Intelligent Devices as Symmetric Partners for End–to–end Data Flows *

Mike Wawrzoniak Nadia Shalaby Larry Peterson
{mhw,nadia,llp}@cs.princeton.edu
Department of Computer Science
Princeton University

July 26, 2002

Abstract

The recent emergence of end-to-end services and multimedia networking applications has led to the desire to support data flows between arbitrary devices. Connecting heterogeneous devices presents an $N \times N$ programming problem since each device must be able to interface with each other device, as well as with code modules that transform the data stream. This paper defines a *partner* architecture that allows the application builder to connect devices to each other—as well as to code modules that transform the data—by wrapping both modules and device drivers in a common interface. Moreover, this interface is *symmetric* in the sense that the device on either side may be either the master or the slave. The paper demonstrates the generality of the partner architecture by giving examples of how video, audio, filesystem, network, and sockets "devices" use the interface. It also presents performance numbers that show that the overhead imposed by the architecture is very small.

1 Introduction

The recent emergence of end-to-end services and multimedia networking applications has led to the desire to support data flows between arbitrary devices. Such applications often need to establish connectons between various devices, in effect providing support for device-to-device data streams; e.g., from a video capture card to the network card, or from the network card to audio device. To connect one device to another, both source and destination devices need to understand each other's interfaces. Unfortunately, the plethora of existing device types all seem to support their own specialized interface, making the realization of an arbitrary device-to-device data switch an $N \times N$ programming problem. To make matters worse, device (and kernel) programming productivity tends to lag its application level counterpart by a factor of three to four [3].

This paper describes a uniform interface that we can wrap around devices, so we can easily interconnect them. The interface is sufficiently general to accommodate a wide range of heterogeneous devices, yet minimal enough to be efficient. Several additional aspects make the design of such an interface a challenge. To satisfy the emerging applications while still maintaining the

*This work supported in part by DARPA contract N66001–96–8518, NSF grant ANI-9906704, and Intel Corporation.

uniformity of our interface, we extend the concept of a "device" beyond peripheral devices (such as network, video and sound cards), to other parts of the kernel (such as the file system), user space (typically accessed via sockets), and even generic pieces of kernel code (such as loadable kernel modules). For example, data might flow from the network card, via a kernel module that processes it, onto an application residing in user space, and at each interconnection, these "devices" communicated via our uniform common interface. Furthermore, our generalized concept of a device might even extend to an external processor, such as a network processor.

Another challenge in the interface design is to account for the fact that given any pair of devices to connect, one or both might want to be the master, with its corresponding driver(s) executing a thread on their behalf. This requires the interface to allow for the point of control being on either side. Moreover, a device might want to serve as input or output. Therefore, to accomodate for all such cases the interface must be symmetric.

This work was inspired by the authors' experience dealing with device drivers when building the x-kernel [6] and Scout [10] operating systems, and the problems we encountered. Extending the design of the partner interface beyond peripheral devices to include code modules in particular, greatly facilitated the realization of Silk [1], which embedded Scout in the Linux kernel.

To this end, we propose the Partner Architecture, a generic interface abstraction that views devices as *symmetric partners* communicating through a *uniform interface*, and this functionality could be added *quickly and seamlessly.* This enables us to keep up with the growing end-to-end applications, facilitate the provision of such infrastructural support in a system, as well as boost programming productivity. To reiterate, the partner architecture was designed with five goals in mind:

1. To provide a clean and minimal level of abstraction without impairing functionality.

2. To be extensible to internal devices (such as file systems), external devices (such as user space interfaces), and devices with varying degrees of intelligence (from a microphone to full fledged external processors).

3. The interface must treat devices as symmetric partners, exporting the same API from each side.

4. Programming new devices to the partner interface should be efficient, simple and fast.

5. It should not adversely impact, but rather improve, performance. For example, by avoiding data copies among modules internal to the kernel, or by centralizing the data flow scheduling logic into one place, thus enabling rigorous optimizations of that section of the code.

The rest of the paper is organized as follows. In the next section we present an architectural overview for system interface boundaries as well as the router OS where we implemented a partner architecture prototype, namely, Silk, Scout in Linux kernel. We then describe the partner architecture in Section 3. We discuss examples of partner drivers in Section 4. Geni, a partner driver for the networking stack of our OS module, Scout, is a more complex example of a partner driver, and is therefore discussed separately in Section 5. We evaluate extensiblity and performance of the partner architecture in Section 6. Related work is briefly covered by Section 7. Finally, section 8 concludes the paper.

2 Architectural Overview

Before describing the architecture, we first conceptualize our view of devices and their interfaces. Even though our architecture provides an abstraction for this conceptual view, and is thereby applicable on any systems platform, we still had to implement our prototype within a concrete system. We chose to implement it within a specialized Linux module, Silk, which preserves the functionality of Scout paths on a Linux OS. We therefore briefly present an overview of Scout paths, their encapsulation into Silk, and a specialized driver Vera, which provides a richer device interface allowing to connect the Linux processor to other more specialized networking processors.

2.1 Devices and Interfaces

We assume an OS running on one or more symmetric processors and take a *driver-centric* view of the world, as shown in Figure 1. From this perspective, not only devices and file systems are viewed as drivers, but we take the same view towards user space interfaces, kernel modules, as well as a potential connection to another processor with its own OS via PCI bus encapsulation. The OS kernel provides the medium via which these devices communicate. For hierarchical networking systems, such as those described in [8, 9, 14], our generalized devices can be found at any of the various levels of the hardware–software hierarchy, which we loosely refer to as *execution levels*.

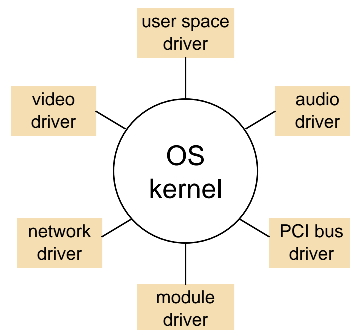

Figure 1: The operating system takes a driver-centric view of the world.

On a typical system, we dissect the driver-centric view of Figure 1 into a hierarchy of execution levels, starting with user space from above and ending with hardware from below, as is depicted in Figure 2. Each execution level of the system hierarchy exports a programming interface to its adjacent level(s). For our purposes, we view the system with four execution levels: user space, kernel space, device driver space and hardware. Such a dissection results in three execution boundaries, described as follows.

OS specification between kernel and user space, provides systems calls to all OS components, such as sockets, filesystems, threads and processes, timers and monitors, security and real time support, etc. Many such APIs exist, including POSIX, Single UNIX Specification versions 2 and 3, AES, XPG4 base and SVID3 base.

Driver specification between the OS and a particular driver. Typically, since most devices have a separate driver for every OS, these specs tend to be much more kernel dependent. Examples would be video, audio and network card device drivers written for a Linux style or NT style OS.

Hardware specification between the driver and hardware execution levels. Every manufactured hardware device provides a hardware spec defining how to program it, indispensable for driver development. For example, every network interface card (NIC), microphone or graphics card provides such a hardware specification interface.

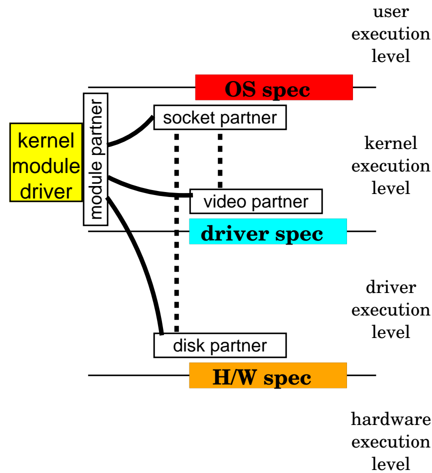

Figure 2: The partner driver of the kernel module interfacing to various "devices" within a system's execution level hierarchy: socket partner interfaces to an OS specfication (between user and kernel space), video partner interfaces to the device's specification (between kernel and device driver space), and disk partner interfaces to a disk's hardware specification. The dotted lines are raw data flows between the partner interfaces.

We designed a uniform interface, called the *partner* interface, that all devices at any execution level export. The idea is to translate the exporting interface of any device to this partner interface. Some such partner drivers, and their mapping onto the execution levels described are shown in Figure 2. Namely, socket, video and disk drivers are depicted as examples. Additionally, we portray a kernel module as a device. Within this kernel–embedded OS, devices can choose to interact directly via the partner interface, without going through the module driver. For example *raw data* can flow between a network interface card and an application via sockets, depicted as dotted lines.

To further elaborate on such raw flows of data, we subdivide generic devices into peripheral devices, such as disk drives, network and sound cards, etc., the user space device, kernel modules, and devices that drive other systems across a bus, , such as the PCI bus. Figure 3 (a) shows how the partner interface (shaded area) can connect two peripheral devices, such as disk to video. Likewise, in (b), a peripheral device can connect to user space, such as network card to sockets. Peripheral devices may also interface to kernel modules as shown in (c), or other processors via a system bus, shown in (d).

Alternatively, rather than constructing a raw data path between devices, data can flow through a kernel module, which may modify it before forwarding it to another device. Figure 3 (e) depicts such a scenario, where a device such as a network card intefaces to a kernel module with its own networking stack. We further elaborate on such a kernel module used in our prototype in the next Section 2.2. The possibly modified data may then flow via the PCI bus to another processor, by

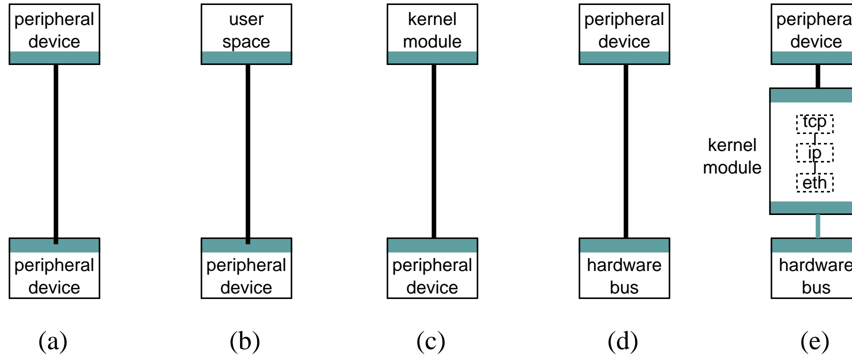

Figure 3: Different kinds of devices connect via the common partner driver interface: (a) peripheral devices to each other, such as sound card and disk; (b) peripheral devices to user space, such as network card to sockets; (c) peripheral devices to kernel modules, such as network card to a loadable kernel module; (d) peripheral devices to external processor, such as a data flow from a file system to a network processor connected via a PCI bus; and (e) a data flow from a peripheral device, through a kernel module, to an external processor, such as a TCP flow between a network card, via the TCP kernel module, to another processor via a PCI bus.

means of the hardware bus device. We describe an example of such a device in Section 2.3.

2.2 Scout Paths in a Kernel Module

Designed to encapsulate I/O flows, a *Scout path* [10] is a structured system activity that provides an explicit abstraction for early demultiplexing, early packet dropping when flow queues are full, resource accounting per flow, explicit flow scheduling, as well as extensibility in adding new protocols and constructing new network services. Each Scout path consists of a string of protocol modules, and encapsulates a particular flow of data, such as a single TCP connection.

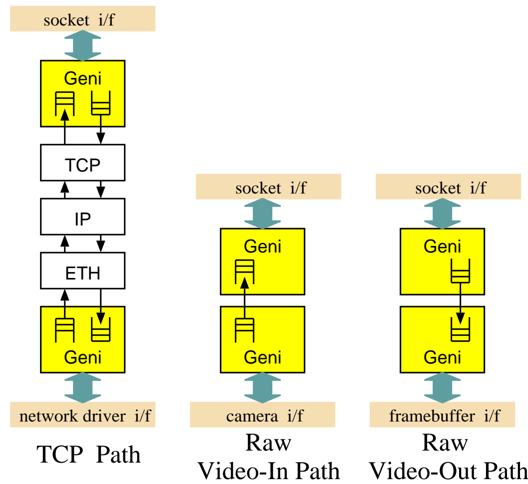

Figure 4: Three Scout paths: a TCP path and two raw video paths, from a camera and onto a framebuffer

Figure 4 portrays three examples of Scout paths. Both ends of a path are delineated by the Geni (Generic Interface) module, Scout's partner driver, which abstracts the device and communication interfaces to the path framework, thus serving as a gateway to every Scout path in a symmetric fashion. Each path consists of a chain of protocol modules that process packets

belonging to the connection, with Geni maintaining input and output queues at each end.

The first path corresponds to a single TCP connection. Packets arrive via the network interface, are classified by Geni into a corresponding path. If the packet belongs to the TCP connection of this path, it is placed in the input queue at the bottom of the path; if the queue is full the packet is dropped. When a path is run, a thread belonging to the path dequeues a piece of data from its input queue, runs the code modules in sequence, and deposits the result in Geni's output queue at the opposite end of the path. In this case, the packets need to undergo Ethernet, IP and TCP processing, respectively. At the top, standard user space applications connect to this path via the socket interface.

The two raw paths depicted in Figure 4 behave in a similar fashion, except that no protocol modules are chained to the path. In one case, packets are generated via a video camera, arrive on the camera interface, and are sent to the application via the socket interface. Geni abstracts the interfaces on each end, and maintains queues if buffering is needed. Likewise, the other raw path portays an application that sends packets via the socket interface to be displayed via the framebuffer interface on video screen.

To simultaneously maintain the benefits of Scout paths, preserve the ability to test and run standard applications on a general purpose and open OS, and support different levels of packet service across any given devices, we designed Silk [2], which encapsulates the Scout path architecture into a loadable Linux module, for Linux versions 2.4 and above. To achieve these goals, Silk provides its own path scheduler and thread package that coexists with the Linux CPU scheduler.

Our scheduling strategy enables Silk to not only decide what share of the CPU to assign to Linux, but to also maintain its own suite of schedulers for scheduling Scout paths, such as fixed priority, Earliest Deadline First (EDF), Weighted Fair Queueing (WFQ), and Best Effort Real Time (BERT). By choosing an appropriate class of schedulers, such as fixed priority, we can implement elaborate quality of service (QoS) processing, (such as providing packet streams with guarantees on the system's resources, for example CPU, bandwidth, memory pools, threads and queues), while still servicing best effort packets at line speeds [11].

The Silk scheduler incorporates a policy decision that determines exactly which flow of packets constitute which path. Certainly each QoS flow is treated as its distinct path, even if the forwarding function is essentially the same as for some other path. On the other hand, multiple best effort flows that share the same forwarding function are classified into the same path.

2.3 Vera

The VERA architecture [8] was designed to hide the forwarding details of a router from its hardware functions. It consists of three abstraction levels, each attempting to strike a balance between being simultaneously rich enough so as not to constrain its corresponding design space, and remain at a level high enough so as to enable modeling and reasoning about the system.

For our purposes, we view the VERA framework as a special driver, *vera.o*, which can connect a general purpose processor, like the Pentium, to a more specialized network processor, such as Intel's IXP 1200 [7] or the RAMiX PMC694 board [12]. Thus, it defines the *vera interface* to interact with Silk from above, and a PCI encapsulation to connect to a processor or device via the PCI bus from below. Figure 5 shows an example configuration where the Pentium (running Silk) is connected to a RAMiX PMC694 board and an Intel IXP 1200 EEB board over the PCI bus by means of the Vera driver. This example system, along with other standard network drivers is mapped to the interface taxonomy of Section 2.1.

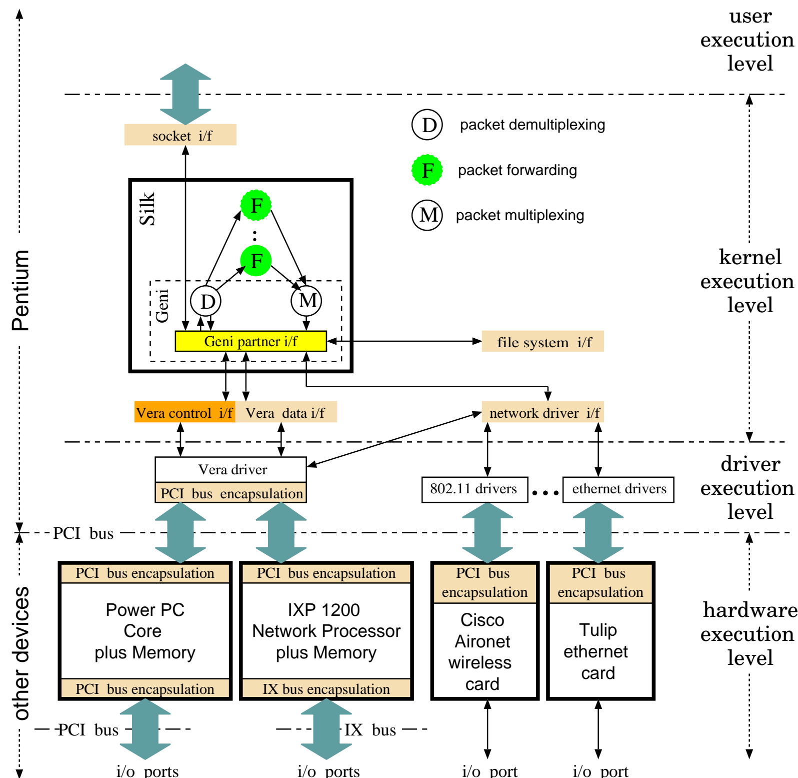

Figure 5: The Vera driver connects a general purpose processor, such as the Pentium, to network processors, through a richer (data and control) interface to the OS kernel. It resides on the *driver* execution level along with simpler and more standard drivers, such as ethernet and wireless 802.11 drivers.

3 Partner Architecture

The partner architecture enables the connection of communication oriented intelligent or simple devices across possibly disjoint *execution islands*, a subclass of execution levels. Such execution islands might be disjoint by a hardware bus, a logical bus, or some other network. The partner architecture enables establishing connections between devices, partial demultiplexing, pooling and interrupt driven transmission methods, propagation of control operations, and cooperative flow scheduling.

Keeping the five goals of Section 1 in mind, the partner architecture was designed to be minimal, extensible, symmetric, easy to program and not harm system performance. The idea is that any legacy or standard hardware specs, drivers or parts of an OS spec, such as a disk drive specification, video driver and the socket interface, respectively, can easily translate their interfaces to the partner interface. This allows communication flows established within a system to uniformly access the networking stack (e.g. via the Geni driver in the case of Scout paths), where data is forwarded or modified. Alternatively, if neither buffering nor additional processing is required, raw data can flow directly between a network interface card and an application via sockets due to the common partner interface.

We stress that although, as we later illustrate, the partner architecture was implemented within Silk, the concept is applicable to any other loadable kernel module, or a system embedded

into a general purpose OS. (Connecting Silk to the Vera driver via the partner interface is a case in point.) Namely, rather than rewrite the drivers for all the interfaces to file systems, devices and various user level APIs, thus effectively redoing all the work done within the general OS for this specific module, we unify all the drivers into the *partner interface*, where they all exchange a common data structure, the *partner vector*, and assume a symmetric *partner modus operandis* after OS initialization. We cover these issues in detail below.

3.1 Partner Vector

In our driver centric world, each intelligent device driver maintains an internal representation for its data in its own private data structure. This internal representation is usually published as part of the driver's API. We refer to these as the *aggregate* internal representations, since along with the actual data buffers, they include metadata to facilitate data manipulation according to the requirements of each particular device driver. Some types of devices and the corresponding aggregate internal representations of their drivers are tabulated in Table 1.

To have the device drivers interoperate, they need to be able to exchange their internal representations with one another. Therefore, as part of our uniform interoperability framework, partner drivers export a *partner vector*, or a *pvec*. The key requirements in designing the partner vector were that it (1) be an aggregate of one or more of aggregagte internal representations of any kind, and (2) that it enables data manipulation among different device drivers without memory copying. The *pvec* pseudo code definition is given in Figure 6.

```
typedef struct pvec {
  partner_t partner;     /* pointer to manager of this pvec        */
  count_t   count;       /* this pvec is comprised of count chunks
                              of other aggregate internal represent.*/
    struct {
      void *    data;    /* ptrs to data within aggregates         */
      size_t    size;    /* sizes of actual memory comprising data */
      partner_t partner; /* partners who own the ir chunks         */
      void *    ir_aggr; /* ptrs to ir aggregate data structures   */
    } pvec_table[MAXCOUNT];
} * pvec;
```

Figure 6: Partner Vector data structure

Figure 7 portrays how the *pvec* data structure includes pointers to data chunks of other aggregate representations as well as the aggregates themselves. Illustrated are a *frame_pool* for video or audio drivers, an *iovec* for socket drivers, an *sk_buff* for the Linux netfilter driver, and a *message* for Geni, the driver for Scout paths. This example obviates how our design allows partner vectors to include any existing or new aggregate data structures used by interfaces, file systems and devices.

3.2 Partner Interface

The partner interface encapsulates interaction with intelligent devices, that generate and consume data. The focal point here is that the partner interface is symmetric. That is, it provides and expects the same functionality to and from its partners, respectively. The five operations of the partner interface are *open*, *close*, *push*, *pull* and *control*. Thus, each device driver must provide its side of the partner interface. This entails an implementation of the relevant subset of the above

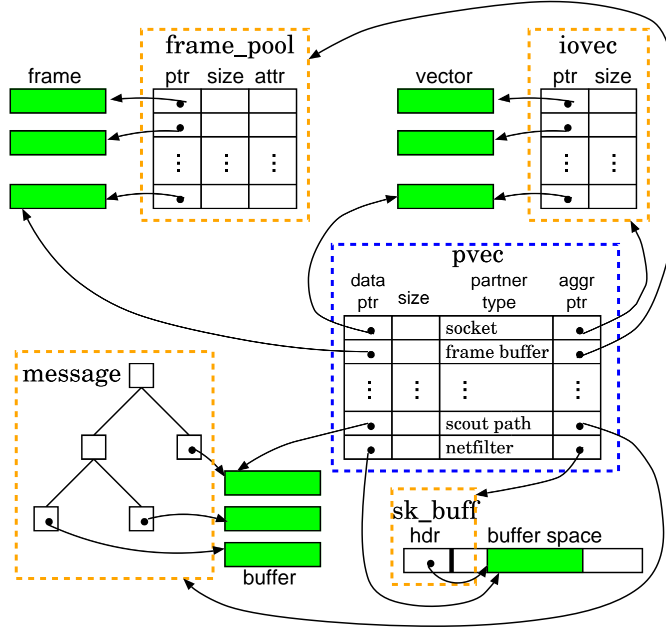

Figure 7: The Partner Vector, *pvec*, is comprised of pointers to chunks of data in memory buffers (shaded boxes), as well as pointers to the aggregate internal representations (in dotted boxes) of the devices who manage these data chunks, such as *sk_buffs*, *frame_pools*, *iovecs* and Scout *messages*.

functions, which, in essence, are translators to the original driver functionality — an exercise, we claim, takes several hours of programming time. We will substantiate this claim in Section 4, by providing the number for the lines of code some sample partner drivers. Pseudo code for the partner API is given in Figure 8.

As the names suggest, *open* opens a partner device by providing it with a configuration template and a set of attributes for a data flow, while *close* closes a data flow on a partner device. Likewise, *push* sends a partner device some data message with its corresponding attributes, intended for a particular flow, indexed by *demux_key*; whereas *pull* asynchronously receives a message with its attributes from a partner device, providing a pointer to a *pvec* for where to access it. The *ctl* operation allows each partner to define its own set of control operations with their corresponding parameters, which could be called on a particular flow stemming from that partner.

The configuration template argument, *flw_templ*, determines the protocol modules that constitute the intended flow. The attributes on the other hand, specify characteristics for processing the flow, such as internet and port addresses, which could be wild–carded, resource reservations for CPU, bandwidth, memory pools, threads or queues, as well as other flags such as realtime constraints, priority, how to handle delays, and possibly synchronization IDs with other flows.

Although we distinguish between flow attributes, *flw_attr*, an argument to *open*, and data attributes, *data_attr*, an argument to *push* and *pull*, they are essentially drawn from the same set of attributes. The difference being that flow attributes enable *out-of-band* style signalling for the flow, such as a separate RTCP flow along with RTP packets, whereas data attributes enable *in-band* flow signalling, such as attaching headers to data packets. Moreover, rather than forcing the partners to choose one signalling style over another, our partner interface design permits mixing the two styles along a spectrum of possibilites.

```
typedef struct partner * partner_t;
typedef struct ops ops_t;
typedef struct buffops buffops_t;

/* partner_t:   partner device on which operation is called
   flw_templ_t: path configurations template e.g. sequence of protocols
   demux_key_t: identifier of data flow
   flw_attr_t:  attributes to a flow
   data_attr_t: attributes to the data (packets) being pushed or pulled
   pvec_t:      partner vector to be pushed into or pulled from partner
   ctlop_t:     control operation number invoked on that partner
   ctlarg_t:    argument list to control operation
   ctlarglen_t: length of control operation argument list
*/
typedef struct ops_t {
 int (* open)  (partner_t, flw_templ_t, flw_attr_t);
 int (* close) (partner_t, demux_key_t);
 int (* push)  (partner_t, demux_key_t, data_attr_t, pvec_t);
 int (* pull)  (partner_t, demux_key_t *, data_attr_t *, pvec_t *);
 int (* ctl)   (partner_t, demux_key_t, ctlop_t, ctlarg_t, ctlarglen_t);
};

typdef struct buffops_t {
  int (* get)   (partner_t, pvec_t);
  int (* alloc) (partner_t, pvec_t);
  int (* free)  (partner_t, pvec_t);
};

struct partner_t {
  int       type;     /* type of this partner:
                               e.g. video, Geni, netfilter, etc.    */
  int       irhandle; /* handle to partner's internal representation
                               e.g. ptr to iovec, skbuff, pvec, etc */
  partner_t partner;  /* pointer to callee partner                  */
  ops_t     ops;      /* function ptr to operations of this partner */
  buffops_t buffops;  /* function ptr to buffer operations
                               of this partner                      */
};
```

Figure 8: Pseudo code for the Partner Interface: the five operations, *open, close, push, pull* and *ctl*, are first class operations of the Partner Interface; the buffer operations, *get, alloc* and *free*, are second class in that they are part of the *partner* data structure, but not explicitly part of the Partner Interface by which devices communicate with each other.

3.3 Partner Modus Operandi

A key challenge in designing the partner interface was the need to account for "symmetry" in device interconnectivity. That is, when two devices are connected, either one, or both may want to be the master. To understand how partner drivers interoperate, we explain how to distinguish between the two sides of the partner interface, where the execution thread executes, how the partners are initialized, in which direction data flows through and the concept of demux propagation.

We classify devices into either *passive*, or *active*. Passive devices do not initiate requests to partner with other devices. Rather, the request is always initiated from the other side, causing the passive device to partner and respond. Conversely, active devices do initiate partnering requests on other devices. Clearly, active devices could also have, and for the most part do have, a passive side to them. Examples of passive devices are microphones, monitors and some network cards, while Scout and sockets are examples of active devices.

Drivers for active devices can be implemented to be passive or active, whereas drivers for passive devices are always passive. Within the partner architecture, we extend this concept one step further, to the partner operations. This means that active partner drivers need to provide two types of operations: *active partner operations* initiated by the driver to be carried out by a partner on the other side, and *passive partner operations* initiated by a partner on the other side to be carried out by this driver. Passive partners on the other hand, need only provide the passive operations. To paraphrase, active partner operations for a particular driver translate the internal functionality of the driver to the partner interface, whereas passive partner operations translate the partner interface to the internal functionality of the driver in question.

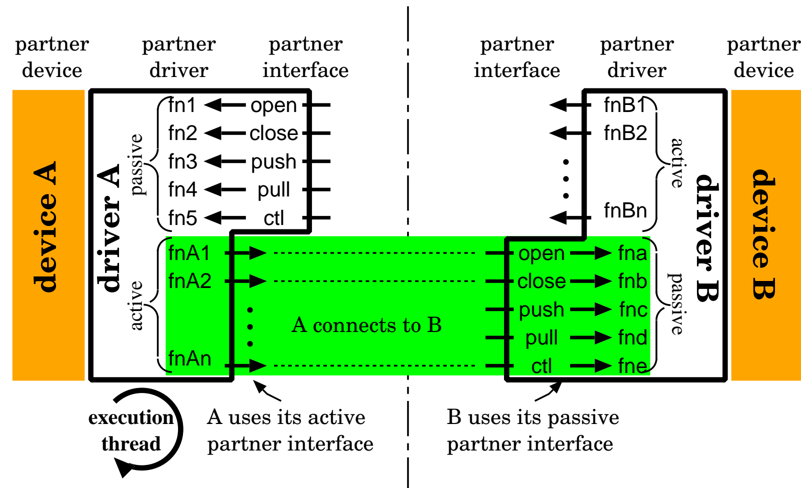

Figure 9: The execution thread is at driver A. Therefore, driver A executes its active partner interface, threreby invoking the passive partner operations of device B.

Between two communicating partner drivers, we refer to the *active* partner as the one on whose side the execution thread executes. This determines which driver invokes its native functions via the active partner interface, thereby invoking the passive partner operations on the other side. Figure 9 shows a partnering scenario of device A connecting to device B via their respective partner interfaces. Since the execution thread is on driver A's side, driver A executes its active

partner interface, thereby invoking the passive partner operations of device B.

The more complex scenario is when the execution thread executes on both sides of the two connected partner drivers. In this case, both devices invoke their active partner interfaces, where they have the control, as well as the passive side of their respective interfaces, where the other side has the control. This is when a partner driver's both passive and active interfaces are utilized simultaneously.

Along another dimension, not all devices and their respective drivers are inherently bidirectional. That is, data can flow *into* a device, *out* of a device, or both *in and out.* Table 1 summarizes a number of devices, their respective drivers, the aggregate internal representation of each such driver, whether the corresponding partner interface is passive or active, and direction of data flow.

Device Name	Driver Name	Aggregate Internal Representation	Partner Interface	Data flow Direction
speakers	audio	*frame_pool*	passive	in
microphone	audio	*frame_pool*	passive	out
monitor	video	*frame_pool*	passive	in
camera	video	*frame_pool*	passive	out
filesystem	filesystem	*iovec*	passive	in/out
network card	netfilter	*sk_buff*	active	in/out
Tulip card	tulip	*sk_buff*	active	in/out
Vera device	Vera	*vvec*	active	in/out
user space	socket	*iovec*	active	in/out
Scout module	Geni	*message*	active	in/out

Table 1: Examples of devices, their respective drivers, the aggregate internal representation of each such driver, whether the corresponding partner interface is passive or active, and direction of data flow.

Partner drivers exchange setup information at OS initalization by calling the partner driver initialization function
partner_t **pdriverInit**(*driver_name_t, attr_set_t, partner_t*);
which binds a driver name to an instance of a particular partner, and initializes the partner interface and data structures between the two driver instances in only one direction, namely, the active partner operations of the calling driver. That is, the active side provides its partner data structure to the passive side as an argument, and gets the passive's side partner as the initialization function's return value. If both drivers are active, and they need to partner both ways, the other driver would also invoke *pdriver_init* on the first. For example, if the socket driver needed to initialize its partnership with netfilter, the function call would be

netfilter0_partner = **pdriverInit**(*netfilter0, attr_set, socket1_partner*);

Conversely, if netfilter needed to initialize its partner interface with sockets, it would make the function call

socket1_partner = **pdriverInit**(*socket1, attr_set, netfilter0_partner*);

Consequently, for a system with n partner drivers, at most $(n^2 - n)/2$ partnerships are ini-

tialized if partner drivers of the same type cannot communicate directly, and $n^2/2$ if they can. Fortunately, this quadratic exchange need only occur once, during system initialization, and does not further impact system performance.

We have outlined how partner drivers can create a string of paths for data flow among different execution levels and execution islands. A notable feature of such a partner architecture is that it can easily propagate demultiplexing as data flows from one device to the next. *Demux propagation* allows the system to increasingly demultiplex, or further specify, the data flow to which a packet belongs as it moves from one device to another. When a partner receives a *pvec* with some demux key, performs demultiplexing, and needs to pass it along to another device driver, it includes a more specific (the result of its demultiplexing) demux key in its respective partner operation call. Thus demultiplexing propagates as the packet moves through our intelligent devices, rather than having this be restricted to the data forwarders.

4 Example Partner Drivers

In this section, we substantiate our claim of versatility and suitability of the partner architecture by describing how a variety of standard and specialized system drivers were translated to export the partner API. By considering the device drivers listed in Table 1, we demonstrate how their functionality is mapped to the partner API. In most cases, by design, this turned out to be a simple and straightforward exercise.

4.1 Audio In/Out Partner Driver

As tabulated in Table 1, the audio partner is a passive driver, driving data out of a camera or into speakers, with *frame_pool*s being its internal aggregates. We henceforth only need to show the mapping of the passive audio partner operations, where speakers only need *push* and a microphone only needs *pull*.

open	$\longrightarrow$	general case: NULL
		special case: return demux on specific channel
close	$\longrightarrow$	free channel state
push	$\longrightarrow$	*write* to speaker device (e.g. /dev/audio)
pull	$\longrightarrow$	*read* from microphone device (e.g. /dev/mic)
ctl	$\longrightarrow$	vary frequency attributes (e.g. volume, treble)

Our audio partner driver was tested on SoundBlaster 16 PCI device using the Ensoniq ES1371 Linux driver, and comprises 120 lines of C code.

4.2 Video In/Out Partner Driver

Likewise, referencing Table 1 suggests that the video partner is a passive partner, driving a monitor (for data in) or a camera (for data out), and using *frame_pool*s as its aggregate internal representation. Therefore, only the passive partner operations exist, with the monitor and camera only requiring *push* or *pull*, respectively.

open	$\longrightarrow$	initialize offset on framebuffer
close	$\longrightarrow$	invalidate offset on framebuffer
push	$\longrightarrow$	*memcpy* to framebuffer pool
pull	$\longrightarrow$	*read* from camera device (e.g. /dev/video)
ctl	$\longrightarrow$	vary attributes (e.g. resolution, color, scale, frequency)

Our partner video driver was implemented and tested for the ATI All Innovator card, on the Range 128 chip, on top of its Linux kernel driver, which only exports the capturing functionality of the device. To interface to the tuner functionality, we needed to access the hardware directly. Consequently, our partner piece is comprised of 610 lines of code, 370 of which initializes the tuner via the PCI bus, and the remaining 240 actually implement the partner operations on top of the Linux driver.

4.3 Filesystem Partner Driver

The next device driver in Table 1 is the filesystem, a passive partner with bidirectional flow of data. This means that both *push* and *pull* partner operations are implemented, and only the passive side exists.

open	$\longrightarrow$	*filp_open*
		filp_close
close	$\longrightarrow$	free channel state
push	$\longrightarrow$	*write* to filesystem
pull	$\longrightarrow$	*read* from filesystem
ctl	$\longrightarrow$	other file operations (e.g. lseek, fsync)

Our current filesystem partner driver handles the Linux filesystem interface, was tested on Ext2 filesystem, and is comprised of 165 lines of C code.

4.4 Network Partner Driver

In this section, we simultaneously discuss the functionality of two partner drivers from Table 1. The first is the tulip partner driver, which drives PCI ethernet network cards based on Tulip chip-sets. The second is netfilter, which drives Linux's netfilter interface (a higher level API), and is ordinarily used to register network packet filters. These filters essentially act as early packet demultiplexers before packets reach the networking stack, in our case, within Silk.

Both partner drivers are active bidirectional partners, using *sk_buff*s as their aggregate internal representations, thereby requiring us to map both, the active and passive sides of the partner interface. From such a perspective, the only functional difference between the tulip and netfilter drivers is that tulip can operate both, as a polling or interrupt–driven driver, whereas netfilter only operates in interrupt–driven mode. The passive partner operations for both drivers are mapped as follows.

open	⟶	NULL
close	⟶	NULL
push	⟶	transmit data to network device
pull (tulip)	⟶	poll on–card queue
pull (netfilter)	⟶	NULL
ctl	⟶	*getname, getaddr, intron, introff*

The active side of the interface, on the other hand, only invokes one partner operation on the other side, namely pushing the data onto its partner. The other four operations are never invoked.

transmit data up the network stack ⟶ *push* data onto partner

We tested the tulip partner driver for the Kingston KNE100TX PCI Ethernet card, based on the 21143 Tulip chip–set, with MIT's modified Linux tulip driver (to provide polling). The additional partner driver has 340 lines of C code. Likewise, the netfilter partner driver was implemented on Linux's netfilter interface and amounts to 370 lines.

4.5 Vera Partner Driver

From our perspective, we view the Vera driver described in Section 2.3 as a highly intelligent network card, which hides the fact that it can pass data to and from other processors, across the PCI bus. In essence, Vera can establish its own paths at lower execution levels, on different execution islands, and part of its functionality is to extend these paths to the upper exection level paths within Silk. As shown in Figure 5, the partner interface is Vera's interface from above. Thus, as stated in Table 1, it vera is an active bidirectional driver, with the passive side of its partner interface described below.

open	⟶	*create* Vera path with path templates and attributes
close	⟶	*delete* Vera path identified by *dmx_key*
push	⟶	*send* data (*pvec*) with its attributes down Vera *dmx_key* path
pull	⟶	*receive* data (*pvec*) from Vera path *dmx_key*, return data (pvec) and data attributes
ctl	⟶	*control* Vera interface: download forwarder, update route cache, update MIB

The active side of Vera's partner interface does not invoke *open* or *close* on the other side. Like ordinary NICs, it would have exchanged information with its partners during initilization. We describe how Vera actively invokes the remaining operations on its partners as follows.

transmit data up the network stack	⟶	*push* data onto partner
ready for new data from network stack	⟶	*pull* data from partner queue
event notification	⟶	*ctl* operation on partner
(e.g. intrusion detection, partner state modification)		

The Vera driver is currently being programmed to drive Intel's IXP 1200 and the RAMiX PMC694 boards, with more network processors to be added. We estimate to have around 500 lines of code pertinent to the active and passive sides of the partner interface.

4.6 Socket Partner Driver

The standard UNIX socket interface bridges user space applications with kernel networking stacks. In our prototype, as explained earlier, Silk hijacks the kernel networking stack from Linux. However, our socket partner interface permits Linux applications that use standard networking protocols via the socket API, such as calls on the *PF_INET* family, to be intercepted and processed by Silk. This allows unmodified legacy applications to access TCP and UDP paths transparently. For applications using experimental or non-standard protocols, or those requiring additional functionality, Silk provides a new, *PF_SCOUT* protocol family, which is registered with Linux at system initialization, at which time it is also initialized to partner with Geni, in both directions. For extremely raw paths involving the socket driver but bypassing Geni, the socket driver would also be setup to partner with the other relevant drivers at OS initialization.

The socket driver is an active and bidirectional partner, and besides Geni, is the most involved and interesting partner driver we coded, due to a number of semantic peculiarities from the active and passive sides of the partner interface.

One peculiarity is that sockets is what we call a *superactive* or *nonpassive* partner driver, meaning that other partners cannot exchange any data with the socket partner unless the thread is on the socket side. For example, sockets do not maintain queues in which to store data sent/received to/from other partners. Therefore, all the passive partner calls do is modify state. This then causes the socket driver to initiate *push* or *pull* on its active interface when its pertinent thread is activated. This passive side of the sockets partner interface is summarized below.

open	⟶	add *dmx_key* to accept queue; wake up blocked process on socket (e.g. accept or select)
close	⟶	set `CLOSED` flag
push	⟶	wake up blocked process on the socket (e.g. recv or select)
pull	⟶	NULL
ctl	⟶	set socket attributes/flags on the socket data structure

From the active side on the other hand, note that the kernel socket driver implements the socket API, which exports a wide range of function calls to user space, with rich semantics, suitable for protocols such as TCP. We therefore start our active partner interface mapping from user space socket functions, the ones the reader is most likely to be familiar with. The partner interface translates the Linux Socket Driver (LSD) into either partner operations on the other side, or tiggers actions back on the LSD itself.

User Space Function		Linux Kernel Function		Socket Partner Driver Actions or Operation
socket	⟶	*create*	⟶	setup socket data str., ops, state, return to LSD
bind	⟶	*bind*	⟶	*open* on partner with local address
connect	⟶	*connect*	⟶	if socket bound then *ctl* to specialize path with remote address; else *open* on partner
listen	⟶	*listen*	⟶	set `ACCCEPT_QUE_LEN`; *ctl* to enable `LISTEN`ing path
accept	⟶	*accept*	⟶	if accept queue nonempty, then create new socket structure from *dmx_key*; return to LSD; else sleep with timeout in LSD (accept blocks)
close	⟶	*release*	⟶	*close* partner of corresponding socket
shutdown	⟶	*shutdown*	⟶	*ctl* `HALF_CLOSE` if half close, else *close*
send/sendto/ sendmsg/write	⟶	*sendmsg*	⟶	extract demux key from socket structure; *push* the corresponding *pvec*
recv/recvfrom/ recvmsg/read	⟶	*recvmsg*	⟶	if data available *pull* on partner else if no data & nonblock. socket, return `NODATA` else do while not timed out {*suspend sspnd_t*; *pull*}
poll	⟶	*poll*	⟶	if `READSET` on, *pull* partner w/ NULL pvec; if `WRITESET` on, *push* partner w/ NULL pvec; if `EXCEPTSET` on, check signals and accept queue if true, set exception flag on; return to LSD
select	⟶	*poll* loop	⟶	same as above per LSD invoked *poll*
getname	⟶	*getname*	⟶	return socket address to LSD
setsockopt	⟶	*setsockopt*	⟶	*ctl* `SETSOCKOPT`
getsockopt	⟶	*getsockopt*	⟶	if option available in socket str., return it to LSD; else *ctl* `GETSOCKOPT`
ioctl	⟶	*ioctl*	⟶	*ctl* corresponding operation on partner
socketpair	⟶	*socketpair*	⟶	*create* 2 sockets (in LSD) with same var dmx_key

It is noteworthy that a *connect* socket is created via the active *open* partner operation invoked on behalf of *bind* or *connect* (the difference is that *bind* and *connect* specify the local and remote socket addresses, respectively). Conversely, to create an *accept* socket, the other partner first invokes *open* on socket's passive interface, which is translated into adding a demux key to the accept queue, and waking up the blocked process on this socket, if any. This causes the active side to trigger accept, which translates into creating a new socket structure.

Another interesting scenario is how we implement poll and select. Depending on what is being polled, the partner interface will invoke an active *pull* for `READSET` on, with the option that checks if data is available to pull. Similarly, if `WRITESET` is on, it invokes a *push* with the option that checks if the path can accept data. Lastly, for `EXCEPTSET` on, it checks the signals in the socket structure as well as if the accept queue is non empty, and returns the result to LSD.

The socket partner currently comprises 1200 lines of C code, and we envisage it expanding to about 1700 as we add more functionality.

5 Geni: Scout Partner Driver

The Geni driver encapsulates the partner interface for Silk, our Linux-embedded OS, which in turn encapsulates Scout paths. More concretely, Scout paths are viewed as a device, and Geni, the driver to that device. Since Scout paths are central within our embedded prototype, and significantly more intellient than other devices within the system, Geni warrants an entire section to explain its implementation of the partner interface in detail.

5.1 Geni Functionality

Being an active partner driver, Geni is conceptually architected to perform three major functions.

1. Implement the passive and active partner operations, thereby translating partner operations to operations on Scout paths and vice versa, respectively.
2. Demultiplex the incoming packets, thus mapping network flows onto Scout paths, and abstracting their forwarding functionality.
3. Multiplex packets from the end of a particular path onto their target output devices, such as a network card or a socket.

A schematic picture of Geni within Silk is given in Figure 10, illustrating how Geni interacts with the other drivers via the partner interace. Portrayed are the partner drivers of each interface from Geni's side and the other device's side, packet multiplexing and demultiplexing. The examples shown are network devices, such as the Vera interface or the Linux network driver interface, and file systems, including the *proc* filesystem, for which connections to these interfaces are all bidirectional. Geni also interfaces to other system devices, such as a one-directional connection from Geni to the sound card and the video frame buffer, and into Geni from the device in the case of the microphone or camera. Additionally, much like the Vera driver provides a bridge between the IXP and the Pentium levels across the PCI bus, Geni bridges the Linux kernel and user space execution levels by providing an interface to sockets and ioctls.

Device partner interfaces are depicted as the small rectangles in Figure 10. Each such dark rectangle conceptually represents an instantiation of Geni partnered with a particular device driver. To illustrate this point, consider again the Scout paths depicted in Figure 4. For the TCP path, the bottom and top Geni modules are instantiations of Geni partnering with the network and socket partners, respectively. In the case of raw paths, the bottom and top modules are instantiations of Geni's partnering with the video and socket partners, respectively.

5.2 Geni Passive Partner Interface

Pseudo code for Geni's passive partner interface, translating the five partner operations into Scout's functionality, are outlined below. [1]

open (*geni, flw_templ, flw_attr*){
    dmx_key = pathCreate(*flw_templ, flw_attr*);
}

close (*geni, dmx_key*){
    pathDelete(*dmx_key*);
}

[1] A detailed explanation of the functions on Scout paths can be found in [10].

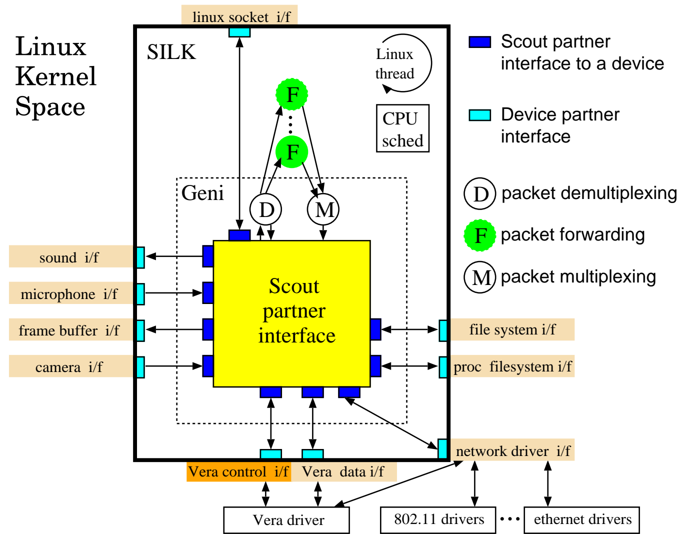

Figure 10: Scout's partner interface, Geni, connects Scout paths to network and media devices, file systems and socket interfaces

```
push (geni, dmx_key, attr, pvec){
      path = demux(dmx_key, pvec, attr);
      if  ( pvec == NULL )
        \* case 1: does path have space to accept more data? *\
        return canPathAcceptData(path);
      elseif  ( path ≠ NULL ) {
        \* case 2: raw path, use pvecs directly *\
        deliver(path, pvec, attr);
      }
      else {
        \* case 3: Scout path, convert pvec to msg  *\
        msg = pvecToMsg(pvec);
        if  ( path = demux(dmx_key, msg, attr) ) ;
          deliver(path, msg, attr);
        else
          drop(msg);
  }
pull (geni, * dmx_key, * attr, * pvec){
      if  ( dmx_key == NULL )
        \* default path leads to default input queue *\
        path = geni→default_path;
      else\* specific path is given to be extracted *\
        path = demux(* dmx_key, * attr);
      if  ( pvec == NULL )
        \* case 1: is data available to pull? *\
        return isDataAvailable(path);
      elseif  ( * pvec == NULL )
        \* case 2: create your own pvec *\
        if ( * pvec = dequeue(path) ) {
```

```
          * attr = computeAttr(pvec);
          return DATA;
        }
        else
          return NODATA;
      elseif   ( * pvec ≠ NULL ∧ aggrSize(pvec) == 0 )
        \* case 3: fill out my pvec w/ your memory *\
        if ( * pvec_q = dequeue(path) ) {
          fillout(pvec, pvec_q);
          * attr = computeAttr(pvec);
          free(pvec_q);
          return DATA;
        }
        else
          return NODATA;
      else   ( * pvec ≠ NULL ∧ aggrSize(pvec) ≠ 0 )
        \* case 4: fill out my pvec w/ my memory *\
        while (unusedMem(pvec) ∧ pvec_q = dequeue(path)) {
          * attr = computeAttr(pvec_q);
          copyInto(pvec, pvec_q);
          if ( notAllCopied(pvec_q) );
            requeue(path,pvec_q));
          else
            free(pvec_q);
        }
        return DATA;

  }
ctl (geni, dmx_key, ctlop, ctlarg, ctlarglen){
      if ( path = demux(dmx_key) )
          \* Ctl op. is on a Scout path *\
          return ctlPath(path, ctlop, ctlarg, ctlarglen);
      elseif ( module = mapModule(ctlop) )
          \* Ctl op. is on a Scout module *\
          return ctlModule(module, ctlop, ctlarg, ctlarglen);
      else
          \* Ctl op. is on the Scout device *\
          return ctlScout(ctlop, ctlarg, ctlarglen);
  }
```

The first partner interface operation, *open*, invokes Silk's *pathCreate*, which creates a Scout path with the given flow template, and identifies the demultiplexing key to the Silk scheduler, which along with the flow attributes schedules the path. The *close* operation deletes the Scout path and frees all its associated resources.

When the scout driver gets a *push* operation, it is handled by two cases: the first demultiplexes the given demux key, pvec and attributes onto a raw path and delivers the pvec onto that path directly; whereas the second case converts the pvec to a Scout message, performs the demultiplexing to obtain the path, and finally delivers the message to that path. If deliver does not succeed, the pvec/message is dropped.

Upon a *pull* operation, if no demultiplexing key is provided, the default path is assumed, which results in a *pvec* being pulled from the default input queue associated with this Geni partner. Otherwise, the scout driver first demultiplexes the demux key and possibly some attributes onto a path, and then splits processing into four cases. In the first case, scout interprets the other side as inquiring if data is available, and responds accordingly. In the second case, the requesting partner indicates that it has no memory allocated for the pvec to be pulled, which means that the scout driver needs to create the aggregate pvec as well as allocate and fill the memory for the message. In the third case, the other side has an aggregage pvec, but is expecting the scout driver to allocate memory for the data (or the Scout message). Finally, in the fourth case, the calling partner has allocated both, the aggregate pvec as well as the memory for the data packets, in which case the scout driver keeps dequeueing pvecs from the path as long as there still is memory available. If the last dequeued pvec could not be fully copied onto the other side's memory, it is re–queued back on the path.

For the latter three cases, the scout driver also computes a set of attributes that it passes to the requesting partner along with the pvec.

In the context of the scout device pseudo code for the partner interface operations, just like demultiplexing is implemented via the Silk *demux* function, the functionality of multiplexing packets onto the queues of output devices is implemented via the *dequeue* function. In reality, two threads are maintained by the scout driver for these purposes: an input process thread, which demultiplexes the packets onto the input path queues, and an output process thread, which multiplexes the packets onto the output path queues. The Silk scheduler schedules these threads according to its scheduling discipline, as well as path and data attributes that get passed via the partner interface opertions.

With such an architecture, we are able to synchronize two or more paths through their data attributes. This would be accomplished by the multiplexing (output) thread that controls dequeueing the pvec off the output queues of the relevant paths. For example, we would be able to synchronize a video-out (to a framebuffer) path and an audio-out (to a microphone) path to display/hear them simultaneously in lock–step.

Finally, the *ctl* operation performed on a scout driver results in three disctint cases. If demultiplexing the demux key results in a valid path ID, a control operation is invoked on that path. In the second case, if the control operation range maps to a known module, then a control operation on a Scout protocol module is invoked. Otherwise, the control operation is invoked on the Scout device itself.

5.3 Geni Active Partner Interface

Before outlining the pseudo code for Geni's active partner interface, we define a Geni *stage*, an instance of a module in a particular Scout path. The corresponding data structure consists of a particular Geni partner, a demultiplexing key, a particular path, as well as the default input and output queues — the psedo code follows.

```
typedef struct geni_stage_t {
  partner_t partner;            /* an instance of the Geni driver */
  demux_key_t demux_key;        /* demux key for the Geni instance */
  path_t path;                  /* a path within the Geni instance */
  output_queue_t output_queue;  /* default output queue for Geni stage */
  input_queue_t input_queue;    /* default input queue for Geni stage */
} * geni_stage;
```

Pseudo code for Geni's active partner interface, translating Scout's functionality to partner operations on the other side, consists of Scout functions that invoke partner operations, namely *geniStageCreate, geniStageDelete, geniTransmit, inputProcess, outputProcess*, and *geniCtlOp*, which are outlined below.

```
geniStageCreate (geni_stage, flw_templ, flw_attr){
        allocAndInitState(geni_stage, flw_templ, flw_attr);
        geni_stage→demux_key = open(geni_stage→partner, flw_templ, flw_attr); }
  }
geniStageDelete (geni_stage){
        deallocAndTeardownState(geni_stage→partner, geni_stage→demux_key);
        close(geni_stage→partner, geni_stage→demux_key); }
  }
geniTransmit (geni_stage, pvec){
      if   ( geni_stage has output process)
        enqueue(pvec, geni_stage→output_queue);
      else
        push(geni_stage→partner, geni_stage→demux_key, attr, pvec);
  }
outputProcess (){
      geni_stage = findMostEligibleStage();
      if   ( pvec = dequeue(geni_stage) )
        push(geni_stage→partner, geni_stage→demux_key, attr, pvec);
  }
inputProcess (){
      geni_stage = findMostEligibleStage();
      if   ( pull(geni_stage→partner, &geni_stage→demux_key, &attr, &pvec) )
        ScoutPush(geni_stage→partner, geni_stage→demux_key, attr, pvec);
  }
geniStageCtlOp (geni_stage, ctlop, ctlarg, ctlarglen){
      internalCtlOp(geni_stage, ctlop, ctlarg, ctlarglen);
      if ( partnerCtlRequired(geni_stage, ctlop, ctlarg, ctlarglen) )
          ctl(geni_stage→partner, geni_stage→demux_key, ctlop, ctlarg, ctlarglen);
  }
```

Scout's *pathCreate* and *pathDelete* functions create and delete, respectively, stages in the path according to the ordering prescribed by the flow templates and attributes. The corresponding creation and deletion functions for the Geni stage, being an end stage for every path, would necessarily actively invoke the *open* and *close* operations, respectively, on the connecting partner.

Actively invoking the *push* operation is more complex. To transmit packets via interrupts, if the relevant Geni stage has an output process running, the partner vector is then enqueued in its output queue; otherwise *push* is invoked with the stage's demultiplexing key. Alternatively, the output process polls the output queue. It determines the currently most eligible Geni stage (via a scheduling policy), dequeues a partner vector from it, and invokes *push* with the stage's demuliplexing key onto the partner.

The *pull* operation is only invoked in the polling mode via an input process, which operates on the input queue in a fashion analogous to the output process. The partner vector obtained by the

pull operation on the most eligible Geni stage is then pushed into Geni's internal data structures via a *ScoutPush* function, without going through the partner interface.

A control operation on a Geni stage potentially consists of two components: an internal control operation on Geni itself, and if necessary, a *ctl* operation on the corresponding partner.

6 Evaluation

Every increase in functionality comes at some cost in performance. The question is, what is this cost? Clearly, maximal performance can be squeezed out of a system by carefully optimizing every specialized case. Therefore, the benefits of generalization should be evaluated as a tradeoff between the added functionality and the performance overhead that it incurs. We thus evaluate the partner architecture along two dimensions, the scope of its extensibility and its performance overhead.

Extensibility can be further subdivided into two categories: expressing an existing legacy device driver in terms of the partner interface, or programming a new device driver from scratch. We have demonstrated the variety of the extensibility of the former kind by implementing and showing the conversions of an audio driver (for microphone and speaker devices), a video driver (for monitor and camera devices), a file system driver, two distinct network interface drivers, and sockets, in Section 4.

Lines of C code	Audio Ensoniq ES1371	Video ATI All Innovator	Network MIT mod. 21143 Tulip
original driver	6278	4083	4970[2]
partner interface code	120	240	340
relative overhead (%)	1.9%	5.6%	6.4%

Table 2: Approximate estimate of the relative overhead of the partner interface compared with the size of the original legacy device driver, quantified in terms of lines of C code.

Table 2 estimates the overhead of this conversion from the existing interface to the partner interface for three select drivers, in terms of relative lines of C code. The percentage overhead is computed as the partner lines of code, divided by the sum of the partner and original driver code. Despite that the estimates were always computed conservatively, the partner overhead for the audio, video and Tulip drivers from our prototype come out to be 1.9%, 5.6%, 6.4%, respectively. Note that even though we regard the filesystem and sockets as just other types of devices, this is not the view taken by our prototype in the Linux operating system. As a result, the filesystem and sockets code is scattered all over the kernel, making it virtually impossible to isolate it into a clean line count, as we were able to do with the more conventionally regarded drivers listed in Table 2.

We demonstrate the extensibility of the latter category of programming new device drivers from scratch in Sections 2.3 and 5, which cover Vera and Geni, respectively. It is therefore natural that they were the most cumbersome and took the longest to write. On the other hand, since they were designed to export the active and passive sides of the partner interface directly, no data structure or function call conversion was necessary, which eliminates the performance overhead.

To quantify the performance of the partner architecture along the second dimension, the incurred overhead, we conduct a set of experiments to measure it. All experiments used a `gcc version 2.96 20000731` compiler on a 1.5GHz Pentium-4 machine, with a 256K cache, and

512MB RAM. Each experiment reports the average obtained from five distinct runs of executing a code fragment 100,000 times.

We regard the incurred partner overhead as an extra indirect function call, such as *push* or *pull* on the respective partner, as well as a data structure conversion from the partner vector to the corresponding internal representation, or vice versa. In our first experiment, we measured the cost of such an indirect function call to be 12 clock cycles, translating to 8 nano seconds on our hardware—a minuscule amount compared to the cost of a memory copy. Subsequent experiments quantify the data structure conversions of *iovec*s, *skbuff*s and Scout *message*s to and from *pvec*s.

To *pvec* **Conversion** from other internal repr.	register operations	memory operations	arithmetic operations	clock cycle count	execution time (nanosec)
iovec (1/2/3/4 vecs) [no memory copies]	7/8/9/10	17/24/31/38	9/11/13/15	26/18/22/26	17/12/15/17
skbuff (1 vector) [no memory copies]	0	13	2	26	17.3
message (1/2/3/4 vecs) [no memory copies]	6/11/16/21	22/33/44/55	13/22/31/40	24/20/36/40	16/13/24/27
message (1/2/3/4 vecs) [w/ memory copies]	N/A	N/A	N/A	290/445/626/769	193/297/417/513

Table 3: Number of register, memory and arithmetic operations, clock cycle counts, and execution time of actually running the code fragment that converts *skbuff*s, *iovec*s or *message*s holding 1200 bytes of data to the corresponding partner vector *pvec*. We report numbers corresponding to each representation splitting the 1200 bytes into one 1200-byte vector, two 600-byte vectors, three 400-byte vectors, and four 300-byte vectors. 'N/A' refers to the fact that it is beyond the scope of the partner architecture, since memory copies due to data conversions is exactly what the partner architecture is designed to avoid.

Table 3 lists the overhead associated with converting the partner vector to other aggregate data structures holding 1200 bytes of data. For each such a target data structure, we run experiments with their aggregates holding the 1200 bytes as one chunk, or split into two, three of four vectors. The purpose of these four data points is to demonstrate the monotonic increase of overhead, as shown in Table 3. We note however, two exceptions for *iovec*s and *message*s, where the clock cycle counts drop from 26 and 24 for one vector, to 18 and 20 for two vectors, respectively. Subsequent counts show the expected monotonic increase. We suspect this phenomenon is due to a branch prediction miss in the Pentium hardware, since we ran the relevant code fragments in isolation from the entire program. By definition, an *skbuff* source data structure can only contain one contiguous data vector, and it is noteworthy that its clock cycle count is equivalent to that of an *iovec* with four 300-byte vectors, namely 26 cycles resulting in 17 nano seconds. A minor but interesting point we note from these numbers, is that the aggregate data structures of an *iovec* is slightly more efficient than that of an *skbuff* in their current implementations, a fact revealed by performing a common operation on them — converting each to the corresponding *pvec*.

To place these cycle counts and execution times into perspective, we measure the overhead of converting a *pvec* into a *message* using message libraries, not taking advantage of the *pvec* aggregate and internal pointer structure (code taken from an older implementation), thus incurring a memory copy. The result is a twenty-fold increase in the incurred overhead.

The conversion in the opposite direction, from the partner vector to the same above data structures, is listed in Table 4. We note that without memory copies, the clock cycle counts and execution times are even smaller than in Table 3. One small difference is that all the numbers are

From *pvec* **Conversion** to other internal repr.	register operations	memory operations	arithmetic operations	clock cycle count	execution time (nanosec)
iovec (1/2/3/4 vecs) [no memory copies]	4/5/6/7	9/15/21/27	5/7/9/11	16/19/24/28	11/13/16/19
skbuff (1/2/3/4 vecs) [no memory copies]	4/7/10/13	14/23/32/41	7/11/15/19	4/4/8/8	2.7/2.7/5.3/5.3
skbuff (1/2/3/4 vecs) [w/ memory copies]	N/A	N/A	N/A	452/636/708/1054	301/424/472/703
message (1/2/3/4 vecs) [no memory copies]	3/4/5/6	7/12/17/22	7/12/17/22	4/4/8/8	2.7/2.7/5.3/5.3
message (1/2/3/4 vecs) [w/ memory copies]	N/A	N/A	N/A	1347/1687/1812/2128	898/1125/1208/1419

Table 4: Number of register, memory and arithmetic operations, clock cycle counts, and execution time of actually running the code fragment that converts the partner vector *pvec* to the corresponding *skbuff*, *iovec* or *message* holding 1200 bytes of data. We report numbers corresponding to each representation splitting the 1200 bytes into one 1200-byte vector, two 600-byte vectors, three 400-byte vectors, and four 300-byte vectors. 'N/A' refers to the fact that it is beyond the scope of the partner architecture, since memory copies due to data conversions is exactly what the partner architecture is designed to avoid.

indeed monotonically increasing with more vectors per aggregate, with no anomalies observed. Another notable difference between the tables is that in this case, we can construct *skbuff*s from any number of vectors, so we can also report numbers for the four cases of the split data sizes. Moreover, the overhead of the conversion with memory copy is more than a hundred-fold increase over the no-copy case for *skbuff*s and approximately three-hundred-fold in the case of *message*s.

An important observation is that all our reported numbers are conservative, precisely because we isolate the relevant code, dissect its instruction counts, and execute it in isolation. As a result, a number of compiler and instruction level parallelism optimizations are not accounted for. To demonstrate this point, we timed the execution of some of these code fragments separately, as reported in Tables 3 and 4, and in the context of other code wrapped around it, and measured the different pieces. In the latter case, the measured overhead for no-memory-copy conversions was almost always zero or close to zero, rather than the numbers reported in Tables 3 and 4.

7 Related Work

To the best of our knowledge, few efforts exist in the operating systems arena with the goals, functionality and characteristics similar to the Partner architecture. Our Partner architecture is designed for intelligent devices, which addresses a data–flow–oriented communication model by implication. Previous work tended to be buffer–oriented, that is, designed towards homogeneous buffering. Other features of the partner architecture mostly absent from other systems are its capability to address heterogeneous devices with varying degrees of intelligence, assume a symmetric mode of communication with the possibility of threads running at either or both sides, and allow for almost infinite extensibility, which for example, would extend to communicating with another processor at the other end of the interface.

The notion of several CPUs has traditionally meant that they are either connected via a network, which would imply network copies in setting up paths; or the CPUs are within a shared memory architecture machine, or a symmetric multiprocessor, in which case all processors are assumed to be homogeneous, with identical instruction sets and programming paradigms. Due

to the heterogeneity feature of the partner interface, our architecture is not restricted by these constraints.

To illustrate these points, we propose to compare the Partner architecture to other systems along a number of dimensions: (1) intended execution level for the interface; (2) size of the interface in terms of number of defined operations; (3) symmetry of operation; (4) functionality; (5) ease of translation to legacy drivers; and (6) performance.

Perhaps the most prominent work related to the Partner architecture is the Uniform I/O (UIO) [4] — a much earlier project providing a uniform I/O interface for distributed systems. UIO specifically addressed the key deficiencies of the UNIX I/O [13], and provided abstractions suited for the remote procedure call (RPC) of a distributed environment, message passing, as well as support for locking, replication and atomic transactions. It was also implemented and used on System V for several years.

Although the UIO set out to be a more encompassing and more ambitious project as compared to the Partner architecture, the functionality provided by both interfaces is very similar. For example atomic transactions, locking can be incorporated into the partner operations via attributes, while RPCs can use the partner interface from user or kernel space in an indirect fashion. Apart from the shared functionality, however, we highlight important differences along the other comparative dimensions. While the UIO is explicitly intended to be implemented above the hardware devices, or "I/O services", the partner interface can be deployed on any execution level as depicted in Figure 2 of Section 2.1. Our partner interface is truly minimal consisting of five simple operations, as opposed to the fifteen major functions of the UIO. We assume a symmetric mode of operation on both sides of the partner interface, whereas the UIO is' designed around a more involved client–server model. The UIO sets up a "generic" data structure to represent its state and a sequence of data blocks, primarily focusing on encompassing the majority of I/O services. The partner vector on the other hand, is explicitly designed to incorporate foreign data structures, in their aggregate and data chunk forms, without incurring any data copies. This aspect highlights the comparative advantage of the partner architecture with respect to ease of conversion to existing device drivers, as well as performance overhead.

A more recent project, NetGraph [5], has been developed in FreeBSD. NetGraph provides a uniform and modular system for the implementation of kernel objects which perform various networking functions. The objects, known as *nodes*, can be arranged into arbitrarily complicated graphs. Nodes have *hooks* which are used to connect two nodes together, forming the edges in the graph. Nodes can thus communicate along the edges to process data and implement protocols. As such, NetGraph alleges a much wider functionality than the Partner architecture, perhaps closer to the functionality provided by the Silk kernel module in Linux. This necessarily leads to a more heavy weight design, which albeit symmetric in nature and non-restrictive to a certain execution level, has eight major methods and data structures of numerous types. Only a subset of the system deals with actual devices, since most of the emphasis is placed on augmenting existing networking protocols or constructing new ones, to be accessed in a uniform fashion. Moreover, due to this mismatch in functionality, it is difficult to compare the relative performance of the two systems — which is further exacerbated by the fact that the architectures were implemented in Linux and FreeBSD, respectively.

An interesting alternative system, NetFS [15], is currently under development at ISI. NetFS is a file system interface to the networking components of an operating system. The idea being that similar to process and kernel file systems, the network file system would represent the system's

various networking components in a file structure. This approach would enable finer grained access control by providing exclusive configuration access to different users, and a uniform API — the NetFS. Other traditional OS APIs such as, the Socket API, Sockopt, IOCTL, SysCTL, and the In-Band API would all be mapped into the NetFS by processes that would translate file operations into network system calls and kernel structure manipulations. Moreover, process-specific views can be provided by a combination of file system access controls and relative file names. These process-specific views, combined with symbolic (or hard) file system links to aggregate subsets of resources in distinct directories would provide support for virtual networking.

At the current phase of the NetFS project, there is a lot in common with the Partner architecture, although NetFS seems to be more ambitious in its intended functionality. The apparent difference being that our data structure, the partner vector, was designed to maximally subsume the data structures of other, new and established APIs, so as to trivialize the conversions from the data structure of one API to another, whereas the NetFS data structure closely resembles the file system only, which might incur a significant performance penalty in its conversion to the internal data structures of other APIs. It is yet premature to conduct a comparison along the dimensions of symmetry, API size, and performance penalty incurred by the overhead of the NetFS abstraction.

8 Conclusions

This work was motivated by the authors' experience dealing with device drivers when building the x-kernel [6] and Scout [10] operating systems. Connecting our OS modules to devices requires specialized code from the OS to interface with that device driver. This entails an inefficient and tedious repetition of the glue programming code in each case. As the devices become more intelligent and it becomes possible and interesting to connect them directly to each other, the problem is further magnified. To this end, we set out to connect these devices in a uniform, quick and seamless manner via a symmetric interface in support of device-to-device data flows. The *partner* architecture was the result of this effort.

A case in point is the design and implementation of Silk [1], embedding Scout paths in the Linux kernel. After designing the partner architecture, connecting Scout paths to new and legacy Linux devices (as well as user space and external processors) was greatly simplified. The main task entailed writing Geni, a partner driver for Scout. Thereafter, all partner device drivers became connectible within Silk.

In the "device–centric" view of our architecture, peripheral devices, user space, kernel modules as well as external processors, can all easily translate their inherent interfaces to the partner interface, resulting in communication flows with a uniform interface, sharing a uniform partner data structure, and uniformly accessing the networking stack. We give concrete examples of partner drivers in our prototype, such as video, audio, filesystem, network, and sockets, as well as more complex drivers for an embedded operating system module and another external processor accessible via a PCI bus. We demonstrate a prototype implemented in Silk where the overhead for this extended functionality is minimal—effectively an indirect function call. We also argue that the partner architecture favorably compares to other related efforts reported in the literature.

References

[1] A. Bavier, L. Peterson, and D. Mosberger. BERT: A Scheduler for Best-Effort and Realtime Paths. Technical Report TR–602–99, Department of Computer Science, Princeton University, March 1999.

[2] A. Bavier, T. Voigt, M. Wawrzoniak, and L. Peterson. SILK: Scout Paths in the Linux Kernel. Technical Report TR–2002–009,, Department of Information Technology, February 2002.

[3] B. W. Boehm. *Software Engineering Economics.* Prenctice Hall PTR, October 1981.

[4] D. Cheriton. UIO: A uniform i/o system interface for distributed systems. *ACM Transactions on Computer Systems*, 5(1):12–46, February 1987.

[5] J. Elischer and A. Cobbs. The netgraph networking system. Available as `http://www.elischer.org/netgraph`, January 1999.

[6] N. C. Hutchinson and L. L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[7] Intel Corporation. *IXP1200 Network Processor Datasheet*, September 2000.

[8] S. Karlin and L. Peterson. VERA: An Extensible Router Architecture. *Computer Networks*, 38(3):277–293, 2002.

[9] F. Kuhns, J. DeHart, A. Kantawala, R. Keller, J. Lockwood, P. Pappu, J. Pawatikar, E. Spitznagel, D. Richards, D. Taylor, J. Turner, and K. Wong. Design of a High Performance Dynamically Extensible Router. In *Proceedings of the DARPA Active Networks Conference and Exposition (DANCE)*, San Francisco, CA, May 2002.

[10] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the Second USENIX Symposium on* , pages 153–167, Seattle, WA USA, October 1996.

[11] X. Qie, A. Bavier, L. Peterson, and S. C. Karlin. Scheduling Computations on a Software-Based Router. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, pages 13–24, June 2001.

[12] RAMiX Incorporated, Ventura, California. *PMC/CompactPCI Ethernet Controllers Product Family Data Sheet*, 1999.

[13] D. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[14] N. Shalaby, L. Peterson, A. Bavier, Y. Gottlieb, S. Karlin, A. Nakao, X. Qie, T. Spalink, and M. Wawrzoniak. Extensible Routers for Active Networks. In *Proceedings of the DARPA Active Networks Conference and Exposition (DANCE)*, pages 92–116, San Francisco, CA, May 2002.

[15] J. Touch and J. Train. NetFS: Networking through the file system. Available as `http://www.isi.edu/netfs`, May 2002.