# A Performance Study of Isotach Version 1.0

Robert G. Bartholet

Technical Report CS-99-13
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
Email: rgb2u@virginia.edu

April 29, 1999

# Contents

# List of Figures

# 1

# Introduction

The Isotach version 1.0 prototype implements isotach logical time on a cluster of Intel-based personal computers linked with a Myrinet local area network. A performance study was conducted on this unoptimized version of the prototype with the following goals:

- determine Isotach's performance characteristics
- look for problems in the software and hardware design
- discover ways to optimize the Isotach implementation

Tests were conducted using custom developed programs and the Communications Analysis and Simulation Tool (CAST), a network benchmarking tool developed by the U.S. Navy and used at their request. In order to give us a baseline, we also performed tests on Fast Messages 1.1, a messaging layer known to provide good performance.

CAST measurements were not as accurate as our custom developed tests due to software overhead in the tool. We explain the reasons for this overhead, and show that CAST accuracy is degraded on networks with submillisecond latencies.

Our performance study measured throughput, latency, server performance under contention, effects of isochron size, and effects of token traffic on non-isotach traffic. The all-software Isotach version 1.0 prototype exceeded many of its performance goals. We expect performance to improve even more when custom-built hardware components replace software. Limitations in the design of the Isotach prototype were discovered, and recommendations were made to improve performance in future Isotach systems.

# 2

# Background

This chapter provides a brief overview of *isotach networks*, our Isotach prototype, and the components that went into building the prototype. It is assumed the reader is already familiar with isotach networks, and the technologies included in the prototype. The interested reader should turn to [15, 14, 1, 13] for more detailed information.

## 2.1 Isotach Networks

Isotach networks [20, 15] are a new class of network designed to reduce synchronization costs in parallel computations. They provide strong guarantees about the order in which messages are received. To obtain these guarantees, normally a process or system would use delays or locks that are very expensive to implement. Isotach networks can provide these guarantees with a simple, yet scalable, implementation.

### 2.1.1 Isotach Logical Time

The key component of an isotach network is *isotach logical time*, an extension of the logical time Lamport described in his classic paper [6]. Lamport proposed a simple distributed algorithm for assigning logical times that captures the "happens before" relation, otherwise referred to as *potential causality*. Isotach logical times extend this relation by constraining the times to be consistent with an invariant relating *logical distance* to logical time.

Isotach logical times are lexicographically ordered n-tuples of non-negative integers. The first component is the *pulse*, and the interpretation of the remaining components can vary. In our prototype, logical time is a 3-tuple

of the form (*pulse*, *pid*($m$), *rank*($m$)). The isotach invariant requires that a message sent at time ($i$, $j$, $k$) will be received at time ($i + \delta$, $j$, $k$), where $\delta$ is the logical distance between the sender and receiver. One of several measures can be used for logical distance; our prototype uses the number of switches through which a message is routed.

The isotach invariant provides a powerful mechanism for coordinating access to shared data. Given the isotach invariant, a process that knows the logical distance for messages that it sends can control the logical times at which its messages are received. This solves race conditions in many types of applications. One important feature of isotach networks is the ease in which a process can send atomic multicasts, called isochrons. An isochron is a set of operations that a process wants executed within the same logical time pulse. These operations can be both local and remote, and appear to be executed at the same time.

### 2.1.2 Isotach Network Implementation

The *isonet* algorithm [15] can be used to implement isotach logical time in a network of arbitrary topology. In this network, each node is either a host or a switch, and all communication is conducted over FIFO links. For the purposes of this discussion, we will assume a failure-free model, although significant progress has been made recently in the area of fault-tolerant isotach networks [21].

Each host in the network has a *switch interface unit* (SIU) between the host's network access (normally a network interface card) and the physical network. Each SIU and switch maintains a logical clock that keeps the pulse component of logical time. SIUs and switches remain loosely synchronized by exchanging *tokens*, which separate pulses of logical time. The sending of a token marks the end of one pulse of logical time, and the beginning of another. Hence, switches and SIUs increment their logical time clocks when they send tokens. When a switch receives token $i$ from all of its neighbors, it sends token $i + 1$ on all outputs. All messages are time-stamped by the sending SIU with the desired logical execution time. This timestamp must be at least *current logical time* + $d$, where $d$ is the logical distance to the receiver of the message. During a pulse, switches and SIUs always handle the available message with the earliest logical timestamp.

## 2.2 Isotach Prototype

The isonet algorithm will maintain isotach logical time. In practice, this algorithm is difficult to implement without making significant changes to the network hardware. The Isotach prototype [14] modifies the isonet algorithm, while still maintaining the isotach invariant. The new algorithm allows messages to traverse the network as quickly as possible. They are then buffered at the receiving node until the correct logical time is reached for execution. The prototype guarantees that once a token ending logical time pulse $t$ is received at host $n$, there are no more messages time-stamped with logical time $t$ that haven't been received at host $n$.

One way to maintain logical time and exchange tokens is to build this functionality into a network switch. This was not a realistic goal for our prototype, so we designed a *token manager* (TM) that sits on one output port of every network switch. Version 1 of the prototype is completely implemented in software on the host and network interface card. Version 2 will be realized shortly, and will have custom-fabricated hardware for the SIUs and TMs.

The prototype was implemented using a Myrinet physical layer for the network. We use a modified messaging layer (FM), which now contains much of the isotach functionality. The hosts have Pentium and Pentium Pro processors ranging in speed from 166 to 200 Mhz, and are running Linux kernel 2.0.30.

### 2.2.1 Switch Interface Unit

The SIU sits between the host and the physical network. In version 1, the SIU is implemented on the Myrinet network interface. In version 2, it will be a custom hardware device located between the network interface and the Myrinet switch. Two key pieces of information that the SIU maintains are the current logical time, and the logical distance to every other host in the network. The SIU maintains logical time by exchanging tokens with its adjacent TM. Logical distance is easily computed for each message in our prototype by counting the number of routing flits in the message header. When the SIU receives an isotach message that the host wants sent onto the network, it timestamps the message with the logical time of execution, at least current logical time + logical distance to receiver. The network guarantees that the message will arrive before the token signifying the message's logical time of execution.

Additionally, the SIU notifies the host of significant logical times by

sending an *end of pulse* (EOP) marker to the host. A logical time is significant if the host has some operation that needs to be executed within that logical time pulse. The SIU is aware of significant logical times because it sees all messages coming from the network to the host process, and extracts from these messages the logical time of execution. In the other direction when a host sends an isochron, it needs to know the logical time of execution of the isochron in the event there are any local components to be executed. The SIU notifies the host of isochron execution times by sending an *end of isochron* (EOI) marker to the host. Eventually the EOI marker will be followed by an EOP marker when the isochron's logical execution time passes.

Because the SIU sits between the host and the switch, it sees all message traffic, isotach and non-isotach. The SIU can immediately tell from a message header whether or not the message is an isotach message. Non-isotach traffic is forwarded immediately in the direction it was traveling.

### 2.2.2 Token Manager

A TM is attached to one output port of every switch in our prototype network. Currently, the TM is implemented in software on the network interface. In version 2, the TM will be a custom fabricated hardware device. The TM exchanges tokens with all neighbors. A neighbor is defined as the hosts attached to the TM's switch and TMs attached to adjacent switches. When a TM receives the $i$th token from all its neighbors, it sends out the $i + 1$ token to each neighbor. This maintains a consistent progression of logical time.

### 2.2.3 Shared Memory Manager

The *shared memory manager* (SMM) is a host-level application library [1] that performs high level isotach functionality. It receives isochrons from an isotach application, and splits them into local and remote components. The local component is placed in the *hit buffer* until an EOP marker is received from the SIU corresponding to the component's logical time of execution. The remote component is shipped to the SIU where it is time-stamped and sent onto the network.

Operations in our isotach prototype are referred to as *srefs*. While awaiting execution, srefs are stored in *pulse buckets*, each bucket corresponding

[1] Eventually, we would like to implement the SMM in hardware as we have done with the SIU and TM

to a different logical time pulse. When the SMM receives an EOP marker from the SIU, it executes the srefs contained in the bucket that matches the time in the EOP marker. Additionally, the EOP marker contains a *sort vector* that has the correct execution order of srefs in the bucket. The srefs are sorted in accordance with the three components of logical time. This gives a *total ordering* to all operations in our isotach network. When the SMM executes the operations contained in a bucket, it may also handle an EOI marker in the bucket. This signifies that an isochron with remote components was sent time-stamped with this pulse. The hit buffer is checked at this time to execute any local srefs in this remote isochron. Additionally, the SMM can execute any srefs that were issued in purely local isochrons, up to the next pending isochron with remote components, while still maintaining sequential consistency.

When an isotach application is building an isochron, it sends each operation individually to the SMM. The operations are buffered in the SMM until the application signals that it has sent the last operation of the isochron. At this time the buffered isochron is shipped to the SIU. This additional buffering in the SMM is significant because it impacts performance as discussed in chapter 5.

As its name implies, the SMM manages an isotach shared memory system. It provides strong consistency, with operations to read, write, schedule, and assign values to shared memory variables. The copyset of shared memory variables is maintained in the SMM. An API is exported to an isotach application allowing it to read and write shared variables transparently while maintaining consistency.

The SMM also exports an API allowing isotach applications to pass messages. Upon execution, messages are passed to user-defined handlers. The prototype has no notification mechanism for message arrival; users must poll in order to extract messages. Infrequent polling may delay a messages' execution in real time, but the message is always guaranteed by isotach to maintain total/causal ordering and sequential consistency.

## 2.3 Myrinet

Myrinet [1] is a local area network with an extremely high data rate developed from technology used for massively parallel processors. A Myrinet link is a full-duplex pair of 1.28 Gbit/s point-to-point channels. The Myrinet physical layer provides a very low error rate combined with high bandwidth and low latency. Flow control is provided on every link. The switches use

cut-through routing. Myrinet networks can be of an arbitrary topology, and performance is very scalable because they don't use a common bus like many data-link layer protocols [8, 2, 16].

The distinguishing features of the Myrinet network interface are its general purpose microprocessor, called a LANai, and up to 1 MB of high-performance SRAM. The network interface is distributed with source code and a C programming language compiler, allowing any customer to tailor the network interface to its needs. This combination of interface, LANai, SRAM, source code, and compiler has recently facilitated a push in the research of high performance messaging layers.

The Myrinet network interface makes use of send and receive DMA engines to move data to and from the network. The host is given access to the SRAM through programmed I/O, and the LANai can move data from the SRAM to the host memory and back with a third DMA engine. The LANai only operates at 30-40 Mhz, so the use of this third DMA engine is critical to obtaining good performance.

## 2.4 Fast Messages

Fast Messages (FM) 1.1 [13] is a high performance user-level messaging layer that runs over Myrinet. Like most recent experimental protocols, FM keeps the operating system out of the critical path by giving the user process access to the network. FM is a good match for isotach networks because it provides reliable FIFO delivery of messages. FM uses sender-based flow control that forces hosts to allocate buffer space for every other host on the network, thereby guaranteeing that buffers cannot be overrun. FM assumes the network is reliable, a good assumption with Myrinet.

FM uses polling as the means of delivering messages to the user. However, when the user polls, all available messages are extracted from the network. In many applications, this type of nondeterministic polling forces the user application to implement its own buffering layer, thereby not allowing the application to realize the full bandwidth potential of FM. Upon message delivery, FM calls a handler specified by the sender. Handlers shouldn't do much more than pass the message to the user application, as they are in the critical path on the receive side.

FM 1.1 achieves a one way latency of about 11 $\mu$s for small messages, and a bandwidth of over 250 Mbits/s for large messages. FM's performance numbers are competitive with other messaging layers for the small message sizes used by most applications.

We modified FM 1.1 for our prototype. First, we ported FM to Linux, an effort that focused on byte endian issues. Additionally, function calls were added that supported isotach functionality. A key point, however, is that the code path for messages using the FM API was not modified; only isotach messages are affected by the modifications.

## 2.5 Chapter Summary

The Isotach prototype was implemented by modifying Fast Messages 1.1 and running it on hosts linked with a Myrinet network. Although the isotach invariant is simple in concept, it is not necessarily simple in implementation. This chapter gave an appreciation for the complexity involved in one implementation of an isotach network.

# 3

# Communications Analysis and Simulation Tool

The Communications Analysis and Simulation Tool (CAST) is a program to help evaluate the performance of a network [11]. The tool was developed by the Naval Surface Warfare Center, Dahlgren Division (NSWCDD). It is an extension of ttcp, a transport control protocol (TCP) benchmark used to measure throughput. NSWCDD extended ttcp to measure latencies and give the experimenter much more flexibility in setting up experiments [3]. Isotach is currently under consideration by NSWCDD for use in future Department of Defense systems. NSWCDD requested that we run CAST over our Isotach prototype so we could give them performance results using their tool. This chapter describes CAST and the work that went into implementing Isotach and FM modules for CAST. Results of performance tests on our Isotach prototype using CAST will be presented in chapter 5.

## 3.1 Description

CAST is a very flexible network performance tool. The user gives command line options to customize a performance experiment. The most typical command-line options determine the architecture to use (sockets, ATM), protocol (TCP, UDP), the name of the remote host, and message size. Other settings include buffer alignment, amount of data or number of packets to send, thread priorities, and output reports. CAST is also useful as a background load while running other experiments. A typical CAST measurement consists of three phases. The first phase determines the offset between the client and server clocks. The second phase measures the one-way and round-trip latencies between the client and server. The third phase measures the

throughput of the data channel.

CAST works well over popular local area network technologies with typical latencies measured in milliseconds. However, in high-performance system area networks where latencies are measured in tens of microseconds, any overhead that is included in the measurement has a much greater impact on the results. We did not notice CAST overhead when we initially tested the tool between hosts on our department LAN where latencies are measured in milliseconds. But when we moved CAST to our Isotach cluster (with sub-millisecond latencies), the overhead's effect on the test results became apparent.

### 3.1.1 Offset Measurement

In addition to round-trip latencies, CAST reports one-way latencies between a client and a server. Typically to obtain one-way latencies, either the host clocks must be synchronized, or the difference between the client and server clocks must be known. There are many network clock synchronization algorithms [9]. The Network Time Protocol (NTP) [10] is a standard in common use, but it only keeps clocks synchronized to within a few milliseconds. Any algorithm that can keep clocks synchronized at a level of granularity useful to system area network performance tests ($\sim$1 $\mu$s) would put enough load on the host processors and the network (in the form of control traffic) to affect the results. CAST takes another approach and measures the offset between the client and server clocks.

CAST uses a round-trip message and the following formula for the offset computation:

$$\mathcal{O}ffset = ((t_b - t_a) - (t_d - t_c)) \div 2 \tag{3.1}$$

where
$t_a$ is time message is sent by client according to client clock
$t_b$ is time message is received by server according to server clock
$t_c$ is time message is returned by server according to server clock
$t_d$ is time message is received by client according to client clock

This formula is accurate only if the one-way latencies are equal on the round-trip used to calculate the offset. Therefore, the offset can only be considered an approximation of the difference between the client and server clocks, because it can not be verified that one-way latencies are exactly equal. In order to make this approximation more accurate, CAST records the timestamps on 1000 round-trips, and uses the round-trip with the lowest delay to obtain

the offset measurement. The delay is computed as:

$$\mathcal{D}elay = ((t_d - t_a) - (t_c - t_b)) \div 2 \tag{3.2}$$

CAST is assuming that the round-trip with the lowest average one-way delay will have one-way latencies that are close to equal. As will be discussed in section 3.3, we found that while this assumption is strong enough for slower legacy LANs like Ethernet, it is not valid for high-performance messaging layers that measure latencies in microseconds.

Computer clocks will drift apart in time. The CAST documentation claims that if the clock drift is less than 300 parts per million, then the offset is accurate for the duration of the test. At a rate of 300 parts per million, clocks will drift apart by 1 $\mu$s every 3.33 milliseconds. As will be discussed in section 3.3, this is too much clock drift for use in our Isotach measurements.

If the CAST user decides to run more than one test iteration in an experiment, the clock offset is recomputed for each iteration to account for clock drift. CAST also gives the user an option to compute the clock drift to help determine the validity of the offset.

### 3.1.2 Latency Measurement

In phase II of a CAST measurement, 100 packets (this number can be changed on the command line) are sent round-trip, and four timestamps are recorded for each packet. These timestamps are taken at the same points as they were in the offset measurement in equation 3.1. From these timestamps, the average round-trip latency (minus the turn-around time) is calculated:

$$\mathcal{L}atency = (t_d - t_a) - (t_c - t_b) \tag{3.3}$$

Additionally, the average one-way latencies in each direction are calculated using the offset:

$$\mathcal{S}end\mathcal{L}atency = t_b - t_a - \mathcal{O}ffset \tag{3.4}$$

$$\mathcal{R}eturn\mathcal{L}atency = t_d - t_c + \mathcal{O}ffset \tag{3.5}$$

For each of these figures, CAST reports the average, minimum, maximum, standard deviation, and variance.

### 3.1.3 Throughput Measurement

Phase III of a CAST experiment is the throughput measurement. Throughput is measured at both the client and server interfaces. CAST computes

*interface throughput* as follows:

$$\mathcal{T}hroughput = \mathcal{S} \div (t_2 - t_1) \tag{3.6}$$

where
$\mathcal{S}$ is amount of data sent over the channel
$t_1$ is time first data byte leaves client (sender throughput)or arrives at server (receiver throughput)
$t_2$ is time last data byte leaves client or arrives at server

Additionally, CAST uses the offset to measure an end-to-end throughput:

$$\mathcal{T}hroughput = \mathcal{S} \div (t_2 - t_1 + \mathcal{O}ffset) \tag{3.7}$$

where
$t_1$ is time first data byte leaves client according to client clock
$t_2$ is time last data byte arrives at server according to server clock

End-to-end throughput is distinct from the interface throughput because it includes the time the data spends on the wire. In a throughput test that measures a packet burst, the end-to-end throughput is an interesting figure because the network latency can potentially dominate the measurement. However, as $\mathcal{S}$ increases, the amount of time it takes to conduct the test increases, and the latency has a much smaller impact on the result. In other words, the end-to-end throughput converges toward the minimum interface throughput as $\mathcal{S}$ increases.

## 3.2 Implementation of Isotach Module

CAST is designed with a modular structure that simplifies the implementation of new communication protocols and architectures. The CAST main module defines generic functions and data structures. A network module is required for each specific type of network and/or protocol the experimenter wishes to use (e.g. isotach). CAST requires that the network modules implement specific functions (setup, close, send, receive, etc.) and data structures. At runtime, the addresses of these specific functions and data structures are linked to the main module's generic functions and structures. The main module can then open/close a channel and send or receive transparently using its generic function calls.

### 3.2.1 Logical Channels

It takes four logical channels to conduct a CAST experiment. The data channel carries the data during the latency and throughput tests. The handshake channel is used for control information. The remote channel carries results of the test from the server back to the client. Finally, the offset channel is used to exchange the timing information to calculate the offset.

CAST does not require use of the same network architecture for the different channels. For example, if there are Ethernet and ATM networks running parallel among hosts, the ATM network can be used for the data channel, and Ethernet sockets can be used for the other three channels. The command line parameters determine what network is used for each channel.

The hosts in our Isotach prototype are connected to a common file server through an Ethernet. In the first implementation of our isotach module, we decided only to implement the data channel. We used the Ethernet for the other three channels with a sockets module provided in the CAST software distribution. Each channel implemented within a module requires a minimum of five specific functions: open client, open server, send, receive, and close. There are other functions required of all modules regardless of which channels it implements. We saved a significant amount of development time by using the existing sockets code to run the remote, offset, and handshake channels.

### 3.2.2 Reliance on UNIX

CAST is programmed for the UNIX operating system and relies on UNIX file descriptors to identify channels when sending and receiving. One advantage to this is the ability to do asynchronous I/O operations by utilizing the `select()` system call to test if any data is available for reading from a file descriptor. The `select()` system call is also useful for multiplexing file descriptors to see if one of a set has data to read. In the CAST main module, the `select()` system call is utilized in several places, thus assuming that all protocol modules utilize file descriptors to identify channels. Isotach 1.0 and FM 1.1 use node identification numbers in the form of integers to identify channels, and these node identification numbers are not compatible with the `select()` system call. Therefore, some creative modifications were made to the CAST main module to work around I/O multiplexing while implementing the Isotach and FM modules. This "hacking" of the main module was inconsistent with one of CAST's design goals. In theory, a protocol module implementation in CAST should not require any changes

to the main module.

### 3.2.3 Additional Buffering Layer

Isotach and FM extract messages from the network using an explicit polling mechanism. During a poll, each waiting message is examined sequentially in arrival order. An index to the appropriate handler is obtained from the message header, and the handler is executed, passing the address and length of the message as parameters. When the handler returns, the message buffer is returned to the system to accept another message. If the user desires that data persist beyond the life of the handler, the data must be copied to user space. The number of messages obtained in each poll is nondeterministic, because the handlers for **every** waiting message are executed.

When the CAST main module calls a receive function, it passes the receive buffer address as a parameter, and expects one message to be placed at that address. But if the CAST receive is linked to the Isotach or FM receive, then the corresponding poll might bring in more than one message. This forced us to implement a buffering layer in the FM and Isotach modules of CAST to handle the event of more than one available message upon the call of the receive function. This gives CAST an additional memory copy on the receive side. It will be shown later that this affects our CAST latency results, but has no effect on the throughput.

The need for this additional copy in the application layer would not have been necessary had the Isotach API allowed the application to specify the number of messages to extract during a poll. This is an option we are exploring for future versions of Isotach.

### 3.2.4 Packet Size

Messages on the CAST data and offset channels contain a data structure holding timestamp information for the latency and offset computations. This data structure is 44 bytes. Isotach messages carry a 20 byte header, so the minimum packet size allowed using the CAST Isotach module is 64 bytes. All Isotach and FM packets are a constant size, determined at compile time.

FM 1.1 fragmented messages that were larger then the compiled packet size, while Isotach 1.0 did not fragment messages. It may be decided that future versions of Isotach contain this functionality. At compile time, the maximum isotach message size was determined by subtracting the header size (20 bytes) from the packet size. This number was set in the `MAX_ISO_MSG_SIZE`

constant. The CAST isotach module reduced the test message size if it was larger than the compiled `MAX_ISO_MSG_SIZE`. The drawback to setting the packet size at compile time was the compilation effort to run tests with larger message sizes. The alternative would have been to implement fragmentation and reassembly in the cast isotach module, but this would have added overhead that we did not want included in the measurements.

## 3.3 Offset Accuracy

CAST measurements that were computed using the offset appeared to be skewed. We expected the one-way latency from the client to the server to be very close to the return latency. CAST results did not support this hypothesis. The measured one-way latency results were very inconsistent, with results varying by as much as 150%, even giving negative one-way latency results on occasion. At the same time, the round-trip results, which did not rely on the offset, were very consistent with very little variance.

As explained in section 3.1.1, the offset is computed from a single round-trip where it is assumed the one-way latencies on that round-trip are equal. To save development time, we did not implement the offset channel over Isotach, but rather used the available Ethernet sockets offset channel implementation. Ethernet is a CSMA-CD protocol, with the potential for multiple resends of frames with random backoff times if collisions are detected. The Ethernet link among our hosts carries NFS and other control traffic, so the potential for collisions is likely. If the round-trip used to compute the offset measurement suffers any collisions, then the offset measurement will likely be skewed.

Increasing the number of offset samples did not improve the accuracy of the offset measurement. In fact, because increasing the number of offset samples increased the length of time to obtain the offset measurement, the offset became even more inaccurate probably due to the effects of clock drift.

The offset measurement client to server `receive()` function did not parallel the server to client `receive()`. On the server side, the offset packet is received with an asynchronous notification, and multiplexing is conducted between the offset and handshake channels. It takes a series of `select()` system calls to perform this type of receive. The client's receive is a synchronous blocking receive, and uses a lot less overhead than the server's receive. These unequal receives contribute to unequal one-way latencies, and hence to an inaccurate offset measurement.

We added offset channel functionality to the Isotach and FM CAST

modules. To make one-way latencies equal, we implemented the receives on both sides without I/O multiplexing. By removing multiplexing and implementing over more predictable messaging layers, we believed we would obtain much more accurate offset measurements. The results supported our views. The one-way latencies were now much closer together, and only occasionally would we obtain one-way latency results that appeared skewed by the offset. This skewing still happened often enough to give us doubt about one-way results.

We measured the clock drift on our test hosts as 10-15 parts per million. This translates to a drift of 10-15 $\mu$s for every second. The typical latency test is only 100 round trips, so a latency test finishes long before the clock drift becomes a factor. However, we were sending enough data for the throughput test to last as long as 8 seconds, which would allow the clocks to drift more than 100 $\mu$s further apart. This discrepancy is unacceptable in a system area network where round trip latencies are typically less than 100 $\mu$s.

The inaccuracies in the CAST offset measurement were not noticeable when we ran CAST over an Ethernet network that measured latencies in milliseconds. However, when we moved CAST to our Isotach cluster where latencies were measured in microseconds, the offset inaccuracy quickly became apparent. For these reasons, we lost confidence in any computations in CAST that used the offset measurement. However, we still had confidence using CAST to measure round-trip latencies, and throughputs at the client and server interfaces.

There was little impact to not using all the CAST measurements. One-way latencies were no more significant to us than round-trip latencies. As explained in section 3.1.3, we sent enough data on the throughput test to make the end-to-end throughput result almost equal to the interface throughput results.

## 3.4 Chapter Summary

CAST is a network performance tool that gives the experimenter flexibility to customize tests. CAST provides an offset measurement that measures the difference between the client and server clocks, allowing additional computations such as one-way latencies. However, we did not gain enough confidence in the offset measurement to find it useful for our performance study, and did not use measurements that relied on the offset. The results of performance tests using CAST are presented in chapter 5.

# 4

# Test Design

Before conducting any testing, we determined the characteristics we wanted to learn about Isotach, and translated those characteristics into measurable metrics. We designed tests to measure these metrics independently of other variables. We decided to run some tests on FM and use the results as a baseline for comparison with Isotach.

## 4.1 Isotach and FM

We ran some of our tests using both Isotach and FM. It is important to note that the comparison was not for the purpose of competition, but rather for establishing a baseline against which we could compare Isotach's performance. We knew the Isotach ordering guarantees were not without cost, and the additional overhead to achieve these guarantees would keep Isotach from achieving the performance of FM 1.1. Isotach 1.0 was implemented by modifying FM 1.1, so keeping Isotach's performance close to FM's performance was a goal. We ensured that tests comparing Isotach and FM were run under similar conditions. Due to Isotach's 20 byte header, for a given packet size, FM can carry 20 more bytes of data per message. For all FM and Isotach measurements using a given packet size, the amount of data per packet was always 20 bytes less than the packet size. This amount of data maximized Isotach's throughput, but not FM's throughput. We wanted to see Isotach's performance compared to FM with equal size packets carrying equal amounts of data. FM would have had a higher round-trip latency and higher bandwidth than the numbers reported in chapter 5 had we maximized the data load per packet.

## 4.2 Message Passing versus Shared Memory

For Isotach tests, we chose to use the message passing API instead of the shared memory API (see section 2.2.3 for a discussion of the difference between these two interfaces). We wanted to compare Isotach to our baseline of FM 1.1, and the shared memory API did not lend itself well to this comparison. The shared memory API sends reads and writes through the network, and the amount of data in a read or write is fixed (either 32 or 64 bit variables). With the shared memory API, we could not get a good indication of how Isotach performance changed as packet size and payload changed.

Since we wanted to compare Isotach to a baseline of FM, we chose to use the Isotach message passing API because it is in the same paradigm as FM, two hosts exchanging messages. However, using the message passing API gave us significantly higher latency measurements for Isotach. Because the shared memory API does not perform a memory copy of data to the user at the turn-around point in round-trip latency measurements, it has lower round-trip latency than the message passing model. Additionally, the return trip on a shared memory read is made using FM, since no ordering guarantees are necessary once the variable value is read. See [14] for initial performance measurements using the shared memory model.

## 4.3 Other Test Parameters

Studies indicate that most network traffic utilizes small packets [4], so we were most interested in Isotach's performance at packet sizes of 256 bytes and fewer. We chose to test a minimum packet size of 64 bytes, because it was the smallest allowable packet size using CAST (due to the minimum 44 bytes of data). However, we needed to see how Isotach scaled to larger packet sizes, and selected 1024 bytes as a representative maximum packet size.

We ran the experiments on both 1 and 2-switch networks. Equipment limitations prevented tests on clusters with larger network diameters. Adding only one switch to the network provided a very modest insight to Isotach's scalability.

We did not have a homogeneous system. We have 2 different processors, 3 CPU speeds, and 2 types of LAN connections. All tests with which results are compared against one another were run on the same equipment, in order to keep other variables from becoming a factor. New equipment will soon

be in place to make the cluster homogeneous.

## 4.4 Latency Test

Latency is a very important measure in tightly coupled systems with fine grain communication patterns. In our tests, latency measures the travel time from the sender's application to the receiver's application. Studies suggest that, in general, applications are most sensitive to software and hardware overhead, as opposed to the gap between packets (affecting bandwidth) or the physical link latency [7]. More specifically, findings indicate that this overhead is the dominant factor in determining communication performance for applications with request-response communication patterns using small messages [5]. An original goal of the Isotach project was (and is) to incur no more than twice the latency of the messaging layer upon which it was built.

Our custom latency test sent 500 messages round trip, one at a time, between a client and a server. The clock began immediately before the first byte was sent, and it ended when the last byte of the last message was received. Upon return from a round trip, the message was copied from the user-defined handler to the user's address space. The average round-trip was computed by taking the total time and dividing it by the number of round-trips. This sequence completed one measurement. We took the average of 50 measurements. The round-trip latency figures excluded the turnaround time at the server ($\sim$0.5 $\mu$s).

This test was run using 1-switch and 2-switch network configurations on both Isotach and FM across the entire range of packet sizes.

## 4.5 Throughput Test

Although current focus in system area network research is moving away from bandwidth toward latency, throughput is still a very important metric. Many applications need to move a large amount of data as quickly as possible.

The custom throughput test measured bandwidth at the client and server interfaces. At each interface, the clock was started when the first byte of data crossed the interface, and stopped when the last byte crossed. The computation was the same as equation 3.6. The measurement was taken 5 times sending 40 Mbytes of data through the link in each measurement, reporting the average measurement in the final result. We decided on 40

Mbytes after running many tests using CAST, and finding this data amount consistently gave results with a variance of less than 1 Mbits/s.

This test was run using 1-switch and 2-switch network configurations on both Isotach and FM across the entire range of packet sizes.

## 4.6 Contention Test

It was discovered early in our performance study that the sender was clearly the bottleneck in our Isotach system. We wanted to see how much room we had to improve the sender's efficiency before the receiver became the bottleneck. We designed a test in which many clients sent messages as rapidly as possible to a common server. This test gave us the ability to observe the maximum throughput at the server interface. It also provided a feel for the stability of the Isotach system under high contention, hence the test name.

The setup phase of this test included the determination of how many clients can send to a common server before adding another client fails to give more throughput at the server interface. Once this was accomplished, the throughput was measured at the client and receiver interfaces, with all clients sending data continuously at their maximum rate for 30 seconds. The code was instrumented to see if flow control throttled the clients. Additionally, the amount of sent data was checked against the amount of received data to see if any packets were dropped. This test was run on both Isotach and FM, at 64 and 1024 byte packet sizes, using only a 1-switch network configuration.

The round-trip latency was also computed under contention by adding a client that only exchanged round-trip messages with the server. We expected the latencies to be very high due to the large number of messages queued on the server. In this test, round trip messages were exchanged with the server for 30 seconds, after which the average round-trip latency was computed.

## 4.7 Isochron Test

A characteristic of an isotach network is its ability to send isochrons, groups of operations that appear to be executed atomically. The simplest way to implement isochrons is to buffer the operations while the isochron is built, and then send the isochron messages in the same logical time pulse with the same receive time. Therefore, in our prototype, isochron elements are held up at the sender while the isochron is constructed. We expected sender

bandwidth to decrease as the size of isochrons increased, because we were buffering messages longer in the sender.

The isochron test measured bandwidth as the size of isochrons increased. The interesting bandwidth measurement was at the sender, because we knew the sender to be the bottleneck. The sender continued to send isochrons until a total of 40 Mbytes of data were sent, and then the resulting sender bandwidth was computed and reported.

## 4.8 Impact of Tokens on Non-Isotach Traffic

Our Isotach prototype ensures that non-isotach traffic is not delayed anywhere in the system due to Isotach overhead. We wanted to show this by measuring FM's performance with an FM background load versus FM's performance with an Isotach background load. We did not resolve the experimental design issue of how to ensure the test streams were comparable. Therefore, we decided to run tests with the more modest goal of examining the effects of token traffic on FM.

We reran the latency and throughput tests on FM, except this time we had the SIUs and TMs exchanging tokens while we were executing these tests. This test at least showed the effect of token traffic on non-isotach streams of data. We expected negligible impact on the FM latencies and bandwidth from the token traffic, since token passing utilizes less than 5% of the available network bandwidth (tokens are only 5 bytes in length), and the SIUs should pass FM traffic on to the host with minimal delay.

## 4.9 Chapter Summary

Tests were designed to measure latency, throughput, performance under contention, impact from the increasing size of isochrons, and the effect of token passing on non-isotach traffic. This chapter detailed the test designs, and the conditions under which the tests were run.

# 5

# Results

In this chapter, we present the results of our performance study. In general, the results are consistent with expectations, although in some instances, Isotach did not perform as well as expected. In these cases, we investigated further to discover the reasons for the lower than expected performance. In all cases, we were able to explain the performance results with a high degree of confidence in our explanations. The knowledge gained through this performance study has led to recommendations to optimize current and future implementations of our Isotach prototype.

## 5.1 CAST

The results of latency and throughput tests on a 1-switch network configuration are presented in figures 5.1 and 5.2. The CAST latency results are significantly higher than those measured by our custom latency test as described in section 4.4. However, the CAST and custom test bandwidths are almost equal. Additionally, we noted that the CAST latencies diverged from our custom test latencies as packet size increased. This section explains our observations.

### 5.1.1 Overhead

CAST latency results were higher than expected. We ran a custom latency test program that measured round-trip latencies between a client and server. At 64 byte packets, CAST round-trip latencies were 14% higher than latencies measured by our custom test. At 1024 byte packets, CAST measurements were 18% higher. There were several reasons why the CAST and custom test measurements were different.

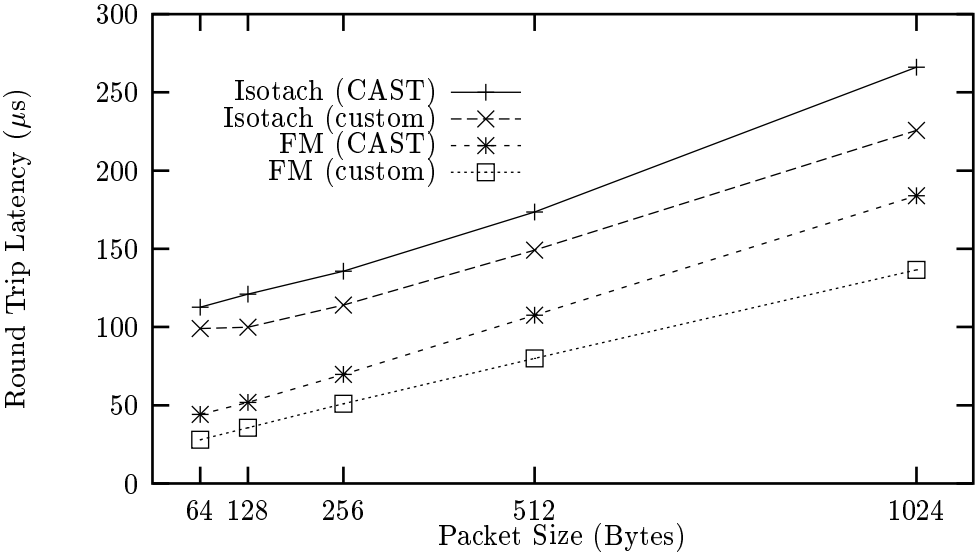


Figure 5.1: Round-Trip Latencies in a 1-Switch Network

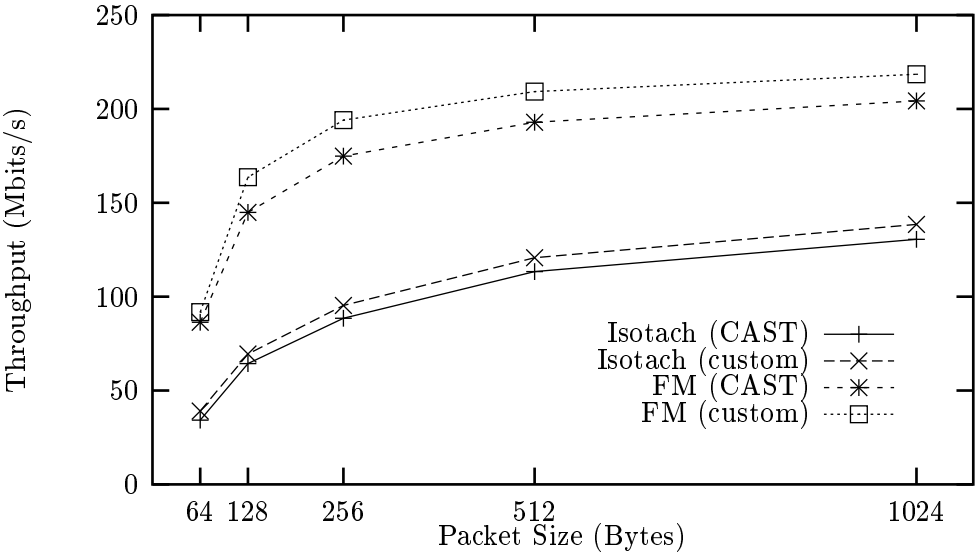


Figure 5.2: Bandwidth in a 1-Switch Network

### System Calls

CAST uses the `gettimeofday()` system call for time stamps. This is an expensive system call, costing an average of 5 $\mu$s on our system. Our custom test uses inline assembly code that reads the CPU cycle counter from a register. Cycles are converted to microseconds by measuring the system clock rate and converting. The cost to obtain a time stamp in our custom test is less than a quarter microsecond.

### Memory Copies

As explained in section 3.2.3, CAST costs Isotach and FM an extra memory copy on the receive side that we did not implement in our custom test. In our custom test, we eliminated this extra copy leaving a single copy from the receive buffer in host memory to the application. Additionally, there is a small cost to do the buffer management associated with the extra memory copy in CAST.

### Function Layering

An additional cost of CAST's modular implementation is function layering in conducting sends and receives. For example, to receive a message through the Isotach module, `Nread()` is called in the main module, which calls `read_data_generic()`, which calls `intrfc_strct_ptr->intrfc_funct_strct.recv_data()`, which is a pointer to the `recv_data_isotach()` function in the isotach module. After the data is received, in some cases there is other code to be executed, before the functions return, that adds to the critical path of the latency measurement. A send has similar layering, but with one less function call. In fairness to CAST, it was only discovered after the performance testing was completed that the optimizer was turned off during compilation of the CAST source code. The optimizer can do function inlining that can make this function layering much more efficient. To see what the optimizer would do, we recompiled CAST with the maximum optimization level, and ran an Isotach performance test with a large packet size. The round-trip latency dropped from 266 $\mu$s to 261$\mu$s, a gain of less than 2%. Therefore, this slight optimization is not reflected in the performance numbers presented in this chapter.

### Test to Remove Overhead

To verify that CAST overhead was causing the increase in latency, we modified CAST to remove some of the overhead. The modification consisted of inserting time stamps using inline assembly code within the Isotach module immediately before the send, and right after the receipt of the data. This modification removed the cost of the function layering and the `gettimeofday()` system calls from the latency test, but still included the additional memory copy and buffer management costs. Without our modification, CAST measured the average Isotach round-trip at 266.1 $\mu$s for 1024 byte packets. The modified version of CAST measured 230.3 $\mu$s. This dramatic decrease in latency shows the cost of the CAST overhead! Our custom latency test measured an average round trip of 225.7 $\mu$s, less than the modified CAST measurement, in part because there were two fewer memory copies on a round-trip.

### Latency Divergence

Figure 5.1 shows that CAST results diverged from the custom test results as packet size increased. This divergence was a direct result of the additional memory copy forced by the non-deterministic retrieval of data during polling. The size of this divergence showed how the memory copy dominates the difference between the custom tests and CAST at 1024 bytes. As discussed in chapter 3, the expensive system calls and function layering also play a role in the additional CAST overhead.

### Bandwidth

In figure 5.2, we present the throughput results at the client interface.[1] CAST performed much more closely to our custom tests measuring bandwidth because two of the three CAST overhead issues were not in the critical path of the bandwidth measurement. CAST did not take expensive time stamps for each message in the throughput test. We will show later that for Isotach and FM, the bandwidth bottleneck is on the client side, not the receiver side. Therefore, there is plenty of time to do the additional memory copy on the receiver before the next message arrives for processing. That only leaves the function layering as a CAST overhead issue for the throughput test, and this layering is less on the client side, where we know the bandwidth bottleneck is located.

[1] In all throughput measurements, the results at the client and server interfaces were essentially equal, so we only present the client results.

### 5.1.2 Recommendations

The CAST overhead can be reduced significantly with one minor change. CAST should obtain time stamps in the protocol modules, immediately before sends and after receives. The obvious benefit of this is to reduce the cost of the function layering. This would also allow module implementers to find creative ways to obtain time stamps without using expensive system calls. But to allow this, CAST would also have to require modules to implement a function translating the difference between two time stamps into microseconds.

The other recommendation for the CAST designers does not affect performance, but rather ease of module implementation. CAST should not be reliant on UNIX. Many modern messaging layers do not use file descriptors to identify channels, and these systems are not compatible with CAST without modifications to the main module. Module implementers should not be forced to make any modifications to the main module.

### 5.1.3 CAST Summary

CAST is a network performance tool that allows maximum flexibility in the execution of network experiments. CAST has additional overhead in its architecture that adds as much as 18% to our round trip latency measurement. This overhead is the result of expensive system calls, function layering, and additional memory copies. When all overhead minus the additional memory copy was stripped from CAST measurements, the results were similar to latencies obtained from custom latency tests. The CAST throughput results were very close to what we measured in our custom bandwidth tests. This is because most of the CAST overhead was not in the critical path of sending messages from the client to the server as quickly as possible.

## 5.2 Latency and Throughput Test Results

Figures 5.3 and 5.4 provide a comparison between Isotach 1.0 and FM 1.1 for both one and two switch networks. When a second switch was added to the Isotach cluster, latency increased with no loss in bandwidth. As discussed in section 5.2.1, the increase in latency is due to the later receive time. This did not affect the bandwidth due to the pipelining of messages within the system.

A goal of the Isotach project was to achieve 50% of the available bandwidth and no more than twice the latency of FM 1.1, the messaging layer

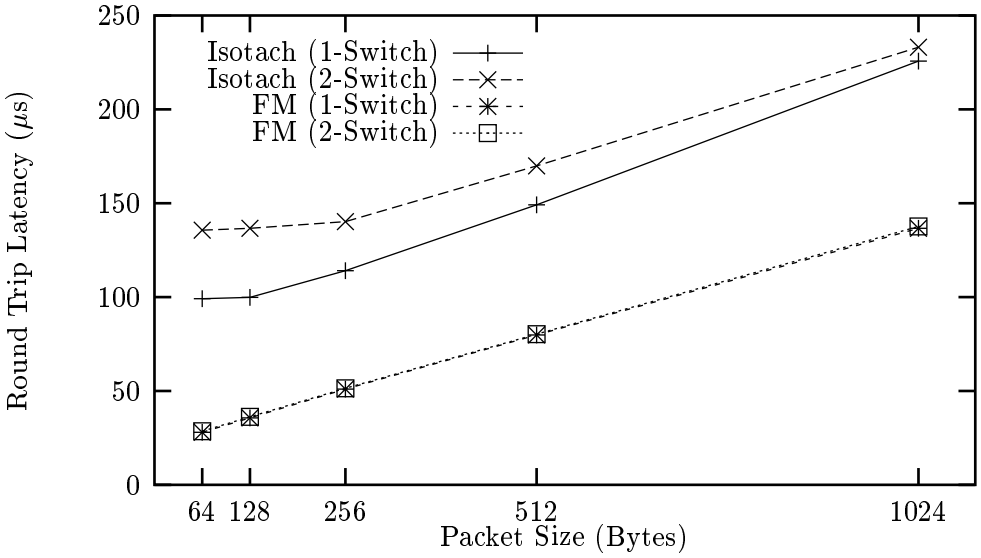


Figure 5.3: Round-Trip Latencies in 1/2-Switch Networks

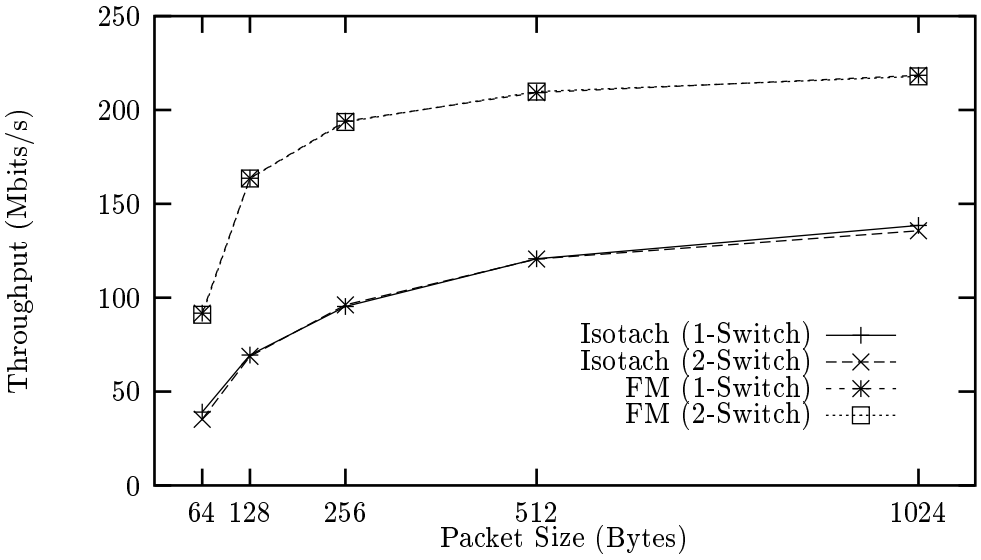


Figure 5.4: Bandwidth in 1/2-Switch Networks

that was modified to implement Isotach 1.0. For 64 byte packets, Isotach's latency was three and a half times the latency of FM, with that number reduced to just over one and a half times at 1024 byte packets. Thus, we did not meet our goal at small packet sizes. For bandwidth, Isotach achieved 63% of FM's bandwidth at large packet sizes, but only 43% with small packets. Once again, we fell short of our goal at small packet sizes. As discussed in section 5.2.2, the memory and I/O bus contention in the host was the main cause of Isotach's lower than expected performance with small packets.

### 5.2.1 Impact of Network Configuration on Performance

Isotach demonstrated increased latency when the second switch was added to the network. This increase was predictable, because the average token inter-arrival time (under no load) increased from 11.6 to 13.2 $\mu$s with the addition of the second switch. Additionally, the isotach invariant forced the SIU to assign later timestamps because of the additional logical distance to travel. At this rate of token exchange, messages will usually have a real-time delay at the receiver waiting for logical pulses of execution. When the Isotach version 2.0 hardware is placed into the network in the near future, we expect these delays to become negligible as the token inter-arrival time may be less than a microsecond.

Even though the latency increased, adding a second switch had no effect on the network bandwidth. The bandwidth was stable because messages were pipelined, and the rate at which they were pushed onto the network did not change. This stable bandwidth was a good sign of the scalability of Isotach networks, though a more convincing result will come when a larger cluster is tested.

### 5.2.2 Evaluating the Results

#### Message Passing versus Shared Memory Latency

The Isotach round-trip latency with 64 byte packets was 99.1 $\mu$s. This was considerably higher than the shared memory remote read latency reported in [14], 54 $\mu$s. The reason for this difference was two-fold. First, the message passing API had two layers of handlers and two copies on each receive (one copy from the host receive buffer into the SMM, and a second copy from the SMM to the application), versus one handler and one copy for the shared memory read at the turnaround point. Additionally, the shared memory remote read was returned via the FM API, so it did not have the Isotach SMM overhead on the return trip. We modified the SMM code to

turn around messages in the isotach layer and return them with an FM send, similar to the treatment of a remote read. This resulted in a message round-trip latency that was equal to a shared memory remote read latency. There was no inherent difference between the Isotach shared memory model and message based model, except in the additional layer of handlers on the receiver.

### Nondeterministic Wait

Isotach networks have the potential for additional latency while messages wait for their logical time of execution. There is a set of events that must occur before an sref is executed. First, the sref is consumed from the receive buffer and placed in a bucket. Some time later, a corresponding EOP marker is consumed which flags the bucket as executable. Then the bucket is sorted and drained, executing each sref's handler in logical time order. Figures 5.5 and 5.6 present two possible receive scenarios. In the first figure, the EOP marker is consumed in the same poll as the sref, thereby draining the bucket before the poll returns control to the user application. In the second figure, the sref is consumed, but the EOP marker has not yet arrived at the host. The sref waits in the bucket until the poll following the EOP marker's arrival in the receive buffer. Once the EOP marker is consumed, the bucket is sorted and drained. The sref's delay in the SMM is dependent upon the frequency of polling, the arrival of the EOP marker, and the number of other operations the SMM must execute for other packets that have arrived in the host receive buffer. For our latency performance test, the real issue is whether the EOP marker is in the host receive buffer when the message is consumed. Stated differently, the issue is whether it takes one poll or many polls before the user level handler is executed. If the user polls often, and the token inter-arrival time is low, messages will have minimal delays in the SMM. We expect these circumstances to be the case in most isotach applications.

### Additional Memory Copies

We were not surprised to see that Isotach had higher latency than FM. The Isotach messaging layer was implemented with 2 levels of handlers resulting in twice as many copies as FM on both the sender and receiver. The first sender copy is placed into the to_niu buffer while the isochron is built. From there it is copied to the network interface using programmed I/O. On the receiver side, a handler copies the message out of the receive buffer into a

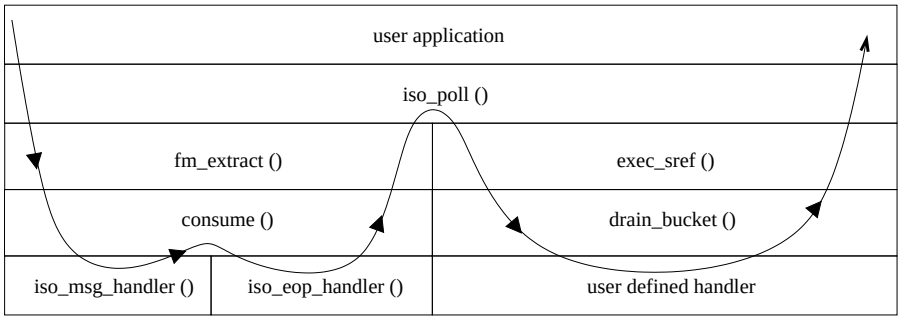


Figure 5.5: Receiver Call Stack: One Poll to Retrieve a Message

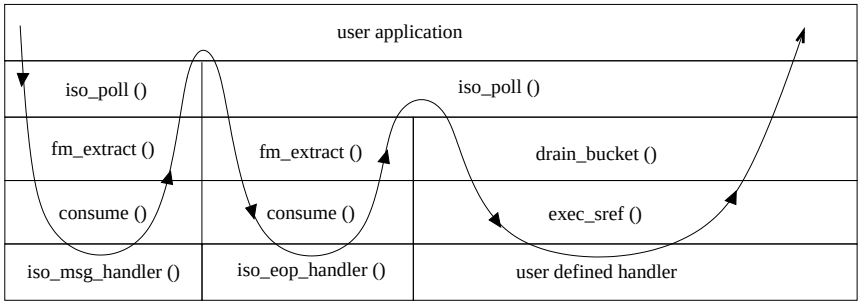


Figure 5.6: Receiver Call Stack: Two Polls to Retrieve a Message

bucket. Once the bucket is executed, the user-defined handler copies the message into user space.

### Isolating the Overhead

Although we understood Isotach to have higher overhead than FM due to layering of handlers and additional memory copies, this explanation did not account for 71 $\mu$s of latency difference between FM and Isotach at 64 byte packets. It also did not account for the fact that this latency difference stayed relatively constant as packet size increased. If the data copies were dominating the overhead, then we expected to see this difference increase linearly as packet size increased. The bandwidth difference between Isotach and FM also stayed relatively constant.

We determined that the sender was the bandwidth bottleneck. The sender was not blocking on sends due to the FM flow control mechanisms. Yet, the sender and receiver bandwidths were equal. This observation meant that the receiver was keeping up with the sender, without blocking the sender. We instrumented the test code to measure the cost of the Isotach send sequence. This sequence consists of three function calls to build and send an isochron: `iso_start()`, `iso_send_msg()`, and `iso_end()`. With 1024 byte packets, this sequence takes on average 58.2 $\mu$s. If a process was doing nothing but this send sequence at the rate of one send every 58.2 $\mu$s, it would achieve a theoretical throughput of 138.0 Mbits/s (with 1004 bytes of payload per packet). Our Isotach throughput test at 1024 byte packets reported a bandwidth of 138.5 Mbits/s. We appeared to have exceeded the theoretical maximum throughput, but that is only because we averaged the send time to the nearest tenth of a microsecond. More significantly, since the measured throughput corresponded to the maximum sending rate achievable by the sender, it proved the sender to be the bottleneck in our system.

### Excessive Polling

The sender was polling the receive queue for every isochron sent. It was necessary for the SMM to poll in order to keep queues from being overrun. Every sent isochron reserves an element of the hit buffer and causes the SIU to send an EOI marker followed some time later by an EOP marker. If a host continually sends without polling, eventually the hit buffer will be overrun or the host receive buffer will fill up with EOI and EOP markers that need to be extracted. The SMM can not afford to assume the user application will poll, but at the same time it does not need to poll every

send, just periodically. We discovered that when polling every eighth send, the SMM kept the buffers from being overrun, while amortizing the cost of this poll over eight sends. This gained us 10 Mbits/s of throughput at 64 byte packets and 4 Mbits/s with 1024 byte packets. All performance test results listed in this report were obtained with this small optimization in place in the system.

One issue with this optimization is that in Isotach 2.0, the EOI markers might be used for SIU flow control. EOI markers are only returned by the SIU once the isochron is on the wire and the buffer space is available. It may be necessary for the host to see these EOI markers to know that the buffer space is available, and polling every 8 sends may need to be studied with this flow control mechanism in mind.

### Bus Contention

While we gained back some of the difference in throughput with the polling modification, it didn't account for most of the performance difference between FM and Isotach. We determined that the programmed I/O moving data from the host to the network interface card was taking more time than expected. The only explanation for the long programmed I/O was bus contention on the memory and I/O buses. We conjectured that possibly the EOI and EOP markers moving from the network interface to the host were blocking the programmed I/O from the host to the network interface.

In our throughput tests, we were sending purely remote isochrons. Essentially, the EOI and EOP markers were overhead on the sender and were not necessary because we knew there were no local isochron components for execution on the sender. For this reason, we were able to instrument the SMM and SIU so that a sent isochron did not trigger the return of markers from the SIU. The receiver still handled EOP markers to mark significant pulses for the received Isotach messages. We reran the throughput tests, and noticed an 82% improvement in bandwidth at 64 byte packets, and a 29% improvement at 1024 byte packets. Figure 5.7 shows the cost of an Isotach send sequence with and without the EOI/EOP markers traveling from the SIU to the host. Additionally, we measured the cost of the additional copy into the to_niu buffer to see the effect of eliminating the host to_niu buffer. The cost of the `FM_send()` call is listed for comparison. The columns labeled "Send" display how the bus contention affects the one-way latency by slowing down the sender. Additionally for each send measurement, the theoretical bandwidth is listed in the "BW" column for a host doing nothing but sending at the rate shown by the send measurement. This table illus-

| | 64 Bytes | | 1024 Bytes | |
|---|---|---|---|---|
| | Send ($\mu$s) | BW (Mb/s) | Send ($\mu$s) | BW (Mb/s) |
| Isotach | 9.4 | 37.4 | 58.2 | 138.0 |
| w/o markers | 5.2 | 67.7 | 49.8 | 161.3 |
| and w/o memcopy | 4.3 | 81.9 | 46.1 | 174.2 |
| FM | 3.3 | 106.7 | 37.5 | 214.2 |

Figure 5.7: Send Cost and Corresponding Bandwidth

trates the cost of the bus contention and a memory copy on both latency and throughput.

The fact that the programmed I/O can potentially contend for the buses with the DMA of EOI/EOP markers is further compounded because our current prototype DMAs an entire `PACKETSIZE` of data for each EOI and EOP marker, even though they may only contain a few bytes of data. For packet sizes of 1024 bytes, this is quite expensive, because an eight byte EOI marker is the payload for a one kilobyte packet.

## 5.3 Contention Test

Results for the contention test described in section 4.6 are summarized in figures 5.8 and 5.9. The Isotach prototype network maintained stability with no dropped packets in a high contention situation where a receiver host was overwhelmed. The throughput data shows that we can improve the performance of the sender significantly before the receiver host will start becoming the bottleneck. Of the four data points obtained in the throughput results in figure 5.8, only Isotach at 64 bytes was obtained without flow control blocking the senders. The Isotach receiver at 64 byte packets could process 83.0 Mbits/s. Without optimizing the receive sequence, this is the maximum receiver throughput, no matter how much we optimize the send sequence. With the other three data points, flow control was blocking the senders. It is not apparent whether the receiver could have attained more throughput, because the blocking prevented us from knowing if the receiver was always busy processing packets.

The results in figure 5.9 show large latencies when sending to a busy receiver. The packets spent a lot of time in the receive buffer (and pulse buckets for Isotach) waiting for other packets in front to be processed. The latencies were low for Isotach at 64 byte packets relative to FM because the

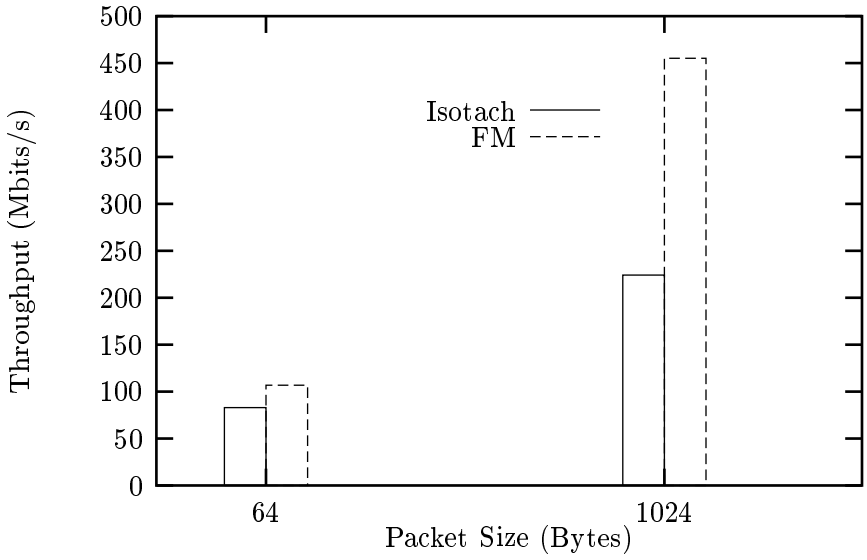


Figure 5.8: Receiver Bandwidths under Contention

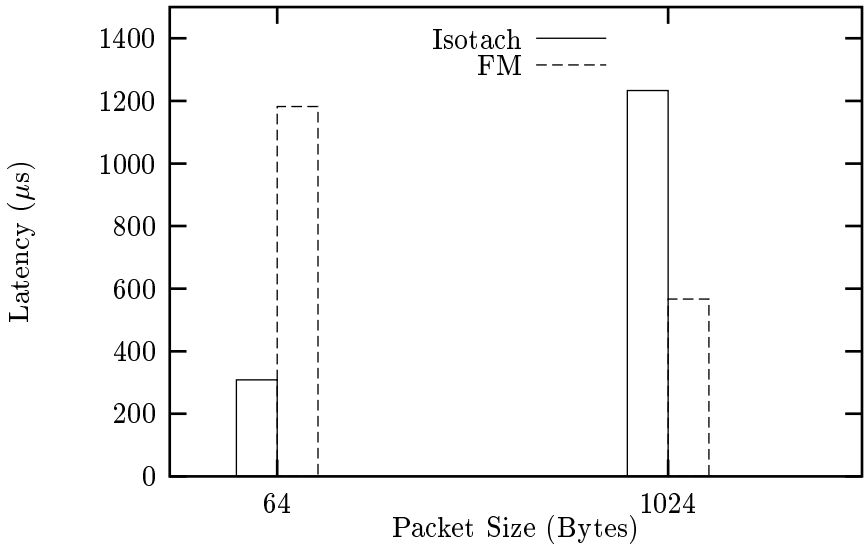


Figure 5.9: Round-Trip Latencies under Contention

Isotach receive queue was shorter than the FM queue. The Isotach receiver was keeping up with the senders as evidenced by the lack of flow control blocking. If we had added more load on the Isotach receiver, these latencies would have increased as the receive queue length increased and the senders started blocking. At 1024 byte packets, both FM and Isotach senders were blocking, so the receive queues were both filled up. Yet the FM latency was much less than the Isotach latency because FM has less overhead on the receive side and can service its queue more quickly.

In running this test, we determined how many senders we could add and still see increases in the receiver throughput [2]. If we continued to add load after that saturation point, the receiver throughput would stay relatively stable, but the measured latencies would increase due to a longer receive queue. Eventually, adding further load on the receiver would cause the senders to time-out waiting for send credits.

## 5.4 Isochron Test

The isochron test described in section 4.7 provided counterintuitive results, as displayed in figures 5.10 and 5.11. We hypothesized that increasing the size of isochrons would decrease the available bandwidth. As Isochrons are built, messages are buffered in the host, thus increasing the wait period on the sender host between sends. However, now that we know that the limiting factor on our bandwidth is bus contention between programmed I/O of messages and DMAs of markers, if we limit this contention, then we increase bandwidth. By increasing the size of isochrons, we amortized the cost of sending EOI and EOP markers from the SIU to the host over more messages. Thus, there was less bus contention per message for the sending host. With 64 byte packets, our bandwidth increased from 38.8 to 60.1 Mbits/s as isochron size increased from one to four. With 1024 byte packets, bandwidth increased from 137.8 to 171.5 Mbits/s. This result shows that we could have improved Isotach throughput test results significantly by sending isochrons with multiple messages instead of just one message.

[2] Three senders saturated the receiver using Isotach at 1024 byte packets. For Isotach at 64 byte packets, and FM at 64 and 1024 byte packets, it took 4 senders to saturate the receiver.

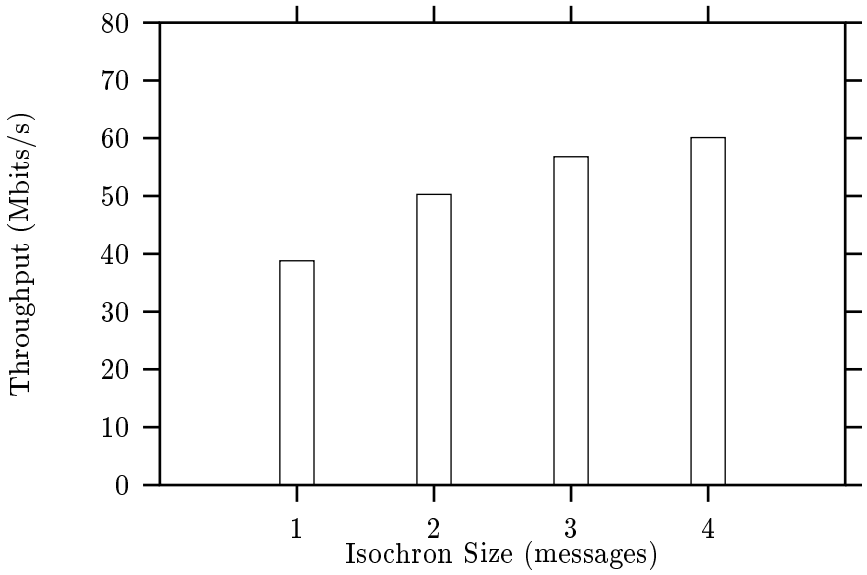


Figure 5.10: Effect of Isochron Size on Bandwidth (64 Byte Packets)

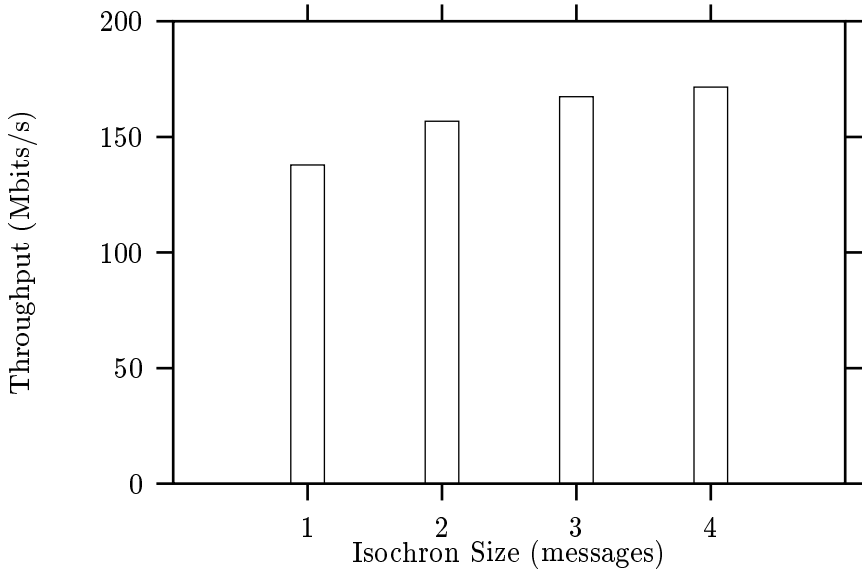


Figure 5.11: Effect of Isochron Size on Bandwidth (1024 Byte Packets)

## 5.5 Impact of Tokens on Non-Isotach Traffic

Figures 5.12 and 5.13 compare the network performance of FM with and without token traffic in the background. In section 4.8, we stated our prediction that tokens (only 5 bytes in length) would have a negligible impact on FM latencies and bandwidth. The results were encouraging, with almost no effect on network latencies. The bandwidth dropped 25% at small packets, but caught up with no discernible effect at large packet sizes. The small packets were probably blocking during the programmed I/O from the host to the network interface card while the SIU processed tokens. We expect when the hardware SIUs are placed in the network, the network interface will process packets from the host more quickly, since it will no longer need to process tokens and determine packet receive times.

## 5.6 Comparison With Other Messaging Layers

Our tests show that Isotach compares very favorably against traditional TCP/IP networking. For 64 byte packets, Ethernet socket and Myrinet socket latencies exceeded Isotach latencies by a factor of 9 and 4 respectively. Consequently, Isotach bandwidth exceeded Ethernet socket and Myrinet socket bandwidths by factors of 9 and 2. Isotach provides strong ordering guarantees, and still outperforms TCP/IP. However, TCP/IP provides reliable message delivery over unreliable networks, something Isotach cannot provide.

In the past decade, a flurry of research has led to the development of high performance messaging layers that have made TCP/IP networking obsolete on system area networks [12, 17, 18]. These messaging layers have focused on reducing the cost of protocol stacks, keeping the operating system out of the critical path, and optimizing data movement between the host and network interface. The result has been one way latencies ranging between 10 and 30 $\mu$s and bandwidths exceeding hundreds of megabits per second. However, these high performance protocols do not provide the guarantees found in Isotach. For a software version of Isotach to be within a factor of three (in both latency and throughput) with the cutting edge of messaging technology is very impressive.

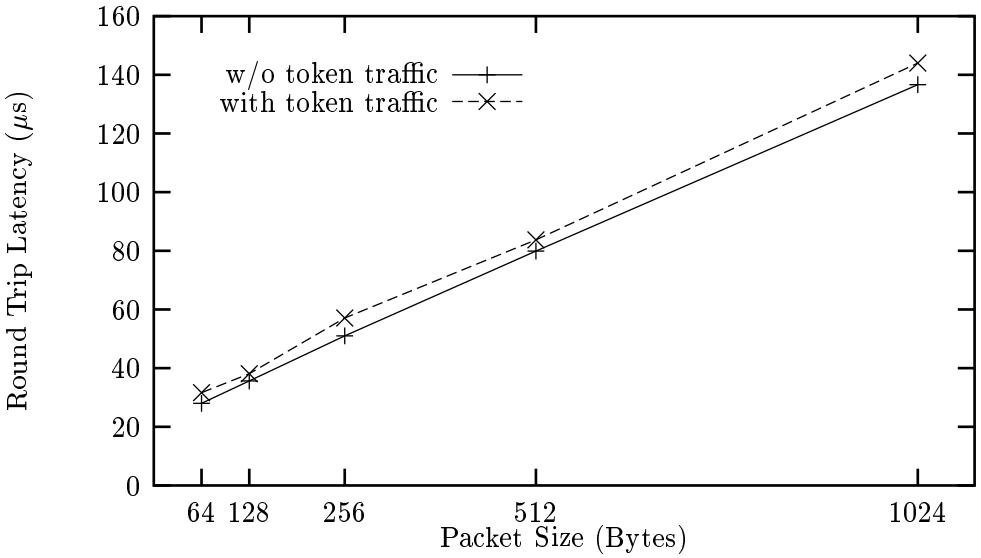


Figure 5.12: Impact of Token Traffic on FM Latency

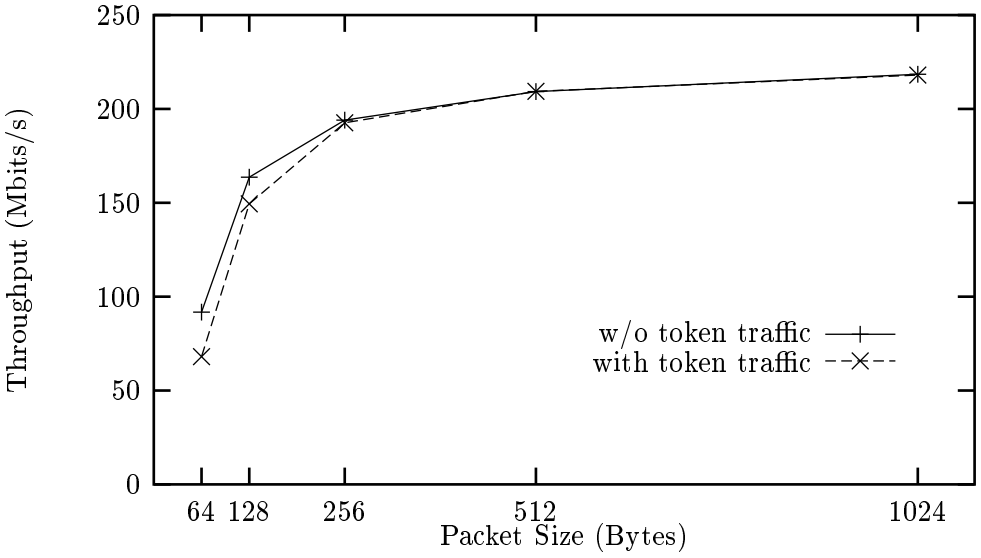


Figure 5.13: Impact of Token Traffic on FM Bandwidth

## 5.7 Chapter Summary

We implemented CAST modules for Isotach and FM, and presented performance results using the CAST benchmarking tool. We discovered overhead in CAST that is not present in our custom network performance programs, and presented our own performance study numbers in comparison.

Although Isotach 1.0 does not meet all the performance goals originally set for the project, we are very pleased with the performance of this interim unoptimized version of the prototype. The addition of custom built hardware TMs and SIUs should improve the performance even more. We investigated and discovered the reasons for Isotach falling short of some of its performance goals.

# 6

# Recommendations for Next Generation Isotach

Our performance study reveals the cost of some of the design decisions in the Isotach prototype. Many of the components limiting Isotach's performance can be improved by rewriting parts of the SMM.

## 6.1 Reduce the Cost of EOI/EOP Markers on the Sender

It was shown that bus contention between programmed I/O of data and the DMA of markers is the most significant issue limiting Isotach's performance. There are several ways to relieve this contention.

First, the current implementation uses FM to send up an entire `PACKETSIZE` of data, regardless of the size of the original message. A rewrite of the messaging layer should include dynamically sized DMAs from the network interface to host memory, thereby limiting bus contention.

Currently, all data moving from the host to the network interface is copied with programmed I/O. Programmed I/O is inefficient for large packet sizes. We need to investigate moving packets from the host to the network interface using DMA to see if it would be more efficient.

To reduce the potential for bus contention, we should investigate schemes that batch EOI and EOP markers from the network interface to the host. This could amortize the cost of these markers over many sends. However, one potential tradeoff to executing this type of scheme is the unnecessary delayed execution of local isochron components.

## 6.2 Reduce System Polling

On every Isotach send, the SMM polls. This frequent system polling helps ensure that buffers and queues in the host are not overrun. This is a safety valve to ensure the application doesn't continually send without allowing the SMM to execute the Isotach functionality that needs to occur (extract packets, place messages in buckets, drain buckets, etc.). It is not necessary to poll with every send, because polling can be done every $i$th send, where $i$ is tuned for system performance. If we assume the application will poll frequently, or as necessary for the application's workload, then we should only make the SMM poll often enough to keep those infrequently polling applications from blocking other processes and over-running SMM data structures. Then well-behaved Isotach applications that poll frequently won't have to pay the price of frequent system polling in addition to their own polling. The tradeoff is that those applications that don't poll frequently, but rely on the system to poll for them, will have much longer polls because the system will have a higher workload per poll.

## 6.3 Reduce the Number of Copies

The sending host copies a message twice. The first copy moves the data into the to_niu buffer while the isochron is built. The second copy moves the message from the host to the network interface. If the to_niu buffer is moved from the host to the network interface, this will reduce one copy per message on the host sender. Messages can be buffered on the network interface card until it receives a signal from the host to push them onto the network. In addition to saving a copy, this has a pipelining effect allowing the host and network interface to work in parallel. Buffering isochrons on the network interface becomes more feasible in Isotach version 2.0 when the SIU functionality is removed from the network interface. Since the LANai processor only operates at 30-40 Mhz, we will need to study whether the buffering puts too much load on the network interface, taking it away from its main mission of sending and receiving packets.

The receiver also has two data copies. The first data copy is from the host receive buffer into the SMM. The second copy is from the SMM into the user application (if the user chooses for the data to persist beyond the life of the handler). A rewrite of the messaging layer could employ a mechanism similar to the endpoints used in U-Net [18] and extended in the VIA standard [19]. There would be a linked list of message descriptors for the receive queue.

When a message is received, it is copied directly into the buffer corresponding to the first free descriptor in the receive queue. The SMM takes descriptors off the receive queue and manages them through the receive sequence until the user-defined handler is called. At this point, the descriptor is passed to the user. When the user is finished with the buffer, it would place the descriptor back on the end of the receive queue. This integrated application-protocol buffer management could also be employed on the send side to save copies. The tradeoff is that passing buffer pointers around saves copies at the expense of increased complexity in buffer management.

Currently, isotach applications may be required to implement their own buffer management due to the nondeterministic number of messages received in a poll. This sometimes costs an additional copy, and buffer management is not free. This copy may eventually reduce the amount of Isotach bandwidth available to the user application. To prevent applications from having to implement their own buffering, a messaging layer rewrite could allow the user to retrieve one message at a time. At this time, the receiver is not the throughput bottleneck, so this kind of mechanism may not give us an increase in performance. However, it would simplify implementing Isotach applications.

## 6.4 Chapter Summary

Recommendations were made to improve the performance of future versions of Isotach. Optimizing the movement of data from the host to the network interface, and the movement of markers in the opposite direction, will reduce bus contention. Reducing unnecessary system polls will make the sender more efficient. Reducing the number of data copies will significantly improve the latency of Isotach packets, especially with larger packet sizes.

# 7

# Summary and Future Work

A performance study was completed on Isotach 1.0. This study accomplished the following objectives:

- Measured the performance of Isotach using a proprietary benchmarking tool and custom developed tests. These tests reported:
  - Latency
  - Throughput
  - Performance under contention
  - Effects of increasing isochron size
  - Effects of token traffic on non-isotach traffic
- Discovered limitations in the design of the Isotach prototype.
- Proposed solutions to optimize the performance of current and future generations of Isotach.

The unoptimized Isotach prototype performed very well, exceeding many of the original goals of the project. We expect the performance to improve even more when we implement optimizations suggested by this study and when custom hardware components replace firmware components.

In the future, we want to perform these same tests on Isotach 2.0 with its custom hardware. Additionally, we suggest implementing the next generation of Isotach using some of the recommendations in this report.

# A

# Isotach Addendum to CAST Documentation

This appendix describes the details necessary to successfully run CAST over Isotach. The reader should be familiar with both the Isotach API [14] and the CAST documentation [11]. This appendix assumes the reader is using the Isotach 1.0 API. CAST will work on future generations of the Isotach prototype as long as there are no modifications to the syntax or semantics of the Isotach 1.0 API. If the API is modified, then the CAST isotach module must be inspected for compliance with the API. This module is located in the `cast/isotach_module` directory.

## A.1 File Structure

There is a main `cast` directory that includes the `Makefile` and CAST documentation. Under this directory are the module subdirectories, each containing a `Makefile`, source code, executables, and scripts. The `cast_main` subdirectory is the main module that comprises all the generic CAST algorithms to run the tests. The other subdirectories implement the protocol modules: `sockets_module`, `isotach_module`, and `fm_module`.

## A.2 Compiling Isotach and CAST

To compile CAST on the Isotach cluster, in the `cast` directory enter `make clean` followed by `make linux`. To compile for the Solaris operating system on another network, enter `make clean` followed by `make solaris`.

## A.3 Running CAST on Isotach

CAST has a long command line, so it is best to save it and modify it in a script file. The following command lines will successfully execute CAST over sockets:

```
cast -t -W d=sockets,rhost=kermit-m,prot=tcp -W \
o=sockets,rhost=kermit-m,prot=udp -W h=sockets -W r=sockets

cast -r -W o=sockets,prot=udp
```

In the first command line, the `-t` designates this host as the client. The `-W` argument sets up the data, offset, handshake, and remote channels. Each `-W` designates the channel, architecture, protocol, and remote host. Those arguments to `-W` not specified will use defaults. The second command line designates the host as the server (`-r`). In this example, the server lets all the channel options default except for the offset channel, where UDP is specified for the protocol. The server does not need to designate a remote host, as it listens to a well-known port for connections from any client.

The following command lines were used to run the CAST Isotach performance tests:

```
cast -t -v -i 5 -l 44 -f 40000000 -j 10 -W d=isotach \
-W o=isotach -W h=sockets,rhost=kermit,prot=tcp \
-W r=sockets,rhost=kermit,prot=tcp

cast -r -v -I -W d=isotach -W o=isotach \
-W h=sockets,prot=tcp -W r=sockets,prot=tcp
```

The first command line is for the client. The `-v` designates verbose mode that prints out more statistics. The `-i` option designates the number of iterations of the test. The `-l` option sets the size of the data buffer sent in each message. The `-f` option designates the amount of data to send in bytes during the throughput phase. The `-j` option sets a timeout. We found it necessary to override the default timeout on our Isotach cluster to keep the program running during channel setup. The next four `-W` options set Isotach as the architecture for the data and offset channels and sockets for the handshake and remote channels. The second command line is for the server. The only new item here is the `-I` which tells CAST to print the server statistics on the server's terminal. Otherwise, all statistics are printed on the client. To run FM tests using CAST, replace each occurrence of "isotach" with "fm" in the previous two command lines.

## A.4 Other Test Parameters

The following bullets are in no particular order and are notes to facilitate CAST performance testing on Isotach or FM.

- The `MAX_OFFSET` constant in cast.h gives the largest allowable difference between the client and server clocks in microseconds. Occasionally, our clocks moved farther apart and this constant needed to be increased.
- Channels that use sockets default to the Ethernet links. In order to run TCP/IP tests over the Myrinet physical links, first run the `ip-up` script located in ~`rgb2u/cast`. Then to use the Myrinet link, designate `-m` immediately following the host name (e.g. `kermit-m`). To bring the Myrinet IP interface back down, run the `ip-down` script. Isotach and FM applications cannot be executed when the Myrinet IP interface is up.
- The node identification numbers for the client and server are hard-coded constants in `cast_isotach.h` and `cast_fm.h`. The server is always 0, and the client is always 1. These node identification numbers must also be hard-coded in the fmconfig file by placing `:<NODE_ID>` after the host name. Here's an example fmconfig file for a 1 switch network:

  ```
  0 bert.cs.virginia.edu:T kermit.cs.virginia.edu:0 \
  cookie.cs.virginia.edu:1 - - - - -
  ```

- `isofm/tmhost` is the token manager application. FM 1.1 as modified in our system also requires a token manager. Turn off token traffic when using FM by changing the initialization value of `send_clock` in `isofm/lcp.c` from 0 to 1.
- Several changes need to be made when testing for different packet sizes because Isotach doesn't fragment messages in its current implementation. First, modify `PACKETSIZE` in `isofm/lcp.h` to a packet size that is a power of 2. Then change `MAX_ISO_MSG_SIZE` in `iso.h` to 20 bytes less than the `PACKETSIZE`. This allows room for the Isotach header in the packet. If for some reason Isotach is being run in paranoid mode, then the Isotach header size changes to 24 bytes and room must be allowed for this in `MAX_ISO_MSG_SIZE`[1]. Finally, change the `-l` option

[1] Since CAST requires a minimum of 44 bytes of data, and the paranoid mode header is 24 bytes, the minimum allowable packet size in paranoid mode is 128 bytes.

on the client's CAST command line to specify the number of bytes to send in each message. If the command line specifies a message size larger than `MAX_ISO_MSG_SIZE`, then the Isotach module automatically overrides the user and brings the buffer size down to the maximum allowed.

- Always recompile CAST after compiling Isotach, as CAST links to the Isotach library.
- The Isotach and FM CAST modules implement a buffering layer. The number of message buffers is statically allocated at compile time with the `NUM_BUFFERS` constant in `cast_isotach.h` and `cast_fm.h`. If these buffers are overrun, CAST will print an error and exit. The user must then increase this constant and recompile.
- The Isotach and FM modules do not implement the handshake or remote channels. These channels must be run over sockets.
- If the token manager or CAST receives a segmentation fault during the FM initialization routines, or if the system slows down to the point where it appears to be page thrashing, there may not be enough available RAM for the system. This can happen with very large packet sizes. One way to fix this is to decrease the `NUM_BUCKETS` and `HIT_BUF_SIZE` constants in `isofm/smm.c`. The user should not do this without intimate knowledge of the Isotach design, as there are minimum values for these constants for the system to operate properly.

# B

# Administrative Guide for Isotach Performance Testing

This appendix has several administrative tips for conducting follow-on performance studies on Isotach that do not belong anywhere else in the document. These bullets are in no particular order.

- Isotach host-to-host mode brings logical time into user space. This mode is good for debugging Isotach, but brings a significant performance penalty to the system. To compile Isotach in host-to-host mode, in the Isotach `Makefile`, remove `tmlcp` from the `OBJS` line and define `HH`.

- The Isotach files used to conduct this performance study are located on the file server in the ~`rgb2u/isofm` directory, and are periodically archived in the `/mnt/uf7/iso-share/rgb2u` directory. The `iso-share` directory is mounted on the department file servers, and backed up daily. However, files must be manually moved to this directory by the account holder.

- The CAST files are located on the file server in ~`rgb2u/cast`. Additionally, they are also archived in `/mnt/uf7/iso-share/rgb2u`. The CAST files are checked into an RCS repository for version control. The CAST files can be archived with `cast/make tar`.

- It is not recommended to use the Isotach paranoid mode. This mode prevents message pipelining, and therefore will not give good performance numbers.

- The Isotach and CAST code used in this performance study contains a lot of instrumentation that can be conditionally compiled into the executable. If `MEASURE_SEND` is defined in the CAST `Makefile`, then the output will contain the average cost of the Isotach or FM send sequence (e.g. `iso_start()`, `iso_send_msg()`, and `iso_end()` in Isotach, `FM_send()` in FM). Defining `FIND_BLOCKING` in the Isotach `Makefile` will output to the screen if the host is blocking due to flow control. Defining `SMM_MEASUREMENT` in both the Isotach and CAST `Makefiles` allows the cost of any piece of code within `isofm/smm.c` to be measured while running CAST. See the current code base in `~rgb2u/isofm` for an example of how to surround the measured code with conditional compilations. Likewise, the same measurements can be made in `isofm/fastmsgs.c` by defining `FM_MEASUREMENT` in both the Isotach and CAST `Makefiles`. If `STRIP_CAST_OVHD` is defined in the CAST `Makefile`, instrumented code will take timestamps closer than CAST to the actual sends and receives, and will output to the screen a more accurate round-trip latency and server turnaround time. The server turnaround time must be subtracted from the client round-trip latency to obtain the actual round-trip latency measurement. It is not recommended to turn on more than one of these conditionally compiled instrumentations at a time, as the instrumentations can impact the system performance.

- The token manager's host will output every five seconds to the terminal the logical clock times of each host and the average token inter-arrival time. The current prototype contains an anomaly in token inter-arrival times. For no apparent reason, Isotach has three inter-arrival times, and the set of three vary with host speed and network configuration. For example, on our most common configuration, the unloaded[1] token inter-arrival time would bounce between 7.62, 11.64, and 11.76 $\mu$s, with 11.64 the most common number. Therefore, we ensured that all tests were run with an unloaded token inter-arrival of 11.64 $\mu$s. This number can affect the performance results slightly, with more effect on latency than bandwidth.

[1]The unloaded token inter-arrival time is the separation between tokens when there is no other network traffic.

# C

# Source Code

## C.1 iso_lat.c

```
/*
 * iso_lat - latency test for Isotach message based model
 */

#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <iso.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <prof.h>

#define COUNT 50    /* # times to run test */
#define CLIENT 1    /* nodeid of client */
#define SERVER 0    /* nodeid of server */
#define BUF_SIZE 44 /* size of data buffer*/
#define HANDLER 22  /* index to message handler */
#define SLEEP 5     /* sleep time while determining speed*/
#define LOOPS 3     /* number of loops to average cpu speed */
/* REPS defined in prof.h */

int client_flag = 0; /* 1 when mesg returned from server */
int server_flag = 0; /* 1 if message arrived at server */
```

```
/* these variables used to measure turn_around time */
long long turnaround_begin;
long long turnaround_end;
long long turnaround_cycles = 0;

char* buf; /* data buffer */

/*
 * this function sends REPS number of
 *  round-trip messages and returns
 */
void time_messages (){
  int x;

  for (x=0; x<REPS; x++) {
    client_flag = 0;
    /* send message to server */
    iso_start();
    iso_send_msg (SERVER, HANDLER, buf, BUF_SIZE);
    iso_end();
    /* wait for the message to return from server */
    while (!client_flag) {
      iso_poll();
    }
  }
}

/*
 * receives REPS * COUNT messages from client
 */
void receive_messages () {
  unsigned int total, x;

  total = REPS * COUNT;
  /* receive "total" messages from client */
  for (x = 0; x < total; x++) {
    server_flag = 0;
    /* wait for next message */
    while (!server_flag) {
      iso_poll();
```

```
    }
    /* turnaround time ends */
    fast_timer (&turnaround_end);
    /* return the message to the client */
    iso_start();
    iso_send_msg (CLIENT, HANDLER, buf, BUF_SIZE);
    iso_end();

    /* add to cumulative turnaround cycles */
    turnaround_cycles += turnaround_end - turnaround_begin;

  }
}

/*
 *  copies data to user space and sets flag signalling
 *  message arrival
 */
void handler (int id, void *data, int len) {

  if (NODEID == SERVER) {
    /* copy the data to user space */
    memcpy( (void*) buf, (const void*) data, BUF_SIZE);
    server_flag = 1; /* signal that message arrived */
    /* begin turnaround time */
    fast_timer (&turnaround_begin);
  }
  else { /* client */
    /* copy the data to user space */
    memcpy( (void*) buf, (const void*) data, BUF_SIZE);
    client_flag = 1; /* signal that message arrived */
  }
}

/*
 *  take a timestamp from CPU cycle counter, sleep,
 *  and take another timestamp.  Do this LOOP times,
 *  and determine CPU speed.
 *  Returns average CPU speed in Hz.
 */
```

```
long get_cpu_speed(){
  long long timer_1, timer_2, cycles;
  long speed;
  int x;

  printf("taking %d seconds to calculate CPU speed for
         the fast_timer measurements\n",
         LOOPS*SLEEP);
  speed = 0;
  /* calculate cpu speed */
  for (x=0; x<LOOPS; x++){
    fast_timer(&timer_1);
    sleep(SLEEP);
    fast_timer(&timer_2);
    cycles = timer_2 - timer_1;
    speed += (cycles / SLEEP);
  }
  /* get the average speed */
  speed /= LOOPS;
  printf("cpu speed is %ld Hz\n", speed);
  fflush (stdout);
  return(speed);
}


void main()
{
  struct timeval start, stop;
  double us, us_sum = 0;
  int i;
  double cum_turnaround_time;
  long cpu_speed;
  double avg_turnaround_time;

  /* initialize the isotach system and set handler */
  iso_init();
  iso_set_handler (HANDLER, handler);

  /* allocate and initialize data buffer */
  buf = (char*) malloc (BUF_SIZE);
```

```
  bzero (buf, BUF_SIZE);

  printf ("I'm node %d\n", NODEID);

  if (NODEID == SERVER) {
    receive_messages ();
    /* calculate and output the average turnaround time */
    cpu_speed = get_cpu_speed ();
    cum_turnaround_time =  ((double)turnaround_cycles *
                            1000000) / (double) cpu_speed;
    avg_turnaround_time = cum_turnaround_time /
      (REPS * COUNT);
    printf ("Average turnaround time is %f us\n",
         avg_turnaround_time);
    printf("done\n");
  }
  else { /* client */
    /* each loop is a test; each test executes REPS #
       of msg round trips; */
    for (i=0; i<COUNT; i++) {
      gettimeofday (&start, NULL);
      time_messages ();
      gettimeofday (&stop, NULL);
      /* calculate and output avg latency for this test */
      subtracttime (&stop, &start); /* defined in prof.h */
      us = (double)stop.tv_sec *1e6 + (double)stop.tv_usec;
      printf ("executed %d round trips in %f usecs;
              %f us/round trip\n", REPS, us, us/REPS);
      fflush (stdout);
      /* add to total number of microseconds */
      us_sum += us;
    }
    /* calculate and output average latency */
    printf("Average round trip latency: %f us\n",
           us_sum/(REPS*COUNT) );
    printf("done\n");
  }
}
```

## C.2 iso_thru.c

```
/*
 * iso_thru - throughput test for Isotach message based model
 */

#include <stdio.h>
#include <sys/time.h>
#include <iso.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <prof.h>

#define CLIENT 1          /* nodeid of client */
#define SERVER 0          /* nodeid of server */
#define BUF_SIZE 1004     /* data buffer size */
#define FIRST_HANDLER 24  /* index to first msg handler */
#define HANDLER 25        /* index to other message handler */
#define DATA_SIZE 40000000/* bytes to push through pipe */
#define ITERATIONS 5      /* times to run the test */

struct timeval start, stop; /* test start and end times */
long received_bytes;        /* number of bytes received */
long sent_bytes;            /* number of bytes received */
char* buf;                  /* data buffer for messages */

/*
 *  sends a fixed amount of data to the server
 */
void send_messages () {

  sleep (5); /* gets rid of race condition between loops*/
  sent_bytes = 0;
  gettimeofday (&start, NULL);/* start the clock */
  /* send first message to start receiver's clock */
  iso_start ();
  iso_send_msg (SERVER, FIRST_HANDLER, (void*) buf, BUF_SIZE);
  iso_end ();
```

```
  sent_bytes += BUF_SIZE;
  /* send until you've sent enough */
  while (sent_bytes < DATA_SIZE) {
    iso_start ();
    iso_send_msg (SERVER, HANDLER, (void*) buf, BUF_SIZE);
    iso_end ();
    sent_bytes += BUF_SIZE;
  }
  gettimeofday (&stop, NULL);/* stop the clock */
}

/*
 *
 */
void receive_messages () {

  received_bytes = 0;
  /* keep extracting messages until we've received enough */
  while (received_bytes < DATA_SIZE) {
    iso_poll();
  }
  gettimeofday (&stop, NULL);/* stop the clock */
}

void handler (int id, void *data, int len) {

  /* copy the data to the user space */
  memcpy( (void*) buf, (const void*) data, BUF_SIZE);
  received_bytes += BUF_SIZE;
}

void first_message_handler (int id, void* data, int len) {

  gettimeofday (&start, NULL);/* start the clock */
  /* copy the data to the user space */
  memcpy( (void*) buf, (const void*) data, BUF_SIZE);
  received_bytes += BUF_SIZE;
}

void main() {
```

```
double sec;               /* number of seconds test took */
double throughput;        /* data rate, in Mbps */

/* used to get average throughput */
double throughput_sum = 0;
double avg_throughput = 0;

int x;

/* allocate and initialize data buffer */
buf = (char*) malloc (BUF_SIZE);
bzero (buf, BUF_SIZE);

/* initialize the system */
iso_init();
iso_set_handler (HANDLER, handler);
iso_set_handler (FIRST_HANDLER, first_message_handler);
printf ("I'm node %d\n", NODEID);
/* run the test a few times */
for (x = 0; x < ITERATIONS; x++) {
  sec = 0;
  throughput = 0;

  if (NODEID == SERVER) {
    receive_messages ();
  }
  else { /* client */
    send_messages ();
  }

  /* defined in prof.h; puts result in stop */
  subtracttime (&stop, &start);
  /* translate to seconds */
  sec = (double)stop.tv_sec + (double)stop.tv_usec / 1e6;

  if (NODEID == SERVER) {
    /* calculate and output server results */
    throughput = ((received_bytes * 8) / sec) / 1e6;
    printf("Received %ld bytes in %f seconds\n",
```

```
     received_bytes, sec);
      printf("Measured receiver throughput for iteration
                %d is %f Mbps\n",
     x, throughput);
    }
    else { /* CLIENT */
      /* calculate and output client results  in Mbps*/
      throughput = ((sent_bytes * 8) / sec) / 1e6;
      printf("Sent %ld bytes in %f seconds\n", sent_bytes,
     sec);
      printf("Measured sender throughput for iteration
                %d is %f Mbps\n",
     x, throughput);
    }
    fflush (stdout);
    throughput_sum += throughput;
  }
  /* calculate and output the overall average of the tests */
  avg_throughput = throughput_sum / ITERATIONS;
  printf("Average %s throughput is %f\n", NODEID ?
 "sender":"receiver", avg_throughput);
}
```

## C.3 iso_contention_lat.c

```
/*
 * iso_contention_lat - measure latency in a
 * contentious network
 */

#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <iso.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <prof.h>
```

```
#define SERVER 0                /* nodeid of server */
#define CLIENT 1                /* nodeid of client */
#define BUF_SIZE 1004           /* data buffer size */
#define FIRST_HANDLER 24        /* index to first msg handler*/
#define HANDLER 25              /* index to other msg handler*/
#define LAST_HANDLER 26         /* index to last msg handler*/
#define CLIENT_MSG_HANDLER 27   /* handle client msgs*/
#define RETURN_TRIP_HANDLER 28 /* handler on return trip*/
#define SECONDS 30              /* number of sec to send*/
#define NUM_CLIENTS 4           /* number of clients sending*/
#define TIMEOUT 0               /* gap between sends in us */
#define SLEEP 5                 /* sleep to determine speed */
#define LOOPS 3                 /* loops to average speed */

struct timeval start, stop, stop_lat;/* hold timing info */
long received_bytes = 0;             /* # bytes received */
long sent_bytes = 0;                 /* # bytes sent*/
int clients_stopped = 0;             /* # clients finished */
long cpu_speed;                      /* in Hz */
char* buf;                           /* data for msg traffic*/
long round_trips = 0;                /* # messages in test */
int client_flag;                     /* 1 if mesg arrived */
int received_from_client = 0;        /* 1 if mesg arrived */

/*
 *  send round-trip packets to server until time runs out
 */
void send_latency_packets () {
  int done = 0; /* 1 if time is up */

  gettimeofday (&start, NULL);/* start the clock */

  while (!done) {
    /* keep sending until time runs out */
    iso_start ();
    iso_send_msg (SERVER, CLIENT_MSG_HANDLER,
  (void*) buf, BUF_SIZE);
    iso_end ();
    /* good nassumption that these calculations
     *are not in critical path
```

```
     * of sending round-trip packets as fast as possible
     */
    sent_bytes += BUF_SIZE;
    gettimeofday (&stop_lat, NULL);
    subtracttime (&stop_lat, &start);
    /* check to see if we have time remaining */
    if (stop_lat.tv_sec >= SECONDS) {
      done = 1;
    }
    /* wait for message to return from server */
    while (!client_flag) {
      iso_poll ();
    }
    client_flag = 0;
  }
  /* stop clock after receipt of last round-trip packet */
  gettimeofday (&stop_lat, NULL);/* stop latency clock */

  /* tell server we are done */
  iso_start ();
  iso_send_msg (SERVER, LAST_HANDLER, (void*) buf, BUF_SIZE);
  iso_end ();
  gettimeofday (&stop, NULL);/* stop throughput time */
  sent_bytes += BUF_SIZE;
}

/* handler for returned packets on client */
void return_trip_handler (int id, void *data, int len) {

  /* copy data to user space */
  memcpy( (void*) buf, (const void*) data, BUF_SIZE);
  /* signal message has arrived */
  client_flag = 1;
  round_trips++;

}

/*
 *  send as quickly as possible to server until time runs out
 */
```

```
void send_messages () {
  long long begin, end; /* timing info */
  double us; /* number of microseconds elapsed */

  /* start the send timer */
  gettimeofday (&start, NULL);

  fast_timer (&begin);/* start the gap timer */
  /* send first message */
  iso_start ();
  iso_send_msg (SERVER, FIRST_HANDLER, (void*) buf, BUF_SIZE);
  iso_end ();
  sent_bytes += BUF_SIZE;
  /* keep sending while time remains */
  do {
    /* gap can be used to quicken or slow the senders */
    /* pause for gap*/
    do {
      fast_timer (&end);
      us = ((double)(end - begin) * 1000000) /
(double) cpu_speed;
    } while (us < TIMEOUT);

    fast_timer (&begin);/* start the gap timer */
    /* send the next message */
    iso_start ();
    iso_send_msg (SERVER, HANDLER, (void*) buf, BUF_SIZE);
    iso_end ();
    sent_bytes += BUF_SIZE;
    gettimeofday (&stop, NULL);
    subtracttime (&stop, &start);/* in prof.h */
  } while (stop.tv_sec < SECONDS);

  /* send last message after send time runs out */
  iso_start ();
  iso_send_msg (SERVER, LAST_HANDLER, (void*) buf, BUF_SIZE);
  iso_end ();
  gettimeofday (&stop, NULL); /* stop the clock */
  sent_bytes += BUF_SIZE;
}
```

```
/*
 *  receive messages until all senders signal they have
 *  sent their final message
 */
void receive_messages () {

  /* receive until all senders have sent their last message */
  while (clients_stopped < NUM_CLIENTS) {
    iso_poll();
    /* if you get a message from the client, return it */
    if (received_from_client) {
      iso_start ();
      iso_send_msg (CLIENT, RETURN_TRIP_HANDLER,
    (void*) buf, BUF_SIZE);
      iso_end ();
      received_from_client = 0;
    }
  }
  gettimeofday (&stop, NULL); /* stop the clock */
}

/*
 *  receive messages from client
 */
void client_msg_handler (int id, void *data, int len) {

  /* copy data to user space */
  memcpy( (void*) buf, (const void*) data, BUF_SIZE);
  received_bytes += BUF_SIZE;
  /* signal that message received from client */
  received_from_client = 1;
}

/*
 *  handler for messages from other senders
 */
void handler (int id, void *data, int len) {

  /* copy data to user space*/
```

```
  memcpy( (void*) buf, (const void*) data, BUF_SIZE);
  received_bytes += BUF_SIZE;
}

/*
 *  receive first message from each client
 */
void first_message_handler (int id, void* data, int len) {

  gettimeofday (&start, NULL);/* start the clock */
  /* copy data to user space */
  memcpy( (void*) buf, (const void*) data, BUF_SIZE);
  received_bytes += BUF_SIZE;
}

/*
 *  receive last message from each sender
 */
void last_message_handler (int id, void* data, int len) {

  /* increment number of senders that have stopped */
  clients_stopped++;
  /* copy data to user space */
  memcpy( (void*) buf, (const void*) data, BUF_SIZE);
  received_bytes += BUF_SIZE;
}

/*
 *  take a timestamp from CPU cycle counter, sleep,
 *  and take another timestamp.  Do this LOOP times,
 *  and determine CPU speed.
 *  Returns average CPU speed in Hz.
 */
long get_cpu_speed(){
  long long timer_1, timer_2, cycles;
  long speed;
  int x;

  printf("taking %d seconds to calculate CPU speed for
         the fast_timer measurements\n",
```

```
 LOOPS*SLEEP);
  speed = 0;
  /* calculate CPU speed */
  for (x=0; x<LOOPS; x++){
    fast_timer(&timer_1);
    sleep(SLEEP);
    fast_timer(&timer_2);
    cycles = timer_2 - timer_1;
    speed += (cycles / SLEEP);
  }
  /* calculate and ouput avg speed */
  speed /= LOOPS;
  printf("cpu speed is %ld Hz\n", speed);
  fflush (stdout);
  return(speed);
}

void main() {

  double sec = 0;
  double throughput = 0;
  double us = 0;

  /* allocate and initialize the data buffer */
  buf = (char*) malloc (BUF_SIZE);
  bzero (buf, BUF_SIZE);

  /* calculate the cpu speed */
  cpu_speed = get_cpu_speed ();

  /* initialize the system */
  iso_init();
  iso_set_handler (HANDLER, handler);
  iso_set_handler (FIRST_HANDLER, first_message_handler);
  iso_set_handler (LAST_HANDLER, last_message_handler);
  iso_set_handler (RETURN_TRIP_HANDLER, return_trip_handler);
  iso_set_handler (CLIENT_MSG_HANDLER, client_msg_handler);
  printf ("I'm node %d\n", NODEID);

  if (NODEID == SERVER) {
```

```
  receive_messages ();
}
else if (NODEID == CLIENT) {
  send_latency_packets ();
}
else { /* SENDERS */
  send_messages ();
}
/* defined in prof.h; puts result in stop */
subtracttime (&stop, &start);
/* translate into seconds */
sec = (double)stop.tv_sec + (double)stop.tv_usec / 1e6;

if (NODEID == SERVER) {
  /* calculate and output the throughput */
  throughput = (((double)(received_bytes) * 8)
/ sec) / 1e6; /* in Mbps */
  printf("Received %ld bytes in %f seconds\n",
 received_bytes, sec);
  printf("Measured receiver throughput is %f Mbps\n",
 throughput);
}
else if (NODEID == CLIENT) {
  /* calculate and output the throughput */
  throughput = (((double)(sent_bytes) * 8)
/ sec) / 1e6; /* in Mbps */
  printf("Sent %ld bytes in %f seconds\n", sent_bytes,
 sec);
  printf("Measured client throughput is %f Mbps\n",
 throughput);
  /* calculate and output the average latency */
  subtracttime (&stop_lat, &start);
  us = (double)stop_lat.tv_sec *1e6 +
    (double)stop_lat.tv_usec;
  printf ("executed %ld round trips in %f usecs;
           %f us/round trip\n",
  round_trips, us, us/round_trips);
}
else { /* SENDERS */
  /* calculate and output the throughput */
```

```
    throughput = (((double)(sent_bytes) * 8) / sec) / 1e6;
    printf("Sent %ld bytes in %f seconds\n", sent_bytes, sec);
    printf("Measured sender throughput is %f Mbps\n",
   throughput);
  }
}
```

## C.4 iso_contention_thru.c

```
/*
 * iso_contention_thru - measure throughput in
 * a contentious network
 */

#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <iso.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <prof.h>

#define SERVER 0        /* nodeid of server */
#define BUF_SIZE 1004   /* data buffer size */
#define FIRST_HANDLER 24/* index to first handler */
#define HANDLER 25      /* index to other message handler */
#define LAST_HANDLER 26 /* index to last message handler */
#define SECONDS 30      /* number of seconds for hosts to */
#define NUM_CLIENTS 3   /* number of clients sending */
#define TIMEOUT 0       /* gap between sends in us */
#define SLEEP 5         /* sleep time while getting speed */
#define LOOPS 3         /* number of loops to average speed */

struct timeval start, stop; /* timing info */
long received_bytes = 0;    /* # bytes received thus far */
long sent_bytes = 0;        /* # bytes sent thus far */
int clients_stopped = 0;    /* # senders finished */
long cpu_speed;             /* in Hz */
```

```
char* buf;                  /* data buffer for messages */

/*
 *  keep sending messages to server until time runs out
 */
void send_messages () {
  long long begin, end;
  double us;

  /* start the send timer */
  gettimeofday (&start, NULL);

  fast_timer (&begin);/* start gap timer */
  /* send the first message */
  iso_start ();
  iso_send_msg (SERVER, FIRST_HANDLER, (void*) buf, BUF_SIZE);
  iso_end ();
  sent_bytes += BUF_SIZE;
  /* send messages while time remains */
  do {
    /* can use gap to turn up or down the senders */
    /* pause for gap*/
    do {
      fast_timer (&end);
      us = ((double)(end - begin) * 1000000) /
(double) cpu_speed;
    } while (us < TIMEOUT);

    fast_timer (&begin);/* start gap timer */
    /* send next message */
    iso_start ();
    iso_send_msg (SERVER, HANDLER, (void*) buf, BUF_SIZE);
    iso_end ();
    sent_bytes += BUF_SIZE;
    gettimeofday (&stop, NULL);
    subtracttime (&stop, &start);
  } while (stop.tv_sec < SECONDS);/* if time remaining */

  /* send last message after send time runs out */
  iso_start ();
```

```
  iso_send_msg (SERVER, LAST_HANDLER, (void*) buf, BUF_SIZE);
  iso_end ();
  gettimeofday (&stop, NULL);/* stop the clock */
  sent_bytes += BUF_SIZE;
}

/*
 * keep receiving until all clients have stopped sending
 */
void receive_messages () {

  /* receive until all clients have sent their last message */
  while (clients_stopped < NUM_CLIENTS) {
    iso_poll();
  }
  gettimeofday (&stop, NULL);/* stop the clock */
}

/*
 *  receive messages
 */
void handler (int id, void *data, int len) {

  /* copy data to user space */
  memcpy( (void*) buf, (const void*) data, BUF_SIZE);
  received_bytes += BUF_SIZE;
}

/*
 *  receive first message and start the clock
 */
void first_message_handler (int id, void* data, int len) {

  gettimeofday (&start, NULL);/* start the clock */
  /* copy the data to user space */
  memcpy( (void*) buf, (const void*) data, BUF_SIZE);
  received_bytes += BUF_SIZE;
}

/*
```

```
 *  receive last message from each sender
 */
void last_message_handler (int id, void* data, int len) {

  /* increment the number of senders finished */
  clients_stopped++;
  /* copy the data to user space */
  memcpy( (void*) buf, (const void*) data, BUF_SIZE);
  received_bytes += BUF_SIZE;
}

/*
 *  take a timestamp from CPU cycle counter, sleep,
 *  and take another timestamp.  Do this LOOP times,
 *  and determine CPU speed.
 *  Returns average CPU speed in Hz.
 */
long get_cpu_speed(){
  long long timer_1, timer_2, cycles;
  long speed;
  int x;

  printf("taking %d seconds to calculate CPU speed for
          the fast_timer measurements\n",
 LOOPS*SLEEP);
  speed = 0;
  /* calculate CPU speed */
  for (x=0; x<LOOPS; x++){
    fast_timer(&timer_1);
    sleep(SLEEP);
    fast_timer(&timer_2);
    cycles = timer_2 - timer_1;
    speed += (cycles / SLEEP);
  }
  /* calculate and ouput average */
  speed /= LOOPS;
  printf("cpu speed is %ld Hz\n", speed);
  fflush (stdout);
  return(speed);
}
```

```
void main() {

  double sec = 0;
  double throughput = 0;

  /* allocate and initialize data buffer */
  buf = (char*) malloc (BUF_SIZE);
  bzero (buf, BUF_SIZE);

  /* calculate the cpu speed */
  cpu_speed = get_cpu_speed ();

  /* initialize the system */
  iso_init();
  iso_set_handler (HANDLER, handler);
  iso_set_handler (FIRST_HANDLER, first_message_handler);
  iso_set_handler (LAST_HANDLER, last_message_handler);
  printf ("I'm node %d\n", NODEID);

  if (NODEID == SERVER) {
    receive_messages ();
  }
  else { /* clients */
    send_messages ();
  }
  /* determine number of seconds */
  /* defined in prof.h; puts result in stop */
  subtracttime (&stop, &start);
  sec = (double)stop.tv_sec + (double)stop.tv_usec / 1e6;

  if (NODEID == SERVER) {
    /* calculate and ouput the throughput */
    throughput = (((double)(received_bytes) * 8) / sec) / 1e6;
    printf("Received %ld bytes in %f seconds\n",
   received_bytes, sec);
    printf("Measured receiver throughput is %f Mbps\n",
   throughput);
  }
  else { /* CLIENT */
```

```
    /* calculate and output the throughput */
    throughput = (((double)(sent_bytes) * 8) / sec) / 1e6;
    printf("Sent %ld bytes in %f seconds\n", sent_bytes, sec);
    printf("Measured sender throughput is %f Mbps\n",
   throughput);
  }
}
```

## C.5 isochron.c

```
/*
 * isochron - measure bandwidth while changing size
 * of isochron; each isochron sends 1 message to a
 * different host
 */

#include <stdio.h>
#include <sys/time.h>
#include <iso.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <prof.h>

#define SENDER 0            /* nodeid of sender */
#define ISOCHRON_SIZE 1     /* number of isochron mesgs */
#define BUF_SIZE 44         /* data buffer size */
#define FIRST_HANDLER 24    /* index to first mesg handler*/
#define HANDLER 25          /* index to other mesg handler*/
#define DATA_SIZE 40000000  /* bytes to send in test */
#define ITERATIONS 5        /* number of tests*/

struct timeval start, stop; /* timing info */
long received_bytes;        /* # bytes received thus far */
long sent_bytes;            /* # bytes sent thus far */
char* buf;                  /* data buffer */

/*
 *  send a fixed amount of data in isochrons
```

```
 */
void send_messages () {
  int x;

  sleep (5); /* gets rid of race condition*/
  sent_bytes = 0;
  gettimeofday (&start, NULL);/* start the clock */
  /* send the first message to each receiver */
  iso_start ();
  for (x = 1; x <= ISOCHRON_SIZE; x++) {
    iso_send_msg (x, FIRST_HANDLER, (void*) buf, BUF_SIZE);
  }
  iso_end ();
  sent_bytes += BUF_SIZE * ISOCHRON_SIZE;
  /* keep sending until we've sent enough */
  while (sent_bytes < DATA_SIZE ) {
    /* send a message to each receiver */
    iso_start ();
    for (x = 1; x <= ISOCHRON_SIZE; x++) {
      iso_send_msg (x, HANDLER, (void*) buf, BUF_SIZE);
    }
    iso_end ();
    sent_bytes += BUF_SIZE * ISOCHRON_SIZE;
  }
  gettimeofday (&stop, NULL);/* stop the clock */
}

/*
 *  receive messages from sender
 */
void receive_messages () {
  double amount_to_receive;

  /* calculate how much data we should receive so we know
   * when to stop
   */
  amount_to_receive = DATA_SIZE / ISOCHRON_SIZE;

  received_bytes = 0;
  /* keep receiving while there's data to receive */
```

```
  while (received_bytes < amount_to_receive) {
    iso_poll();
  }
  gettimeofday (&stop, NULL);/* stop the clock */
}

/* handle incoming messages */
void handler (int id, void *data, int len) {

  /* copy the data to user space */
  memcpy( (void*) buf, (const void*) data, BUF_SIZE);
  received_bytes += BUF_SIZE;
}

/*
 *  receive the first message to start the clock
 */
void first_message_handler (int id, void* data, int len) {
  /* start the clock */
  gettimeofday (&start, NULL);
  /* copy the data to user space */
  memcpy( (void*) buf, (const void*) data, BUF_SIZE);
  received_bytes += BUF_SIZE;
}

void main() {

  double sec;
  double throughput;
  double throughput_sum = 0;
  double avg_throughput = 0;

  int x;

  /* allocate and initialize the data buffer */
  buf = (char*) malloc (BUF_SIZE);
  bzero (buf, BUF_SIZE);

  /* initialize the system */
  iso_init();
```

```
iso_set_handler (HANDLER, handler);
iso_set_handler (FIRST_HANDLER, first_message_handler);
printf ("I'm node %d\n", NODEID);

/* run the test a fixed number of times */
for (x = 0; x < ITERATIONS; x++) {
  sec = 0;
  throughput = 0;

  if (NODEID == SENDER) {
    send_messages ();
  }
  else { /* client */
    receive_messages ();
  }
  /* determine how much time test took */
  /* defined in prof.h; puts result in stop */
  subtracttime (&stop, &start);
  sec = (double)stop.tv_sec + (double)stop.tv_usec / 1e6;

  if (NODEID != SENDER) {
    /* calculate and ouput throughput */
    throughput = ((received_bytes * 8) / sec) / 1e6;
    printf("Received %ld bytes in %f seconds\n",
   received_bytes, sec);
    printf("Measured receiver throughput for iteration
           %d is %f Mbps\n",
   x, throughput);
  }
  else { /* SENDER */
    /* calculate and output throughput */
    throughput = ((sent_bytes * 8) / sec) / 1e6;
    printf("Sent %ld bytes in %f seconds\n", sent_bytes,
   sec);
    printf("Measured sender throughput for iteration %d is
           %f Mbps\n",
   x, throughput);
  }
  fflush (stdout);
  /* increment throughput total used for averaging */
```

```
    throughput_sum += throughput;
  }
  /* calculate and output average throughput*/
  avg_throughput = throughput_sum / ITERATIONS;
  printf("Average %s throughput is %f\n",
 NODEID ? "receiver":"sender", avg_throughput);
}
```

# Bibliography


[1] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet – A Gigabit-per-Second Local-Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[2] R.A. Dirvin and A.R. Miller. The MC68824 Token Bus Controller: VLSI for the Factory LAN. *IEEE Micro Magazine*, 6:15–25, June 1986.

[3] Philip M. Irey IV, Robert D. Harrison, and David T. Marlow. Techniques for LAN Performance Analysis in a Real-Time Environment. *Real-Time Systems - International Journal of Time Critical Computing Systems*, 14(1), January 1998.

[4] Jonathan Kay and Joseph Pasquale. The Importance of Non-Data Touching Processing Overheads in TCP/IP. In *SIGCOMM93*, pages 259–268, 1993.

[5] Kimberly K. Keeton, Thomas E. Anderson, and David A. Patterson. LogP Quantified: The Case for Low-Overhead Local Area Networks. In *Proceedings of Hot Interconnects III: A Symposium on High Performance Interconnects*, August 1995.

[6] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[7] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture ISCA-97*, volume 25. ACM Press, June 1997.

[8] R.M. Metcalfe and D.R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19:395–404, July 1976.

[9] David .L. Mills. Algorithms for Synchronizing Network Clocks. Technical Report IETF RFC 956, M/A-COM Linkabit, September 1985.

[10] David .L. Mills. Network Time Protocol (Version 3) Specification, Implementation and Analysis. Technical Report IETF RFC 1305, University of Delaware, March 1992.

[11] Naval Surface Warfare Center, Dahlgren Division. *User Manual for Communications Analysis and Simulation Tool*, July 1998.

[12] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 1997.

[13] Scott Pakin, Mario Lauria, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. *Supercomputing*, December 1995.

[14] John Regehr. An Isotach Implementation for Myrinet. Technical report, University of Virginia, May 1997.

[15] Paul F. Reynolds, Jr., Craig Williams, and Raymond R. Wagner, Jr. Isotach Networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(4), April 1997.

[16] F.E. Ross and J.R. Hamstra. Forging FDDI. *IEEE Journal on Selected Areas in Communication*, 11:181–190, February 1993.

[17] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the International Symposium on Computer Architecture*, 1992.

[18] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 40–53, December 1995.

[19] Thorsten von Eicken and Werner Vogels. Evolution of the Virtual Interface Architecture. *IEEE Computer*, pages 61–76, November 1998.

[20] Craig C. Williams. *Concurrency Control in Asynchronous Computations*. PhD thesis, University of Virginia, January 1993.

[21] Craig C. Williams and Paul F. Reynolds, Jr. Fault-Tolerant Isotach Systems. Internal working paper.