DYNAMIC OFFLOADING INFRASTRUCTURE FOR PROGRAMMABLE DEVICES

A Thesis Submitted in fulfillment
of the requirements for the degree of
Master of Science

by

Maxim Grabarnik

Supervised by

Prof. Danny Dolev

School of Engineering and Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel

November 2007

Acknowledgments

First, I would like to deeply thank my advisor Prof. Danny Dolev, for giving me opportunity to participate in fascinating research.

Special thanks go to Yaron Weinsberg, my co-advisor, for his efforts and support. I really enjoyed working together. I thank Yaron for being a huge inspiration source, for carefully reviewing my drafts and providing precious advice and helping to improve this thesis with his original ideas and insights.

Next, I thank Dr. Tal Anker who's ideas, advices and enthusiasm about this work helped to shape it.

Special thanks to my parents Irina and Alexander Grabarnik, for constantly boosting my moral.

Last but not least, I would like to thank my wife Ola for her patience during my studies.

Abstract

Traditionally, embedded systems, and computer peripherals in particular, serve very specific roles. They perform tasks statically hard-coded by a manufacturer lacking the ability to be configured dynamically with different functionalities.

Today, modern peripherals have substantial memory and computational resources. These resources could be partly utilized by user level applications if these devices would have supported dynamic offloading capabilities. Thus allowing applications to improve their performance and reduce their burden on the host machine.

Dynamic configuration has several inherent limitations. First, the linking process with a target device is tightly coupled with the device's code structure (e.g., exported methods) or runtime environment (such as virtual machines). Secondly, most offloaded components have restricted flexibility; and last, the devices may have heavy resource requirement for linking and loading.

This research describes a generic dynamic offloading framework allowing efficient deployment of arbitrary executable code into an embedded environment. It has very relaxed requirements from the target device CPU-wise and memory-wise. This framework has been deployed in the HYDRA framework [WDWAa] to facilitate its offloading aspects. The dynamic offloading framework has been implemented and evaluated for a network device on top of a Network Interface Card Operating System (NICOS [WADK06]), which was developed as a part of this research.

Table of Contents

Chapter 1

Introduction

1.1 Offloading Overview

Many computer peripherals are software driven. They are in fact programmable devices with some memory and CPU, programmed to achieve goals set by designer usually, but not necessarily, inspired by device category: network controllers send and receive packets, disk controllers read and write blocks of data, etc.

Programmable peripherals usually execute some kind of control software, however there are examples when richer software stacks are being embedded into devices. One of the most famous probably are TCP Offload Engines (TOEs) [Cur04] that have been built into Network Interface Cards (NICs) firmware. This is a classic example of ”host functionality” being statically offloaded into peripheral in order to achieve better performance, lower latency, decreasing host CPU utilization.

There are other attempts to offload pieces of functionality, traditionally performed by host, into some peripheral. Quite a few such attempts were made around distributed communication algorithms, particularly in clusters networks environment. Examples: fast

barrier using programmable NICs [BPS01], speeding up broadcast/multicast destribution [BPDS00], message passing interface (MPI) with rich set of performance enhancements achieved by teaching NIC about MPI [ZKW02]. There were also similar attempts in more consumer oriented areas like multimedia applications that were speed up using intelligent peripherals [FMOB98a].

All mentioned above are examples of static offloading. Where an optimization is hard-coded into peripheral's firmware in order to achieve the highest possible performance gain. While such approaches have proved successful in improving performance, they suffer from few drawbacks. First and immediate one is that making such an optimization is a complex task accompanied by severe potential consequences of erroneous code, thus requiring embedded system experts. Second, hard-coding features into the peripheral firmware is inflexible. Each offloaded optimization usually specifically designed for a particular application. But resources available on peripherals are very scarce, hence only limited number of those may be compiled into the firmware at a given time.

Dynamic offloading tries to address flexibility and development complexity issues while preserving the benefits of the static one. To make this possible peripheral's software is being designed in a way allowing future extensibility. Several design choice were practiced. One of them is Virtual Machine based firmware. It was used in Myrinet clusters [WJPR04] providing grounds for optimizing several MPI primitives. Another one is basing the firmware on an operating system (OS) specially tailored for run-time configurability (usually an academic one). Both approaches improves application optimization range to some extent, yet they don't come for free. Virtual machines trade off performance for configurability and OSs tend to have rather complicated loaders with not negligible footprint [BMW03]. And also those design choices may lay pretty far from industry common practices.

Present systems deploying dynamic offloading didn't get very far from static offloading

ones. They target narrow class of application rather then a specific application. However, system aiming to gain a practical use of dynamic offloading is very inflexible in peripheral choice and setup configuration options.

There were times when peripheral devices provided minimum functionality leaving the rest to host. However this is no longer the case. Technology trends in digital design have followed an exponential increase in the number of transistors on an integrated circuit. This ongoing trend of decreasing cost and increasing density of transistors motivates hardware and embedded system designers to use programmable solutions in their products. The proliferation of *programmable* peripheral devices for personal computers open 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 interface cards, video cards and more. 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.

1.2 HYDRA Overview

HYDRA [WDWAb, WDWAa, Y. 07] proposes a novel offloading framework, that enables utilization of various peripheral devices. The motivation for such a frmaeowrk becomes clearer as peripheral devices become powerfull and programmable.

In every modern PC there is a wealth of unused computing resources. The NIC has a CPU; the disk controller is programmable; some high-end graphics adapters are already more powerful than host CPUs.

The HYDRA 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. It defines a model where applications execute cooperatively and concurrently in host processors and in device peripherals. In this model, applications can *offload* specific tasks to devices to improve the overall application's performance. The framework also provides the necessary abstractions, programming constructs and development tools for developing such applications.

1.3 Dynamic Offloading Framework

This research introduces a general framework that supports *dynamic offloading*. The framework is used as a building block in HYDRA's offloading framework. The purpose of this research is to enable an efficient and transparent deployment of components into embedded devices. This research provides HYDRA with the offloading mechanisms that are used for offloading application's code to programmable peripheral devices.

1.4 Framework Requirements

- Genericity — The proposed dynamic offloading framework should be general enough to accommodate any programmable peripheral.

- Modularity — Providing a modular framework that can be easily modified according to a specific peripheral is a major design requirement. This research brings an object oriented methodology into heterogeneous embedded environments by defining host centric protocol for an individual task offloading. The protocol is device (and interconnect) independent and could be easily adopted by vast variety of embedded

peripherals. This approach provides easy *extensibility* for programmable devices limiting required developer knowledge.

- Ease of Use — To be able to implement offloadable modules, a developer will merely need to know public device's APIs (similar to using dynamic link libraries). Providing compatible APIs for different devices, will speed up development time and will enable code *reusability*, e.g., will enable to offload the same binary module into several devices. This option isn't available using current tools.

All the above raise natural desire for a generic offloading framework. HYDRAframework will enable application developers to choose how to subdivide their programs, and make it easy to alter the components at runtime to adapt to different hardware. The offloading framework will perform the actual offloading of specific modules to their target devices.

Chapter 2

Related Work

Most of today's work focuses at extension to a **specific** peripheral device and usually target narrow class of potential applications. This section summarizes offloading attempts along with techniques to create portable executable content.

2.1 Special OSs and OS Specific Object Loader

Stefan Beyer, Ken Mayes and Brian Warboys at University of Manchester experimented with Dynamic Configuration of Embedded Operating Systems [BMW03]. They focused on modifying OS behavior by means of exchanging process managers. In order to achieve it they used special purpose operating system and full fledged binary loaded. They also used FTP like network module repository.

2.2 Graphical Processing Units

Modern computer graphics hardware contain extremely powerful graphics processing units (GPU). These GPUs are designed to perform a limited number of operations on very large

amounts of data. They typically have more than one processing pipeline working in parallel with each other. Some of them provide methods for host applications to offload matrix manipulations like vertex shaders, which are been triggered on the GPU during appropriate stage of graphics pipeline. Microsoft DirectX provide a common API mechanism for applications to offload incorporated shaders into GPU as a part of application flow. Shaders are compiled by a specific compiler provided by GPU vendor (like CUDA compiler of NVIDIA). The shaders compiler actually creates a specific target opcodes, which are executed on the device. Most of the job is performed on the host.

However lack of compatibility between various GPU vendors make a reuse virtually impossible. Even when shaders are available in source codes it can be compiled only by target device compiler.

2.3 Java Applets and JavaME

Java applets on the other hand do hop. They're loaded into a remote client once requested and executed on this remote target. However, Java applet is executed by virtual machine (VM) running on the target client , which links the applet at load time. VM analyzes it prior to execution and provides all needed references (stitches it against types and objects in VM's runtime). So, applets require no pre-transfer-to-target preparation due to their interpreted nature, nor applets care about capabilities of a specific physical host they run on, due to their execution environment being in fact virtual machine.

Java Platform, Micro Edition (Java ME) provides a robust, flexible environment for applications running on mobile and other embedded devices like mobile phones, personal digital assistants (PDAs), TV set-top boxes and printers. Java ME includes flexible user interfaces, robust security, built-in network protocols, and support for networked and offline applications that can be downloaded dynamically. Applications based on Java ME

are portable across many devices, yet leverage each device's native capabilities.

However virtual machine introduces inevitable application slowdown.

2.4 Myrinet Clusters with VM based NICs

NIC-Based Offload of Dynamic User-Defined Modules for Myrinet Clusters [WJPR04] presents NIC-based Virtual Machine that allows users to dynamically add and remove code modules from the NIC. The code is added by the user in source form and compiled into an intermediate format, which is later interpreted by a special purpose virtual machine embedded in the NIC firmware. Though virtual machine was used, offloaded code achieve reasonable performance because the VM was specifically tuned for a specific sort of tasks.

This project was motivated by narrow class of distributed application and target only Myrinet NICs.

2.5 Active Networks

Active networks allow individual user, or groups of users, to inject customized programs into the nodes of the network. "Active" architectures enable a massive increase in the complexity and customization of the computation that is performed within the network, e.g., that is interposed between the communicating end points.

2.6 Microsoft's COM/DCOM Framework

COM is a framework for integrating components. This framework supports interoperability and reusability of distributed objects by allowing developers to build systems by assembling reusable components from different vendors, which communicate via COM. DCOM

takes that notion one step further as it allows an application to be built from many COM objects that reside in different machines communicating over network. Using COM/DCOM objects one can built a modular application, which can be distributed between several physical hosts. So, some application modules need to communicate with others residing elsewhere. But the objects themselves do not travel across those domains. So, such an application could not deploy itself on remote targets.

Java applet and COM/DCOM objects are well known technics to create portable executable contents. We'll compare them to *Dynamic Offloading* in order to better explore fine details of its nature.

2.7 Object Storage Disk (OSD)

OSD development started in the Parallel Data Lab at Carnegie Mellon University originally under the direction of Garth Gibson [Ruw, RG97, RGF98, Uni92]. OSD is a protocol that defines higher-level methods of communicating with the creation, writing, reading and deleting of data objects on disk. OSD is a level higher than a block-level access method but one level below a file-level access method. It is a technology intended to help make existing and future data storage protocols more effective in areas that include: storage management, security, data sharing, scalability and device functionality.

Since the OSD has processing power, it can be further extended. Recently, Huston et al. described a system called Diamond [HSW+04]. 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.

This work uses offloading of hard coded type of functionality in order to optimize narrow class of database applications. However, no actual offloading was implemented. Active storage was simulated by a regular PC and has been extended to provide SQL filtering as an internal OSD component.

2.8 Spine

Spine [FMOB98b] developed by Marc E. Fiuczynski (http://www.cs.princeton.edu/ mef/) now also at Princeton based on Spin [BSP+95]. They enable dynamic installation of device extensions, which are methods written in Modula-3. They implement a virtual machine (VM) on a NIC, which is their major focus. Each handler is identified by a number. The host or a peer can send an "Active Message" with an handler ID that will be executed by the extension. There is no concept of a "task" or a standalone application (or server) in the card. The internal interfaces are given as methods that create active messages, so a handler can call another one by sending it a message.

All extension invocations are executed as a result of an event (a message arrival). There is no programming model for building the applications. The application needs to create a specific extension (in a specific language - Modula 3), install it and "talk" with it using active messages (from the host).

However, neither a way to specify functional dependencies between extensions, nor a method to define which extensions get offloaded exist.

2.9 FarGo and FarGo-DA

Although not dealing with offloading, FarGo [HBSG99a, HBSG99b, Hol98, HBSG99d] and FarGo-DA [WBS02, HBSG99c] propose a programming model that enables a developer to program relocation and disconnection semantics in a separate phase during the application development cycle. The basic assumption for their work is that the application is fully comprised of a set of components that are tagged by a specific interface (called: *Complet*). The components are hosted in a virtual machine and can migrate to remote VM using marshaling and unmarshaling mechanisms (much like in the RPC [Sri95, BN84], RMI [Sun98, rmi99], CORBA [Sie98], DCOM [BK98] or WebService [Org] models).

2.10 TCP Offload Engines (TOE)

For some applications, the network stack's capabilities provided by the host OS can consume a significant amount of CPU cycles as well as memory bandwidth. This can create a bottleneck for applications that have significant networking traffic and are either limited by the amount of CPU cycles available to the application, or by the amount of available main memory bandwidth. To increase the capacity of these applications, either faster CPU/memory architectures need to be deployed or protocol off-load mechanisms need to be examined.

Protocol stack overhead can directly affect the performance of server applications that are constrained by CPU or memory bandwidth. For example, database networking stacks can consume as much as one third of the total CPU utilization. If that consumption was reduced to 5%, then additional CPU resources would be available to the database applications (the database application would run 42% faster).

While TOE technology [Cur04] have continued to gain popularity since 2000, TOE

has been less-than-successful from a deployment standpoint. There are only few IHVs[1] shipping TOE chips and NICs in volume. Most TOE IHVs have chosen to follow an all-inclusive approach for TCP/IP offload, including offload of connection-setup, data path offload, and support for ancillary protocols such as DHCP, ARP, ICMP, and IGMP. In general, IHVs are shipping TCP/IP offload as a parallel protocol stack.

[1] Short for Independent Hardware Vendor, a hardware-manufacturing company that specializes in a specific type of hardware device and not a complete computer system.

Chapter 3

Dynamic Offloading

Common peripheral devices usually provide only the necessary functionality to meet expectations of a particular class they implement, or optionally they may have a finite set of extra functionality. Yet programmable, they have rather limited resources to meet price expectations. We will define ***dynamic offloading*** as a process of extending device's functionality by injecting executable module(s) into a device during its operational cycle without interrupting it.

3.1 Motivation

Similar to Dynamic Link Libraries (DLLs), *dynamic offloading* provides the benefit of reusing piece of logic by several applications requiring this particular functionality. It also allows a particular application to better decouple from a particular peripheral device supporting its 'bizarre' needs and equally utilize any device of similar kind supporting downloading framework [WDWAb, WDWAa].

I/O consuming applications can utilize different portions of peripherals in an unforeseen, application specific manner. Using *Dynamic Offloading*, application could benefit greatly adapting peripheral to its specific need. Leveraging the proximity between the computational task and the data on, which it operates may boost the system's performance and reduce the load on the host processor and memory subsystem. Offloading to several devices at once adds a new dimension to our ability to handle information close to its source with limited involvement of the central CPUs. In particular, expensive memory bus crossings are eliminated.

One more similarity between *Dynamic Offloading* and Dynamic Link Libraries [GLDW87] is the necessity to (re)link modules prior to offloading them. Dynamic library is loaded once and then every application depending on it is relinked to the statically placed DLL image. In dynamic offloading scenarios core embedded application plays the "load-once" role, while every extension module is being linked against it. Functional relationship between application and library is very similar except that DLLs usually aren't self sufficient while embedded applications are.

However, comparing to DLLs, *Dynamic Offloading* has a new dimension of complexity introduced by the fact that:

1. peripherals has different (from host) and non-uniform execution environment(s), memory map etc.

2. binary data need to be transported to a remote device.

3.2 Design Considerations

In order to design *Dynamic Offloading* mechanism we need first to understand better the environment, which it should operate in. There are two participants: the host and the

peripheral. Responsibilities between those two along offloading might vary. We have considered different approaches to design *Dynamic Offloading*:

- The simple solution would be to hand over the OFFCODE to the target device and require that each device implement binary loader for a particular executable format used by the device. However, in general, memory and CPU resources in such devices are scarce. And this naive solution is quite expensive in terms of device resources.

- On the other extreme, device can be totally unaware of any offloading. Thus host will take all the burden of device dynamic configuration providing it with a complete firmware image each time. This approach imposes no device requirements however most of the devices wouldn't be able to preserve their runtime contexts. So, eventually hurting the desired dynamicity of system reconfiguration (or *Dynamic Offloading*).

- The middle ground is to define minimum requirement for device to be capable of truly dynamic offloading, leaving the rest for the host. Thus, keeping the impact on device resources as small as possible, while being able to inject functionality into operating device without disturbing its normal operation.

We proceed with the third approach as it doesn't require any special properties from the devices and in the same time allows to design generic, device independent offloading protocol. We'll refer it as *target-light* approach.

Chapter 4

Hydra Overview

This section provides a short overview of the HYDRA framework. For brevity, we only present the basic abstractions provided by the framework. The interested reader is advised to read [Y. 07].

4.1 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 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.

4.1.1 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 4.4 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. 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.

4.1.2 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.

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

4.2.1 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.

4.3 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.

4.4 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.

4.5 Software Architecture

The HYDRA runtime is comprised of several components as shown in Figure 4.1. 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 generic 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. This unit receives the offloading layout graph as input and produces the mapping between offcodes and target devices. The module can be easily extended to support future offloading constraints.

We have extended HYDRA with the following offloading capabilities:

- *User Level Library* — OA-Applications are linked with *libDynOffload.so* shared library. The library interacts with the kernel runtime *Offloading APIs* in order to facilitate the dynamic offloading of application specific OFFCODEs. The library is

also responsible for the compilation and linkage process, which is required in order to adapt a specific OFFCODEto a target device;

- *Offloading APIs* — This module intercats with the channel provider for the specific target device in order to execute the dynamic offloading protocol stages (See 6.2) against the device's loader OFFCODE;

- *Loader* – Each device has a light weight component that interacts with the host level library. The loader performs the required actions relating to memory allocation and OFFCODE deployment.

The red compoennets in Figure 4.1 mark the dynamic offloading framework's specific components.

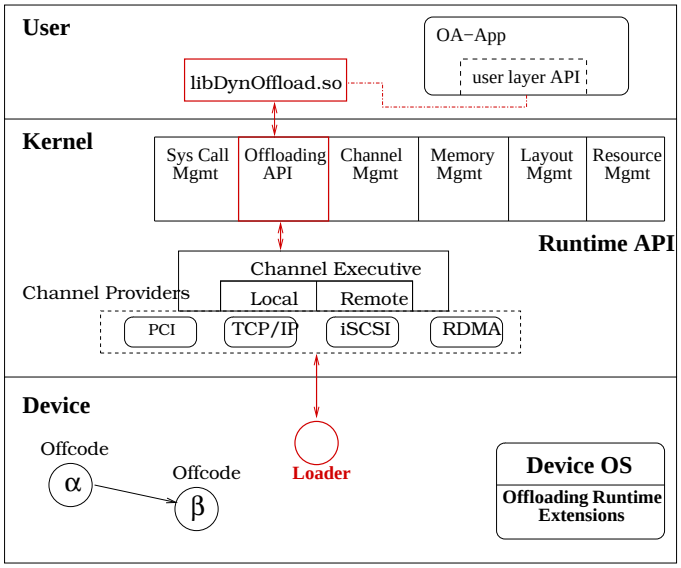

Figure 4.1: System Architecture

When application decide to offload a particular OFFCODE to a device, it calls user space library implementing offloading protocol described in Section 6.2. The library uses

offloading APIs of HYDRA kernel runtime to communicate with the target device and pass the OFFCODE to it. HYDRA kernel runtime is responsible for establishing and handling *OOB Channel* with the device and exchange data with it upon library request. On the device side there is a *Loader*. It's a slave part of the protocol, which performs actions required by host.

Chapter 5

Approaching Dynamic Offloading

We have chosen a programmable network interface card (NIC) with open source firmware to be our experimental periphery device. We have designed a NIC Operating System called NICOS that facilitates the evaluation of HYDRA applications [WDWAb, WDWAa, WADK06]). In this chapter we describe NICOS and its architecture, provide rationale for designing NICOS and outline its main features.

5.1 NIC Architecture

Our target device is a programmable NIC based on Tigon2 chipset. The Tigon programmable Ethernet controller is used in a family of 3Com's Gigabit NICs (for example, 3Com 3C985B-SX). 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. In the Tigon, each processor also has a private on-chip scratch pad memory, which serves as a low-latency

software-managed cache. Hardware DMA and MAC controllers enable the firmware to transfer data to and from the system's main memory and the network, respectively.

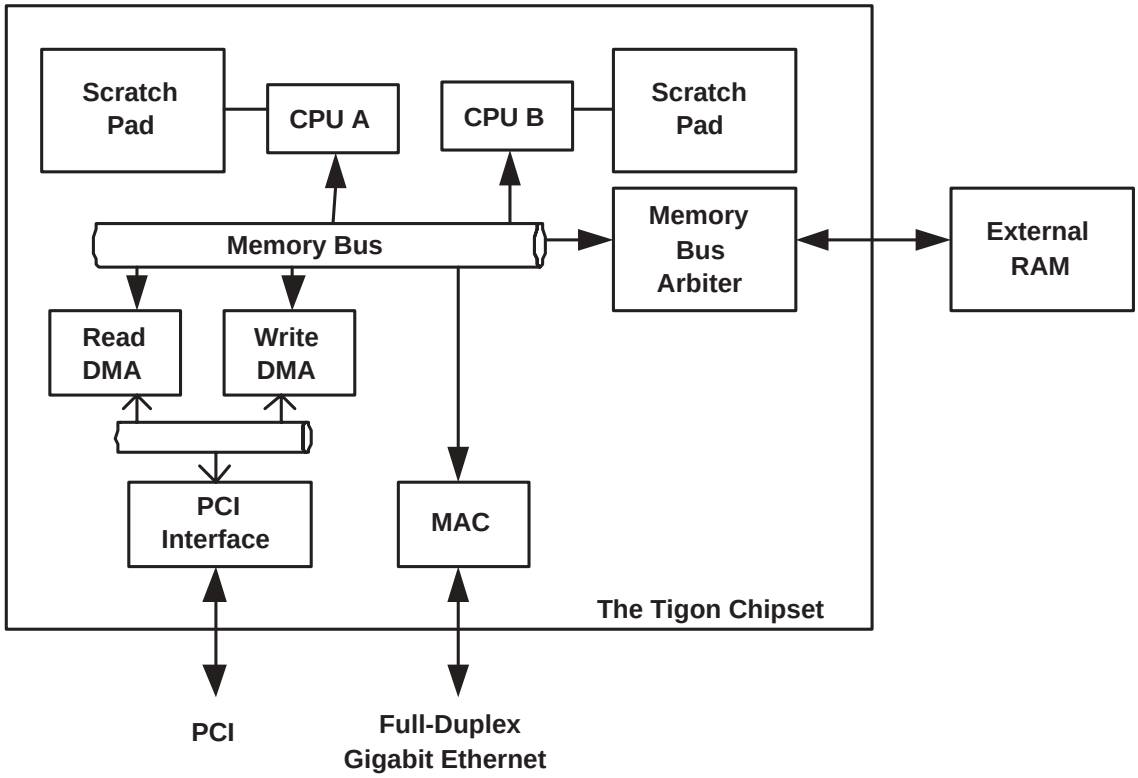

Figure 5.1: Tigon Controller Block Diagram

The Tigon controller uses an *event-loop* approach instead of an interrupt driven logic. 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.

5.2 NICOS Motivation

Developing NIC specific tasks can be a tedious task that involves hooking the existing NIC's firmware. Having a NIC level operating system may ease this task and facilitate the integration of application specific extensions at the NIC. The user extensions are realized in the form of NICOS tasks. NICOS also enable a generic interface for applications hosting through a well defined set of APIs.

5.3 NICOS Hardcore

This section presents the NIC operating system in a nutshell by introducing NICOSś main functional blocks and services they expose (henceforth, NICOS services). We start by describing NICOS general OSs responsibilities : tasks and scheduling and related APIs (e.g. create/destroy a task etc.), memory management and time service. Then we'll describe class specific services : the NICOS networking and the NICOS filtering APIs. There also instrumentation APIs that are mostly intended for debugging purposes.

Using a combination of standard tools for performance benchmarking, like Chariot and VTune, and proprietary device side profilers, we performed comparative analysis of native FW vs. NICOS based FW. It showed that NICOS introduces no performance impact relatively to base firmware. Moreover, it allows several typical network applications to gain a significant performance improvement (more details here [WDWAb, WDWAa, WADK06]).

5.3.1 Tasks & Scheduling

Original Tigon chipset has MIPS R4000-based event driven processor (rather then interruptible). Signals from various HW blocks are gathered into *event register*, which must be polled by SW in order to service HW events. For that purpose Tigon CPU has two

dedicated instructions , "pri & joff", for fast retrieval of highest asserted bit index and jump table operation. Hence, Tigon original firmware was driven by dispatch loop calling to event handlers functions for servicing HW requests. NICOS converted all those functions into high priority tasks preserving the same functionality but gaining flexibility and extendability introduced by operating system mechanisms.

NICOS task is an independent thread having its own CPU context. We implemented a classic *setjmp/longjmp* pair for saving/restoring CPU context within Task Control Block (TCB) for proper task switching. Each task has it's own name, priority, entry function and stack of controllable size (specified during task creation). Also during task creation an optional destructor routine could be specified. It is being called (if present) after task's entry function returns. All those task properties are stored in TCB along with CPU context.

NICOS provides several task management APIs that enable a developer to create/destroy tasks and to control their life-cycle state. The API enables a developer to create, yield, put asleep, suspend, resume and kill a task. Tigon chipset has hardware timer, which was used by NICOS to implement timers queue system for registering and processing time based events. On top of it NICOS provides *sleep(μsecs)* service and genuine periodic tasks.

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 completely aware of them since their creation. Such a design allows the OS to minimize the ready-to-running[1] latency. Providing the timeliness guarantees required by NICOS has been a major challenge due to the non-preemptive architecture of these NICs.

As already mentioned, NICOS utilize a non-preemptive scheduling due to non-interruptible nature of Tigon CPU. Thus having cooperative multi-tasking manifested by voluntary CPU release via *yield*, *sleep* or task termination. NICOS scheduling infrastructure designed in

[1]The time from the moment task become ready-to-run till it starts execution.

a way that the routine for choosing next-task-to-run can be easily replaced. Setting fertile ground for experimenting with different scheduling algorithms [WADK06].

NICOs default scheduler is a static priority one with 32 priority levels. There are several ready task queues (per-priority). Each queue holds tasks of the same priority. There is one additional queue that holds blocked tasks. Tasks become blocked as a result to *yield*, *sleep*, or *suspend* calls. The scheduler always executes the highest priority task that is ready to run.

Task related API

- *nicosTask_Create* - Allocates stack and task control block (TCB), initializes the TCB and releases this task into ready queue.
- *nicosTask_Kill* - This routine causes a specified task to cease to exist and deallocates its stack and TCB.
- *nicosTask_Yield* - Triggers rescheduling.
- *nicosTask_Self* - Returns task ID of the caller.
- *nicosTask_Suspend* - Suspends the task specified by id from further execution by placing it in the suspended state. This state is additive to any other blocked state that the task may already be in. The task will not execute again until another task issues the *nicosTask_Resume* directive for this task and any other blocked state has been removed.
- *nicosTask_Resume* - This directive removes the task specified by id from the suspended state.
- *nicosTask_Exit* - Ends caller task.

Time based Services

- *nicosTask_CreatePeriodic* - Similar to create, but instead of releasing the task, creates periodic timer with specified period parameter.

- *nicosTask_Sleep* - Suspends the task and creates a timer, which allows the task to resume after specified time interval has elapsed.

5.3.2 Memory Management & Data Structures

NICOS has to allocate memory each time a task, queue or packet is created. The *nicos_malloc* and *nicos_free* functions are used for this purpose. NICOS default dynamic memory allocation algorithm is based on the "boundary tag method" described in [PRW95], 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 (the amount of time taken to execute the function will differ from call to call). Since different realtime systems may have very different memory management requirements, a single memory allocation algorithm will probably won't be appropriate.

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 reduced the end-to-end latency.

There are also APIs for pools and lists management. Pools and lists can be created from any chunk of consecutive memory whether allocated dynamically or a static array. Pool API allows to initialized provided chunk to be a block pool (number of objects of same type). Then objects can be allocated from that pool or be released back into it. Lists

management APIs allowing to create and manage general linked lists, FIFO queues, stacks and sorted lists.

Memory APIs

- *nicos_malloc* - allocates memory chunk of specified size.
- *nicos_calloc* - allocates memory chunk of specified size and set its value to zero.
- *nicos_free* - releases memory chunk into dynamic memory pool.
- *nicos_memset* - initializes memory area specified by pointer and size with given value.

Pool APIs

- *POOL_T* - macro for pool type declaration using specified type for poll members.
- *NICOS_POOL_ALLOC* - macro for allocating pool entry.
- *NICOS_POOL_FREE* - macro for releasing block to the pool.

List APIs

- *LIST_NODE_T* - macro for list node type declaration with specified data type.
- *nicosList_Add* - adds item to the head of the list
- *nicosList_InsertSorted* - inserts item into the list according to specified comparator function.
- *nicosList_Pop* - removes item from the head of the list.

- *nicosList_Top* - returns item pointed by the head of the list.
- *nicosList_Bottom* - return item pointed by the tail of the list.
- *nicosList_Remove* - removes specified item from the list.
- *nicosList_Size* - returns number of items in the list.
- *nicosList_IsEmpty* - returns true if no item in the list.

5.3.3 Filtering & Classification

When deciding which functionality is needed to be offloaded to the NIC, we looked for common building blocks in todays 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 usefull 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 NICOS filtering mechanism is comprised of two phases. In the first phase a packet is matched against a given filter. Once a match is found, a user callback is executed. The callback can decide to drop the packet or operate on it (e.g. modify it, take some statistics measurements, etc).

NICOS filtering services enable the user to group several filters into a named group. A named filter group can have an aggregated operation on the matched packet. I.e., in case there is a match on ALL filters in the group, the corresponding group callback is invoked.

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

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
  ptrValueMask = valueMask;
  /* 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 = &ptrValueMask;
  /* create the filter */
  pingDropFilter.filter_type = STATIC_PATTERN_FILTER;
  pingDropFilter.pattern_filter = &pattern_filter;
  /* add the filter to the Tx flow */
  nicosFilter_Add(&nicosTxFilters,&pingDropFilter,DROP,NULL,
                  GENERAL_PURPOSE_FILTERS_GROUP,&pingFilterTxId);
  /* also add the filter to the Rx flow */
  nicosFilter_Add(&nicosRxFilters,&pingDropFilter,DROP,NULL,
                  GENERAL_PURPOSE_FILTERS_GROUP,&pingFilterRxId); }
```

Filtering/Classification API

- *nicosFilter_Add* - registers provided pre-built filter into the specified filter group
- *nicosFilter_DeRegister* - removes specified filter
- *nicosFilter_GroupEnable* - makes certain filter group enabled. All filter belong to it will be matched against every packet.
- *nicosFilter_GroupDisable* - excludes all filters belonging to the specified group from matching until further notice.

5.3.4 Networking

NICOS is designed to be an operating system for network class devices. Therefore, it provides method for sending and receiving data packets.

Transmit service

NICOS provides an API function for transmitting raw packets. The payload is provided by application writer must have all of the desired protocol headers. User also can provide different priorities for packet to be transmitted. NICOS transmit API supports synchronous and asynchronous calls. The asynchronous is non blocking. When using the synchronous mode, the execution is blocked until frame transmission is completed. Upon completion, the provided callback is called.

NICOS has extended transmit path with packet priority queues. All transmit packets could be classified, prioritized or filtered using filtering & classification API 5.3.3. However, no extra data copy was introduced. The priority system implementation is scalable, and can be easily adjusted to any number of priority levels in the system.

Receive service

Receiving a packet is currently done only via filter registration 5.3.3.

Transmit API

- *nicosTx_SendData* - schedules specified data buffer for sending.

5.3.5 Embedded Application Instrumentation

NICOs provide mechanism for application instrumentation. This mechanism accumulates trace messages in a circular buffer. Host periodically drains this buffer and directs those messages into system debug stream. This mechanism supports several levels of verbosity allowing to get messages on errors, warnings, informatic events or even FW flow progress. Trace messages from an arbitrary defined piece of code can be bundled into a *trace module*. Instrumentation mechanism allows to control verbosity level per each *trace module* separately. NICOs has several predefined *trace modules* for convenient way to instrument any kernel block or service.

Trace API

- *nicos_trace* - places parameterized string into trace buffer.

Chapter 6

Dynamic Offloading Prototype

In this chapter we define generic offloading protocol using "target-light" approach described in Section 3.2. We also describe prototype protocol implementation for network interface card that has NICOS [WDWAb, WDWAa, WADK06] OS based firmware extended with HYDRA runtime. Description of used tool chains is included.

6.1 Prototype pre-Requisites and Tools

6.1.1 Binutils and Compiler

We used GNU gcc cross-compiler and binutils [tig] to build NIC's firmware. We used the same tools throughout prototype implementation for OFFCODE creation and processing. All tools we used for prototype have being part of original Tigon firmware's tool chain.

- **gcc** - C cross-compiler
- **ld** - linker. combines a number of object and archive files, relocates their data and ties up symbol references.

- **nm** - lists the symbols from object files
- **objcopy** - copies the contents of an object file to another. It should be able to copy a fully linked file between any two formats.
- **readelf** - displays information about one or more ELF format object files.

6.1.2 Standard Linux Utils

We used GNU gcc cross-compiler and binutils [tig] to build NIC's firmware. We used the same tools throughout prototype implementation for OFFCODE creation and processing.

- **grep** - searches input for patterns
- **sed** - stream editor that can be used to perform basic text transformations on an input stream

6.1.3 In-house Tools

We used GNU gcc cross-compiler and binutils [tig] to build NIC's firmware. We used the same tools throughout prototype implementation for OFFCODE creation and processing.

- **get_sizes** - perl script that uses readelf to calculate effective size needed for ELF sections : text, data, rodata.
- **allocate_offcode** - C application that calls HYDRA Offloading API to allocate memory on device.
- **download_offcode** - C application that calls another HYDRA Offloading APIs to download OFFCODE to the device and activate its entry routine.

6.2 Dynamic Offloading Protocol

We have considered different approaches for dynamic loading problem, as were presented at Section 3.2 and the *target-light* approach was chosen. Using this approach target's role in the offloading procedure is being as little as possible while host does most of the processing. In this section we finally define the protocol following those principles.

There are several things to be done in order to deploy binary object (OFFCODE from now on) on a running target:

- Memory must be allocated on the target to hold the OFFCODE
- OFFCODE need to be prepared to execute from the location it's about to be placed
- OFFCODEś code and data need to be transported to the target
- Optionally, OFFCODE activation/initialization may be triggered.

We divide those actions between host and device following chosen *target-light* approach. Device must know to allocate memory and accept the binary data, while host is responsible for all the rest. E.g. parse binary format, calculate effective size, prepare OFFCODE etc.

Figure 6.1 presents the message transfers that occur in loading a single OFFCODE . In the sake of generality, we intentionally did not include OFFCODE activation/initialization in the protocol flow as explicit message handshake (only mentioned it on the side). This action is optional and rather a by-product of specific system design choice then a protocol stage. We'll show an example of such decision while describing our prototype implementation.

Once the host-based loader calculates the OFFCODEś size, it asks the device's loader to allocate memory for it. The runtime loader allocates the memory and returns the device's

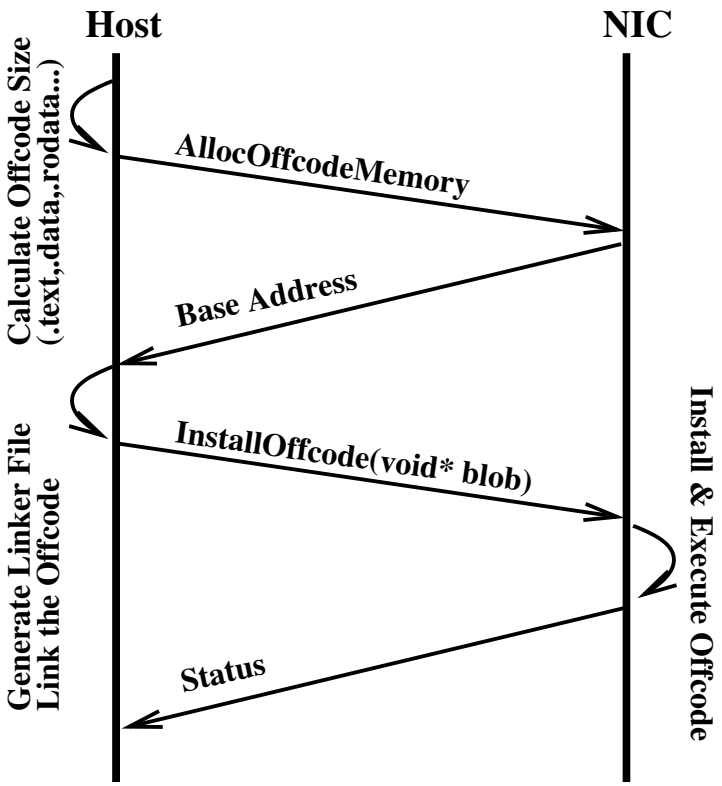

Figure 6.1: OFFCODE Dynamic Loading Flow

memory address to the host. The host dynamically generates a linker file (described in details later) adjusted by the returned base address and links the OFFCODE object. It then transfers the linked OFFCODEś sections to the target device where it is placed and executed. All the above interactions make use of the OOB channels that are created for the host and target HYDRA runtimes. As a proof of concept, we have created such a loader for our programmable network card 5.1.

The proposed protocol requires no direct target access to peripheral. Thus introducing no restriction on peripheral's interconnect. Due to the fact that all communication is solicited by the host, interrupt capabilities aren't required from peripheral and polling based

solution are possible as well.

6.3 Integrated Offloading Prototype

In the previous sections we described generic offloading protocol. In this section we present prototype implementation of the protocol for a specific target device. Both host and device parts of the protocol have been prototyped. Our target device is a programmable NIC based on the Tigon2 chipset. We have extended original NIC's firmware with NICOs and integrated the HYDRA runtime into it.

6.3.1 *Offloading Protocol Implementation*

In our prototype we went on explicit OFFCODE activation. So, here are 4 sequential stages to be performed:

- OFFCODE size retrieval and NIC memory allocation
- OFFCODE target specific linking
- OFFCODE sections retrieval and actual offloading
- OFFCODE activation

In order to focus on the offloading itself, our offloading application is a perl script that implements protocol's high-level logic. I.e. performs each of the stages described above. It uses target specific binutils (see Subsection 6.1.1) and library API to communicate with target through *O-O-B Channel* (see Subsection 6.3.2)

Figure 6.2 provides the full version of the protocol script implementation. The script is invoked with two parameters, OFFCODE object file base name and its activation routine name. Lines 6-7 extracts those parameters.

```
1 #!/usr/bin/perl

2 if ($#ARGV != 1) {
3     print STDERR "Usage: $0 <ofcode_binary> <init_func>
4     exit 1;
5 }

6 $ofcode = shift; $init_func = shift;
7 $ofcode_obj = $ofcode . ".o"; $ofcode_h = "ofcode.h";

8 $size = `get_sizes --linux -ptigon2 ofcode.bin0`;
9 $addr = `allocate_ofcode eth2 $size`;

10 `sed -i -e 's/\\.\\s=\\s0x4000\\;/\\. = $addr\\;/' alteon.x`;
11 `$NICOS_TOOLS/bin/ld -nostdlib -T alteon.x -o ofcode.bin $ofcode_obj`;

12 `$NICOS_TOOLS/bin/genfw  --linux  -ptigon2 ofcode.bin $ofcode_h`;
13 `../tools/build_env2.9.5/bin/nm ofcode.bin | sort > ofcode.map`;
14 if (`grep $init_func ofcode.map` =~ /0*([\dabcdef]+)\s\w\s(\w+)/)
15 { $launch_addr = hex($1); }

16 `download_ofcode eth2 $launch_addr`;
```

Figure 6.2: Perl script offloading implementation

Stage 1 begins with determining OFFCODE binary content size (line 8) using *get_sizes* perl script, which merely extracts code and data sizes (text, data, rodata, bss and sbss sections) using *readelf* binutil and summates them. First stage is accomplished by calling library API (line 9) that asks device *O*FFCODE *loader* to allocate required amount of memory and returns its initial address.

Stage 2 is accomplished using linker scripts (see Subsection 6.3.3) that instructs linker how to build output image. There is a base linker script (named alteon.x) that is been updated with base address for OFFCODE placement (line 10). On line 11 linker is been invoked and provided with linker script and OFFCODE object file. Its output is ELF executable image file containing ready-for-download OFFCODE . Important to say that though there are dependencies between core firmware and the OFFCODE core firmware image is

not required for the link stage nor any other stages of the prototype.

Stages 3 and 4 are undertaken by single library API call on line 16. But there are several preparations to be done first. Line 12 invoke *genfw* utility provided with target firmware build environment. It extracts code and data sections' binary content from ELF binary. *genfw* is a perl script that uses *readelf* binutil for sections extraction. Lines 13-15 create OFFCODE post-linkage map and retrieve from there activation routine address. Library API call on line 16 uses this address along with *genfw* products to offload and activate the OFFCODE .

6.3.2 Channel and Library API

Our prototype implementation uses default *O-O-B Channel* provided by HYDRA for supported device. We deliberately avoided specialized *Channel* with device side zero-copy and single DMA transaction per download request.

This is suboptimal design choice for given device. On the other hand it allows us to model behavior and draw conclusions for wider family of devices including those having no master access capabilities and/or using packet based interconnects like USB, SDIO, etc.

We defined a specific format for data passed down the *O-O-B Channel* by the offloading library, which is interpreted by device's *O*FFCODE *Loader*. There are 3 types of actions host can request from device: memory allocation, binary content download and function execution. Figure 6.3 provides the exact format of those actions.

Due to the fact that our top most application is a perl script and it's been interpreted rather then compiled, for cleaner prototype implementation we packaged user space offloading API in two stand alone executables instead of link library (see Figure 6.2). They

```
typedef struct _OFFL_ACTION {
    U32 action;
    union {
        struct {
            U32  dest_addr;
            U32  len;
        } binary_chunk;
        struct {
            U32 size;
            U32 reserved;
        } mem_alloc;
        struct {
            U32 init_func;
            U32 reserved;
        } launch;
    } descr;
    U32 data[0];
} OFFL_ACTION, *POFFL_ACTION;
```

Figure 6.3: Action header format

can be packaged in a link library with no functional change. Description of those executables immediately follows:

- **allocate_offcode** - uses provided figure to assemble single *Action* of mem_alloc type and send it through the *O-O-B Channel*. Then it reads the channel to get from NIC start address of allocated space.

- **download_offcode** - performs another 2 stages. Downloads the linked OFFCODE to the NIC using one or more *Actions* of binary_chunk type. Then it sends another *Action* of launch type assembled using provided activation function address. See Subsection 6.3.4 for details.

6.3.3 Preparing an OFFCODE for Offloading

As we already mentioned OFFCODE linkage in our experiment was done by host. Prototype offloading procedure received all OFFCODEs as a pre-compiled object files along with names of their activation/initialization routines.

Following issues need to be addressed by host to get an OFFCODE prepared for dynamic offloading:

- Resolve external dependencies - OFFCODE is not a stand alone application. It's allowed to use NICOS API, which is provided by core NIC firmware. In general case OFFCODE is a target application that may use static public APIs provided by target.

- Relocate the binary code - executable sections need to be properly placed and text's content need to be adjusted with correct data/rodata/globals location. Only after allocating memory on NIC final OFFCODEś binary section locations are computable.

The linkage is performed by regular GNU binutils cross linker [BU]. Same one that's used during build of target core firmware.

To achieve those 2 goals, linker is supplied with a special linker script [Lan] instructing the linker where to place sections and providing symbols definition for all potential OFFCODE external dependencies. The only piece of information not known in advance is the start address of where the OFFCODE will be placed in target address space.

Our prototype has linker script template (See Figure 6.4), which in general case can be provided by embedded target vendor. First in the template appear all NICOS public APIs (with their target addresses) exported by the device for OFFCODEs. Then start address placeholder followed by section placement instruction (Refer to [Lan] for details).

The only thing to accomplish during the preparation stage is to replace default value

". = *0x4000;*" in the template appearing under "*Text Start Point*" comment with a valid address and use the resulting script for linking an OFFCODE. The outcome of the script guided linkage is a ready-to-be-offloaded ELF binary.

6.3.4 Offloading & Activation

Both offloading and activation are done by **download_offcode** as already mentioned in subsection 6.3.2. We use *genfw* utility provided with target firmware build environment that converts ELF binary to C header file with constant arrays containing the sections' binary content. *genfw* is a perl script that uses readelf tool from binutils [BU] for sections extraction (line 12 on Figure 6.2). Last line (16) of protocol implementation executes **download_offcode** which uses sections' contents extracted by *genfw* and target address provided as parameter.

First, sections are iterated in the same order they appear in the linker script (See Figure 6.4) and a download procedure is performed on each. At this stage *Offcode* memory location on target is already known along with sections sizes, so start address of each section is easily computed. Download procedure iterates on sections' binary data, wraps it with *Action* header and passes them through *O-O-B Channel* to the NIC. Each *Action* is equipped with an address for data it contains to make NIC Loader's job easier (see format at Figure 6.3).

As we mentioned in Subsection 6.3.2 our implementation also models packet based interconnect. So, we imply an arbitrary upper boundary of 1.5K on data size to be passed to device in one channel transaction. So, sections can be larger then a single *Action* payload. After offloading is done, another *Action* is built and sent to the NIC passing the OFFCODEś init function address supplied to **download_offcode** as a command line parameter.

6.3.5 NIC OFFCODE Loader

We have implemented a simple *O*FFCODE *loader* task on NIC. It has only two functions : *LoaderInit* and *ActionParser*. *LoaderInit* registers a NICOSfilter to be able to catch *Actions* arriving from *Channel*.

ActionParser is called when offloading *Action* is recognized by filter match on data going down the *O-O-B Channel*. Then it parses its header and perform one of the three predefined operations (see full source at Figure 6.5):

1. Memory Allocation — allocates consecutive memory of size specified in the *Action* and passes the address via channel back to host.

2. BLOB Copy — copy specified number of bytes from *Action* payload to the address specified in *Action* header.

3. Function Execution — jumps to an address specified inside Action

Due to the fact that *Actions* bringing binary data also provide an address where to place it, NIC *O*FFCODE *loader* is very simple and requires no book keeping or state accumulation.

```
OUTPUT_FORMAT("elf32-bigmips", "elf32-bigmips","elf32-littlemips")
OUTPUT_ARCH(mips) ENTRY(_start)
SECTIONS {
  /* NICOS & library symbols definition*/
  . = 0;

  /* NICOS Task API */
  nicosTask_Create = 0x192dc;
  nicosTask_CreatePeriodic = 0x17e08;
  nicosTask_Kill = 0x18150;
  nicosTask_Yield = 0x193a0;
  nicosTask_Sleep = 0x18680;
  nicosTask_Suspend = 0x18a84;
  nicosTask_Resume = 0x18c6c;

  /* NICOS Filter API */
  nicosFilter_Add = 0x19af0;
  nicosFilter_DeRegister = 0x19d98;
  nicosTx_SendData = 0xb440;

  /* NICOS Memory API */
  nicos_malloc = 0x1aa78;
  nicos_free = 0x1ae0c;
  nicos_alloc_packet = 0x1af40;

  /* NICOS Trace API */
  nicos_trace = 0x1953c;

  /* NICOS global data structs */
  NicosScheduler = 0x25310;
  nicosTxFilters = 0x1ef80;
  nicosRxFilters = 0x22930;

  /* Text Start point */
  . = 0x4000;

  _ftext = . ;
  .text      :  { *(.text) } =0
  .rodata    : { *(.rodata)  }
  . = ALIGN(4);
  .data    :   { *(.data) }
  _edata  =  .;
  .sbss      : { *(.sbss) *(.scommon) }
  .bss       :
  {
   *(.dynbss)
   *(.bss)
   *(COMMON)
  }
  _end = .;
}
```

Figure 6.4: Linker script for OFFCODE proper placement and relocation

```
void actionParser(PACKET* pBlob,Call* pcall) {
    U32 addr = 0;
    PACTION_T pAction = (PACTION_T)pBlob->data;
    NICOS_TRACE1(TEST,INFO, __FUNCTION__, pBlob->length);

    switch (pAction->action) {
        case OFCODE_ACTION_ALLOCATE:
            NICOS_TRACE0(TEST, INFO, "Ofcode mem_alloc");
            NICOS_TRACE1(TEST, INFO, "size:", pAction->descr.mem_alloc.size);
            addr = (U32)nicos_malloc(pAction->descr.mem_alloc.size);
            *(pcall->out_buffer) = addr;
            nicos_call_finish(pcall);
            break;

        case OFCODE_ACTION_CHUNK:
            NICOS_TRACE3(TEST, INFO, "len & addr" ,
                         pAction->descr.binary_chunk.len,
                         pAction->descr.binary_chunk.dest_addr);

            bcopy((U32*)pAction->data,
                      (U32*)pAction->descr.binary_chunk.dest_addr,
                      pAction->descr.binary_chunk.len);
            break;

        case OFCODE_ACTION_LAUNCH:
        {
            OFCODE_LAUNCH_FUNC  init_func = NULL;
            NICOS_TRACE0(TEST, INFO, "Offcode launch");
            NICOS_TRACE1(TEST, INFO, "init:", pAction->descr.launch.init_func);
            init_func = (OFCODE_LAUNCH_FUNC)pAction->descr.launch.init_func;
            init_func();
            break;
        }

        default:
            NICOS_TRACE1(TEST, WARN, "Loader: wrong action", pAction->action);
    }

    NICOS_TRACE0(TEST,INFO, "<<" __FUNCTION__);
}
```

Figure 6.5: NIC Loader

Chapter 7

Evaluation

The goal of the experiment was to establish feasibility of dynamic target loading by a general framework rather then per-target proprietary solution and its operational aptness for offload-aware applications deployment. In order to establish that, we measured performance of different aspects of dynamic target offloading prototype implementation and compared it with an equivalent user-level host application.

We measured temporal and spacial overhead of offloading infrastructure on both NIC and host. Those measurements are restricted to the "target-light" loading approach (described in Section 3.2). Measurements were done on dual 450Mhz Pentium CPU machine running Fedora Core 4 Linux OS. In order to emphasize the generality of the approach, all OFFCODEs we used in the experiment were provided as a pre-compiled object files, while the protocol implementation uses no additional knowledge about their nature.

We would like to start from justifying the name we used for chosen offloading approach : "target-light" . Hence, first metric we present is embedded loader footprint. Experimental loader we built requires only **372 bytes** of target memory. In our prototype we tried to cover richer functionality to make stronger point. Depends on implementation

design choices Loader footprint can be much smaller. (If for instance OFFCODEś initialization/activation routine will be placed at the beginning of the designated memory space and called right after download.)

7.1 OFFCODE preparation performance

It's not unreasonable to assume that peripherals would not expose arbitrary large sets of external APIs. Based on that assumption, applications having potential to be offloaded and executed on some target device will have finite small number of external symbols in corresponding binaries. We performed all measurements on ELF binary objects with 5 external symbols and various code segment size. Due to the nature of offloading implementation total object's binary data size is a crucial parameter rather then particular section size. So the mentioned ELF objects above had sections other then code of size zero.

We measured OFFCODE linkage time and compare it to equivalent host procedure. We compare it to the case when host executes OFFCODE equivalent task implementation packaged in a DLL. In such a case host loads and executes ELF object of properties similar to OFFCODE. We simulated host activity by explicit call for ld linker, linking OFFCODE with NIC's core firmware and measured linkage execution time. In this experiment also objects' data distribution between sections isn't important because the only difference between two linkage technics is whether additional object of fixed size need to be parsed to resolve external dependencies. OFFCODE script driven linkage expected to be faster then DLL because all external dependencies are provided by the script (See Subsection 6.3.3). Thus requiring parsing of no additional ELFs.

Figure 7.1 shows that reality matches the expectations. OFFCODE linkage (script driven) is faster by a constant margin then DLL linkage, while both link methods have similar growth tendency as a function of OFFCODE size.

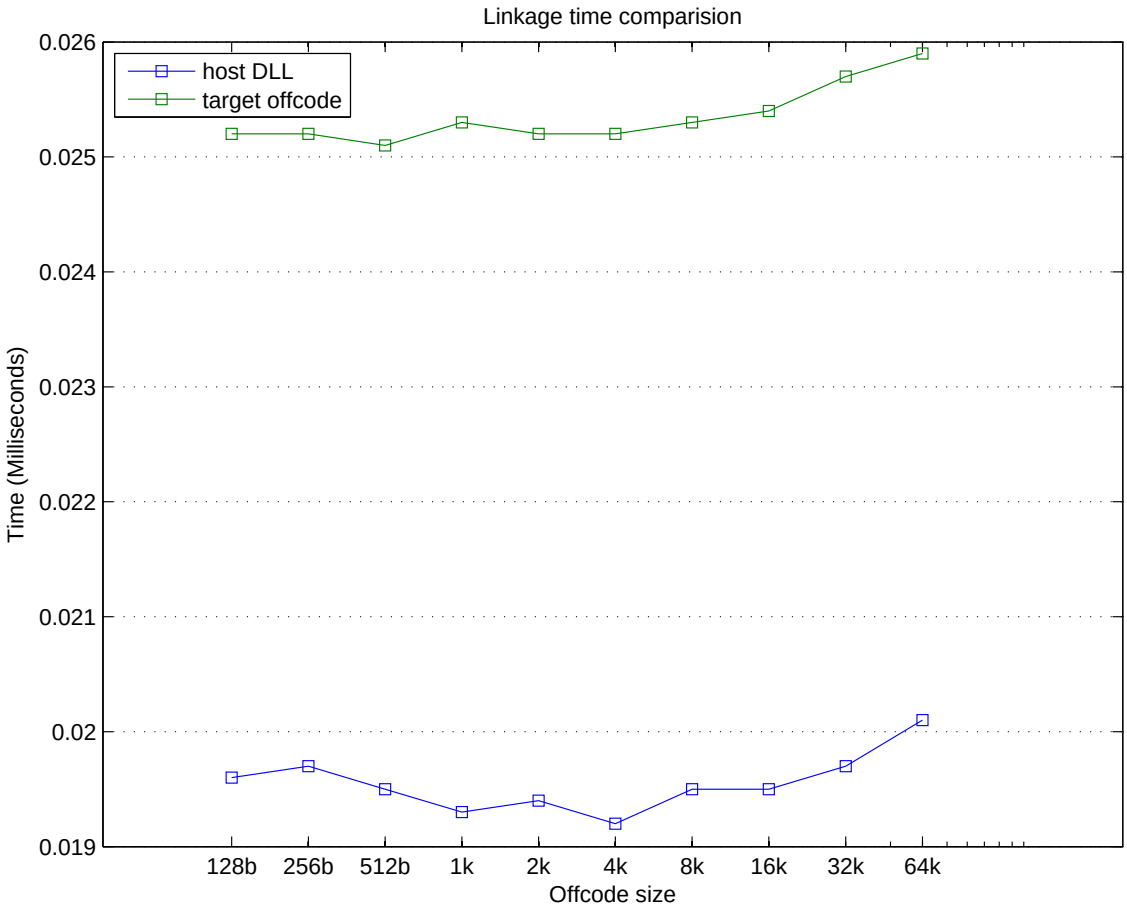

Figure 7.1: Offcode linkage performance

7.2 Offloading performance

Beside the linkage we also measured the total time spend on offloading and activation. It takes at least 3 *Actions* to allocate, offload and activate a particular *Offcode*. It could be more then 3 because OFFCODE can be larger then single *Action* maximum data size we imposed for packet based interconnect modeling. Due to the fact that *Actions* are created in user space library, each causes trap to kernel and a context switch. For that reason within offloading measurement we also registered time on target between *Actions* arrivals.

Table 7.1: Offload performance

Ofcode size	ActionFrame count	Total time	Total handling time	Avg. interarrival
32b	3	55,449	89	27,680
64b	3	54,676	92	27,292
128b	3	83,707	79	41,814
256b	3	65,939	98	32,921
512b	3	39,782	108	19,837
1k	3	56,430	128	28,151
2k	4	200,437	184	66,751
4k	5	133,058	281	33,194
8k	8	153,415	494	21,846
16k	14	426,019	909	32,701
32k	25	1,051,045	1,745	43,721
64k	47	1,113,196	3,392	24,126

Ofcode size - size of text section. Other sections are of 0 size.
ActionFrames count - total number of ActionFrames sent by channel during offload and activation
Total handling time* - time spent by NIC on handling an *Action* and performing required actions
Total time* - time taken by the WHOLE offloading and activation process
Avg. interarrival - mean time between each 2 *Actions* arrivals to NIC during the process
* time units are **micro**seconds in all time related columns

Measurement Methodology: During Offloading and activation measurement (**allocate_offcode** and **download_offcode** calls) time stamps was gathered both on the host and on the NIC. NIC time stamping allowed very accurate inter-*Action* time measurement. Host and NIC approaches showed virtually the same numbers validating our results.

Figure 7.1 shows all time measurements results including total offloading procedure performance. We see that starting from 2K *Offcode* size time is roughly a linear function of size. More accurate to say is that time is affected most by number of ActionFrames used in the process.

Major use case of offloading is application deployment. For those purposes even a naive approach used in this consciously suboptimal prove-of-concept experiment is sufficient.

Chapter 8

Conclusions and Future work

In this thesis we have designed generic offloading protocol suitable for wide variety of devices. We claim that chosen approach makes very moderate demand from device vendors yet provides very powerful opportunity for device extensibility and reuse. As a proof of concept we implemented prototype on selected device and measured its performance. Based on those measurements we conclude that OFFCODE preparation is comparable to one of dynamic link library. And even naive implementation of transporting the OFFCODE to the target device is suitable for application deployment. While for bus master devices (like PCI/PCIe) proper use of Host→Target DMA can improve OFFCODE transfer performance by roughly 3-4 orders of magnitude.

In our experiment NIC dynamically allocated one consecutive memory range that host all binary content (text, data, etc.). There is a need to define extended protocol for requirement/capabilities handshake between host and target that will provide support for scattered memory. For instance some target may have different banks of memory for instruction and data, which raise the need to deal with several memory ranges. Second motivation is target with some degree of memory fragmentation that have the needed amount of free memory

but not a consecutive range. (Separate memory banks example can be treated as a subcase of the second one). Different approach can be used in fragmented memory case. Host can be notified of target's available memory map and host will use advanced link technics (yet to be developed) to fit the OFFCODE in.

OFFCODE Authentication wasn't mentioned at all in this research though security matters are especially important in this field. Malicious OFFCODEs have much greater negative potential due to direct access to HW then user space modules those OFFCODEs replace. Need to be developed efficient yet strong technique to verify OFFCODE authenticity online with minimal performance impact guaranteeing targets from malicious code execution.

Architecture independent OFFCODE representation - there is a natural desire to deploy same functionality on devices of different makings. It can't be achieved if OFFCODE repository contain objects pre-compiled for a specific target. On the other hand it's unreasonable to require all OFFCODEs to be provided in source code. Also, in our prototype we worked with single device and used the complete tool chain to prepare OFFCODE for offloading. It's quite wasteful space-wise, because OFFCODE preparation uses small portion of linker capabilities. Situation gets worth if there are several HYDRA enabled devices with different CPU architectures need to be supported. This can be especially critical for Ultra Mobile Devices (or Mobile Internet Devices), which have less resources.

Definition of architecture independent OFFCODE representation along with designing extensible light linker with per CPU architecture plug-ins could solve both issues above, also providing great deal of scalability.

Bibliography

[BK98] N. Brown and C. Kindel. *Distributed Component Object Model Protocol — DCOM/1.0. Internet Draft*, January 1998. Available at http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-02.txt.

[BMW03] S. Beyer, K. Mayes, and B. Warboys. Dynamic configuration of embedded operating systems, 2003.

[BN84] A. Birell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BPDS00] D. Buntinas, D. K. Panda, J. Duato, and P. Sadayappan. Broadcast/multicast over myrinet using NIC-assisted multidestination messages, 2000.

[BPS01] D. Buntinas, D. K. Panda, and P. Sadayappan. Performance benefits of NIC-based barrier on Myrinet/GM. In *Proceedings of the Workshop on Communication Architecture for Clusters (CAC) held in conjunction with IPDPS '01*, April 2001.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 267–283, New York, NY, USA, 1995. ACM Press.

[BU] GNU Binary Utils. Available at site: http://www.gnu.org/software/binutils/.

[Cur04] A. Currid. TCP offload to the rescue. *Queue*, 2(3):58–65, 2004.

[FMOB98a] M. Fiuczynski, R. Martin, T. Owa, and B. Bershad. On using intelligent network interface cards to support multimedia applications, 1998.

[FMOB98b] M. E. Fiuczynski, R. P. Martin, T. Owa, and B. N. Bershad. 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*, pages 7–12, New York, NY, USA, 1998. ACM Press.

[GLDW87] R. A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks. Shared libraries in sunOS. *Proceedings of the USENIX 1987 Summer Conference*, pages 131–145, 1987.

[HBSG99a] O. Holder, I. Ben-Shaul, and H. Gazit. Dynamic layout of distributed applications in FarGo. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 403–411, Los Angeles, CA, May 1999.

[HBSG99b] 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.

[HBSG99c] 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.

[HBSG99d] 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.

[Hol98] O. Holder. The design of the FARGO system. Technical Report EE Pub No. 1171, Technion — Israel Institute of Technology, February 1998.

[HSW+04] 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 *Proceedings of Usenix File and Storage Technologies (FAST)*, April 2004.

[Lan] GNU Linker Command Language. Available at site: http://www.math.utah.edu/docs/info/ld_3.html.

[Org] World Wide Web Organization. Web services activity. Available at: http://www.w3.org/2002/ws/.

[PRW95] M. Neely D. Boles P. R. Wilson, M. S. Johnstone. Dynamic storage allocation: A survey and critical review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), 1995.

[RG97] E. Riedel and G. Gibson. Active disks - remote execution for network-attached storage, 1997.

[RGF98] E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 62–73, 24–27 1998.

[rmi99] Sun Microsystems Inc. *Java Remote Method Invocation Security Extension (draft)*, 1999. Available at: http://java.sun.com/products/jdk/rmi/.

[Ruw] T. M. Ruwart. OSD: A tutorial on object storage devices.

[Sie98] J. Siegel. OMG overview: CORBA and the OMA in enterprise computing. *Commun. ACM*, 41(10):37–43, 1998.

[Sri95] R. Srinivasan. RPC: Remote Procedure Call protocol specification version 2, 1995.

[Sun98] Sun Microsystems, Inc. *Java Remote Method Invocation (RMI) Specification*, October 1998. Available at http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html.

[tig] Alteon Tigon tools. Available at: http://www.osc.edu/~pw/tigon/,.

[Uni92] C. University. School of computer science, 1992.

[WADK06] 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.

[WBS02] 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.

[WDWAa] Y. Weinsberg, D. Dolev, P. Wyckoff, and T. Anker. Accelerating distributed computing applications using a network offloading framework. In *IPDPS'07*.

[WDWAb] Y. Weinsberg, D. Dolev, P. Wyckoff, and T. Anker. Hydra: A novel framework for making high-performance computing offload capable. In *LCN'06*.

[WJPR04] A. Wagner, Hyun-Wook Jin, D.K. Panda, and R. Riesen. Nic-based offload of dynamic user-defined modules for myrinet clusters. *cluster*, 0:205–214, 2004.

[Y. 07] Y. Weinsberg. *An Operating System Specification for Dynamic Code Offloading to Programmable Devices*. PhD thesis, 2007. Supervisor-Prof. Danny Dolev.

[ZKW02] Q. Zhang, C. Keppitiyagama, and A. Wagner. Supporting mpi collective communication on network processors. *cluster*, 00:75, 2002.