Bobnet: high-performance message passing for commodity networking components

Chris Csanady
Scalable Computing Lab, Ames Laboratory
Wilhelm Hall, Ames, IA, 50011, USA
Phone 515 294 4286 FAX 515 294 4491
ccsanady@scl.ameslab.gov

Pete Wyckoff*
Sandia National Laboratories, MS 9011
P.O. Box 969, Livermore, CA 94551, USA
Phone 925 294 3503 FAX 925 294 1225
wyckoff@ca.sandia.gov

Abstract

Bobnet is a low-latency message-passing protocol designed for use as the high performance networking software in a loosely coupled cluster of computers. It features zero-copy sends directly from user-space data, and one-copy receives. Bobnet was especially designed to operate as efficiently on commodity networking components (*e.g.,* generic 10 or 100 Mb/s ethernet) as it does on faster, more expensive network interface devices (*e.g.,* Myrinet). Support for multiple higher-layer protocols is provided, including MPLite, TCP, and the emerging VIA standard. Transmission reliability can be selected on a per-communication channel basis, ranging from none to full receipt acknowledgment. Performance figures for fast ethernet are 96 Mb/s bandwidth, and under 30 μs one-way latency. Implementations for Linux and FreeBSD are available.

Keywords: Message passing, networking, commodity, ethernet, cluster computing.

1 Introduction

As background information to explain the motivation behind the project, we provide an overview of cluster computing, technological developments which have brought us to this situation, and the current state of the art in high-performance networking protocols.

1.1 Cluster computing

A community of enthusiasts of commodity high-performance computing platforms has lived at the fringes of the computing community for many years now, and this bunch is seeing its numbers grow following the general demise of the massively parallel processor (MPP) manufacturers. Even the government research laboratories are joining the fray. Sandia and Ames National Laboratories are two United States Department of Energy research facilities which can no longer satisfy their thirst for FLOPS by buying monolithic multi-million dollar machines, as there is not sufficient market demand to keep vendors in business.

The idea of cluster computing is to aggregate machine rooms full of relatively cheap hardware, connected with some sort of network, and apply the combined force of the individual machines on a single calculation. Problems arise, though, in attempting to operate this set of machines as a single unit. As it is not feasible to run a single operating system on the entire cluster, the alternate paradigm of message passing is used instead. Each processor (which could also be a small shared-memory multiprocessor) maintains a disjoint address space, and messages are passed between machines as driven by the requirements of each application.

The hardware employed in a cluster is generally the most readily available in the volume personal computing market, so as to leverage the cost advantages of buying commodity hardware. The down-side to this is that some critical pieces of hardware for cluster computing are completely irrelevant for the mass

* Corresponding author

market, namely the interconnect. The advent of shared 10 Mb/s ethernet was a giant step, and remains the basis of the standard "fast" networking infrastructure, as it has been for the last 15 years. Within the last few years, though, the price of 100 Mb/s ethernet cards have approached the reach of most users, and commodity gigabit network components are on their way. More esoteric networking components, such as Myrinet and HiPPi are available to those willing to incur the additional costs, and are also improving with time.

Cluster computing designs based around such commodity hardware components frequently also involve the use of commodity *software* components, namely messaging protocols. By and far the most popular is TCP/IP, a scheme designed for use on complex, low-reliability, wide-area networks. While TCP/IP has proven itself very flexible and indeed invaluable for most connectivity requirements, cluster computing applications are poorly served by it. In a system-area network consisting of a well-known set of machines and links, frequently in a single machine room, the overheads of dynamic routing, layered protocol processing, and congestion control algorithms are unwelcome. Cluster computer users can make very good assumptions about the status of their network which permit increases in effective bandwidth, and reductions of user-to-user latencies.

1.2 Enabling technologies

Until recently computer users bought a set of hardware components and expected to use software provided by the hardware vendor. Independent software vendors have always existed, and are willing to sell application codes which will run on multiple hardware designs. Some elements of the software were too closely coupled to the particular processor architecture or hardware setup, however, to make it feasible to have multiple choices thereof, such as the compiler and operating system. The strength of the Gnu compilers and tools has grown to rival, and frequently surpass, that of the tools provided by the hardware manufacturers themselves, for instance.

Only very recently has it become possible to outright replace the entire operating system, however. Two major projects grew out of research efforts to provide a Unix-like operating environment on commodity piece-built machines centered around the Intel x86 architecture. Linux and FreeBSD are both very flexible operating systems, and can be well-tuned to provide performance necessary for large-scale cluster computing. Better yet, both are available free of charge in source code form. Bobnet can be used in conjunction with either OS.

The standardization of message passing semantics in the form of MPI also enables us to address a wide range of parallel applications by ensuring that we sufficiently support a basic set of MPI functionality. Traditional Unix semantics embodied in `read()` and `write()` do not allow for the split-phase, asynchronous communication necessary to achieve optimal performance in terms of latency and bandwidth. The fact that application coders are using MPI moves them closer to exploiting the full power of the underlying hardware. To a lesser extent today, VIA (discussed below) also aids the effort in moving away from traditional blocking communication semantics.

1.3 Networking protocols

The use of networking in computing emerged as a simple, reliable tool used to join geographically isolated research communities. The emphasis in the use of the network was on reliable transport and standard protocols, and eventually grew into modern TCP/IP [8] as discussed above. The high performance crowd has recently been spending much effort in attempting to speed up system-area network communications, but no single effort has crystallized into a noticeable research community. We offer in this paper yet another attempt at a unifying high-speed communication protocol, but believe that the advantages of Bobnet are in its adoption of an emerging standard, and in its abstraction from the hardware layer. First, though, other research efforts are summarized.

U-Net [2] is one of the simplest attempts at circumventing the networking path through the kernel by providing direct access to the network device. It uses the connection-oriented communication concept of an endpoint, and message tags to demultiplex incoming packets. All data to be sent and received must be assembled in a single predetermined buffer area, which can be accessed by both the network device and the user-level program. This necessarily either constrains the form of a parallel message-passing program, or introduces extra buffer copies on both the sending and receiving sides of the communication. Using a

specialized device driver for a card based on the Digital 21140, they do report latencies identical to ours, and very good bandwidth for small messages. Messages larger than the MTU size (1500 bytes) are not supported.

The Virtual Interface Architecture (VIA) is a protocol [3] designed by a consortium of hardware and software companies including Intel, Microsoft, and Compaq. It is heavily influenced by U-Net, and shares the basic design of preposting message receive buffers, and locking into memory user receive buffers. VIA was designed without an underlying wire specification, and is only an interface layer with which operating system libraries are meant to interact with the hardware. Bobnet implements a similar interface, as well as a small VIA compatibility library consisting basically of function name remappings, and further adopts wire protocols for some popular hardware, including ethernet, Myrinet, and ATM. While VIA is designed to guide commodity network interface designers of the future, Bobnet is built to adapt to current hardware technology, and to evolve with little effort to future VIA-compliant hardware devices.

Other research projects exist which attempt to squeeze more performance out of high-end networking components. Active messages [5] embeds extensive support for parallel applications by including a variety of transport operations, including the automatic execution of a "handler" function upon receipt of certain types of messages. The work is focused on Myrinet networks, where each card has a fairly sophisticated processor and some local RAM, and is used to cache virtual-to-physical memory mappings of most recently used addresses. Commodity network interface cards could not be used for this purpose. Performance numbers are good, and reported for Myrinet and FDDI in [6, 7]. Fast Messages [4] from the University of Illinois is similar in its use of custom networks such as Myrinet and that on the Cray T3D, and its use of receive message handlers. In contrast, we focus particularly on high-performance communications on low-cost networks in this work.

Lacking in all of Bobnet, U-Net, VIA, FM, and AM is a suitable wide-area networking scheme. This is understandable and acceptable given the design point for these architectures, for use in machine-room area networks. The design of Bobnet takes into account, though, that legacy codes may exist and need to access the same physical infrastructure that is used by the high-performance communication layer. To this end, Bobnet includes a set of compatibility features to support the full TCP/IP protocol suite. An application can use TCP features on top of a Bobnet network driver, or in the case of ethernet, both protocols can coexist on the same physical medium, as mediated by the particular hardware driver.

As alluded to above, other salient features of generic wide-area communications are avoided in the design of Bobnet by specifying that it is an appropriate protocol only for the system area network. In the current implementations this is certainly true, but some easy extensions would allow for encapsulation of Bobnet packets inside IP, or split up into individual AAL5 ATM cells, which could then be routed using the existing infrastructure. Multicast and broadcast are not supported either. Their inclusion would be trivial using encapsulation methods, as for routing, but their use with a reliability specification is seen now as too much of a burden on the basic high-performance design.

The remainder of the paper is organized as a presentation of Bobnet from the "highest" user-level layers down to the lowest hardware implementations. Figure 1 shows the conceptual overview of the material to be discussed in Sections 2, 3, and 4. Performance numbers on a variety of hardware and software configurations are given in Section 5, followed by a list of directions we plan to pursue in the near future.

2 Library Design

As much as possible, the design of Bobnet will be exposited from the "top" user-visible aspects down to the hardware layer. The application program makes calls into the kernel, with the aid of a library, to arrange for its virtual memory to be made immobile, and to move messages to and from the network. The specific device driver handles the network interface adapter, programming it to perform DMA transfers to and from the network as instructed by the user, and also requests services of the kernel for exceptional conditions.

2.1 User interface

Most users will not be directly aware of the presence of Bobnet, as they will program their parallel application codes in MPLite [1], a subset of the current standard in high-level message passing libraries for single-program multiple-data (SPMD) applications. MPLite is the crucial subset of all MPI calls which satisfy

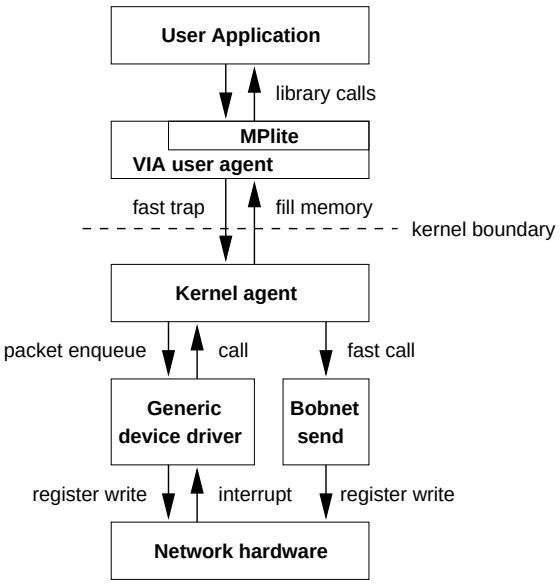

Figure 1. Illustration of conceptual separation between the various components in Bobnet. The two different paths to the hardware are explained in Section 3.

most applications. Support for the official MPI specification would certainly be possible, but not crucial to support high-performance applications.

The general sequence of message passing in many codes takes the structure shown in Figure 2. Each processor in this data-parallel SPMD application manipulates its assigned portion of the data, interspersed with instructions to exchange subsets of the data with other participating processors. To achieve the highest levels of scalability, code must be written to expose as much parallelizability as possible. Application programmers frequently use split-phase operations, as shown in Figure 2, to hide the latencies of moving data to and from the network.

```
for (;;) {
    // do some work
    calculate_results();
    // inform the driver it may begin to send data
    register_receives();
    start_data_sends();
    // meanwhile, do some work on the data that arrives
    while (receives_outstanding()) {
        wait_for_some_data();
        calculate_properties();
    }
    // make sure all the data has moved before reclaiming it
    verify_sends();
}
```

Figure 2. High level code example.

The code sequence shows an iteration, possibly over time or steps in a relaxation method, where the same operations occur in order. Ignoring initialization, first each processor does its "main" work, such as calculating the right-hand sides of a set of ordinary differential equations, for its assigned set of data. Next the processors must exchange data. Each informs the network card where it would like incoming data to arrive, then tells the card (or driver) which data it would like to send. As the network is much slower than

the processor, the latter must wait for the data motion to complete before going on. However, as each chunk of data arrives, some amount of processing may be able to occur, helping to hide network latencies.

The above discussion appeals to the current state of the art in application programming, and ignores a bevy of potentially very valuable research in alternative programming styles. Multi-threaded codes employ efficient task switching to allow, for instance, a network thread to block on the completion of a data receive operation while a computational thread is working on data which is already available. While such an approach will simplify the work of the code developer, the sequence of events "under the hood" will remain the same due to fundamental algorithmic dependencies (*i.e.,* a certain calculation must wait for data to arrive before it can proceed). Thus we continue with the naïve view of the application programming world.

One complexity which must be dealt with is paging. Operating systems like to reserve the freedom to move data around to increase system throughput, such as moving unused data pages from memory to disk when free memory grows low. To avoid requiring the network driver to be aware of the status of all memory pages, and to avoid having to wait for a page to be swapped back in from disk, Bobnet arranges for memory regions which are to be used in an application for message passing to be immobile. The VIA specification describes a set of calls to communicate these regions, and Bobnet implements a working protocol.

2.2 User library

The foremost visible library component is an implementation of the MPLite [1] specification as portrayed in the previous section. These standard calls are expressed internally in terms of Bobnet primitives, which will be detailed here. The functions are designed to match those specified by VIA, although we take the liberty to use native types in this discussion where it is more intuitive. Some functionality indicated in the VIA specification is omitted, namely support for completion queues, remote DMA, and some other higher-level calls are not yet available.

Many calls specify a `VIP_RETURN` type as a parameter, which is an integer enumeration of `VIP_SUCCESS` and various error return conditions. Interesting "data" is returned through argument pointers.

Device Interface

The physical interface is opened and closed by each application by the calls:

```
VIP_RETURN VipOpenNic(const char *DeviceName, int *NicHandle);
VIP_RETURN VipCloseNic(int NicHandle);
```

which serve to maintain a reference count of outstanding users on each device. On close, outstanding transactions and associated memory handles are deleted.

Each application generally deals with only one physical device, but may have multiple virtual interfaces to that device which vary in attributes such as transfer reliability level and quality of service. Virtual interfaces may be associated with send and receive queues used for the aggregation of events posted to multiple VIs.

```
VIP_RETURN VipCreateVi(int NicHandle, struct vi_attribs *ViAttribs,
                       int SendCQHandle, int RecvCQHandle, int *ViHandle);
VIP_RETURN VipDestroyVi(int NicHandle, int ViHandle);
```

An important part of the `vi_attribs` structure includes the specification of the remote node with which to communicate. The VIA specification is conspicuously silent about the network addressibilty aspects such as identifier format and name resolution protocols. We have adopted the six octet hardware address present in all ethernet cards as our naming scheme, along with a two octet port number, and assumed that no routing is necessary. Clearly more work will be required to support networks using mixed media types to convert between the various addressing schemes.

Memory Management

Memory regions to be used in communications of the user process must be registered with the kernel so that it can ensure that the memory remains physically accessible by the network hardware. The kernel generally does this by "pinning down" the memory against mechanisms which attempt to move inactive regions to slower areas (such as disk).

The basic calls are:

```
VIP_RETURN VipRegisterMem(int NicHandle, void *VirtualAddress, int Length,
                          struct vi_mem_attr *MemAttribs, int *MemoryHandle);
VIP_RETURN VipDeregisterMem(int NicHandle, void *VirtualAddress, int MemoryHandle);
```

Registration takes the starting address (in the natural virtual memory space of the process) and length of the region, along with attributes which can specify usage patterns, cacheability, and alignment. The `MemoryHandle` is used to identify the region in later transactions. Deregistration is not strictly necessary, but can help the kernel to free up unused areas. Some physical devices may not support a large addressable memory map and may require deallocation to retain access to a fixed address space. (The virtual address is not needed by the current library in the deregistration call, but serves as a reminder to the code developer. Future implementations may allow for multiple overlapping mappings of memory, partial deallocation, and the mapping of shared memory segments, in which cases the virtual address may be required.)

Sending and receiving

Data motion operations operate on descriptor structures which have the form:

```
struct vi_cseg {
    vi_void64 Next;
    u_int32_t NextHandle;
    u_int16_t SegCount;
    u_int16_t Control;
    u_int32_t Reserved;
    u_int32_t ImmediateData;
    u_int32_t Length;
    u_int32_t Status;
};
```

```
struct vi_dseg {
    vi_void64 Data;
    u_int32_t Handle;
    u_int32_t Length;
};
```

(called `VIP_CONTROL_SEGMENT` and `VIP_DATA_SEGMENT`, respectively, by the VIA specification) and serve to carry information about locations in memory that are the targets of send or receive operations, allowing for a general scatter/gather capability. Many of the fields are not intended to be filled from the user side, but are maintained by the hardware device (or kernel hardware device driver) as it processes its queue of outstanding operations. In our case the library manipulates the `Next` field without intervention from the operating system when it detects an already pending queue of operations. Users may chain together multiple requests to allow for general scatter/gather operations using a single control segment and multiple disjoint data segments (as indicated in the `SegCount` field), and thus a single interaction with the lower layers. The sum of the `Lengths` of the data segments is presented in the `Length` of the control segment. The `Status` field is marked by the network device as it processes the descriptors.

All of the descriptors are present in the user segment of memory, which can be made accessible both to the network device and to the operating system. Placing the descriptors in either kernel-only memory, or on the card would make some accesses much slower, or break protection boundaries, or both. Addresses in a descriptor are user virtual addresses as well.

The basic operation used to move data to a remote node is:

```
VIP_RETURN VipPostSend(int ViHandle, struct vi_cseg *DescriptorPtr, int MemoryHandle);
```

The memory referenced by the `DescriptorPtr` will not be copied into send buffers, and must not be altered until the send operation has completed. This differs from the standard Unix semantics of `write()` or `send()`, and corresponds roughly to the MPI call `MPI_Irsend`. Higher level libraries such as MPI or PVM frequently use a user-specified tag value to select among multiple message types. In Bobnet this mechanism is fully controlled by the upper-level library, in our case MPLite, as this avoids overloading the basic message moving mechanisms with operations in the critical path which are not needed by all applications.

To initialize a receive operation, the following call is used:

```
VIP_RETURN VipPostRecv(int ViHandle, struct vi_cseg *DescriptorPtr, int MemoryHandle);
```

Again, the memory specified by `DescriptorPtr` may be filled with an incoming message at any time and must not be relied upon to retain its current values. Each posted receive can accept only a single incoming message, though the incoming message need not fill the entire set of receive buffers. This allows for the

receipt of messages for which only an upper bound of the message size is known, and obviates a second send to transfer the message size (but only when explicitly requested in the attributes structure).

Probing and Waiting

Synchronization points must be included in any correct program to ensure that operations on data which are specified to be sent or received complete by the time their values are needed. On the sending side, the basic call:

```
VIP_RETURN VipSendDone(int ViHandle, struct vi_cseg **DescriptorPtr);
```

checks for any outstanding sends on the `ViHandle`, and returns in `DescriptorPtr` the first one which has completed, or `VIP_NOT_DONE`. This call can be used inside a probing loop for completion.

The equivalent blocking call is:

```
VIP_RETURN VipSendWait(int ViHandle, unsigned long TimeOut,
                       struct vi_cseg **DescriptorPtr);
```

which can be viewed as a loop around calls to `VipSendDone()`. The `TimeOut` parameter specifies in arbitrary (but monotonically increasing) units how long to spend in a busy-wait loop before switching to an alternating sleep/test loop. A `TimeOut` value of zero indicates that the call should never sleep.

In a completely symmetric fashion, the message completion test calls to be used on the receive side are:

```
VIP_RETURN VipRecvDone(int ViHandle, struct vi_cseg **DescriptorPtr);
VIP_RETURN VipRecvWait(int ViHandle, unsigned long TimeOut,
                       struct vi_cseg **DescriptorPtr);
```

The blocking form of the calls were included because the implementation allows for a blocking wait which is much more efficient than the equivalent user-level test-and-loop version. These two primitives (in each direction) together can be used by the message passing library to implement a variety of message send completion probing. The current implementation does not yet support generic completion queues which were introduced in the call `VipCreateVI`; this feature will soon allow for efficient testing of completed operations on multiple handles.

We have not yet implemented the completion queue mechanism, which would be equivalent but hopefully faster than the generic Unix `select()` mechanism.

3 Kernel Implementation

The initial implementation of Bobnet was written under FreeBSD 3.0. However, the Bobnet kernel agent should be fairly portable to other architectures and operating systems as it attempts to provide a flexible interface to the device hardware. Depending on the particular device, the interface with the kernel agent may be done efficiently using a small set of routines. This layer, however limited, is key in allowing Bobnet both to coexist with other networking protocols, as well as leverage from an extensive existing hardware driver base. It will also allow Bobnet users to take immediate advantage of the next commodity performance jumps, such as will occur in standard ethernet, as they occur. Our goal is to keep as much functionality *out* of the kernel as possible, since transitioning the thread of control from a user's application into the kernel incurs a significant amount of overhead (on the order of 1 μs at best).

Most current high performance messaging implementations are tied very tightly to the hardware. Although this approach has proven itself in very good performance, it has also led to considerable fragmentation of both coding and conceptual efforts, becoming a barrier to the wide scale adoption of clustering technologies. One testament to the current problem can be found in the growing number of custom protocols and control programs for the popular Myrinet network interfaces. On the other hand, the recent VIA specification is an attempt to standardize the hardware interface as seen from a user perspective, but its very nature precludes the use of technologies such as ethernet. It also specifies neither a wire protocol nor a coherent and compatible user interface.

Taking these issues into account, Bobnet was designed with a different approach. Internally, Bobnet uses a descriptor-based architecture which is very similar to VIA or U-Net, but externally Bobnet maintains the goal of supporting a simple message passing interface. In VIA, the descriptors are passed directly to the

hardware, while in Bobnet they are simply used to describe the data layout specified by the user in a generic way.

It is up to the kernel and low-level library how the data is actually interpreted, and how it is sent or received by the actual device. Indeed, the descriptors may even be used by a VIA-compliant interface, or translated into the format expected by non-compliant (commodity) hardware. Using a descriptor-based architecture also provides for an inherently asynchronous interface, allowing for the more effective overlap of computation and communication which can mask latency in the network.

In the particular case of Bobnet on ethernet, the kernel and library glue translates the user's send descriptor into kernel headers which are presented directly to the network interface with a pointer to the user's data. When packets arrive on the interface, Bobnet fields the interrupt and copies the incoming data directly to the appropriate location, which is calculated by searching through the table of posted receives, looking for the descriptor attached to the head of the incoming message. Together, the implementation consists of a zero-copy send, single-copy receive directly from user space, and uses the existing hardware driver provided by the operating system.

3.1 Memory Registration

As described earlier, the application's buffers must be made immobile before attempting to send or receive data. To accomplish this, Bobnet provides an implementation of VIA-style memory registration. User memory regions which are to be accessed by the network device must be wired down by informing the operating system that the region should not be moved. Bobnet commands which refer to locations in memory use a memory handle, which is internally an index into an array of structures which contain information such as the physical and virtual addresses of the region, its length, and any special attributes. Kernel and network routines use the memory handle and the associated table to convert quickly between user-space addresses and kernel-space (or physical) addresses, which can then be converted to bus addresses by the device driver.

The alternative to this memory handle scheme would be to force the user to present all data in physical addresses, or to allow the Bobnet routines access into the kernel core memory management tables to check for accessibility of all addresses, and to perform translation. If messages arrive when the requesting user process is not in context in the kernel, the virtual mappings are not naturally available to the interrupt service routine, and they would have to be converted and checked. Registered memory handles are used to reduce this overhead and uncertainty.

3.2 Endpoints and Connections

To share the network interface, Bobnet provides an endpoint abstraction, similar to a Virtual Interface in VIA. Each endpoint is uniquely identified by an address/port pair, and can be connected to a similar endpoint on a remote host. For each endpoint, there is an associated queue of send and receive descriptors, with which the application program communicates.

The VIA specification states that all compliant devices will support unreliable delivery, but support of reliable delivery or reception is optional. To implement reliable delivery (ACK-based), Bobnet appends some extra fields in the header of each packet which contains a sequence number and an acknowledgment number. The selection of reliable delivery occurs by the user setting the appropriate flag in the `vi_attribs` structure during handle creation.

The sending and reception of messages larger than the hardware maximum transmission unit (MTU) is achieved in the Bobnet layer of the kernel, as most non-programmable underlying device drivers are unable to perform this functionality by themselves. Since Bobnet uses a descriptor-based architecture which can refer to arbitrary addresses in user space, it is possible to queue the send or receive in MTU-sized pieces to the device with little overhead.

3.3 Send and Receive

As described earlier, send and receive commands in Bobnet are each described by an address, a length, and a message tag. The tags are managed at the user level, being associated with the appropriate descriptors. To initiate the operation, the Bobnet user agent (in the routine `VipPostSend` or `VipPostReceive`) adds a descriptor to the appropriate queue, filling in the address and length. Messages longer than a hardware MTU

are handled by the driver, and do not complicate matters at this level. Once the descriptors are queued, the Bobnet user agent makes a fast trap into the kernel to start the operation in the hardware.

The fast trap mechanism is used by Bobnet as current hardware device drivers do not have the hardware doorbell mechanism as required by the VIA specification. This trap typically takes less than 1 μs as shown by our measurements. When hardware arrives which can support the doorbell mechanism, this trap time will be removed at both the send and receive transactions.

Upon the fast trap, the Bobnet kernel agent looks through the newly queued descriptors, and performs the necessary translation into the format required by the underlying driver, or by the hardware itself. The basic method of interacting with the hardware goes through the generic device driver provided by the kernel. This has the advantage of providing a great deal of flexibility and enabling Bobnet to operate on a wide range of hardware. The more specialized method of interacting with the hardware affects only the send operations, and is shown on the right in Figure 1.

For a FreeBSD ethernet driver for example, Bobnet will translate the virtual addresses in the descriptors into `mbuf` chains. The first `mbuf` will contain a copy of the ethernet header and Bobnet header. Depending on whether the data region crosses a page boundary in memory, there may be one or two more `mbufs`. The `mbufs` following the header are of type `M_EXT`, which designates the use of external storage, and may each have their own free routine associated with them. Once the chain is built, it is queued, and the driver's output routine is called.

The hardware device notifies the kernel through an interrupt mechanism when data has arrived at the interface. The kernel looks at the type field in the ethernet header, and passes Bobnet packets to our input routine. It ensures that a pre-posted receive is waiting for the packet at its destination endpoint, copies the data into the user's posted buffer, updates status fields for reliable transport operation, and frees the incoming buffer. This single buffer copy is necessary to use existing ethernet hardware in a generic fashion. The hardware expects a certain format for its incoming buffer space which does not map well to the Bobnet (or VIA) format, in particular, most hardware is incapable of determining the destination address for an incoming packet but simply writes it into the next available receive slot.

The rest of the kernel agent handles the mundane tasks of keeping reference counts for open and close operations, and creating and destroying VI handles when called to do so by the user agent. The interface to these non-critical path routines is through the standard character device operations including `open`, `close`, and `ioctl`.

4 Device drivers

We take two major approaches to driving the actual hardware present in a system using Bobnet. The first and most portable is to use the existing "native" general-purpose drivers, and insert hooks to Bobnet by adding it as another ethernet protocol type (or in general, IEEE 802.2 logical link control). The various types of traffic which can appear on an ethernet network are segregated by using distinct values of the `type` field in the standard ethernet packet header. (Devices which communicate using non-ethernet media have similar methods of classifying types of traffic present on the wire.) This approach has the large advantage that no new device driver code must be written, and that other ethernet traffic continues to exist on the network along with the new Bobnet traffic. It also has the advantage of providing a great deal of flexibility and enabling Bobnet to operate on a wide range of hardware "out of the box."

We have found, though, that it is possible to achieve lower latencies by replacing the stock driver with a custom module designed for the needs of Bobnet. The largest performance win is in avoiding the translation from Bobnet to a linked list of `mbufs` (or `sk_buffs`) to the native format of the device, but this savings is only on the order of one microsecond saved. With a hand-crafted driver, Bobnet data is written directly to the card without constructing the intervening `mbufs`. An intermediate approach, which we have yet to attempt, would be to extract the shared components of both the "stock" and the Bobnet versions of the device driver into a shared module. These would be register specifications of the card, and the maintenance of locks on exclusive card resources. The Bobnet and `mbuf`-based interfaces could then coexist and access this shared module by acquiring locks for exclusive card access.

Actually, only one routine of each driver is replaced: the packet output routine. The generic input and interrupt-handling routines are found to be sufficient. A further improvement would be to add a specialized

input routine as well. This would be suitable only for programmable (and probably non-commodity) cards, but would allow a true zero-copy receive. This specialized input routine would also destroy the backward compatibility which we maintain by using the stock device driver. The performance results below use only the generic device driver.

5 Performance

In this section we present the relevant quantitative network performance numbers for a sampling of ethernet devices, and compare against TCP/IP, just to illustrate the baseline quantities. In all cases, the tests were performed between two identical machines, connected directly back-to-back. The machines are dual Pentium Pro 200 MHz, 512 kB cache with Asus P/I-P6Up5 motherboard and 256 MB of memory.

Data for three different network interface cards are presented. "Tulip" are SMC EtherPower 100 Mb/s cards based on the Digital 21140 chip. "Intel" are Intel EtherExpress 100 Mb/s, based on the Intel 82558. Together these two basic network card designs cover most of the interesting installed base for cluster computing. The other possible contender would be, perhaps, one of the recent 3Com cards. These, however, have given us quite an array of problems over the years and it was decided they were not worth further consideration. The throughput performance achieved with both Bobnet and traditional TCP are shown in Figure 3. All bandwidth numbers refer to user payload only; accounting for ethernet and Bobnet headers would increase all the figures by a few percent.

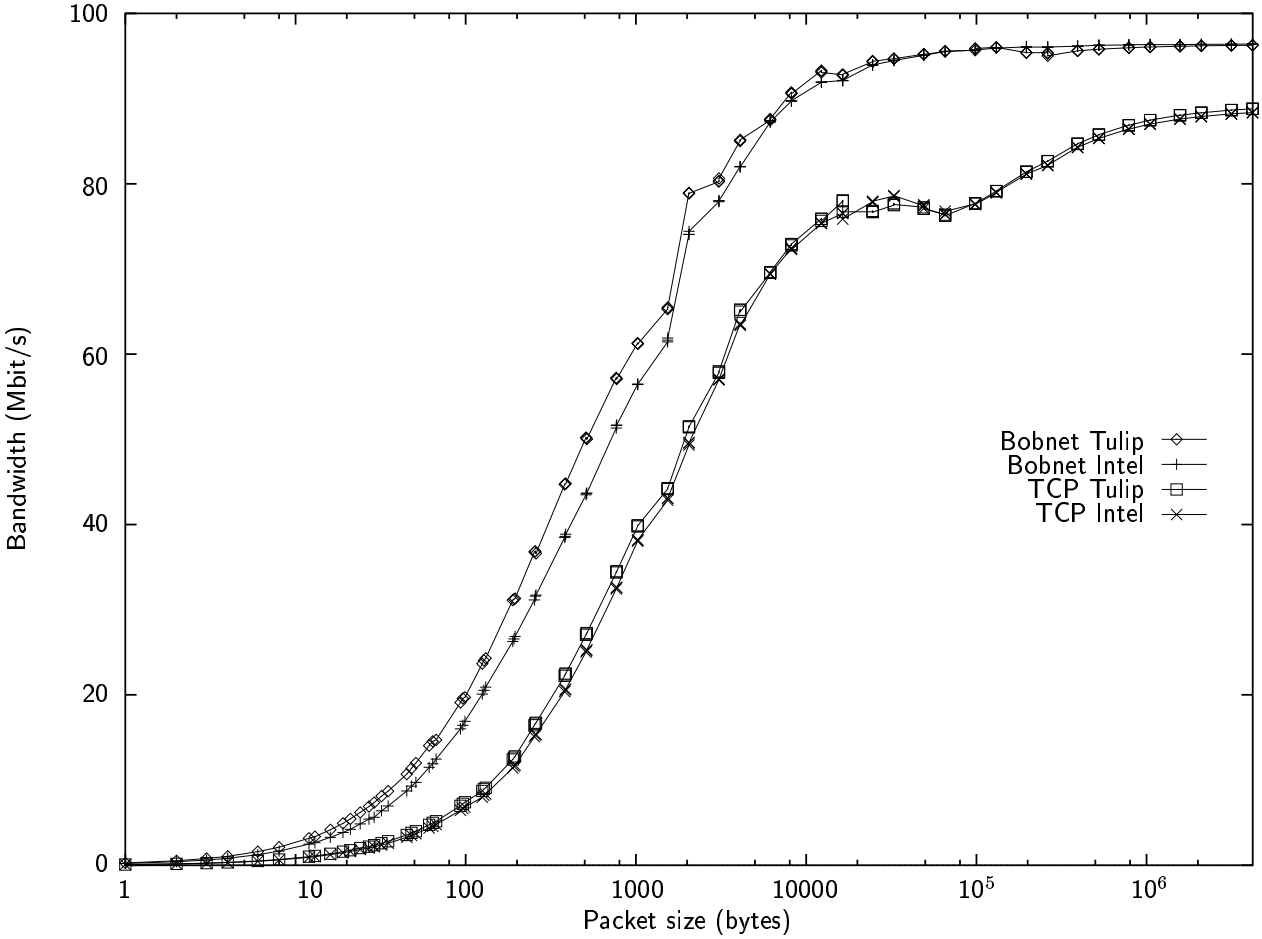

Figure 3. Bandwidth figures for the 100 Mb/s devices.

The third card is a prototype Gigabit Ethernet (IEEE 802.3z) implementation from Packet Engines. Results shown in this section were obtained using the G-NIC (revision 2) PCI design, but we hope soon to

test the newer design (G-NIC II) which has been promised to improve throughput performance, mainly by solving the PCI burst size limitation present in the current cards. To our knowledge, no one has been able to achieve the theoretical bandwidth using these devices. Nevertheless, they are significantly faster than fast ethernet, and not too much more expensive (in terms of MB/s/$). Our results are shown in Figure 4, which uses a different y-axis scale than Figure 3.

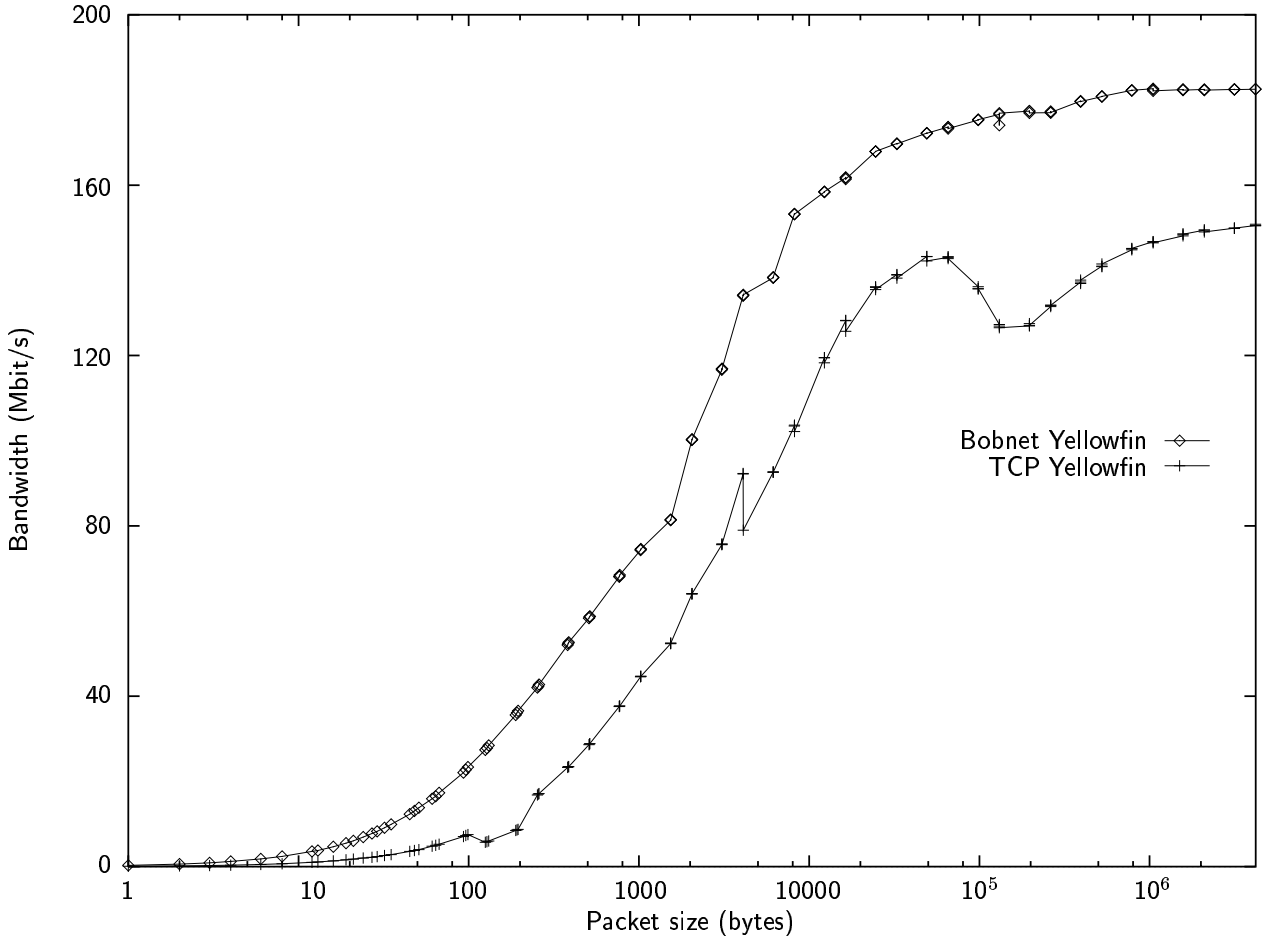

Figure 4. Bandwidth figures for Gigabit Ethernet.

Overall one-way latencies were measured for all three cards, using both Bobnet and TCP/IP, and are shown in Figure 5. All the curves level off beyond a message size of 10 kB or so, at slopes shown in Figure 6. Bobnet outperforms TCP even in the large packet domain for latency, and the Gigabit Ethernet device achieves the best performance by nearly a factor of two. Timings in the kernel of various sources of latency show that only 3–4 μs are spent in each of the transmit and receive routines, which leaves roughly 18 μs unaccounted for that must occur inside the network devices themselves. Propagation delay across the short copper or fiber optic runs is negligible on this scale. Incidentally, the results presented for U-Net [2] agree with this finding. This may be the unavoidable edge achieved by high-cost networking components such as Myrinet—their fundamentally lower hardware latency.

An explanation for the large latencies delivered by the Yellowfin hardware in both the Bobnet and the TCP cases may be found in the descriptor format used by the card. Multiple data buffers must be posted each in a separate descriptor, while the Tulip can group multiple sources of host data in a single card descriptor format. The card may then require twice as long in its interaction with the host computer bus to perform the same amount of work as the Tulip or the Intel.

It was mentioned above that we have written replacement output routines for these devices, hoping to save some time spent in `mbuf` manipulation. In fact, though, these routines save about 1 μs on each side of

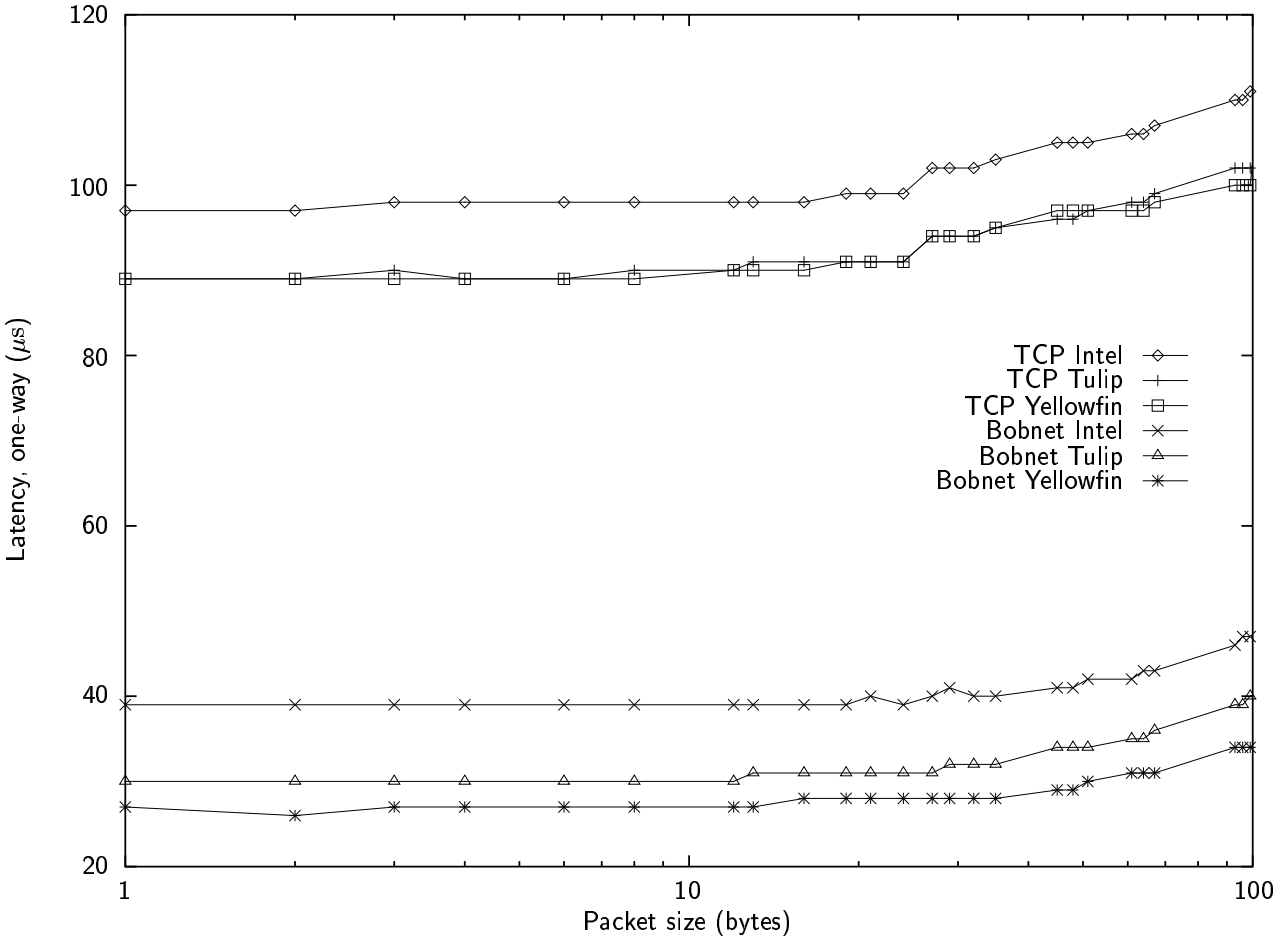

Figure 5. One-way latencies for all network interface cards.

the transaction, which is not too significant in the presence of the large delays introduced by the hardware itself. Future versions of these cards, especially the Gigabit Ethernet devices, may make this source of performance improvement significant, however.

Interface	Slope (μs/byte)
TCP Intel	0.0859
TCP Tulip	0.0855
TCP Yellowfin	0.0497
Bobnet Intel	0.0829
Bobnet Tulip	0.0830
Bobnet Yellowfin	0.0437

Figure 6. Asymptotic latencies.

The test used to gather the above data is a simple two-machine ping-pong. The sender machine delivers a message to the receiver, which bounces it back as soon as it can. The latency is then half of the total time as measured at the sender from message transmission to full reception, and the bandwidth is twice the size of the message divided by the total time. These tests are repeated hundreds of times in succession to swamp out measuring noise. In actual practice the systems will be used in asynchronous mode, where the sending machine does not wait for the message to be returned to it, but can continue to send more data. This pipelining of multiple packets leads to much increased bandwidth in the small packet case, but it is harder to measure reliably the latency of such a transaction. We suspect that the actual latency in the

asynchronous, streaming case will be lower, too, as the FreeBSD kernel requires about 5 μs between the last instruction of the Bobnet receive routine and the first instruction of the following call to the send routine for this ping-pong test.

We do not present any timing numbers for the higher level TCP emulation interface, or for the MPLite message-passing interface. Future efforts will involve optimizing those sections of the code base so that they perform as well as the underlying Bobnet protocol itself.

6 Status

The initial implementation of Bobnet consisted of a kernel module for either Linux (v2.1.107 and higher) or FreeBSD (v2.2.6 and higher), and a library for communicating with the module using standard `ioctl()`, `read()`, and `write()` system calls. Higher layer libraries communicate with this interface library and require no system-specific information. Stable source distributions and more recent development efforts are available at `http://dancer.ca.sandia.gov/bobnet`.

7 Future directions

Enhancement of the reliability layer in Bobnet, as expressed in VIA primitives, is seen as an important short-term goal for increasing usability of the network for urbane applications. The current naïve implementation of reliability protocols is too costly for Bobnet, and will be augmented with a combination of negative and positive acknowledgment protocols which attempt to optimize for the bandwidth-delay product of the network.

Generic wide-area network support is not planned, as Bobnet introduces a new ethernet type which may not be routed by the current infrastructure. Trivial encapsulation of Bobnet packets inside IP is conceivable but not recommended given the performance implications. The focus for this network protocol is for machine room sized networks.

The addition of optimized drivers for the newest hardware will continue to be a closely pursued moving target. In particular, we would like to have a native ATM driver to employ on dedicated long-haul high-bandwidth networks. The acceptance of the VIA standard by the networking community brings promise of the introduction of very high-speed and inexpensive network interface cards. The lack of vendor support for software drivers for "fringe" operating systems such as Linux and FreeBSD puts Bobnet in a position as being likely to extract most of the performance of such new hardware with the minimum of impact in terms of new driver development. We currently are attempting to arrange with Giganet for the sharing of information regarding their VIA product which is in the beta testing stage today.

Implementations on programmable network interfaces are also possible, and Myrinet is a good candidate. Depending on the target application, clusters using Myrinet may exhibit the correct price/performance point and warrant the implementation of Bobnet on that hardware.

We have plans to support heterogeneous clusters by implementing a receiver-makes-right scheme for data transfer. Optimally this would happen during the single copy necessary at the receive side in the kernel, but extra information regarding the types of the various words in the data must be available for the kernel to do this swapping or conversion. The code is written using explicit word length typing (*i.e.,* we use "uint32" instead of "unsigned long" when we mean that) to facilitate porting to architectures of different intrinsic word length. All development and testing was performed on Intel IA-32 hardware.

8 Conclusion

We have presented an overview of the design and implementation of a new low-latency (less than 30 μs) and high-bandwidth (near 100% of theoretical, accounting for headers) message-passing protocol compatible with the emerging VIA specification. Bobnet features zero-copy sends, and single-copy receives, and support for all network interface drivers present in either of the currently supported operating systems (FreeBSD and Linux). Higher level protocols, including MPLite and TCP, are supported in user libraries. Transmission reliability is optional, and managed by the kernel agent.

One of the greatest stumbling points in current cluster computing attempts is the lack of an efficient network protocol. TCP/IP is very standard, and very slow. Custom hardware such as Myrinet can be fast at times, but is expensive. With the introduction of Bobnet, we hope to unify the worlds of standard ethernet hardware and high-performance message passing software to enable a new level of cluster computing usability and performance.

References

[1] Turner, Dave. "MPLite: A light-weight message passing library." `http://cmp.ameslab.gov/cmp/clusters/MP_Lite.html`.

[2] Welsh, M., Basu, A., and von Eicken, T. "Low-Latency Communication over Fast Ethernet." *EuroPar '96*, Lyon, France, August 1996.

[3] *VI Architecture.* `http://www.viarch.org`.

[4] Pakin, S., Karamcheti, V., and Chien, A.. "Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors." *IEEE Concurrency*, **5**, April–June 1997, pp. 60–73.

[5] Mainwaring, A. and Culler, D. "Active message applications programming interface and communication subsystem organization." Technical report, UCB. `http://now.cs.berkeley.edu/Papers/Papers/am-spec.ps`.

[6] Lumetta, S., Mainwaring, A., and Culler, D. "Multi-protocol active messages on a cluster of SMP's." Proceedings of SC97, San Jose, CA, November 1997.

[7] Martin, R. "HPAM: an active message layer for a network of HP workstations." Proceedings of Hot Interconnects, 1994. `ftp://ftp.cs.berkeley.edu/ucb/CASTLE/Active_Messages/hotipaper.ps`.

[8] Stevens, W. R. *TCP/IP Illustrated.* Addison-Wesley, 1994–1996.