An Isotach Implementation for Myrinet

John Regehr

Technical Report CS-97-12
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
Email: regehr@virginia.edu

May 16, 1997

Contents

1 Introduction 1

2 Background 3

2.1 Isotach Logical Time 3

2.1.1 Definition 3

2.1.2 Implementation 5

2.2 The DARPA Prototype 6

2.3 Myrinet 7

2.4 Fast Messages 8

2.5 Linux 11

2.6 The Prototype Network 12

3 Implementation 13

3.1 User Programs' View of Isotach 13

3.2 The Shared Memory Manager 15

3.2.1 Design Issues 16

3.2.2 Data Structures 17

3.2.3 Message Types 18

3.2.4 Issuing an Isochron 19

3.2.5 Executing Isotach Operations 20

3.2.6 Implementation of Isotach Operations 21

3.2.7 Isochronous Messages 22

3.3 The Switch Interface Unit 22

3.4 An Example . . . 26
3.5 The Token Manager . . . 27
3.6 The Front End . . . 28
3.7 Implementation Difficulties . . . 28
3.7.1 Porting Fast Messages . . . 28
3.7.2 Debugging LANai Code . . . 29

4 Correctness and Performance 31
4.1 Correctness . . . 31
4.2 Performance . . . 32
4.2.1 Read Latency . . . 33
4.2.2 Isochron Throughput . . . 36
4.2.3 Rate of Progress of Logical Time . . . 37

5 Summary and Future Plans 39

A Example Isotach Programs 41
A.1 lat.c . . . 41
A.2 thru.c . . . 46
A.3 philo.c . . . 48

B The Isotach API 53
B.1 Introduction . . . 53
B.1.1 Conventions . . . 53
B.2 Isotach configuration files . . . 54
B.2.1 fmconfig . . . 54
B.2.2 shmem_map . . . 54
B.3 Isotach library routines . . . 54
B.3.1 iso_init() . . . 54
B.3.2 iso_deinit() . . . 55
B.3.3 iso_start() . . . 55
B.3.4 iso_end() . . . 56

B.3.5 iso_read32() 56

B.3.6 iso_read64() 56

B.3.7 read_iso_var32() 57

B.3.8 read_iso_var64() 57

B.3.9 iso_write32() 58

B.3.10 iso_write64() 58

B.3.11 iso_sched() 58

B.3.12 iso_assign32() 59

B.3.13 iso_assign64() 59

B.3.14 iso_poll() 60

B.3.15 iso_send_msg() 60

B.3.16 iso_set_handler() 61

B.4 Isotach constants and system variables 61

B.4.1 NODEID 61

B.4.2 NUMNODES 61

B.4.3 MAX_ISO_MSG_SIZE 62

B.4.4 MAX_ISO_HANDLERS 62

B.5 Isochronous Messages 62

B.6 Example isotach programs 63

B.6.1 Skeleton isotach program 63

B.6.2 Dining philosophers 63

B.6.3 LU decomposition 63

B.7 The FM 1.1 configuration file 63

List of Figures

2.1 Access sequence for an isotach variable. 5
2.2 Alternative configurations of the prototype network. 12

3.1 Structure of an isotach node. 14
3.2 Schematic diagram of the SMM. 15
3.3 Schematic diagram of the SIU. 23
3.4 Token paths. 27

4.1 Timing analysis of a remote read. 34

1

Introduction

An isotach network provides strong guarantees about message delivery order. We show that an isotach network can be implemented efficiently entirely in software, using commercial off-the-shelf hardware. This report describes that effort. Parts of this implementation could be performed much more efficiently in hardware; we are currently developing custom hardware components to do this. The all-software version then serves several purposes:

- to develop a working isotach system rapidly, for use as a platform for development of higher level software
- to find potential problems in the hardware design
- to pre-test software components of the system so that they will not have to be debugged at the same time as hardware
- to achieve as much performance as possible without hardware acceleration

The implementation was successful; it works, and performs well. A number of features that will be present in the hardware version are not yet implemented; they will be added in the near future.

2

Background

2.1 Isotach Logical Time

2.1.1 Definition

An *isotach network* [4, 16] is designed to reduce synchronization overhead in parallel programs by providing strong guarantees about the order in which messages are delivered. These guarantees allow us to enforce *atomicity* and *sequential consistency* over the operations performed by a program. A group of operations is executed atomically if the operations all appear to be executed at the same time, and an execution is sequentially consistent if the operations issued by each process appear to be performed in the order in which they were issued.

Lamport [5] proposed logical time as a way to represent the ordering of events in a distributed system. Isotach logical times are an extension of Lamport's logical times; they are ordered n-tuples of non-negative integers. The first component is the *pulse*, followed by *pid* and *rank*. Times are ordered lexicographically, with pulse being the most significant component. In networks that are point-to-point FIFO, the rank component is unnecessary and may be omitted.

The key guarantee provided by an isotach network is the *isotach invariant*. It states that a message sent at time (i, j, k) will be received at time $(i + \delta, j, k)$, where δ is the logical distance between the sender and receiver. In other words, messages have a velocity of one unit of distance per *pulse*. A pulse is a unit of logical time. Assuming distances are known, a process can predict the receive time of messages it sends. This is the basis of isotach concurrency control techniques.

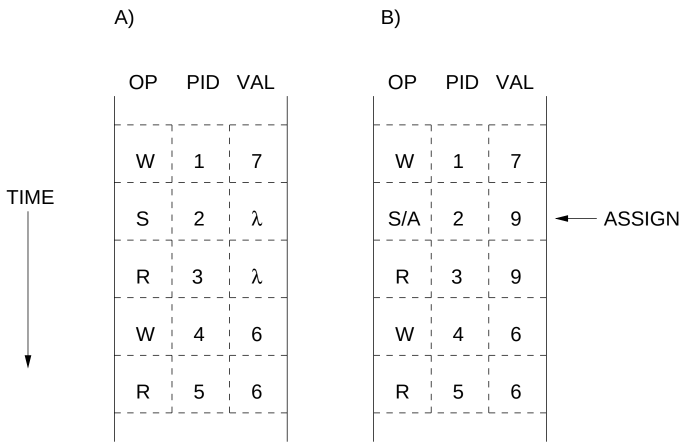
An *isochron* is a *totally ordered*, *issue consistent* multicast, i.e. the multicast is received in a consistent order at all destinations, and that order is consistent with isochron issue order. The operations in an isochron appear to be executed atomically. To issue an isochron, the isotach runtime system converts the isochron into messages, and times the sending of the messages so that all components will be received during the same pulse. The receivers execute all messages received during a pulse in $(pid, rank)$ order. This ensures sequential consistency. If the triangle inequality holds for logical distances, then message delivery is *causal*.

Isochrons that only read from and write to memory are *flat*— they contain no internal data dependencies. The SCHED and ASSIGN operations allow *structured* atomic actions to be performed. A SCHED reserves the capability to write to a memory location at the logical time that the SCHED is executed, without specifying the value to be written. Issuing an ASSIGN that *corresponds* to the SCHED fills in the value. An isochron that contains SCHEDs effectively reserves a consistent slice of logical time, and can fill in the values at a later time. Between the times that the SCHED and ASSIGN are executed, the value of a memory location is *unsubstantiated*, and attempts to read it are blocked.

We can represent an *isotach variable*, the smallest addressable unit of isotach memory, as an *access sequence*. An access sequence is the sequence of operations performed on the variable over time. For a non-isotach variable, the only available value is the value most recently written; SCHEDs and ASSIGNs make access sequences of isotach variables more interesting.

Figure 2.1 shows an example of an isotach variable's access sequence at two different times. Unsubstantiated values are represented by λ. The notation we use for isotach operations is $\mathrm{OP}(pid, v)$, where OP is the operation being performed, *pid* is the pid of the issuing process, and v is the value to be written, or the variable to have a value read into.

The operations that have been performed in A) are: $\mathrm{WRITE}(1, 7)$, $\mathrm{SCHED}(2, -)$, $\mathrm{READ}(3, x)$, $\mathrm{WRITE}(4, 6)$, $\mathrm{READ}(5, y)$. The READ issued by process 3 is blocked,



A)

OP	PID	VAL
W	1	7
S	2	λ
R	3	λ
W	4	6
R	5	6

B)

OP	PID	VAL
W	1	7
S/A	2	9
R	3	9
W	4	6
R	5	6

Figure 2.1: Access sequence for an isotach variable.

but the READ issued by process 5 has completed because it follows a WRITE. In B), the operation ASSIGN$(2,9)$ has been performed; this has unblocked the read by process 3. Operations in the access sequence past the first WRITE following the SCHED are unaffected by the ASSIGN.

2.1.2 Implementation

Assume a network of arbitrary topology, connected by switches. Every node has a switch interface unit (SIU) that mediates its access to the network. In the *isonet* algorithm for implementing isotach, adjacent switches and SIUs are loosely synchronized by exchanging *tokens*. Tokens separate pulses of logical time; when an SIU receives the token for pulse i, any messages that it receives in the future are guaranteed to have pulse greater than i. When a switch has received token i from all inputs, it sends token $i+1$ to all outputs. An SIU behaves in the same way, although it has only one input and one output. During a pulse, a switch always handles the available message with the smallest logical time. SIUs process the outgoing or incoming message with the lowest tag.

The isonet algorithm is easy to understand, but would be difficult to implement because it requires many changes to the network hardware. This report describes the implementation of an equivalent algorithm using only off-the-shelf components. In this equivalent algorithm, we avoid delaying messages either at the source, or as they pass through the network. Instead, messages traverse the network as quickly as possible and are buffered at the receiving node until the correct logical time for their execution is reached. We continue using tokens to separate pulses.

2.2 The DARPA Prototype

We are building an isotach system under a DARPA contract, using Myrinet, a switched gigabit network. Three components implement the isotach system: the SIU, the shared memory manager (SMM), and the token manager (TM). In version 1 (V1) of the prototype, described in this report, all components are implemented in software. In version 2 (V2), the TM and SIU will be realized as hardware; the SMM will be a software descendant of the V1 SMM.

The SIU maintains isotach logical time; it exchanges tokens with a TM, timestamps outgoing messages with the correct receive time, and notifies the SMM of important logical times. The V1 SIU is implemented in software running on a Myrinet board. The V2 SIU will be a custom hardware device located *in-link*, i.e., between the Myrinet interface board and the switch to which the interface is connected.

Every switch in the isotach network has a TM associated with it. The TM exchanges tokens with all hosts attached to its switch, and with all TMs attached to switches directly connected to its switch. In other words, it exchanges tokens with all hosts of distance one and all TMs of distance two. Like the V1 SIU, the V1 TM is implemented in software on a Myrinet board. The host whose Myrinet interface is being used as a token manager is effectively a life-support system for the TM, and does not participate in the isotach network. The V2 TM will be a custom hardware device attached to a switch port.

The SMM is responsible for maintaining isotach shared memory. It receives isochrons from its host and splits them into local and remote components; the remote component is sent to the SIU for delivery onto the network. Local operations, along with remote requests received from the SIU, are executed at the correct logical time.

2.3 Myrinet

Myrinet [2] is a local area network based on technology developed for massively parallel processors. Distinguishing features of Myrinet are:

- high bandwidth and low latency
- very low error rate
- cut-through routing
- arbitrary topology
- flow control on all links
- programmable network interfaces

A Myrinet link is a full-duplex pair of 1.28 Gb/s point-to-point channels. A channel transmits 160 million 9-bit *flits* per second. Each flit is either a byte of data or a control symbol. Control symbols are used for flow control, to mark packet boundaries, and to indicate error conditions. The bit error rate of a Myrinet link is below 10^{-15}. [1]

A Myrinet switch is a crossbar; current models have 4 or 8 ports, and 16-port switches should be available in 1997. Routing is cut-through: as a packet arrives, the switch immediately directs it to the correct output port. The output port is determined by decoding the first flit of the message and interpreting it as an offset from the message's input port. The switch consumes the route flit; this establishes the invariant that at each switch the first flit of a message indicates the correct output port.

If the output port required by a message is being used, the switch uses flow control to block the incoming message. Every link has flow control, so the sender will eventually be blocked if the output port remains unavailable. This protocol ensures that the network never drops a message. Deadlock is possible in a cyclic network; deadlock avoidance or

[1] This figure, from [2], applies to cables up to 25 m long using the slow (80 Mhz) Myrinet protocol. There is no reason to believe that the error rate is any higher for the current protocol, which runs at 160 Mhz over cables up to 10 m long.

recovery is the responsibility of higher network layers. An algorithm to map a Myrinet network, and to construct mutually deadlock-free routes between hosts, can be found in [8].

Myrinet is a switched point-to-point network rather than a bus such as Ethernet or FDDI. For reasonable topologies, aggregate bandwidth scales with network size because packets may be using many channels concurrently. An 8-port switch has a bisection bandwidth of more than 10 Gb/s.

A Myrinet host interface is controlled by a general purpose microprocessor called a LANai. The LANai executes a Myrinet control program (MCP) that is stored in a block of high performance SRAM. Additionally, the SRAM is used to buffer incoming and outgoing messages that go through the LANai's send and receive DMA engines. The SRAM is accessible to the host through programmed I/O, and a third DMA engine allows the LANai to transfer data between the SRAM and the host computer's main memory.

Because the LANai is relatively slow (30–40 MHz), use of DMA is crucial to a high performance MCP. It is also important to avoid loading the LANai with functionality that could be implemented by the host. [2]

We decided to use Myrinet because it is a very fast, scalable network. Also, Myricom provides source code for all Myrinet software, and has worked with us to resolve problems. [1] compares Myrinet with ATM and Fast Ethernet in the context of parallel cluster computing.

2.4 Fast Messages

Historically, networks such as Ethernet have been slow enough that the overhead associated with the TCP/IP stack has not been a limiting factor in communication bandwidth or latency. However, Myrinet is fast enough that protocol processing and related data copying can easily become performance bottlenecks. A common approach to avoid this overhead is to use a user-level network layer, which resides in the same address space as the user program and accesses the network hardware directly. This keeps the operating system out of the critical path for performance, and allows the protocol be tailored to the specific

[2] The LANai 5.0, currently under development, will be much less of a bottleneck than current LANai processors. It runs at 66 Mhz and has 64-bit internal data paths.

network and application.

Illinois Fast Messages version 1.1 [11, 12] is a high performance user-level messaging layer for Myrinet and the Cray T3D. It provides reliable, in-order delivery of messages, as well as buffering to decouple the network from the processor. Because Fast Messages (FM) is a purely user-level protocol, the application must poll the network for new messages. Polling adds overhead even when no messages are arriving and can be inconvenient, but has the advantage of allowing the application to chose when messages are received. This helps avoid cache pollution and eliminates most synchronization concerns — message *handlers* are atomic. The handler for a message is a function specified by the sender that is executed by the receiver when it polls the network after a message has arrived. Good discussions comparing polling and interrupts can be found in [3] and [6].

Recent versions of FM have no facility for packet retransmission; they assume that the network will never drop or corrupt a packet. In general, this is a safe assumption for Myrinet. FM must still provide flow control, because even if the network is reliable, multiple senders can overwhelm a receiver with data and force it to drop packets due to lack of buffer space. It is inappropriate to use Myrinet flow control to prevent receive buffers from overflowing; this could block packets inside the network for relatively long periods of time, causing other routes to become unavailable.

FM implements a sender-based flow control protocol. Every host in the network manages a block of receive buffer space on every other host. This allows a sender to determine when to stop sending data to a receiver, so packets will never be dropped even if the receiver does not poll the network for long periods of time. Because buffers for different nodes are independent, the many-to-many flow control problem is reduced to the point-to-point case. Packets coming from the network are held on the Myrinet board for as short a time as possible. As soon as the LANai-to-host DMA engine is available, it is used to transfer messages into host receive buffers. Receive buffers must be located in DMA-able memory; on our hardware this means that it must be non-pageable and physically contiguous.

After the receiver consumes a packet by executing its handler, a credit is sent to the sender to inform it of the newly available buffer space. Credits are aggregated and piggybacked onto normal messages, so no explicit credit messages are sent when traffic is reasonably bidirectional.

When there is not enough credit to perform a send, the send is blocked and FM enters a polling loop until credit for the receiver becomes available. Consider the case in which an FM handler attempts to send a message, and no send credit is available. The only way to get the credit necessary to complete the send is to process messages later in the incoming message stream than the message whose handler is currently running. This violates the FIFO delivery guarantee provided by FM. Therefore, handlers are *not* allowed to send messages. [3]

Although sender-based flow control is simple and fast, it uses buffer space inefficiently because it pessimistically allocates the same amount of memory to each host, regardless of actual communication patterns. In order to achieve reasonable performance, the non-pageable receive buffer space allocated to FM must increase linearly with network size. This means that Fast Messages will not scale gracefully to networks containing hundreds or thousands of nodes.

Fast Messages 1.1 achieves a one way message latency of about 11 μs for small messages, and a bandwidth of over 250 Mb/s for large messages. Although this is well below the peak bandwidth of Myrinet, for the small messages that isotach uses FM is competitive with all other messaging layers that we considered. Because it has very low overhead per message, the message half-power point (the message size at which half of the peak bandwidth is achieved) for FM is quite low.

In summary, FM was chosen because it is a high performance network layer for Myrinet that provides strong message delivery guarantees. Also, it is available in source form, and the code is well written and easy to modify. Other messaging layers under consideration were Berkeley's Active Messages (AM) [15] and Myricom's Myrinet API [10]. AM is similar to FM in many ways, although it does not guarantee in-order delivery of messages; the decision to use FM rather than AM was therefore somewhat arbitrary. We rejected the Myricom API because it does not guarantee reliable or in-order delivery, and is significantly slower than FM and AM.

[3] Handlers in Fast Messages 2.0 are allowed to send messages, but must be prepared for the send to fail if there is insufficient credit.

2.5 Linux

Linux [7] is an implementation of Unix that is being developed cooperatively over the Internet. It provides virtual memory, preemptive multitasking, TCP/IP networking, and most other features that are expected of a modern Unix. Linux 2.0 runs on a variety of platforms including the x86, Alpha, Sparc, and PowerPC. Several features of Linux make it an especially suitable base for the isotach system:

- source code is freely available under terms of the GNU Public License
- a complete and easy-to-install Linux distribution is available
- it supports symmetric multiprocessor machines
- a Myrinet device driver is available

Our development machines run Red Hat Linux, a commercial distribution that is available over the Internet. It includes a package manager, system administration tools, and a sufficient variety of software packages that little or nothing has to be downloaded and installed by hand.

Symmetric multiprocessor machines based on x86 CPUs are becoming increasingly popular, and provide a cheap way to increase the performance of a PC. Linux supports up to 16 CPUs, although its naive kernel locking policy would cause machines that large to perform poorly. [4] It works very well for dual processor machines.

Since it uses user-level networking, the isotach prototype's dependence on Linux is minimal. It could be easily ported another operating system for which a Myrinet driver is available. The operating systems for PC hardware that Myricom currently supports are Linux, Solaris x86, BSDi, and OSF-1. Windows NT support will be available soon.

An important reason for choosing Linux is simply that it works well. It is powerful and efficient, and supports a wide variety of PC hardware. If development kernels and cheap hardware are avoided, Linux is very stable. Generally, bugs are fixed quickly once they become known to the developers.

[4] Linux 2.1, currently under development, is beginning to support the fine-grained kernel locking that is necessary to perform well on large multiprocessors.

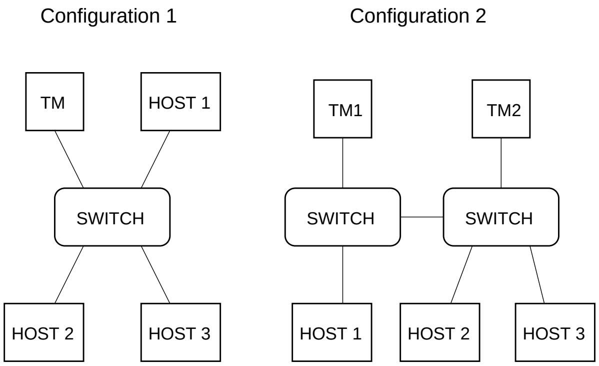


Figure 2.2: Alternative configurations of the prototype network.

2.6 The Prototype Network

Our network (figure 2.2) contains three dual-processor 180 Mhz Pentium Pros and two uniprocessor 166 Mhz Pentiums, each equipped with a Myrinet interface. With two 8-port switches, we have two interesting topologies:

1. Three nodes and one token manager connected to one switch.

2. Two nodes and a TM on one switch and one node and a TM on the other.

We are in the process of acquiring three more dual-processor Pentium Pros, to expand our network to eight machines.

3

Implementation

Together, the shared memory manager (SMM) and switch interface unit (SIU) provide a host's interface to the isotach network. A number of isotach hosts and token managers (TMs) connected with switches implement an isotach network. This chapterdescribes the implementation of the SIU, SMM, and TM in detail. Figure 3.1 gives a high-level view of an isotach node.

3.1 User Programs' View of Isotach

To an application, the isotach system appears to be a large memory array with special access semantics. A node may perform READ, WRITE, SCHED, and ASSIGN operations on memory locations; operations grouped into isochrons are performed atomically.

Isotach operations are non-blocking; this allows pipelining within the network. An issuing node is blocked only when a value requested in a read is actually used. In other words, only true data dependencies stall a processor. If an application is able to issue reads far enough ahead of when the values are needed, then all network latency is hidden and the computation is never blocked.

Isotach memory is divided into pages. Each page has a *copyset*—the set of nodes that have a local copy of that page. Pages are composed of isotach *variables*, the smallest addressable units of memory. The page location of an isotach variable does not affect its semantics, but may impact performance. A READ to an isotach variable goes to the

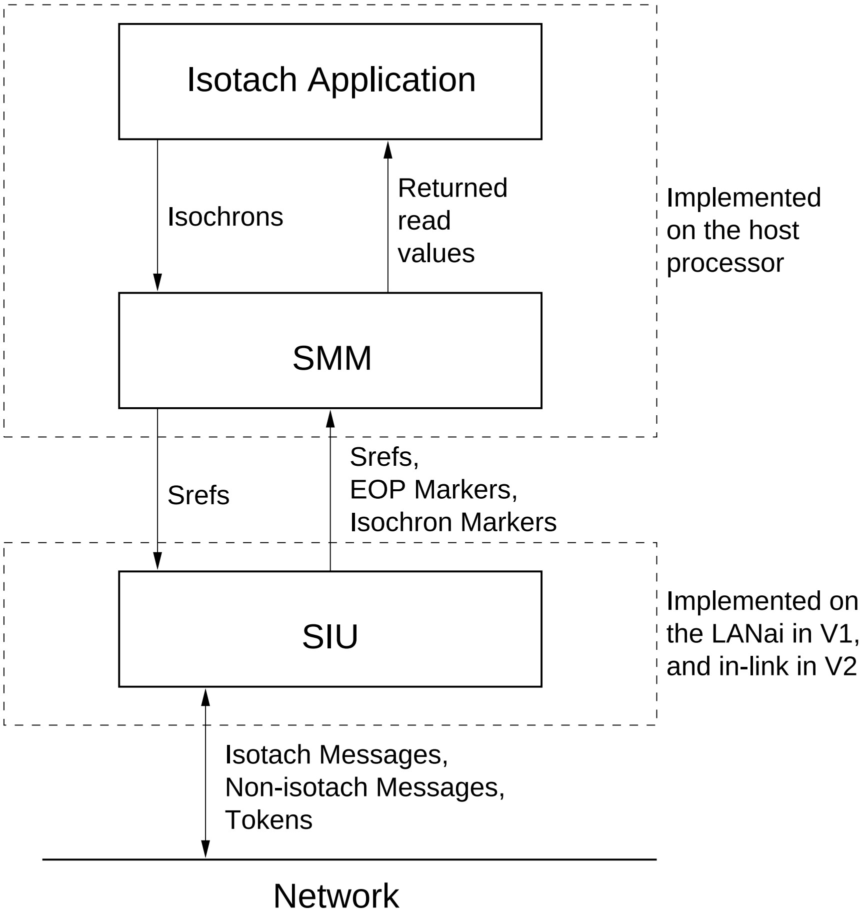

Isotach Application
Isochrons
Returned read values
Implemented on the host processor
SMM
Srefs
Srefs, EOP Markers, Isochron Markers
SIU
Implemented on the LANai in V1, and in-link in V2
Isotach Messages, Non-isotach Messages, Tokens
Network

Figure 3.1: Structure of an isotach node.

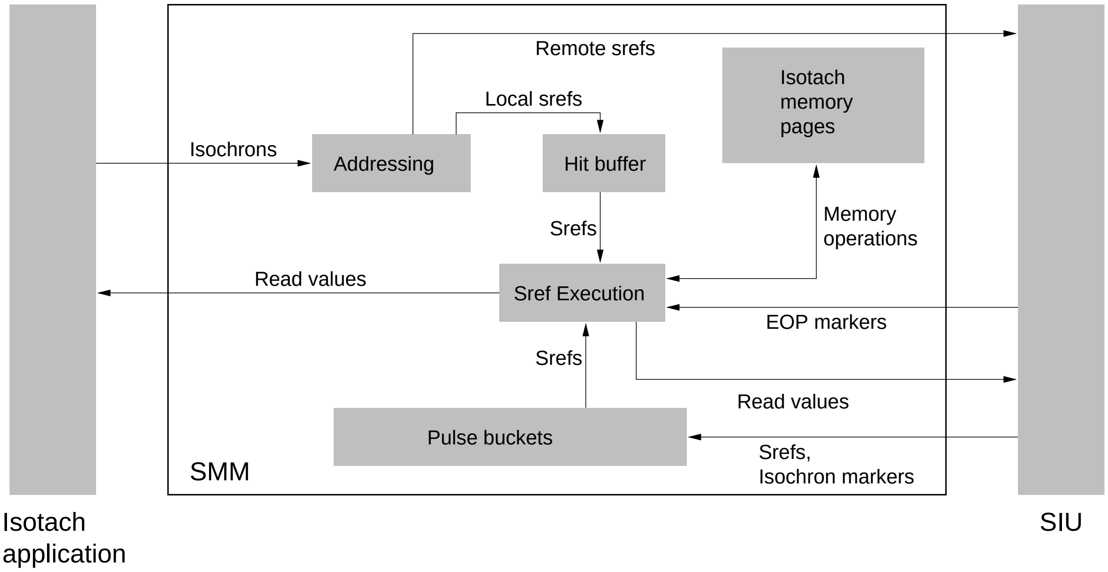


Figure 3.2: Schematic diagram of the SMM.

nearest node in the copyset; WRITEs, SCHEDs, and ASSIGNs are multicast to the entire copyset. Pages are placed at system initialization time, and do not move while the system is running. This is a restriction imposed by the design of our prototype network, not a limitation of isotach.

To issue an isochron, an application calls `iso_start()` followed by one or more of `iso_read32()`, `iso_write32()`, `iso_sched()`, and `iso_assign()`. A call to `iso_end()` completes the isochron and issues it to the SIU. The interface is documented in the Isotach Programmer's Reference Manual (appendix B).

3.2 The Shared Memory Manager

The SMM performs high level isotach functionality that is too complex to implement in hardware as part of the SIU. Figure 3.2 depicts the structure of the SMM, which we will describe in detail in this section.

3.2.1 Design Issues

We considered a number of possible forms for the SMM. They were:

- inside the Linux kernel
- a user process
- a thread in the same address space as the isotach application
- a set of library routines

We decided against putting the SMM in the kernel for a number of reasons. First, Fast Messages is designed to run in user mode — it requires high frequency polling, which is inefficient to do from the kernel. Also, we did not want the isotach program to have to cross the user/kernel protection boundary for each operation. Finally, such an approach would have tied us closely to Linux (and possibly to a particular kernel version).

Putting the SMM in a user process (the "isotach daemon" approach) would have given us more flexibility than any of the other options. However, unless a processor could be dedicated to the SMM, many context switches would be required to poll the network and to perform isotach operations, and it is likely that they would have been a significant factor in overall overhead. Also, an extra data copy would be required; at Myrinet speeds this is undesirable. Extensive use of shared memory could reduce the number of data copies, but some isotach variables (those allocated on the stack, for example) cannot be accessed easily by another process.

Unfortunately, the two remaining options do not facilitate supporting multiple isotach processes on a single machine. We decided that one process per machine is an acceptable limitation for the software prototype, especially in light of the lack of coordination between user-level networking and the kernel scheduler. Without coscheduling, user-level networking will perform very poorly in the presence of multiprogramming.

Implementing the SMM as a kernel thread has many of the same drawbacks as the isotach daemon approach, but they are less severe. Again, it makes sense only on a machine where one processor can be dedicated to the SMM thread, that can spend all of its time either performing isotach operations or polling the network. Since the application and SMM are in the same address space, data sharing is easy. We plan to eventually implement the SMM as a kernel thread. Then, each isotach host will devote a CPU to running the SMM. Currently, however, the SMM is a set of library routines. This eliminates data sharing and synchronization problems.

Because the V1 SMM is a library, its interface with the user program was easy to implement. The SMM performs well since no context switches or IPC are required. The disadvantages of the library approach are: lack of concurrency between the application and SMM, the application must poll the network, and there can be only one isotach process per host (or rather, per Myrinet interface).

Another design issue for the SMM was the granularity of isotach data. In the interest of simplicity, we do not permit isotach variables of arbitrary size. They may be either 32 or 64 bits wide (only 32-bit isotach variables are implemented in V1).

The SMM operates “outside” of logical time — it is not capable of responding to network events within a bounded amount of logical time. The SMM could operate inside logical time if it had the ability to prevent logical time from progressing; we decided against this because the SMM runs at the user level and can be descheduled for long periods of time.

3.2.2 Data Structures

The data structures in the SMM serve two main purposes: to maintain the state of isotach memory, and to handle SREFs as they move through the system. SREFs are messages that encode shared memory operations. Whenever possible, we use statically allocated memory for these data structures to improve performance and make debugging easier.

SREFs An SREF encodes a single reference to an isotach variable. It contains a timestamp indicating logical execution time, the pid of the source node, the opcode of the operation being represented, an isotach memory address, and several additional fields depending on what kind of operation the SREF represents. Isochrons are converted into SREFs by the SMM as part of the process of issuing an isochron. Throughout this chapterwe will use *isochron* to refer to two things: the group of operations that issued by an application, to perform atomically, and resulting group of SREFs produced by the SMM.

The Page Table For every page of isotach memory, we keep track of the copyset of the page. From the copyset we can derive the maximum logical distance to any node in the copyset, and whether the page is local or remote. However, these values are stored explicitly since they are used frequently. If the page is local, we store a copy of the page itself, represented as an array of isotach variables. Isotach variables contain a 32-bit value and a flag indicating whether the variable is substantiated or not.

Pulse Buckets SREFs that arrive from the network must be buffered until they are executed at the correct logical time. To facilitate this, they are sorted into buckets by logical receive time.

The Hit Buffer When an isochron produces SREFs to be executed locally, they are placed in the *hit buffer*. The SREFs cannot be placed directly into pulse buckets because the SMM operates outside of logical time, and does not know when to execute them. Isochrons have a *remote component* if any SREF produced by the isochron must be executed remotely; isochrons with no remote component are *purely local*.

Purely local isochrons are executed as soon as possible after they are issued. To maintain sequential consistency, they must be executed after any previously issued isochrons; if there are no *pending* (issued, but not executed) isochrons, a purely local isochron may be executed immediately. For isochrons with a remote component, the SIU will notify the SMM when the local component may be executed.

3.2.3 Message Types

Myrinet messaging layers are required to put a 16-bit message type field at the beginning of every message. This allows different protocols to interact, or at least to ignore messages that they cannot interpret. Fast Messages (FM) uses the message type field to demultiplex incoming messages. We have added several message types to indicate new packet types.

Non-isotach Message These messages bypass the isotach code paths and are essentially regular FM packets. They are used to avoid the overhead of the isotach code when it is not needed. For example, values returned from remote reads are sent as non-isotach messages, as are debugging messages.

Isotach Message All SREFs are sent across the network as isotach messages. They have a fixed packet format and are subject to the constraints of logical time.

Token Tokens separate and define pulses of logical time. They circulate in waves between SIUs and TMs, and pairs of TMs. Tokens never reach the SMM.

Isochron Marker When the SMM issues an isochron to the SIU, the SIU returns an isochron marker indicating the logical execution time of the isochron. Then, the SMM can execute the local component of the isochron at the correct time. Isochron markers are placed into pulse buckets along with isotach messages.

End of Pulse (EOP) Marker Although the SMM operates outside of logical time, it still has to know about some logical times — specifically, ones in which it must execute SREFs. To achieve this, the SIU keeps track of pulses in which it has delivered SREFs to the SMM. When one of these pulses ends, it sends the SMM an *end of pulse* marker. The SMM then executes any SREFs in the bucket corresponding to that logical time. The network and the rules for assigning timestamps ensure that once we receive the EOP marker for a pulse, we are guaranteed to have all SREFs that are to be executed in that pulse. Providing this guarantee is a crucial part of implementing an isotach network.

3.2.4 Issuing an Isochron

As the application issues an isochron, the SMM converts it into SREFs that are stored in either the hit buffer if they are local, or a send buffer otherwise. There are several tasks that must be performed for each shared memory reference.

Addressing Every reference to shared memory is either local, remote, or both; the SMM can determine which by consulting the page table. READs are converted into an SREF addressed to the nearest host in the copyset; WRITEs, SCHEDs, and ASSIGNs are converted into SREFs that are multicast to the entire copyset.

Logical Distance Computation The *isochron distance* is the maximum logical distance that any SREF in the isochron has to travel. This information is available from the page table once the addresses of all SREFs produced by the isochron are known.

Delivery to the SIU Before remote SREFs are delivered to the SIU, the SMM must ensure that flow control will not block the send part-way though. This is because the SREFs in an isochron are delivered to the SIU as a block to minimize the (real) time that logical time is prevented from progressing. If the SMM is forced to stop sending SREFs in the middle of an isochron, logical time can be halted for an unbounded amount of time as it waits for send credit. Stopping logical time is detrimental to the performance of the entire isotach system, whereas delaying a single SMM while credits are gathered slows computation only at that node.

3.2.5 Executing Isotach Operations

Recall that to maintain sequential consistency, isotach operations must be executed in $(pulse, pid, rank)$ order. Since EOP markers are received in strictly increasing pulse order, and the design of the SIU and SMM ensures that no isotach message arrives after the EOP marker for the pulse in which it is to be executed, the first component of the ordering is satisfied.

In our prototype, pid is equivalent to host number. Messages in a pulse are received in arbitrary order. In order to satisfy the second component of the ordering, the contents of a bucket must be sorted by sender pid before execution. We require no explicit rank field because the network is point-to-point FIFO, although a stable sort must be used to avoid reordering the contents of a bucket.

When an EOP marker is received, the contents of the corresponding bucket are sorted and executed. When an isochron marker is encountered in the bucket, SREFs are executed from the hit buffer until it is empty, or until the *second remote isochron* is reached. In other words, we execute only one remote isochron (the one that the isochron marker corresponds to) and as many purely local isochrons as possible.

Since sorting messages is potentially a time consuming part of executing the contents of a bucket, sorting will be performed in the V2 SIU. The EOP marker sent by the SIU will contain a sort vector that determines the order in which operations should be executed in the SMM. The V2 SMM must still be able to sort SREFs because the sorting module of the V2 SIU will have finite capacity; some pulse buckets may contain more SREFs than it can sort. The sorting capacity of the SIU will be chosen so that sorting by the SMM is unlikely to happen often.

Recall from section 2.4 that an FM handler may not send messages. Because executing SREFs involves sending messages (for values returned from READs), we cannot execute the contents of a bucket from the EOP marker handler. Rather, we enqueue the marker and execute the contents of the bucket later, after the handler has returned control to the main body of the SMM.

3.2.6 Implementation of Isotach Operations

As SREFs are executed from the pulse buckets and hit buffer, their opcodes are examined to determine the specific action that must be taken.

Read READs to substantiated memory locations are easy to perform. If the request is local, the value is simply put into the location specified by the application; if the request is remote, a non-isotach reply message containing the requested value is sent to the remote host. If the location is unsubstantiated, we cannot complete the READ until a value is available — the SREF is stored in a "pending read" list, along with the isotach address of the READ and the pid of the process that issued the SCHED that the READ is waiting on.

Write Conceptually, a WRITE is an optimized SCHED/ASSIGN pair. However, the implementation is simple — the value is written into the isotach memory location specified by the issuing application.

Sched The isotach memory location is marked as unsubstantiated, and the pid of the host issuing the SCHED is recorded. In the current version, we assume that the process that issues a SCHED is always the same process that issues the corresponding ASSIGN. This restriction simplifies the implementation of SCHED and ASSIGN, and will be removed in the future.

Assign All blocked READs with matching address and process id are allowed to proceed. An ASSIGN that does not correspond to any SCHED is dropped. Currently, matching READs are located by a linear search of the pending READ list.

3.2.7 Isochronous Messages

Not all computations are best expressed using a shared memory model. To support message passing programs, we have implemented isochronous messages, which may be issued in isochrons just like shared memory references. Since the system enforces isotach guarantees over all operations, an application may arbitrarily interleave messages and shared memory references.

Isochronous messages are sent to a specific recipient, and (like FM messages) are consumed by a sender-specified handler function at the receiving node. They are treated exactly like SREFs by the system until they are executed; then the isotach message handler is called instead of the SREF execution code. Isochronous messages are the basis for an implementation of a parallel rule-based system using isotach, which is currently being developed. [1]

3.3 The Switch Interface Unit

The V1 SIU is implemented as a modified and augmented FM Myrinet control program (MCP). Therefore, it is not only responsible for isotach functionality, but also for putting packets onto the network, and for transferring incoming messages into receive buffers in host memory. The V2 (hardware) SIU will reside between the host network interface and a

[1] See [14] for a detailed description of a parallel rule-based system using isotach.

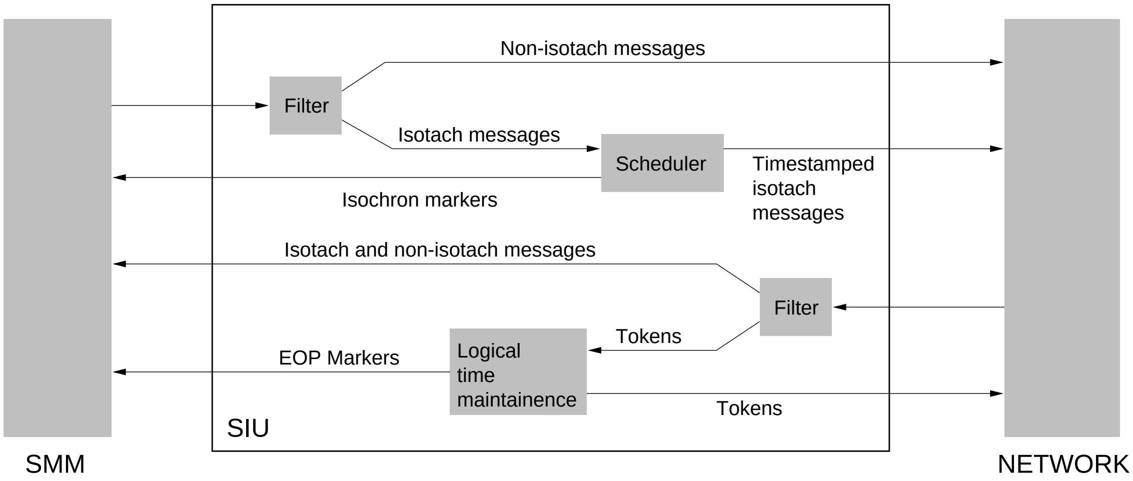


Figure 3.3: Schematic diagram of the SIU.

switch, and will buffer packets briefly as it timestamps outgoing messages and sorts incoming messages. Figure 3.3 shows the structure of the SIU; it applies to both versions.

The SIU has three functions: to maintain logical time, to assign a logical receive time to isochrons and send them out onto the network, and to take messages from the network and deliver them to the SMM.

Maintaining Logical Time When the SIU receives a token, it sends another token back to the TM as soon as possible, waiting if necessary until it has finished sending the current isochron. If the SIU has passed any SREFs to the SMM that must be executed in the logical pulse ended by the token, it sends an EOP marker to the SMM. The SIU stores the *current logical time*, which is equal to the number of tokens that it has sent since the system was initialized.

Scheduling The SIU chooses the timestamp (logical execution time) for an outgoing isochron according to the following criteria:

1. Every SREF must arrive at the correct destination prior to its execution time.

2. No SREF may be executed before any part of a previously issued isochron. [2]

3. Isochrons should be executed as soon as possible, without violating the previous constraints.

To implement these requirements, the SIU keeps a variable called `last_receive_time`, which stores the receive time of the last isochron that was issued. When an isochron arrives at the SIU, it updates `last_receive_time`, and then writes it into the timestamp field of each SREF in the isochron before sending it. `last_receive_time` is updated as follows:

$$\texttt{last\_receive\_time} = \max(\texttt{last\_receive\_time}, t + \delta)$$

where t is the current logical time, and δ is the isochron distance for the isochron that is about to be sent. Observe that this method of generating timestamps satisfies the constraints:

1. SREFs have an execution time large enough to allow them to arrive at their destination on time, because isotach messages travel one unit of logical distance per pulse of logical time.

2. Timestamps are non-decreasing, so no SREF will be executed before an SREF in a previously issued isochron (recall that for SREFs executed in the same pulse, we can break ties by examining source pid and issue order).

3. We always increase `last_receive_time` by the smallest amount necessary to guarantee that constraints 1 and 2 hold.

There is a send queue in the SRAM on the Myrinet board; it is written by the SMM and read by the SIU. To send a non-isotach message, the SIU simply prepends the proper route flits to the outgoing message and sends it using the LANai's send-DMA engine. Isotach messages are sent in the same way, but they are timestamped first.

While the SIU is sending an isochron, logical time is prevented from increasing; no tokens are sent. Along with the method for choosing the logical receive times and the

[2] The V2 SIU allows us to relax this requirement when sequential consistency is not necessary.

point-to-point FIFO-ness of the network, this ensures that all SREFs will be received in time to be executed, and that no SREF in the current isochron will be executed before any SREF in a previously issued isochron.

If isochrons are long enough that freezing logical time during isochron sending significantly slows the progress of logical time, the formula for updating `last_receive_time` could be changed to:

$$\texttt{last\_receive\_time} = \max(\texttt{last\_receive\_time}, t + \delta + x)$$

where x is the maximum number of tokens that can be sent during the transmission of an isochron. In other words, the sending SIU can give long isochrons a conservative (higher than necessary) timestamp so that we can send tokens (and thereby increase logical time) while they are being sent without compromising the requirement that all components of the isochron arrive at their destinations before their execution time. This feature is not implemented in the V1 SIU. The rate of progress of logical time is critical to the performance of an isotach network because we are buffering operations at the receiver until a specific logical time is reached; the resulting delay should not dominate the latency of operations.

In addition to timestamping outgoing messages, the scheduling unit must deliver isochron markers to the SMM because the SMM needs the execution time of each isochron in order to execute the local component at the correct time. One isochron marker is sent for each outgoing isochron.

Receiving The receive queue has two components: one in SRAM on the Myrinet board, and one in the host processor's main memory. As messages arrive at the SIU, it DMAs them into host memory as quickly as possible, where they are available to the SMM.

Incoming messages are verified to be of the expected length and type, and to have no CRC errors; only valid messages are passed to the host. The SIU notes logical receive times of isotach messages it sends. When a token arrives that concludes a pulse in which the SMM has received at least one SREF, the SIU sends the SMM an EOP marker.

3.4 An Example

Since the system is somewhat complex, we present a simple example of an isochron's path through the system. Assume two isotach hosts, 1 and 2, that are separated by logical distance 5. All memory resides in one page, whose copyset is $\{1, 2\}$. The application on host 1 executes the following code:

```
iso_start();
iso_read32 (10, &var1);
iso_write32 (20, 555);
iso_end();
```

The effect of this code is to issue an isochron that atomically reads the value from isotach memory location 10 into `var1`, and writes 555 into location 20.

The `iso_read32` generates a local SREF that is stored in the hit buffer of the SMM for host 1 (SMM1), because READs are addressed to the nearest node in the copyset. The `iso_write32` causes one SREF to be put into the hit buffer, and one SREF to be put into the send buffer, because WRITEs are sent to all members of the copyset. As SREFs are issued by the application, SMM1 computes the isochron distance. Consequently, during the call to `iso_end` the isochron distance is known to be 5. The remote SREF is marked with this distance and delivered to SIU1 after the SMM1 has ensured that enough send credit is available to send it. SIU1 marks the remote SREF with logical receive time $t + 5$, where t is the current time, and sends it to host 2. It also delivers an isochron marker with time $t + 5$ to SMM1.

SIU2 receives the SREF and delivers it to SMM2, which places it into a pulse bucket. At time $t + 5$, an EOP marker is delivered to SMM2, which then executes the WRITE. Also at time $t + 5$, SIU1 sends an EOP marker to SMM1, which matches the EOP marker with the isochron marker in the bucket, and executes the READ and write from the hit buffer. This concludes the execution of the isochron.

If the application on host 1 had tried to use the value from `var1` immediately after issuing the isochron, it would have been blocked until the local component of the isochron had completely executed.

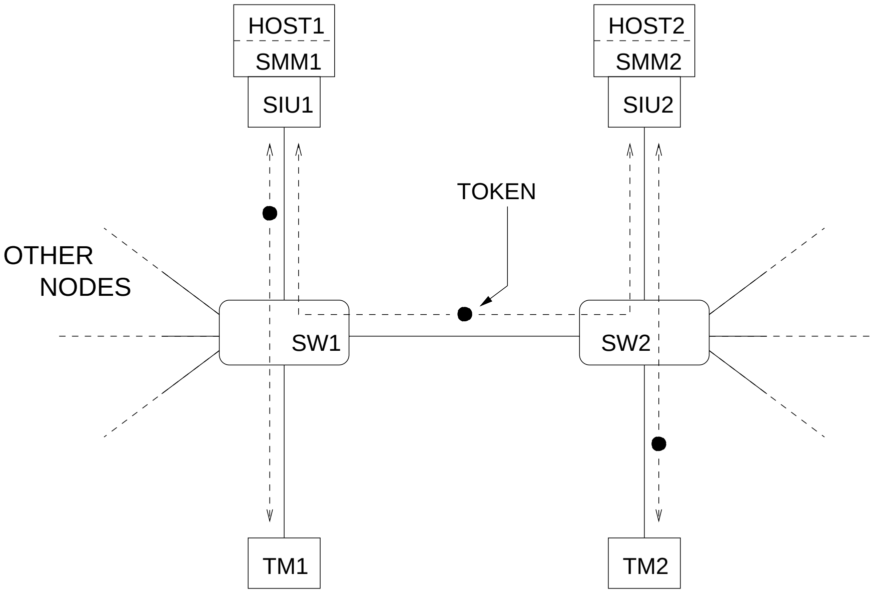


Figure 3.4: Token paths.

3.5 The Token Manager

Token managers are critical to the scalability of an isotach network. Without them, every SIU would have to exchange tokens with every other SIU — clearly impractical for large networks.

Every SIU associated with an isotach host exchanges tokens with one TM, and every TM exchanges tokens with some number of SIUs and other TMs. The V1 SIU and TM assume that there is exactly one TM per switch.

The TM algorithm is simple: it waits for token t to arrive on all inputs, and then sends token $t + 1$ to all outputs. The inputs and outputs are the nodes attached to the TM's switch, and all TMs attached to switches attached to that switch. Therefore, the maximum number of tokens that a TM must handle during a pulse is one less than the degree of its switch. Note that the TM performs the token handling required of the switch by the *isonet* algorithm from section 2.1.2.

Conceptually, there is a "loop" between each SIU and a TM. This loop contains a token that circulates back and forth; figure 3.4 shows the token loops for one of the configurations of our prototype network. To increase the rate at which logical time progresses, we can put more than one token into each loop. Our system currently has two tokens in each loop; this is implemented by having both the SIU and TM send tokens at system initialization time.

Similar to the SIU, the TM is implemented on a LANai board as a modified FM MCP. The current version offers no fault tolerance; a dropped token will halt the system. Because Myrinet does not drop packets in practice, implementation of fault tolerance has been a low priority. The V2 TM will be able to recover from dropped or corrupted tokens.

3.6 The Front End

Initially, launching an isotach application required an open shell on each participating host. This became cumbersome, so we developed a front end program that is capable of starting a program on a number of machines at once, and displaying output in different windows. It was written in Python, a high level interpreted language, using the tkinter interface to the Tk toolkit.

In addition to providing a convenient interface to isotach, the front end solves another problem — preventing multiple users from accessing the network at once. There is unfortunately no built-in way to stop a user from loading a Myrinet program and halting whatever had been running previously. The front end can check a global lock file to ensure that the network is not in use.

3.7 Implementation Difficulties

3.7.1 Porting Fast Messages

FM 1.1 was developed for SPARC processors running SunOS or Solaris, with Myrinet interfaces containing LANai 2.3 chips. [3] Before we could begin work on the isotach prototype,

[3] FM 2.0, which has become available since the initial isotach implementation was completed, supports x86 PCs running Linux, using interfaces containing LANai 4.x chips. We do not plan to use it because some of the architectural changes conflict with our modifications, and because the source code is not available.

we had to port FM to x86 PCs running Linux, and the LANai 4.x. [4]

FM is a user-level protocol, so the operating system it runs under is not much of an issue, especially since the initialization routines provided by Myricom take care of OS specific operations such as allocating DMA-able memory and mapping LANai memory into user address space.

Because Sparc and LANai CPUs are both big-endian, the original FM 1.1 code freely shares data between the processors. This does not work on the x86, which is little-endian. A conservative solution would be to change all data sent from the host to the LANai into network byte order. This would add overhead at least equivalent to an extra data copy, and is overkill for a homogeneous network. Instead, we concentrated on finding all packet fields that are interpreted by the LANai and changing those, using the `htonx()` and `ntohx()` macros, and leaving all other data in native host format.

The LANai 2.3 is a 16-bit processor, and the LANai 4.x is a 32-bit processor. Most of the changes to the MCP involved removing code intended to make the older LANai interoperate smoothly with a 32-bit host. Other modifications included changing the chip-specific initialization, and using the Send Align register to send routing bytes onto the network using DMA instead of using the slower PIO method.

Because the x86 and LANai 4.x are 32-bit processors, and because code for both chips is compiled using gcc, structures can be freely shared between the processors provided that byte-ordering issues are taken into account; this greatly simplified code development.

3.7.2 Debugging LANai Code

A number of factors make LANai programs hard to debug. First and most obviously, the LANai can do no screen or file I/O, and lacks a source level debugger, making it difficult to know what it is doing. There is an external LED on the Myrinet board that is under LANai control; in early testing, flashing it in different patterns was the best way to determine what was happening on the chip.

A more subtle problem was caused by the fact that the LANai is on the wrong side of

[4] We began our implementation using LANai 4.0 chips, and upgraded to the nearly identical LANai 4.1s when they became available.

the x86 memory protection hardware. With its DMA engine, it can write to any location in host memory without causing a protection violation. Before the FM port became stable, it was not uncommon for a wild DMA from the LANai to crash the machine being used for testing; the OS crash then made it difficult to figure out what had happened.

A final difficulty is a result of the fact that the LANai is a pipelined processor, and the status register and Myrinet link control registers are mapped into memory. Following a control register access, there can be a delay of several cycles before the corresponding status bit in the interrupt status register (ISR) changes. Since the compiler does not understand this, the programmer is effectively forced to add delay slots to the program to avoid race conditions.

To write or modify an MCP, it is essential to have access to the Myrinet User Documentation [9]. A useful high-level tutorial about writing Myrinet control programs is [13].

4

Correctness and Performance

4.1 Correctness

It was somewhat difficult to verify that the isotach system is correct. As in any complex concurrent system, exhaustive testing is not possible and the possibility of race conditions and deadlock exists. We have developed several programs designed to test various aspects of the system. After observing their behavior, inspecting the source code, and including debugging code and sanity checks, we are convinced that the system operates properly.

Test Programs The first test program picks a number of isotach variables scattered over a number of pages, each with different copysets. Each node repeatedly picks a random value, issues an isochron that writes the value to all of the isotach variables, and then issues another isochron that reads all of them. If any read fails to return the same value for all variables, then sequential consistency has been violated and an error is flagged.

Another test program sends isochronous messages between nodes, which verify the contents and order of all messages that they receive. Other isotach test programs include solutions to the dining and drinking philosopher problems. An error in the system would cause a violation of the problem constraint, e.g., neighboring philosophers dining at the same time.

Paranoid Mode To aid debugging, the system has a *paranoid* mode, in which senders tag all messages with sequence numbers that receivers examine to ensure that no messages are dropped or reordered. Also, the execution of any remote SREF results in a message being returned to the sender indicating the time at which the SREF was executed; these are checked against the execution time from the isochron marker. After a node issues an isochron, it waits for all outstanding SREFs to be executed before issuing another one; this prevents pipelining of isochrons by allowing only one pending isochron, and therefore simplifies the operation of the system. Recall that when the system is operating normally, a node may have many pending isochrons.

4.2 Performance

To get timing information, we instrumented the system by using the very high resolution timer found on Pentium and Pentium Pro systems. The RDTSC instruction causes the number of clock cycles since the processor was reset to be stored into two 32-bit registers. This gives us a timing resolution of 5.6 ns on a 180 Mhz processor. The overhead to read the timer using inline assembly language is only three instructions since there is no need to trap to the OS. The actual cost is slightly higher because the instruction overwrites two registers, and the processor may stall while writing the value to memory.

Each CPU on a multiprocessor has its own time-stamp counter, and the CPUs are reset at different times, so we had to be careful to avoid erroneous readings due to context switches. Fortunately, the Linux scheduler implements processor affinity, so a CPU-bound process does not tend to move between processors on a lightly loaded machine. Also, we validated the fine-grained timing by looking at average latency over many operations, which was calculated using a timer provided by the OS.

Performance data was gathered using the single-TM configuration from figure 2.2, because testing was done while one of our network interface cards was being replaced.

4.2.1 Read Latency

The latency of remote operations is an important characteristic of a distributed system; low latency is critical if fine-grained computations are to be performed. We will examine the best case read latency of the isotach system, occurring when a node issues an isochron containing a single read and other nodes in the network are in a polling loop. The source code for the latency test program can be found in appendix A.1.

Local Reads To test the overhead of just the SMM, we can issue a read to an isotach memory location that is present on the issuing node, at a time when we know that no isochrons are pending. This ensures that the read will be performed immediately. We obtained performance data as follows:

1. Read the time-stamp counter.
2. Issue an isochron containing a single read.
3. Read the time-stamp counter again.
4. Use the value of the read to ensure that the read has actually completed.
5. Read the value of the timer a third time.

Without instrumentation, a local read takes 1.9 μs (about 340 cycles) to complete, calculated from the average over many reads. With instrumentation, it takes 2.3 μs; the overhead probably comes from write buffer overflow in the CPU, since a lot of timing data is being written. Of the 2.3 μs, 2.1 μs is used to issue the isochron, and 0.2 μs to get the value that was read.

Remote Reads We determine the remote read latency in the same way as local latency, although there are more events to track. On the issuing node, we read the timer six times: before starting to issue the isochron, after issuing it, at the beginning and end of the isochron marker handler, at the beginning of the returned read handler, and once the datum is available to the application. On the remote node, we read the timer five times:

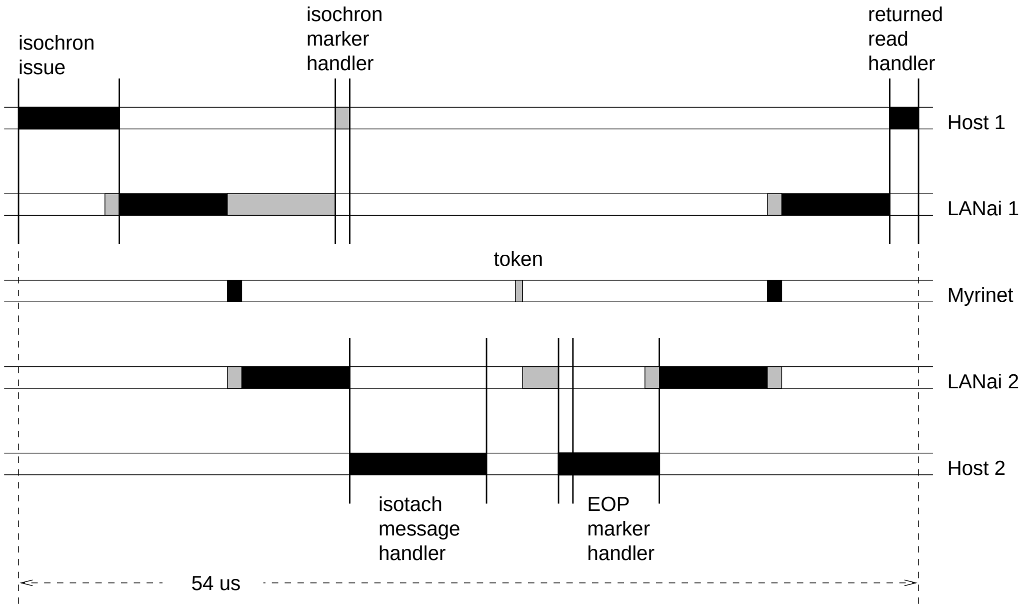


Figure 4.1: Timing analysis of a remote read.

at the start and end of the isotach message (SREF) handler, at the start and end of the EOP marker handler, and after the READ has executed completely. The total latency for a remote read is $54\,\mu$s.

Figure 4.1 depicts the delays as host 1 issues a read to an isotach variable present on host 2. Horizontal bars (drawn to scale) are timelines for the various participants, and vertical bars represent timing measurements. Black areas represent times that the processor or network is known to be busy and in the critical path; gray areas show where an element is busy, but is not in the critical path. It is important to know that the "grey areas" exist, since they may limit the amount of pipelining between the host CPU, LANai, and network that can take place when the system is under load. Interestingly, the timing code did not measurably increase the latency of remote reads; this is probably because there is enough idle processor time for the Pentium Pro write buffer to stay empty enough that it can hide write latency associated with storing timing data.

As a first approximation, we break the read latency into three components: the time it takes to issue the isochron, the time spent waiting for the response, and the time spent processing the response. This distinction is useful because work can be performed during the time spent waiting for the response (except when the isochron marker is being processed); the other times are pure overhead. It takes 5.9 μs to issue an isochron containing a single remote read; the response is received 46 μs later, and takes 1.7 μs to process. The numbers given here are averages; variance was typically not more than 10%, except when noted.

Of the 46 μs between completion of issuing the isochron and receipt of the response, 18 μs is spent at the remote node processing the read. Of this time, the first 8.3 μs are used by the isotach message handler, which places the incoming SREF into a pulse bucket. After the message handler has exited, the SMM must wait for an EOP marker before it can execute the SREF; this is received 4.5 μs later, on average. This number is highly variable because the token arrives independently of the isotach message. The number varied between 3.5 μs and 9.8 μs; this seems reasonable given the average token turnaround time of about 14 μs (see the discussion of token timing in section 4.2.3). The EOP marker handler runs in 0.42 μs, and then the SREF execution routine runs for 5.1 μs; this includes the time spent in FM code sending the value of the read to the issuing node.

27 μs are still unaccounted for. This is the network latency, which we will assume is split evenly between the two trips (because the timers on the test machines are not synchronized). In other words, the network interfaces and network impart about a 14 μs delay in each direction. Dividing the latency equally between the two messages is not quite fair because there is isotach overhead in one direction and not the other (remember that returned reads are sent as native FM messages), but the inaccuracy is not severe because the isotach overhead is only a few tens of instructions. Including the 8-cycle LANai DMA setup time, sending or receiving an isotach messages over the Myrinet takes 0.75 μs (an isotach message is 80 bytes long). Therefore, we can conclude that the total time taken by the LANai to send or receive a message is about 6.4 μs, or 192 LANai clock cycles. [1]

writes/isochron	1	2	4	8	16	32
Exp. 1: μs/all-remote isochron	14.8	20.9	33.4	62.9	121	238
Exp. 2: μs/mixed isochron	13.7	20.4	30.6	54.3	104	204

Table 4.1: Isochron issue times.

4.2.2 Isochron Throughput

Although latency is important, a latency test fails to demonstrate one of the main strengths of an isotach system: the ability to pipeline isochrons within the network. To test throughput, we simulate a workload somewhat arbitrarily by distributing shared memory references uniformly over a number of pages. For simplicity, we only perform writes, and only node 0 issues isochrons; nodes 1 and 2 sit in a polling loop waiting for remote operations to arrive. The numbers shown are the average over many operations; pipelining effects and lack of synchronized clocks between machines make a detailed timing analysis difficult. The source code for the throughput program is listed in appendix A.2.

In the first experiment, we distribute writes uniformly over three pages whose copysets, respectively, are $\{1\}$, $\{2\}$, and $\{1, 2\}$. These are all possible copysets for three nodes if no page is resident at node 0, the issuing node. This experiment is designed to measure only the time taken to issue isochrons. Each write generates an average of 1.3 SREFs, all of which are remote.

In the second, more realistic experiment, there are seven [2] isotach pages representing all possible copysets for three nodes. In this experiment, every isotach write results in an average of 1.7 SREFs being generated; 0.6 of them local, and 1.1 of them remote (distributed over the two remote nodes). Note that some of the time attributed to issuing isochrons is used executing local SREFs; this is acceptable because we are trying to characterize isotach throughput under a realistic workload.

To measure throughput, we determine the time to issue a single isochron, averaged over many isochrons. The results of the experiments appear in table 4.1. They show that there

[1] The LANai runs at PCI bus speed; 30 Mhz on a 180 Mhz Pentium Pro.

[2] There are $2^3 - 1$ combinations because we do not allow pages with an empty copyset.

writes/isochron	1	2	4	8	16	32
μs/token	17	18	20	21	25	28

Table 4.2: Token interarrival times.

is a small per-isochron overhead, and then issue time is proportional to the number of operations performed. The second experiment shows higher throughput even though more SREFs are sent per operation, which illustrates the benefits of performing local operations. We would expect a real application, whose pages have been placed such that most shared memory references are local, to show better performance than either of these experiments.

4.2.3 Rate of Progress of Logical Time

Recall that the rate at which logical time increases is important to the performance of an isotach network because we buffer received messages until their logical execution time is reached. Also recall that two tokens circulate between each SIU and its TM (and between TMs).

When the system is free of non-token traffic, the average time between token arrivals at an SIU is 14 μs. This number is obviously dominated by MCP overhead — the Myrinet traversal time for a token 8 bytes long is on the order of 50 ns.

Under load, the token interarrival time increases. We report in table 4.2 the average time between token arrivals when all three nodes in the system are running a network-intensive program: the second throughput experiment from the last section. Note that all nodes are issuing isochrons as rapidly as possible, not just node 0.

Because tokens cannot be sent while isochrons are being sent, logical time slows down as the size of isochrons increases. The increase is not severe because care was taken to minimize the time during which logical time is frozen while an isochron is being sent.

5

Summary and Future Plans

We have shown that an isotach network can be efficiently implemented in software, using COTS hardware. Our accomplishments are:

- implementing the isotach system
- verifying that the system operates correctly, using a number of test programs
- showing that it performs well
- running a nontrivial application, the parallel rule-based system
- interesting other sites in isotach; they have begun purchasing PC and Myrinet hardware

We want to do many things to improve the isotach system. There are features that will be in the hardware SIU and TM, that are currently not in the software version. Implementing these is a high priority. They include:

- Signals — a one-to-many communication mechanism, to be used to initiate a checkpoint or rollback, reset the network, or carry application specific data.
- Barriers — a traditional barrier synchronization, to be used during the checkpoint process, and by applications.

- Host-to-host mode — expands the token loop to include the SMM; this puts the SMM inside of logical time, which gives us some extra capabilities.

We want to integrate the hardware TM and SIU when they appear. The TM will be easy to integrate; the SIU will take more effort because it requires major changes to the SMM and software SIU. The software SIU will not disappear completely, since we still need an MCP. Rather, we will use something similar to the original FM MCP.

In addition to integrating hardware, there are many improvements to be made to the isotach software:

- Using GM, a new messaging layer provided by Myricom, rather than FM as our network layer. GM allows multiple processes to use the network interface, and implements a method of flow control that scales to very large networks.
- Implementing the SMM as a thread, rather than a user library, as discussed in section 3.2.1.
- Providing fault tolerance through checkpointing.
- Implementing distributed shared memory using isotach.

These improvements are largely independent of the hardware components.

A

Example Isotach Programs

A.1 lat.c

```
/*
 * lat - isotach latency test program
 */

#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <sys/time.h>
#include <sys/types.h>
#include <iso.h>

#include "prof.h"

#define COUNT 50

/*
 * do some reads, gathering timing info
 */
void time_reads (int loc, int num)
{
  int j;
  struct iso_var val;
  for (snd_stat_num=0; snd_stat_num<num; snd_stat_num++) {
    fast_timer (&snd_stats[snd_stat_num].start);
    iso_start ();
    iso_read32 (loc,&val);
    iso_end ();
    fast_timer (&snd_stats[snd_stat_num].issued);
    j = read_iso_var (&val);
    fast_timer (&snd_stats[snd_stat_num].finish);
  }
}

/*
 * do some reads without timing
 */
void do_reads (int loc, int num)
{
  int i, j;
  struct iso_var val;

  for (i=0; i<num; i++) {
    iso_start ();
    iso_read32 (loc,&val);
    iso_end ();
    j = read_iso_var (&val);
    if (j != 7777) {
      printf ("oops\n");
      exit (1);
    }
  }
}

void handler (int id, void *data, int len)
{
  int i;
  struct rcv_timer rcv;

  for (i=0; i<REPS; i++) {
    printf ("%d:  hfinish:%lld  mstart:%lld  mfinish:%lld
              done:%lld  total:%lld\n", i,
    rcv_stats[i].msg_handler_finish -
  rcv_stats[i].msg_handler_start,
  rcv_stats[i].mark_handler_start -
  rcv_stats[i].msg_handler_finish,
  rcv_stats[i].mark_handler_finish -
  rcv_stats[i].mark_handler_start,
  rcv_stats[i].send_done -
  rcv_stats[i].mark_handler_finish,
  rcv_stats[i].send_done -
  rcv_stats[i].msg_handler_start
  );
}

rcv.msg_handler_start = 0;
rcv.msg_handler_finish = 0;
rcv.mark_handler_start = 0;
rcv.mark_handler_finish = 0;
rcv.send_done = 0;

for (i=0; i<REPS; i++) if (i != 2) { /* kludge */
  rcv.msg_handler_start +=
    rcv_stats[i].msg_handler_finish -
    rcv_stats[i].msg_handler_start;
  rcv.msg_handler_finish +=
    rcv_stats[i].mark_handler_start -
    rcv_stats[i].msg_handler_finish;
  rcv.mark_handler_start +=
    rcv_stats[i].mark_handler_finish -
    rcv_stats[i].mark_handler_start;
  rcv.mark_handler_finish +=
    rcv_stats[i].send_done -
    rcv_stats[i].mark_handler_finish;
  rcv.send_done +=
    rcv_stats[i].send_done -
    rcv_stats[i].msg_handler_start;
}

printf ("average:  hfinish:%lld  mstart:%lld
         mfinish:%lld  done:%lld  total:%lld\n",
rcv.msg_handler_start/(REPS-1),
rcv.msg_handler_finish/(REPS-1),
  rcv.mark_handler_start/(REPS-1),
  rcv.mark_handler_finish/(REPS-1),
  rcv.send_done/(REPS-1));

  printf ("rcv_stat_num = %d\n", rcv_stat_num);

  iso_deinit();
}

void main (void)
{
  char *buf;
  struct timeval start, stop;
  double us;
  int i;
  struct snd_timer snd;

  buf = (char *)malloc(128);
  bzero (buf, 128);

  iso_init();
  iso_set_handler (22, handler);

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

  if (NODEID != 1) while (1) iso_poll();

  iso_start();
  iso_write32(10, 7777);
  iso_end();

  for (i=0; i<COUNT; i++) {

    gettimeofday (&start, NULL);

    if (i <= COUNT/2) {
      printf ("no timing ");
      do_reads (10, REPS);
    } else {
      printf ("timing ");
    time_reads (10, REPS);
  }

  gettimeofday (&stop, NULL);

  subtracttime (&stop, &start);
  us = (double)stop.tv_sec *1e6 + (double)stop.tv_usec;

  printf ("executed %d remote reads in %f usecs; %f us/read\n",
  REPS, us, us/REPS);
}

for (i=0; i<REPS; i++) {
  printf ("%d  iss:%lld  hstart:%lld  hfinish:%lld
           finish:%lld  total:%lld  schr:%lld  fchr:%lld\n", i,
  snd_stats[i].issued - snd_stats[i].start,
  snd_stats[i].handler_start - snd_stats[i].issued,
  snd_stats[i].handler_finish - snd_stats[i].handler_start,
  snd_stats[i].finish - snd_stats[i].handler_finish,
  snd_stats[i].finish - snd_stats[i].start,
  snd_stats[i].chr_start - snd_stats[i].start,
  snd_stats[i].chr_finish - snd_stats[i].chr_start
  );
}

snd.start = 0;
snd.issued = 0;
snd.handler_start = 0;
snd.handler_finish = 0;
snd.finish = 0;
snd.chr_start = 0;
snd.chr_finish = 0;

for (i=0; i<REPS; i++) {
  snd.start +=
    snd_stats[i].issued -
    snd_stats[i].start;
  snd.issued +=
    snd_stats[i].handler_start -
    snd_stats[i].issued;
    snd.handler_start +=
      snd_stats[i].handler_finish -
      snd_stats[i].handler_start;
    snd.handler_finish +=
      snd_stats[i].finish -
      snd_stats[i].handler_finish;
    snd.finish +=
      snd_stats[i].finish -
      snd_stats[i].start;
    snd.chr_start +=
      snd_stats[i].chr_start -
      snd_stats[i].start;
    snd.chr_finish +=
      snd_stats[i].chr_finish -
      snd_stats[i].chr_start;
  }

  printf ("\naverage:  iss:%lld  hstart:%lld  hfinish:%lld
           finish:%lld  total:%lld  schr:%lld  fchr:%lld\n",
  snd.start/REPS,
  snd.issued/REPS,
  snd.handler_start/REPS,
  snd.handler_finish/REPS,
  snd.finish/REPS,
  snd.chr_start/REPS,
  snd.chr_finish/REPS);

  printf ("snd_stat_num = %d\n", snd_stat_num);

  iso_start();
  iso_send_msg (0, 22, &start, 1);
  iso_end();
}
```

A.2 thru.c

```
/*
 * thru - isotach throughput test program
 */
```

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

#include "prof.h"

#define NUM 10000

/*
 * number of pages to touch
 */
#define NUMPAGES 7

/*
 * jumps are page sized
 */
#define JUMPSIZE 1024

/*
 * isochron size
 */
#define SIZE 16

void do_write (void)
{
  int i;
  static int k = 0;
  static int j = 0;

  iso_start ();
  for (i=0; i<SIZE; i++) {
    iso_write32 (k*JUMPSIZE, j++);
    k++;
    k = k%NUMPAGES;
  }
  iso_end ();
}

void main (void)
{
  int i,j;
  double us;
  struct timeval start, stop;

  iso_init ();

  for (j=0; j<500; j++) {

    gettimeofday (&start, NULL);

    for (i=0; i<NUM; i++) {
      do_write();
    }

    gettimeofday (&stop, NULL);
    subtracttime (&stop, &start);
    us = (double)stop.tv_sec *1e6 + (double)stop.tv_usec;
    printf ("%d isochrons issued in %fus, %f us per isochron\n",
    NUM, us, us/NUM);
  }

  printf ("\n\nall done\n");
  fflush (stdout);

  while (1) iso_poll ();

  iso_deinit();
}
```

A.3 philo.c

```
/*
```

```
 * philo - isotach dining philosophers
 */

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

#include <iso.h>

#define REPS 1000

/*
 * slow things down for demo purposes
 */
void delay(int c)
{
  int x = 0, i;

  int d = (lrand48()%15000) + 15000;
  if (c == 0) d = d/2;

  for (i=0; i<d; i++) {
    usleep (10000);
    iso_poll ();
    if (c) if (x++%1000 == 0) {
      printf (".");
      fflush (stdout);
    }
  }
  printf ("\n");
}

int left, right;

void main (void)
```

```
{
  struct iso_var *forks;
  int i, val1, val2;

  srand48 (time (NULL));

  iso_init ();

  forks = (struct iso_var *) malloc (sizeof(struct iso_var)*NUMNODES);

  printf ("I'm %d, there are %d philosophers\n", NODEID, NUMNODES);

  left = NODEID;
  right = (NODEID+1)%NUMNODES;

  if (NODEID == 0) {
    iso_start();
    iso_write32 (left+1024, NODEID*1000);
    iso_write32 (right+1024, NODEID*1000+1);
    iso_end();
  }

#ifdef VERBOSE_PH
  delay (0);
#endif

  fflush (stdout);
  fflush (stderr);

  for (i=0; i<REPS; i++) {
#ifdef VERBOSE_PH
    printf ("issuing pickup isochron\n");
    fflush (stdout);
#endif
    iso_start ();
    iso_read32 (left+1024, &forks[left]);
    iso_read32 (right+1024, &forks[right]);
    iso_sched (left+1024);
    iso_sched (right+1024);
    iso_end ();
#ifdef VERBOSE_PH
    printf ("waiting on forks\n");
    fflush (stdout);
#endif
    val1 = read_iso_var (&forks[left]);
    val2 = read_iso_var (&forks[right]);

#ifdef VERBOSE_PH
    printf ("got forks  val1 = %d  val2 = %d\n", val1, val2);
    printf ("EATING");
    fflush (stdout);
#endif

#ifdef VERBOSE_PH
  delay (1);
#endif

#ifdef VERBOSE_PH
    printf ("issuing release isochron\n");
    fflush (stdout);
#endif
    iso_start ();
    iso_assign (left+1024, val1+10);
    iso_assign (right+1024, val2+10);
    iso_end ();
#ifdef VERBOSE_PH
    printf ("forks released\n");
    printf ("THINKING");
#endif
    fflush (stdout);

#ifdef VERBOSE_PH
    delay (0);
#endif
  }

  printf ("\n\nall done\n");
  fflush (stdout);

  iso_deinit ();
```

```
}
```

B

The Isotach API

An online copy of this document is available at:
`http://grover.cs.virginia.edu/~jdr8d/prm/`

B.1 Introduction

This document describes the API for version 1.0 of the isotach system for Myrinet.

B.1.1 Conventions

The following conventions are followed in this document:

- File names are printed in **bold**
- Function names are printed in **bold**
- Constants are printed in **BOLD** and in CAPS
- Globals are printed in **bold**
- Arguments to functions are printed in *italics*
- Structure fields are printed in *italics*
- Example code is printed in `courier`

B.2 Isotach configuration files

B.2.1 fmconfig

The **fmconfig** file describes the topology of the Myrinet network to the Fast Messages layer that the isotach system uses. See section B.7 for the original FM documentation. The isotach **fmconfig** differs slightly from the one described there.

Every switch must have a token manager associated with it. Currently, token managers are implemented by a dedicated MCP, and therefore run on a workstation on the network. To indicate that a node is a token manager, append `:t` to the name of the node in the config file.

For example, if two nodes `token1` and `token2` were added to the example **fmconfig**, it would look like this:

```
0  token1:t 1 green-hornet.cs.uiuc.edu blue-whale.cs.uiuc.edu
1  - token2:t pink-panther.cs.uiuc.edu 0
```

B.2.2 shmem_map

The **shmem_map** file tells the isotach layer the copyset of each memory page. It consists of blank lines, comment lines beginning with a "#" character, and lines of the form:

```
page[-page]:node[,node]*;
```

That is, a page or page range followed by a node or list of nodes. Page size is determined by the `PAGE_SIZE` macro in **iso.h**. `MAX_USER_PAGES` indicates the maximum page number. If, during execution, an isotach program references a page which doesn't appear in **shmem_map**, a runtime error will be generated.

B.3 Isotach library routines

The following are procedure calls available to the isotach programmer and compiler. Each procedure call is defined in terms of local and shared memory and a brief description is given.

B.3.1 iso_init()

USE: iso_init()

TYPE: Function

FILE: iso.h

ARGUMENTS: *none*

RETURN TYPE: void - the program is aborted with an error message if anything goes wrong.

PURPOSE: **iso_init()** initializes the isotach network and loads the static copy set. It acts as a barrier, and therefore doesn't return until all nodes and token managers are ready to proceed.

CAVEATS: Must be called before any other isotach function.

B.3.2 iso_deinit()

USE: **iso_deinit()**

TYPE: Function

FILE: iso.h

ARGUMENTS: *none*

RETURN TYPE: void

PURPOSE: **iso_deinit()** announces to the rest of the network that the calling node will not issue any more isotach operations. It acts as a barrier, since the calling node may still need to service requests from other nodes that aren't finished.

CAVEATS: Bad form not to use it.

B.3.3 iso_start()

USE: **iso_start()**

TYPE: Function

FILE: iso.h

ARGUMENTS: *none*

RETURN TYPE: void

PURPOSE: **iso_start** marks the start of an isochron.

B.3.4 iso_end()

USE: iso_end()

TYPE: Function

FILE: iso.h

ARGUMENTS: *none*

RETURN TYPE: void

PURPOSE: **iso_end** marks the end of an isochron.

B.3.5 iso_read32()

USE: iso_read32(shaddr, laddr)

TYPE: Function

FILE: iso.h

ARGUMENTS: *shmem_addr_t shaddr, struct iso_var32 *laddr*

RETURN TYPE: void

PURPOSE: **iso_read32()** schedules a read access to one word of shared memory and specifies the address of a local variable to put the value in. Issuing a read is non-blocking - the process blocks when it tries to access the value of a read which has not returned yet.

CAVEATS: Must be called inside of an **iso_start()**...**iso_end()** block.

B.3.6 iso_read64()

USE: iso_read64(shaddr, laddr)

TYPE: Function

FILE: iso.h

ARGUMENTS: *shmem_addr_t shaddr, struct iso_var64 *laddr*

RETURN TYPE: void

PURPOSE: **iso_read64()** schedules a read access to a double word of shared memory and specifies the address of a local variable to put the value in. Issuing a read is nonblocking - the process blocks when it tries to access the value of a read which has not returned yet.

CAVEATS:

- Not implemented in version 1.0.
- Must be called inside of an **iso_start()**...**iso_end()** block.
- 64 bit reads must be to even shared addresses.

B.3.7 read_iso_var32()

USE: **read_iso_var32(var)**

TYPE: Function

FILE: iso.h

ARGUMENTS: *struct iso_var32 *var*

RETURN TYPE: long int

PURPOSE: **read_iso_var32** returns the value from a local copy of a 32 bit isotach variable. It blocks the reading process if the value hasn't been returned yet.

B.3.8 read_iso_var64()

USE: **read_iso_var64(var)**

TYPE: Function

FILE: iso.h

ARGUMENTS: *struct iso_var64 *var*

RETURN TYPE: long long int

PURPOSE: **read_iso_var64** returns the value from a local copy of a 64 bit isotach variable. It blocks the reading process if the value hasn't been returned yet.

B.3.9 iso_write32()

USE: iso_write32(shaddr, val)

TYPE: Function

FILE: iso.h

ARGUMENTS: *shmem_addr_t shaddr, long int val*

RETURN TYPE: void

PURPOSE: **iso_write32()** writes a value to a shared memory address. It is equivalent to calling **sched()** and then **assign32()**. Writes are nonblocking.

CAVEATS: Must be called inside of an **iso_start()**...**iso_end()** block.

B.3.10 iso_write64()

USE: iso_write64(shaddr, val)

TYPE: Function

FILE: iso.h

ARGUMENTS: *shmem_addr_t shaddr, long long int val*

RETURN TYPE: void

PURPOSE: **iso_write64()** writes a value to a shared memory address. It is equivalent to calling **sched()** and then **assign64()**. Writes are nonblocking.

CAVEATS:

- Not implemented in version 1.0.
- Must be called inside of an **iso_start()**...**iso_end()** block.
- 64 bit writes must be to even shared addresses.

B.3.11 iso_sched()

USE: iso_sched(shaddr)

TYPE: Function

FILE: iso.h

ARGUMENTS: *shmem_addr_t shaddr*

RETURN TYPE: void

PURPOSE: **iso_sched()** schedules an assign to a shared memory location.

CAVEATS:

- Each node may have only one outstanding sched for a given shared memory location. An outstanding sched is one for which the corresponding assign has not yet been executed.
- Must be called inside of an **iso_start()**...**iso_end()** block.

B.3.12 iso_assign32()

USE: iso_assign32(shaddr, val)

TYPE: Function

FILE: iso.h

ARGUMENTS: *shmem_addr_t shaddr, long int val*

RETURN TYPE: void

PURPOSE: **iso_assign32()** substantiates a scheduled shared memory access.

CAVEATS:

- Every call to **iso_assign32()** must be preceeded by an **iso_sched()** to the same variable.
- Must be called inside of an **iso_start()**...**iso_end()** block.

B.3.13 iso_assign64()

USE: iso_assign64(shaddr, val)

TYPE: Function

FILE: iso.h

ARGUMENTS: *shmem_addr_t shaddr, long long int val*

RETURN TYPE: void

PURPOSE: **iso_assign64()** substantiates a scheduled shared memory access.

CAVEATS:

- Not implemented in version 1.0.
- Must be called inside of an **iso_start()**...**iso_end()** block.
- 64 bit writes must be to even shared addresses.
- Every call to **iso_assign64()** must be preceeded by an **iso_sched()** to the same variable.

B.3.14 iso_poll()

USE: iso_poll(void)

TYPE: Function

FILE: iso.h

ARGUMENTS: *none*

RETURN TYPE: void

PURPOSE: **iso_poll()** processes any messages that have arrived from the network since the last poll; this means that an isotach process is not responsive unless it is polling frequently. Note that issuing isochrons implicitly polls the network, so polling need only be performed during lengthy computations that require no isochrons to be issued.

CAVEATS:

- Must not be called in an **iso_start()**...**iso_end()** block.

B.3.15 iso_send_msg()

USE: iso_send_msg(target, handler, data, size)

TYPE: Function

FILE: iso.h

ARGUMENTS: *int target, int handler, void *data, int size*

RETURN TYPE: void

PURPOSE: **iso_send_msg()** sends a message isochronously. *target* is the NODEID of the receiving node, and *handler* is the index of the handler function to be called at that node. *data* is a pointer to the data to be sent, and *size* is the size of the message in bytes.

CAVEATS:

- Must be called in an **iso_start()**...**iso_end()** block.

B.3.16 iso_set_handler()

USE: iso_set_handler(num, handler)

TYPE: Function

FILE: iso.h

ARGUMENTS: *int handler, iso_handler *handler*

RETURN TYPE: iso_handler *

PURPOSE: **iso_set_handler()** sets handler *num* to be function *handler*. It returns the value that the handler was previously set to (or *NULL* otherwise). To clear a handler, call **iso_set_handler()** with *NULL* as the handler.

B.4 Isotach constants and system variables

The following are isotach-specific system variables.

B.4.1 NODEID

USE: NODEID

TYPE: externally defined global signed long int

FILE: iso.h

PURPOSE: NODEID is the logical process id of the referencing process. IDs are contiguous, and range from 0 to NUMNODES-1.

CAVEATS: Undefined until **iso_init()** has returned.

B.4.2 NUMNODES

USE: NUMNODES

TYPE: externally defined global signed long int

FILE: iso.h

PURPOSE: NUMNODES holds the number of processes involved in the shared memory computation

CAVEATS: Undefined until **iso_init()** has returned.

B.4.3 MAX_ISO_MSG_SIZE

USE: MAX_ISO_MSG_SIZE

TYPE: glocal constant

FILE: iso.h

PURPOSE: MAX_ISO_MSG_SIZE is the largest number of bytes that can be sent or received using **iso_send_msg()**.

B.4.4 MAX_ISO_HANDLERS

USE: MAX_ISO_HANDLERS

TYPE: global constant

FILE: iso.h

PURPOSE: MAX_ISO_HANDLERS is the largest number of handlers usable by an isotach program. That is, handlers may be numbered 0 through MAX_ISO_HANDLERS-1.

B.5 Isochronous Messages

When a message is delivered to a node, a *handler* is called. All handlers are initially invalid; a message arriving for an invalid handler causes an isotach program to terminate with an error message.

Handlers are of type *void (*handler)(int sender, void *data, int length)* where *sender* is the *NODEID* of the sending process, *data* points to the received data, and *length* is the number of bytes received.

When the handler exits, the *data* area is returned to the system – this means that any data that must outlast the handler has to be copied into a new location. Handlers are atomic with respect to the rest of the isotach system, and may not call any isotach functions.

B.6 Example isotach programs

B.6.1 Skeleton isotach program

put code here

B.6.2 Dining philosophers

put code here

B.6.3 LU decomposition

put code here

B.7 The FM 1.1 configuration file

```
fmconfig - Fast Messages network configuration file

The network configuration file describes the topology of the Myrinet
network.  The file is read as part of the Fast Messages
FM_initialize() call.  The default name of the network configuration
file is ‘‘fmconfig’’, but this can be overridden by setting the
environment variable FMCONFIGFILE or by using FM_set_parameter().

The configuration file contains one line for each switch in the
network.  The first field on a line is a switch number, which is
assigned by the user to uniquely identify that switch.  Following the
switch number is a field for each port of the switch that specifies
what that port is connected to: either another switch, a workstation,
or nothing.  Switches are specified by their switch number,
workstations by their name, and dangling ports by a ‘‘-’’.

EXAMPLE: Consider the following network, which contains three machines
(blue-whale.cs.uiuc.edu, green- hornet.cs.uiuc.edu, and
pink-panther.cs.uiuc.edu) and two 4-port Myrinet switches:
```

```
                            +------+-----+      +------+-----+
       +--------------+     |      0     |      |      0     |
       |              |     |            |      |            |
       |  blue-whale  +-----+3 switch#0 1+------+3 switch#1 1+
       |              |     |            |      |            |
       +--------------+     |      2     |      |      2     |
                            +------+-----+      +------+-----+
                                   |                   |
                                   |                   |
                                   |                   |
                           +-------+------+    +-------+------+
                           |              |    |              |
                           | green-hornet |    | pink-panther |
                           |              |    |              |
                           +--------------+    +--------------+
```

The corresponding network configuration file might look like this:

```
0  - 1 green-hornet.cs.uiuc.edu blue-whale.cs.uiuc.edu
1  - - pink-panther.cs.uiuc.edu 0
```

NOTES:

Machines must be specified by their full name as returned by hostname(1).

The maximum network diameter is currently limited to 4.

Lines in the network configuration file that start with ''#'' are considered comments and are ignored.

FM 1.1 assumes that the network configuration is static throughout program execution. There is no way to dynami- cally add or delete nodes.

Currently, for deadlock reasons, the network may contain no cycles. This restriction will be lifted in future

```
releases of FM.

There are two types of Myrinet switches: absolute- addressed and
relative-addressed.  FM 1.1 assumes that all switches in the network
use the same addressing scheme (relative, by default).

FM 1.1 does not require that all switches have the same number of
ports.

CREDITS: The Illinois Fast Messages interface was designed and
implemented by the Concurrent Systems Architecture Group at the
University of Illinois at Urbana-Champaign.

       Principal investigator
       Andrew A. Chien

       Team members
       Scott Pakin
       Vijay Karamcheti
       Mario Lauria
       Steve Hoover
       Olga Natkovich

Questions and comments should be directed to us at:

fast-messages@red-herring.cs.uiuc.edu
```

Bibliography

[1] Henri Bal, Rutger Hofman, and Kees Verstoep. A Comparison of Three High Speed Networks for Parallel Cluster Computing. *Workshop on Communication and Architectural Support for Network-based Parallel Computing (CANPC'97)*, pages 184–197, February 1997. Available from `ftp://ftp.cs.vu.nl/pub/amoeba/orca_papers/canpc97.ps.Z`.

[2] 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. Available from `http://www.myri.com/research/publications/Hot.ps`.

[3] Eric A. Brewer, Frederic T. Chong, Lok T. Liu, John Kubiatowicz, and Shamik D. Sharma. Remote Queues: Exposing Network Queues for Atomicity and Optimization. *ACM Symposium on Parallel Algorithms and Architectures*, 1995. Available from `http://www.cs.berkeley.edu/~brewer/papers.html/spaa.ps`.

[4] Paul F. Reynolds Jr., Craig Williams, and Raymond R. Wagner Jr. Isotach Networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(4), April 1997. Available from `ftp://ftp.cs.virginia.edu/CS-95-09.ps.Z`.

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

[6] Koen Langendoen, John Romein, Raoul Bhoedjang, and Henri Bal. Integrating Polling, Interrupts, and Thread Management. *The 6th Symposium on the Frontiers of Mas-*

sively Parallel Computation (Frontiers '96), pages 13–22, October 1996. Available from `ftp://ftp.cs.vu.nl/pub/amoeba/orca_papers/frontiers96.ps.Z`.

[7] The Linux Documentation Project. Available from `http://sunsite.unc.edu/mdw/`.

[8] Alan M. Mainwaring, Brent N. Chun, Saul S. Schleimer, and Daniel S. Wilkerson. System Area Network Mapping. In *9th Annual ACM Symposium on Parallel Architectures Algorithms and Architectures*, May 1997. Available from `http://HTTP.CS.Berkeley.EDU/~alanm/Papers/mapper.ps`.

[9] Myricom. Myrinet Documentation. Available from `http://www.myri.com/scs/documentation/`.

[10] Myricom. The Myrinet API. Available from `http://www.myri.com/scs/documentation/mug/development/api.html`.

[11] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 1997. Available from `http://www-csag.cs.uiuc.edu/papers/fm-pdt.ps`.

[12] Scott Pakin, Mario Lauria, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. *Supercomputing*, December 1995. Available from `http://www-csag.cs.uiuc.edu/papers/myrinet-fm-sc95.ps`.

[13] Anthony Skjellum, Gregory Henley, Nathan Doss, and Thomas McMahon. A Guide to Writing Myrinet Control Programs for LANai 3.x. Available from `http://WWW.ERC.MsState.Edu/labs/icdcrl/learn_mcp/`.

[14] Rashmi Srinivasa. Parallel Rule-based Systems on Isotach Networks. Master's thesis, University of Virginia, 1996.

[15] 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.

[16] Craig C. Williams. *Concurrency Control in Asynchronous Computations*. PhD thesis, University of Virginia, January 1993. Available from `ftp://ftp.cs.virginia.edu/pub/dissertations/9301.ps.Z`.