Isotach Networks

Paul F. Reynolds, Jr.
Craig Williams
Raymond R. Wagner, Jr.

Abstract — We introduce a class of networks called *isotach networks* designed to reduce the cost of concurrency control in asynchronous computations. Isotach networks support several properties important to the correct execution of parallel and distributed computations: atomicity, causal message delivery, sequential consistency, and memory coherence in systems in which shared data can replicate and migrate. They allow processes to execute atomic actions without locks and to pipeline memory accesses without sacrificing sequential consistency. Isotach networks can be implemented in a wide variety of configurations, including NUMA (non-uniform memory access) multiprocessors and distributed as well as parallel systems. Networks that implement isotach time systems are characterized not by their topology, but by the guarantees they make about the relative order in which messages appear to be delivered. These guarantees are expressed in logical time, not physical time. Physical time guarantees would be prohibitively expensive, whereas logical time guarantees can be enforced cheaply, using purely local knowledge, and yet are powerful enough to support efficient techniques for coordinating asynchronously executing processes. Empirical and analytic studies of isotach systems show that they outperform conventional systems under realistic workloads, in some cases by an order of magnitude or more.

---

1. INTRODUCTION

Isotach networks are a new class of networks designed to support concurrency control. The term *concurrency control* is from the database literature, but the problem to which it refers — the problem of coordinating access to shared objects — is fundamental to asynchronous computation. Concurrency control is required for message-based model (MBM) computations in which processes communicate by sending messages as well as for shared-memory model (SMM) computations in which processes communicate by accessing shared variables, and for distributed computations in which processes communicate over local or wide area networks as well as for parallel computations on tightly-coupled multiprocessors. In its simplest form, concurrency control concerns enforcing atomicity and sequential consistency, i.e. ensuring that each atomic action appears to be executed indivisibly and that each process's accesses appear to be executed

in the order specified by its program. In systems in which objects may migrate or replicate, as in systems with caches or a distributed shared memory, concurrency control also encompasses the problem of maintaining memory consistency. Existing concurrency control techniques are costly in execution time and frequently also in programming effort.

Concurrency control is hard because existing interconnection networks offer such weak guarantees about the relative order in which they deliver messages. Few networks offer guarantees any stronger than FIFO delivery order among messages with both the same source and destination. As a result, a process can neither predict nor control the order in which its messages are received relative to concurrently issued messages and must use delays and higher-level synchronization mechanisms such as locks to ensure that execution is consistent with the program's constraints. Isotach networks offer built-in support for concurrency control in the form of a guarantee called the *velocity invariant* that relates a message's logical communication time to its logical communication distance. The invariant is expressed in logical time because a physical time guarantee would be prohibitively expensive, whereas the logical time guarantee can be enforced cheaply, using purely local knowledge, and yet is powerful enough to provide a sufficient basis for concurrency control. Isotach networks are characterized by this invariant, not by any particular topology or programming model. They can be implemented in a wide variety of configurations, including NUMA (non-uniform memory access) multiprocessors. They are suited to both distributed as well as parallel systems and SMM as well as MBM computations.

We are well aware of the recent controversy ([5, 7, 27]) over implementing synchronization services at a low level in end-to-end systems [29]. We believe isotach networks are justifiable on a cost/benefit basis — the guarantee they offer can be implemented cheaply and is useful to a wide class of computations in enforcing basic synchronization constraints. A process in an iso-

tach system can execute atomic actions with little or no synchronization, without acquiring locks or otherwise obtaining exclusive access rights to the objects accessed, and can pipeline memory accesses without sacrificing sequential consistency. Isotach systems also permit highly concurrent access to replicated data, a capability that is important to achieving low-latency access to shared data. Our studies of isotach systems, described in section 5, show order of magnitude performance improvement under realistic workloads. In extreme cases of high contention for shared objects, conventional systems cease to perform acceptably, but isotach systems continue to perform well.

Section 2 of this paper defines isotach logical time, the logical time system in which the velocity invariant is expressed. Section 3 motivates isotach logical time by demonstrating the power of the velocity invariant in enforcing atomicity and sequential consistency. Section 4 gives an efficient distributed algorithm for maintaining isotach logical time. Section 5 describes empirical and analytic studies of isotach networks. Section 6 is a summary and discussion.

2. ISOTACH LOGICAL TIME

A logical time system is a set of constraints on the way in which events of interest are ordered, i.e., assigned logical times. Isotach logical time is an extension of the logical time system defined by Lamport in his classic paper on ordering events in distributed systems [16]. In Lamport's system, the events of interest are the sending and receiving of messages. Times assigned to these events are required to be consistent with the *happened before* relation, a relation over send and receive events that captures the notion of potential causality: event *a happened before* event *b*, denoted $a \rightarrow b$, if 1) *a* and *b* occur at the same process and *a* occurs before *b*; 2) *a* is the event of sending message *m* and *b* is the event of receiving the same message *m*; or 3) there exists some event *c* such that $a \rightarrow c$ and $c \rightarrow b$. In Lamport's system $a \rightarrow b \Rightarrow t(a) <$

$t(b)$, where for any event x, $t(x)$ denotes the logical time assigned to x. A logical time system that extends Lamport's by requiring $a \rightarrow b \Leftrightarrow t(a) < t(b)$ has been used as the basis for a package of communication primitives for distributed computation [4].

After defining his logical time system, Lamport gives a simple distributed algorithm that implements it. Each process has its own logical clock, a variable that records the time assigned to the last local event. When it sends a message, a process increments its clock and timestamps the message with the new time. When it receives a message, a process sets its clock to one more than the maximum of its current time and the timestamp of the incoming message. This algorithm ensures $a \rightarrow b \Rightarrow t(a) < t(b)$, as required.

In an isotach logical time system, times are lexicographically ordered n-tuples of integers of which the first and most significant component is the *pulse*. The number and interpretation of the remaining components can vary. In this paper, each isotach logical time is a 3-tuple of the form $(pulse, pid(m), rank(m))$, where $pid(m)$ is the identifier of the process that issued message m and $rank(m)$ is the issue rank of the message, i.e., $rank(m) = i$ if m is the *ith* message issued by $pid(m)$. In an isotach logical time system, logical times must be consistent with both the $\rightarrow$ relation and with the *velocity invariant*. For any message m, let $d(m)$ denote the logical distance m travels, $t_s(m)$ the logical time at which m is sent, and $t_r(m)$ the logical time at which m is received. The velocity invariant states that each message m is received exactly $d(m)$ pulses after it is sent. Thus $t_s(m) = (i, j, k)$ implies $t_r(m) = (i + d(m), j, k)$.* Isotach logical time is so named because all messages travel at the same velocity in logical time — one unit of logical distance per pulse. In this paper, unless otherwise stated, logical distance is routing distance, i.e., the number of switches through which m is routed.

For any message m for which $d(m) = 0$, e.g., a cache hit or a message between collocated processes, the velocity invariant requires $t_s(m) = t_r(m)$. For this reason, we relax Lamport's requirement that $a \rightarrow b \Rightarrow t(a) < t(b)$. In isotach logical time, $a \rightarrow b \Rightarrow t(a) \leq t(b)$.

3. APPLICATIONS OF ISOTACH LOGICAL TIME

Before showing how to build a system that implements isotach logical time (section 4), we show why. We motivate isotach logical time by showing the ease with which isotach systems can enforce atomicity and sequential consistency. Our discussion of atomicity and sequential consistency is in SMM terms. We refer to messages as operations and to data objects as variables. Note however that atomicity and sequential consistency are also relevant to MBM computations. We end the section with a brief description of other applications of isotach systems.

An *atomic action* is a group of operations on shared variables that appears to be executed indivisibly, i.e., without interleaving with other operations [19, 20]. In many contexts the atomic action is also a unit of recovery from hardware failure. We assume a system that is fault-free or that appears at the application level to be fault-free. Conventional systems enforce atomicity with some form of locking. Drawbacks of locking include overhead for lock maintenance in space, time, and communication bandwidth, unnecessarily restricted access to shared variables, and the care that must be taken when using locks to avoid deadlock and livelock. In an isotach system, a process can execute *flat* atomic actions, (atomic actions containing no internal data dependences among shared variables) without synchronizing with other processes and can execute *structured* atomic actions (atomic actions with such dependences) without acquiring locks or otherwise obtaining exclusive access rights to the variables accessed.

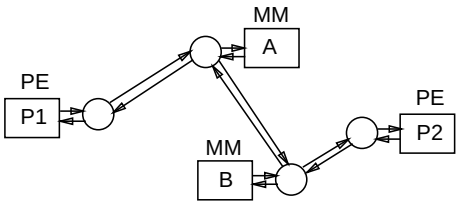
An execution is *sequentially consistent* if operations are executed in an order consistent with the order specified by each individual process's sequential program [17]. This property is so basic it is easily taken for granted, but it is expensive to enforce in non-bus-based systems because stochastic delays within the network can cause operations to be received in an order different from the order in which they were sent. The conventional solution is to prohibit pipelin-

ing of operations, i.e., each process delays issuing each operation until it receives an acknowledgement for its previous operation. Since pipelining is an important way to decrease effective memory latency, this solution is expensive. The high cost of enforcing sequential consistency has led to extensive exploration of weaker memory consistency models, e.g., [9, 30]. These weaker models are harder to reason about and still impose significant restrictions on pipelining, but make sense given the cost of maintaining sequential consistency in a conventional system. In an isotach system, processes can pipeline memory operations without violating sequential consistency.

The key to enforcing atomicity and sequential consistency in an isotach system is the velocity invariant. Given the velocity invariant, a PE that knows $d(m)$ for each operation m it sends can control the logical time at which its operations are *received* by controlling the logical time at which the operations are *sent*. Assuming each MM executes operations in the order in which they are received, PE's can ensure atomicity and sequential consistency as follows:

> **ATOMICITY**. Send operations from the same flat atomic action so they are received in the same pulse.
>
> **SEQUENTIAL CONSISTENCY**. Send each operation so it is received in a pulse no earlier than that of the operation issued before it.

These rules, the *send order rules*, are applicable to any topology. In a system with an *equidistant* network, i.e., a network in which every message travels the same logical distance, they call for each PE to send operations in the order in which they were issued and to send all operations from the same flat atomic action in the same pulse. In a system with a non-equidistant topology, the send order rules may require that a PE send operations in an order that differs from the order in which they were issued. Note the distinction between issuing and sending. A message is *issued* when control over the message passes from the process to the kernel or messaging system and is *sent* when control passes to the network.



(a)

PE	operation	t_s	d	t_r
1	read A	(i+1,1,x)	2	(i+3,1,x)
	read B	(i,1,x+1)	3	(i+3,1,x+1)
2	write A	(i+1,2,y)	3	(i+3,2,y)
	write B	(i,2,y+1)	2	(i+3,2,y+1)

(b)

Fig. 1. Executing atomic actions. (a) Network topology. (b) Operation send and receive times.

Example. Assume process P_1 is required to read and P_2 is required to write shared variables A and B and that each process's accesses must be executed as an atomic action. In a conventional system, P_1 and P_2 must obtain locks, either on individual variables or on the section of code through which the variables are accessed. In an isotach system, P_1 and P_2 can execute their accesses without locks. Consider an isotach system with the non-equidistant topology shown in Fig. 1(a). Each circle in Fig. 1(a) represents a switch and each rectangle an MM or PE. Fig. 1(b) shows one of many possible correct executions of P_1 and P_2's accesses on this network. In accordance with the first send order rule, each process sends its operations so that they both arrive in the same pulse, e.g., P_1 sends the operation on A one pulse after the operation on B since the routing distance from P_1 to A is one less than to B. If, as in the execution shown, all four operations happen to be received in the same pulse, operations on each shared variable will be received and executed in order by *pid*. As a result, each atomic action will appear to be executed without interleaving with other operations.

The correctness proof for the send order rules is similar to serializability proofs of database schedulers, see e.g. [21]. We show that for any execution on an isotach system that conforms to the send order rules there is an equivalent serial execution that is atomic and sequentially consistent. In this context, we say an execution is *atomic* if every flat atomic action is executed atomically. To establish that two executions of the same program are *equivalent* it is sufficient to

show that each shared variable is accessed by the same operations and that operations on the same variable are executed in the same order in both executions.

Theorem 1. *Any isotach system execution E that satisfies the send order rules is atomic and sequentially consistent.*

Proof. Let E_s be the serial execution in which the operations in E are executed in order by their logical receive times in E. Consider two operations op_i and op_j on the same shared variable V. Without loss of generality, assume op_i is received in E before op_j. Since isotach logical times are consistent with the $\rightarrow$ relation and no two operations have the same logical receive time, $t_r(op_i) < t_r(op_j)$. Thus op_i is executed before op_j in E_s. Since operations on the same variable are executed in E in the order in which they are received, the operations are executed in the same order in E. Thus E and E_s are equivalent executions. Since, for any flat atomic action A, all operations in A are received in the same pulse (first send order rule), issued by the same process, and have consecutive issue ranks, the logical receive time of any operation not in A is outside the interval of logical time delimited by the receive events for operations in A. Thus the operations from each atomic action are executed in E_s without interleaving with other operations. For any pair of operations op_i and op_j issued by the same process where $rank(op_i) < rank(op_j)$, op_i and op_j are either received in the same pulse or op_i is received in an earlier pulse than op_j (second send order rule). In either case, op_i is executed before op_j in E_s. Since E is equivalent to E_s and E_s is sequentially consistent and atomic, E is sequentially consistent and atomic. □

Example. Consider again the execution shown in Fig. 1. The *space-time* diagram in Fig. 2(a) shows one of the many possible ways in which this execution can occur in physical time. Note that events must occur in logical time in the order in which they occur in physical time only if they are causally connected. Fig. 2(b) shows the serial execution in which operations are

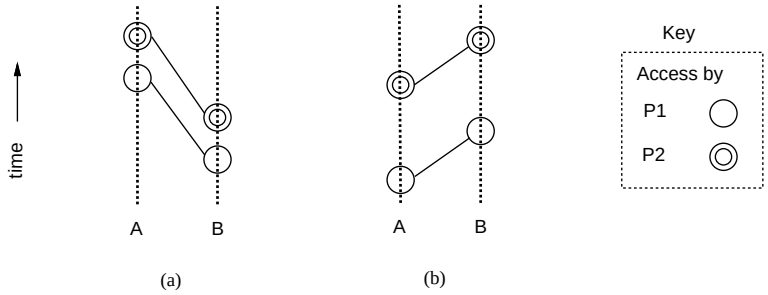


Fig. 2. Space-time diagrams of equivalent executions.

executed in order by their logical receive times. Each dotted vertical line represents a shared variable and each circle the execution of an operation. A solid line connecting circles means the associated operations are from the same atomic action. Since each variable is accessed by the same operations in the same order in both executions, the executions are equivalent. In discussing a similar pair of diagrams, Lamport provides an alternative way to view the equivalence among the executions in the figure [16]:

> Without introducing the concept of time into the system (which would require introducing physical clocks), there is no way to decide which of these pictures is the better representation.

Intuitively, any vertical line in the diagram can be stretched or compressed in any way that preserves the relative order of the points on the line and the result will be equivalent to the original execution. Without memory-mapped peripherals or other special instrumentation, the executions are indistinguishable. Since the second of the equivalent executions in Fig. 2 is atomic, so is the first, even though it interleaves execution of atomic actions in physical time.

Structured atomic actions cannot be executed in the same way as flat atomic actions because data dependences among operations prevent issuing all the operations in a batch, but the techniques for executing flat atomic actions together with a class of operations called *split operations* support execution of structured atomic actions [34]. Isotach based techniques for

executing structured atomic actions require synchronization only in the case of a true data dependence, otherwise known as a flow or write/read dependence.

Other applications for isotach logical time include the following:

• Cache Coherence — Isotach based techniques for enforcing atomicity and sequential consistency extend to systems with caches. The resulting cache coherence protocols [33, 34], are more concurrent than existing protocols for non-bus-based systems in that they support multiple concurrent reads and writes of the same block; allow processes to pipeline memory accesses without sacrificing sequential consistency; and allow processes to read and write multiple shared cache blocks atomically without invalidating the copies held by other processes or otherwise obtaining exclusive rights to the accessed blocks.

• Combining — Combining is a technique for maintaining good performance in the presence of multiple concurrent accesses to the same variable [14]. An isotach network need not implement combining, but if it does, it can combine operations not combinable in other networks, resulting in improved concurrency in accessing shared memory [35].

• Causal Message Delivery — Message delivery is *causal* if messages are delivered in an order consistent with the order in which they were sent. Causal ordering can be obtained by the network or by reordering messages at the destination before delivery to the application process. For any message m, let $s(m)$ denote the event of sending and $r(m)$ of receiving message m. A network implements causal message delivery, if for any two messages m' and m'', $s(m') \rightarrow s(m'') \Rightarrow \neg(r(m'') \rightarrow r(m'))$. Any isotach network in which logical distance is consistent with the triangle inequality implements causal message delivery [34]. The networks previously shown to implement causal message delivery — shared buses and a class of tree-structured networks called race-free networks [18] — are not applicable to as wide a range of topologies.

The causally ordered multicast is a generalization of causal message delivery to multicasts that has been found to be useful in distributed MBM programming applications [4, 26]. Existing causal multicast protocols for non-bus-based systems [3, 4, 22, 31] require multiple message rounds. Isotach networks support a single round multicast [34].

• Totally Ordered Multicasts — A totally ordered multicast is a multicast that is received in a consistent order at all processes in the system. Total ordering is useful in MBM computations for a variety of purposes including maintaining consistency of replicated data. Most implementations of totally ordered multicasts obtain the ordering using a serialization point, either in the network itself or in a communication pattern superimposed on the computation, and therefore do not scale well. In an isotach system, totally ordered multicasts in MBM computations are implemented in the same distributed way as flat atomic actions in SMM computations.

Other applications include support for migration mechanisms, checkpointing, wait-free communication primitives, and highly concurrent access to linked data structures. Important special applications include parallel and distributed databases and production systems [34].

Whether the applications of isotach logical time described in this section are practical depends, of course, on the efficiency with which isotach logical time can be implemented. We have identified several very different approaches to implementing isotach logical time. In the next section we describe a distributed algorithm for maintaining isotach logical time that is designed to be implemented in hardware within the network switches and interfaces. In Section 5, we describe the performance of this network.

4. ISOTACH NETWORKS

One way to implement isotach logical time is with an isotach network — a network that delivers messages in an order that allows send and receive events to be assigned times that are

consistent with both the $\rightarrow$ relation and the velocity invariant. Several networks have been proposed that can in retrospect be classified as isotach networks. The alpha-synchronizer network proposed by Awerbuch to execute SIMD graph algorithms on asynchronous networks [1] and a network proposed to support barrier synchronization [2, 10] can be viewed as isotach networks that maintain a logical time system in which each logical time consists of the pulse component only. A network described by Ranade [23, 24] as the basis for an efficient concurrent-read, concurrent-write (CRCW) PRAM emulation can be viewed as realizing the more general and powerful *n*-tuple isotach logical time system.

In this section, we give an algorithm for implementing isotach networks on arbitrary topologies. We begin by describing the *model network,* an isotach network that is easy to reason about but hard to build and then show how to change the network to make it practical.

4.1. Model Network

We consider a network of interconnected nodes in which each node is either a switch or an *element*. For simplicity we assume each element is either a PE or an MM, though isotach systems can be implemented on wrap-around topologies that pair each PE with an MM. Each element contains a switch interface unit (SIU) that connects it to a switch. We require that adjacent nodes communicate over reliable FIFO links. To postpone considering the issue of communication deadlock, we begin by assuming each switch has infinite buffers.

The work of maintaining isotach logical time is done by the switches and the SIUs. The SIUs are also responsible for applying the send order rules. Processes in an isotach system execute asynchronously and need not be aware of the progression of logical time within the SIUs and network. An isotach system requires only a few, easily realized assumptions about the elements: 1) each element signals the end of each flat atomic action; 2) each flat atomic action is

issued atomically (this requirement can be met by atomically passing a pointer from the element to the SIU assuming the atomic action is assembled before the pointer is passed); 3) each element communicates with the associated SIU over a reliable FIFO link; and 4) each element handles messages in the order in which they are delivered. Some applications of isotach networks (e.g., cache coherence) also require that each MM issue responses to operations in the order in which the operations were executed. We assume each message is issued as part of an atomic action. This assumption does not introduce any new or artificial constraints. An atomic action can consist of a single message. We give the model network algorithm in two parts: the first describing the interchange of *tokens* and the second the routing of messages.

Tokens. A *token* in an isotach system is a signal sent to mark the end of one pulse of logical time and the beginning of the next. Each switch stays loosely synchronized with each adjacent switch and SIU by exchanging tokens. Initially each switch sends a token wave, i.e., it sends a token on each output, including the outputs to adjacent SIUs, if any. Thereafter, each switch sends token wave $i+1$ upon receiving the ith token on each input, including the inputs from adjacent SIUs. Each SIU processes the token it periodically receives from the adjacent switch by returning the token to the switch. The token waves can be viewed as the timing mechanism for a distributed logical clock. The pulse component of the local logical time at an SIU is the number of tokens processed and at a switch is the number of token waves sent.

Messages. As each atomic action is issued, for each message m in the atomic action, the SIU 1) timestamps m with a *route-tag* consisting of the *pid* of the issuing process followed by m's rank; 2) determines $t_s(m)$ by computing the earliest pulse in which m can be sent in conformance with the send order rules; and 3) adds m to the list of messages already scheduled to be sent in $t_s(m)$, maintaining the list in route-tag order, i.e, in increasing order first by the *pid* of the

source process and second by rank. Determining $t_s(m)$ is trivial in the case of equidistant networks and requires a small constant number of steps per operation otherwise. Note that if a single process is running on the element, m can simply be appended to the list for $t_s(m)$.

Each time it processes a token, the SIU sends the messages scheduled to be sent in the new pulse and receives the messages delivered in the pulse, i.e., all messages up to the next token. The SIU interleaves sending with receiving so as to handle the messages in route-tag order. We will show that the messages delivered by the network to each SIU in each pulse arrive in route-tag order. Since the messages to be sent are also in route-tag order, the SIU can decide which message to handle next by comparing the route-tags of the next messages to be sent and received. If its network input holds neither a message nor a token, the SIU does not know the route-tag of the next message to be received and must wait. If the network input holds a token, the SIU sends all the messages remaining to be sent in the pulse before processing the token.

The switches in an isotach network also handle messages in route-tag order within each pulse. Each switch routes messages in the same way as a switch in a conventional network except that it merges the streams of messages arriving on its inputs in each pulse by route-tag so that the messages it sends on each output in each pulse are sent in route-tag order. As in the case of an SIU, a switch must wait if one of its inputs holds neither a token nor a message.

To show that the model network is an isotach network we 1) give a rule for assigning logical times to send and receive events and 2) prove that this assignment is consistent with the velocity invariant and the $\rightarrow$ relation. Let $s\_pulse(m)$ denote the pulse component of the logical time at the sending SIU when it sends m and $r\_pulse(m)$ the pulse of time at the receiving SIU when it receives m. Time assignment rule: for each message m, $t_s(m) = (s\_pulse(m), pid(m), rank(m))$ and $t_r(m) = (r\_pulse(m),\ pid(m), rank(m))$.

THEOREM 2. *The model network maintains isotach logical time.*

PROOF. Since a message routed by a switch in pulse i is routed at the next switch in pulse $i+1$, for any message m, $r\_pulse(m)$ - $s\_pulse(m) = d(m)$. Since $t_s(m)$ and $t_r(m)$ differ only in the pulse component, the velocity invariant holds. Showing that the logical time assignment is consistent with the $\rightarrow$ relation requires showing 1) for each message m, $t_r(m) \geq t_s(m)$; and 2) for each SIU, the times assigned to local events are non-decreasing, i.e., local logical time doesn't move backwards. The first part follows directly from the velocity invariant. Since for any message m, $d(m) \geq 0$, $t_r(m) \geq t_s(m)$ by the velocity invariant. Since each SIU sends messages in route-tag order, and each switch merges the streams of messages arriving on its inputs so that it maintains route-tag on each output, a simple inductive proof over the number of switches through which a message travels shows that messages received at an SIU within each pulse are received in route-tag order. Thus the logical times assigned to receive events at each SIU are non-decreasing. Since an SIU interleaves processing sends and receives so as to handle the events in route-tag order, the times assigned to these events at each SIU are non-decreasing. □

4.2. Implementation Issues

Although the model network algorithm given above is practical in some ways, e.g., the SIU algorithm requires a small constant number of steps per message, it is impractical in others. In this section we discuss decreasing bandwidth overhead, preventing deadlock without assuming infinite buffers, improving latency and throughput, and providing fault-tolerance.

Bandwidth. The route-tags pose a problem in relation to bandwidth overhead because the rank component of the route-tags is an unbounded value. The rank component can however be eliminated or reduced to a single bit. Note that the rank component is used in the model network only to ensure routing in route-tag order. (Rank is also used in assigning logical times, but the

time assignment is a preliminary to the proof, not part of the algorithm.) In a network with a single routable path for each source/destination pair, the rank component is unnecessary since the relative rank of messages from the same process is implicit in the order in which they arrive at a switch. In a network with multiple routable paths between source/destination pairs, the rank component can be represented as one bit if each process is limited to no more than two messages per pulse per variable (SMM) or destination process (MBM). An alternative approach is to place no limit on the processes, but require that the SIUs send a single large message in place of multiple small messages that would otherwise require a rank tag to ensure that they are received in sequence. The bandwidth required for the *pid* component of route-tags is not a cost of maintaining isotach time because messages are typically required to carry the identity of the issuing process for reasons unrelated to maintaining isotach time. Tokens can be piggybacked on messages at a cost in bandwidth of one bit per message. Thus the bandwidth overhead attributable to maintaining isotach time is one or two bits per message.

Latency and Throughput. The principal cost of implementing isotach logical time by means of the model network algorithm is the additional waiting implied by the requirement that switches route messages in route-tag order. One way to reduce this waiting is by using *ghosts*. A ghost [23] or null message [6] is a copy of a message with a control bit set to indicate it is not a real message. When a switch sends an operation on one output it sends a ghost with the same route-tag on each of its other output(s). A switch receiving a ghost knows all further operations it receives on the same input will have a larger route-tag than the ghost and this knowledge may allow it to route the operation on its other input(s). Ghosts improve isotach network performance by allowing information about logical time to spread more quickly and are needed in some networks to avoid deadlock. Since a ghost can always be overwritten by a newer message, ghosts take up only unused bandwidth and buffers.

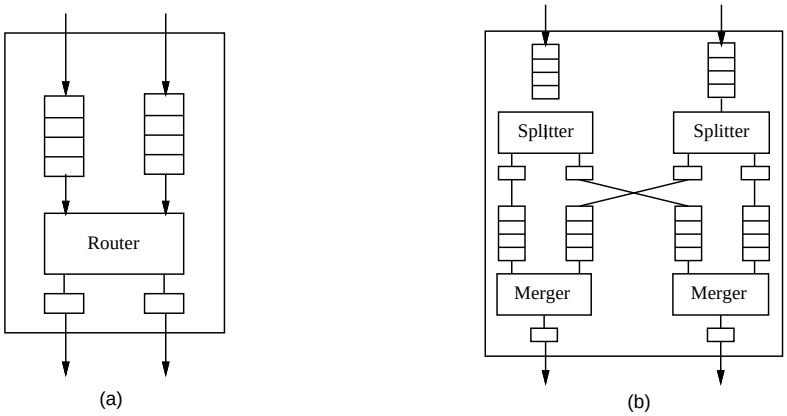


Fig. 3. Switch Designs. (a) Simple switch. (b) Switch with internal buffers.

Another way to reduce waiting in the switches is to change the placement of buffers within the switch. In a switch with input queueing only, such as the switch shown in Fig. 3(a), a blocked message at the head of an input queue blocks subsequent messages arriving on the same input, a phenomenon known as head of line (HOL) blocking. Buffers placed internally, as shown in Fig. 3(b), reduce HOL blocking [12, 15]. Unless the internal buffers are full, a message in a switch with internal buffers blocks other messages only if they are contending for the same output. The internal buffer design adapts easily to isotach networks and not only reduces HOL blocking but also makes it possible for a switch to route an operation on every output every switch cycle. By contrast, an isotach switch with the design shown in Fig. 3(a), can route at most one operation per switch cycle. Our simulation study of isotach networks shows that internal buffers are of significantly more benefit in isotach networks than in conventional networks.

Deadlock. The requirement that switches route messages in route-tag order makes deadlock freedom a harder problem in isotach networks than in conventional networks. The model network uses infinite buffers at the switches to prevent deadlock. One way to bound the number of buffers is to limit the number of messages sent per PE per pulse. Although such

bounds may be useful for other reasons, the number of buffers required to ensure deadlock freedom using this approach is very large. A better approach is to find routing schemes that ensure deadlock freedom. We have proven deadlock free routing schemes for equidistant networks using one buffer per physical channel and for arbitrary topologies using $O(d)$ buffers and virtual channels per physical channel, where d is the network diameter [32].

Fault tolerance. One way to tolerate faults in a computer system is to provide extra nodes and connections which may be put to use in the event of a fault. Most applications of isotach systems require that each SIU know the logical distance to each node to which it sends messages. Although this requirement implies that the distance between each pair of nodes is static, the path itself need not be. An isotach system can use any adaptive routing algorithm that finds a path of a predeterminable length, a family of algorithms that includes minimal adaptive routing algorithms, i.e. adaptive routing algorithms that always find a minimum length path. Discussion of other issues relating to fault tolerance in isotach networks can be found in [32].

We are currently exploring alternative isotach network algorithms with the goal of finding ways to support gigabit/second data rates. We have identified and are now exploring a number of feasible alternative designs for isotach networks that are consistent with switching techniques such as wormhole routing and virtual channels. Some of these designs push all or most of the work of maintaining isotach logical time onto the SIUs and would allow isotach systems to use off-the-shelf networks.

5. PERFORMANCE

We have studied the performance of equidistant isotach networks using both simulation [28] and analytic modelling [32].

5.1. Simulation Study

Isotach networks do more work than conventional networks — they route messages and deliver them in an order consistent with the velocity invariant and the $\rightarrow$ relation. Thus isotach networks can be expected to have less *raw power* than comparable conventional networks. Raw power, as determined by network throughput and latency, measures the ability of a network to deliver a workload of generic messages without atomicity or sequencing constraints to their destinations. The simulation study is designed to answer the following questions:

(1) How do the raw power of an isotach network and a conventional network relate?

(2) Under what conditions, if any, does an isotach network make up for an expected loss in raw power through more efficient support of synchronization?

The simulation compares isotach systems to more conventional systems that enforce atomicity with two-phase locking (2PL) [8] and that enforce sequential consistency by restricting pipelining. We ran the simulations under a variety of synthetic workloads to capture different atomicity, sequencing, and data dependence constraints among operations.

We simulated four networks: two conventional (C1, C2) and two isotach (I1, I2). Each is composed of 2x2 switches interconnected in a reverse baseline topology. The message transmission protocol is store-and-forward using send-acknowledge [25]. We assumed the network switches operated synchronously but our results are applicable to self-timed networks as well. The networks differ only in the design of the individual switches and the algorithms the switches execute. C1 and I1 use switches with the simple design shown in Fig. 3(a); C2 and I2 use the internal buffer design shown in Fig. 3(b). Each isotach switch uses the same algorithm as the corresponding conventional switch except it selects operations for routing in route-tag order and it sends tokens (piggybacked on operations) and ghosts.

Parameter	Comment	Base Case Value	Other Values Tested
net_size	# network stages	5 (32 PE's/MM's)	4,6,8,10
read_prob	probability that an access is a read	.75	0,.25,.5,.9,1
aa_mean	average atomic action size	3	1-10, 16
aa_cap	cap on # outstanding atomic actions per PE	-	1,3,6,12,16,20,24 unlimited
traffic_model	distribution of accesses	uniform	hot spot, warm spot

Table 1. Simulation Parameters.

All latency results are reported in *cycle units*, which represent physical time. Cycle unit duration depends on low-level switch design and technology. We assume switches in C1 and I1 have the same *best case* time — the time required for an operation to move through a switch assuming no waiting — of one cycle unit, and switches in C2 and I2 have best case times of two cycle units. This assumption does not imply that *average* switch latency is the same in conventional and isotach networks of comparable design. Average switch latency tends to be longer in isotach switches because operations must be routed in route-tag order. Throughout the study, throughput is the average number of operations arriving at memory, per MM, per cycle.

The simulated workloads ranged from one with no constraints on operation execution order to one with atomicity, sequencing, and data dependence constraints. Parameters and their default (base case) values are shown in Table 1. Atomic action sizes are exponentially distributed with mean, aa_mean, truncated at 10 * aa_mean. A more detailed report of the study appears in [28].

5.1.1. Raw Power Series: Series A

We measured the raw power of each of the networks, i.e., throughput and delay under a workload of operations with no atomicity (aa_mean = 1) or sequencing constraints. In this series, workload consists of generic operations — reads and writes are not distinguished. Only

the forward, i.e., PE→MM, network is simulated. Each MM is a sink for operations. We assume memory cycle time is the same as switch cycle time — each MM can consume one operation per cycle. Each process is independent, executing on its own PE. Although the workload has no sequencing constraints, there is no way to relax enforcement of sequential consistency in I1 and I2. Thus series A actually compares isotach networks that enforce sequential consistency with conventional networks that do not.

We simulated each network with a workload in which a PE generates a new operation each cycle with probability *r*. Fig. 4 shows throughput and delay as a function of the offered load *r*. Delay as reported in this graph includes source queueing. Delay is measured from the time an operation is generated until it is delivered to the MM. As can be seen in Fig. 4, C1 and C2 can handle a higher load with less delay than I1 and I2, respectively. Conventional networks have more raw power because isotach network switches are subject to the additional constraint that they must route operations in order by pulse and route-tag. For workloads in which a PE generates an operation whenever its output buffer is empty, the results were the same: per stage delay was consistently lower and throughput consistently higher for C1 and C2, independent of

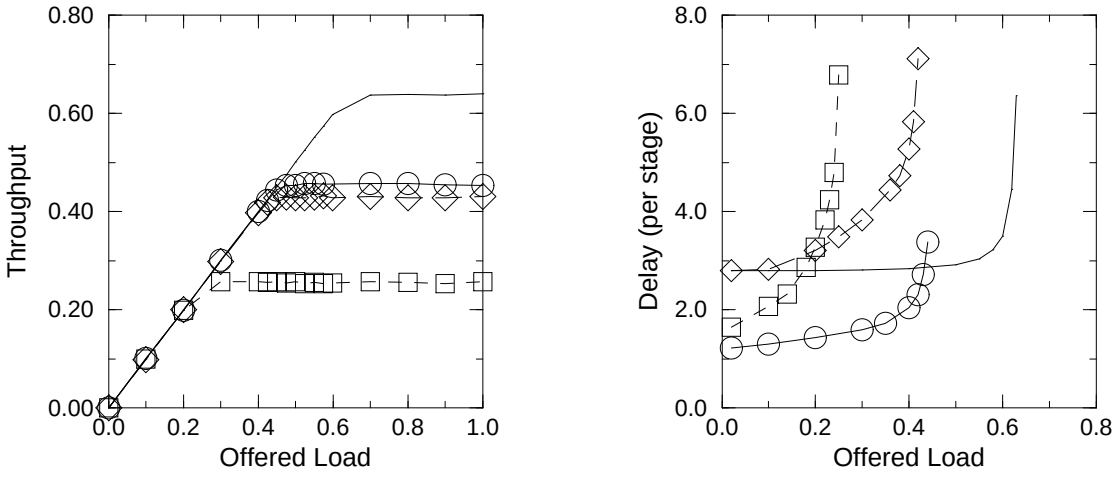


Fig. 4. Series A — Varying offered load. ○ - C1, — - C2, □ - I1, ◊ - I2

the number of stages in the network. Finally, note the throughput/latency tradeoff offered by the switch designs depicted in Fig 3. C2 and I2 offer higher throughput at the cost of higher latency than C1 and I1.

In the remaining series, we report results only for C1, I1, and I2: in all remaining series C1 provides more throughput with less delay than C2. C2 performs poorly relative to C1 because C2 retains its high latency but cannot take advantage of its high throughput. Both C1 and C2 are under-utilized in the remaining series because the workload's synchronization constraints prevent processes from issuing operations fast enough to saturate the network.

5.1.2. Sequential Consistency Only Data: Series B

Series B compares the networks under a workload model requiring sequential consistency but not atomicity. In this series, as in subsequent series, traffic in both the forward (PE→MM) and reverse (MM→PE) directions is simulated. The reverse networks are all conventional networks. I2 uses C2 as the reverse network. I1 and C1 use C1. In each cycle, each MM (PE) can receive one operation (response) from the network and issue one response (operation).

Initially we assume that no data dependences constrain the issuing of operations. In I1 and I2, operations can be pipelined without risk of violating sequential consistency. Each PE issues a new operation whenever its output buffer is empty (aa_cap = unlimited). C1 enforces sequential consistency by limiting each PE to one outstanding operation at a time. Note the isotach network simulations in series A and B are the same except that the reverse network is simulated.

Scalability — Fig. 5 shows the effect of varying the network size. Delay is round-trip delay normalized for network size, i.e., the number of cycles between the time an operation is placed in the PE's output buffer and the time the PE receives the response to the operation, divided by the number of stages (one-way) in the network. Delay for each network is higher in this series than

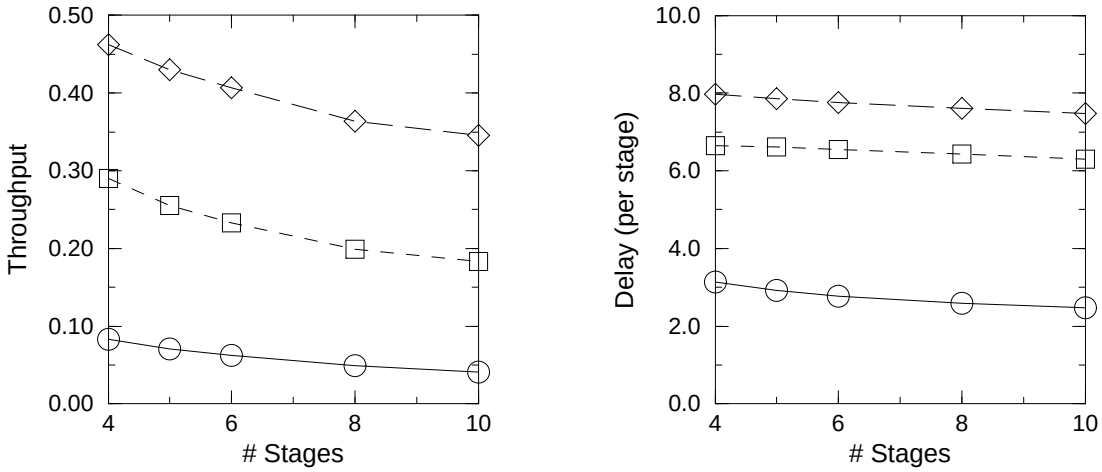


Fig. 5. Series B — Varying the number of network stages. ○ - C1, □ - I1, ◊ - I2

in series A because delay is round trip latency. In the base case (stages = 5), throughput for C1 is about 15% of its series A throughput, reflecting the high cost of enforcing sequential consistency in a conventional network. For I1 and I2, performance is the same as in series A, since the isotach networks enforce sequential consistency in both series. Because they permit pipelining, the isotach networks outperform C1 in terms of throughput, but their delay is longer than C1's, partly because pipelining causes the isotach networks to be more heavily loaded than C1. As network size increases, throughput slowly decreases and delay per stage remains roughly constant.

Data Dependences — Data dependences among operations issued by the same process tend to diminish throughput in isotach networks because they prevent the networks from taking full advantage of pipelining. Fig. 6 shows the effect of changes in data dependence distance (modelled by changes in the aa_cap). When the data dependence distance is 1 (aa_cap=1), meaning each process needs the result of its current operation before it can issue its next operation, the throughput of the isotach networks is low. As the data dependence distance increases, the throughput of I1 and I2 climbs sharply until network saturation. The delay (shown here as total roundtrip delay, not normalized for network size) also increases as the networks become more

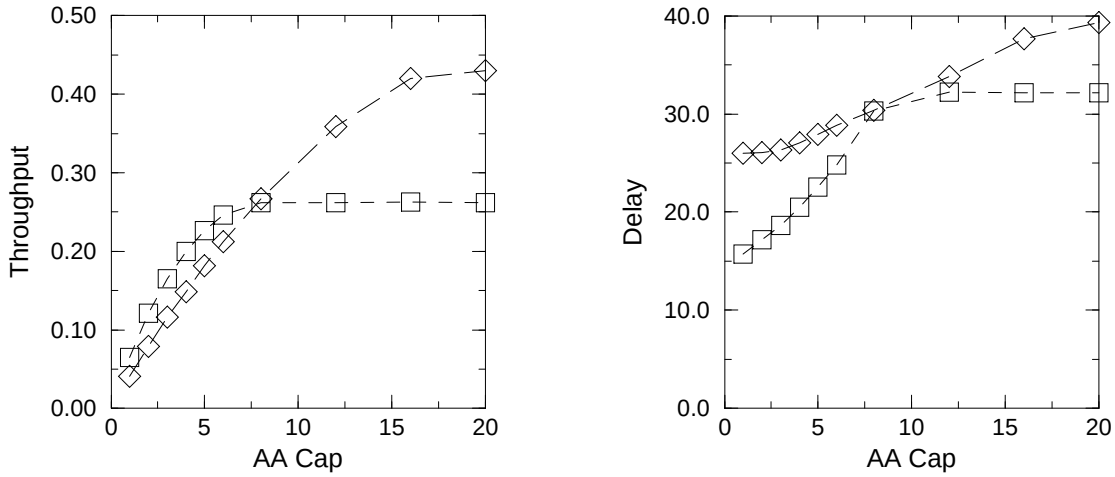


Fig. 6. Series B — Varying data dependence distance. □ - I1, ◊ - I2

heavily loaded. Because its latency is longer and its saturation point higher, I2 requires a larger data dependence distance than I1 to take full advantage of its ability to pipeline. I2 performs less well than I1 at small dependence distances because I2 cannot take advantage of its higher throughput and is hurt by a higher response time due to higher latency. Since C1 cannot pipeline regardless of the data dependence distance, throughput and delay for C1 do not vary with the aa_cap. C1's throughput is 0.6 and delay is 14.75. Thus, although their delay is worse, the throughputs of I1 and I2 begin to exceed C1's at a data dependence distance of only 2.

5.1.3. Atomic Action Data: Series C

Series C compares the performance of isotach and conventional systems under a workload with atomicity and sequencing constraints. Atomic actions are assumed to be flat and independent, i.e., there are no data dependences among or within atomic actions. Isotach systems enforce atomicity by issuing all operations from the same atomic action in the same pulse. The conventional system we simulate enforces atomicity using 2PL. A lock is associated with each variable and a process acquires a lock for each variable accessed by the atomic action before releasing any of the atomic action's locks. To avoid deadlock, each process acquires the locks it

needs for each atomic action in a predetermined linear order.

Our lock algorithms are more efficient than those used in most systems. Instead of spinning, lock requests queue at memory. We distinguish read locks from write locks and only the latter are exclusive. Instead of sending a lock request for an operation, a process sends the operation itself. Each operation implicitly carries a request for a lock of the type indicated by the operation. When it receives an operation, the MM enqueues it. If no conflicting operation is enqueued ahead of it, the operation is executed and a response returned to the source PE. A PE knows it has acquired a lock when it receives the response. Eliminating explicit lock requesting and granting messages reduces traffic and eliminates the roundtrip delay from memory to the process and back when a lock is granted. When the PE has acquired all the locks for the atomic action, it sends a lock release for each lock it holds.

Execution in this series is required to be sequentially consistent. C1 enforces sequential consistency by limiting the number of outstanding atomic actions per PE to one (aa_cap = 1). In I1 and I2 a PE may have any number of active atomic actions without sacrificing sequential consistency (aa_cap = unlimited).

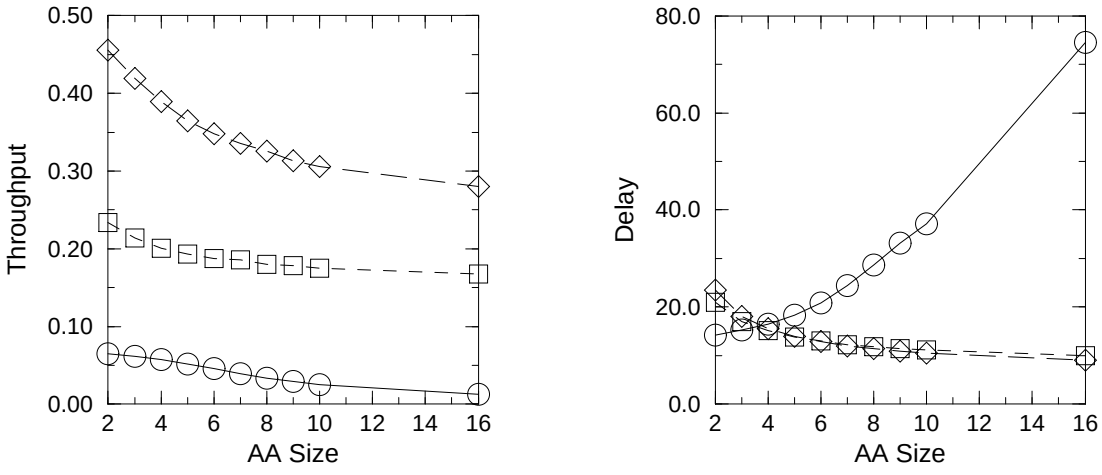


Fig. 7. Series C — Varying atomic action size — uniform traffic. ○ - C1, □ - I1, ◊ - I2

Atomic Action Size — Fig. 7 shows the effect of varying the average size (aa_mean) of atomic actions. Delay is delay per operation, i.e., the average number of cycles from the time an atomic action is generated until it is completed, divided by aa_mean. For C1, the time required to release locks is not included in delay. A process in C1 generates a new atomic action as soon as it has received all the responses due in its current atomic action. In I1 and I2 a process generates a new atomic action whenever its output buffer is empty. The graphs show C1 performs poorly for large atomic actions. As the atomic action mean size increases, C1's throughput drops and its delay rises steeply. In I1 and I2, by contrast, the delay per operation actually decreases as the size of atomic actions increases, since increasing size means more concurrency. For large atomic actions, the isotach systems perform markedly better than the conventional systems. When the average atomic action size is 16, the delay and throughput are both better in isotach systems by an order of magnitude. The steep rise in delay in the conventional systems is attributable to the use of locks. As atomic actions grow larger, not only does the number of operations contending for access grow, but also the length of time each lock is held.

Series C shows that isotach systems perform well in relation to conventional systems under a workload with atomicity and sequencing constraints. Conventional systems cannot take full advantage of the greater raw power of their networks because the rate at which operations can be issued is limited by locks and the restriction on pipelining. The performance of isotach systems, by contrast, is limited only by the raw power of the networks. Though the raw power of the isotach networks is somewhat lower than that of the conventional networks, the synchronization support the isotach networks provide allows the isotach systems to outperform the conventional systems by a wide margin.

We also compared the performance of conventional and isotach systems for a workload of structured atomic actions in which data dependences exist both within and among atomic actions [28]. The isotach networks continued to outperform C1, though by a smaller margin since the data dependences prevent the isotach networks from taking advantage of their ability to pipeline operations.

5.1.4. Non-Uniform Traffic Model Data: Series D

Series D compares performance of the networks under the flat atomic action workload model from series C using a warm-spot traffic model. A warm-spot traffic model has several *warm* variables instead of a single hot variable and is more realistic than either the uniform or hot-spot traffic models. The warm-traffic model we use is based on the standard 80/20 rule (see e.g. [13]), modified to lessen contention for the warmest variables [28].

Fig. 8 shows the effect of varying atomic action size under a warm-traffic model. Delay is reported in the same way as in series C. The performance of I1 and I2 is similar to their series C performance. The performance of C1, in contrast, is considerably worse in this series than in the

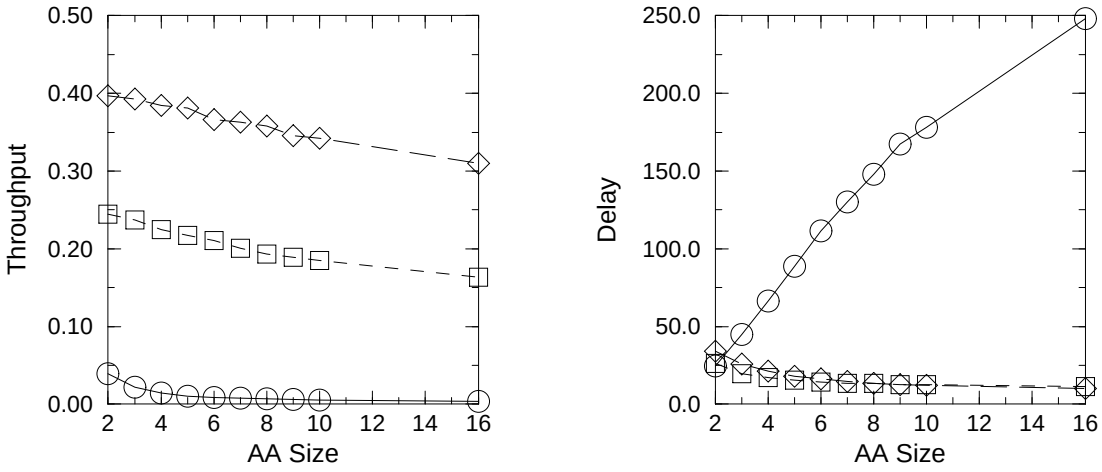


Fig. 8. Series D — Varying the atomic action size — warm spot traffic. ○ - C1, □ - I1, ◊ - I2

previous series. When the probability of conflicting operations is higher, locks have more impact on performance. The use of locks to enforce atomicity means C1's performance suffers when accesses are not uniformly distributed. When atomic actions are large and traffic is non-uniform, C1's performance is very poor relative to I1 and I2. For large atomic actions (aa_size = 16), throughput for I2 is about 78 times higher than C1's and its delay about 24 times lower. Hot spot traffic results show the same pattern.

Series D shows that isotach systems outperform conventional systems by a wider margin when accesses are not uniformly distributed, particularly when atomic actions are large.

5.1.5. Simulation Study Summary

Our study shows conventional networks have higher raw power than isotach networks, but with atomicity and sequencing constraints, isotach networks outperform conventional networks, in some cases by a factor of ten or more. Conventional systems perform relatively poorly in spite of their greater raw power because the means by which they enforce atomicity and sequencing constraints work by restricting throughput and imposing delays. Isotach networks perform best in relation to conventional networks when the distance between data dependent operations within the same process's program is large enough to permit pipelining; atomic actions are large; and contention for shared variables is high. When the workload has two or more of these characteristics, an isotach network performs markedly better than a conventional network. These results indicate that isotach based synchronization techniques are worthy of further study. We intend to extend our simulation to include isotach based cache coherence protocols [33].

5.2. Analytic Model

Analytic models were developed for the four simulated networks (C1, C2, I1, I2). That work led to the discovery of novel modelling methods that will be presented in a future paper. We outline the approach and results we obtained here. Interested readers are referred to [32] for a more detailed presentation.

We extended a mean value analysis technique first presented by Jenq [11] for banyon-like MIN's. Our extensions enabled the modelling of 1) routing dependencies, 2) a measure of bandwidth we have called *information flow*, and 3) message types. Routing dependencies exist in isotach networks in the form of constraints requiring routing in route-tag order (Jenq did not include this kind of dependence). Information flow is a measure of the bandwidth of a given stage of a MIN. Including information flow allowed us to introduce independence among network stages, gaining tractability with little sacrifice in accuracy. Finally, certain optimizations that would be made in an implementation of an isotach network, for example, piggy-backing of tokens on messages of content and ghost messages, needed to be reflected in the analytic model. We succeeded in capturing the effects of these message types.

For networks I1 and I2 the analytic model predicted throughput with no worse than 12% deviation from the simulation results presented earlier. The model tended to be optimistic, which we expected because it didn't account for delays that variations in uniform distribution of messages would create in the network. Those delays were captured in the simulation. Prediction of delay was even better with a maximum error of 5%. In both cases the results were for networks ranging in depth from two to ten stages.

6. CONCLUSION

We have proposed a new class of networks called *isotach networks* for the purpose of supporting concurrency control. Isotach networks implement isotach logical time, a system of logical time that assigns times to send and receive events that are consistent with Lamport's $\rightarrow$ relation and with the velocity invariant, an invariant that relates communication time to communication distance. This invariant provides a powerful coordinating mechanism that allows processes to execute atomic actions without locks and to pipeline operations without sacrificing sequential consistency. We described a distributed algorithm for implementing isotach logical time designed to be implemented in hardware within the network switches and interfaces and reported the results of analytic and empirical studies of this algorithm that show that isotach systems outperform conventional systems under realistic workloads.

The approach to concurrency control presented in this paper reduces the problems of enforcing each of several basic correctness properties — e.g. atomicity, sequential consistency, memory coherence — to the problem of implementing isotach logical time. We have described a practical design for an isotach network and shown that it performs well in relation to conventional systems, but the network described is just one of many ways to implement isotach logical time. We are currently studying alternative implementations of this promising new approach to concurrency control.

REFERENCES

[1] B. Awerbuch, ‘‘Complexity of Network Synchronization’’, *J. ACM 32*, 4 (October 1985), 804-823.

[2] Y. Birk, P. B. Gibbons, J. L. C. Sanz and D. Soroker, ‘‘A Simple Mechanism for Efficient Barrier Synchronization in MIMD Machines’’, Tech. Rep. RJ 7078, IBM, October 1989.

[3] K. P. Birman and T. A. Joseph, ‘‘Reliable Communication in the Presence of Failures’’, *ACM Trans. Computer Systems 5*, 1 (February, 1987), 47-76.

[4] K. P. Birman, A. Schiper and P. Stephenson, ‘‘Lightweight Causal and Atomic Group Multicast’’, *ACM TOCS*, August 1991, 272-314.

[5] K. Birman, ‘‘A Response to Cheriton and Skeen’s Criticism of Causal and Totally Ordered Communication’’, *OS Review 28*, 1 (January 1994), 11-21.

[6] K. Chandy and J. Misra, ‘‘Distributed Simulation: A Case Study in Design and Verification of Distributed Programs’’, *IEEE TRANS on Software Engineering 5*, 5 (September 1979), 440-452.

[7] D. R. Cheriton and D. Skeen, ‘‘Understanding the Limitations of Causally and Totally Ordered Communication’’, *OS Review, 14th ACM Symp. on OS Principles*, December 1993, 44-57.

[8] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, ‘‘The Notions of Consistency and Predicate Locks in a Database System’’, *Comm. ACM 19*, 11 (November 1976), 624-633.

[9] K. Gharachorloo, et al., ‘‘Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors’’, *4th ASPLOS*, April, 1991, 245-257.

[10] P. B. Gibbons, ‘‘The Asynchronous PRAM: A Semi-Synchronous Model for Shared Memory MIMD Machines’’, 89-062, ICSI, Berkeley, California, December, 1989.

[11] Y. Jenq, ‘‘Performance Analysis of a Packet Switch Based on Single-Buffered Banyan Network’’, *IEEE Journal on Selected Areas in Communications 1*, 6 (December 1983), 1014-1021.

[12] M. J. Karol, M. G. Hluchyj and S. P. Morgan, ‘‘Input Versus Output Queueing on a Space-Division Packet Switch’’, *IEEE Transactions on Communications 35*, 12 (December 1987), 1347-1356.

[13] D. E. Knuth, in *Fundamental Algorithms*, vol. 3, Sorting and Searching , Addison-Wesley, 1973, 397.

[14] C. P. Kruskal, L. Rudolph and M. Snir, ‘‘Efficient Synchronization on Multiprocessors with Shared Memory’’, *ACM Trans. Prog. Lang. and Systems 10*, 4 (October 1988), 579-601.

[15] M. Kumar and J. R. Jump, ‘‘Performance Enhancement of Buffered Delta Networks Using Crossbar Switches and Multiple Links’’, *Journal of Parallel and Distributed Computing 1* (1984), 81-103.

[16] L. Lamport, ‘‘Time, Clocks, and the Ordering of Events in a Distributed System’’, *Comm. ACM 21*, 7 (July 1978), 558-565.

[17] L. Lamport, ‘‘How to Make a Multiprocessor Computer That Correctly Executes Multiprocessor Programs’’, *IEEE Trans. on Computers 28* (1979), 690-691.

[18] A. Landin, E. Hagersten and S. Haridi, ‘‘Race-Free Interconnection Networks and Multiprocessor Consistency’’, *Proc. 18th ISCA*, 1991, 106-115.

[19] D. B. Lomet, ‘‘Process Structuring, Synchronization, and Recovery Using Atomic Actions’’, *SIGPLAN Notices 12*, 3 (March 1977), 128-137.

[20] S. Owicki and D. Gries, ‘‘An Axiomatic Proof Technique for Parallel Programs I’’, *Acta Informatica 6* (1976), 319-340.

[21] C. Papadimitriou, *Database Concurrency Control*, Computer Science Press, 1986.

[22] L. L. Peterson, N. C. Bucholz and R. D. Schlichting, ‘‘Preserving and Using Context Information in Interprocess Communication’’, *ACM Trans. Computer Systems 7*, 3 (August 1989), 217-246.

[23] A. G. Ranade, ‘‘How to Emulate Shared Memory’’, *IEEE Annual Symp. on Foundations of Computer Science*, Los Angeles, 1987, 185-194.

[24] A. G. Ranade, S. N. Bhatt and S. L. Johnsson, ‘‘The Fluent Abstract Machine’’, Tech. Rep. 573, Yale University, Dept. of Computer Science, January, 1988.

[25] D. A. Reed and R. M. Fujimoto, *Multicomputer Networks; Message-Based Parallel Processing*, The MIT Press, Cambridge, Massachusetts, 1987.

[26] R. Renesse, ‘‘Casual Controversy at Le Mont St.-Michel’’, *Operations System Review 27*, 2 (April 1993), 44-53.

[27] R. Renesse, ‘‘Why Bother With CATOCS?’’, *OS Review 28*, 1 (January 1994), 22-27.

[28] P. F. Reynolds, Jr., C. Williams and R. R. Wagner, Jr., ‘‘Empirical Analysis of Isotach Networks’’, Tech. Rep. 92-19, University of Virginia, Dept. of Computer Science, June, 1992.

[29] J. H. Saltzer, D. P. Reed and D. D. Clark, ‘‘End-To-End Arguments in System Design’’, *ACM TOCS 2*, 4 (November 1984), 277-288.

[30] C. Scheurich and M. Dubois, ‘‘Correct Memory Operation of Cache-Based Multiprocessors’’, *Proc. 14th ISCA*, June 1987, 234-243.

[31] A. Schiper, J. Eggli and A. Sandoz, ‘‘A New Algorithm to Implement Causal Ordering’’, in *Distributed Computing*, vol. 89 , Springer-Verlag, Berlin-Heidelburg-New York, 1989, 219-232.

[32] R. R. Wagner, Jr., ‘‘On the Implementation of Local Synchrony’’, CS-93-33, University of Virginia, Dept. of Computer Science, June 1, 1993.

[33] C. Williams and P. F. Reynolds, Jr., ‘‘Delta-Cache Protocols: A New Class of Cache Coherence Protocols’’, Tech. Rep. 90-34, University of Virginia, Department of Computer Science, December, 1990.

[34] C. C. Williams, ‘‘Concurrency Control in Asynchronous Computations’’, Ph.D. Thesis, University of Virginia, 1993.

[35] C. C. Williams and P. F. Reynolds, Jr., ‘‘Combining Atomic Actions’’, *Journal of Parallel and Distributed Computing (to appear)*, 1995.