HYDRA: A Novel Framework for Making High-Performance Computing Offload Capable

Yaron Weinsberg* Danny Dolev†
The Hebrew University Of Jerusalem

Tal Anker‡
RadLan - Marvell

Pete Wyckoff§
Ohio Supercomputer Center

Abstract

The proliferation of programmable peripheral devices for computer systems opens new possibilities for academic research that will influence system designs in the near future. Programmability is a key feature that enables application-specific extensions to improve performance and offer new features. Increasing transistor density and decreasing cost provide excess computational power in devices such as disk controllers, network interfaces and video cards.

This paper proposes an innovative programming model and runtime support that enables utilization of such devices by providing a generic code offloading framework. 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 framework also provides the necessary development tools and programming constructs for developing such applications.

We test our framework by implementing a packet generator on a programmable network card for network testing. The offloaded application produces traffic at five times the rate, and with inter-packet variability that is many orders of magnitude smaller than the non-offloaded version.

1 Introduction

According to the International Technology Roadmap for Semiconductors, by 2007 one million transistors will cost less than 26 cents. This ongoing trend of decreasing cost and increasing density of transistors motivates hardware and embedded system designers to use programmable solutions in their products. Such designs are cheaper and more flexible than custom ASIC solutions. Performance capabilities of programmable products, and microprocessors in particular, will extend well up into the range of applications that formerly required DSPs or custom hardware designs.

*Email: wyaron@cs.huji.ac.il

†Email: dolev@cs.huji.ac.il, part of the research was done while visiting Cornell University

‡Email: tala@marvell.com

§Email: pw@osc.edu

This paper considers the model where applications execute cooperatively in the host processor as well as in device peripherals. In this model, applications can delegate tasks to devices with various architectures and constraints. Using programmable devices has traditionally been very difficult, requiring experienced embedded software designers to implement conceptually simple tasks. Interfacing a new device feature with the host operating system would be performed from scratch and customized for the particular design. The availability of cross-compilation tools and remote debugging environments are making the programming tasks simpler, but integration with the host operating system is still very difficult.

We introduce the concept of an "*offloading layout*" as a new 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.

Currently, there is no generic programming model and corresponding runtime support that enable a developer to design the *offloading* aspects of an application. The current paper involves the design and implementation of a framework to address these challenges.

1.1 Programming Facets

The programming model is divided into two loosely-coupled facets.

1. *Application logic programming* — This is the mechanism of designing the basic logic of the application. *Offcodes* are provided as a set of reusable components from the vendor or custom made by the developer.

2. *Offload layout programming* — This task defines the mapping between components and peripheral devices, both in software and hardware. It also sets offloading priorities and channel characteristics between offcodes and the host.

One of the major challenges in designing such a model is in actually keeping the two facets separate, though complimentary. Programming the application logic should resemble programming a regular application. Programming the layout should

affect the application logic as little as possible. This separation is not possible today.

1.2 Offcode

An offcode defines the minimal unit for offloading. Offcodes are provided as source code, which needs to be compiled to the target device, or as a pre-compiled binary. An offcode is further described by an Offcode Description File (ODF) that uses XML to describe the offloading layout constraints and the target device hardware and software requirements.

An offcode can implement 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*. An offcode object file implements only one offcode, and it has a GUID that is unique across all offcodes. All offcodes implement a common interface that is used by the runtime to instantiate the offcode and to obtain a specific offcode's interface.

1.2.1 Offcode Creation

Offcodes are created by an OA application by calling the runtime *CreateOffcode* API. The runtime generates and uses an offloading layout graph to offload the OA-applications' offcodes. Section 1.5 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. Since peer offcodes may have not been offloaded yet, the offcode can access local resources only. Once all the related offcodes have been offloaded, the *StartOffcode* method is called. At this point inter-offcode communication is facilitated. 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 under, e.g. OBSOLETE_TIME and OFFLOAD_PRIORITY. The latter can be used to affect the offloading sequence.

1.2.2 Offcode Invocation

HYDRA provides two ways to invoke an offcode: transparently and manually. Achieving syntactic transparency for offcode invocation requires the use of some "proxy" element that has a similar interface as the target offcode. 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.

1.3 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 usually used to notify the offcode regarding management events and availability of other channels.

1.3.1 Channel Creation

The OOB-channel can be used for simple data transfer between the application and offcodes and among offcodes. For high performance communication, a specialized channel that is tailored to the needs of the application and the offcode can be created. Enabling a specialized channel is performed in two steps. First, the channel creator determines the channel characteristics and creates its own endpoint of the channel. Second, 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 of 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.

Creating a channel 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. Note that a multicast channel can utilize hardware features, if available, to send a single request to multiple recipients simultaneously.

1.4 Offload Layout Programming

The offloading layout is usually statically defined or set during deployment. The reasoning behind this is to minimize the overhead concerned with the offloading operations. We envision the offcodes as specialized components performing one task on a specific device. The overhead imposed by enabling migration of offcodes between devices is superfluous if this feature is rarely used. We intend in the future to support the rare migration of offcodes between target devices and the host kernel when required. Channel constraints are used to direct the placement of offcodes when multiple offcodes are required to support an application.

The collection of channel constraints and their related semantics are defined below.

Link Constraint: The Link constraint is denoted as $\alpha \overset{link}{\Rightarrow} \beta$. This is the default basic channel constraint from α to β, which actually posess no constraints: α and β may or may not be mutually offloaded (to the same or different target device).

Pull Constraint: The Pull constraint is denoted as $\alpha \overset{pull}{\Leftrightarrow} \beta$. This reference is used to ensure that both offcodes will be offloaded to the **same** target device. This definition implies several additional constraints. First, neither α nor β can be offloaded separately. Second, α can be a target of a Pull reference, i.e., $\delta \overset{pull}{\Leftrightarrow} \alpha$ making Pull transitive - offloading δ will offload α, and hence β.

Gang Constraint: Gang constraint is denoted as $\alpha \overset{gang}{\Leftrightarrow} \beta$. This constraint is used to ensure that both offcodes will be offloaded to **their** target devices, respectively. Gang constraint is also transitive and the only difference from Pull is that the offload target can be a set of devices instead of a single device.

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 *L*ink reference is encountered.

1.5 Offcode Manifesto

An offcode manifesto is the mean by which an offcode defines its requirements from a target device and peer offcodes. The manifesto is realized in an Offcode Description File (ODF). An ODF contains three parts: The first part describes the structure of the offcode's package, containing the binary code and other general properties. The second part defines the target device's hardware. The last part of the ODF concerns the software environment. The offcode declares the 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.

2 Experimental Results

In the previous sections we described the HYDRA programming model, in this section we demonstrate the use of HYDRA through a toy example application. We implemented an offload-aware traffic generator that targets to generate packets with fixed inter-packet delays. We evaluate the performance of this application and compare the results with an equivalent user-level application. We have integrated the HYDRA runtime into the original NIC's firmware.

2.0.1 Traffic Generator Evaluation

We have implemented the application once using HYDRA and once without the use of an offloaded component. We evaluated 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 tried 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 is 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.

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

Table 1. User Space Traffic Generator Results

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.

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

Table 2. Offload-Aware Traffic Generator Results

Offload-Aware Traffic Generator : The results from the offload-aware traffic generator are summarized in Table 2. The data shows that the inter-arrival times are uniform with small standard deviation. 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 every $\sim 10\mu$s. 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. This specific limit is related to the number of MAC descriptors at the NIC and the processing overhead involved in managing them (which is further exacerbated when dealing with small packets). 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^{opt} table entry) the throughput has been significantly improved. This sort of optimization may be undertaken as needed by particular applications that use HYDRA.