On a NIC's Operating System, Schedulers and High-Performance Networking Applications

Yaron Weinsberg[1], Tal Anker[2], Danny Dolev[1], and Scott Kirkpatrick[1]

[1] The Hebrew University Of Jerusalem
{wyaron,dolev,kirk}@cs.huji.ac.il
[2] RADLAN - A Marvell Company, Israel
tala@marvell.com

Abstract. Today's modern high-end Network Interface Cards (NICs) are equipped with an onboard CPU. In most cases, these CPU's are only used by the vendor and are operated by a proprietary OS, which makes them inaccessible to the HPC application developer. In this paper we present a design and implementation of a framework for building high-performance networking applications. The framework consists of an embedded NIC Operating System with a specialized scheduler. The main challenge in developing such a scheduler is the lack of a preemption mechanism in most high-end NICs. Our scheduler provides finer-grained schedules than the alternatives. We have implemented several network applications, and were able to increase their throughput while decreasing the host's CPU utilization.

1 Introduction

Today HPC clusters are built from off-the-shelf components, such as standard Intel based servers. In an HPC application, latency is extremely important for inter-process communication among compute nodes. Additionally, computation tasks running on the compute nodes need all the CPU cycles that they can get.

Today's modern high-end NICs are equipped with a CPU onboard and line speeds reaching up to 10Gbps. Such high speeds pose a great challenge for today's CPUs and bus architectures. As a result, in order to better utilize the new high speed network infrastructure, the industry is moving towards an approach that offloads part of the hosts' protocol processing to the NIC (e.g. TCP Offload Engines (TOEs) [1]). In most cases, the NICs' CPUs are not fully utilized by the vendor and are usually running a proprietary OS that limit the applications ability to take advantage of them.

In this paper we propose a design and implementation framework that enables a developer to build a high-performance networking application. The developer can design tasks that will be executed at the NIC. The framework enables a developer to easily divide the application logic between the host and the NIC. The framework inherently

This paper will appear at the 2006 International Conference on High Performance Computing and Communications.

facilitates the communication between the host and the NIC portions of the application. Another important contribution of this work is a modified NIC level scheduler. Most high-end NICs do not support preemption, thus when trying to schedule user tasks on the NIC, using a common real time scheduling algorithm, we found that there was a great inefficiency in the resulting schedule and the cluster throughput. Our new scheduling scheme (henceforth called: <Sched>++) is capable of extending any given non-preemptive scheduling algorithm with the ability to create finer-grained schedules.

2 Related Work

Today, HPC clusters can increase their achieved throughput by TOEs. However, to the best of our knowledge, there is no NIC oriented operating system that enables developers to design *applications* that can utilize the NIC's functionality. Although some vendors have developed a proprietary OS for their platforms, such an OS does not allow the application developer to integrate parts of the application code into the NIC. The ability to immigrate user tasks to the NIC can further increase the performance of HPC applications.

Spine – is a safe execution environment [2] that is appropriate for programmable NICs. Spine enables the installation of user handlers, written in Modula-3, at the NIC. Although Spine enables the extension of host applications to use NIC resources it has one major limitation. Since Spine extensions are executed as a result of an event, building stand-alone applications at the NIC is very difficult. Even for event-driven applications, the developer is forced to dissect the application logic to a set of handlers.

Arsenic – is a Gigabit Ethernet NIC that exports an extended interface to the host operating system [3]. Unlike conventional adaptors, it implements some of the protection and multiplexing functions traditionally performed by the operating system. This enables applications to directly access the NIC, thus bypassing the OS. The Ethernet Message Passing (EMP) [4] system, of the Ohio Supercomputer Center (OSC), is a zero-copy and OS-bypass messaging layer for Gigabit Ethernet. The EMP protocol processing is done at the NIC and a host application (usually MPI applications) can directly manipulate the NIC. Arsenic and EMP provide very low message latency and high throughput but lack the support for offloading.

TOE – is a technique used to move some of the TCP/IP network stack processing out of the main host and into a network card [1]. While TOE technology has been available for years and continues to gain popularity, it has been less than successful from a deployment standpoint. TOE only targets the TCP protocol, thus, user extensions are out of its scope.

3 Environment

Our programmable interface card is based on the Tigon chipset. The Tigon programmable Ethernet controller is used in a family of 3Com's Gigabit NICs. The Tigon controller

supports a PCI host interface and a full-duplex Gigabit Ethernet interface. The Tigon has two 88 MHz MIPS R4000-based processors which share access to external SRAM. Each processor has a one-line (64-byte) instruction cache to capture spatial locality for instructions from the SRAM. Hardware DMA and MAC controllers enable the firmware to transfer data to and from the system's main memory and the network, respectively. The Tigon architecture doesn't contain an interrupt controller. The motivation is to increase the NIC's runtime performance by reducing the overhead imposed by interrupting the host's CPU each time a packet arrives or a DMA request is ready. Furthermore, on a single processor the need for synchronization and its associated overhead is eliminated.

4 NIC Operating System (NICOS)

This section presents the NICOS services. We start by describing the memory management service of NICOS, the NICOS task related APIs, the NICOS networking and the NICOS filtering APIs. Following that we provide a detailed description of the NICOS scheduling framework. We then conclude with several sample applications that use NICOS and we show the significant gain in the applications performance.

4.1 Memory Management

NICOS has to allocate memory each time a task, a queue or a packet is created. NICOS default memory allocation algorithm is based on the "boundary tag method" described in [5], which is suitable for most applications. Implementing a "generic" memory allocation mechanism is problematic: It takes up valuable code space, it is not thread safe and it is not deterministic. Since different realtime systems may have very different memory management requirements, a single memory allocation algorithm probably will not be appropriate. To get around this problem the memory allocation APIs provided in NICOS can be easily replaced by using the filtering APIs (see Section 4.4). A user's task can easily replace the default methods by installing a special kind of a filter. The registered method (i.e., the "filter" action) will be called instead of the default allocation routine. NICOS memory allocation APIs can also enable a developer to choose the *target* of the allocated memory. Memory consuming applications can allocate memory at the host. The memory is transparently accessed using DMA. This scheme is also suitable for developing OS bypass protocols, which removes the kernel from the critical path and hence reduces the end-to-end latency.

4.2 Task Management

NICOS provides several task management APIs that enable a developer to create/destroy tasks and to control their lifecycle state. The API enables a developer to create a periodic or non-periodic task, to yield, sleep, suspend, resume and kill a task. Although periodic tasks can be implemented by a developer on top of a sleep API, we added an explicit facility for periodic tasks so the OS is aware of them. Such a design allows the OS

to minimize the ready-to-running latency. Providing the timeliness guarantees required by NICOS has been a major challenge due to the non-preemptive architecture of these NICs.

4.3 Networking

The current networking API is very simple. NICOS provides only a single method that sends raw data. The data is provided by the developer and includes all of the necessary protocol headers. NICOS supports synchronous and asynchronous send calls. The asynchronous ones are non-blocking. When using the synchronous mode, the execution is blocked until frame transmission is completed. Upon completion, the provided callback is called. Receiving a packet is currently done only via filter registration.

4.4 Filtering

When deciding which functionality is needed to be offloaded to the NIC, we looked for common building blocks in today's networking applications. We have found that the ability to inspect packets and to classify them according to specific header fields is such a building block. For instance, the classification capability is useful for firewall applications, applying QoS for certain traffic classes, statistics gathering, etc. Therefore we enhanced the NICOS services with a packet filtering (classification) capability, and the optional invocation of a user installed callback per packet match. In NICOS, a filter is a first class object - it can be introspected, modified and created at runtime.

The "Ping Drop" task (Program 1), which drops all ICMP packets, demonstrates the ease of use of the NICOS filtering API.

4.5 Scheduling

Schedulers for non-preemptive environments usually use an event-driven model. For example, the programmable NIC we are using for evaluating NICOS, provides a special hardware register whose bits indicate specific events. This event register is polled by a dispatcher loop that invokes the appropriate handler. Once the event handler runs to completion, the dispatcher loop resumes.

Providing timeliness guarantees for NIC based tasks can be beneficial for real-time and HPC applications. NICOS enables a developer to easily install a custom scheduler, implementing whatever scheduling policy is needed. NICOS provides several non-preemptive schedulers and an innovative scheme that can further improve their schedule. We have used this scheme to implement an enhanced version of the *Earliest Deadline First (EDF)* scheduler, which is described in details in the next section.

The time from the moment a task becomes ready-to-run until it starts execution.

In the future we plan to write a minimal networking stack for the NIC.

Program 1 Installing "Ping Drop" Filters

```
void registerPingDropFilters(void) {
  /* we would like to match ICMP packets */
  valueMask[0] = ICMP_PROTOCOL;
  bitMask[0] = 0x1; // match 1 byte
  /* start matching at ICMP_PROTOCOL_BYTE */
  pattern_filter.startIndex =ICMP_PROTOCOL_BYTE;
  pattern_filter.length = 1;
  pattern_filter.bitMask = bitMask;
  pattern_filter.numValues = 1;
  pattern_filter.valueMask = &valueMask;
  /* create the filter, add to Rx/Tx flows */
  pingDropFilter.filter_type = STATIC_PATTERN_FILTER;
  pingDropFilter.pattern_filter = &pattern_filter;
  nicosFilter_Add(&nicosTxFilters,&pingDropFilter,DROP,NULL,
                  GENERAL_PURPOSE_FILTERS_GROUP,&pingFilterTxId);
  nicosFilter_Add(&nicosRxFilters,&pingDropFilter,DROP,NULL,
                  GENERAL_PURPOSE_FILTERS_GROUP,&pingFilterRxId);
}
```

4.6 <Sched>++ Algorithm

Common Schedulers. The first scheduling algorithm we have implemented is the simple *Cyclic-Executive* scheduler [6]. The primary advantages of the Cyclic-Executive approach are: being simple to understand, easy to implement, efficient and predictable. Unfortunately, the deterministic nature of a Cyclic-Executive requires a lot of ad-hoc tweaking to produce deterministic timelines, which then must be tested thoroughly. The second scheduling algorithm we have implemented is the non-preemptive version of the EDF algorithm [7]. In EDF, the task with the earliest deadline is chosen for execution. In the non-preemptive version of EDF, the task runs to completion.

Both EDF and Cyclic-Executive are not optimal for a non-preemptive environment such as in our NIC architecture. For a set of scheduable tasks, the resulting task schedule meets the tasks' realtime requirements, however with a rather low CPU utilization. Therefore, we have devised a new task scheduling scheme (denoted as <Sched>++) which can be used to enhance any given non-preemptive scheduler. This scheme utilizes the **compiler** capabilities in order to create an optimized tasks schedule.

Related Definitions. A task is a sequence of operations to be scheduled by a scheduler. A task system $T = \{T_1, \cdots, T_n\}$, where each task T_i is released periodically, is called a *periodic task system*. Each task T_i is defined by a tuple (e_i, d_i, p_i, s_i), where e_i is the task's Worst Case Execution Time (WCET), s_i is the first time at which the task is ready to run (also known as the start time), d_i is the deadline to complete the tasks once it is ready to run, and p_i is the interval between two successive releases of the task. Thus, a task T_i is first released at s_i and periodically it is released every p_i. After each periodic release, at some time t, the task should be allocated e_i time units before deadline $t + d_i$. A *non-periodic* task is a task that is released occasionally, and at each invocation that task may require a different execution time. A *hybrid task system* is a

system that contains both periodic and non-periodic tasks. To differentiate between the periodic and non-periodic tasks, a periodic task will be denoted as $\tilde{T}$.

<Sched>++ assumes a *hybrid task system*, where for each periodic task, $d_i = p_i$. To represent the runtime instance of a task, the notion of a *ticket* of a task is introduced. A ticket of a periodic task, $\tilde{T_i}$ is defined as the tuple (e_i, p_i, Pr_i), where e_i and p_i are the execution and the period of the task, and Pr_i is the task's priority. The ticket of a non-periodic task, T_j, is (e_j, Pr_j). This assumes that any type of task scheduler used by the OS can be extended using this ticket.

Algorithm Overview <Sched>++ uses several compile-time techniques, which provide valuable information that can be used at runtime. The developer uses <Sched>++ specific compiler directives in order to define the system's tasks and tickets. The compiler uses these tickets as simple data structures in which it can store the calculated WCETs.

The compiler uses the generated control flow graph in order to calculate the WCET of the periodic and non-periodic tasks. Typical periodic tasks are comprised of a single calculated WCET, while non-periodic tasks may be comprised of a set of WCETs. In our context, a WCET is defined as the worst case execution time between two successive yields.

The ability of a compiler to modify the developer's code, at predefined places, is also utilized. By modifying the code, the ticket primitive is maintained automatically. The enhanced compiler updates the ticket with the task's next WCET prior to each *yield* invocation. This technique also eliminates the need to introduce a complicated runtime structure that contains all the WCETs of a given non-periodic task. A *single* ticket is recycled to represent the next task segment WCET at runtime.

<EDF>++ Algorithm In order to implement the enhanced version of the EDF algorithm, the ticket of a periodic task $\tilde{T}$ is extended to be (e, p, Nr, Nd, Pr), where the additional fields Nr and Nd are the next release time and deadline of the task, respectively. Figure 1 presents the main logic behind the <EDF>++ algorithm, which is invoked by the *Yield()* function call. Part I of the algorithm starts with the classical EDF algorithm. The algorithm selects the next periodic task T_{next} that has the earliest deadline among all periodic tasks that are ready to run.

Part II of the algorithm is invoked when no periodic task is ready to run. The algorithm uses the tickets of the non-periodic tasks in order to select the next task to run. The chosen task should be able to run without jeopardizing the deadline of the next (earliest) periodic task. The scheduler considers the subset of non-periodical tasks that are ready to run, such that their next execution time is smaller than the slack time (the time until the next periodic task is ready). Among such tasks, the algorithm can use various criteria to pick the next task to be scheduled. For instance, one can use the algorithm in [8], which chooses a set of tasks that minimizes the remaining slack time. Any such algorithm would use the next execution time (WCET) of the tasks listed in their tickets. When there is no suitable task for execution, the IDLE task is invoked until the next periodic task is ready to run (part III).

Yield() called from task T_k:

I: $T_{next} = \{\tilde{T}_i | \tilde{T}_i.Nd = \min(\tilde{T}_j.Nd | \tilde{T}_j.Nr \geq t)\}$;

```
/* If no periodic task is ready, then
choose from the non-periodic tasks */
```

II: `if` $(T_{next} = NULL)$

$SlackTime =$ `duration until next periodic task is ready;`

```
/* Pick the next non-periodic task
that will run at most 'SlackTime'
time units */
```

$T_{next} = PickNonPeriodicTask(SlackTime)$;

```
/* if no task is ready, the Idle task
will run for the time duration until
the next periodic task is ready */
```

III: `if` $(T_{next} = NULL)$

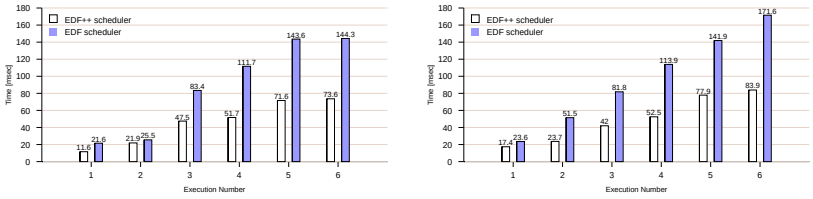
$T_{next} = Idle\_Task(Timeout)$

`SwitchTo(`T_{next}`);`

Fig. 1. <EDF>++ Scheduler

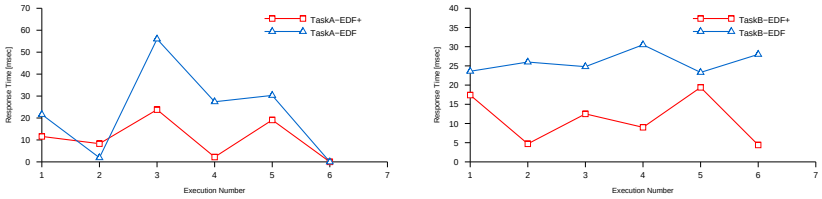
Notice that the scheduling algorithm strives to schedule non-periodic tasks whenever there is an available time slot in the schedule. Available time slots may exist between periodic slots or whenever a task completes its execution ahead of time, which can only be determined at runtime.

<EDF>++ Evaluation. We have implemented an experimental system with both EDF and <EDF>++. Our task set includes twenty tasks where about half of them are periodic. We have executed the system with various periods and constraints. On average, for plain EDF, the IDLE task has been executed 28.6% of the time, yielding a CPU utilization of 71.4%. For the <EDF>++ algorithm, the IDLE task ran 14.7% of the time corresponding to 85.2% CPU utilization, an increase of 20% in the system's throughput.



(a) Task A (b) Task B

Fig. 2. Invocation Times



(a) Task A (b) Task B

Fig. 3. Response Times

We have also compared the response times of the tasks. Figure 2 shows a sequence of invocation times for a sample task measured from the system's start time. The x-axis shows the number of invocations, where the y-axis presents the time when the specific invocation occurred. The response times, in-between invocations, for the non-periodic tasks are presented in Figure 3. For example, the average response time for task A, using <EDF>++, is $10.83ms$ with standard deviation of $8.51ms$ versus $22.86ms$ and $18.87ms$ using EDF (a 53% decrease in the average waiting time). For task B the values are: $11.23ms$ and $5.78ms$ against $26.03ms$ and $2.54ms$ (57% decrease in the average waiting time). The graphs clearly show that the response times for the non-periodic tasks using the <EDF>++ scheduler are improved.

Regarding the response times for periodic tasks, the average response time is approximately the same ($1.37ms$ vs. $1.33ms$ with standard deviation of $2.49ms$ vs $2.39ms$). Thus, the improved response for the non-periodic tasks didn't affect the response time for the periodic tasks.

4.7 Sample Applications using NICOS

A Firewall Application – An application of particular promise for offloading is a firewall application. Since a firewall is an application that filters packets according to a user defined security policy, earlier filtering (especially discarding packets) has a potential for significant improvements in performance. A firewall application on a NIC also has the additional advantage that it is harder for an adversary to modify than a software application running at the host.

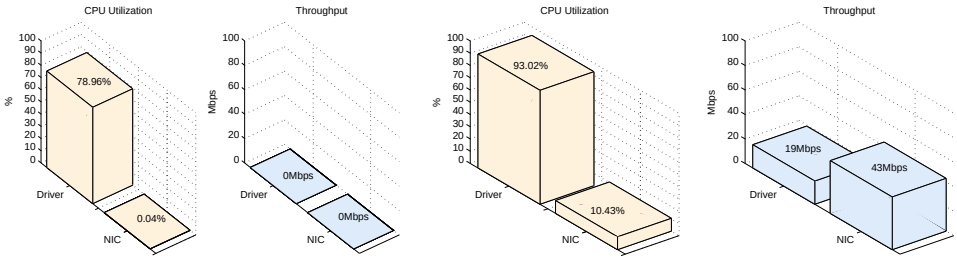
We have designed and implemented a firewall application, called *SCIRON* [9], for a NIC. As presented in Section 4.4, NICOS provides a framework that enables a developer to install filters. Filters can be installed both at the firmware level and/or at the kernel level. SCIRON's firewall is implemented as a set of such firmware filters.

In order to simulate common kernel-based firewalls for performance evaluation, we have also installed filters at the driver layer. All comparisons shown below, compare the same firewall code (with the same filtering policy) between the driver based firewall and the NIC based firewall.

Currently, the firewall code is fully stateless, thus the state is not saved between successive filter action invocations.

Firewall Evaluation. This section compares the performance between host based and NIC based firewalls. Many parameters have an impact on the firewall's performance. For example, the number of rules, current CPU utilization, packet size, ratio of incoming to outgoing packets, total number of packets, number of packets accepted vs number rejected, can all potentially influence the performance.

Performance can be measured using two parameters. The first is the load on the CPU and the second is the throughput. In this section we discuss several typical scenarios. In the first scenario we present (Figure 4(a)), the firewall discards all the packets it receives. During this scenario the CPU is only running system processes. The CPU is on the left of the graph and throughput is on the right.



(a) Receiving - Discarding all packets (b) Receiving - 50% acceptance rate

Fig. 4. Firewall Performance

As expected, in this scenario the CPU utilization when using the firewall implemented on the NIC is approximately zero, whilst for the same firewall on the host it is quite high. The second scenario presented is given in Figure 4(b). This scenario is probably a more realistic behavior for a typical host machine. It is evident that the NIC based firewall has better performance both in CPU utilization and throughput.

STORM. Occasionally, clustered HPC applications need to synchronize the cluster's activities or to perform cluster-wide operations. Therefore, the cluster software usually needs to implement a basic locking and/or consensus algorithms that consumes a lot of bandwidth and degrades the cluster's performance.

A STORM cluster [10] consists of five nodes running Linux where each node is hosting a programmable NIC. As a proof of concept application we have implemented Lamport's Timestamp ordering algorithm [11] providing an agreed order on transmitted messages. This messaging system's performance is much better (latency of 84 us, and throughput of 768 Mb/s) than the host level implementation (latency of 200 us, and throughput of 490 Mb/s).

The benchmark ran on two hosts (Intel Pentium 4 CPU 2.4Ghz, 512MB) connected via 100Mbps ethernet.

5 Conclusions

This paper presented a novel framework for building high-performance applications that can benefit from offload capabilities. The framework is comprised of a NIC Operating System and an innovative scheduler. We have implemented several HPC applications using this framework and have demonstrated increased application throughput. According to the International Technology Roadmap for Semiconductors (ITRS), by 2007, one million transistors will cost less than 20 cents. This current trend motivates hardware and embedded system designers to use programmable solutions in their products. We believe that programmable NICs will soon become widespread. The need for such a framework is apparent.

Acknowledgments. We would like to acknowledge Udi Weinsberg for his helpful suggestions regarding the scheduling algorithm. We would also like to acknowledge all of the NICOS project members: Maxim Grabarnik, Adamovsky Olga and Nir Sonnenschein.

References

1. Currid, A.: TCP offload to the rescue. Queue **2**(3) (2004) 58–65
2. Fiuczynski, M.E., Martin, R.P., Owa, T., Bershad, B.N.: Spine: a safe programmable and integrated network environment. In: EW 8: Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications, New York, NY, USA, ACM Press (1998) 7–12
3. Pratt, I., Fraser, K.: Arsenic: A user-accessible gigabit ethernet interface. In: INFOCOM. (2001) 67–76
4. Shivam, P., Wyckoff, P., Panda, D.: Emp: zero-copy os-bypass nic-driven gigabit ethernet message passing. In: Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM), New York, NY, USA, ACM Press (2001) 57–57
5. Wilson, P.R., Johnstone, M.S., Neely, M., Boles, D.: Dynamic storage allocation: A survey and critical review. In: Proc. Int. Workshop on Memory Management, Kinross Scotland (UK) (1995)
6. Cheng, S.C., Stankovic, J.A., Ramamritham, K.: Scheduling algorithms for hard real-time systems: a brief survey. (1989) 150–173
7. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM **20**(1) (1973) 46–61
8. Shmueli, E., Feitelson, D.G.: Backfilling with lookahead to optimize the performance of parallel job scheduling. In Feitelson, D.G., Rudolph, L., Schwiegelshohn, U., eds.: Job Scheduling Strategies for Parallel Processing. Springer Verlag (2003) 228–251 Lect. Notes Comput. Sci. vol. 2862.
9. Weinsberg, Y., Pavlov, E., Amir, Y., Gat, G., Wulff, S.: Putting it on the NIC: A case study on application offloading to a Network Interface Card (NIC). In: Consumer Communications and Networking Conference, IEEE CCNC. (2006)
10. : Super-fast Transport Over Replicated Machines (STORM) (2004) Available at site: `http://www.cs.huji.ac.il/~wyaron`.
11. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (1978) 558–565