, named *softirq_vec*, and exits. The remaining work of packet processing will be performed in the bottom half.

In Linux, the bottom half becomes activated by a kernel thread, named *ksoftirqd*. The kernel thread periodically checks out the softirq vector array looking at pending bits. When it finds that the network packet reception's pending bit is set, *ksoftirqd* invokes a system call, called `net_rx_action()`, to handle packets one by one.

In the proposed NISP scheme, we make modifications to the `net_rx_action()` system call so that it handles packets only if their priorities are no lower than the priority of the process that is being executed. Since the Linux kernel maintains a pointer to the executing process, named *current*, we can get a priority from its process control block. Recall that the input packet queue is sorted by a priority. When our modified packet handler finds a packet with a priority lower than that of the executing process, it stops handling incoming packets.

V. EXPERIMENT

This section evaluates the performance of network interrupt handling on the original Linux 2.6.24 (Linux) and its modified version, patched with our NISP scheme (Linux-NISP). Specifically, we compare their network interrupt handling subsystems in terms of their effect on real-time higher-priority processes in the presence of pending network interrupts of semantically lower-priority.

A. Experiment Setup

Our experimental environment includes two machines interconnected through a 10/100 MB network switch in a private network. One machine (M1) has a 2.8-GHz AMD Phenom CPU, 4 GB main memory, and RTL8111/8168B PCI Gigabit Ethernet controller. The other machine (M2) is equipped with a Gumstix Verdex Pro XL6P embedded board [4]. It comes with a 600-MHz Marvell PXA2700 processor, 128 MB main memory and 10/100 MB SMC911x Ethernet chipset.

Our experiment involves a pair of processes that performs UDP communication sitting on either of the two machines. A UDP-sender process runs on the M1 machine, and a UDP-receiver process executes on the M2 machine. To make our experiment realistic, the UDP-sender generates a burst of packets according to a two-state Markov Modulated Poisson Process (MMPP-2), which generates a wild fluctuation in the network traffic.

Our experiment aims to measure the effect of network interrupt handling on a higher priority process, while such network interrupts are handled on behalf of a lower priority process. Hence, the M2 machine hosts a process, which has a higher priority than that of the UDP receiver process. The higher-priority process is a *face detector* process that finds faces in a given image of 128 × 128 pixels. This program is built upon the OPENCV library, which is the most widely used image-processing library. This process is highly CPU-intensive, since most of the face detecting mechanism consists of lots of image processing and array calculation.

In order to compare the effect of the original Linux and our patched Linux, the M2 machine is equipped with Linux and Linux-NISP, respectively.

The bottom half daemon, named *ksoftirqd*, has a priority of 15 by default on Linux. We assign the same priority to *ksoftirqd* during the experiment. The UDP-receiver process is given a priority of 18, and the face detector process is assigned a priority of 17. Notice that a priority of a smaller number indicates a higher priority.

Our experiment has 20 cases both on Linux and Linux-NISP. In each experiment, the UDP-sender process runs on the M1 machine, and the face detector process and the UDP-receiver process execute on the M2 machine. The face detector process is a sporadic process with a minimum inter-arrival time of 3 seconds and an estimated execution time of 1.2 seconds. This sporadic process has two job release patterns as follows: 1) one case where the inter-arrival time of two consecutive jobs is exactly 3 seconds, and 2) the other case where the inter-arrival time is longer than 3 seconds. The former case happens when a previous job finishes in 3 seconds after its release. The latter case takes place when the response time of a previous job is larger than 3 seconds. In the latter case, a next job is released as soon as the previous job finishes.

In each experiment, the face detector process executes 1000 consecutive jobs. Hence, each experiment runs for at least 3000 seconds. Each experiment runs under a different network workload. The network workload is measured as the average number of packets per second generated by the UDP-sender process.

B. Experiment Results

This section presents the results of the experiment. We investigate the performance of network interrupt handling on Linux and Linux-NISP, in supporting the real-time behavior of a HP process under various network workloads associated with a LP process. Specifically, we compare their performance in terms of response time, deadline miss ratio, and jitter. We then evaluate potential overheads imposed by Linux-NISP, in comparison to Linux. In our experiment, the face detector process works as a HP process. For simplicity, we refer to the face detector process as HP and call the UDP-receiver process LP.

Response Time. A response time is the duration from the release time to the finishing time. We measure the response time of HP, in order to assess how much interference network interrupt handling affects the execution of higher priority processes. Fig. 4 shows average response times of HP on Linux and Linux-NISP under various network workloads. The error bar in the Fig. 4 represents the standard deviation of each experiment case.

On both Linux and Linux-NISP, the response time of HP grows as the network workload increases. This is because network interrupt handling interferes more with the execution of HP in order to handle more network packets. However, the response time increases more rapidly on Linux than on Linux-NISP since both the top half and the bottom half interrupt handling interfere with HP on Linux, while only the top half does so on Linux-NISP.

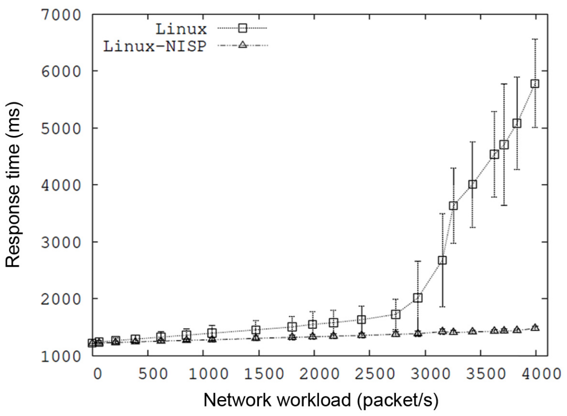

Fig. 4. Comparison of response time. NISP: network interrupt scheduling with process priority.

One interesting aspect shown in Fig. 4 is that the response time of HP starts increasing sharply from a certain point. This point corresponds to the network workload of about 2700 pkt/s; for reference purposes P_{2700} will denote this point. This phenomenon can be explained as follows. As mentioned earlier, the face detector (HP) process has two job release cases, depending on the inter-arrival time of two consecutive jobs: 1) one case where it is exactly 3 seconds, and 2) the other case where it is strictly greater than 3 seconds. The former case corresponds to a situation where a previous job finishes in 3 seconds from its release, and the next job is released exactly 3 seconds after a previous release. In this case, there is a gap between the finishing time of the previous job and the release time of the next job. This gap can be used to perform network interrupt handling without imposing any interference on the HP. The latter case happens when a next job is released as soon as a previous job finishes 3 seconds after its release. In this case, there is no such gap to perform interrupt handling that does not interfere with HP. It is shown in Fig. 4 that the latter case starts appearing from P_{2700}, and this makes the response time of HP increase rapidly from that point.

On the other hand, on Linux-NISP, the response time of HP remains fairly stable, in comparison to Linux, even though the network traffic increases. This is because only the top half handler can interfere with the HP process, and the bottom half handler is delayed in order not to interfere with the HP process. In particular, the response time of HP does not start increasing sharply, as that does in the original Linux case. This is because the bottom half handlers never interfere with the HP process, and it does not concern whether or not gaps exist between two con-

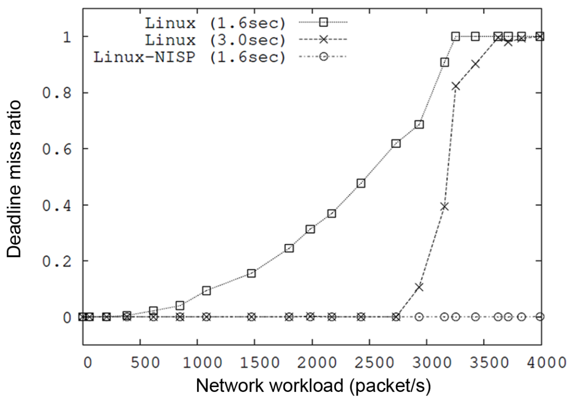

Fig. 5. Comparison of deadline miss ratio. NISP: network interrupt scheduling with process priority.

secutive jobs. Therefore, NISP particularly supports the stable execution behavior of real-time tasks, such as periodic and sporadic tasks, even under a high volume of network packets to process on behalf of LP processes.

Deadline Miss Ratio. In order to evaluate the impact of Linux-NISP, in comparison to Linux, on the real-time behavior of higher priority processes, we measure the deadline miss ratio of the HP process. Fig. 5 shows deadline miss ratios of the HP processes under different deadline values. Its deadline is given as either 1.6 seconds or 3.0 seconds, in order to see the different deadline miss patterns in two different job release cases of the HP process. In the original Linux, as the number of packets to handle increases, the deadline miss ratio keeps increasing smoothly when a deadline is 1.6 seconds. It shows a slightly different phenomenon when the deadline is 3.0 second. The HP process does not experience a deadline miss until a certain point, but it starts experiencing deadline misses rapidly from that point. In our experiments, the point is P_{2700}, which is the same point from where the response time starts to rapidly increase on Linux in Fig. 4. On the other hand, on Linux-NISP, the HP process is able to meet all the deadlines even under heavy network workloads, because the bottom half handler does not interfere with it at all. This shows that NISP is effectively supportive of HP processes, ensuring that they meet their deadlines.

Jitter. A jitter is the time variation of two consecutive outputs, and it is measured as the time difference between the finishing time of a job and the finishing time of its previous job. Jitter is critical to the system stability in many control systems and to the quality of service of multimedia applications. In our experiments, the MMPP-2 traffic model generates a wild packet fluctuation, and this potentially affects the jitter of the HP process. Figs. 6 and 7 show the jitters of HP under various network workloads on Linux and Linux-NISP, respectively. The differ-

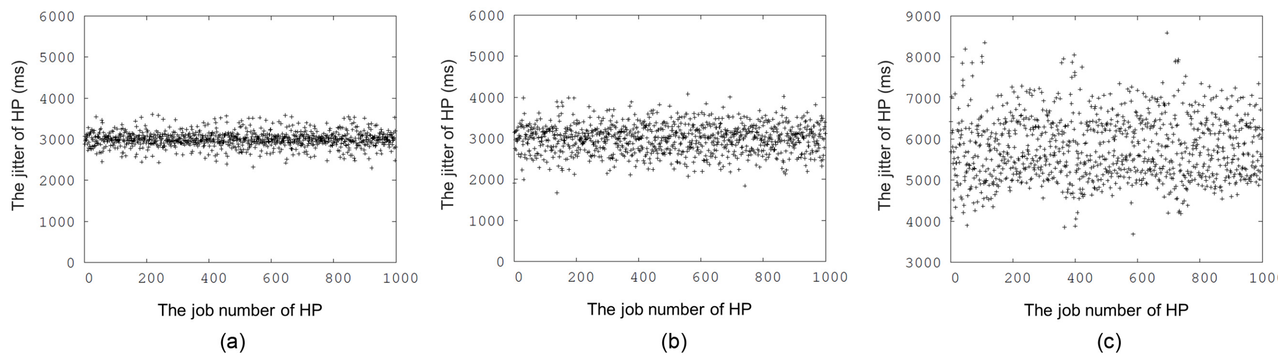

Fig. 6. Jitters on Linux. (a) 1079 pkt/s, (b) 2427 pkt/s, and (c) 3989 pkt/s. HP: higher priority.

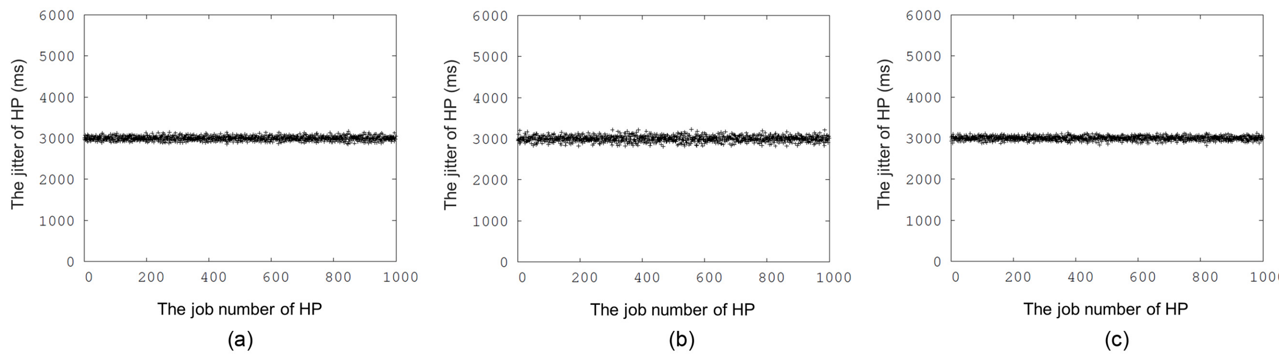

Fig. 7. Jitters on Linux-NISP. (a) 1079 pkt/s, (b) 2427 pkt/s, and (c) 3989 pkt/s. NISP: network interrupt scheduling with process priority, HP: higher priority.

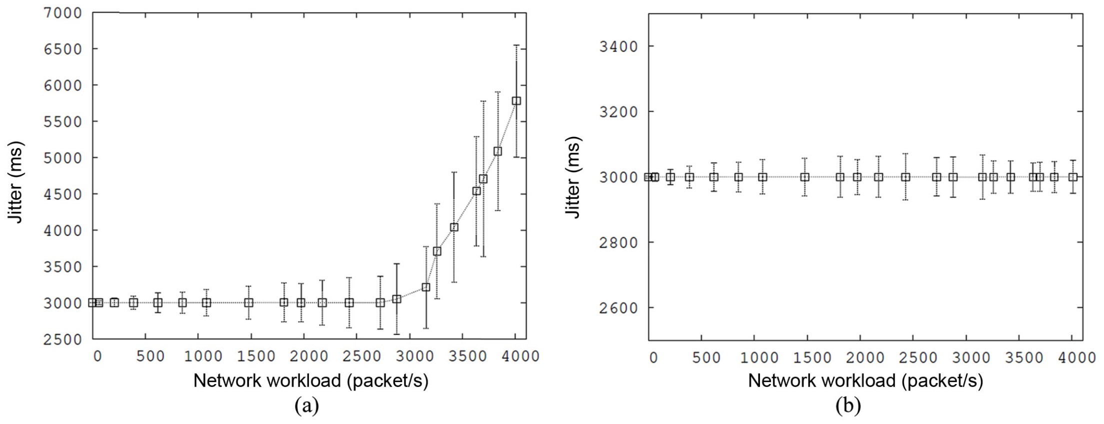

Fig. 8. Average jitters under different network workloads. (a) Linux and (b) Linux-NISP. NISP: network interrupt scheduling with process priority.

ent network workloads in Figs. 6 and 7 are 1079 pkt/s, 2427 pkt/s, and 3989 pkt/s. The x-axis of the graphs represents the job number of the HP process. In each case, 1000 jobs of HP execute.

Fig. 6 shows that the jitter becomes more diverging on Linux as the network workload increases. This is because interrupt handling takes place irregularly according to the packet fluctuation generated by the MMPP-2 traffic model, and such interrupt handling interferes with HP process irregularly. On the other hand, Fig. 7 shows that the jitter is relatively constant in Linux-NISP even though the network traffic increases. Even though the traffic establishes fluctuation, the HP process is little affected by such traffic fluctuation. This is because interrupt handling does not affect the HP process much, while traffic fluctuation can affect such interrupt handling. This shows that NISP is effective in protecting HP processes from irregular interrupt handling, and it is practically suitable for jitter-sensitive real-time applications. This is because in reality most traffic is from variable workloads, rather than constant ones.

Fig. 8 shows average jitters with standard deviations (with error bars) in various traffic models. In stable systems, the mean value of jitters is generally 0 (i.e., it is 3s in our experiment cases), and their standard deviation represents how jitters are significant, i.e., how diverging they are. In Fig. 8(a), the average value of jitters remains as 3s until a certain point, and then it starts increasing. The certain point is P_{2700}; it is from this point that the response time and the deadline miss ratio of HP starts increasing rapidly. From P_{2700}, the average jitter keeps increasing sharply as the network load increases. Furthermore, its standard deviation also keeps increasing. On the other hand, Fig. 8(b) shows that the average jitter remains constantly at 3s regardless of the network loads. We note that Fig. 8(b) has a different y-axis scale.

Network throughput. Finally, we evaluate the overhead of Linux-NISP, in terms of network throughput, in comparison to Linux. Linux-NISP imposes two over-

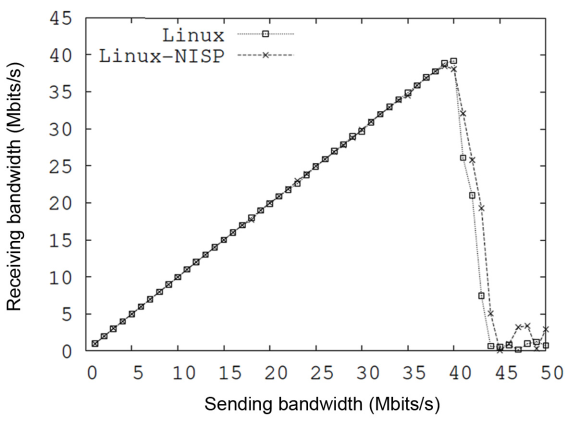

Fig. 9. Comparison of network throughput. NISP: network interrupt scheduling with process priority.

heads in interrupt handling. One is early demux in the top half, and the other is the bottom half priority-based scheduling. We evaluate such overheads by measuring the throughput of the network connection between two sender and receiver processes on the M1 and M2 machines. We make use of a commonly used open source network testing tool on Linux, named *iperf* [5]. Two *iperf* programs execute on M1 and M2, with one on M1 as a sender and the other on M2 as a receiver. The sender on M1 sends UDP packets at a constant bandwidth, and the receiver on M2 receives the packets, and measures the receiving bandwidth. We note that the face detector process is excluded in the experiment. The overhead imposed by Linux-NISP would increase interrupt handling time, and thereby decrease the number of packets delivered to the receiver the *iperf* program. Fig. 9 shows the receiving bandwidth on Linux and Linux-NISP. It shows that the receiving bandwidth goes up to 40 Mbit/s on Linux, while it goes up to 39 Mbit/s on Linux-NISP. This shows that the overhead of Linux-NISP could degrade network performance by 2.5%. This would effectively support the real-time behavior of HP processes.

VI. RELATED WORK

Several research studies propose predictable interrupt management schemes for real-time systems. Interrupt handling has been traditionally separated from process scheduling, with a higher priority placed on interrupt handling. This can lead to a priority inversion problem, in which semantically LP interrupt handling can block the execution of a HP process. Towards addressing this problem in a real-time context, a couple of studies [1, 6] propose the integration of interrupt handling and process scheduling into a single priority space, but in opposite ways. One approach [1] describes a mechanism to treat interrupt handlers as threads, and the other one [6] introduces a platform that manages threads through interrupt handling. However, these approaches do not consider associating individual interrupts with their corresponding processes.

A few studies propose another framework to address the priority inversion problem. This framework aims to identify the relationship between an individual interrupt and its corresponding process, and connects individual interrupting handling with process scheduling according to the corresponding process' priority. A couple of techniques [2, 7] are introduced to identify such a corresponding process. One is a probabilistic approach [7]. When an I/O device raises an interrupt, there exists a set of processes waiting on the corresponding I/O device, in particular, in the case for blocking I/O system calls (e.g., blocking `read()` requests). This approach employs a prediction scheme that considers the highest priority process among the waiting processes as the corresponding process of the interrupt of interest. This probabilistic approach may not find an exact corresponding process of an interrupt, but it could be applied to general I/O device interrupts.

Another approach [2] proposes to identify an exact corresponding process of an interrupt, in the context of network interrupt handling. Incoming network packets generally have certain destination addresses. When a network device issues an interrupt for an incoming packet, in particular, with a certain destination address, there exists an intended receiver process waiting on the destination socket. Early demuxing is employed to identify such intended receiver processes accurately at an earlier interrupt phase. Then, the priority of the corresponding process can be effectively used in making decision on when the bottom-half interrupt handling takes place. The lazy receiver processing (LRP) approach [2] is most relevant to our NISP scheme. LRP aims to support stable overload behavior and fair resource allocation under a heavy load from the network, but does not consider real-time aspects much. LRP can successfully achieve system stability and fairness by scheduling incoming network traffic at the priority of the corresponding process and discarding excess traffic early. However, LRP can unnecessarily delay the execution of HP processes. Specifically, LRP performs network packet processing in the context of user process performing system calls (i.e., `receive()`). As indicated in [2], this way can delay the delivery of a packet to the corresponding process and, thereby, the execution of the process. On the other hand, our NISP scheme does not unnecessarily yield such a delay.

A few studies [7, 8] introduce an account of interrupt handling costs in order to support the predictable real-time behavior of user processes. Such accounting can be orthogonally applied to our NISP scheme as well.

RTLinux [9] takes a different approach to the priority inversion problem. It draws a clear separation between real-time processes and non-real-time processes to ensure that interrupt handling on behalf of non-real-time processes cannot interrupt the execution of real-time processes. However, a priority inversion problem still exists between HP processes and LP processes by means of interrupt handling. Our approach can be orthogonally applied to RTLinux to further address such a problem.

Many protocols have been introduced to address the priority inversion problem when tasks are accessing critical sections in a mutually exclusive manner. Such protocols include the Priority Inheritance Protocol (PIP) [10], the Priority Ceiling Protocol (PCP) [11], and Stack Resource Policy (SRP) [12]. While these protocols address the priority inversion problem within the context of process scheduling, our work considers process scheduling and interrupt handling together.

VII. CONCLUSION

This paper presents the design and implementation of a new Linux network interrupt handling scheme (NISP) for incoming packets. It combines packets with the priorities of their corresponding receiver processes, and their interrupt handling is performed according to priorities. This approach prevents the priority inversion problem, in particular, between the currently executed process and the receiver process of a packet under interrupt handling.

We demonstrate the effectiveness of the proposed NISP scheme by implementing it on Linux. Experiments show that it effectively provides the predictable execution of HP processes. Specifically, it shows that addressing the priority inversion problem involved in networking interrupt handling can effectively support the predictable execution behavior of HP processes, even when a high volume of network packets is being sent to a LP process. Even though the load of network interrupt handling, on behalf of the LP process, keeps increasing, the execution of the HP process remains stable with steady response times and jitters.

This paper only covers UDP packets. Our future work includes accommodating more sophisticated protocols, such as TCP. While TCP employs flow control, delaying interrupt handling can defer TCP acknowledgments, and may make the TCP communication unstable. We plan to incorporate TCP packets to address such concerns.

REFERENCES

1. L. E. Leyva-del-Foyo, P. Mejia-Alvarez, and D. De Niz, "Predictable interrupt management for real time kernels over conventional PC hardware," in *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, San Jose, CA, 2006, pp. 14-23.
2. P. Druschel and G Banga, "Lazy receiver processing (LRP): a network subsystem architecture for server systems," in *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, Seattle, WA, 1996, pp. 261-275.
3. G. Parmer and R. West, "Predictable interrupt management and scheduling in the composite component-based system," in *Proceedings of Real-Time Systems Symposium*, Barcelona, Spain, 2008, pp. 232-243.
4. Gumstix Inc., http://www.gumstix.com.
5. Iperf (network testing tool), http://iperf.sourceforge.net.
6. W. Hofer, D. Lohmann, F. Scheler, and W. Schroder-Preikschat, "Sloth: threads as interrupts," in *Proceedings of 30th IEEE Real-Time Systems Symposium (RTSS'09)*, Washington, DC, 2009, pp. 204-213.
7. Y. Zhang and R. West, "Process-aware interrupt scheduling and accounting," in *Proceedings of 27th IEEE Real-Time Systems Symposium (RTSS'06),* Rio de Janeiro, Brazil, 2006, pp. 191-201.
8. K. J. Jung, S. G. Jung, and C. Park, "Stabilizing execution time of user processes by bottom half scheduling in Linux," in *Proceedings of 16th Euromicro Conference on Real-Time Systems (ECRTS 2004)*, Catania, Italy, 2004, pp. 71-78.
9. Real-Time Linux Foundation Inc., http://www.realtimelinuxfoundation.org.
10. L. Sha, J. P. Lehoczky, and R. Rajkumar, "Task scheduling in distributed real-time systems," in *IECON'87: Automated Design and Manufacturing (Proceedings of SPIE 0857)*, V. K. Huang, editor, Bellingham, WA: SPIE, pp. 909-917, 1987.
11. R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *Proceedings of Real-Time Systems Symposium* (*RTSS)*, Huntsville, AL, 1988, pp. 259-269.
12. T. P. Baker, "Stack-based scheduling of realtime processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67-99, 1991.

Minsub Lee

Minsub Lee is a software engineer. He received the B.S. degree from Soongsil University, Seoul, Korea in 2009. He received the M.S. degree from Korea Advanced Institute of Science and Technology, Daejeon, Korea in 2011. His research interests are in embedded system and speech recognition.

Hyosu Kim

Hyosu Kim is a Ph.D. student in School of Computing at KAIST, South Korea. He received the B.S. degree from Sungkyunkwan University, Suwon, Korea in 2010. He received the M.S. degree from Korea Advanced Institute of Science and Technology, Daejeon, Korea in 2012. His research interests include resource management and device collaboration on mobile platforms.

Insik Shin

Insik Shin is an associate professor in School of Computing at KAIST, where he joined in 2008. He received a B.S. from Korea University, an M.S. from Stanford University, and a Ph.D. from University of Pennsylvania all in Computer Science in 1994, 1998, and 2006, respectively. He has been a post-doctoral research fellow at Malardalen University, Sweden, and a visiting scholar at University of Illinois, Urbana-Champaign until 2008. His research interests lie in mobile computing, real-time embedded systems, and cyber-physical systems. He has served various program committees in real-time embedded systems, including RTSS, RTAS, ECRTS, and EMSOFT. He received best paper awards, including the Best Paper awards from RTSS in 2003 and 2012, the Best Student Paper Award from RTAS in 2011, the Best Paper award from CPSNA in 2014, and Best Paper runner-ups at RTSS and ECRTS in 2008.