HEWLETT PACKARD

Hamlyn: a high-performance network interface with sender-based memory management

Greg Buzzard, David Jacobson,
Scott Marovich and John Wilkes

Computer Systems Laboratory
Hewlett-Packard Laboratories, Palo Alto, CA

HPL–95–86
July 1995

HP Laboratories Technical Report

This paper appears in the proceedings of the Hot Interconnects III Symposium, Stanford University, Palo Alto, CA, 10–12 August 1995.

Hamlyn: a high-performance network interface with sender-based memory management

Greg Buzzard, David Jacobson, Scott Marovich, and John Wilkes

Computer Systems Laboratory,
Hewlett-Packard Laboratories, Palo Alto, CA

The Hamlyn architecture for fast processor-interconnect interfaces uses sender-based memory management to eliminate receiver buffer over-runs, and a combination of protection algorithms that allows untrusting applications direct, concurrent access to the interface hardware, with full protection between them.

We report here on some advances in the Hamlyn design since our original paper,. We also give detailed performance information for both a prototype of the interface itself using the Myrinet interface from Myricom, and for a software protocol layer that sits atop the hardware interface. We show that Hamlyn performance is comparable to aggressive implementations of Active Messages on the CM-5, but Hamlyn also adds protection between applications and against faulty processors. Our analysis shows that if the interface were constructed as a set of hardware state machines, its performance would be limited almost solely by host bus and link speeds.

1 Introduction

Processors are getting faster quickly, and message-passing interconnects for multicomputers are doing the same, thanks to developments in Gb/s links and low-latency switching fabrics. But published papers about state-of-the art multicomputers continue to discuss application-to-application message times that are typically measured in hundreds of microseconds. It seems that moving bits back and forth between processors and interconnects continues to be a performance bottleneck.

Hamlyn is an architecture for processor-interconnect interfaces that addresses this difficulty. It has several important features:

- *sender-based memory management:* the sender never sends unless it knows that there is space at the receiver, which eliminates costly receiver buffer-overruns and the need to retry sends under high load;
- *direct application access to the interface hardware:* eliminates the need for operating system (OS) intervention in send and receive operations, giving very low latencies;
- *zero-copy protocols:* data are read directly from and written directly to their end-points in application space, with no intervening memory-to-memory copy and no need for page alignment or remapping by the OS;
- *full inter-application protection:* allows many applications to use the same interface concurrently without having to trust one another;
- *automatic message reassembly* even if packets arrive out of order, which allows use of adaptive-routing networks having greater throughput and fault-tolerance;
- *simple design:* the interface can be implemented directly in hardware state machines for speed;
- *protection against node failures and rogue messages:* eliminates need for per-message software checks;
- *a rich set of message-delivery notification schemes* gives applications choices between minimizing latency and maximizing functionality.

The first Hamlyn design incorporated most of these features [Wilkes92]. It was developed as a way to integrate a fast, packet-switched interconnect, known as FedEx,[1] which supported adaptive routing into a large-scale MIMD multicomputer. This work expands on that design by:

- reporting performance data from software and hardware prototypes;
- extending the schemes available for notification of message arrival;
- extending the original packet-counting scheme to a more powerful one that supports generalized group-receive semantics;
- presenting low-level protocols that sit atop the hardware interface.

Hamlyn was designed with a RISC-like philosophy: make the common cases fast, and the less common ones

[1] FedEx was the successor to the PostOffice switching chip developed by the HP Labs Mayfly project [Davis92].

possible. An explicit design goal was that Hamlyn be simple enough to implement directly in hardware state-machines, so that it would not require a programmable controller, since these are often performance bottlenecks.

The Hamlyn design is being prototyped in hardware and software as this paper is being written. This paper presents an overview of its architecture, discusses the design and implementation of the low-level interface library that we have constructed on top of it, and presents performance data for our approach, demonstrating its advantages. We also discuss related work before summarizing what we've learned from our research.

2 The Hamlyn architecture

We begin with an overview of the Hamlyn architecture proper.

2.1 Assumptions and requirements

The architecture was designed for closely-clustered computers or MIMD multicomputers. Our main goal was to reduce the end-to-end latency and overhead of communications while providing a choice of message-arrival semantics. We began by identifying desirable properties of interconnects used in such environments that we could exploit, including:

- *Very low transient error rates:* dedicated, enclosed multicomputer networks are better thought of as extended backplanes than as high-error-rate network links. This meant that we could rely on application-level recovery mechanisms for the very few cases where things actually did go wrong. In particular, retry mechanisms in the low-level protocols to gain efficiency were no longer needed [Saltzer84], and a sender may discard its copy of a message as soon as it has moved into the interconnect.
- *Relatively small packets,* which permit simpler, faster switches and better throughput guarantees; in turn, this implied the need for message segmentation and reassembly. (Hamlyn would also work with interfaces that don't do packetization.)
- *Possible out-of-order packet delivery,* since this allows low-level packet-routing optimizations, such as adaptive routing, to get higher effective bandwidth with automatic hot-spot and fault-avoidance.
- *The interconnect would never partition:* the combination of low failure rates and internal redundancy in the interconnect meant that we could simply declare that partitioning had been legislated out of existence.
- *A physically secure network,* so that messages would not need to be encrypted. A few bad packets might still be generated by a failing processor as it went down, so we did need a mechanism to handle short-lived bursts of erroneous messages. Because we also assumed that individual nodes would run a trusted OS, which would prevent the most egregious security violations, we did not need other protection against sustained, malicious attacks.

To provide really *low latency,* we felt that it was essential to: (1) provide direct, application-level access to the interface hardware; (2) eliminate memory-to-memory data copying ("zero copy" protocol stacks); and (3) provide a fast, low-cost mechanism to notify application processes when one or more messages had arrived, rather than do so on every packet or message-fragment.

To provide *high bandwidth,* we needed to integrate direct memory access (DMA) capabilities into the interface, while still allowing applications direct access to it.

In addition, because we were interested in supporting a general-purpose computing system on our multicomputer, we felt that it was imperative to give several, mutually suspicious applications access to the interconnect interface simultaneously, at full speed. This requirement rules out approaches like partitioning the multicomputer, draining the interconnect during program-switches as is done on the CM-5, or requiring gang-scheduling of applications, as in Active Messages. The same reasoning led us to minimize the need for polling the interface hardware.

The Hamlyn interface architecture embodies solutions to all of these problems.

2.2 Sender-based memory management

The first—and perhaps most important—Hamlyn feature is *sender-based memory management*.

Receiver buffer-overruns are a serious problem for low-latency computer systems: resolving them typically requires time-out mechanisms in order to trigger a retry, often at a relatively low level in the protocol stack. Of course, this usually occurs under high load conditions—precisely when such retransmissions will only make the problem worse. In a packet-based environment, even a low rate of packet loss can result in a much higher rate of message loss.

The basic idea behind sender-based memory management is that if the *sender* has responsibility for laying out messages in the receiver's memory, then there is never a reason to experience a receiver buffer-overrun.

Hamlyn achieves this by letting senders dictate where in the receiver's memory a message should be placed. The interface puts message packets directly into this managed space, rather than buffering them on a limited-capacity interface card. Hardware delivery is needed because the cost of software packet-handling at the receiver is precisely the problem causing receiver overruns.

In order to decouple the sender's and receiver's virtual-to-physical address mappings, Hamlyn provides a level of indirection: each sender is given one or more *message* *areas*—logically contiguous pieces of memory—into which it can direct messages. Message areas are allocated by the receiver so that the latter retains overall resource control. These areas are mapped directly into receiving applications' address spaces so that incoming messages arrive without copying.

To make this possible, each message is labeled with a destination <message-area, offset> pair. If the message must be segmented into packets for transmission, then each packet must include a <message-area, offset> pair.[2]

Message areas are allocated from the processor's main memory, and they are protected by standard hardware page-protection schemes. Thus, there is no application cost to access their contents, and all accesses are subject to the usual protection checks. Each message area is described by a vector of physical page addresses; the Hamlyn interface interprets offsets into the message area as offsets into this vector, and it performs the required address calculations "on the fly" as it writes packets into the receiving processor's memory.[3]

To ensure that senders cannot randomly write to message areas without permission, the sender puts a protection key into outgoing packets, and the receiving hardware checks it against a key associated with the message area. Only if the keys match is writing allowed. Since keys are large (32 or 64 bits) and sparsely allocated, this provides full inter-application protection: an application can only write to message areas for which it has been given permission in the form of a key.

2.3 Termini: direct access to the interface hardware

To avoid OS overhead at the sender, Hamlyn gives *each* sending application a private, hardware *send terminus*. This is a set of registers and a work queue that are mapped into the application's address space using normal OS virtual-memory protection mechanisms. The application can read and write its terminus to provoke message-sends without further OS intervention: no system call or interrupt occurs when sending a message.

Short messages are pushed from the host processor onto the terminus queue using regular store instructions: no additional protection checks are needed. We call this *direct I/O*. The terminus is then "prodded" to examine the queue if it wasn't already doing so, after which the application may proceed to other work. If the terminus queue is full, the processor spin-waits for a few microseconds until a message has been sent and there is space to insert its new request. Each terminus has its own queue so that each application may have several messages queued for transmitting—64 in our prototype.

Long messages may be sent by an asynchronous DMA mechanism built into the interface, which frees up the processor for other activity. To do this, a DMA request block is constructed and put onto the terminus work queue. A DMA request resembles a direct I/O request, except that in-line data are replaced by a pointer and length.

If an application could put an arbitrary memory address in a DMA request, then it could send any part of its node's memory across the network, which would clearly be a security violation. We prevent this by making the sender specify outgoing DMA messages in terms of a <message-area, offset, length> tuple, which the interface checks against a list of message areas stored in the sending terminus, translating addresses as the data are transmitted.

The Hamlyn interface scans round-robin around the termini with outstanding sends, transmitting a *packet* from each one in turn. This provides fairness amongst applications, even if some are sending long messages and others short ones. Indeed, the effect is to give short messages preferential access to the interconnect. This in turn increases the likelihood that short control requests will be sent with low latency.

This scheme could be embellished with priorities, although we didn't include this in our prototype: if sending termini have associated priorities, then the interface can service higher-priority termini before lower-priority ones.

2.4 Slots: sharing message areas

We wanted to make it possible to share receiver message areas amongst many senders (e.g., the collaborating processes of a distributed application). To do so, we introduced *slots*, which have two functions. A slot has a base-limit pair, identifying some or all of a message area, and it holds a protection key used to guard against misdirected or unauthorized messages. Multiple senders may share all or part of a given message area, and this permission can be revoked on a per-sender basis if each sender is given a separate slot. Figure 1 puts this all together.

Our packet protocol can address 2^{16} slots in an interface. The Hamlyn architecture allows the interface hardware to store slot information in its entirety, or to cache information for a subset of slots. Both the number of slots supported and how they are stored are implementation choices, hidden from the controlling software, so that the cost and complexity of a Hamlyn implementation can be adjusted to meet resource limits or price goals. For example, our prototype supported about a thousand slots.

2.5 Message reassembly and arrival notification

Packets of a segmented message may arrive at the receiving hardware in any order. The incoming message data are moved directly to the desired spot in the user

[2] This restriction could be lifted if the interconnect were to provide completely loss-free, in-order delivery.

[3] In the original Hamlyn proposal, the interface held an explicit page-address vector. In the prototype, this address conversion was folded into the bus converter that was part of the host processor's I/O system.

space via DMA: no staging through an OS buffer occurs, so no copying is needed. We refer to this design as a *zero-copy* protocol.

Once all the packets of a message have arrived, the receiving application should be told. We call this process *message-notification*. A unique aspect of Hamlyn is its flexible notification scheme, which has three parts:

- Hamlyn hardware automatically detects when all of the packets of a message, or a group of messages, have arrived—without any software intervention by the receiver.
- Once a message has arrived, the hardware writes an entry into a *notification queue*. (By default, there is one queue for each application process.)
- An interrupt is generated only if the receiving process is asleep, and even then, interrupts are coalesced when possible to minimize overheads.

The mechanism for generating a notification is based on an extended form of packet-counting.

Each sending terminus has a 32-bit *packet counter* that can be initialized by the sending application to some value Y_0.

At the receiver, each application has a pool of *notification assembly areas*. Each contains a 32-bit counter, space used by higher-level protocols for out-of-band data, and (optionally) additional protection information. A notification area can be recycled as soon as the receiver does a release operation on the associated buffer, so the number of areas needed is the number of messages potentially in flight at one time, not the total number sent. The counter is called a *notification assembly counter,* and it is always initialized to zero.

Each arriving packet carries an integer value in a field of its header called the delta field, and the index of a notification assembly area. In each outgoing packet

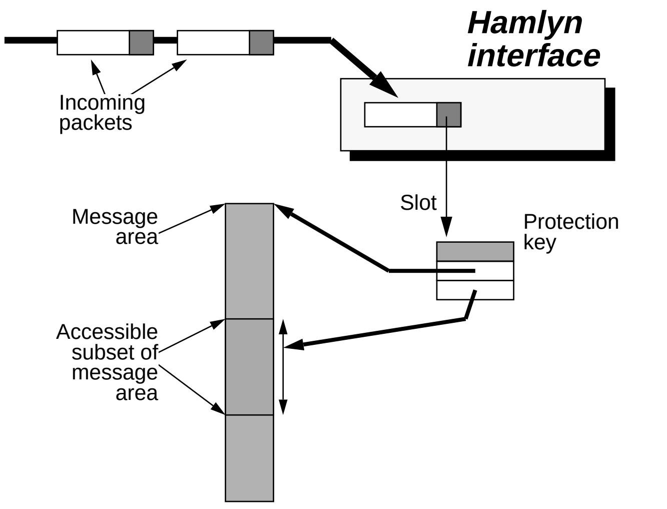

Figure 1: processing an incoming packet: checking the header to find where to put data.

except the last, the packet's delta field is set to 1 and the packet counter is decremented. In the last packet, the current value of the packet counter is put in the delta field. Each step leaves the sum of the packet counter and the deltas invariant, so that the sum of the delta fields of all the packets equals Y_0 modulo 2^{32}.

As the packets of a message arrive, their delta fields are summed into a notification assembly counter. A notification is generated when the counter again becomes zero, which can occur only if the sum of the incoming packets' delta fields is 0 modulo 2^{32}.

This deceptively simple mechanism provides:

Message reassembly. The problem here is that packets may arrive in a different order than they were sent. To handle this, the sender lets $Y_0 = 0$. If there is only one packet, its delta field will be zero and notification will occur on its arrival. If there is more than one packet, then all but the last packet have delta=1,while the last will have delta = $-(p-1)$, where p is the number of packets in the message. Notification will occur when all the packets have arrived, even if the packets arrive out of order.

Static group receive. A single notification can be delivered after a set of messages from a known list of senders has arrived (e.g., for a barrier). In this case, the *i*th sender is given an initial value Y_i such that the sum over all participating senders is 2^{32}. As the last-sent packet from each sender arrives, it cancels the packet-count from the other packets from the same sender, and contributes Y_i to the packet accumulator. The sum reaches 2^{32} and wraps around to zero exactly when all packets have arrived.

Single-sender scatter-gather. One sender may send several related messages and have the receiver be notified only when all of them have arrived. An example is: sending data to one area and a control message to another area. Because the interconnect may deliver messages out of order, a control message could overtake its associated data and arrive before it. This usage is just a special case of a static group receive in which each scatter-gather area is a separate message, but all come from the same sender.

Dynamic group receive. This is similar to static group receive except that the set of senders is initially unknown—in fact, no node may ever know the set's entire composition. An example occurs in database systems where one process delegates work to other processes. Here, a process that has received a value Y_i for its send counter can delegate work to (say) two other processes as long as their initial send-counter values, Y_{i1} and Y_{i2}, sum to Y_i. For correct operation, the senders must also partition the receiver's buffer area between them, and there must be a means by which the receiving application knows where to find the data.

2.6 Summary

The primary costs of the Hamlyn scheme are two-fold:

1. Message areas must be pinned in memory. We made a deliberate choice to do this: it buys considerable performance with a small amount of memory, which is becoming cheaper by the day. Besides, low-latency communication won't much help an application that has its active data paged out ….
2. Each message or packet must include some Hamlyn-specific header data. This bounds the smallest sensible packet size to a hundred bytes or so. We don't think that this is a major restriction: Hamlyn was originally designed for a packet-switching network with 128-byte packets, which seems a reasonable compromise for interconnects of this sort. For comparison, FibreChannel uses packet sizes in the 2KB range, and the Myricom network we are using for a prototype does not limit packet size.

In return for these, Hamlyn provides fully-protected, direct application access to the interconnect; message reassembly "for free," including the cases of multiple messages and multiple senders; protection against rogue messages and failing processors; and both direct and DMA sends. The next section discusses how all of this provides higher-level communication facilities while preserving performance and simplicity of the low-level hardware.

3 Rats—an interface library for Hamlyn

An interface library called Rats has been designed to support the rapid development of middleware code—such as MPI [Corbett95], Active Messages [vonEicken92], distributed database facilities—and other applications that are tuned for the Hamlyn architecture. The goal of Rats was to provide a thin layer of useful abstractions with which to manage Hamlyn resources, and to send and receive messages efficiently.

To use Hamlyn, a sender must manage space on the receiver into which the messages are delivered, space into which out-of-band data is deposited, and a notification queue. The way these resources are managed varies according to the communication scheme being used. For example, streaming a sequence of records requires that the receiver's notification queue and message area are not overrun, whereas updating a remote location with status information (e.g., the processor load average) may not need to generate notifications and message-area overrun is not a concern. Rats was designed to support a variety of protocols while isolating the programmer from the details of the hardware interface.

Rats was implemented in C++. It defines a collection of Hamlyn resource classes and member functions for operating on them. Rats code executes in the application's address space and invokes the OS only to acquire or release communication slots and termini—which occurs much less often than sending or receiving messages. Sending a message need never involve the OS; receiving one only does so to awaken a sleeping process that has blocked itself inside the OS.

Rats has two layers: one is concerned with the details of the Hamlyn hardware interface, the other with offering different communication abstractions. We discuss each of these in turn.

3.1 The Rats hardware interface level

The lower level of Rats provides a procedural, user library interface to the Hamlyn hardware and hardware-manipulated data structures. There are two main functions: sendmsg and get_notification_record.

The sendmsg function accepts a *ticket* and a buffer, then launches a message. The ticket specifies a destination node, slot, notification assembly area, protection key, offset, and range. The buffer's virtual address is converted to an offset from the beginning of the sender's message area. If the message is small, it is written directly into the interface using direct I/O. If it is large, a DMA control block is constructed and queued in the interface. The value returned by this function is a handle that can be used to determine the status of the send.

The get_notification_record function returns a pointer to the next notification entry, or zero if there isn't one.

Other lower-level functions communicate with the OS to allocate slots, allocate pinned memory, set protection keys, prepare information needed to wake a sleeping process upon message arrival, and so on.

3.2 The Rats higher-level protocols

The upper level of Rats consists of a Hamlyn interface manager and a collection of protocol modules that provide routines to send and receive data. The manager administers pinned memory using a malloc/free paradigm, and it allocates notification areas associated with slots owned by the application process.

The upper level of Rats supports the notion of a *protocol instance*—one instantiation of a protocol for a particular application process with a server and a client side. (These are treated a bit like UNIX[4] file descriptors.) Rats exports two functions that together resemble the UNIX select call: next_ready determines which protocol instance has data ready, if any, and poll blocks until there is data for one of the application's protocol instances, at which point it calls the instance's process_arrival function. (See below.)

Implementing the upper level in C++ provided name-space control (each module can have its own send, recv, close, etc. calls without name conflicts), in-line function expansion and abstract data types. All of the modules have similar interfaces, so that most of what a programmer knows about one applies to all.

4. UNIX is a trademark licensed exclusively by X/Open Corporation in the USA and other countries.

A novel feature of the protocol modules is how users establish connections. We wanted to avoid depending on a global distributed operating system, whose instances on the nodes would have to cooperate to establish connections. For most protocols, an application first creates a *server* (more strictly: server instance), which is usually the receiver of data. Each server supports a make_seed call that returns a *seed*, which is a data structure that encapsulates all the information necessary to send a message to that server—its site, slot, protection key, and any protocol-specific information. A seed can be communicated to another node using any available mechanism. A remote application calls the C++ constructor for the client side of the protocol, passing the seed as a parameter. This constructor sets up any necessary framework for normal operations, and contacts its server if necessary. Seeds are often communicated as *metadata* with a message: this is a small amount out-of-band data written separately from the main message at the recipient.

Except to awaken a sleeping process, message reception runs entirely at user level. The poll function calls get_notification_record in order to get information about the next incoming message or message group, then it identifies the associated server and calls a (C++ virtual) process_arrival function, which is provided by every server to handle the incoming data in a protocol-specific manner. Where needed, the recv operation returns a handle that can be used to find the message and to release the buffer.

Sending functions and the stream protocol's flush call return a handle so that the application can query the status of the send operation. Possible status values are error (and an error code), pending, copied, and sent.

Rats is modular and easily extensible: the tagged remote write protocol, described below, is about 2 pages of C++ code. There are currently seven supported protocols:

Remote Write. The seed used to create a client carries a ticket for the destination. The write call specifies a buffer, length, and destination offset. The only thing to happen at the receiving side is that data is written. In-order delivery is not guaranteed; scatter-gather is supported.

Tagged Remote Write. The seed used to create a client carries a ticket for the destination. The write call specifies a buffer, length, destination offset, and an integer tag. At the receiver, data is written to the destination message area, and the tag is included in a notification queue entry. Tags can be retrieved with a get_tag or get_specific_tag call. All messages of this type must fit in a single packet. (This and the Initial Request protocols are the only ones with this restriction.) In-order delivery is not guaranteed.

Datagram. This resembles UNIX's UDP, except it is reliable and the receiving application obtains a pointer to the message, rather than supplying one. When the application is finished with a buffer, it must explicitly release it. Behind the scenes, the release function sends a ticket back to the client so that the buffer can be reused. This protocol can be many-to-one by creating multiple seeds. As with UDP, in-order delivery is not guaranteed.

Simple datagram. This protocol differs from the datagram protocol above in that each message carries a seed for a reply; also, applications may send additional out-of-band data of their own, use Hamlyn's group receive mechanisms, and specify where the buffer space lies. The "simple" part of this protocol is that each instance manages a single receive buffer, rather than queues of several, which makes for a simple, efficient protocol.

The seed used to create a client is simply a ticket. Seeds can be *split*, which is the metaphor used to access group receive functions. The split_seed function creates a new seed and mutates the original in order to partition the destination buffer into non-overlapping ranges. It also makes the sum of delta values in the resulting seeds equal to the old delta value (modulo 2^{32}), so that an application receiving a split message will only be told when all the messages from all the senders have arrived.

A client protocol exports send, which accepts a buffer and a possibly null piece of metadata containing a reply seed. An application calls recv at the server, which waits for a message to arrive, and it can ask where data was deposited. The function reply_seed returns the reply seed sent in the message; metadata_for_self returns a pointer to a piece of metadata containing a ticket for its own instance. Thus, applications can ping-pong messages back and forth using the following sequence at each end, where srv is the server:

```
    srv.reply_seed().send(  sendbuffer,
                            sendbuffer + buffer_length,
                            srv.metadata_for_self()   )
```

The seed provides the send function in the call above to avoid explicitly constructing and discarding a client instance just for the reply.

Remote Read. Clients support remote_read and async_remote_read. A ready function indicates whether the async_remote_read has returned. Only one remote read can be in flight at any time for a single protocol instance.

Stream. This protocol provides a one-way, one-to-one, in-order connection from a sender to a receiver. Either a sender or a receiver can be created first. The make_seed function for a sender or receiver returns a seed for the opposite party.

The stream sender supports write_bytes, write_record, and flush. The stream receiver supports read_bytes and read_record; both return a <start-pointer, length> pair describing their result: no copying occurs.

Any stream can be written or read as either bytes or records, or these calls can be interleaved. A read_bytes call returns as many bytes as there are left in the current message, possibly crossing a record boundary. A read_record call returns one record (as sent by write_record), or the remainder of the current message if it contains no more record boundaries.

Large buffers are sent as-is; small ones may be coalesced by copying, in which case the returned handle's state will be the value copied and there is no guarantee of immediate transmission. Transmission can be ensured by a call to flush, which guarantees that all previously-copied data are transmitted. A release operation on a buffer frees its memory space and that of all buffers that arrived earlier.

Initial request response. This protocol is designed to be used for initial connection requests where there are no pre-allocated resources at the server. The client seed is an ASCII string, so it can be passed by some simple out-of-band process, such as being written in a file (similar to /etc/services), passed in an environment variable, put in a command-line argument., hard-coded into programs or distributed by a name server. The client does a send, the server gets it with a recv, it replies with sendreply, and the client receives that with recvreply. Since there are no pre-allocated resources, a transmission may arrive when the designated resource is busy and be lost, so the protocol uses time-out, retransmission, timestamps and sequence numbers to synthesize reliability. Applications must explicitly call release when they are done with the reply buffer.

4 Performance evaluation

To evaluate the design alternatives for Hamlyn, we undertook several different activities:

- a software emulator for the low-level hardware interface let us experiment with the Rats protocols before the hardware was ready;
- a prototype hardware implementation using the Myrinet interconnect [Boden95] let us try out the effects of the Hamlyn ideas in a near-full-speed prototype (this was built in cooperation with the University of California, Berkeley; see Figure 2);

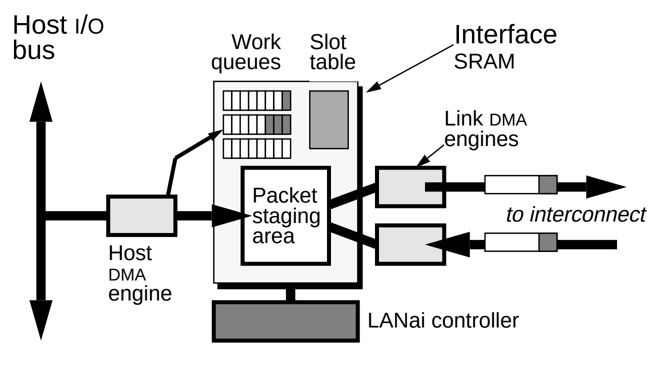

Figure 2: the structure of our prototype interface.

- ports of several large applications on top of the Rats software to learn how well they are supported by its interfaces.

Together, these allowed us to answer a range of questions. We continue to use them to explore which parts of the Hamlyn architecture bring value to applications, and which parts can be elided or delegated to software. This section presents an overview of some of these activities, and discusses performance data that we have obtained from them.

4.1 The software interface emulator

We built a software emulator of the Hamlyn hardware interface in order to help us develop the Rats interface library. We instrumented it to measure a number of small test programs. Because the emulator ran directly on PA-RISC hardware, we were able to determine precise instruction counts and timings for the Rats software layers.

Instruction counts we report here were obtained by disassembling compiler output. We measured the cycles-per-instruction value for an HP9000 Series J200 processor to be 1.27 when executing the sendmsg code, with the cache and TLB hot. Combining this with the processor clock speed of 120MHz gave us timings.

Table 1 shows instruction counts for a tagged remote write of *d* data bytes; Table 2 shows them for simple datagrams. A datagram can be sent with reply information (which needs 44 bytes of metadata), or without (which needs only 12). The table shows timings for a 16-byte message sent with and without reply data. Note that DMA costs are independent of message size at the Rats level.

The tables should be read as follows: the column labelled on-cp (on the critical path) gives the latency of the operation. The time the host processor is busy is given by the sum of on-cp and off-cp values: the latter represents work that can be overlapped with interconnect activity.

The tables indicate that Rats contributes about 1.6µs to the critical path of a tagged remote write (send + receive) of 16 data bytes, and 1.7µs for a 512-byte DMA. In the case of a simple datagram that includes the information needed for a quick reply, the values are 2.2µs (2.3µs for DMA)—a total of 4.4µs (4.6µs) of software costs for the round trip.

[Karamcheti94] reports instruction-counts (but no times) for Active Messages on a CM-5 (CMAM) that are roughly comparable with ours, although they were measured on a SPARC processor, ours are for PA-RISC. The CMAM finite-sequence, multi-packet delivery protocol seems to provide functionality somewhat below our simple datagram protocol, in that it doesn't support our group-receive operations, but does handle out-of-order packet delivery. [Karamcheti94] quotes 397 instructions to do a 16-word (64-byte) unidirectional send: our value is 215

Table 1: instruction counts to do a tagged remote write of *d* bytes in Rats. The notation *on-cp* means instructions on the critical path; *off-cp* instructions off it.

Tagged remote write		— Direct I/O —		— DMA —	
	function	on-cp	off-cp	on-cp	off-cp
sender	tagged_write::write	10	6	10	6
	sendmsg	69+3d/4	17	88	13
	total	*79+3d/4*	*23*	*98*	*19*
	16-byte msg total	**0.96µs**	**0.24µs**	**1.03µs**	**0.20µs**
receiver	poll	31	4	31	4
	process_arrival	16		16	
	get_tag	15	20	15	20
	total	*62*	*24*	*62*	*24*
	16-byte msg total	**0.66µs**	**0.24µs**	**0.66µs**	**0.25µs**
both	*total*	*141+3d/4*	*47*	*160*	*43*
	16-byte msg total	**1.63µs**	**0.48µs**	**1.69µs**	**0.45µs**

Table 2: Rats instruction counts for sending and receiving a simple datagram, with *m* words of metadata and *d* words of data. Instructions labelled *on-cp* are on the critical path, ones labelled *off-cp* are not. The case labelled 12+16 msg is a 16-byte message with 12 bytes of metadata; 44+16 has a full return ticket in its 44 bytes of metadata.

Simple datagram		— Direct I/O —		— DMA —	
	function	on-cp	off-cp	on-cp	off-cp
sender	send	10		10	
	sendmsg	67+3(m+d)/4	10	88+3m/4	7
	send total	*77+3(m+d)/4*	*10*	*98+3m/4*	*7*
	12+16 msg	**1.04µs**	**0.11µs**	**1.13µs**	**0.07µs**
	44+16 msg	**1.29µs**	**0.11µs**	**1.39µs**	**0.07µs**
receiver	poll	28	4	28	4
	process_arrival	39		39	
	recv	14	22	14	22
	receive total	*81*	*26*	*81*	*26*
	receive total	**0.86µs**	**0.28µs**	**0.86µs**	**0.28µs**
both	*grand total*	*158+3(m+d)/4*	*36*	*179+3m/4*	*33*
	12+16 msg	**1.90µs**	**0.39µs**	**1.99µs**	**0.35µs**
	44+16 msg	**2.15µs**	**0.11µs**	**2.25µs**	**0.35µs**

instructions using direct I/O, 188 using DMA. It is not clear from [Karamcheti94] how this count scales if the packet size increases to 32 bytes: our increases by 12 instructions in the direct I/O case, and doesn't change using DMA.

This comparison shows that Hamlyn is providing essentially the same performance as a well-tuned Active Message implementation, with the addition of full inter-application protection and no receiver overruns.

[Karamcheti94] argues that the underlying network interconnect should provide in-order delivery, deadlock freedom, and fault-tolerant packet transmission. We conclude instead that Hamlyn can provide the first two of these on top of an arbitrary interconnection network that provides the last, allowing the interconnect designers to optimize for performance rather than high-level protocol support: [Davis92] argues that an adaptive-routing network can achieve roughly double the throughput of a non-adaptive one.

4.2 The hardware prototype

We are building a hardware prototype based on the Myrinet interconnect, in cooperation with the University of California at Berkeley. The Myrinet switch is a wormhole router configured as a non-blocking 8×8 crossbar. It provides 80MB/s of bandwidth per port, simultaneously in each direction, with about 0.5µs of switching latency. Myrinet's LANai version 3 network interface chip contains three DMA units (inbound from the switch, outbound to the switch and to or from host memory), a 32-bit programmable processor, and on-card static memory (SRAM).[5]

We used the LANai chip and SRAM to prototype the Hamlyn design by microcoding the LANai to emulate a Hamlyn interface. In order to determine timings, we built a cycle-counting instruction-set simulator that included detailed models of the interface card, the host bus, and SRAM contention.

We consider the following basic cases: direct send of a short, single-packet message, both with and without updating a word in host memory to indicate that the message has been sent; a DMA send of a long message; fast inbound receive of a small packet with no notification; and reception of an inbound packet with notification to a busy-waiting host process.

In what follows, we use an HP9000 Series J200 as a host: it runs at 120MHZ, and has a GSC+ I/O bus that runs at 40MHZ connected to a 120MHZ processor bus. We assume there is no additional resource contention for buses beyond that intrinsic to the interface card design; that both the host and LANai processors are in busy-wait loops at the appropriate points (i.e., there isn't anything else going on in the system); Hamlyn protection keys are 32 bits long; and that no errors occur.

The numbers we report are in both 25ns LANai clock ticks and microseconds. Tables 3 and 4 summarize our results for single-packet messages, Table 5 for a large message transfer using 512-byte packets.

If we combine these numbers with those for the Rats software layer, we see total times of 8.8µs for an application-to-application send + receive of a 16-byte tagged data message, and 42.2µs for a round-trip that moves a 512-byte datagram and returns a 16-byte acknowledgment. Even this simple data-transfer protocol achieves a bandwidth of 12.1MB/s, but pipelining sends or sending larger messages makes it easy to achieve much higher numbers: 27.0MB/s with pipelined 512-byte messages, 49.2MB/s with 4KB

[5]. Performance for the older LANai 2 interface is 2–3 times worse than for LANai 3 because the older processor has only a 16-bit CPU and is less able to overlap data-transfer and control operations.

Table 3: details of Myrinet LANai 3 interface card costs when processing fast-path Hamlyn messages with *d* bytes of payload data + metadata. Numbers are given in both 25ns LANai 3 cycle counts and microseconds. Items in [square brackets] are off the critical path. "Total" values are critical-path values for 16-byte messages.

Direct IO of small message		**LANai cycles**	**time (µs)**
sender	send packet	81 + d/2	2.03 + 0.013d µs
	notify host	[102]	[2.55µs]
	total	*89*	*2.23µs*
switch		20	0.5µs
receiver	link to interface	71 + d/4	1.78 + 0.006d µs
	interface to host[a]	14d/16	0.022d µs
	notify host	[89]	[2.23µs]
	total (no recv notify)	*89*	*2.23µs*
	total (with recv notify)	*178*	*4.45µs*
all	**total (no recv notify)**	**198**	**4.96µs**
	total (with recv notify)	**287**	**7.18µs**

a. This is the GSC+ bus time; it becomes 18d/32 for 32-byte payloads and larger.

Table 4: details of Myrinet LANai 3 interface card costs when processing Hamlyn messages from the host RAM by DMA with *d* bytes of payload data + metadata. Timings are given in both 25ns LANai 3 cycle counts and microseconds. Items in [square brackets] are off the critical path. "Total" values are for a 512-byte message and include host notification of message arrival. These values assume no cut-through at the LANai card on sending.

DMA of large packet		**LANai cycles**	**time (µs)**
sender	start data to card	106	2.65
	load payload[a]	18d/32	0.38 + 0.014d
	send packet[b]	21 + d/2	0.53 + 0.013d
	notify host	[102]	[2.55]
	512-byte total	*671*	*16.78*
switch		20	0.5
receiver	link to interface[c]	71 + min(71, d/4)	1.78+ min(1.78, 0.006d)
	interface to host	59 + 18d/32	1.82 + 0.014d
	notify host	89	2.23
	512-byte total	*578*	*14.45*
all	**512-byte total**	**1269**	**31.73**

a. This is overlapped with the LANai processor loading the packet-header: send-packet can commence as soon as both header and data are loaded. The payload-loading term dominates for payloads larger than ~60 bytes.
b. The d/2 term is the cost to move data across the 80MB/s Myrinet link: 6.4µs for 512 bytes.
c. The DMA engines cycle-steal from the LANai processor's access to the SRAM. For d>=284, the effect is that the processor runs at half speed.

transfers, and 67.8MB/s with 1MB transfers. **This shows that we can achieve both very low latencies and very high bandwidths with the same architecture.**

The cut-over point between direct I/O and DMA is a function of the specific interface hardware and low-level bus protocols. For the design we describe here, we find

Table 5: details of Myrinet LANai 3 interface card costs when processing a large multi-packet message with *d* bytes per packet of payload data + metadata, and *Nd* total bytes. Timings are given in both 25ns LANai 3 cycle counts and microseconds. Items in [square brackets] are off the critical path. "Total" values are critical-path times for a 1MB message sent with 512-byte packets (N=2048). These values assume no cut-through at the LANai card on sending.

DMA of large message		**LANai cycles**	**time (µs)**
sender	start data to card	106	2.65
	1st load payload	18d/32	0.38 + 0.014d
	(N-2) load/send	(N–2)×18d/32	0.014 (N–2)d
	last send packet	21 + d/2	0.53 + 0.013d
	notify host	[102]	[2.55]
	1MB/512-byte total	*589.9K*	*14.75ms*
switch		20	0.5
receiver	link to interface	2×71	3.56
	1st interface to host	59 + 18d/32	1.82 + 0.014d
	(N–1) packets to host	(N–1)×18d/32	0.56(N–1)d
	notify host	89	2.23
	1MB/512-byte total	*590.1k*	*14.75ms*
all	**1MB/512-byte total**	**590.1k**	**14.75ms**

DMA requests break even for a data transfer of only 30 bytes or more.

Projecting what could happen if we were to embed the Hamlyn design in a set of state machines, but keep the same basic architecture as the Myricom card (i.e., one DMA engine between host RAM and on-board SRAM, and two DMA engines between the SRAM and the external link), we believe that we could cut the times down to close to the sum of:

- a GSC+ bus cycle to move the "start transmission" request to the interface;
- the time to assemble and transmit the header;
- the time to traverse the switch;
- the time to read the header at the receiving interface;
- and the time to pipeline the data through the two I/O buses, the interfaces, and the switch chip.

Putting all of this together gives us a best-possible hardware time of about 46+18*d*/32 LANai cycles, or 1.15 + 0.014*d* µs—this system would be GSC+ limited for large transmissions. The best-case bandwidth achieved under these assumptions is 71.1MB/s; perhaps even more interesting is that 64-byte packets achieve more than half of this (41.3MB/s) while 512-byte packets get over 90% of it (65.2MB/s). Since even the LANai design is largely bus-limited at large transfer sizes, the main advantage of moving to the state-machine implementation is reduced latency. Assuming the same Rats layer, we reduce the application-to-application 16-byte send + receive time to only 3.0µs, split roughly evenly between hardware and software.

5 Related work

There has been a lot of work in the field of interface design for high-speed interconnects. Here is a short summary of what we consider most relevant to the Hamlyn design:

- OS/360 provided variants of the put and get file-system calls that avoided data copying by having the OS specify the location of the buffer to use, rather than the application [Clark66, Belady81]. This is similar to the Rats library's supplying data pointers, rather than copying data to a network buffer in the OS, as UNIX applications often do.
- SHRIMP [Blumrich94] is a low-latency remote-memory-access scheme. It uses local virtual-memory protection algorithms to control access to a portion of I/O space that is mapped onto similar memory at other machines. As a result, it doesn't have the notification schemes we provide, nor can it cope with out-of-order packet delivery.
- *Active messages* [vonEicken92, vonEicken94, Martin94] provide a set of arrival semantics for packets by including the address of a function to call in each message. The function is typically invoked in a restrictive environment, with no protection barriers around it.

  We believe that a variant of Active Messages is a fine thing to layer on Hamlyn, and plan to do so. However, much of Hamlyn's value is its hardware support for message-reassembly, inter-application protection, and smart notifications; the traditional active message model does none of these, leaving them to be added on top of the underlying mechanism.[6]

  On the other hand, we believe that *Active Packets* would **not** be a good idea: the context-switches would prove too expensive. Indeed, the current trend in processor design seems to be towards ever-larger amounts of processor state, which will make this more costly still. Hamlyn addresses this concern by automating packet-reassembly in hardware.
- *Cranium* [McKenzie94] is almost a proper subset of Hamlyn, and shares many of its design goals and approaches. (We note that it was designed two years after Hamlyn.) The main differences between the two is that the Cranium design makes a few trade-offs in favor of simplicity at the cost of less flexible protection, and it allows receiver-resource overruns in certain circumstances. For example, Cranium supports an "append to message-area" operation for incoming packets, which can provoke the kind of receiver buffer overrun we wanted to avoid. (We agree that it is an attractive function for certain limited uses, and considered adding it to Hamlyn. However, we decided against it because of the dangers it represented, and its limited applicability—for example, messages have to fit in a single packet.)
- Various shared-memory models, including *Alewife* [Kranz93] and *Typhoon* [Reinhardt94] provide a very different programming paradigm, and they require explicit processor support. Hamlyn does not—it can fit on a standard I/O bus.

[6] We understand that Alan Mainwaring at UC Berkeley is working on adding protection to Active Messages. Frans Kaashoek reported on relaxing locking constraints in an Active Message implementation at the 6th SIGOPS European Workshop, Dagstuhl, Sept. 1994.

6 Summary

Based on observations about the nature of modern multicomputers interconnects and the nature of software for such machines, the Hamlyn interface provides message passing with a combined hardware and software cost of a few microseconds, while providing full inter-application protection and resilience in the face of nodes that do not fail gracefully. This paper showed how we achieve this; including a description of trade-offs that we can make (and have made) in the design, together with detailed performance data.

The Hamlyn interface architecture is optimized for closely-coupled multicomputer systems. It gives better performance than loosely-coupled clusters of autonomous computers, and much better fault tolerance than shared-memory systems. It also provides inter-application protection at very low cost. All these needs must be addressed if large-scale parallel machines are to make a significant impact on general-purpose computing. The Hamlyn architecture is an important step in that direction.

The Hamlyn interface provides necessary and—we believe—sufficient, mechanisms to let application software manage most aspects of message communication. Thanks to this, it offers outstanding performance while retaining full protection against erroneous or malicious applications. This seems to be a significant advance in the state of the art.

References

[Belady81] L. A. Belady and R. P. Parmelee, and C. A. Scalzi. The IBM history of memory management technology. *IBM Journal of Research and Development*, **25**(5):491–503, September 1981.

[Blumrich94] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felton, and Jonathan Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. *Proceedings of 21st International Symposium on Computer Architecture* (Chicago, IL). Published as *Computer Architecture News*, **22**(2):142–53. ACM/IEEE, 18–21 April 1994.

[Boden95] Nannette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles E. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: a

Gigabit-per-second local area network. *IEEE Micro*, pages 29–36, February 1995.

[Clark66] W. A. Clark. The functional structure of OS / 360: part III, data management. *IBM Systems Journal*, **5**(1):30–51, 1966.

[Corbett95] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO parallel I / O interface. *3rd Annual Workshop on I/O in Parallel and Distributed Systems (IOPADS'95)* (Santa Barbara, CA), pages 1–15, 25th April 1995.

[Davis92] Al Davis. Mayfly: a general-purpose, scalable, parallel processing architecture. *Lisp and Symbolic Computation* **5**(1–2):7–48, May 1992.

[Karamcheti94] Vijay Karamcheti and Andrew A. Chien. Software overhead in messaging layers: where does the time go? *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA). Association for Computing Machinery, 5–7 October 1994.

[Kranz93] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating message-passing and shared-memory: early experience. *Proceedings of 4th ACM Annual Symposium on Principles and Practice of Parallel Programming*, May 1993.

[McKenzie94] Neil R. McKenzie, Kevin Bolding, Carl Ebeling, and Lawrence Snyder. Cranium: an interface for message passing on adaptive routing networks. *Proceedings of Parallel Computer Routing and Communication Workshop* (Seattle, WA), pages 266–80, May 1994.

[Reinhardt94] Steven K. Reinhardt, James R. Larus, and David A. Wood. Typhoon and Tempest: user-level shared memory. *Proceedings of 21st International Symposium on Computer Architecture* (Chicago, IL). Published as *Computer Architecture News*, **22**(2):325–36. ACM / IEEE, 18–21 April 1994.

[Saltzer84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, **2**(4):277–88, November 1984.

[vonEicken92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schuser. Active messages: a mechanism for integrated communication and computation. *Proceedings of 19th International Symposium on Computer Architecture* (Gold Coast, Australia), pages 256–66, 19–21 May 1992.

[Wilkes92] John Wilkes. *Hamlyn—an interface for sender-based communications*. Technical report HPL–OSR–92–13. Operating Systems Research Department, Hewlett-Packard Laboratories, Palo Alto, CA, 30 November 1992.

The authors can be contacted as follows: greg_buzzard@imsystems.com, and {jacobson, marovich, wilkes}@hpl.hp.com. Please address correspondence concerning the paper itself to David Jacobson.