Accelerating Distributed Computing Applications Using a Network Offloading Framework

Yaron Weinsberg[1], Danny Dolev[1], Pete Wyckoff[2], Tal Anker[1,3]
[1]The Hebrew University Of Jerusalem [2]Ohio Supercomputer Center [3]Marvell
{wyaron,dolev}@cs.huji.ac.il pw@osc.edu tala@marvell.com

Abstract

During the last two decades, a considerable amount of academic research has been conducted in the field of distributed computing. Typically, distributed applications require frequent network communication, which becomes a dominate factor in the overall runtime overhead. The recent proliferation of ***programmable*** *peripheral devices for computer systems may be utilized in order to improve the performance of such applications. Offloading application-specific network functions to peripheral devices can improve performance and reduce host CPU utilization. Due to the peculiarities of each particular device and the difficulty of programming an outboard CPU, the need for an abstracted offloading framework is apparent. This paper proposes a novel offloading framework, called* HYDRA *that enables utilization of such devices. The framework enables an application developer to design the offloading aspects of the application by specifying an "offloading layout", which is enforced by the runtime during application deployment. The performance of a variety of distributed algorithms can be significantly improved by utilizing such a framework. We demonstrate this claim by evaluating several offloaded applications: a distributed total message ordering algorithm and a packet generator.*

1. Introduction

The development of distributed computing applications is very challenging. When messages may be lost, corrupted or delayed, robust algorithms must be used in order to build a coherent system. Distributed algorithms rely on interchanging messages among compute nodes. The processing of the network protocols consumes a significant amount of a server's CPU resources, which directly affects the performance of such algorithms.

Many previous and ongoing approaches aim to improve network processing by offloading parts of the *protocol* to the networking devices, as distinct from modifying or offloading the application. Current efforts have centered on specialized TCP Offload Engine (TOE) devices [6] that implement parts of the TCP/IP networking stack. TOE devices perform well for specific types of applications, but do not provide the expected performance gains for many kinds of networking and distributed applications [16].

The proliferation of *programmable* peripheral devices for computer systems opens new possibilities for research into alternate sources of performance improvement [25]. The capabilities offered by the high-performance microprocessors available in disk controllers, network interface cards and graphic accelerators, are extremely underutilized today. This paper proposes HYDRA, a generic offloading framework that enables a developer to utilize programmable peripherals to improve application performance.

Today, there is no generic programming model and runtime support that enables a developer to design the *offloading* aspects of an application. This work involves the design and implementation of a framework to address these challenges. We introduce the concept of an "*offloading layout*" as an additional phase in the process of an application development. After designing the application's logic, the programmer will design the offloading layout using a generic set of abstractions. The layout describes the interaction between the application and the offloaded code at various phases, such as deployment, execution and termination.

The rest of this paper is organized as follows. Section 2 describes the related work concerning offloading, Section 3 describes the HYDRA programming model. Section 4 presents HYDRA's software architecture. Section 5 discuss several distributed algorithms that may benefit from the proposed framework and offloading capabilities. Section 6 provides some case studies of using HYDRA, and Section 7 concludes the paper.

This paper will appear at the 2007 IEEE International Parallel and Distributed Processing Symposium. Distribution of this paper is prohibited.

2. Related Work

Offloaded applications have been designed for particular needs in the past using specific devices. Some of this work has led to the availability of near-commodity products. We discuss the previous work in three subsections according to the device type.

2.1. Storage Offload

Object Storage Devices (OSD) came from a research project called Active Disks from CMU [19], which influenced a recent OSD standardization by the ANSI T10 group. OSD is a protocol that defines higher-level methods for the creation, writing, reading and managing of data objects on a disk. Implementing OSD requires a high degree of processing capability at the disk controllers or the devices themselves and can offer the potential for extension by custom programmability at the device. One example of a storage-specific extension is the Diamond system [12]. Unlike traditional architectures for exhaustive search in databases, where all of the data must be shipped from the disk to the host computer, the Diamond architecture employs "early discard." Early discard is the idea of rejecting irrelevant data as early in the pipeline as possible. By exploiting active storage devices, one can eliminate a large fraction of the data before it is sent over the interconnect to the host. Diamond applications can install filters at the active disk for eliminating data.

2.2. Network Offload

One of the more fruitful areas for exploiting programmable devices is in the area of networking. As wire speeds increase and demand extensive host processing power, moving some of the work to the network card becomes an attractive alternative.

Spine [8] is a safe execution environment that is appropriate for programmable Network Interface Cards (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 a few major limitations. In particular, it requires an event-driven programming model and does not include a handler deployment process nor a framework for design of offloading aspects in the host application.

Arsenic [17] is a Gigabit Ethernet NIC program that exports an extended interface to the host operating system. 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) [20] system is a zero-copy and OS-bypass messaging layer for Gigabit Ethernet. EMP protocol processing is done at the NIC and a host application (usually through an MPI library) can directly manipulate the NIC. Arsenic and EMP provide very low message latency and high throughput but are very task-specific and lack the support for generic offloading or host application integration.

TCP Offload Engines (TOE) [6] are adapters that move some of the TCP/IP network stack processing out of the main host and into a network card. 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. Practical concerns such as the inability to modify TOE behavior for evolving TCP protocol changes or to implement non-trivial firewalls also limit the utility of non-programmable TOEs. Other approaches to reducing network processing overheads are possible as well. iWARP [18] is an approach that takes advantage of remote direct memory access and processor offload to increase throughput and reduce host overhead. iWARP network cards conceptually include TOEs and other functionality needed to implement the higher-layer protocols.

Previous research has also considered using programmable components to accelerate network processing in specific situations [9, 15]. Our goal in this work is to enable more general access to programmable components for arbitrary networking, computing or I/O tasks.

2.3. Computation Offload

Specific devices to assist a host processor with some of its computational burdens have existed for many years and seem to be experiencing a recent resurgence. Field-Programmable Gate Arrays (FPGAs) in particular are available as add-in PCI cards and integrated into supercomputer systems. Each FPGA vendor provides varying level of support for the development of host applications and device programs ranging from a single high-level language and auto-generating compilers down to explicit device gate design. What is lacking in FPGA development is any generic interface or commonality that would enable applications to run on platforms other than where they were developed. Also the communication models for FPGAs are typically primitive compared to the networking and storage examples described above. Our HYDRA approach is potentially very well suited to FPGA devices.

3. HYDRA Programming Model

Our proposed programming model enables one to develop an "*Offload-Aware (OA)*" application by using a set of special components called *Offcodes*. An offcode defines the minimal unit for offloading and exports a set of well-defined interfaces. Offcodes are interconnected via "communication channels" that determine various communication properties between them.

We follow the "layout programming" design methodology first presented in FarGo [10, 11] and then in FarGo-DA [24]. Although not dealing with offloading, FarGo and FarGo-DA propose a programming model that enables a developer to program relocation and disconnection semantics between components in a separate phase during the application development cycle.

Similarly, OA-applications are designed by two orthogonal aspects. One aspect defines the basic logic of the application. Components which are potential candidates for offloading are identified and tagged as *Offcodes*. In the second aspect, the offloading constraints of the application are defined. In this phase, mapping between components and peripheral devices, both in software and hardware, is set, including the offloading priorities and channel characteristics among offcodes, and between offcodes and the host.

3.1. Offcode

An offcode defines the minimal unit for offloading. Offcodes can be provided as source code, which is then compiled for the target device, or as pre-compiled binaries. An offcode is further described by an Offcode Description File (ODF) that describes the offloading layout constraints and the target device hardware and software requirements.

An offcode can present multiple interfaces, each of which contains a set of methods that perform some behavior. Each interface is uniquely identified by a globally unique identifier (GUID). An OA-application communicates with an offcode using an abstraction called a *Channel* (described in Section 3.2). All offcodes implement a common interface that is used by the runtime to instantiate the offcode and to obtain a specific offcode's interface.

3.1.1 Offcode Creation

Offcodes are created by an OA application by calling the *CreateOffcode* method provided by the HYDRA runtime environment. The runtime generates and uses an offloading layout graph to offload the OA-application's offcodes. Section 3.4 details the mechanism used for the mapping of offcodes to their respective devices. Once the offcode is constructed at the target device, it is initialized and executed by the HYDRA runtime. Offcode initialization is performed in two phases. First, the *Initialize* method is called and the offcode acquires its *local* resources. Once all the related offcodes specified by the layout graph have been initialized, the *StartOffcode* method is called.

Figure 1 presents an offcode deployment process that is executed by the runtime. The OA-Application running on the host creates a single offcode α that requires a second offcode β. Since the offcode is automatically created, the runtime constructs an offloading-layout graph (Section 3.3) and performs the actual offloading process.

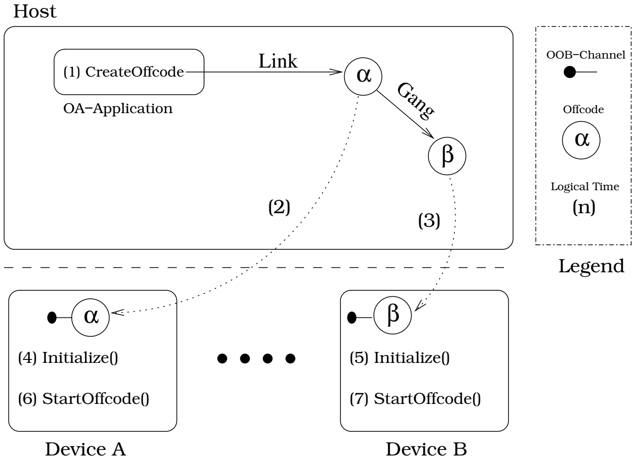

Figure 1. Offcode Deployment

Once an offcode has been explicitly created, a set of attributes can be applied to it. HYDRA provides an API to get and set offcode attributes. There are several attributes already defined, including OBSOLETE_TIME, WATCHDOG_TIME and OFFLOAD_PRIORITY. The latter can be used to affect the offloading sequence, as will be further elaborated in Section 3.3.

3.1.2 Offcode Invocation

HYDRA provides two ways to invoke an offcode: transparently and manually. A transparent invocation requires a proxy component that shields the client from the complexity involved in invoking the target offcode directly. The proxy has a similar interface as the target offcode and allows the client program to invoke an offcode as if the offcode were a local component. When a user creates an offcode, a proxy object is loaded into user-space. All interface methods return a *Call* object that contains the relevant method information including the serialized input parameters. Once a *Call* object is obtained, it can be sent to a target device (or several devices) by using a connected channel. The manual invocation scheme consists of manually creating the *Call* object, and using a custom encoder to marshal arguments and invoke the channels' methods.

3.2. Channels

Offcodes are connected to each other and to the host application by communication *channels*. Channels are bidirectional pathways that can be connected between two endpoints, or connectionless when only attached to one endpoint. The runtime assigns a default connectionless channel, called the *Out-Of-Band Channel (OOB-channel)* for every OA-application and offcode. The OOB-channel is identified by a single endpoint used to communicate with the offcode without the need to construct a connected channel, such as for initialization and control traffic that is not performance critical. The OOB-channel is the default communication mechanism between peer offcodes and between offcodes and OA-applications. The OOB-channel is also used to notify the offcode regarding management events and availability of other channels.

3.2.1 Channel Creation

For high performance communication, a specialized channel that is tailored to the needs of the application and the offcode would be created as well as the default OOB-channel. Creating a specialized channel is performed in two steps. First, the application or offcode determines the channel characteristics and creates its own endpoint. Next, the creator attaches an offcode to the channel. This action implicitly constructs the second endpoint at the target device, and notifies the offcode about the newly available channel. Once the channel is connected, the channel's API can be used for communication. The channel API contains typical operations to read, write and poll. The channel API also supports registration of a dispatch handler that is invoked each time the channel has a new request.

Channel creation involves configuring the channel type, synchronization requirements and buffer management policy. A channel can be of type *Unicast*, that can only interconnect two offcodes, or *Multicast*, that can interconnect more than two offcodes. A channel can be either unreliable or reliable, where the latter type is careful not to drop messages even though buffer descriptors are not available. A multicast channel can utilize hardware features, if available, to broadcast a single request to multiple recipients.

3.3. Offload Layout Programming

The offloading layout is usually statically defined or set during deployment (See Section 3.4) to minimize the overhead of offloading operations. As opposes to FarGo's primary motivation of enabling the dynamic relocation of distributed components (See 3), we envision the offcodes as specialized components performing one task on a specific device, thus purposefully do not implement offcode migration, for instance.

Channel constraints are used to direct the placement of offcodes on target devices when multiple offcodes are required to support an application. HYDRA currently supports the following constraint types:

- ***Link Constraint***: The Link constraint is the default basic channel constraint between two offcodes. It does not require that they run on the same or different target devices, just that both be present in the system.
- ***Pull Constraint***: The Pull constraint ensures that both offcodes will be offloaded to the same target device.
- ***Gang Constraint***: The Gang constraint is used to ensure that both offcodes will be offloaded to **their** target devices, respectively.

An OA-Application can also influence layout by setting the offload priority for each offcode that it directly requires. Once a reference priority is defined, it is inherited by subsequent offcodes required by the top-level offcode until a Link reference is encountered.

3.4. Offcode Description File

An offcode description file (ODF) summarizes the available offcode interface functions and required hardware capabilities. An ODF contains three parts: first, the structure of the offcode's package and required files on the host. The second part defines the target device's hardware. The last part declares software interfaces used in its implementation that should be defined in the target device's execution environment. Currently, all required interfaces must be defined by a GUID (much like offcodes themselves). The basic runtime interfaces defined by HYDRA are available to all offcodes without an explicit interface requirement. Figure 2 presents a snippet from a typical ODF, containing the three sections just described, including the use of a *Pull* constraint to specify a peer offcode.

4. HYDRA Software Architecture

In the previous section we introduced the programming model, focusing on the separation between application logic programming and offload-layout programming. In this section we present the design of the runtime system. The system implements the model and provides facilities for programming, testing, deploying, and managing OA-applications and offcodes. Both the host OS and the target device firmware must support the interfaces defined by the programming API and implement the runtime functionality.

Runtime library requirements for a particular target device may be provided by the device manufacturer, system integrator, or by application developers themselves. The

```
<offcode bindname="Hydra.net.utils.NetAPI">
 <GUID>6060842</GUID>

 <!-- offcode package info -->
 <package>
  <device>   <!-- offload device -->
   <id>0x0001</id>
   <file>/lib/offcodes/NetLib.oc</file>
   <!-- a netlib stub for this device -->
   <file>/lib/stubs/NetLibStub.oc</file>
  </device>
   <host>   <!-- host proxy -->
     <os>Linux FC4</os>
     <ver>2.6.10</ver>
     <file>/lib/proxy/NetLibProxy.so</file>
   </host>
 </package>

 <!-- software environment section -->
 <sw-env>
   <import>
     <reference type=Pull pri=0>
     <bindname>Hydra.net.utils.Checksum</bindname>
     <GUID>6060843</GUID>
     <file>"/lib/offcodes/checksum.oc"</file>
   </import>
 </sw-env>

 <!-- hardware environment section -->
 <hw-env>
   <hydra-device id=0x0001>
     <name>Netgear GA-620T</name>
     <vendorID>0x1385</vendorID>
     <deviceID>0x620A</deviceID>
     <bus>pci</bus>   <!-- (optional) -->
     <address>0x0011</address>
  </hydra-device>
 </hw-env>
</offcode>
```

Figure 2. Sample offcode description file

second half of the runtime system exists on the host as operating system extensions. Our host implementation for Linux is modular, in that it maintains strict separation between device-specific code and generic code. It is implemented as a set of loadable kernel modules, requiring no kernel source code modifications.

4.1. HYDRA Runtime

The HYDRA runtime is comprised of several components as shown in Figure 3. It is accessed through an offloading access layer that consists of a user-level library linked to each OA-Application, and a kernel-level set of services.

The kernel layer consists of several functional blocks. The *System Call Management* and *Offloading API* blocks implement the various APIs defined in the programming model. The *Channel Management* unit manages the channels by interacting with the *Channel Executive*. This module handles channel creation by using a particular *Channel Provider*. These providers are target-specific and provided as an extended driver for each programmable device. A channel provider creates various specialized channel types to the device and provides a cost metric regarding the "price" for communicating with the device through a specific channel, in terms of latency and throughput. The executive uses this capability information to decide on the best provider for a specific offcode. The *Resource Management* unit keeps track of all active offcodes and related resources. Resources are managed hierarchically to allow for robust clean-up of child resources in the case of a failing parent object. The *Memory Management* module exports memory services such as user memory pinning that is used by zero-copy channels. The *Layout Management* unit performs layout related functionalities such as analyzing the offloading layout graph.

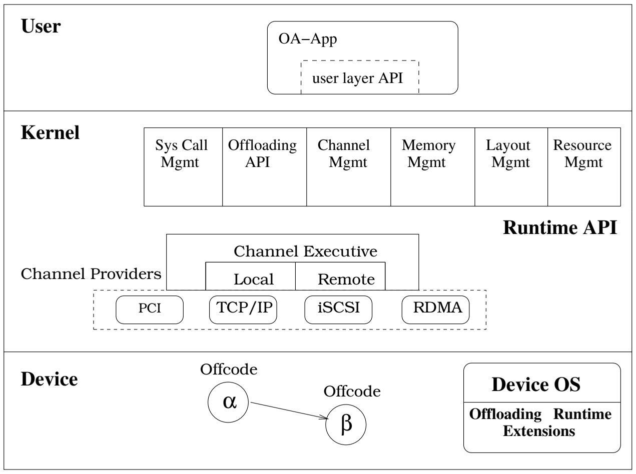

Figure 3. System Architecture

4.2. Offcode Dynamic Loading

Supporting dynamic offcode loading is an important building block in the HYDRA framework. We have considered different approaches for implementing dynamic loading. The simple solution would be to hand over the offcode to the target device and require that each device implement a simple offcode loader. However this naive solution is quite expensive in terms of device resources. Another approach would be to fully perform the linking process at the host, and only transfer the offcode when it is ready to be deployed (at a specific memory region). The device's loader will merely need to initialize the offcode and execute it.

HYDRA runtime is built to support both approaches. HYDRA support for dynamic offloading is provided by a set of device-specific loaders that implement a generic interface for offcode loading. The interface is intended to be implemented by the device driver of each target peripheral. Each loader can decide whether to transfer the offcode as is, or to perform some processing at the host first, depending on features of the target.

The loader for our programmable network card is implemented both in the device and in the host. A device-specific host-based loader is implemented in the NIC's driver; it uses the OOB channel time to communicate with the target device loader. Four message transfers are used to load a single offcode. Once the host-based loader calculates the offcode's size, it asks the device's loader to allocate memory for it. The runtime loader does so and returns the device's memory address. The host dynamically generates a linker file adjusted by the returned address and links the offcode object. It then transfers the linked offcode to the target device where it is placed and executed.

5. Application Scenarios

Distributed algorithms are often designed to function correctly despite unpredictable and unreliable infrastructures. Following is a partial list presenting some potential building blocks that are used by many distributed applications that can benefit from the offloading capability offered by HYDRA.

Network Oriented Components. Distributed applications operate by interchanging messages among nodes. The message exchange networking protocols are potential candidates for offloading. For example, the reliable broadcast service that ensures that all hosts in a group of nodes deliver the same set of messages to the application layer can be easily offloaded to the networking device. This service can be used as a building block to construct value-added multicast services, such as agreement and total ordering, or it can be utilized to support the applications that involve groups of cooperating hosts.

Total Order. Many distributed algorithms need guarantees on message order. Section 6.2 provides a detailed discussion of our case-study of offloading such a protocol to the networking device. Another example of an ordering protocol that can be easily offloaded is the token-ring protocol used in the Totem [1] system.

Virtual Synchrony. The virtual synchrony model [4] offers stronger guarantees required by applications such as replicated database systems. The implementation overhead involved can be drastically reduced by offloading the critical components to the networking card.

Cluster Synchronization. Real-time guarantees can be implemented on programmable peripheral devices [23] and used as a building block for a variety of distributed applications. For instance, the work of Verissimo et al. [21] presents a set of distributed algorithms that assume the existence of a Timely Computing Base. Having such a component simplifies the design complexity of these algorithms. This timely component is an ideal candidate for offloading, as it exports a simple interface that is ideal for a programmable clock, network card, or encryption engine.

Self-Stabilizing Algorithms. These algorithms are designed to return a system to normal functioning, irrespective of the severity and nature of transient failures, as long as there is a sufficiently long time interval for convergence. Offloading some of the functionality can significantly reduce convergence time. E.g. due to the higher reliability of the NIC, the self-stabilizing algorithm may significantly decrease the time required to trust the coherence of the received messages by verifying them with the NIC.

Distributed File Systems. Networked storage can use offloading to enhance application performance by moving common functions to an assist device. While RDMA-capable networks can successfully bypass the operating system for bulk transfers, other protocol activities such as cache validation can generate message and interrupt overhead to a host.

Providing a toolbox of reusable offcodes that implement a variety of distributed algorithms may simplify the development and deployment of distributed systems. We argue for the need of such a toolbox, with proven correctness and performance guarantees. Future work should provide such components. The next section presents several of our case study components used in evaluating HYDRA.

6. HYDRA Evaluation

In the previous sections we described the HYDRA system, including its programming model and its internal design. In this section we demonstrate the use of HYDRA through several sample applications.

6.1. Traffic Generator

Generating steady network traffic at high rates is difficult given the variety of sources of delays and unpredictability in a modern computer system, including devices' interrupts, cache and TLB misses, and power management changes. We implemented an offload-aware traffic generator that produces a packet stream with fixed inter-packet delays. We evaluate the performance of this application and compare the results with an equivalent user-level application.

The traffic generator is comprised of two components: a GUI that is used by the user to setup the system, and a *StreamGenerator* component that generates the stream of packets given user settings on protocol type, length, ports, inter-packet delay, burst size, etc. The *StreamGenerator* component is designed as an offcode. The GUI is the offcode's controller and creates a specialized, zero-copy, channel for communication. The APIs for interaction between the GUI and the *StreamGenerator* offcode ore omitted here for brevity, as are details of the offcode description file.

We implemented the application once using HYDRA and once without the use of an offloaded component. We evaluate the designs using two hosts, Intel Pentium 4 2.4 GHz with 512 MB and a Tigon2 programmable network card, interconnected by a 100 Mb/s switch. We attempt to fully utilize the link capacity by generating packets at fixed inter-packet delays and for different frame sizes.

User-Space Traffic Generator

The benchmark results for the user-space application are given in Table 1. Although the achieved throughput is quite good, the dispersion of the inter-arrival times is enormous, so large as to make the average almost meaningless. Figure 4 shows the Cumulative Distribution Function (CDF) for three packet sizes to better display the distribution of arrival times and illustrate the wide dispersion in these measurements.

Size	*Tput*	*Avg. Arrival ± Std*	*CPU ± Std*
Bytes	*Mb/s*	*µs*	*%*
64	6.0	140 ± 8 000	100 ± 3
80	13.4	141 ± 9 000	99 ± 7
96	21.8	159 ± 11 000	99 ± 8
192	56.8	164 ± 6 000	98 ± 11
384	96.7	175 ± 4 000	81 ± 11
768	97.8	205 ± 4 000	37 ± 28
1514	98.6	244 ± 5 000	33 ± 5

Table 1. User Space Traffic Results

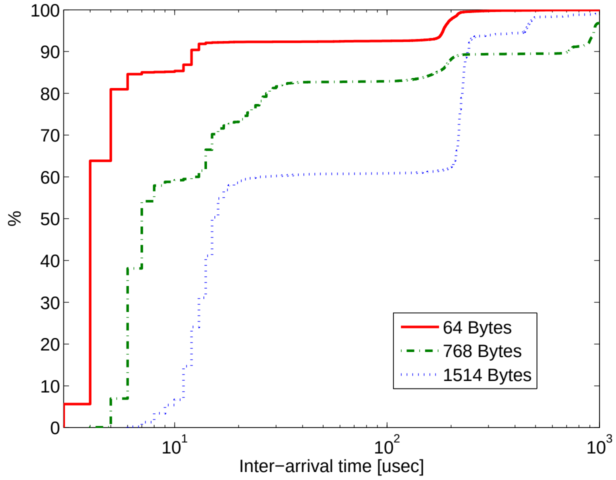

Figure 4. User-Space Traffic Distribution

It is also evident from the table that delivering the generated data to the application is difficult due to the very high CPU load, especially with small packet sizes. The processor capacity problem, driven by the costs associated with interrupts, directly impacts the throughput seen by the applications. As an example, the calculated inter-arrival times for 1500 byte ethernet frames is approximately 120 µs for 100 Mb/s, 12 µs for 1 Gb/s and 1.2 µs for 10 Gb/s ethernet. We have observed that the interrupt overhead for an empty interrupt handler is between 5–10µs, consuming all but only 17% of the total available CPU cycles.

Size	*Tput*	*Avg. Arrival ± Std*	*CPU*
Bytes	*Mb/s*	*µs*	*%*
64	23.9	34 ± 6	2
$64^{\star}$	51.5	16 ± 8	2
768	98.4	65 ± 13	2
1514	98.8	126 ± 50	2

Table 2. Offload-Aware Traffic Results

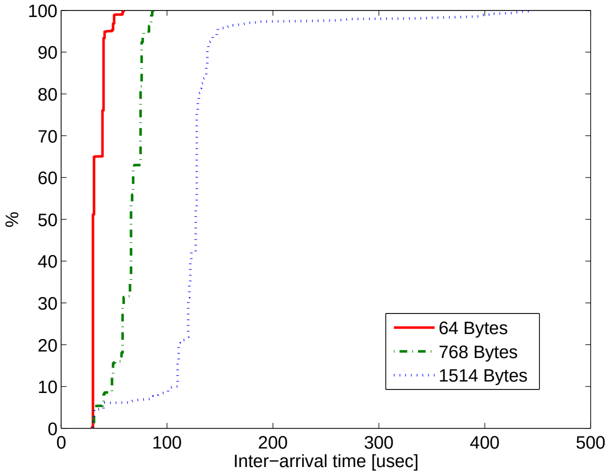

Figure 5. Offload-Aware Traffic Distribution

Offload-Aware Traffic Generator

The results from the offload-aware traffic generator are summarized in Table 2 and shown as a CDF in Figure 5. For both tests, in order to accurately measure the throughput and the inter-arrival times, we have used a second NIC with a simple traffic analyzer offcode. The data shows that the inter-arrival times are uniform with small standard deviation. The sharp vertical edges in the CDF indicate that the majority of the packets arrived within the same expected inter-arrival time. Notice that for 64-byte packets, the achieved throughput is only a quarter of the link's bandwidth. In order to achieve the full link capacity, a generator must produce a 64-byte packet approximately every 5 µs. Because we have not tried to optimize the HYDRA runtime for this or any specific application, the generator can only send packets at a rate limited by the device's OS constraints, which in this case is limited by the number of MAC descriptors at the NIC and the processing overhead involved in managing them. In order to further improve the throughput for such small packets, we have created an optimized version of the device's OS that can reuse a single MAC descriptor for sending the same packet multiple times. The table shows that for the optimized version (indicated by the $64^{\star}$ table entry) the throughput has been significantly improved. This sort of optimization may be undertaken as needed by particular applications that use HYDRA.

6.2. Total Ordering

Total Order (TO) algorithms have been extensively studied in the literature [7]. A TO algorithm is a fundamental building block in the construction of distributed fault-tolerant applications. They are typically used to provide a communication primitive that allows processes to agree on the set of messages they deliver and also on their delivery order. Total ordering is particularly useful for implementing fault-tolerant services, database replication and locking services [2]. A TO algorithm that assumes an unreliable failure detector is equivalent to the consensus problem [5]. It has been shown that consensus cannot be solved in this type of systems in fewer than two communication steps [13]. Many TO algorithms for asynchronous systems use consensus as a building block, but the implementation can be expensive both in terms of communication steps and number of messages exchanged between hosts. This overhead is further exacerbated if in addition to the TO algorithm, the host also executes a resource-demanding application such as a typical High Performance Computing (HPC) application.

Offloading a TO algorithm, either in full or for particular components, can greatly improve the performance of distributed applications for several reasons. First, a TO algorithm packaged as an offcode can be easily reused by a variety of applications. Second, the reduced load on the host machine will improve the performance of such applications; and third, an offloaded TO may take advantage of specific hardware capabilities in order to improve its overall performance. For example, as shown in the traffic generator example above (Section 6.1), the small dispersion of the inter-arrival times of ethernet packets may be used to implement better accurate failure detectors and to maintain finer-grained timeouts for message retransmissions.

6.2.1 Offload-Aware TO Architecture

We have implemented a simple offload-aware total order application. To simplify the proof of concept implementation, we assume that there are no physical link disconnections, switch failures, or process or node crashes. We do not assume a reliable message transmission—messages can be lost due to buffer overflow at the NIC, host or switch. The sample application is comprised of several components that appear on the left side of Figure 6.

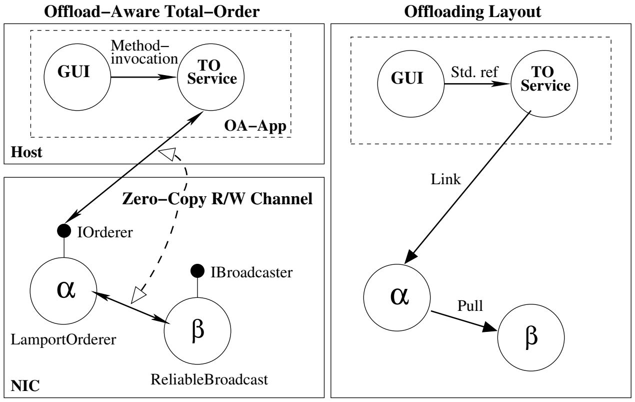

Figure 6. Total-Order Offload Architecture

1. *GUI*: The Graphical User Interface controls the TO application. It enables the user to define the rate at which messages are transmitted and their size. The GUI presents the message order once it is determined.

2. *TO Service*: The TO Service is an application library used by the GUI, that in turn uses the offcode to provide two basic total-order APIs: *TO_Broadcast* and *TO_Receive*. The first API broadcasts a message and the second receives the next message for which the TO has been established.

3. *LamportOrderer*: This offcode (denoted by the letter α) presents the *IOrderer* interface that implements a TO algorithm. Specifically, we have implemented the Lamport's Timestamp ordering algorithm [14]. This offcode interacts with the *TO Service* in a well defined interface, discussed below.

4. *ReliableBroadcast*: This offcode (denoted by β), provides the reliable broadcast service that is needed by the *LamportOrderer* offcode. In our implementation, multicast is used in order to efficiently send messages to peer hosts. Albeit our simplifying assumptions, message omissions may still occur due to buffer overflow. To address this issue, this component implements a simple negative acknowledgment scheme.

The left side of Figure 6 also indicates the HYDRA communication channels that are used. A reliable unicast channel with a zero-copy policy for read and write is used in order to eliminate the OS networking stack overhead. Basically, the *TO Service* manages the application's memory descriptors (See Section 3.2) and effectively determines the control-flow policies of the application (descriptors for received messages are also posted by this component). In order to send a message, the *TO Service* creates a *Call* object and invokes the channel. The NIC-resident HYDRA runtime DMAs the message and notifies the *LamportOrderer* offcode that a new message should be transmitted. The "orderer" offcode timestamps the message and multicasts it using the *Broadcaster* interface, which is implemented by the *ReliableBroadcast* offcode.

Received packets are first handled by the *ReliableBroadcast* offcode. The offcode is operated in two phases: At the first phase, the offcode transfers the received packet to a pre-posted descriptor at the host using DMA. Note that the message cannot be delivered to the application yet, since the message order has not been determined. Because the NIC has a small amount of memory, it is better to release the NIC's memory as soon as possible. The message identifier and timestamp are the only data that is saved on the NIC by the *LamportOrderer* offcode. The second phase begins once the message order has been determined by the TO algorithm. The *LamportOrderer* offcode creates a *Call* with the messages' order and invokes the channel connected to the *TO Service*. Once the order is known at the *TO Service* component, the ordered messages can safely be delivered to the application.

The right side of Figure 6 presents the offloading layout that is designed by the developer. The *GUI* holds a standard reference to the *TO Service* component. This component holds a *Link* reference to the "orderer" components α, since it has no special offloading constraints. On the other hand, the "orderer" offcode must be offloaded *with* the broadcast offcode (i.e, β) hence a *Pull* constraint is used. Note that in order to compare the results of this offload-aware TO algorithm with a non-offloaded version, a developer merely needs to interchange the two constraints and re-execute the application. The effect of doing so is that the "orderer" will be executed at the host while the broadcaster remains at the networking device.

6.2.2 Total Ordering Evaluation

We used five Intel Pentium 4 2.4 GHz systems, with 512MB of RAM and 32-bit, 33 MHz PCI bus. Each machine was equipped with programmable Netgear 620 NICs, which have 512 kB of memory. We used Linux version 2.6.11 with the HYDRA module enabled. The hosts were interconnected by a Gigabit ethernet switch (Dell PowerConnect 6024). The right side of Table 3 presents the maximum throughput and latency measurements for the offload-aware TO when all nodes act as both senders and receivers. Each node generates traffic at a rate bounded by the flow control mechanism imposed by the *TO Service* component. The presented latency is defined as the time elapsed between the *TO_Broadcast* and *TO_Receive* method invocations that refer to the same message.

We compare our results with those from a recent work by Dolev et al. [3], which are given on the left side of the table with title "Hardware-based TO". That work implements a wire-speed total order algorithm using hardware-based component comprised of two switches connected back-to-back. Each host is equipped with two NICs: one NIC is used for transmitting (connected to the first switch) and one for receiving (connected to the second switch). The back-to-back switch connection serializes the packets, thus effectively acts as a hardware sequencer. In addition to the switch configuration, a lightweight user-space TO algorithm is invoked at each node. The *throughput* obtained from the offload-aware TO application is close to that of the hardware-based solution. Note that the throughput increases with the number of nodes due to PCI bus properties as explained in previous work [3, 22].

Although with HYDRA we have used a *software* algorithm to order the messages, we found that bypassing the OS networking stack overhead enabled us to significantly increase the throughput over typical user-based total ordering. This fact strengthens the motivation for offloading and specifically for using HYDRA.

As for the measured latency, the results are approximately twice those in the hardware-based configuration. Although we have offloaded the ordering algorithm to the NIC, a distributed solution requires an extra round of communication that is not required in centralized solutions (like the hardware-based solution). In addition, Lamport's timestamp algorithm is known to be very expensive in terms of communication overhead and latency; messages must be received from every node in order to be able to determine the messages' order. We expect to improve the performance by deploying more efficient ordering algorithms.

7. Conclusions and Future Work

This paper has presented HYDRA—a novel framework for building high-performance distributed applications that can benefit from offload capabilities of modern peripherals. HYDRA proposes a new dimension of flexibility for the architects of distributed applications: the ability to program offloading layout policies separately from the application's logic. We have developed a programming model that carefully balances between programmer scalability and system scalability. We believe that programmable devices will continue to grow in popularity. The need for a framework such as HYDRA is to enable the use of these devices to improve performance and capability of a broader range of applications. We have evaluated HYDRA by implementing several applications and discussed its potential use for accelerating distributed computing. In the future we intend to provide a toolbox of offcodes consisting of reusable building blocks that will suit a variety of distributed applications. We expect to release an experimental version of HYDRA towards the end of this year.

Nodes	Hardware-based TO		Offload-Aware TO	
	Throughput [Mbps]	Latency [ms]	Throughput [Mbps]	Latency [ms]
3	310.5	4.2	301.8	8.7
5	362.5	4.1	324.6	9.5

Table 3. TO Performance (all-to-all)

Acknowledgments

We would like to thank Ariel Daliot and Ilya Shnayderman, members of the DANSS group at the Hebrew University of Jerusalem, for their careful reading and thoughtful comments. Special thanks to Ophir Holder, the forefather of the FarGo project, who have provided his valuable insights regarding the programming model.

References

[1] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, 1995.

[2] Y. Amir and C. Tutu. From total order to database replication. In *ICDCS '02*. IEEE Computer Society, 2002.

[3] T. Anker, D. Dolev, G. Greenman, and I. Shnayderman. Wire-speed total order. In *IPDPS'06*, 2006.

[4] K. Birman, R. Constable, M. Hayden, C. Kreitz, O. Rodeh, R. van Renesse, and W. Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *DISCEX 00*, 2000.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, pages 225–267, 1996.

[6] A. Currid. TCP offload to the rescue. *Queue*, 2(3):58–65, 2004.

[7] X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[8] M. E. Fiuczynski, R. P. Martin, T. Owa, and B. N. Bershad. Spine: a safe programmable and integrated network environment. In *EW 8*, 1998.

[9] P. Gilfeather and A. B. Maccabe. Modeling protocol offload for message-oriented communication. In *Cluster 2005*, 2005.

[10] O. Holder, I. Ben-Shaul, and H. Gazit. Dynamic layout of distributed applications in fargo. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 163–173, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[11] O. Holder, I. Ben-Shaul, and H. Gazit. System support for dynamic layout of distributed applications. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99)*, pages 163–173, Austin, TX, May 1999.

[12] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *FAST '04*, 2004.

[13] Keidar and Rajsbaum. On the cost of fault-tolerant consensus when there are no faults. *SIGACT News*, 2001.

[14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[15] A. B. Maccabe, W. Zhu, J. Otto, and R. Riesen. Experience in offloading protocol processing to a programmable NIC. In *Proc. of IEEE International Conference on Cluster Computing*, 2002.

[16] J. C. Mogul. TCP offload is a dumb idea whose time has come. In *HotOS*, pages 25–30, 2003.

[17] I. Pratt and K. Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *INFOCOM*, pages 67–76, 2001.

[18] R. Recio, P. Culley, D. Garcia, J. Hilland, and B. Metzler. An RDMA protocol specification. `http://www.ietf.org/internet-drafts/draft-ietf-rddp-rdmap-04.txt`, 2005.

[19] E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *VLDB*, pages 62–73, 1998.

[20] P. Shivam, P. Wyckoff, and D. Panda. Emp: zero-copy os-bypass nic-driven gigabit ethernet message passing. In *SC'01*, New York, 2001. ACM Press.

[21] P. Verssimo and A. Casimiro. The Timely Computing Base model and architecture. *IEEE Transactions on Computers*, 51(8), 2002.

[22] W. Wadge. Achieving gigabit performance on programmable ethernet network interface cards. B.Sc. Final Report, University of Malta, 2001.

[23] Y. Weinsberg, T. Anker, D. Dolev, and S. Kirkpatrick. On a NIC's operating system, schedulers and high-performance networking applications. In *HPCC-06*, 2006.

[24] Y. Weinsberg and I. Ben-Shaul. A programming model and system support for disconnected-aware applications on resource-constrained devices. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 374–384, New York, NY, USA, 2002. ACM Press.

[25] Y. Weinsberg, D. Dolev, P. Wyckoff, and T. Anker. Hydra: A novel framework for making high-performance computing offload capable. In *Proceedings of 31st IEEE Conference on Local Computer Networks (LCN 2006)*, Tampa, Florida, USA, November 2006.