Tapping into the Fountain of CPUs — On Operating System Support for Programmable Devices *

Yaron Weinsberg †, Danny Dolev, Tal Anker ‡
The Hebrew University Of Jerusalem
{wyaron,dolev,anker}@cs.huji.ac.il

Muli Ben-Yehuda
IBM Haifa Research Lab
muli@il.ibm.com

Pete Wyckoff
Ohio Supercomputer Center
pw@osc.edu

Abstract

The constant race for faster and more powerful CPUs is drawing to a close. No longer is it feasible to significantly increase the speed of the CPU without paying a crushing penalty in power consumption and production costs. Instead of increasing single thread performance, the industry is turning to multiple CPU threads or cores (such as SMT and CMP) and heterogeneous CPU architectures (such as the Cell Broadband Engine). While this is a step in the right direction, in every modern PC there is a wealth of untapped compute resources. The NIC has a CPU; the disk controller is programmable; some high-end graphics adapters are already more powerful than host CPUs. Some of these CPUs can perform some functions more efficiently than the host CPUs. Our operating systems and programming abstractions should be expanded to let applications tap into these computational resources and make the best use of them.

Therefore, we propose the HYDRA framework, which lets application developers use the combined power of every compute resource in a coherent way. HYDRA is a programming model and a runtime support layer which enables utilization of host processors as well as various programmable peripheral devices' processors. We present the framework and its application for a demonstrative use-case, as well as provide a thorough evaluation of its capabilities. Using HYDRA we were able to cut down the development cost of a system that uses multiple heterogenous compute resources significantly.

Categories and Subject Descriptors D.2.11 [*Software Architectures*]: Domain-specific architectures; D.3.4 [*Processors*]: Runtime environments; C.0 [*General*]: System Architectures

General Terms Design, Experimentation

Keywords Offloading, Operating Systems, Programming Model

* To appear in ASPOLOS08

† Currently affiliated with Microsoft Corporation

‡ Also affiliated with Radlan-Marvell

ASPLOS'08, March 1–5, 2008, Seattle, Washington, USA.

1. Introduction

Today's modern operating systems (OSes) are complex programs that perform multiple tasks, doing much more than just multiplexing the computer's hardware among applications. An OS provides many of the programming APIs and run-time libraries needed by application developers. Even the simplest task, such as connecting to a peer host over a network, is performed by user level libraries and complementary kernel runtime support.

State-of-the-art peripheral devices allow one to program the peripheral device and adapt its functionality. For example, modern graphic adapters can perform matrix operations much faster than host CPUs. Today peripheral devices are largely ignored and their increasingly powerful computational capabilities are not being exploited. If peripheral devices could be adapted dynamically to an application's needs, and if their extra computing power could be harnessed to serve the application, developers would be able to create larger and more powerful computer systems.

This paper considers a model in which 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 performance. Using programmable devices has traditionally been very difficult, requiring experienced embedded software designers to implement conceptually simple tasks. In such cases, interfacing any new device feature with the host operating system would have to be performed from scratch and customized for the particular design. Cross-compilation tools and remote debugging environments do make programming tasks simpler, but integration with the host operating system is still difficult. The need for new abstractions and tools for programming such heterogeneous multi-core systems is apparent.

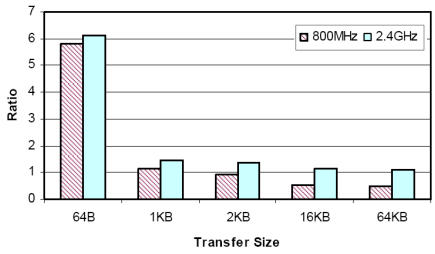
1.1 Offloading and Onloading

Offloading has been traditionally synonymous with TCP Offload Engine (TOE) devices (3). Although TOE is a controversial topic, it is agreed that TOE devices perform well for specific types of workloads and applications (7). The offloading concept can be generalized to any programmable peripheral device and extended to include more than network protocols. For example, file system related functionality such as indexing or searching could be offloaded to a programmable disk controller. 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.

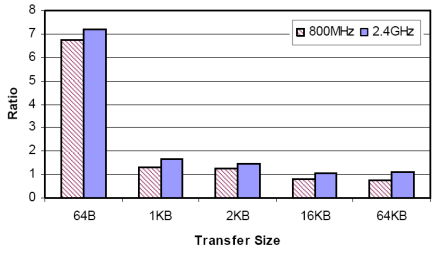
An offloading skeptic will typically claim that although peripheral devices are powerful, today's PCs have several underutilized

host processors that could be used instead. In response, we present the following arguments in favor of offloading:

1. *Memory bottlenecks* — Modern processors have large L2 caches in order to minimize cache misses caused by application execution and context swapping. Operations running on peripherals utilize local memory and filter out the information that needs to be brought to and from main memory, hence reducing memory pressure and cache misses on the main processors.
2. *Timeliness guarantees* — Operations running on peripheral devices can benefit from real-time programming paradigms. A peripheral device can provide operation timeliness guarantees that cannot be matched by a general purpose kernel (13).
3. *Reduced power consumption* — There is a major effort to reduce the power consumption of modern processors, which can be significantly aided by offloading. A Pentium 4 2.8 GHz processor consumes 68 W whereas an Intel XScale 600 MHz processor, commonly found in peripheral devices, consumes 0.5 W, two orders of magnitude less. By offloading suitable operations to low powered peripherals, we reduce the overall system power consumption.
4. *Increased throughput* — Network bandwidth has reached the point where host CPUs can spend all of their cycles just processing network traffic (5). Specifically, Figure 1 shows the GHz/Gbps Ratio ($= \frac{\%\text{cpu\_util} \times \text{processor\_speed}}{\text{throughput}}$) in the transmit and receive cases for different transfer packet sizes.[1] TCP offloading is a special case of offloading. Our current work suggests further opportunities in the area of network offload.



(a) GHz/Gbps Transmit Ratio



(b) GHz/Gbps Receive Ratio

Figure 1. GHz/Gbps Ratio

A recently proposed alternative to offloading is "onloading." Rather than moving functionality to the device, "onloading" proposes using host processors for I/O processing. For example, the Piglet (8) operating system dedicates one or more host CPUs to provide a "Virtual Device Interface." Although onloading part of the device's functionality to a host processor can yield better performance, eventually the data will still need to be transferred between the host CPU and the device and will then incur the bus-crossing overhead. We also note that Piglet can complement our proposed framework.

Another onloading direction has been recently proposed by Regnier et al. (9). The authors proposed to use one of the host's processors for TCP processing while using several techniques for reducing the protocol computation, data manipulation, and interrupt handling overheads. A step forward in this direction is to fully integrate the network controller with the host CPU (1). This work presents a simple integrated NIC (SINIC) device that is equivalent to a conventional NIC and is integrated with the host CPU. The SINIC device utilizes zero-copy techniques and was shown to significantly improve the host's throughput.

In the remainder of this paper, we present an offloading framework called HYDRA and an example application that derives great benefit from the framework. The TiVoPC is a software implementation of the commercial TV appliance Tivo (12). A classical Tivo appliance is a set-top box that allows for digitally recording all of one's favorite TV shows, and enables playback of them at a later time. Our implementation of the Tivo appliance provides a selected subset of Tivo's features. Specifically, we provide online-recording while watching a media stream and support its playback at a later time. A typical user-space software implementation of such an appliance would require the following components listed in Table 1.

Component	Description
GUI	Provides the viewing area and user controls (play, pause, rewind and resume).
Streamer	Processes the media stream (either from network or storage).
Decoder	Decodes the MPEG media stream.
Display	Displays the movie on screen.
File	Reads or writes previously stored data from storage.

Table 1. TiVoPC Components Outline

When analyzing a TiVoPC operation, one can see that a major part of the application logic is invested in transferring packets from one peripheral device to another. Specifically, the Streamer component transfers each received packet to the File component, in order to support future playback, and to the Decoder component. The decoding component hands a decoded frame to the Display component, which transfers the raw video frames to the graphics subsystem.

In order to demonstrate the use of our framework we have implemented a version of the TiVoPC application that uses multiple peripheral devices. The resulting data flow of the offload-aware TiVoPC application is presented in Figure 2. Once a packet is received at the NIC, it is directly transferred to both the GPU and the disk controller.[2] A decoder component running on the the GPU can directly decode the MPEG stream and transfer each frame to the GPU's internal framebuffer, making it appear in the GUI window without involving the host CPU at all. In case a user wishes to replay the stored media stream, a Streamer component running on the disk controller will transfer previously stored packets to the Decoder. Section 6 describes the implementation in detail.

The rest of the paper is organized as follows: Section 2 presents the requirements from an offloading framework and discusses some of the challenges. Section 3 provides an overview of the programming model and Section 4 discusses the software architecture. Sec-

[1] These figures appear in Foong et al. (5) and are used with the authors' permission.

[2] Note that if the bus architecture allows it (e.g., PCIe), this packet could be transferred in a single bus transaction.

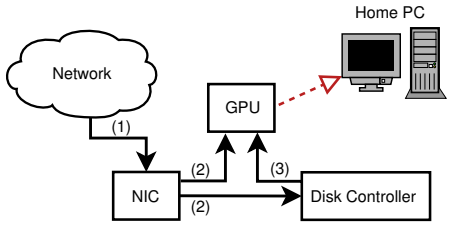


Figure 2. TiVoPC Data Flow

tion 5 provides a mathematical formulation for optimizing complex offload scenarios. Section 6 provides a case study for developing an offload-aware application and evaluates its performance. Finally, Section 7 presents the related work, Section 8 discusses some future directions and Section 9 concludes the paper.

2. Bridging the Offloading Gap

Offloading code to a programmable device today is a manual, tedious process. Offloading stand-alone code is difficult; offloading a software component that is part of a larger system with complex interdependencies even more so. In this section we present the requirements from an offloading framework and some of the challenges inherent in offloading and in creating an offloading framework in particular. Offloading code to a programmable device requires the following (manual) steps:

- Write it with the specific constraints of the target environment in mind: Does it have an MMU? What sort of run-time support does it have? Does it support dynamic memory allocation? Is there a toolchain that targets that device for the programmer's preferred language and environment?
- Compile and link it, using a device-specific toolchain. Some of the device-specific aspects mentioned previously might be handled by the toolchain. Linking is usually done with the device's run-time support libraries, which constrains the programmer to using an API specific for that particular device.
- Deploy it on the device. Each device has its own process for transferring the code from host memory to the device, such as through a firmware update.

Additionally, writing offloaded code presents several challenges. First, there is a steep learning curve. An offload programmer needs to be acquainted with all the relevant hardware specifications and the relevant SDKs. Offcode programming typically also requires kernel level development skills. Second, it requires embedded development skills. Usually, it will take an experienced embedded engineer to develop an efficient, stable and robust system. Third, it requires dealing with performance issues, e.g., how does one communicate efficiently with code running on the host CPU? This makes getting inter-component information transfer working correctly and efficiently tricky. Finally, the bulk of the work needs to be redone for every new device.

An offloading framework should facilitate and automate as many of the aforementioned steps as possible. It should also ease the aforementioned challenges of writing offloaded code. The holy grail is for the programmer to be completely unaware of the fact that parts of the system she is writing will be running on a programmable device.

To achieve these goals, an offloading framework must meet the following requirements: (1) it should not require the programmer to learn a new language or a new environment; (2) it should abstract the specific details of given devices as much as possible, so that the framework will handle the adaptation of the offloaded code to a specific offload target, rather than the programmer; (3) it should ease deployment, by deciding when and where to deploy a given component, as well as facilitating communication of the deployed component with the rest of the components, regardless of which host or device CPU they are running on.

3. Programming Model

The programming model provided by HYDRA (16) enables an application developer to design offload-aware applications. Such applications can utilize any available computing resource that offers programmability support. The model proposes an object-oriented methodology for developing such applications. Developers use a set of special components called Offcodes. An Offcode is a component that contains its state, a well-defined interface and a thread of control.

Communication between Offcodes is facilitated by communication channels with various communication properties as will be presented in Section 3.2. The programming model enables designing the application by using two orthogonal yet related aspects: the *Application logic* aspect and deployment aspect. The developer is encouraged to reuse Offcodes that are provided as a set of components from the vendor or custom made by the developer. The process of placing Offcodes at the peripheral devices involves defining the mapping between components and peripheral devices, both in software and hardware.

3.1 Offcodes

We envision openly accessed libraries of Offcodes that are provided as source code, or as object files that can be linked together with the target device's firmware. An Offcode is described by an Offcode Description File (ODF) that uses XML to describe the supported interfaces, dependencies on other Offcodes, and the target device's hardware and software requirements. A detailed description of the ODF file and the deployment process is given below.

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 GUID and is also described by the ODF file using the standard WSDL specification language (2). An offload-aware application (henceforth, 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 (*IOffcode*) that is used by the runtime to instantiate the Offcode and to obtain a specific Offcode's interface.

Offcodes are created by an OA-application by calling the runtime *CreateOffcode* API, which receives an ODF file name that identifies the target Offcode. 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 so that the can Offcode acquire its *local* resources. Since peer Offcodes may not have 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.

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.

3.2 Channels

Offcodes communicate with each other and with 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.

For high performance communication, a specialized channel that is tailored to the needs of the application and the Offcode would 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 to read, write and poll. The channel API also supports registration of a dispatch handler that is invoked each time the channel has a new request.

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.

Figure 3 presents the typical sequence of operations required to initialize a channel and connect it to a specific device. In this code, a reliable unicast channel is constructed with a zero-copy policy for read/write and sequential synchronization guarantees. A callback handler is then installed at the OA-application side of the channel. The corresponding handler is invoked by the runtime whenever data is available on the channel, as opposed to requiring the application to poll. Connecting an Offcode to a previously created channel is easily performed by calling the channel's *ConnectOffcode* method which takes the target Offcode reference as a parameter.

```
/* get our runtime and create the Offcode */
Runtime *rt = GetRuntime();
IOffcode *ocode = rt->v->CreateOffode(rt,"/offcodes/checksum.odf",
                                      &IID_Checksum);
/* get the channel executive */
ChannelExecutive *exec;
ErrorCode res = rt->v->GetOffcode(rt, "hydra.ChannelExecutive",
                                  &IID_ChannelExecutive, &exec);
/* set up the channel */
ChannelConfig config;
config.type = UNICAST_CHANNEL | RELIABLE_CHANNEL;
config.sync = SYNC_SEQUENTIAL;
config.buffering = DIRECT_READ | DIRECT_WRITE;
config.targetDevice = ocode->v->GetDeviceAddr(ocode);

/* create the channel to our target */
Channel *channel = exec->v->CreateChannel(exec, &config);
/* install a callback handler */
channel->v->InstallCallHandler(channel, MyHandler);
```

Figure 3. Creating a Channel

3.3 Offcode Manifesto

An Offcode manifesto is the means by which an Offcode defines its dependencies on peer Offcodes and its requirements from the target device and software environment.

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 binding name of the Offcode at the target device, and the Offcode's supported interfaces. We have chosen to use the WSDL specification for this purpose.

The second part describes the Offcode's dependencies on peer Offcodes. It enables a developer to "design" the offloading process that will occur at deployment time. HYDRA provides several constraints that can be used between any two Offcodes (denoted by α and β):

Link: The Link constraint is denoted as $\alpha \overset{Link}{\Leftrightarrow} \beta$. This is the default constraint from α to β, which actually poses no constraints: α and β may or may not be mutually offloaded (to the same or different target device). It does, however, indicate that at least one of the Offcodes needs the other to function.

Pull: 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.

Gang: The gang constraint is denoted as $\alpha \overset{Gang}{\Leftrightarrow} \beta$. This constraint is used to ensure that both Offcodes will be offloaded to **their respective** target devices. That is, if α is offloaded, β will be too, albeit on perhaps a different device.

Asymmetric Gang: This constraint is denoted as $\alpha \overset{\sim Gang}{\rightarrow} \beta$ and provides the asymmetric version of Gang. Offloading β doesn't imply offloading α.

The runtime processes an Offcode's ODF file to produce an *Offloading Layout Graph*. This graph is later used by the runtime for deciding on the actual placement of Offcodes. The last part of the ODF is concerned with device mappings. In order to enable dynamic mapping between Offcodes and peripheral devices, on different host configurations, a developer is required to supply a list of potential target *device classes* that can be used for offloading. Section 3.4 will further provide the motivation for this intentional design choice.

Figure 4 presents a typical import section defined in an Offcode's ODF. The fragment shows how an Offcode defines a *Pull* constraint to a peer Offcode. Also note that the developer only states the *class* of a potential device on which it can operate. It is the runtime's responsibility to locate an instance of such an Offcode which is suitable for running at one of the local devices that is in one of the listed classes.

3.4 Deployment Process

This section briefly presents the runtime deployment process. Figure 5 presents the various phases of this process. Once an Offcode is created by calling the *CreateOffcode* API, the appropriate Offcode ODF files are processed by the runtime to construct the application's offloading layout graph. Following that, the runtime determines the mapping between the Offcode device requirements and the physical devices that are installed in the specific host. This process uses the runtime resource management module and the resulting offloading layout graph. Typically, the runtime uses a local library that is used for storing the actual instances (object files) of the Offcodes. If such a mapping can not be allocated (due to resource limitations or incompatibility), the runtime tries to find an Offcode that is capable of executing at the host CPU.

```
<offcode>
 <!-- offcode package info -->
 <package>
  <bindname>hydra.net.utils.Socket</bindname>
  <GUID>7070714</GUID>

  <interface>
    <!-- WSDL interface specification >
    <include>"/offcodes/socket.wsdl"</include>
  </interface>
 </package>

 <!-- offcode dependencies -->
 <sw-env>
   <import>
     <file>"/offcodes/checksum.xdf"</file>
     <bindname>hydra.net.utils.Checksum</bindname>
     <reference type=Pull pri=0>
     <GUID>6060843</GUID>
   </import>
 </sw-env>

 <!-- device classes -->
 <targets>
   <device-class id=0x0001>
     <name>Network Device</name>
     <bus>pci</bus>        <!-- (optional) -->
     <mac>ethernet</mac>
     <vendor>3COM</vendor> <!-- (optional) -->
   </device-class>
   ...
 </targets>
</offcode>
```

Figure 4. Sample Offcode Description File

The next step involves adapting the specific Offcode instances to the target devices either by executing a corresponding compiler (for open source Offcodes) or by invoking the dynamic linkage process. The last phase is the actual offloading of the Offcode which is further described in Section 4.2.

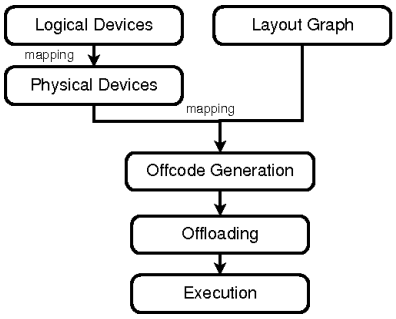


Figure 5. Deployment Control Flow

4. Architecture

In this section we present the architecture of the runtime system. The system implements the model and provides facilities for programming, testing, deploying, and managing OA-applications and Offcodes. Both the host OS and the target device firmware must support the interfaces defined by the programming API and implement the runtime functionality. A critical decision is to modularize the framework into independent parts, so that modifying one will not affect the rest.

Runtime library requirements for a particular target device may be provided by the device manufacturer, system integrator, or by researchers and the open source community. The second half of the runtime system exists on the host as operating system extensions. Our host implementation for Linux is modular, in that it maintains strict separation between device-specific code and generic code. It is implemented as a set of kernel modules that are loadable on demand and do not require kernel source code modifications.

The runtime is comprised of several components. It is accessed through the *Offloading Access Layer* (OAL) that consists of a *kernel-level layer* and *user-level layer*. The user layer is a loadable library module instantiated by each OA-application that uses offloading services. This layer interacts with the kernel layer via a generic, but operating-system specific, interface.

The kernel layer consists of several functional blocks. The *System Call Management* and *Offloading APIs* modules 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 will be provided as an extended driver for each programmable device. A channel provider is specialized in creating various 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. This module uses the *Offcode Depot* to store the logical mappings of Offcodes. 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 functions such as analyzing the offloading layout graph using the *Offload Layout Resolver*.

We distinguish between pseudo Offcodes and user Offcodes. Pseudo Offcodes are runtime components that happen to be implemented as Offcodes, but were not written by the user for a particular application. Components may be implemented as pseudo Offcodes because they export well-defined interfaces, or in order to reduce the processing time needed for dynamic linking of user Offcodes. Dynamic linking of user Offcodes requires resolving all undefined references of an Offcode binary while installing it at the target device—having the Offcodes communicate with the run-time through pseudo Offcodes is an easy way of limiting the number of symbols that need to be resolved.

One example of a pseudo Offcode is the "hydra.Runtime" Offcode which provides the runtime functionality through a well defined interface. The runtime's *GetOffcode* method enables a user Offcode to get an interface to any Offcode currently registered at the runtime by providing it the Offcode's GUID. Another example is the "hydra.Heap" Offcode, which provides an interface to the OS memory routines.

4.1 Channel Internals

Channels interconnect Offcodes and are created by the channel providers. Since a channel is hardware specific, it is created according to the target device-host interfaces. Figure 6 shows a sample zero-copy channel architecture implemented for a programmable NIC. The right side of the figure presents the logical view as seen from the OA-application while the left side presents the internal architecture.

The figure presents an OA-application that communicates with Offcode α through a proxy connected to a private channel identified by a channel descriptor. The HYDRA runtime maps each channel descriptor to an internal channel object that is created by the target device channel provider. This specific provider constructs two kernel buffer rings to communicate with the target Offcode. The *In-*

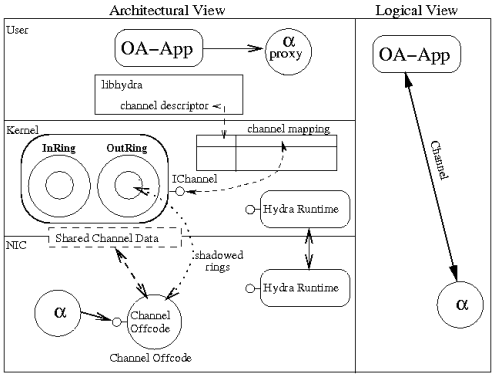


Figure 6. Example Zero-Copy Channel

Ring holds memory descriptors that point to host memory locations that contain the *Call* objects. Although a *Call* object usually contains a return descriptor for delivering the invocation return value, the *OutRing* is necessary since it contains pre-posted application descriptors that are used by the runtime at the device for spontaneous messages triggered by the Offcode. The channel endpoint at the device holds a shadowed copy of the ring descriptors; and, channel management is maintained using a dedicated shared memory region per channel. The *Call* object is copied using the NIC's DMA bus master capabilities to an internal buffer owned by the target Offcode. The *Call* is de-serialized and the Offcode is invoked. The Offcode uses the embedded return descriptor to DMA the return value back to the application and optionally notifies the application using an event (usually interrupt) described by the shared memory region.

4.2 Dynamic Offcode Loading

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

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

As a proof of concept, we have created such a loader for our programmable network card. The dynamic offloading logic is implemented both in the device and in the host. A device-specific host-based loader is implemented at the NIC's driver; it uses the OOB-channel of the device's runtime to communicate with the target device loader, which is actually a pseudo Offcode at the target device. The offloading process is performed in several phases. First, the host-based loader calculates the Offcode's size and invokes the *AllocateOffcodeMemory* function exported by the device's loader. This method allocates the memory region that will be used to store the Offcode binary and returns the device's memory address to the caller. The host-based loader dynamically generates a linker file adjusted by the returned address and links the Offcode object. It then transfers the linked Offcode to the target device where it is placed and executed.

5. Optimizing Complex Layouts

The strength of the proposed programming model lies in the the ability to reuse the Offcode components. On the one hand, reusability may simplify and speed-up the development cycle, but on the other hand, in multi-user environments, reusing the same Offcode in several applications may substantially complicate the offloading layout design. Intuitively, the problem of defining an optimal offloading layout graph for a group of offload-aware applications may introduce an infeasible combinatorial problem. This section provides an Integer Linear Programming methodology (ILP) for optimizing such complex layouts. The purpose of such a formulation is to enable expression of an offloading layout graph as a set of linear equations. Any ILP solver can then be used to solve the equations given a target optimization function. We provide the mathematical presentation of an offloading layout graph and later present, as an example, two possible criteria that could be used as target optimization functions. We note that simple graphs are usually trivial to solve, while for complex scenarios a greedy solution is not always optimal. Hence the need for the linear equation formulation.

5.1 Formulation

As the offloading layout design essentially produces a graph, it is natural to express the dependencies among the graph vertices (e.g., Offcodes) mathematically. This section provides the ILP formulation that is required for optimizing the offloading layout graph.

5.1.1 Definitions

We begin by defining the basic elements of the layout graph. The layout graph $G = (V,E)$ includes the set of Offcodes as vertices, and the channel constraints among them are the edges. At deployment time the runtime associates with each node n (Offcode) a compatibility target vector $\vec{C_n}$ representing the potential target devices that can host the Offcode. Note that the host CPUs are included in the list of devices. Let $N = |V|$ be the total number of Offcodes, and let $K = |\vec{C}|$ be the number of HYDRA compatible devices.

NOTATION 5.1 *Let $\vec{C}$ be a constant binary bit vector. $C_n^k = 1$ if Offcode n* can *be offloaded to device k. $\forall n \in N, k \in K$, $C_n^k \in \{0,1\}$.*

To simplify the presentation we assume the that the first entry in each vector $\vec{C}$ corresponds to the host CPUs.

NOTATION 5.2 *Let $\vec{X}$ be the ILP output vector. $X_n^k = 1$ if Offcode n* should *be offloaded to device k. $\forall n \in N, k \in K$, $X_n^k \in \{0,1\}$.*

The following equation guarantees unique placement of each Offcode, i.e. each Offcode can be offloaded to a single device:

$$\sum_{n=1}^{N} \sum_{k=1}^{K} X_n^k C_n^k = 1 \,. \tag{1}$$

Additionally, an Offcode n is **not** offloaded (remains in the host CPU) if $X_n^0 = 1$.

5.1.2 Constraints Formulation

For each one of the channel constraints (See Section 3.3), an integer linear equation is defined.

NOTATION 5.3 *Let $E_m^n = (m,n)$ be an edge from Offcode m to Offcode n.*

The following equations formulate the various channel constraints.

Pull Constraint:

$$\forall E_m^n \in \textit{Symmetric Pull}, \forall k : X_n^k = X_m^k \,. \quad (2)$$

Gang Constraint:

$$\forall E_m^n \in \textit{Symmetric Gang} : \sum_{k=1}^{K} X_n^k = \sum_{k=1}^{K} X_m^k \,. \quad (3)$$

Asymmetric Gang Constraint:

$$\forall E_m^n \in \textit{Asymmetric Gang} : \sum_{k=1}^{K} X_n^k \leq \sum_{k=1}^{K} X_m^k \,. \quad (4)$$

These equations are sufficient to represent the joint offloading layout graph as a set of linear equations.

5.1.3 Optimization Objectives

We have identified several optimization functions, two of which are presented below. The list is by no means complete; additional objective functions can be easily added to address various applications needs.

1. Maximized Offloading – This objective is used to offload as many Offcodes as possible. The motivation for such a goal is to minimize the CPU usage and memory contention at the host:

$$\max \left(\sum_{n=1}^{N} \sum_{k=1}^{K} X_n^k \right) .$$

2. Maximize Bus Usage – This objective aim is to fully utilize the bus interconnect bandwidth among devices. A "Price" value is assigned to each Offcode. This value represents the estimated *average* bus bandwidth that is required by the specific Offcode. The bigger the value, the more bandwidth is required by the Offcode. In addition, we define a capability matrix per host. This matrix describes the *maximal* bus bandwidth between every pair of peripheral devices. This matrix is used to limit the number of offloaded Offcodes as the ILP solution must reflect physical bus limitations.

6. Inside TiVoPC: A Framework Use Case

This section present a case study for developing the TiVoPC application using our proposed framework. We focus on showing how HYDRA simplifies the design and development of offload-aware applications.

6.1 TiVoPC Architecture

The system architecture of the TiVoPC application is presented in Figure 7. The figure shows a client-server architecture comprised of a *Video Server* and a *Video Client*. The *Video Server*, on the left hand side of Figure 7, corresponds to the cable TV broadcaster. Typically, Network Attached Storage (NAS) devices are used to store massive amounts of broadcasted media (MPEG movies, radio channels, etc.). In order to emulate such a broadcaster, we have implemented a software-based server that is executed on a standard PC. The server reads the media from a NAS device using NFS, and sends the media to the client as a stream of UDP packets.

The architecture of the *Video Client* is shown on the right hand side of Figure 7. The client PC hosts the following programmable peripherals:

- *NIC* – This device is connected to the multimedia streaming server and receives incoming UDP packets.
- *"Smart Disk"* – Although programmable disk controllers are common, in order to speed up prototyping, we have decided to emulate one by using a programmable NIC. Our "Smart NIC" exports a standard block device that interacts with an NFS server to store the data (i.e., the streamed video is effectively stored on a remote disk). Essentially, we have created an NFS Offcode that implements various parts of the NFS protocol.
- *GPU* – The graphics processing unit is responsible for rendering and displaying the movie.

6.2 TiVoPC Logic

As the programming model suggests (Section 3), the first phase in the development process should be designing the TiVoPC logic. This phase is usually performed without considering the physical placement of the various components. Figure 7 presents the following TiVoPC components.

- *GUI* – The user interface contains a viewing area, to display the received video stream, and several controls used to rewind, pause and play the movie.
- *Streamer* – This component handles incoming packets. Specifically it extracts the payload that contains the three types of MPEG frames: the I-frame, P-frame and B-frame. The component also processes data that is received from the storage device. It implements a *callback* method that is invoked each time a packet is received. Upon invocation, the *Streamer* extracts the payload and passes it to the *Decoder* component.
- *Decoder* – This component decodes the MPEG frame for later display on the screen, and works in conjunction with a *Display* component.
- *Display* – This component represents the display. For example, in a host-level implementation, this object could wrap an OpenGL FrameBuffer object or simply use a memory map of the GPU's physical memory.
- *File* – This component provides the basic file level APIs, such as open, read, write and close.
- *Broadcast* – This component is used at the *Video Server* to broadcast the movie frames to the client. This component provides unreliable message delivery as it uses UDP as its transmission protocol.

Some components could be omitted. For example, the Streamer could directly access the local file system using the standard APIs, without the need for an additional *File* object. Alternatively, the Decoder could directly manipulate the display without the need for another level of indirection that is realized as the *Display* component. Although these omissions are possible, introducing finer grained objects improves the flexibility of the design. For instance, if a *Display* Offcode for the local GPU is found, either locally or in the vendor's Offcode library, it will be used at the GPU, thus increasing overall application performance.

Once the components have been identified, we decide which of them will be implemented as Offcodes. Additionally, the Offcode communication channels are also specified. Following are three characteristics that typically indicate a component should be implemented as an Offcode:

1. The component can use specialized capabilities that exist only at a peripheral device.
2. Offloading the component reduces the amount of traffic on host busses.
3. The component is tightly coupled to another offloaded Offcode.

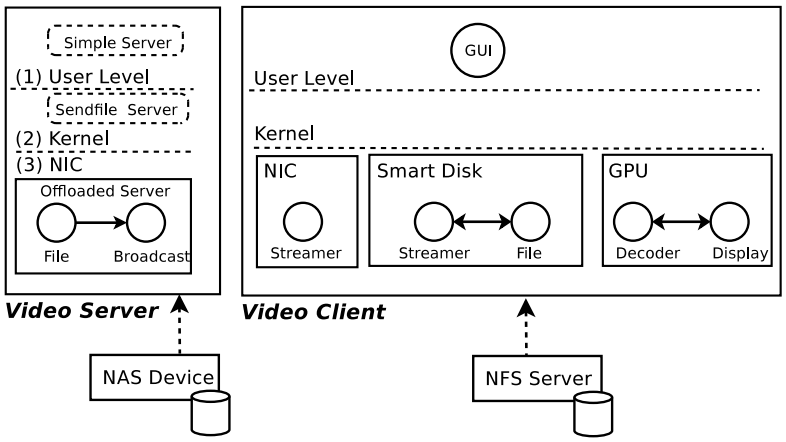


Figure 7. TiVoPC Software Architecture

In our example, all the components except the GUI fall into one of these three categories and thus will be implemented as Offcodes.

6.3 TiVoPC Offloading Layout

The offloading layout of the TiVoPC application matches an Offcode to a peripheral device. The ODF discussed in Section 3.3 contains this information in addition to the Offcode's constraints regarding its peer Offcodes. For brevity we omit the ODF details and instead provide considerations for designing the offloading layout as depicted in Figure 8.

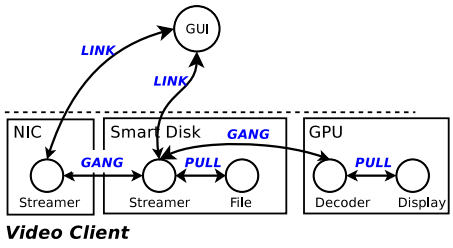


Figure 8. TiVoPC Offloading Layout

The *Streamer* Offcode resides at the NIC and at the "Smart Disk" devices. Reusing the same component at both devices is achieved by storing the received frames, without modification, at the storage device, so that the source of the media packet becomes oblivious to this component. Since we do not want packets to traverse the bus twice, a *Gang* constraint is imposed between the two components.

Intuitively, the *Display* Offcode should be placed at the GPU device, while the *Decoder* Offcode could be placed either at the NIC or at the GPU. In both cases, one bus transfer is required to move the media packet from the NIC to the GPU. The preference of placing the *Decoder* at the GPU comes form two reasons. First, the GPU may have specialized MPEG support on board. Second, a single *Decoder* could be used instead of duplicating the component at the NIC and at the "Smart Disk." In essence, requiring a *Gang* constraint between the two Offcodes will minimize the number of bus crossing operations. Therefore, the *Streamer* Offcode holds a *Gang* constraint to the *Decoder*, which holds a *Pull* constraint to the *Display*.

The *File* Offcode should reside at the "Smart Disk" and should be *Pulled* with the *Streamer* as both Offcodes tightly interact while the movie is stored to or loaded from the storage device.

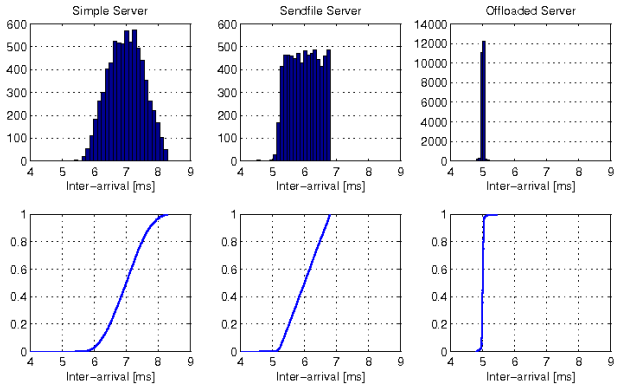


Figure 9. Jitter Distribution

A simple *Link* constraint is sufficient between both *Streamers* and the *GUI* since only control information passes between them. As this is the default channel constraint, it can be omitted from the layout specification.

Once the application logic and the offloading layout have been coded, communication channels between the various components are set. In the TiVoPC application, we use a zero-copy read/write channel for all communication channels except for the two channels between the *GUI* and the *Streamer* Offcodes. Communication between the *GUI* and the *Streamers* utilize the default, low priority, *OOB-Channel*.

6.4 Benchmarks Description

Our experimental test-bed consists of two 2.4 GHz Intel Pentium IV computers with 512 MB RAM and 256 kB L2 cache. The hosts are interconnected by a Dell PowerConnect 6024 Gigabit switch through a programmable 3Com 3C985B-SX NIC. The hosts execute Linux kernel version 2.6.15-1. The TiVoPC server is configured to send a single 1 kB packet every 5 ms. For demonstration purposes only, we did not send packets at video frames boundaries. What we did is to send the video stream in arbitrary chunks of 1 kB while maintaining the required bit rate. Specifically, for a video stream of 200 kB/s we send a 1 kB chunk every 5 ms. We executed the following benchmarks on an idle system.

Video Server Packet Jitter

Three versions of the *Video Server* have been implemented as indexed by the numbers 1–3 at the left hand side of Figure 7.

The first implementation (indexed by number 1) uses two UDP socket endpoints. Every 5 ms, a movie frame is read to a statically allocated buffer of size 1 kB, then a connected UDP socket targeted at the client host is used to send the packet to the TiVoPC client.

The second implementation (indexed by number 2) utilizes the "sendfile" system call. This call operates in two steps. In the first step, the file content is copied into a kernel buffer by the device's DMA engine. In our case, the server uses a NAS to store the movies, hence the NIC is the one that acts as the DMA master. In the second step, a socket buffer is initialized with the required information about the location and length of the data just received. Scatter-gather hardware support is required at the networking device in order to be able to handle such a socket buffer. In cases where the hardware fails to support this feature, the CPU copies the data to the socket buffer.

The third implementation is an offload-aware server (indexed by number 3). This server is implemented as a simple Offcode residing at the networking device. It uses the *File* Offcode to read the data from the NAS device, and the *Broadcast* Offcode to transmit the data to the client.

Figure 9 shows, for each server implementation, a histogram and the corresponding cumulative distribution function (CDF) of packet jitter as measured at the client machine. A low level of jitter is more important than reliable delivery in video applications, as an unsteady packet rate is easily detectable by a human viewer. Figure 9 clearly shows that the offloaded version of the streaming server produces a significantly lower jitter. This observation is further strengthened by the corresponding CDF. The user level version that uses "sendfile" produces better results than the "Simple Server" due to fewer context switches and data copying operations.

Table 2 provides the jitter statistics of received packets corresponding to the execution of the three servers.

Scenario	Median	Average	Std Dev
Simple Server	6.99	7.00	0.5521
Sendfile Server	6.00	5.99	0.4720
Offloaded Server	5.00	5.00	0.0369

Table 2. Client Side Jitter Statistics

Video Server CPU Utilization

This benchmark is intended to validate our assumption that offloading certain parts of an application will reduce pressure on the host memory subsystem. The L2 cache miss rate that is experienced by the kernel is measured on the server during each one of the following tests. Samples were taken every 5 seconds during a 10 minute run. All measurements were normalized to the miss rate experienced by an otherwise idle system. Figure 10 shows the results. Although the TiVoPC application is mostly I/O bound, executing it at the host incurs a 7% increase in the L2 cache miss rate, as seen in the "Simple Server" column.

The second implementation uses the "sendfile" API that avoids unnecessary buffer copies between kernel and user buffers. As indicated by Figure 10, the effect on the L2 cache is negligible. The reason becomes clear as the "sendfile" source code is examined. Since most of today's network devices support scatter-gather operations, the kernel essentially follows a zero-copy data path between the two sockets. This approach reduces the number of context switches and totally eliminates data duplication inside the kernel. The third implementation, "Offloaded server," achieves the best results, with L2 cache miss rates on par with an idle system.

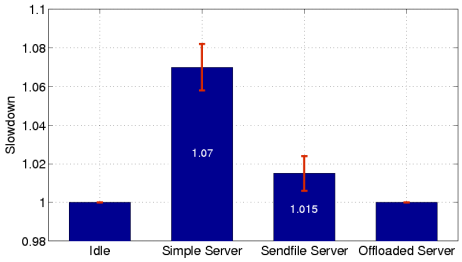


Figure 10. L2 Slowdown (Server Side)

Scenario	Median	Average	Std Dev
Idle	2.90%	2.86%	0.09%
Simple Server	7.50%	7.50%	0.12%
Sendfile Server	5.90%	6.20%	0.08%
Offloaded Server	2.90%	2.86%	0.09%

Table 3. Server Side CPU Utilization

Table 3 presents the CPU utilization at the server side. Each row corresponds to one of the three scenarios presented in Figure 10. Notice that the CPU utilization of the offloaded version of our server aligns with the *Idle* scenario results, as the host processor is oblivious to the underlying activity.

Video Client Memory and CPU Utilization

The client side implementation, shown on the right of Figure 7, involves five offloaded components, compared to the two components in the server, and more interesting constraints. However, its performance results closely resemble the server side results.

Scenario	Median	Average	Std Dev
Idle Client	2.90%	2.86%	0.09%
User-space Client	7.30%	6.90%	0.32%
Offloaded Client	2.90%	2.86%	0.09%

Table 4. Client Side CPU Utilization

Table 4 shows that the offloading is complete in the sense that there are no components left on the host processor. An idle machine and a machine that is running the fully offloaded client both consume the same background level of CPU cycles. The non-offloaded user-space client consumes more CPU, although the load is very small compared to the total capability of the host processor. In terms of L2 cache misses (not shown), the idle machine and offloaded client have the same count, while the non-offloaded client generates 12% more misses. Much of this is due to the MPEG decoding process.

7. Related Work

Most of today's work focuses on extensions to **specific** peripheral devices. This section describes the state-of-the-art in offloading research, ordered by its relevance to this work.

Spine (4) is a safe execution environment for programmable network interface cards. It enables the installation of user handlers, written in Modula-3, on the NIC. Applications and extensions communicate via a message-passing model based on Active Messages (14). Although Spine enables the extension of host applications to use NIC resources, it has several major limitations. First, since all extensions are executed when an event occurs, building stand-alone applications on the NIC is difficult. Even for event-driven applications, the developer is forced to dissect the applica-

tion logic to create a set of handlers. Second, Spine's runtime does not support the deployment process of handlers or provide a way to design the offloading aspects of the host application.

Object-based Storage Devices (OSD) grew from a research project called Active Disks from CMU (10) and have recently been standardized by the ANSI T10 group (15). OSD is a protocol that defines higher-level methods for the creation, writing, reading and deleting of data objects on a disk. Implementing OSD requires a high degree of processing capability on the disk controllers or the devices themselves and can offer the potential for extension.

Although not dealing with offloading, **FarGo** (6) and **FarGo-DA** (17) propose a programming model that enables a developer to specify relocation and disconnection semantics in a separate phase during the application development cycle. The basic assumption for this work is that the application is fully comprised of a set of components that are tagged by a specific interface. The components are hosted in a virtual machine and can migrate to a remote VM using marshaling and unmarshaling mechanisms. Our framework goes beyond this model by defining an offloading layout that is used to define the offloading aspects of the application.

Ethernet Message Passing (EMP) (11) is a zero-copy and OS-bypass messaging layer for Gigabit Ethernet. EMP protocol processing is done on the NIC and a host application (usually through an MPI library) can directly manipulate the NIC. It provides very low message latency and high throughput but is very task-specific for MPI and lacks the support for generic offloading or host application integration.

8. Future Work

In the near future a handful of underutilized computing resources will be available in any home PC. Treating these computing resources as first class citizens by seamlessly offloading computation to them whenever possible will enable a new breed of applications. This section briefly presents some of the potential fields that will benefit from offloading capabilities.

Virtualization – Rapidly improving virtualization technologies allow one to run multiple OSes simultaneously on one physical machine, as "virtual machines." Offload-capable devices could perform more efficiently some of the tasks that are performed today on the host CPUs, such as multiplexing incoming network packets directly to the destination virtual machine.

Advanced Storage Services – Programmability support that will soon be offered by advanced disk controllers will open new possibilities for implementing advanced storage services directly inside the disk or controller. Programmable disks will provide an opportunity to run I/O-intensive computations efficiently by running them closer to the data. Potential applications include content indexing and searching, virus scanning, storage backup, mirroring, snapshots and continuous data protection.

9. Conclusion

Hardware and software are neck and neck, pushing each other forward. This paper claims that it is the OS's turn to act. Hardware manufacturers have provided an excessive amount of computing resources, which rest idle most of the time. It is time for the OS community to design tools and programming abstracts that will enable a developer to efficiently utilize every programmable component in the system. As a proof of concept, we have presented the HYDRA framework which proposes a unique new dimension of flexibility for the architects of high performance applications: the ability to program offloading layout policies separately from the application's logic. We have developed a programming model that carefully balances between programmer scalability and system scalability. We believe that programmable devices will continue to grow in popularity. It is only a matter of time until an OS on a workstation or a PC will be considered as a switching element among heterogenous processing cores.

Acknowledgments

We would like to thank Maxim Grabarnik for spending many hours working on the TiVoPC application. We would also like to thank Ittai Abraham and Adamovsky Olga for assisting in the ILP formulation. Special thanks go to Galen Hunt and Prof. Ken Birman for the opportunity to discuss this work with them and for their valuable insights regarding the framework.

References

[1] N. L. Binkert, A. G. Saidi, and S. K. Reinhardt. Integrated network interfaces for high-bandwidth TCP/IP. In *ASPLOS-XII: Proc. of the 12th intl. conf. on arch. support for programming languages and operating systems*, pages 315–324, New York, NY, 2006. ACM Press.

[2] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Service Definition Language (WSDL). http://www.w3.org/TR/wsdl.

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

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

[5] A. P. Foong, T. R. Huff, H. H. Hum, J. R. Patwardhan, and G. J. Regnier. TCP performance re-visited. In *ISPASS '03: Proc. of the 2003 IEEE intl. symp. on perf. analysis of systems and software*, pages 70–79, Washington, DC, 2003. IEEE Computer Society.

[6] 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, 1999. IEEE Computer Society Press.

[7] J. C. Mogul. TCP offload is a dumb idea whose time has come. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 25–30, Berkeley, CA, 2003. USENIX Association.

[8] S. J. Muir. *Piglet: an operating system for network appliances*. PhD thesis, 2001. Supervisor: Jonathan M. Smith.

[9] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP onloading for data center servers. *Computer*, 37(11):48–58, 2004.

[10] 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, 1998.

[11] P. Shivam, P. Wyckoff, and D. Panda. EMP: zero-copy OS-bypass NIC-driven gigabit ethernet message passing. In *SC'01*, Nov. 2001.

[12] TiVo Inc homepage. Available at site: `http://www.tivo.com`.

[13] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick. System noise, OS clock ticks, and fine-grained parallel applications. In *ICS '05: Proc. of the 19th annual intl. conf. on supercomputing*, pages 303–312, New York, NY, 2005. ACM Press.

[14] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

[15] R. O. Weber. Information technology—SCSI object-based storage device commands -2 (OSD-2). Technical Report T10/1731-D, INCITS Technical Committee T10, Oct. 2004.

[16] Y. Weinsberg. *An Operating System Specification for Dynamic Code Offloading to Programmable Devices*. PhD thesis, The Hebrew University Of Jerusalem, October 2007.

[17] Y. Weinsberg and I. Ben-Shaul. A programming model and system support for disconnected-aware applications on resource-constrained devices. In *ICSE '02: Proc. of the 24th intl. conf. on software engineering*, pages 374–384, New York, NY, 2002. ACM Press.