On the Implementation of Local Synchrony

Raymond R. Wagner, Jr.

UVA Computer Science Technical Report
TR-93-33

June 1993

On the Implementation of Local Synchrony

Raymond R. Wagner, Jr.

Abstract

Local synchrony is a distributed approach to providing logically synchronous capabilities in an asynchronous system. Local synchrony provides a logical timeline to memory access, and thus a total ordering to the steps of a distributed computation, without many of the inherent consequences of global synchronization. Local synchrony offers a framework for the implementation of various concurrency control mechanisms, including combining and isochrons (parallel operations). We show the implementability of local synchrony, and demonstrate its compatibility with existing protocols for fault-tolerance in general purpose multiprocessors. The results of our research are threefold. We show first that practical implementation of local synchrony is feasible in general, asynchronous, MIMD architectures. We show that such implementations are compatible with existing hardware fault tolerance systems and present a correct rollback algorithm for our general implementation. Finally, we present both analytic and empirical studies which illustrate the performance characteristics of such implementations. Our analytic modeling technique extends in novel ways probabilistic modelling methods often employed for interconnection networks. Our work establishes the practicality and feasibility of local synchrony as a means of providing support for concurrency control.

Table of Contents

1.0 Introduction

A major focus of research in parallel processing is global memory: where to situate it, how to access it, and how to maintain its coherence. An important emphasis in any parallel architecture design is the speed with which processing elements in the system may access global memory. To a large extent, the speedups in processing power to be gained by using multiple processors to attack a single problem are directly related to memory cycle time. The longer memory cycle times of parallel processing architectures place severe limits on the processing speedups that can be achieved by utilizing the concurrency of such machines.

Much attention has been focused on improving memory access speed in both shared-memory and message-passing architectures. The proposals made by researchers in this field generally ask “In what ways can we shorten the cycle time of a single memory access in a multiprocessor system?” Their solutions include speeding up the transfer of a message through a given system by implementing better routing algorithms, and/or distributing memory in ways that reduce slowdowns caused by message traffic bottlenecks. Other work has proposed introducing multiprocessor cache memories [YYF85] [DSB88] [BNR89] [RAK89] and distributing memory [BaJ88], or balancing the message load [ReT86] to limit contention.

Another approach would be to ask: “Can we achieve better system utilization by increasing the amount of productive activity during global memory references, and reducing the amount of inter-process synchronization required?” Combining systems [GGK83] [GLR83] [Pfi85] [RBJ88] [TuR88], which reduce actual memory traffic by merging accesses to the same global variable, take this approach.

A method along these same lines would be to limit non-productive global memory references, thus freeing up system resources that could then be used for productive activity. Concurrency control requirements can generate a significant amount of the global memory traffic in a parallel computation. Concurrency control traffic only indirectly performs useful work, by facilitating (in a way that meets the synchronization requirements) the execution of later productive

accesses. When multiple entities (i.e., processes) are competing for the use of multiple resources (i.e., global variables), extreme care must be taken to enforce proper accessing principles. Concurrency control principles guarantee that the result of the parallel computation will be equivalent to some possible sequential execution. Our work seeks to limit or eliminate non-productive global memory accesses by building into the system architecture the concurrency control requirements necessary for correct parallel execution.

In [RWW89], we addressed the problem of parallel atomic access to global memory. A system that provides atomic access to multiple global memory locations eliminates much of the inter-process synchronization currently necessary in parallel programming. To provide such atomic access, we must enforce a specific ordering of distributed events. In asynchronous systems it is generally impossible to order events at different locations by their physical occurrence [Lam79]. However, if events are arbitrarily ordered and executed such that all sequencing (atomicity and sequential consistency) constraints are met in the same way at all locations, the effect will be as if all events occurred with the correct synchronization in force.

In order to implement parallel memory access operations, or isochrons, Williams [Wil93] proposed a system of logical clocks, which provide a logical ordering of global memory accesses, in turn facilitating the synchronization necessary to guarantee atomicity and sequential consistency of parallel memory accesses. This system of logical clocks describes a logical timestamp ordering system for the global memory accesses of a parallel computation.

One method of implementation for this system of logical clocks is by:

1. Assigning a unique logical timestamp to each global memory access. The logical timestamps are designed to provide a total ordering of global memory accesses meeting all sequencing constraints.
2. Constraining all elements within the system to process global memory accesses in logical timestamp order. Thus accesses are emitted from the processes in order, kept ordered as they traverse the network, and executed in order when they reach global memory.

We call this method *local synchrony.* We are interested in this conservative logical timestamp ordering system for several reasons: it can be implemented using only local decision-making

by elements within the system; the timestamping system can be implemented implicitly, so an access need not be significantly larger than already necessary for conventional routing; and the system does not require elements to reach physical barriers in order to achieve synchronization.

Consider a shared memory architecture consisting of a number of processing elements (PEs) connected to a number of memory modules (MMs) by an interconnection network of switching elements. PEs communicate only by sending accesses to global memory. In a local synchrony implementation, each PE assigns a unique identifier to each access. The identifier acts as a timestamp for the access' execution at the MM. PEs release accesses in ascending order by timestamp. Switching elements merge incoming streams of sorted accesses to keep all access streams in timestamp order. MMs execute accesses in timestamp order. Nodes (PEs, MMs, switching elements) stay in loose synchronization because they must process accesses in order, which means that they must have input from each neighbor in order to make processing decisions. This is local synchrony.

An example of a local synchrony implementation is an isotach network [RWW89] [WiR91]. Such an implementation provides hardware support for the task of controlling the concurrency of parallel computations—in particular, in enforcing atomicity and sequential consistency.

Atomicity and sequential consistency are important properties of parallel computations that are expensive to enforce using existing techniques. The atomic action is a group of one or more operations issued by the same process that *appears* to be executed indivisibly, without interleaving with other operations.

Atomic actions are typically specified by operations on locks or semaphores or by critical sections or monitors implemented with locks or semaphores. Existing hardware support for atomic actions is designed to make locking more efficient. The principle drawback to the use of locks in implementing atomicity is the unnecessarily restricted access to shared variables implied by locking. Variables cannot in general be partitioned so that each atomic action can, by acquiring a single

lock, control all the variables it must access and no others. Atomic actions either acquire a single lock, and lock some variables unnecessarily, or acquire multiple locks. To avoid deadlock, an atomic action that requires multiple locks must typically obtain the locks sequentially. During this lock acquisition phase, the variables controlled by already-acquired locks are unavailable to other processes. A local synchrony implementation enforces atomicity without using locks or semaphores.

Sequential consistency means that the order in which accesses are executed is consistent with the order specified by each individual process's sequential program [Lam79]. Maintaining sequential consistency is a problem in multiprocessors because stochastic delays in the network allow operations issued by the same process to arrive at global memory in an order inconsistent with the order in which operations were issued. The simplest solution, which would be to disallow the pipelining of memory accesses, is undesirable because pipelining is an important way to lessen effective memory latency. A local synchrony implementation enforces sequential consistency without restricting the pipelining of operations.

A local synchrony implementation can be used to implement data-dependent atomic actions (where one or more actions within an atomic action are dependent upon the results of one or more of the other constituent actions) using split operations [WiR89]. Williams has described a family of cache-coherency protocols [Wil93] based on isotach networks. Local synchrony also facilitates FIFO combining [Ran87], which allows access combining to be implemented without the costly associative search necessary in conventional systems. In light of the many possible uses of local synchrony implementations, it is desirable to know whether local synchrony can be implemented correctly and efficiently, in order to provide the benefits described above to parallel computations.

Our work concerns the implementation of local synchrony. We address implementation issues for local synchrony on general parallel processing architectures. We discuss some of the fault-tolerance capabilities of a local synchrony implementation. We also provide both analytical

and simulation studies that model and evaluate the performance of local synchrony, compared to conventional implementations, for several operating environments.

Previous work has provided some general implementation information for local synchrony [RWW89] [Wil90]. We explore this area in depth. We present an implementation plan for local synchrony on an NYU Ultracomputer-type architecture, including operative algorithms, a low-level switch design realizable with current technology, and a proof of deadlock freedom. We then present a general implementation that can be realized on any parallel architecture and give a formal proof of correctness for the technique. Our general solution is a significant advance over other possible approaches because it provides greater opportunities for parallelism with smaller memory requirements.

A locally synchronous system has several properties which adapt well to current fault-tolerance research in both hardware and software. We explore these properties and how they may be advantageously exploited. We then present and prove correct a local synchrony checkpointing and rollback procedure that is compatible with our general implementation technique.

In order to demonstrate that local synchrony implementations are both valuable and feasible, we must show convincingly that the performance costs of implementing such a system are moderate compared to the benefits gained. We first present analytical models of local synchrony implementations by advancing existing modelling techniques. These analytical studies validate our simulation studies of the same implementations. The simulation studies compare local synchrony implementations to conventional implementations under workloads that include atomicity and sequential consistency constraints. We show that, while the conventional implementations outperform local synchrony in situations where these constraints are minimal, local synchrony outperforms by a significant factor conventional systems under workloads with high atomicity and sequencing constraints.

The remainder of this dissertation is as follows:

Chapter 2 contains an overview of MIMD architectures, background information on local synchrony and related topics, and discussion of fault-tolerance and performance issues related to the implementation of local synchrony.

Chapter 3 includes a motivation for our work and a specific implementation of local synchrony, including a proof of deadlock freedom and a low-level switch design for the communication network. Also we present an alternate deadlock-freedom proof, covering certain architectures, that does not require the use of ghost messages.

Chapter 4 presents a general implementation technique that allows local synchrony to be implemented on most parallel processing architectures. We prove that implementations developed using our technique are correct and deadlock-free, and show that our technique provides more opportunities for parallelism and uses less memory than other approaches.

Chapter 5 discusses the fault-tolerance capabilities of local synchrony implementations and presents a simple scheme to provide a measure of fault tolerance in the presence of hardware redundant paths. Then we present a checkpointing and rollback procedure, which provides rollback capability for the general implementation, and prove the algorithm correct.

Chapter 6 presents several analytical models of local synchrony implementations on an NYU Ultracomputer-type architecture using both a low-throughput and a high-throughput switch design. We use these analytical models to validate our later simulation studies. The simulation studies compare the performance of local synchrony and conventional implementations under various workloads with varying degrees of sequencing constraints. We also compare local synchrony to another timestamp ordering implementation, *ltu ordering*.

Chapter 7 presents conclusions and an overview of the future work we intend for this area.

2.0 Background

This section presents relevant topics, literature review, and a background overview as an introduction to our area of research. Sections 2.1 and 2.2 present an overview of concurrent architectures, define several terms which will be of importance in later discussions, and discuss the nature of communication in a multiprocessor environment. These sections act as an introduction to the general area of our work.

Section 2.3 introduces combining systems and reviews relevant research in this area. Section 2.4 presents Williams' [Wil93] approach to concurrency control. This is the main theoretical background for our work. The basis of our work is the implementation of a concurrency control mechanism which meets Williams' constraints of atomicity and sequential consistency, and ensures conflict-free serial schedules [Wil93] for any execution. These conditions are discussed in this section.

Section 2.5 presents the concurrency control mechanism *isochrons* [Wil93]. Isochrons are a concurrency control mechanism, designed for multiprocessing systems, which are compatible with the concurrency control approach defined in section 2.4. Isochrons provide a limited form of atomic access in multiprocessing architectures.

Section 2.6 presents local synchrony as a method for the implementation of isochrons. The main thrust of this dissertation is research leading to implementations of local synchrony on interesting multiprocessing systems, which will facilitate the implementation of isochrons and other high-level concurrency control mechanisms. Section 2.7 reviews research related to concurrency control, isochrons, and local synchrony.

Sections 2.8 and 2.9 present previous work on fault tolerance and performance issues in multiprocessing systems.

2.1 MIMD Architectures

Our research deals with asynchronous multiprocessing computers utilizing the *Multiple-Instruction Stream, Multiple-Data Stream* (MIMD) [Fly66] system architecture. A MIMD proces-

sor generally consists of a number of nodes connected by communication channels that are–usually–organized in a regular pattern (e.g., a hypercube). Each node may include zero or more processing elements and zero or more memory elements, as well as the routing hardware necessary for each of the node's input/output channels. Each processing element may execute a different program—as opposed to *Single-Instruction Stream, Single-data Stream* (SIMD) architectures, where each element executes the same program, albeit with different data.

This model encompasses both *message-passing* systems, in which processes communicate by sending messages to each other, with each processing node 'owning' a section of global memory, and *shared-memory* systems, in which processes communicate through global memory, which is shared. Message-passing systems may be cast as a subset of shared memory systems in which any node containing a section of global memory also contains a processing element which controls access to that memory. We choose to use the shared-memory model for our discussion and research, given the ease with which message passing can be implemented in it.

Multiprocessing systems may be either *synchronous*, in which processing occurs in lock-step phases governed by a global clock, or *asynchronous*, in which processing elements operate independently. The operation of a synchronous system is well suited to applications which are highly data-parallel in nature, consisting of groups of similar processes performing similar operations. These types of systems are very close to SIMD systems in operation. Many applications, however, are not easily cast in the data-parallel paradigm. These applications are better suited to the asynchronous model. Our work investigates the efficiency and fault-tolerance of local synchrony as a way to achieve the benefits of synchronous processing on asynchronous processors, without unduly compromising the benefits (e.g., performance) of asynchronous operation.

Padmanabhan [Pad90] distinguishes between *direct* and *indirect* interconnection schemes. In a direct interconnection scheme, processing elements are connected in a network using point-to-point links. An indirect scheme separates the architecture into processors and memory (or combined processing/memory nodes) and a switching network. Generally, indirect architectures are

designed under the shared memory paradigm, while direct architectures are usually message passing systems. Exceptions to this rule include the indirect binary cube, which is an indirect message passing design.

Equidistant architectures are a special kind of multiprocessing system. *Distance* refers to the number of network stages or point-to-point steps which a given memory access must traverse in order to reach its destination. In an equidistant system, the distance an access requested by any processing element travels to any shared memory location is constant for all processing elements and memory locations. Equidistant architectures generally take the form of shared-memory, indirect machines (e.g., the NYU Ultracomputer [GGK83]).

Non-equidistant systems are usually directly interconnected and include most message-passing and several shared memory designs. In a non-equidistant architecture, a node generally includes both processing and memory elements, along with switching hardware. Examples of non-equidistant architectures are numerous and include hypercubes [Sei85], cube-connected cycles [PrV81], rings, toroids, and meshes.

2.2 Routing, Message Transfer, and Deadlock

A multiprocessor with a *uni-path* network architecture has a unique physical path from any processing element to any memory element. Uni-path systems tend to be limited to equidistant systems, but neither is a requirement for the other. The Ultracomputer is again a good example of this class of architectures. Routing decisions in such a system are straightforward. In order to reach its destination, a message has only one possible choice of path at a given point, so routing decisions may be hard-wired into the switch. With the advent of hardware redundancy schemes for fault tolerance (section 2.8), this type of system has largely been replaced by *multi-path* systems.

In a multi-path system, a message bound for a given memory location may travel any of a number of possible paths in order to reach its destination. Routing schemes for such systems traditionally emphasize deadlock freedom first. A second priority for these systems is to route messages along the shortest possible paths in order to limit network latency. Since much of our work

involves multi-path routing systems that can be implemented in the hypercube environment, we discuss several types of hypercube routing schemes here.

Each node in an *n*-dimensional binary cube is labeled with an *n*-digit binary number and is directly connected to *n* of its neighbors. The label assigned to each of these neighbors differs from that of the given node in one binary digit, or one dimension of the cube. Thus, the Hamming distance—or length of the shortest path—between any two nodes is the number of digits with different addresses.

A simple routing scheme in this environment would compare the current node address of the message with its destination, then route the message along one of the output lines crossing a dimension that the message must cross in order to reach its destination. This scheme is provably minimal. The selection of possible lines along which to route the message can be non-deterministic, or can take into account any criteria, such as network load or faulty lines.

Dally and Seitz [DaS87] describe a system called *E-Routing*, where the message always crosses the highest-order uncrossed dimension that needs to be crossed. E-routing may also be *low-order*, crossing the lowest-order dimension first. Dally and Seitz prove that E-routing is deadlock-free for the hypercube. The authors present a method for creating deadlock-free routing schemes using *Virtual Channels* [DaS87], and Dally goes on to show that virtual channels also can provide increased efficiency in parallel communication systems [Dal90].

It is important to distinguish here between *physical*, *virtual*, and *routable* paths. The network topology of any given architecture defines physical paths, which are made up of actual physical links between nodes. Some routing schemes will cast a given physical link as several virtual links for resource allocation and deadlock freedom purposes. Thus, several virtual paths can exist over a given physical path. Finally, routing schemes for multi-path architectures often will not utilize all physical paths from one node to another. Thus, a routable path is a physical or virtual path used by the routing scheme. For example, in the Hypercube architecture, there are two minimum-

length paths from node 11 to node 00, 11-10-00 and 11-01-00. High-order E-routing will only pass messages along the first path, however, so 11-01-00 is a physical path but not a routable path.

More interest has recently focused on dynamic systems that route messages based on current network load conditions and the presence of faulty nodes or links in the network [ChS89]. One important factor in the design of these types of systems is the method of gathering local or global information on load conditions and faults. Another important issue is the level of compatibility with concurrency control mechanisms and network performance enhancements such as combining, which require certain path restrictions.

An entirely separate routing issue concerns how messages are transferred from node to node. In *Store-and-Forward* networks, a message is stored completely at a node, and forwarded to the next node as a whole. This is the simplest scheme, and a significant volume of work has been produced on deadlock-free routing algorithms for these systems [Gun81][MeS80][Gel81]. These algorithms assign a partial order to resources by utilizing a structured buffer pool, ensuring that circular wait situations which cause deadlock cannot occur.

Wormhole systems [Dal87] break messages into small manageable packets. Routing information is contained in flow-control digits or flits. In wormhole routing, the head of a message (including routing flits) is advanced directly from incoming to outgoing channels at a node. As packets appear on the input channel, they are passed to the output channel. As this process occurs, messages can spread out across the network, with the requirement that the packets comprising a message always occupy contiguous channels. If the head of a message is blocked, the backed-up packets of that message then prohibit any channels they occupy from being used by other messages until the message header becomes unblocked and the message resumes transmission.

Because message flits cannot be interleaved in buffers, the priority buffer approach to deadlock freedom is not feasible for wormhole systems. However, Dally and Seitz [DaS87] show that assigning a partial order to channels rather than using buffers is sufficient for deadlock free-

dom in wormhole systems. This is the *virtual channels* approach, in which links between nodes are treated as multiple virtual links through context switching.

A third approach with similarities to both systems is *Virtual Cut-Through*[KeK79]. Here messages are broken up and allowed to spread throughout the network, but a node may not accept any part of an incoming message unless it can buffer the whole message (in the event that it is blocked). The deadlock properties of this type of system are identical to those of store-and-forward systems, although in low-traffic situations Virtual Cut-Through may offer greater throughput at less cost and network complexity.

Communication schemes are important to our work, because the implementation of local synchrony may alter network switching and routing algorithms. We show that changes in the communication systems of multiprocessors due to the implementation of local synchrony will not introduce the possibility of deadlock into the system, and that local synchrony will be compatible with the communication schemes (or variants) discussed above.

2.3 Combining

The objective of memory access combining is to reduce message traffic to and from heavily used global memory locations in a multiprocessor. One method of achieving this goal involves identifying accesses bound for the same memory location during transmission through an interconnection network, and molding these accesses into a single access, the result of which implies a given sequential ordering of the combined accesses. The NYU Ultracomputer was the first to combine accesses to global memory [GLR81][GLR83]; combining was later proposed for the IBM RP3 [Pfi85].

The Ultracomputer is an equidistant, shared-memory multiprocessor consisting of a number of processing elements, each with its own private memory, and a number of memory elements, which house global memory. Processing elements and memory elements are separated by an omega or banyan interconnection network, which provides a unique, same-length path from each

processing element to each memory location. An example of a small Ultracomputer is shown in figure 1.

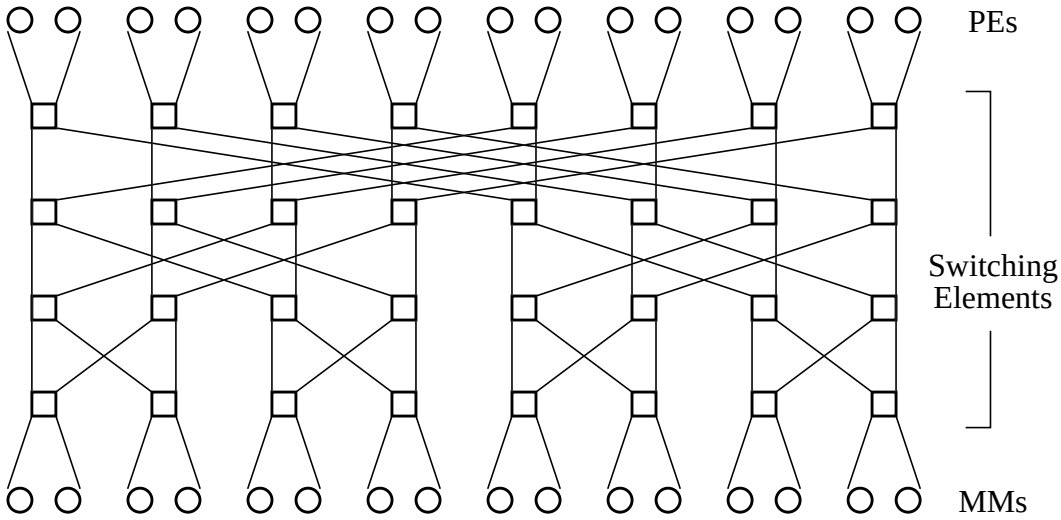


FIGURE 1. Ultracomputer.

As a simple example of combining, consider the following: Process 1 releases access WRITE(A,5) into the interconnection network. Process 2 releases the access READ(A). Assume at some point, at some switching element in the network, the two accesses are held concurrently. The switching hardware can then combine the two accesses into one access: WRITE(A,5). When this operation is completed, the result is as if both operations had been completed in sequence: WRITE(A,5); READ(A). The new WRITE operation is sent toward memory, while the combining information for the original messages is stored in the switch. When the combined operation returns successfully to the switch, the relevant combining information is extracted from the combining information queue through associative search. The READ access from Process 2 is sent back with the value 5, and the WRITE access from Process 1 is returned successfully.

In order to add power to the combining mechanism, Kruskal, Rudolph, and Snir [KRS88] have proposed the implementation of the READ-MODIFY-WRITE [Rud81] formalism in combining systems. Such a system would have as its basic memory access operation RMW(X,f), where X is the shared variable and f is a function to be performed on that variable. The RMW operation is equivalent to an indivisible execution of the function:

```
function RMW(X,f)
   begin
   temp := X;
   X := f(X);
   return(temp)
end
```

A READ operation may be represented in this formalism by performing the identity function, f(X)=X, while a WRITE operation is represented by performing the operation f(X)=c, where c is the constant to be written supplied by the process making the access.

The RMW formalism allows the implementation of much more powerful atomic memory operations. Using RMW operations as the basic, atomic form of memory access in a multiprocessing system gives the designer the freedom to implement almost any operation of the form:

memval := memval ***op*** *val*

where ***op*** is an operation which need not be associative. Possible implementations include the four arithmetic functions, all sixteen Boolean functions, and synchronization functions like full/empty bits. Kruskal, Rudolph, and Snir prove that their combining algorithm is correct by showing that the execution of a given combined operation is equivalent to a possible serial execution of the operations from which it is composed [KRS88].

A by-product of this work, which is important to our research, concerns the *orientation* of combined operations. When two operations are combined, the resulting combined operation, when performed, achieves the same result as some serial execution of the original operations. Consider a system which implements two basic operations: READ and WRITE. In this system, WRITE(B,3) and WRITE(B,4) might be combined into WRITE(B,4), which would, when executed, have the same effect as WRITE(B,3) followed immediately by WRITE(B,4). Thus, the orientation of the combined operation is the serial execution WRITE(B,3); WRITE(B,4). [Note: throughout this work, separation by semi-colons represents sequential orientation, while separation by commas denotes parallel (to the observer) orientation.]

A system implemented as above could combine two READ or two WRITE accesses in any orientation. In the case where a READ and a WRITE operation are to be combined, however,

a specific orientation is necessary in the combined operation. The combined operation must be a WRITE, as the variable must be modified, and, because WRITE operations are not generally designed to return the previous value of the variable, the READ operation must be oriented after the WRITE.

Now consider a system implementing RMW operations. RMW operations always return the previous value of the memory location, and can thus be combined in any orientation. In general, any pair of RMW operations that can be combined can be combined in either orientation [KRS88].

Little research has gone into memory access combining on non-equidistant architectures. The most important addition to the combining literature has been the work at Yale on the Fluent Architecture [Ran87] [RBJ88]. While the goal of this work had little to do with combining systems (its goal was efficient PRAM emulation), Ranade was able to show that, given some synchronization, combining operations in the network could be greatly simplified, removing the need for associative search when decombining. The use of synchronization in this case helps to ensure that messages leaving a given node will return to that node *in the same order.* The Fluent machine is a non-equidistant, shared-memory machine, but in the implementation of the combining system, the designers emulate an equidistant network in order to achieve their results. A message leaving a given node will travel the same distance before returning, regardless of the distance to its destination. This means that most messages will travel significantly farther than necessary in order to be serviced.

In [WiR91], Reynolds and Williams show that atomic actions can be combined in a recombining network, without compromising the integrity of the atomic actions. We use this result in designing combining switches for local synchrony, which allow the component accesses of isochrons to be combined if necessary.

2.4 An Approach to Concurrency Control

Perhaps the most important problem when working in an asynchronous multiprocessing environment is concurrency control. When multiple entities (i.e., processes) are competing for the use of multiple resources (i.e., global variables) extreme care must be taken to enforce proper accessing principles. The motivation for our own work in this area is parallel memory operations [Wag87] [RWW89] or *isochrons* [Wil93]. Isochrons allow a process to perform exclusive operations on sets of global variables without the need for locking or global synchronization. Local synchrony is a concurrency control mechanism which supports isochrons.

> **Definition:** The execution trace of a parallel program is equivalent to a serial schedule if all accesses to shared variables are executed in an order that is consistent with some serial execution of the same program.

In [Wil93], Williams characterizes the goal of concurrency control for parallel programs as 'to ensure that every execution is "serializable."' Williams' notion of serializability considers several important factors which extend beyond the definition of a serial schedule given above. The following discussion draws on definitions and results appearing in [Wil90].

A parallel execution and a serial execution are said to be *conflict equivalent* if, in both executions, each shared variable is accessed by the same operations and conflicting operations are executed in the same order. Operations conflict if their combination changes the variable, in other words if they are not both reads. Since read operations do not change the state of a variable, they cannot conflict. Williams formally describes a parallel execution to be serializable if there exists a conflict equivalent serial execution, E_S, with the following properties:

1. Atomicity - Accesses specified as atomic actions in the parallel execution are executed in E_S in sequence, without interleaving with other accesses.

An *Atomic Action* is a group of accesses, issued by the same process, that appear to be executed indivisibly, without interleaving with other accesses. An atomic action has the effect of modifying the state of a group of global variables as if the process submitting the atomic action has exclusive access to those variables while the atomic action is being executed.

2. Sequential Consistency - Any two accesses made by the same process are executed in the order specified by the process, if any.

The accesses of a given process are sequentially consistent if the order in which they are executed is consistent with the order specified by the process's sequential program [Lam79]. In other words, the memory access requests of a process must be executed in the order in which the process submitted them. Violations of sequential consistency can arise in multiprocessing systems because of network delays and the varying distances that access requests must traverse in order to be executed.

3. Version Consistency - Any two accesses to the same variable are executed in E_s in the order specified by the program, if any.

Version consistency means that the execution is consistent with any data dependencies specified by the program. These types of dependencies arise when the program specifies the order in which different accesses to the same data should occur. For example, a program might specify that a given variable, modified by a writer, must be read by a specified reader before it can be again modified by the writer. Version consistency is generally enforced by different types of locking mechanisms.

Williams' characterization of the goal of concurrency control for parallel programs is influenced by literature on concurrency control for concurrent databases, by Lamport's definition of sequential consistency [Lam79], and by Shasha and Snir's definition of correct execution of parallel programs [ShS88]. These and related works are discussed in more detail in the following sections. Our work with local synchrony is derived from this goal as stated, and from the need to provide sequentially consistent and atomic execution of parallel programs in asynchronous multiprocessor environments.

2.5 Isochrons

Our main goal is to explore implementation issues regarding local synchrony. Local synchrony meets the criteria for sequential consistency and atomicity as defined in [Wil93] and discussed in section 2.4. One benefit would be that local synchrony supports *isochrons* [Wil93], a concept derived from parallel operations [Wag87] [RWW89]. An isochron consists of a set of memory accesses emitted by a given process which appear to be executed simultaneously, so that

memory coherency is maintained. Preserving the coherency of memory means that it must not be possible for a process to observe an inconsistent state in global memory due to interleaving of other accesses with those of the given process.

Isochrons are a limited form of *atomic action*. The general form of atomic actions is a group of operations which are performed *atomically* [OwL82], meaning that no process other than the issuer can observe or affect an intermediate state in the execution. This general form of atomic actions allows data-dependent operations. For example, a process issuing an atomic action is able to perform an operation which reads the value of one global variable and writes that value to another, atomically. The issuer is thus able to observe the intermediate state of the execution of the atomic action. This situation allows the issuer to make conditional changes to the atomic action, based on intermediate results of that action. Isochrons, however, enforce a stronger requirement, which removes this capability. In a system implementing isochrons, no process, *including the issuer*, can observe or affect any intermediate state in the execution of an isochron. Thus, isochrons do not support global data-dependent operations.

Consider the readers and writers problem for a pair of shared variables, A and B. A and B contain values which may be read by a number of readers, and may be updated by a number of writers. A reader must read both variables, and a writer must write both. If process 1 emits two READ operations, READ(A) and READ(B), while process 2 emits WRITE(A,2) and WRITE(B,3), it is possible that the order of access to the shared variables would be READ(A); WRITE(A,2); WRITE(B,3); READ(B). In this case, process 1 does not receive a coherent set of responses, but instead receives half ‘old’ data and half ‘new’ data.

A common solution to this problem is to utilize a lock or locks to ensure exclusive access to the relevant data. In order to modify A and B, a process would have to acquire the locks on the objects before it would be able to modify the data. The locks would have to ensure the exclusion of both readers and other writers during the time the given writer was accessing the data. This serial-

izing of accesses to shared data inhibits concurrency significantly if several processes are accessing the same data regularly.

In general, the processes of a parallel program will need atomic access to different sets of shared variables at different points during their execution. In order to avoid deadlock due to processes circularly waiting for each other to give up locks, processes must be made to acquire locks in a specific order. This situation can cause processes to contend severely in acquiring high priority locks, which further slows access to the required data by requiring more global memory accesses. Alternatives to resource ordering in this situation admit the possibility of starvation.

Isochrons provide deadlock- and starvation-freedom without the use of access-intensive locks. The use of isochrons in the above example would allow only access orders in which the order of execution *at each memory location* is consistent with a serial schedule providing atomic access. To a monitoring process, the order of execution would appear to be one of the following: READ(A), READ(B); WRITE(A,2), WRITE(B,3) or WRITE(A,2), WRITE(B,3); READ(A), READ(B). Note, however, that the actual order of execution of the memory accesses could be READ(A); WRITE(A,2); READ(B); WRITE(B,2). While there are several different possible execution orders, from the viewpoint of any given process the set of operations in a given isochron would appear to have been performed *indivisibly,* or simultaneously.

The execution of a parallel program is said to be sequentially consistent if “the result of any execution is the same as if the operations were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program” [Lam79]. An isochron program, when executed, must be sequentially consistent and also ensure the atomicity of parallel accesses to global memory. This idea is similar to Reed’s notion of Concurrent Atomicity [Ree83]:

> For all executed steps o not in A (the atomic action), either o precedes all steps in A, or o follows all steps in A.

Isochrons are an attempt to apply this property to memory access in distributed systems. They have the potential to be an extremely powerful concurrency control mechanism: they could

reduce the need for much of the locking and synchronization of memory access in shared memory architectures, and thus significantly reduce global memory access traffic.

The implementation of isochrons requires a system which guarantees exclusive access to distributed memory locations. Our own requirements for such a system exclude explicit locking of locations and process synchronization in making this guarantee. Local synchrony [RWW89] [Wil93] meets these requirements though implicit logical timestamping of global memory accesses. Our research studies the implementation of local synchrony on various asynchronous architectures. The next section presents Williams' [Wil90] theoretical definitions for Locally Synchronous systems.

2.6 Local Synchrony

In [Wil90], Williams describes a set of clock conditions. These conditions describe the operation of a timestamping system for global memory accesses in a parallel architecture. A system operating under these conditions is shown to have a conflict-free serial execution which meets the requirements for sequential consistency and atomicity discussed in section 2.4. In [RWW89], a method of implementation is described which meets these clock conditions. This concurrency control mechanism is called *local synchrony.*

Williams distinguishes the following types of events:

> **Definition:** An **issue event** occurs when a process queues a global memory access to be sent for execution. Issue events occur in the sequentially consistent order specified by the issuing process.
>
> **Definition:** An **execute event** is when a global memory access actually occurs.
>
> **Definition:** An **emission event** occurs when an access issued by a process is actually sent. An access issued by a process may, for one reason or another, be held until a later time before being sent to memory. The actual sending of such an access would constitute an emission event.

By definition, the sequence of events for an access is always: issue, emission, execute. Because local synchrony is a *logical* timestamping system, two or more of these events might

occur at the same logical time. Each event is assumed to be atomic. In other words, each process issues operations one at a time, and each operation is executed at memory one at a time.

In [Wil90], two relations are defined on operations:

$OP_i \rightarrow OP_j$ (precedes) if
(1) each accesses the same variable and the execute event of OP_i occurs before that of OP_j. or
(2) each is issued by the same process and the issue event of OP_i occurs before that of OP_j. or
(3) OP_k exists such that $OP_i \rightarrow OP_k$ and $OP_k \rightarrow OP_j$.

OP_i *iso* OP_j if both OP_i and OP_j are from the same isochron.

Consider a global clock mechanism which assigns a 'time', $C(OP_i)$, to every execute event. Let $C(Op_i)$ consist of an ordered pair *(tick, tock)*, and let $C(OP_i) < C(Op_j)$ if $C(OP_i)$ lexicographically precedes $C(Op_j)$. The *tick* component of $C(OP_i)$ defines the timing relationship between OP_i and other execute events scheduled by other PEs. The *tock* component defines the relationship between OP_i and other execute events scheduled by the same PE. An execution E_p of a parallel program is then serializable if it conforms to the following clock conditions:

Condition 1: $OP_i \rightarrow OP_j \Longrightarrow C(OP_i) < C(OP_j)$

Condition 2: OP_i iso $OP_j \Longrightarrow C.tick(OP_i) = C.tick(OP_j)$

These conditions guarantee that no two execute operations are executed in E_p at the same logical time. Williams goes on to show that any isochron program, E_p, that conforms to these conditions is sequentially consistent and preserves atomicity of isochrons by proving that E_s is sequentially consistent, atomic, and conflict equivalent to E_p [Wil90]. Since all execute operations are executed at distinct logical times, those times define a *serial schedule*, or total ordering, of the global memory accesses of E_p.

While it provides a total logical ordering to memory accesses in a distributed system, local synchrony requires physical ordering of only accesses to the same memory location, or, more realistically, to the same memory element. One can see that inconsistencies in the state of global memory *can* exist with this implementation. Consider the readers and writers example in section 2.5, where Process 2 must perform indivisible writes on two global variables, A and B, located at mem-

ory elements M1 and M2, respectively, and Process 1 must perform indivisible reads. If process 2 submits the isochron:

P2: A:WRITE(2) || B:WRITE(3);

there is no guarantee that these accesses will be performed at the same *physical* time. For example, at physical time *i*, M1 might be performing accesses in logical time unit *t*, and perform the write of A. Meanwhile, M2 might be only executing accesses in logical time unit *t-2*. Due to heavy access traffic at its MM, M2 might not get to logical time unit *t* until physical time *i+j*. So there is a physical interval of length *j* during which another process, P3, might read both A and B and receive an inconsistent version of the state which A and B together represent. However, in order to do this, P3 must emit separate, non-isochronic accesses which will be performed during logical time units *t+1* and *t-1*, respectively. Atomicity is only guaranteed for isochronic accesses, or accesses performed at the same logical time unit. Sequential consistency, with respect to accesses to the same global variable, is guaranteed for all accesses.

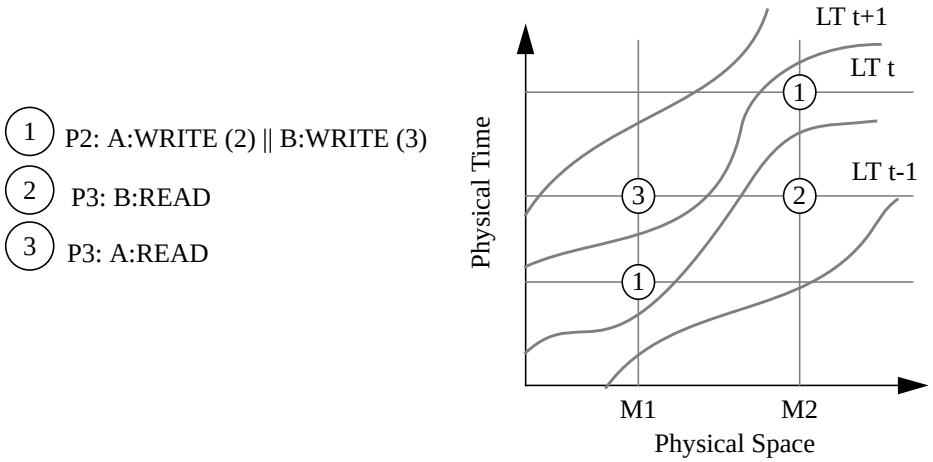


FIGURE 2.

Figure 2 gives an accurate representation of this scenario. One can see that, in order for a process within the system to ‘see’ an inconsistent state of memory, its accesses to that memory would have to be during different logical time units, and thus not isochronic. It is impossible for a process within the system to receive inconsistent data from an isochronic access to memory.

The importance of this limited form of synchronization should not be underestimated. Local synchrony provides a computation with total orderings of events and synchronization with-

out requiring globally synchronous processing. A global memory access which occurs *physically* before another might occur *logically* after, given that the two events represent accesses to different global memory locations. This flexibility allows a system to perform in a generally asynchronous manner, while gaining much of the power of synchronous operation.

The main focus of our work pertains to local synchrony as defined in this section. Our research concerns the implementation of local synchrony in various multiprocessing environments. The definition of local synchrony outlined in this section, and specifically the clock conditions of local synchrony defined by Williams, provide a basis for any implementation of the system.

2.7 Schemes Related to Isochrons and Local Synchrony

In database concurrency control, timestamp ordering protocols guarantee concurrency control for any set of accesses without specific locking of the relevant data. Each access is assigned a specific ‘execution time’, and accesses to a given global variable are constrained to be executed in timestamp order. To be executed, a READ access must have a timestamp greater than the last WRITE, and a WRITE access must have a timestamp greater than any preceding access [BeG81]. In the basic timestamp ordering system, this means that an access will be aborted if it does not meet the above requirements.

Conservative timestamp ordering systems eliminate rollbacks by executing accesses only if no older access will ever be received [BeG80] [Mil79]. This type of system is generally based on global knowledge about accesses currently in the system and the current timestamps of processes. Timestamp ordering systems generally enforce a policy of sequential consistency, which dictates that processes issue timestamps in ascending order.

Local synchrony may be cast as a conservative timestamp ordering system. Local synchrony is constrained to produce sequentially consistent serial schedules for global memory accesses. The atomicity of isochrons is guaranteed with no possibility of rollback. Bernstein and Goodman [BeG81] state that, in a conservative system, “When a scheduler [node] receives an

operation O that might cause a future restart [logical time conflict], the scheduler [node] delays O until it is sure that no future restarts [logical time conflicts] are possible. Either of the two general implementation techniques (sorted transmission, sorting at memory) fits this description.

There are several differences between databases and applications programming which affect the problem solver in this area [Wil93]. First, database systems generally require full recoverability of data in the case of system failure, whereas a parallel program can just be rerun. A database manager is therefore generally able to incur greater overhead in ensuring that transactions are performed correctly, whereas a concurrency control concept for a parallel operating system must incur a minimum of overhead in order to preserve memory cycle time. Concurrency control in databases generally requires only atomicity of transactions, because the nature of the database environment guarantees sequential consistency. Concurrency in parallel programs also requires sequential consistency, but it must be explicitly enforced by a system such as local synchrony.

Atomic Actions are a concept designed to provide exclusive access to multiple data items. An atomic action is a set of accesses released by a process which, when executed, appear to have occurred simultaneously, allowing no interleaving of accesses by other processes during their execution. Atomicity may be ensured by an underlying system based on two-phase locking or timestamp ordering. Atomic actions were developed in the database literature by Eswaran et al [EGL76]. Reed [Ree83] suggests a timestamp ordering implementation of atomic actions based on timestamping by physical clocks and the use of version histories in memory. A physical clock is based at each processing element, and the clocks are kept loosely synchronized by a global clock synchronization algorithm. Again, while this implementation does allow the implementation of atomic actions, the problem of aborted operations still exists.

Concurrency control as applied to parallel architectures and operating systems has only recently come under study. In [Lam78], Lamport proposed a system of *logical clocks*, which would provide partial ordering to the events of a system of processes. These clocks would consist of a function that assigns a number to each event occurring at a given process, in increasing order,

where the number would be considered the *time* that the event occurred, although this number would have no relation to actual physical time. When a process sends a message to another process, Lamport requires only that the receive time of the message be greater than the send time, since one would generally assume that a message must be sent before it is received. Lamport's clock conditions are:

> Condition 1: If a and b are events in process Pi, and a $\Longrightarrow$ b, then Ci(a) < Ci(b).
>
> Condition 2: If a is the sending of a message by process Pi, and b is the receipt of that message by process Pj, then Ci(a) < Cj(b).

Since ordering by physical occurrence of events at different processes in a distributed system is not generally possible, Lamport first generates partial orderings for events occurring in individual processes, and to inter-process communication, using the logical clock, then provides a total ordering of events by arbitrary ordering of the processes themselves. Note that this total ordering represents a sequentially consistent (by the clock conditions) serial schedule, meaning that this ordering of events is equivalent to a possible ordering of events if the computation were simulated on a traditional Von Neuman machine. Lamport uses this ordering to solve synchronization problems.

Jefferson [Jef83][Jef85] has proposed a more general theory of logical time in *Virtual Time Systems*. Such a system consists of a group of processes that execute in coordination with logical clocks, and which obey certain conditions. These conditions echo Lamport's clock conditions, except that they also restrict the processing of events received by a particular process:

> Condition 1: The virtual send time [$C_i(a)$] of a message must be less than or equal to its virtual receive time [$C_j(a)$].
>
> Condition 2: All messages directed to a particular process must be processed in non-decreasing virtual receive time order.

The semantics of such systems obey the requirement that, "If an event A has a virtual time less than that of event B, then the execution of A and B must be scheduled so that A appears to be completed before B starts" [Jef83]. Note that there is no requirement for actual physical ordering

of events in such a system. The requirement is only that, to the processes in the system, the events appear to have occurred in the given order.

The goal of Virtual Time systems is to create a temporal coordinate system in which a distributed computation can be embedded that will provide correct, logical orderings of the events of that computation. Again, the correctness of the logical orderings corresponds to their providing a proper serial schedule. Jefferson's protocol is optimistic, utilizing state-saving and roll back in order to meet its clock conditions. Cascading rollbacks are a significant problem with the Time Warp approach, although Jefferson has shown that there is a limit beyond which rollback cannot occur in a given computation: namely, the earliest current virtual time of any event in the system.

Local synchrony is a conservative timestamp ordering protocol, which guarantees serial scheduling and sequential consistency, as well as the atomicity of isochrons, without the possibility of rollback.

Awerbuch [Awe85] utilizes a system similar to local synchrony in order to simulate synchronous architectures on asynchronous networks. Awerbuch uses the notion of *safety*—where a node knows that it is finished with its operation for a given time unit—to generate a barrier for the whole network. This barrier is passed when a simulated global clock recognizes that it has been reached (all nodes are safe) and generates the next synchronous step. Safety is recognized by a node when it receives confirmation from each of its neighbors that they will send it no more work prior to the next barrier. Local synchrony elements use safety in order to safely proceed to the next logical time unit.

Ranade [Ran87] uses *end-of-stream* (EOS) messages, similar to local synchrony tokens, to emulate a CRCW PRAM on a butterfly. Unlike local synchrony, EOS messages are sent after *each* access. The EOS messages act to synchronize the operation of the system. The passage of EOS messages from an input to an output line represents timesteps of the 'global' clock as represented at each switch. Combining is implemented in such a way that only one access is executed at any memory location during any timestep. A main difference between Ranade's work and local syn-

chrony is that Ranade requires that all messages travel the same distance to reach memory. This distance is larger than the minimum necessary traveling distance. Local synchrony allows messages to travel the minimum distance defined by the architecture to reach memory. The Fluent machine is also a specifically targeted implementation. Our implementation of local synchrony is a general approach, which can be implemented on any connected architecture (as defined in chapter 4). Ranade also failed to note that atomicity could be guaranteed using such systems.

Ranade also, in order to guarantee bounds on the emulation, is able to implement efficient (no associative search) combining by keeping accesses sorted as they traverse the network. Efficient (FIFO) combining is discussed in section 2.2.

In [ChM79], Chandy and Misra present a scheme for distributed simulation. The Chandy/Misra system is a conservative timestamp ordering system much like local synchrony. The authors' implementation scheme is less efficient than ours, however, and their proof of deadlock freedom is slightly flawed. Kumar [Kum86] proves the Chandy/Misra system to be deadlock-free for feedforward networks consisting of fork and join nodes. In [ChM87], Chandy and Misra develop the idea of conditional knowledge, which increases the efficiency of their distributed simulation scheme.

This section provides us with a background with which to present an *Operative Condition* of local synchrony. This condition will constrain our research toward efficient and fault-tolerant implementations of local synchrony by focusing our efforts on implementations which specifically meet this condition. The operative condition of local synchrony stems from the clock conditions discussed in the previous section, and is stated in this way:

> A local synchrony implementation is a conservative (logical timestamp) ordering system for global memory accesses. Thus, an entity (PE, switch, or MM) operating in a Locally Synchronous manner must perform actions on a given resource (channel, memory location) in a manner which preserves a logically serializable schedule of global memory access.

An entity operating in a conservative manner is constrained not to perform any action until it has, with certainty, performed all (logically) previous actions. Local synchrony requires only

that the MMs operate conservatively in processing global memory accesses. However, an implementation of local synchrony may include conservative operation by switches and/or PEs as well.

2.8 An Implementation of Local Synchrony

In [Wil93], Williams presents basic implementation suggestions for local synchrony on an equidistant network and discusses extensions to this technique for non-equidistant architectures. Williams does not present specific local synchrony implementations or prove deadlock freedom. We present here the general implementation technique. For simplicity, we make several assumptions, which are based on those of Williams in [Wil93]:

1. The system is a network of nodes communicating over FIFO channels with finite but unbounded delay.
2. Each node consists of a switch, a processing element (PE) and a switch, a memory module (MM) and a switch, or a PE, an MM, and a switch.
3. Each process runs on its own PE. There is no multiprogramming or process migration.

Local synchrony may be characterized by the serializability, or constructible serial schedule, of all accesses performed. This serial schedule is represented by timestamp ordering of global memory accesses. Each node stays consistent with this serial schedule by loose synchronization with neighboring nodes.

This loose synchronization among nodes is implemented in the following way: each node in the system periodically emits a set of control signals, or *tokens,* one along each of its output connections to other nodes. These tokens represent the boundaries between the logical timesteps of the node's own local logical clock. For the purposes of further discussion, we will define ltu_i at a given PE to be that period of time between the sending of token *i-1* and token *i*. During an ltu, a PE may emit zero or more accesses (accesses may only originate in nodes containing a PE).

In order to implement *local* concurrency control, the switch associated with each node must route all accesses received from its neighbors during its previous ltu before generating its next ltu. Switches may route accesses without regard to their timestamp order within the current ltu, in which case accesses must be buffered and sorted at the MM, or by merging sorted streams of

accesses by timestamp. We shall discuss these and other schemes for the transport of accesses in section 6.4. For simplicity, the reader may wish to assume in the following discussion that accesses are kept sorted by timestamp throughout their traversal of the network. Given the operative condition of local synchrony, MMs *must* process accesses (perform execute events) in a conservative manner.

Extended from [Wil93], a PE schedules emission events subject to two constraints:

1. All the accesses in any given isochron are emitted in such a way that the execute event of each will occur during the same ltu.

Constraint 1 is a restatement of the second clock condition, which guarantees atomicity for isochronic accesses. In a general implementation, isochronic accesses may have different distances to traverse in order to be executed. In this case, Constraint 1 requires the PE to guarantee that accesses arrive at memory during the same ltu. Constraint 1 is relatively simple to meet in an equidistant implementation. A PE must emit all the accesses in a given isochron during the same ltu. In a non-equidistant implementation, however, guaranteeing Constraint 1 is somewhat more difficult. Ways to guarantee Constraint 1 include holding accesses with shorter distances to travel for later emission, or routing messages along longer than necessary paths.

Let *T(I)* be the beginning logical time unit in which isochron will be emitted into the network. Let *D(I)* be the maximum distance any one access of *I* will have to travel to memory. Finally, let OP_i be a member access of isochron I. $d(OP_i)$ is the distance that OP_i must travel in order to reach memory. OP_i will then be emitted into the network at time:

$$t(OP_i) = T(I) + D(I) - d(OPi).$$

This is one possible implementation of Constraint 1; it guarantees that all the member operations of a given isochron will arrive for servicing during the same ltu. The main requirement for this type of operation is that a given processing element must be able to ascertain the distances that its accesses must travel in order to be serviced.

2. Any pair of accesses from a given process that access the same variable must be emitted in such a way that the execution events take place in the order specified by the process.

Constraint 2 provides sequential consistency for the accesses of a given PE. Sequential consistency is guaranteed in the execution of accesses by ensuring that they are executed *in issue order*. This ordering may be guaranteed through different combinations of routing, buffering, and sorting schemes for accesses. For example, rather than emit accesses in issue order, a PE might route an access in such a way that it is executed after an access emitted later but routed along a shorter path. This works because the logical time to traverse a path to memory is a function of distance, or the number of switching elements an access must go through to reach its destination.

The logical time of execution of each access is determined by the issuing PE through explicit timestamping, implicit timestamping (determined by ordering conventions implemented in the system), or a combination of the two. The timestamp for access OP_i, denoted $TS(OP_i)$ [Wil93], is an ordered pair *(tick, tock)*, where the *tick* component is itself an ordered pair *(ltu, pid)*. The ltu component of $TS(OP_i)$ is the ltu in which the execute event of OP_i will occur. In order to assign *ltus*, the PE is required to be able to compute the number of local synchrony ltus that will pass while a given access travels to its destination. This number of ltus is equal to the distance which the access must travel.

Note that, due to Constraint 1 above, an access issued by a process may be held by the PE for later emission. In order to preserve the definition of the *precedes* relation ($\rightarrow$) given in section 2.6, the PE must ensure that *ltu* components are assigned in non-decreasing order as accesses are *issued*, rather than emitted. This topic will be discussed further in chapters 3 and 4.

The *pid* component is an ID unique to the issuing PE. The use of this *pid* in the timestamp assigns a disjoint interval of logical time during each ltu to the accesses issued by the given PE. This disjoint interval provides atomicity for isochrons. This technique for providing atomicity is widely used in database concurrency control [Ree83] [RSL78].

The *tock* component of the timestamp provides sequentially consistent ordering for any operations accessing the same variable, from the same process, in the same ltu, i.e. $TS(OP_i).tock =$

j if OP_i is the j^{th} access from the PE identified by $TS(OP_i).tick.pid$ to be executed during ltu $TS(OP_i).tick.ltu$.

Williams makes the following propositions regarding the operation of the system as described. These propositions relate equally well to the technique presented here:

Proposition 1: For any pair of accesses, OP_i and OP_j,
OP_i iso $OP_j \Rightarrow TS(OP_i).tick = TS(OP_j).tick$.

Proposition one is true by PE Constraint 1, which guarantees that the *ltu* component of the timestamps of each operation in an isochron will be the same, and by the fact that an isochron is defined to be a group of operations emitted by one process, and thus all having the same *pid*.

Proposition 2: For any pair of accesses, OP_i and OP_j, issued by the same process, if OP_i is issued before OP_j, $TS(OP_i) < TS(OP_j)$.

Proposition 2 is true because the assignment of timestamps is constrained to be non-decreasing by issue order. If OP_i is issued before OP_j, then either $OP_i.ltu < OP_j.ltu$, or $OP_i.tock < OP_j.tock$, and thus $TS(OP_i) < TS(OP_j)$.

Proposition 3: For any pair of operations, OP_i and OP_j, that access the same variable, if the execute event OP_i occurs before OP_j, then $TS(OP_i) < TS(OP_j)$.

Proposition 3 stems from the requirement that operations either be transmitted through the network in timestamp order, or be sorted by timestamp order at memory before their execute events occur. This, of course, guarantees that accesses will be executed in timestamp order.

Proposition 4: For any operation OP_i, $C(OP_i) = TS(OP_i)$.

Williams shows that the algorithm specified by the above propositions and the conditions on operation of the PEs and switches correctly implements the clock conditions for equidistant systems[Wil90]. Clock condition 2 follows directly from propositions 1 and 4. Clock condition 1 is implied by propositions 2, 3, and 4. For any pair of operations, OP_i and OP_j, $OP_i \rightarrow OP_j \Rightarrow C(OP_i) < C(OP_j)$. There is a case for each alternative in the definition of the $\rightarrow$ relation:

Case 1: If OP_i and OP_j access the same variable and OP_i is executed before OP_j then $C(OP_i) < C(OP_j)$ by propositions 3 and 4, together with the fact that each MM executes operations in the order in which they arrive at memory.

Case 2: If OP_i and OPj are issued by the same process and OP_i is issued before OP_j then $C(OP_i) < C(OP_j)$ by propositions 2 and 4.

Case 3: Otherwise, if $OP_i \rightarrow OP_j$, there is a sequence of operations $OP_i \rightarrow ... OP_k ... \rightarrow OP_j$. By induction on the length of the sequence, $C(OP_i) < C(OP_k)$ and $C(OP_k) < C(OP_j)$. Thus $C(OP_i) < C(OP_j)$.

This argument may be applied directly to non-equidistant systems, given that the restrictions on the assigning of *ltu* and *tock* components of the timestamp discussed above are in force.

It is important to note that explicit timestamps as described above are not necessary to the implementation of local synchrony. The *ltu* component of the timestamp can be implicit in the general implementation technique. The *tock* component is implicit if one assumes that accesses are transported along FIFO channels, and that any sorting of messages by timestamp, if performed, is stable. In a non-equidistant system such as the hypercube, only the *pid* need be explicitly represented in each access, in order to ensure correct ordering of accesses from different PEs within logical time units.

The notion of *convexity* was introduced in [RWW89]. For a wide class of networks, including the Ultracomputer's Omega or Banyan network, there exists a convex labeling; i.e., an assignment of *pids* to PEs, such that the source of all input channels to a given switch is a set of PEs with contiguous *pids*, and, for any two inputs, A and B, the *pid*s of all sources covered by input A must be uniformly greater or less than the *pids* of all sources covered by input B. For example, consider a switching element having two input channels, A and B. If the inputs on channel A may come only from the nodes with *pids* 0, 1, 2, and 3, and the inputs on channel B may come only from the nodes with *pids* 4, 5, 6, and 7, then the inputs to the switching element are convex. Let us assume that the lower *pids* have higher priority. Note that, if the element is merging sorted streams of accesses coming in along channels A and B by timestamp, it will always choose channel A over channel B during the same ltu. When implementing local synchrony in such systems, even the *pid* component of the timestamp need not be explicit, as the switching decisions concerning which access to route first are statically based on the input channels themselves.

The implementation of local synchrony in convex systems is highly compatible with combining. In order to facilitate combining of accesses PEs can add a fourth component to the implicit

timestamp (although this component is for combining purposes only and *does not* affect the logical ordering of events). Accesses are then sorted and issued in timestamp order, by *ltu*, *combining component*, *pid*, and finally rank (of accesses from the same PE to the same memory location). The *combining component* of the timestamp is generally implemented as the address of the memory location to be accessed, although several options are possible.

Switches in this implementation perform the sorted merge, which brings accesses bound for the same variable together for combining. Accesses are combined in the orientation (serial ordering), which preserves the *pid* ordering that is essential to correct ordering of execution events. All four correctness propositions hold for this alternative implementation. The level of compatibility of combining and local synchrony for systems not easily shown to be convex will be an important topic of our research, and will be discussed in the next chapter.

This section has presented a general implementation technique for local synchrony which is shown to meet the clock conditions defined in section 2.6. While this technique provides groundwork for our research into local synchrony implementations, there are several important areas which are not addressed here. Of major importance is the question of deadlock. Deadlock situations are most complex and prevalent in direct, non-equidistant architectures such as the hypercube.

A simple solution to the problem of deadlock can be found in the work on the topological equivalence of different types of networks and architectures. In [Wu80], Wu and Feng demonstrate the equivalence of data manipulators, flip networks, omega networks, regular SW banyan networks, and indirect binary n-cubes. Agrawal [Agr83] goes on to show that all such $\log_2 N$ stage interconnection networks are topologically equivalent. Padmanabhan [Pad90] presents the exact structural relationship between binary n-cubes and the hypercube, bridging the gap between equidistant and non-equidistant architectures. Saad and Schultz [SaS88] also show that various topologies, most importantly linear arrays (meshes), may be mapped into the n-cube topology.

A deadlock-free implementation of local synchrony on an Ultracomputer, then, could be translated to the hypercube through topological equivalence, which allows the hypercube to emulate the Ultracomputer architecture. Of course, the efficiency of any such implementation would be questionable. The main advantage of a direct, non-equidistant architecture such as the hypercube is that it is not equidistant; i.e., that groups of processes may be clustered on closely connected sets of nodes. To emulate equidistance on such an architecture would negate this advantage.

The implementation technique presented in this section also leaves open the questions of efficiency and fault tolerance of a given implementation. This section presents general guidelines for a local synchrony implementation. Our work proves this implementation to be deadlock-free and suggests a specific implementation plan.We go on to present a correct and deadlock-free general implementation technique which is applicable to any useful parallel architecture.

2.9 Fault Tolerance

It is important that a complex system degrade gracefully in the presence of hardware faults of one kind or another. Simple error detection schemes and error correction codes protect most systems from transient errors, which affect the system for short periods of time. When an entity fails entirely, for a finite period or indefinitely, the computation may be completely compromised by loss of information. The types of faults important to our work are connection faults, where the connection between two nodes fails, and node faults, where a node ceases to process correctly or fails entirely, with or without a loss of information. The tolerance of these types of faults has two facets: how to detect faults, and how to tolerate them. The detection of faults in a multiprocessing system generally occurs in the low level operating system, or in the hardware itself. Our work will focus on ways that a locally synchronous system can tolerate system faults once they have been detected.

A main thrust of fault-tolerance research is *redundancy*: tolerating faults by providing extra nodes or connections which may be put into use in the event of a fault. Banyan type interconnection networks for uni-path, equidistant architectures such as the NYU Ultracomputer can be

given a measure of connection fault tolerance by connecting switching units in the same stage [TYZ85] [TYZ86]. Adding an extra stage to the network makes it a multi-path design, which can be used to tolerate both node and connection faults by replicating messages along all possible paths [BaD89], or by utilizing non-faulty paths after faults have been detected [PaL83]. Redundancy schemes have also been proposed for the Butterfly [BaB87] and the Hypercube [IIS82], where an extra dimension is added to the cube for fault-tolerance and load-balancing measures.

Once faults are detected in a multi-path system, be it a redundancy enhanced NYU Ultracomputer or a normal multi-path Hypercube, most research on tolerating those faults has been in the area of dynamic routing systems which can detour messages around faulty nodes or connections in their normal path. An important condition that must be maintained is *Dynamic Full Access* (DFA) [VaR86], which means that a message from any given process within the system must be able to reach any destination available to it. If DFA does not hold, then at least part of the system has been cut off from the rest by faults, and either the computation must be reconfigured to run on one part of the fractured system, or it must be halted.

The designer of a dynamic routing system must take into account the degree to which his design depends on global knowledge, the gathering of which can seriously impact processing efficiency. Fault-tolerant routing in a multi-path system such as the Hypercube can be implemented as a simple depth-first search [ChS89], where a node can initiate a forward search of the system in order to efficiently route messages around faults. Chen proposes either simple fault avoidance, where a node reroutes a message only when it has direct knowledge of a fault in the path, or single-layer communication, where a node receives information about faults local to its neighbors as well.

In a local synchrony implementation, the problem of fault tolerance becomes more complex, because local synchrony makes explicit presumptions concerning the length of the path along which a given access will travel in order to be executed. Combining also restricts paths to the extent that combined accesses must be decombined during their return from memory. A multiply combined access must be decombined correctly; i.e., in the reverse of the order in which it was

combined. These issues must be addressed to provide a fault tolerant implementation of local synchrony and combining.

Another facet of fault tolerance research that we address is rollback recovery. A compromised computation must be rolled back to a previously saved coherent state. Randell [Ran75] and others have pointed out that the *domino effect*, in which a rollback causes further rollbacks in a chain, can be eliminated if state savings can be synchronized. Others [TKT92 et al.] have proposed different methods for achieving this type of synchronization.

Local synchrony provides built-in logical synchronization which facilitates coherent state saving and rollback. We present a rollback algorithm for the general implementation of local synchrony in section 5.3. Our work draws on the logical time aspects of local synchrony. We use the local synchrony ltu as a *soft barrier* [BGSS89] in order to save state and perform rollback. There is no need for physical synchronization to achieve coherent rollback without the possibility of cascading rollbacks in our implementation.

2.10 Performance Issues

The analytical study of the performance of interconnection networks and multiprocessing architectures is an important and difficult problem that has attracted much research in the last decade. Most of this research has dealt with the study of interconnection networks for indirect systems: delta, banyan, and omega networks.

Research in this area, in order to make analysis tractable, generally makes several simplifying assumptions about the system and traffic. The most important general assumption relating to our work is that networks operate *synchronously.* Synchronous operation is assumed in all known performance studies of shuffle-exchange networks. Many of these studies also assume that messages in the system may not block; if a message is blocked it is lost from the system. In the case of local synchrony, this assumption would be invalid, as accesses must be executed once they have been emitted. Thus, our discussion will consider studies in which messages may not be removed from the system, but must be blocked.

There are three general approaches to analytical modeling of these types of systems. Researchers using Markov analysis have had some success [LiK91] [WuL92] [Mer91], but the combinatorial explosion of states involved in this technique makes complex models difficult. Queueing networks have also been suggested [WiE90]. Both of these techniques have significant drawbacks when applied to locally synchronous systems. In general, neither field can handle the multiplicity of message types or the infinite priority scheme which is inherent in a timestamping system of this type.

The third approach is probabilistic mean value analysis. This type of analysis has been applied with success to banyan networks and their equivalents [Jen83] [YLL90] [KiL90] and to direct binary n-cubes [AbP89].

In [Jen83], Jenq presents an analytical model for the performance of Banyan (Omega) networks composed of 2-input, 2-output switching elements in a synchronous, single buffered, packet-switching system. This model forms the basis for later work by Yoon et. al. [YLL90], which extends the model both to n-input, n-output switching elements and to multi-buffering of packets, and others. In [KiL90], the authors apply the probabilistic technique to non-uniform traffic patterns. Jenq's model is outlined here, and in section 6.2 we present an analytical model for locally synchronous systems based on and extending Jenq's probabilistic method.

Jenq assumes a synchronous network, where a clock cycle, t, consists of two phases, t_1 and t_2. During t_1, information relating to the state of input buffers passes from the last stage backward to the first stage, so individual switches may make decisions as to which packets may be forwarded. Packet switching then occurs during t_2. Jenq also makes the following simplifying assumptions:

1. The packet load on the inputs to the network, that is, the input buffers to those switches in stage 1, is uniform.
2. The packets arriving at each input link are destined randomly.

With these simplifying assumptions, Jenq points out that the state of a given switching element is statistically identical to that of any element of the same stage. Jenq further assumes that the

states of the input buffers of a given switching element are statistically independent, although he points out that, because of packet collisions, this is not generally the case. This assumption is made in order to simplify the model, and Jenq later presents a more complicated model which takes packet collisions into account, and shows that the simple model provides good analysis at a significant reduction in complexity.

Jenq introduces the following notation:

n = The number of stages in the switching network.

$p_0(k, t)$ = Probability that an buffer of a switching element at stage k is empty at the beginning of the t^{th} clock period.

$p_1(k, t)$ = Probability that an buffer of a switching element at stage k is full at the beginning of the t^{th} clock period.

$q(k, t)$ = The load on an input buffer at stage k during the t^{th} clock period. In other words, the probability that a packet is ready to be transmitted to the buffer during the t^{th} clock period.

$r(k, t)$ = Probability that a packet in a buffer at stage k is able to move forward during the t^{th} clock period, given that there is a packet in that buffer.

Jenq then goes on to present the following equations which govern the relationships among these variables:

$$q(k,t) = 0.75p_1(k-1,t)p_1(k-1,t) + 0.5p_0(k-1,t)p1(k-1,t) + 0.5p_1(k-1,t)p_0(k-1,t) \quad \text{for } k = 2,3,4,...,n \qquad \text{(EQ 1)}$$

$$r(k,t) = [p_0(k,t) + 0.75p_1(k,t)]\,[p0(k+1,t) + p_1(k+1,t)r(k+1,t)] \quad \text{for } k = 1,2,...,n-1 \qquad \text{(EQ 2)}$$

$$r(n,t) = p_0(n,t) + 0.75p_1(n,t) \qquad \text{(EQ 3)}$$

$$p_0(k,t+1) = [1-q(k,t)][p_0(k,t) + p_1(k,t)r(k,t)] \qquad \text{(EQ 4)}$$

$$p_1(k,t+1) = 1 - p_0(k,t+1) \qquad \text{(EQ 5)}$$

These equations define a state transition for the model, and the quantities $q(k)$, $r(k)$, $p_0(k)$, and $p_1(k)$ converge to time independent quantities for any given *n* and $q(1)$. The important performance measures, normalized throughput, *S*, and normalized mean delay, *d*, may then be quantified by:

$$S = p_1(k)r(k) \text{ for any } k \qquad \text{(EQ 6)}$$

$$d = (1/n)\sum(1/r(k)) \text{ for } k = 1 \text{ to } n \qquad \text{(EQ 7)}$$

Jenq presents data derived from the model for various *n* and $q(1)$.

Several other performance studies also consider throughput and delay for different types of networks. Dias and Jump [DiJ81b] [DiJ81a] consider the performance of both buffered and unbuffered delta networks, compared to crossbar-type switches. This analysis is extended in [KuJ86]. Kruskal and Snir [KrS83] present asymptotic analysis of unbuffered and buffered Banyan networks, compared to equivalent networks.

Analytical studies have also considered the bandwidth, or number of requests accepted (by the network) per cycle, of several types of both unbuffered and single-buffered shuffle-exchange networks [Pat81] [YLL87]. Other performance analyses have been conducted for fault-tolerant versions of these types of networks [KuR85] [YoL89].

Finally, performance studies have been presented for combining systems [Lee86] [Won86]. These studies report performance measures for combining systems in normal operation, where a random distribution of requests is assumed, and in cases of *hot spots* [Pfi85], situations in which more requests are targeted for certain outputs.

The studies discussed so far have considered only indirect and mostly equidistant network configurations. [AbP89] and [Abr90] take up the analytical study of the performance of directly connected architectures related to the hypercube. Abraham provides mathematical models and discusses performance characteristics of direct binary d-cube networks. The results of these analyses compare favorably to those for indirect shuffle-exchange networks.

This section has reviewed work in performance analysis, specifically the probabilistic method of Jenq. In Chapter 6, we present analytical performance models of local synchrony implementations based on and extending the probabilistic approach.

3.0 Preliminaries

This chapter provides a motivation for our later work on the implementation of local synchrony. First we review some of the terminology used throughput this work.

A shared memory architecture consists of *element*s: PEs, MMs, and switches. Several elements may be grouped into a single physical *node*.

An *access* is a request sent by a process to shared memory. An MM will return a *response* to each access. When distinguishing accesses and responses is unnecessary, we may use the term *message*.

In local synchrony, accesses are assigned unique *logical timestamps*. Each timestamp includes information about the logical time unit (*ltu*) of the access, the source (*pid*) of the access, and the *rank* (among accesses from the same source) of the access. The timestamp may also include information about the destination (*dest*) of the access.

A *ghost message* [Ran87] is a message which contains only logical timestamp information. A ghost message serves only to increase network efficiency and facilitate deadlock freedom.

The local synchrony *ordering constraint* forces PEs, switches, and MMs to process accesses in order by their logical timestamp.

The *logical time* of an entity within a local synchrony implementation is the logical timestamp of the most recent message processed by the entity.

A message traverses a *step* in the network when it moves from one element to a neighboring element. A message must travel a *distance* of some number of steps to reach its destination.

In local synchrony implementations, *tokens* are passed to mark the barriers between ltus.

Isotach networks [ReW91] are special local synchrony networks which enforce the *isotach velocity invariant*, that a message requires one ltu to proceed one step in the network.

An architecture exhibits *locality* if accesses to shared or distributed memory from a PE in the architecture can have varying distances to travel. For example, the Ultracomputer exhibits no locality (all accesses travel the same distance), while a hypercube exhibits moderate locality (there are log n different distances an access might have to travel). We assume that shared memory can be emulated on distributed memory architectures.

An architecture exhibits *convexity* if the source of inputs to a given element is a set of PEs with contiguous pids. Convexity was defined in more detail in chapter 2.

The *raw power* of a network is a measure of the network's throughput and delay under loads which do not include any concurrency, atomicity, or sequential consistency constraints. In all analytical studies and in most simulations, the raw power is measured for a full input load (i.e., the network inputs are always full).

In section 3.1 we discuss various methods of concurrency control by contrasting locking methods (two-phase locking [EGL76]) with access-ordering methods. Using simple arguments, we show that local synchrony can outperform traditional locking systems if the local synchrony implementation has greater than a certain fraction of the raw power of a similar locking implementation. In section 3.2 we detail a banyan-network implementation of local synchrony that includes a proof of deadlock freedom, based on the use of ghost messages. In section 3.3 we present a low-level switch design for the implementation discussed in section 3.2. Finally, in section 3.4 we outline the minimum requirements for deadlock freedom in a local synchrony implementation. We prove that the *inherent knowledge* possessed by a given switch is sufficient for deadlock freedom in a convex network, without the use of ghost messages.

3.1 An Alternative to Locking

Our motivation is to investigate alternatives to locking as a method of enforcing atomicity in multiple-access shared memory operations. Local synchrony is one of several solutions that guarantee sequential consistency and atomicity for all accesses at the hardware level, instead of higher-level software approaches such as locking.

Before an atomic access can be executed in a simple locking system, at least one access is needed to acquire the lock for each memory location. Even if these lock accesses can be pipelined (admitting the possibility of thrashing and livelock), at the very least the delay incurred by acquiring locks doubles the time required to execute the atomic action. Truly livelock- and deadlock-free locking systems must serialize their locking accesses, which considerably lengthens atomic access times. This is true even without considering the possibility of contention for locks, which may block the progress of lock acquirement for extended periods.

We have investigated alternative systems such as local synchrony in the hope of demonstrating that implementing sequential consistency and atomicity constraints for all accesses at a lower (hardware) level can lead to better performance in comparison with locking systems. We call these systems *access ordering systems*, because their fundamental purpose is to ensure that

accesses are executed in a specific order. Although these systems may lengthen the latency of a single access, an atomic action can be requested without generating the overhead of locking accesses, and any atomic action, once requested, is guaranteed to execute without further involvement of the requesting PE.

The latency cost of execution for an atomic action consists of four elements: C_t, the cost of transporting the action to and from memory; C_e, the cost of actually executing the action; C_a, the cost of ensuring atomicity; and C_c, the (potential) cost of contention for access to memory locations. Note that all networks suffer reduced efficiency as contention rises, because more accesses are bound for fewer buffer cells and memory locations (causing serialization delays)—access combining can be helpful in limiting this effect. We consider here concurrency control mechanism-specific costs related to contention. We assume that C_e is constant and unaffected by any concurrency control system we discuss.

In simple locking, a PE generates a request for each lock it needs to acquire. There is no ordering constraint on lock requests, so they may be pipelined. If all locks are acquired, then the atomic action can be sent and executed. If all locks are not acquired, any acquired locks must be relinquished and re-acquired with the entire group in order to avoid deadlock. Thrashing and livelock are still possible, however.

Livelock-free locking (2PL) [EGL76] orders lock requests according to a pre-set priority and emits them one at a time. As each lock is acquired, the next may be requested. This system eliminates deadlock and livelock, at a larger lock acquirement cost. Some enhancement will ensure that a lock request will always be filled before later requests, guaranteeing fairness and eliminating possible starvation. In either locking system, contention causes delay as lock requests become less successful. We note that more recent enhancements have improved the efficiency of the 2PL approach. In this discussion we are merely making motivational arguments, and we feel it unnecessary to unduly complicate our discussion by introducing these enhancements to the comparison. Our simulation studies in section 6.3 utilize a more realistic enhanced version of 2PL.

In memory sorting, accesses are buffered at the MMs. Each PE sends *End-of-Stream* (EOS) tokens that indicate when all accesses (in the current set) have been sent. When the MM receives the EOS token from each PE, accesses can be executed according to a pre-set priority which guarantees atomicity. The handling of EOS tokens may slow progress through the communication network. Accesses are generally blocked for several network cycles prior to execution while the MM receives all accesses and all EOS tokens, and possibly while being sorted (in-line sorting may also occur as each access arrives). Contention at any given MM causes longer delays there, because the MM must accept and sort more accesses before execution.

Table 1 presents a comparison among the C_t, C_a, and C_c of several types of concurrency control systems. ‘Minimum’ means there is no extra cost specific to the type of system identified. Note, accesses that compose the actual atomic action can be pipelined in all cases.

Table 1: Latency cost comparison for several concurrency control systems.

Type	Name	C_t	C_a	C_c
Locking	Simple Locking	Minimum.	Lock accesses must be generated and executed. They can be pipelined.	Locks acquired must be relinquished and lock accesses resent if all not acquired. Thrashing, livelock, starvation possible.
Locking	Livelock-Free Locking	Minimum	Lock accesses must be generated, sorted, and executed. They cannot be pipelined.	Lock accesses which do not acquire locks must be resent.
Access Ordering	Memory Sorting	Some added latency due to EOS token handling.	Accesses buffered at memory and sorted before execution. Limit on buffered accesses.	Longer wait for access execution.
Access Ordering	Local Synchrony	Extra cost associated with access sorting within the communication network	Accesses sorted before entering the communication network.	Minimum.

Our thesis is that the extra costs associated with C_a and C_c in locking systems will outweigh the extra costs associated with C_t in access ordering systems, local synchrony in particular.

3.1.1 Local Synchrony vs. Locking

This section acts as a motivation for our work in the area of implementation of local synchrony. In the following discussion, we assume that the latency of a locally synchronous network is a linear function of the latency of a conventional network, and compare the expected performance of a local synchrony implementation to a 2PL implementation. Our assumption of linearity is based on networks under a full access load that is uniformly distributed (e.g., there are no hotspots), for which out simulation data (presented in Chapter 6) suggests a roughly linear relationship in the networks' performance. We make these assumptions only to simplify and facilitate the arguments in this section. Our simulations in Chapter 6 utilize a highly efficient implementation of 2PL which is more in line with current concurrency control trends in order to make better conclusions about the actual value of local synchrony implementations.

Consider the cost of performing an atomic access to *n* global variables in shared memory. For our comparison with local synchrony we choose a simple locking system that acquires each of *n* variable locks, then performs the multiple access. We assume the architecture consists of an equidistant, shared-memory multiprocessor. Delay is measured in units of cycle time of switching elements within the ICN.

Some definitions:

L_N - The latency (measured in switch cycles) of the network in normal operation

L_{LS} - The latency of the network in local synchrony operation

C_{2PL} - The cost of the multiple access using simple locking

C_{LS} - The cost of the multiple access using local synchrony

We first consider the livelock-free locking system (2PL) presented in section 3.1. In order to guarantee that livelock cannot occur, each lock must be acquired sequentially in a specified order. We consider here only the best case for livelock-free locking, in which all locks are granted on the first request.

We will assume, for simplicity, that $L_{LS} = F*L_N$, where F is a simple factor. In order to both simplify the comparison and give weight to our final results, we will discount the effects of

contention for locks in our calculations. We assume that locks are always acquired but that the process must acquire each lock in sequence before acquiring the next.

Since local synchrony is not required for accesses returning from memory, we assume a low-latency network with latency L_N is in place. The true total latency for normal operation (L'_N)and local synchrony (L'_{LS}) is the sum of the time spent crossing the network to memory (term 1), the time spent actually performing the access at memory (we assume a single cycle—term 2), and the time spent returning to the PE (term 3):

$$L'_N = L_N + 1 + L_N = 2*L_N+1 \quad \text{(EQ 8)}$$

$$L'_{LS} = L_{LS} + 1 + L_N = F*L_N + L_N + 1 = L_N * (F + 1)+ 1 \quad \text{(EQ 9)}$$

The cost of an *n* variable multiple access for 2PL is then:

$$C_{2PL} =(n * L'_N) + (L'_N+(n -1)) = (2n+2)*L_N + 2n \quad \text{(EQ 10)}$$

To acquire a single lock takes L'_N cycles. Since the locking system must acquire locks in order, and cannot acquire a lock until the previous lock has been granted, it takes a minimum of $n*L'_N$ cycles to acquire the locks. The actual accesses can then be pipelined and will be finished in L'_N+(n-1) cycles (the first access takes L'_N cycles, with the succeeding accesses arriving one per cycle after that). For local synchrony, the cost of an *n* variable multiple access is:

$$C_{LS} = L'_{LS} + (n - 1) = L_N * (F + 1)+ n \quad \text{(EQ 11)}$$

Using local synchrony, only the actual accesses need be sent, and these accesses may be pipelined. Thus the entire atomic action can be performed in $L'_{LS}+(n-1)$ cycles (the first access takes L'_{LS} cycles, with the succeeding accesses arriving one per cycle after that).

We can measure the break-even point in relation to n by substituting values for F into the equation $C_{LS} = C_{2PL}$. By solving for F, we find that local synchrony will have lower latency for F < 2n+(n/Ln)+1. For example, if we were performing 5-access actions on a 5-stage banyan network (assuming minimum switch latency), F <= 11 would mean that local synchrony has lower latency for the multiple access. In fact, we measure F to be between 2 and 4 in both our analytical and simulation studies (this is a rough measurement of F based on latency from PE to MM only–the actual value would be less if using a low-latency return network). This indicates that local synchrony is a

significantly better concurrency control mechanism than simple locking, although the cost of locally synchronous operation is higher for a single access. We suggest that an architecture utilizing a locally synchronous network for concurrency and a low-latency network for singleton operations (similar to the RP3 [Pfi85]) might achieve an extremely high degree of efficiency.

If we assume that the local synchrony implementation must use a locally synchronous return network (as would be the case if simplified FIFO combining were in effect), then the cost of a single locally synchronous access (again assuming a single cycle for the execution of the access at the MM—term 2) would be:

$$L'_{LS} = L_{LS} + 1 + L_{LS} = 2*F*L_N + 1 \quad \text{(EQ 12)}$$

Each traversal of the network (to [term 1] and from [term3] memory) now requires the locally synchronous network latency L_{LS}. The total cost of a locally synchronous multiple access would grow to:

$$C_{LS} = L'_{LS} + (n - 1) = 2*F*L_N + n \quad \text{(EQ 13)}$$

In this scenario, local synchrony will have lower latency if $F < n+(n/2L_N)+1$. In the example discussed above, F< 6.5 would be necessary. Again, our rough estimate from analytical and simulation studies is that F is between 2 and 4. Furthermore, the equations illustrate that as n increases, the ‘break-even’ value of F also increases, while the general performance differences between the two systems remain static. As the networks perform larger and larger multiple accesses, local synchrony provides an improving alternative to locking.

Let us now assume that lock requests may be pipelined. This is the simple locking system presented in section 3.1. Note that the pipelining of lock requests introduces the possibility of livelock and thrashing. We consider here only the best case for simple locking in which all locks are granted on the first request. Pipelining of lock requests significantly affects the latency of an atomic access. Equation 5 gives the cost of a multiple access in this paradigm:

$$C_{2PL} = (L'_N + (n -1)) + (L'_N + (n -1)) = 2 * (2*L_N + n) \quad \text{(EQ 14)}$$

The lock requests are pipelined (the first access takes L'_N cycles, with the succeeding accesses arriving one per cycle after that). After all lock requests have been granted, the access

requests can also be pipelined (the first access takes L'_{LS} cycles, with the succeeding accesses arriving one per cycle after that). Again, we can measure the break-even point in relation to *n* by substitution. Even with the advantage of pipelining of lock requests, local synchrony will have lower latency for all n>1 if F is reduced to 3 (this uses the original, non-combining network assumption for local synchrony). Both simulation and analytical results for the simple equidistant implementation indicate F is between 2 and 3.

Note that this comparison considers only the best-case scenario (no contention) for the locking systems discussed. In any useful computation there will be some contention for locks, which will adversely affect system performance for locking systems. In an access ordering system the effect of contention is limited to possible slowdowns that are due to multiple accessing of the variable itself. The effect is less significant here because an access is still guaranteed to be executed (a lock request may not be granted). The use of access combining can also help to limit this effect.

This argument assumes that each network always accepts new inputs immediately (throughput = 1.0). Chapter 6 will demonstrate that, in terms of raw power, local synchrony implementations have generally lower throughput than locking systems. We point out, however, that slightly lower throughput (our data show the throughput of the local synchrony implementation is always greater than half the throughput of the locking system) has little effect on the cost argument made above. This is because lower throughput only extracts a linear cost *before* accesses enter the network, not during their journey through the network, and because locking requires the release of at least twice the number of accesses as local synchrony for the same atomic action.

For example, our simulation studies for banyan networks with similar switch architectures predict throughputs of approximately 0.46 for the locking system and 0.26 for local synchrony (our measurements of local synchrony operation indicate that its throughput is always greater than half that of simple locking). The throughput of the network corresponds to its acceptance rate. We can rewrite equations 3 and 4 to take this acceptance rate into account. It takes an access an aver-

age of *(1/0.46) - 1* cycles to be accepted into the locking network, while a local synchrony access takes *(1/0.26) - 1* cycles to be accepted. The cost of an *n* access atomic action in the locking system is then:

$$C_{2PL} = (1/0.46) - 1 + (n+1) * TL_N + (n-1)*((1/0.46) - 1)$$
$$= (2n+2)*L_N + 1 + n/0.46 \quad \text{(EQ 15)}$$

It takes the first lock request *(1/0.46) - 1* cycles to be accepted into the network. The rest of the lock requests and the first access are performed serially, and we assume that network inputs are not multiplexed, so each of those enters an empty input and is not assessed the acceptance charge (the only traffic ahead in the input are previous lock requests which have already returned). The rest of the accesses may be pipelined and thus are assessed the acceptance charge. The cost of an *n* access atomic action in the local synchrony system is:

$$C_{LS} = n*((1/0.26)-1) + TL_{LS} = ((0.74*n)/0.26) + L_N * (F + 1) + 1 \quad \text{(EQ 16)}$$

Since all accesses are pipelined in local synchrony, all are assessed the acceptance charge of *(1/0.26)-1* cycles. We again solve for F and find that $F=2n+1+(n/(0.46*L_N)) - ((0.74*n)/(0.26*L_N))$. For five access atomic actions in a five stage network, the ‘break-even’ point for local synchrony would be approximately F<=10.4, which is not considerably different than our previous result (F<=11).

3.2 An Implementation of Local Synchrony

In this section, we present an implementation of local synchrony on an equidistant Banyan-type network, discuss various implementation mechanisms, including the switch operation algorithm, and present a proof of deadlock freedom for the implementation. In Chapter 4 we present a general technique which allows local synchrony to be implemented on any shared memory architecture.

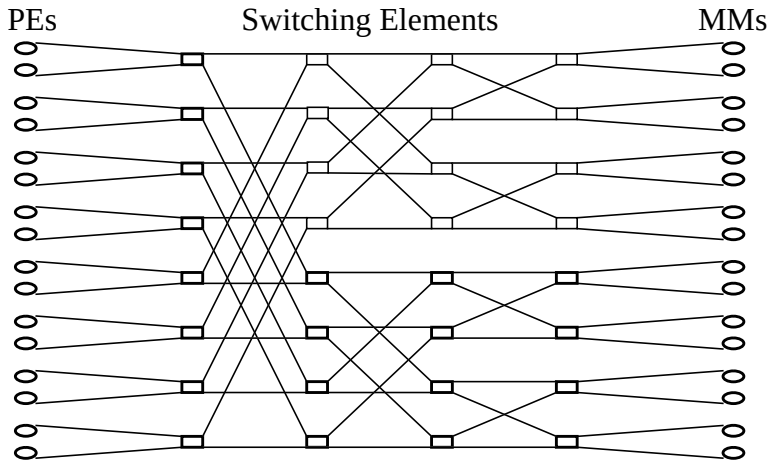


FIGURE 3. An *n*=16 Banyan architecture.

3.2.1 Implementation Plan

We consider the implementation of local synchrony on an Ultracomputer-like Banyan network consisting of $\log_2 n$ stages of *n/2* switching elements per stage, where *n* is the number of processors (PEs) and memory modules (MMs). For this implementation, we assume switching elements are 2-input, 2-output. Figure 3 illustrates an *n=16* Banyan architecture. We assume the basic, indivisible shared-memory access for such an architecture to be the atomic *fetch-and-op*.

We discuss seven issues of importance in any such implementation with respect to switch architecture: the transport mechanism, the routing mechanism, buffer management, flow control, the message ordering system, the combining system, and the deadlock avoidance system. The first four of these have been identified in [ReF87]. Figure 4 shows a diagram of a simple SE.

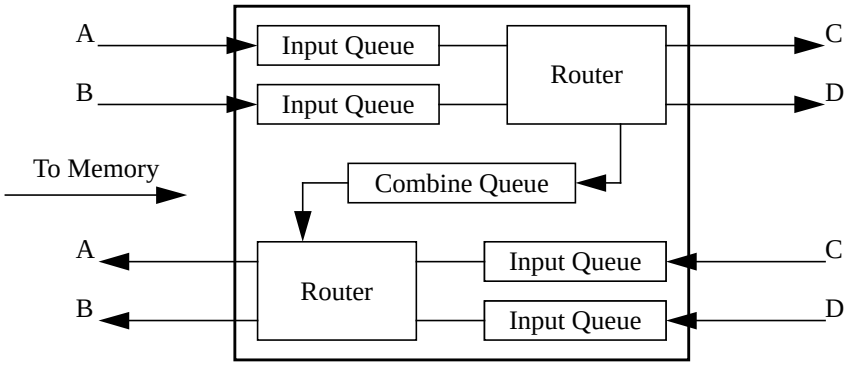


FIGURE 4. Simple Switching Element.

3.2.1.1 Message ordering system

We discuss this system first because several other issues rely on the decisions made here. Local synchrony requires that shared memory accesses each be assigned a unique timestamp, which represents a logical execution time for that access. Accesses are then ordered by timestamp at some point prior to execution. This timestamp includes information pertaining to the logical time of execution, the ID of the sending PE, and the access order for the accesses from that PE. Combining information (the destination address) is also included in the timestamp if the implementation is to include FIFO combining.

Accesses could be transferred through the network unsorted, leaving the MMs to ascertain when to conservatively execute a given access. Scalability arguments lead us to choose a system in which the sending PEs sort accesses before emission. The network acts as a merging network for sorted streams of accesses. Note that a given PE must generate *all* the accesses that will be emitted during a given ltu before any accesses can actually be emitted, unless the PE can guarantee that accesses will be generated in timestamp order.

Local synchrony requires only that accesses be executed in timestamp order at shared memory. It is therefore possible to implement a separate, low-latency network to return the results of executed accesses to the PEs. Such a network would route accesses without regard to their timestamps, which could shorten access latency times. Such an implementation, however, eliminates the system's ability to perform FIFO combining. In order to demonstrate the feasibility of FIFO combining, we assume in our further discussion that accesses remain sorted on return from shared memory.

3.2.1.2 Transport mechanism

We choose the store-and-forward access transport paradigm for the implementation, although local synchrony is compatible with wormhole routing [DaS87] and virtual cut-through [KeK79]. We assume that shared memory accesses have a fixed size, which greatly simplifies the model (for this discussion, we assume word access, although local synchrony is not limited in this

respect). Because the implementation of local synchrony provides a basis for isochrons, word size accesses may be used without losing the flexibility of variable size memory access. An isochron provides atomic access to any number of shared memory words.

The size of the shared memory access packet depends on several factors. Generally, however, the additional information required by a local synchrony implementation can be implicitly represented. Variables in the packet size are the number of PEs and the size of the virtual shared memory, the size of operands for the memory operations to be performed, and combining information, if combining is in effect. FIFO combining, as discussed in section 2.4, can increase the size of the packet, if packets carry information about where they have been combined.

Consider, for example, an Ultracomputer architecture with 64K processors, an address space of four gigabytes, and 32-bit operands. In such a system, a packet (data unit) size of 16 bytes would be sufficient to implement local synchrony. Four bytes would cover the destination address in shared memory (2^{16} address spaces per MM). The PID element of the timestamp would consume 2 bytes. Other elements of the timestamp can be implicitly represented (we will assume that separate ‘dummy’ packets mark changes in the ltu and that accesses from the same PE cannot ‘pass’ one another.). The operation to be performed, along with a ghost message bit, would fit easily into one byte. The operand of the operation fills a 4 byte word. Finally, combining information for the network (16 stages) would take 2 bytes. This leaves more than 3 bytes of space for error correcting codes and other information.

Buffering complexity using the store-and-forward paradigm with fixed-sized packets is relatively simple. At least one packet buffer is necessary at each input channel at all times. More complex buffering strategies are possible as long as they meet this criterion.

3.2.1.3 Routing mechanism

The routing mechanism employed by a local synchrony implementation may be separated into two distinct steps. First, the router chooses the next packet to be routed from among those at the head of each of its input channels by choosing the packet with lowest logical timestamp.

Because the streams of input packets are already sorted, this amounts to a conservative choice of the earliest timestamped, not-yet-routed packet. We also note that the routing mechanism must have a packet (or a ghost message, which in this implementation is represented by a packet) on each of its input channels in order to determine which will be routed next.

The second step is to route the chosen packet to the correct output channel. If we assume that the portion of shared memory held by a given MM is a contiguous set of addresses, and that the MMs are ordered contiguously in the architecture, then routing is simply checking whether the destination address of the packet is greater than or less than a given address and choosing the output channel accordingly.

3.2.1.4 Deadlock avoidance system

Ghost messages [Ran87] [RBJ88] avoid deadlock within the network by disseminating information from SE to SE as soon as it is available. Ghost messages are similar to the null messages found in parallel simulation literature [ChM79]. Null messages ensure deadlock freedom in groups of processes that communicate via message passing in a parallel simulation.

When an SE passes (or is waiting to pass) an access on a given output channel, its other output channel may be idle. This condition can cause deadlock. Deadlock can be avoided by passing a *ghost message,* a copy of the temporal components of the access most recently sent (or waiting to be sent) on the other output channel. The timestamp of this ghost message can break potential deadlocks further along in the network.

Ghost messages are identified by a special bit set within the access packet. Alternatively, the destination address of a ghost message will never be within the destination range covered by the receiving SE. Ghost messages require little or no processing and do not clog the network, since they utilize otherwise idle communication channels and otherwise empty buffer cells.

A ghost message provides an SE with the lowest possible timestamp of any future access that will arrive on that channel. This allows the SE to make more routing decisions and break pos-

sible deadlocks. An incoming ghost message that becomes the next to be routed by a given SE must be propagated along each of that SE's output channels.

The information contained in a ghost message is superseded by the arrival of a normal access or another ghost message, either of which would have a later timestamp. Ghost messages are overwritten (erased) in such a case. Erasing ghost messages that have been superseded by new ghost messages or actual accesses limits the proliferation of ghost messages within the network, although it is possible for ghost messages to reach the memory edge of the network, where they would be deleted.

In section 3.2.2, we present a proof of deadlock freedom based on ghost messages.

3.2.1.5 Buffer management

Buffer management can be simple or complex in this implementation. Simple buffer control reserves a certain number of packet cells for each input channel. More complex buffer management can draw from a pool of cells to queue incoming and outgoing packets on all channels. Local synchrony, which cannot abort (remove without executing) accesses once they enter the system, requires that each input channel have access to at least one packet cell (or *already* be using one) at all times to avoid buffer deadlock. An implementation might also include buffers on output for flow control purposes. For the purposes of analysis, we can think of output buffer cells as extra input buffer cells for the next stage.

Because local synchrony is a conservative system, the simple buffer management scheme is appropriate. Deciding how many cells to assign to a given input channel must be done with care, however, since excess buffer cells will adversely affect the combining system, as discussed in section 3.3.1.7.

3.2.1.6 Flow control

A receiver-controlled protocol can achieve adequate flow control of packets within the network. We will assume that packet transmission fault-tolerance is controlled by a lower-level send-acknowledge system, which we will not discuss here, because the implementation of local

synchrony has no effect on such systems. In normal operation, the receiving node uses a control line to signal the sender that it has open input buffers for that channel. When the receiver's input buffers are full, the control line signal is changed, notifying the sender that no input will be accepted until the control signal returns to normal.

Because local synchrony depends on a continual flow of information for the network to operate smoothly, this receiver-controlled protocol must be augmented when necessary to allow receivers to request action from idle senders. Receivers would use another control line to request action on a given channel when holding input on other channels. This request requires the sender to make progress as soon as possible. Senders can request action on their own input channels if necessary.

This system would be especially useful at the input stage of the network, where SEs could request that idle (with respect to shared memory access) PEs pass information that will allow the network to avoid blocking. In response to an action request, an idle PE would increment its logical clock and send a token.

A flow control policy must also be implemented in order to use the mechanism described. This policy would depend on the buffer management mechanism in place. In a simple system, where each input channel is permanently assigned a single packet buffer, this policy would be straightforward. Where buffer cells are dispensed from a pool, however, the flow control policy must enforce limits on the number of cells available to any one channel and guarantee that each channel has access to at least one buffer cell at all times. This guarantee is necessary to avoid buffer deadlock within the network.

3.2.1.7 Combining system

Dancehall-type networks like the Banyan network are highly compatible with access combining [GLR81] [KRS88]. Because the implementation presented here assumes that accesses are ordered during transmission to and from shared memory, accesses combined at a given SE will return in the same order in which they left the switch. This allows us to implement FIFO combin-

ing, which in turn allows combining information to be processed more quickly without changing the combining mechanism.

The size of a combining buffer and the size of an access are the same, given that accesses themselves carry information about whether they were combined at a given stage (see section 2.4). Combining two accesses changes, at most, only the operation and operand of the forwarded combined access, and the combining bit for that stage of the network. The other access may be copied into the combine buffer, and used to decombine the accesses on return.

Since combined accesses return to a given SE in the same order they left it, decombining information can be kept in a FIFO queue [Ran87], instead of the traditional associative search queue. Each access packet can include bits that determine whether a given access was combined at a given stage in the network. An SE then needs merely to check one bit of the returning packet and, if the packet was combined at that SE, pop the decombining information off the FIFO combining information queue.

The timestamping system can also be modified by including the destination address in the timestamp. In the Banyan network, the paths to a given MM from the PEs form a tree. The paths of two combinable accesses, sent by different PEs in the same ltu, will converge at some point in the network. If the streams of accesses are sorted by destination address and the other components of the timestamp (in this order: *ltu*, destination address, *pid*, *tock*, as defined in section 2.5), combinable accesses will be combined at this point. This guarantees that all combinable accesses within a given ltu will be combined, given the size limits on the combining information queues. We expect that streamlining and simplifying the combining mechanism in this way will expand its power and usefulness.

The size of the necessary combining queue in a given node in the network depends on the size of the network, the stage in which the node resides, and the amount of space allotted to communication buffers in each node. Local synchrony causes the network beyond the outputs of a switch to act like a large FIFO queue. The size of this ‘virtual’ queue is the maximum number of

outstanding combined accesses that a switch can have at any given time. The maximum necessary combining information queue size is given by:

$$C_{MAX} = nS+oM+t \quad \text{(EQ 17)}$$

where S is the number of switches reachable from the given switch, *n* is the number of buffer cells in a switch available to an access from the given switch, *M* is the number of MMs reachable from the given switch, *o* is the number of buffers available at an MM, and *t* is the number of buffer cells at the given switch that can hold combined messages before decombining. Note that this limit does not take into account the limits on the number of accesses that a given switch might receive within a given logical time unit. If that number is less than the above limit, the presence of tokens in the access stream will further reduce C_{Max}.

It may be practical to limit the size of the combining buffer, since it is unlikely that the network will be operating at capacity, that all inputs will be combinable, and that the 'virtual' queue ahead of a given switch will contain only accesses that passed through that switch. More complex input buffering schemes can allocate many more cells to a given channel, thus making C_{Max} quite large. Local synchrony is not compromised if two combinable accesses are not combined. The only effect of this situation is a potential loss of throughput.

3.2.2 Deadlock freedom

Our discussion of deadlock freedom for local synchrony on Banyan (Equidistant) networks relies on the following assumptions about the implementation:

1. The network communicates with fault-free systems that provide input and consume output at the rate the network operates.

In other words, there is always input at the network inputs, and output is consumed immediately at network outputs. This assumption only simplifies progress assertions about the PEs and MMs attached to the network, and does not limit the applicability of the proof.

2. Switches in the network are able to buffer at least one access (packet) on each input channel. Once a packet occupies a buffer cell it can only progress toward its destination; it may not be aborted.

Because local synchrony guarantees that accesses will ultimately be executed in order, to make routing decisions switches must buffer incoming accesses on each input channel. Accesses must not be aborted once they have entered the network, since this will, in most cases, compromise the operative conditions of local synchrony.

3. Switches in the network perform routing operations sequentially.

In other words, an SE may perform only one operation at a time. An SE may not, for example, block, attempting to output a routed access while at the same time choosing and routing its next access.

Assumption 3 is a simplifying assumption that clarifies the proof. Alternatively, SEs may multiplex operations for more efficiency, given that conservatism is maintained and that an SE will not block on a lower priority operation without returning to a previously blocked higher-priority operation.

4. Switches in the network are able to make routing decisions based on a known priority scheme.

In order to implement local synchrony, accesses are assigned timestamps, which arbitrate the routing decisions of switching elements. These timestamps provide a static priority system guaranteed to arbitrate the routing decision for any two accesses. The routing operation of a switch merges two sorted (by timestamp) streams of accesses.

We assume the network itself consists of 2-input, 2-output switching elements and that all SEs and communication channels are fault-free.

Deadlock is characterized by the occurrence of a circular set of dependencies, in which each element within the set requires progress by another element within the set before the element itself may progress. In the case of a Banyan network with the above assumptions, the set may only include switching elements.

It is clear that the zero-stage (1 PE, 1 MM) and the one-stage (2 PEs, 2 MMs, 1 SE) cases are deadlock free, since the largest possible set of SEs in either case has fewer than two members.

We will first discuss the two-stage case (4 PEs, 4 MMs, 4 SEs). We will then state lemmas leading to our proof of deadlock freedom.

Figure 5 illustrates the two-stage case. The architecture is shown in figure 5a. We assume that information flow is left to right. Figure 5b illustrates the two possible circular dependencies.

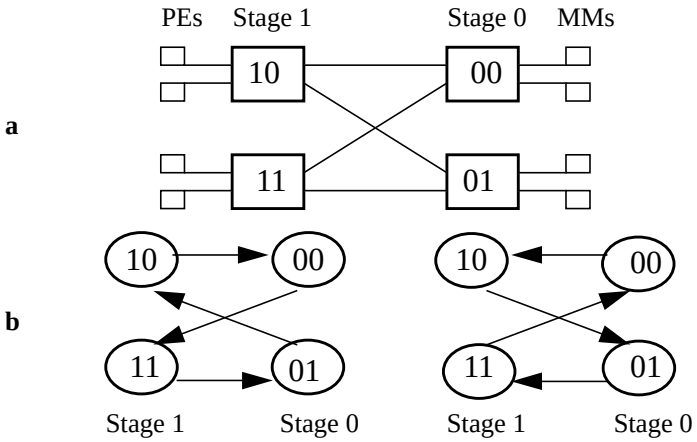


FIGURE 5. a) Two-stage Banyan Network b) deadlock dependency graphs

In each case, a left arrow (whether horizontal or diagonal) represents a consumer's dependency on the producer. A stage 0 node (consumer) awaits input from a stage 1 node before it may itself progress.This is a *type 1* dependency. Since the producers (10 and 11) in this situation are also blocked, deadlock has occurred.

The right arrows (whether horizontal or diagonal) represent a producer's dependency on the consumer. A node in stage 1 cannot progress because the input buffer in the node that will receive its next access is full. The stage 1 node then depends on the stage 0 node to progress before it may progress. This is a *type 2* dependence.

Deadlocks, such as the one in figure 5b, can be broken if information about each producer's latest action passes from the producer to *each* of its consumers. This information already passes to the consumer receiving the latest access from the given producer. The other consumer may receive this information as a *ghost message* [Ran87], which reports the last timestamp of an access sent by the producing switch.

With the information supplied by the ghost message, one of the two consumer nodes (00 and 01) will be able to progress, because no access will arrive on its empty input channel that has a lower timestamp than the access on its other input channel. The access can then be passed, breaking the deadlock.

In figure 6a, for example, SE 10 sends an access with timestamp t=5 to SE 00, then blocks trying to send an access with timestamp t=6 to SE 00. SE 11 sends an access with timestamp t=7 to SE 01, then blocks trying to send an access with timestamp t=8 to SE 01. Deadlock occurs. However, SE 11 can send a ghost message to SE 00 based on the access sent to SE 01,having timestamp t=7, as seen in figure 6b. Now SE 00 is able to process the timestamp t=5 access sent from SE 11, as it can conservatively choose that as its next action. Thus deadlock is avoided.

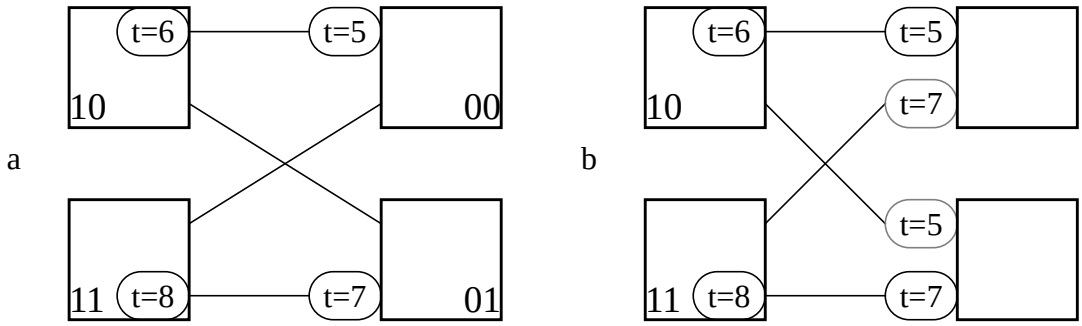


FIGURE 6. a) Deadlock b) Ghost message solution

While local synchrony provides a timestamping system guaranteeing that all timestamps are unique, this is not a requirement for deadlock freedom.Ties may be broken, in the case of a ghost message and an access, by sending the access (and erasing the ghost message, since it is no longer useful).

Furthermore, a ghost message having a later timestamp can supersede an earlier ghost. message. An arriving ghost message may therefore overwrite a ghost message already in the input queue. This limits the proliferation of ghost messages in the network.

The ghost message may take several forms, depending on the timestamping scheme in use. The rest of our discussion will characterize circular dependencies in larger banyan networks and show that ghost messaging is sufficient to guarantee deadlock freedom for all such cases.

We now consider deadlock in a network of any given size. Such a situation can be characterized by a *deadlock dependency graph* (DDG) [DaS87], which includes each node in the set of deadlocked nodes and arcs representing a given node's dependencies on others within the set.

There are only two types of dependencies that can cause an SE to block and possibly contribute to deadlock in a banyan network implementing local synchrony. The only situations in which an SE can block are type 1) waiting on input from one of its input channels, and type 2) waiting to output on a full output channel.

In the figures below, type 1 blocking will be represented by a left arrow, and type 2 blocking by a right arrow. This corresponds to the uni-directional flow of information [in this case, from left (PEs) to right (MMs)] in the banyan network. A node's inputs are on the left, its outputs on the right. Figure 7 gives an example of a DDG.

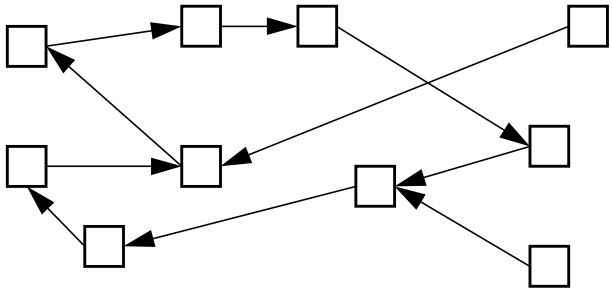


FIGURE 7. Sample deadlock dependency graph (DDG).

> ***Lemma 3.1:*** In a local synchrony implementation on a Banyan network, the DDG representation of a deadlock must include both type 1 and type 2 arcs.

Proof: Type 1 arcs point left. Type 2 arcs point right. Due to the nature of the network in question, arcs may not 'wrap around' from one side of the network to the other. Lemma 3.1 must hold for there to be any circularity--and thus any deadlock. ■

There are certain conditions under which ghost messages will be generated and propagated within the deadlocked set of nodes (those nodes in the DDG). Since a ghost message must use the same channels as normal accesses, only idle channels (those on which a type 1 dependency is based) may be used for their transmission among the nodes in the DDG. For the purposes of this

proof, we need only consider ghost messages originating at nodes in the DDG on which another node holds a type 1 dependency.

Also, since a ghost message comes from information about an access to be output by the node, ghost messages are only generated at nodes in the DDG having a type 2 dependency (nodes that have routed an access but are unable to send it due to full buffers ahead). An example of such a *type L node* within the DDG is shown in figure 8. Note that input channel dependencies are unimportant.

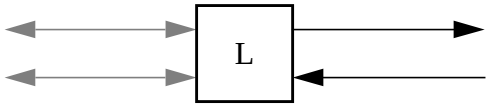


FIGURE 8. Type L node within the DDG.

Lemma 3.2: At least one type L node must exist within the DDG.

Proof: A DDG must include a minimum of one circular path of dependencies which, if followed from a given node, will return to that node. Consider any circularity within a DDG: there must be at least one leftmost node within the circularity (a ‘leftmost’ node would be any node in the leftmost stage of all those nodes within the circularity). Any one of the set of leftmost nodes is a type L node. None of its inputs may be involved in the circularity, so both outputs must be. The node cannot have the same type of dependencies (both type 1 or both type 2) on each output, or there would not be a circularity in the DDG. It must thus be a type L node. Any node in the DDG that is also in the leftmost stage of the network (nearest the PEs) must be a type L node, by virtue of assumption 1 above and the node’s inclusion in the DDG. ■

Lemma 3.2 guarantees that at least one ghost message will be created in the DDG by showing that at least one node that will generate a ghost message (type L) must exist within the DDG. Lemma 3.3 discusses properties of ghost messages created in the DDG.

> **Lemma 3.3:** When a set of nodes is deadlocked, for each ghost message, G, generated by a node within the DDG, there exists a blocked access A, at the front of an input queue of some node in the DDG, such that timestamp t_G of G is greater than timestamp t_A of A.

Proof: Figure 9 depicts two nodes from a DDG. As we have discussed, only type L nodes within the DDG will generate and propagate ghost messages. Node A is a type L node, and thus will generate a ghost message, G2, based on A2, the access it has most recently queued for output to node

B. Node B, then, holds at the front of its input queue an access, A1, which, by virtue of the conservatism of local synchrony, is guaranteed to have timestamp $t_{A1} < t_{A2} = t_{G2}$. ■

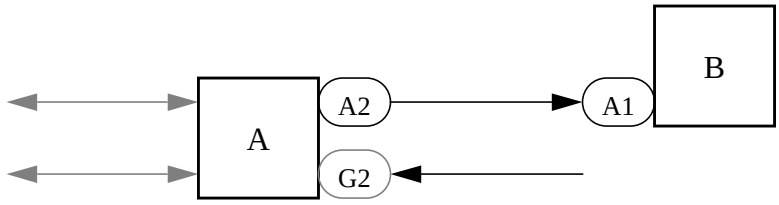


FIGURE 9. Illustration of proof of Lemma 3.3.

> **Lemma 3.4:** Ghost messages generated at type L nodes in the DDG will cause information (either a ghost message or an access) to be propagated on every channel in the DDG having a type 1 dependency.

Proof: By contradiction. Note that type 1 dependencies must exist within the DDG, according to lemma 3.1. Consider a channel (A), with a type 1 dependency, that will not receive information (either ghost message or access) from other nodes in the DDG. Figure 10 illustrates this situation. The node where the dependency arrow ends (N1) cannot have a type 2 dependency on its other output channel, because it would then be a type L node and would create a ghost message that would be propagated along channel A. Node N1 also cannot have type 2 dependencies on both input channels, because then it would not be blocked. N1, therefore, must have at least one type 1 dependency on one of its input channels (B). Neither ghost message nor access information may propagate on channel B, because N1 would then be able to propagate a ghost message or an access along channel A. [Note that if N1 has type 1 dependencies on *both* its input channels, then no information can be propagated on one of those channels (B, in our discussion), because N1 would then have input on both channels and would thus be able to send information on channel A.] This scenario, in which no information is propagated on either input channel, is a generalization of the case we discuss. Each channel is then part of a chain of dependencies that eventually reaches a contradiction. We consider one of those channels (B) next.

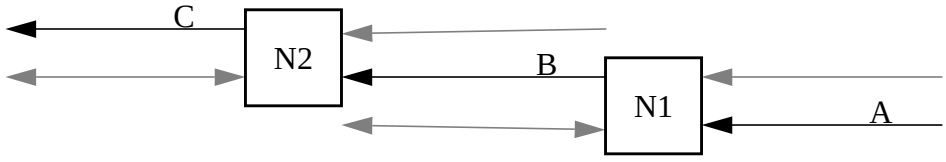


FIGURE 10. A chain of dependencies.

Since channel B receives no propagated information, the node where channel B ends (N2) must have the same dependency topology as N1, and so on; thus, a chain of dependencies exists. This chain of dependencies (channels A, B, C,...) are all type1. It cannot go on indefinitely, but only to

the left edge of the network. We have established that the nodes in the chain must have the dependency topology of N1, and—because of the nature of the chain—stages, the chain must end at a node N' that is in the leftmost stage. N' must be blocked, because of the type 1 dependency on it, and its inclusion in the DDG. Assumption 1 places input on each of N's input channels. If blocked, N' must have a type 2 dependency on its other output channel, and will therefore be a type L node. This will generate ghost message at N', which will propagate ghost messages or accesses on the chain of dependencies in question. Each node within the chain of dependencies is blocked waiting for input and may therefore propagate information on the output channels that form the chain when input arrives, in the form of a ghost message or access. This contradicts our hypothesis and proves lemma 3.4. ■

We now present the main theorem of this section:

> **Theorem 3.1:** Ghost messages are sufficient to break any possible deadlock in a local synchrony implementation on a Banyan (equidistant) network.

Proof: According to Lemma 3.1, a non-empty set of nodes with one or more type 2 dependencies on their input channels must exist within the DDG. Choose from this set the node (N) that has A1, the access of least timestamp, at the front of one of its input queues. Node N cannot have type 2 dependencies on either of its output channels, because the nodes connected to those channels would then have earlier timestamped accesses, according to the conservative properties of local synchrony. If Node N has type 1 only or no dependencies on its output channels, then it must have a type 1 dependency on at least one input channel, or it would not be blocked. Thus, choosing node N places A1 at the head of the input queue of a channel with a type 2 dependency. Node N is illustrated in figure 11.

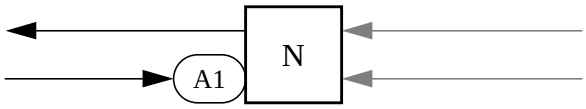


FIGURE 11. Node N, as chosen in the proof of theorem 3.1.

Lemma 3.3 and the choice of node N means that A1 has an earlier timestamp than *every* ghost message generated by nodes in the DDG. Lemma 3.4 guarantees that a ghost message or an access will be propagated to node N along the other input channel. Either situation guarantees progress and node N will be able to make a routing decision. Figure 12 illustrates each possibility. Within the DDG, only ghost messages are generated after deadlock has arisen. Because of this, the transfer of an access from node to node within the DDG denotes progress. Thus, if the information propagated to node N is an access (A2), the deadlock is already broken, and progress will continue as node N routes either A1 or A2 (see figure 12a).

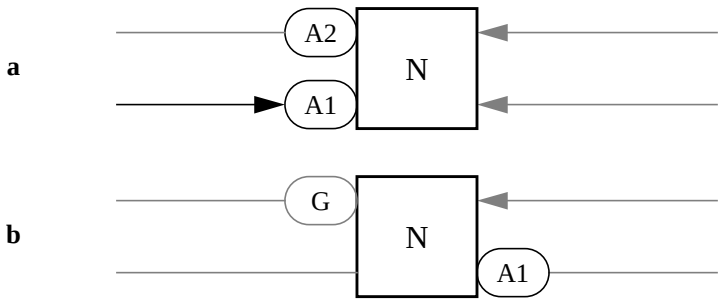


FIGURE 12. Progress at node N, due to ghost message propagation. a) Arriving access A2 denotes progress has already occurred. b) Arriving ghost message allows routing of access A1.

If the propagated information is a ghost message (G), the router will choose A1, as it has a lower timestamp, according to lemma 3.3. Once chosen for routing, A1 may be sent, since there are no type 2 dependencies on the output channels of node N. Thus a node (N) makes progress within the DDG, and the deadlock is broken (see figure 12b). ■

3.3 A Local Synchrony Switch Design

We present the design for a locally synchronous combining switch to be used in the MIN interconnection network of an equidistant Banyan shared memory multiprocessor (e.g. the NYU Ultracomputer [Got87]). The switch is synchronous and consists of two components: a forward path component, which handles the transfer of accesses to the shared memory and combining; and a reverse path component, which handles the transfer of responses to the PEs and decombining. Local synchrony allows simplified combining and decombining of accesses, which requires no associative search of combining queues.

The switch design is for a 2x2 switch node. The access message length is 64 bits, which allows the architecture to include, for example, 64 PEs and MMs, each with 2^{16} memory locations, with a small set of supported memory operations. Routing information is included in the fixed-length access message. While a single access is designed to manipulate only a single 32-bit memory word, locally synchronous operation allows double- or multi-word loads, stores, and other operations to be performed atomically, eliminating the need for variable length access messages or specific design for double-word operations (in fact, local synchrony in combination with split operations [Wil93] allows any memory operation to be performed on double-length or more mem-

ory words using fixed length accesses, whether or not the memory is contiguous or distributed among several MMs). Since we are able to eliminate the need for variable length accesses, our design may be implemented in a single cycle switch, as opposed to other designs, which require multiple-cycle operation.

We find, after considering the implementation of this design using off-the-shelf components, that expected performance is comparable to current combining switches for non-locally synchronous operation [DiK92]. Our work compares directly to these designs.

3.3.1 Architecture Overview

The multistage interconnection network that links PEs and MMs in the NYU Ultracomputer and other similar architectures generally consists of two separate networks, a forward network, which transmits global memory accesses to the MMs, and a return network, which transmits responses to the PEs.

Each switch within the network, then, consists of one component for each of these networks. The forward path component (FPC) and the return path component (RPC) are separate entities, each performing its own operations. A combining network must communicate information about access combining between the FPC and the RPC. Figure 13 presents an overview of the switch architecture. Note that information (about combining of accesses) flows only from the FPC to the RPC; this is the only connection between the two components.

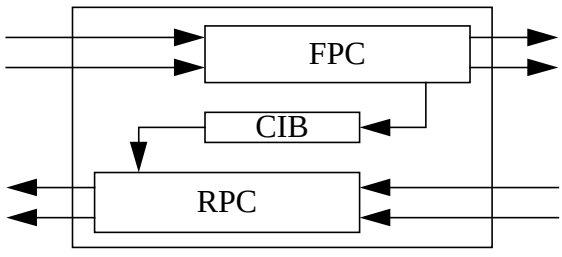


FIGURE 13. Overview of switch architecture.

Combining information is stored in a combining information buffer (CIB). For our purposes, we will include the CIB in the design of the RPC. Another advantage of local synchrony is that responses will arrive at a switch in the same order that their accesses left that switch. Combin-

ing information can therefore be kept in a FIFO queue and merely popped off of the queue in order, unlike the associative lookup that is necessary in the general implementation of the combining switch.

The implementation we present here takes advantage of the limitations on the number of possible outstanding combined accesses from any one switch. The combining information queue is implemented in a way that guarantees all possible combining (within the limits of the local synchrony ltu). Locally synchronous operation already guarantees that any combining operation will occur at the earliest possible switch within the network. This is not the case with conventional combining systems.

3.3.1.1 Packet Format

An access packet in our implementation consists of two 32-bit sub-packets, the routing/address sub-packet (fields 1–6) and the data sub-packet (field 7). Each is handled separately but concurrently within the switch, because each requires a different manipulation. Table 2 presents the packet format.

Table 2: Packet Format

Ghost	Token	MM Addr	Mem Loc	Op Code	PE Addr	Data
1 bit	1 bit	6 bits	16 bits	2 bits	6 bits	32 bits

The first two fields indicate non-access designation for the packet. An arriving packet may be a ghost message, which transmits flow control data only among switches, or a token, which delineates local synchrony ltus

The 6 bit MM address and 6 bit PE address allow 64 PEs to access any of 64 MMs. In order to differentiate between responses, most conventional systems must apply an outstanding request index to pipelines accesses. Local synchrony eliminates the need for the PE to explicitly include an outstanding request index in the access message. The PE receives responses in the order their respective accesses were sent out. The MM/PE address and the memory location accessed

combines with the implicit priority induced by tokens and the order in which the accesses are released to create a unique identifier, or local synchrony timestamp, for each access.

3.3.1.2 Operations Supported

For this design, we implement a small set of completely combinable memory operations. While this set of operations is smaller than those associated with other combining systems, in many cases the added power of locally synchronous operation replaces the need for more complex accessing. Our goal was to limit combinable access types to a set of operations that could be combined in any combination. This is not generally the case in conventional combining systems [DiK92]. Limiting the size of global memory contained within each MM, or limiting the size of the network, would allow more types of memory operations, if necessary.

Local synchrony requires that combinable operations be combinable in any orientation (perceived order of execution). This is possible for many types of operations. Our design allows only four types of operations. We choose a FETCH&STORE operation (the value returned is the previous value of the memory location) and a REPLACE&ADD operation (the value returned is the final value of the memory location, rather than the initial value, as in FETCH&ADD) to implement simple access operations. We choose REPLACE&ADD rather than FETCH&ADD because the combine and decombine operations are simpler in the former. The other two operations are left undefined, but might be used for other combinable operations or non-combinable operations such as split operations [Wil93], broadcast, or partial word operations. The REPLACE&ADD operation with operand of zero implements a simple load.

Table 3: Combining Situations

Input 0	Input 1	Sent	Saved	Return 0	Return 1
R&A(A)	R&A(B)	R&A(A+B)	R&A(B)	R&A(R-B)	R&A(R)
R&A(A)	F&S(B)	F&S(B)	R&A(A)	R&A(R+B)	F&S(R)
F&S(A)	R&A(B)	F&S(A+B)	R&A(B)	F&S(R-B)	R&A(R)
F&S(A)	F&S(B)	F&S(B)	F&S(A)	F&S(R)	F&S(A)

Table 3 indicates the ways in which these operations are combined and decombined within the switch. Any combination of these simple operations, in any order, may be combined within the network, if they are bound for the same memory location. The Sent column represents the combined operation forwarded, while the Saved column represents the operation saved in the combine buffer. The Return columns indicate the decombined responses to the original accesses (R denotes the value returned in the combined response).

The situations defined by rows 2 and 3 of Table 3 are delineated by use of the ghost bit in the saved operation. Since no ghost message will ever be combined (see section 5.3.1.7), the ghost bit of the saved operation will always be zero. The FPC uses the ghost bit to differentiate between the two different combinings, which have the same sent and saved accesses. The RPC recognizes the difference by checking this bit in order to make the correct decombining decisions, then resetting the bit to 0 when the response is decombined.

3.3.1.3 Flow Control Logic

The use of ghost messages disseminates flow control information within the network. We require that some message--either a true access, a ghost message, or a token message--be transmitted from each output of each switch during each cycle. Because ghost messages travel only on otherwise idle channels, and are overwritten by later ghost messages or true accesses, they do not adversely affect the performance of the system. Ghost messages only enhance the flow of true accesses within the network.

This design implements flow control by using load inhibit signals, which instruct receiving edge sensitive latches to accept input if the previous input has been forwarded. The load inhibit signals for the input latches of a switch are also sent to the sending switches at the previous stage, where they signal re-transmission of the previous output, which has been saved in an output latch.

The switch operation generates a ghost message for each idle channel during each cycle. Untransmitted ghost messages are ignored, because a message (ghost or true access) with similar or updated information will be generated during the next cycle. Untransmitted ghost messages in

input latches may be flushed as well. Ghost messages therefore use only idle channels and will not slow down or hold back true access or responses.

3.3.2 Forward Path Component Design

Figure 14 presents the design of the FPC. The components of the FPC and RPC are compatible with current off-the-shelf technology. Each input and output channel is 64 bits wide, with an incoming packet divided into its constituent sub-packets and placed in 32 bit edge sensitive latches (A).

The routing control (G) makes a routing decision based on several inputs, and tells the MUX components (B, C) which sub-packets to forward to the output buffers (D). Edge sensitive output latches save the outputs in case they are not accepted by the next stage of the network--a condition indicated in each cycle by the load inhibit line signals presented by each switch at the end of the previous cycle.

In this design, the switch passes at most one true access per cycle, and from that access creates a ghost message to send on the unused output. The routing control signals which outputs will be ghosts, and this signal becomes the ghost bit directly at the output.

The routing control also informs the combining control (H) whether inputs are to be combined and which packets to send to the combining information buffer at the switches RPC. Finally, the routing control sets control lines that inhibit loading of input and output latches and enable output from output latches and buffers.

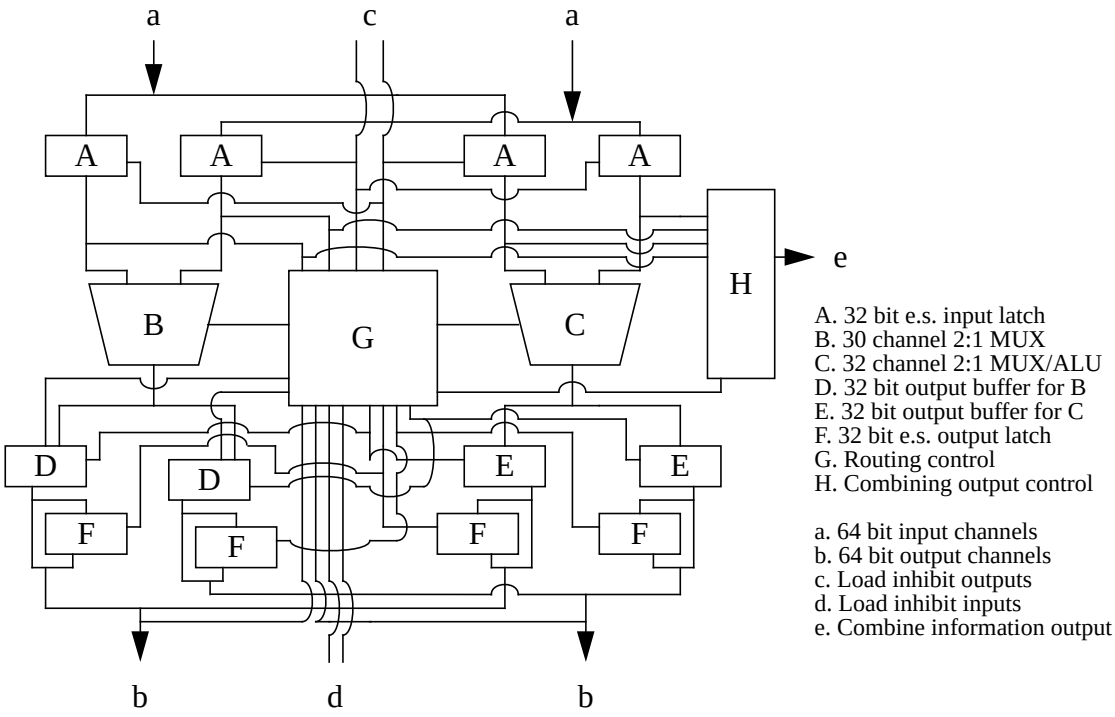


FIGURE 14. Forward Path Component architecture.

The operation of the forward path component during a normal cycle is as follows:

1. Input (and output) latches [As and Fs in figure 14] latch in new packets.
2. Routing control [G in figure 14] compares incoming packets, deciding which to route next. The comparison also indicates combinability.
3. Routing control disseminates decision data. This data tells B, C, and H which packets to propagate (and tells C whether to add). Routing control also sets output enable and load inhibit lines for combining control [H], latches, and output buffers.
4. B,C,H propagate packets. Routing control sets ghost bits for each output buffer.
5. End of cycle. Output latches latch in previous output (in case it is not accepted at the next stage). Input latches latch in new data, if open.

3.3.2.1 Routing Control

The routing control (figure 15) makes routing decisions based on the values of several inputs, both from the packets currently in both input and output latches, and from signals received from the next stage of the network. These inputs are:

Load inhibit input lines - These signals are sent by the switches at the next stage connected to the outputs. They indicate whether the previous cycle’s outputs were accepted (These outputs are held in the output latches [F in figure 14]).

Output ghost bits - These signals (from the output latches) indicate whether or not the previous cycle's outputs were ghost messages. If a ghost is not accepted, it does not need to be re-transmitted (its successor will be at least as informative).

Token bits - The token bits from each input indicate the inputs that are tokens, which are treated differently than other messages.

Comparator signals - The comparator operates on the MM address and the memory location as a local synchrony timestamp. Its output determines which input to route (the least timestamp) or, in the case of operations bound for the same memory location, whether to combine accesses.

Operation bits - The type of operation for each input informs the data MUX/adder (C in figure 14) how to proceed, and informs the combine output control (H in figure 14) which, if any, packets to send to the combining information buffer in the RPC.

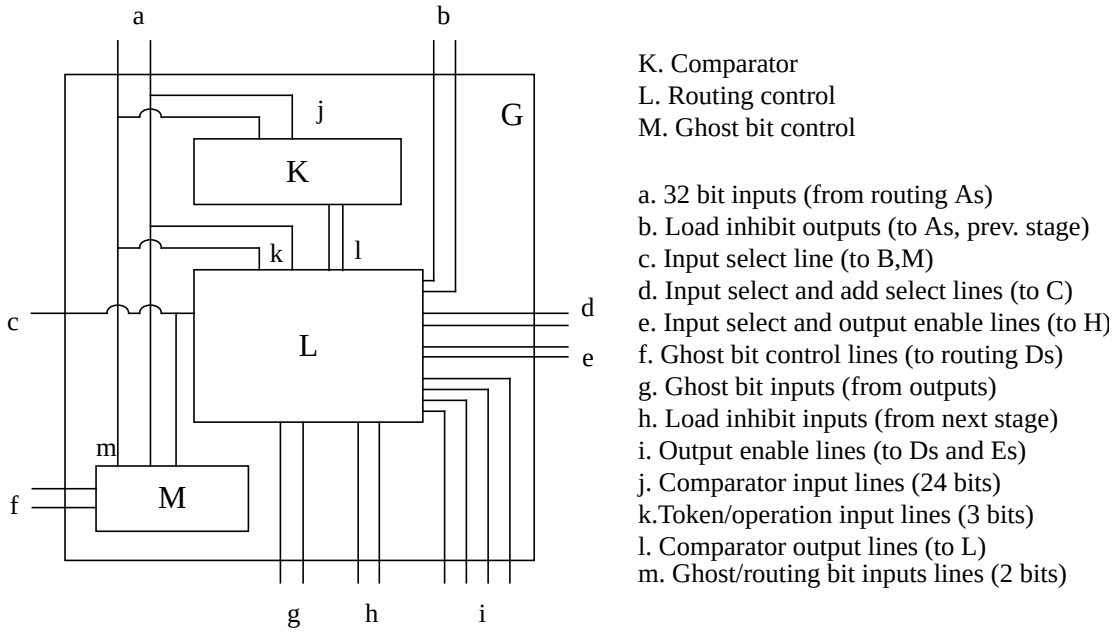


FIGURE 15. Routing control architecture

3.3.2.2 Routing Control Algorithm

Following is the algorithm by which the routing control makes its decision for a given cycle. The algorithm takes the form of a large case statement:

1. If one or more load inhibit input lines are ON, and at least one of the inhibited lines has its ghost bit OFF (check input from output latch): inhibit load on all input and output latches and signal a negative output enable to the combining control, which will be forwarded to the RPC. In this case, there has been no real change (i.e. transfer of true accesses) during the last cycle, so the switch must perform those operations again.

2. Otherwise (output latches open/buffers enabled for all further cases): if both token bits are ON, propagate either input (it will be sent on both outputs) and open both input latches (will flush both tokens).

3. Otherwise: if one token bit is ON, propagate the non-token, opening only its latch.

4. Otherwise: if the comparator output signals combinability (true on the Port1=-Port2 control line) and neither input is a ghost (or a token, checked for above), combine operations as described in table 2. This involves propagating one of the routing sub-packets and either one or a sum of the data sub-packets, signaling the combining control which sub-packets to send to the RPC, and determining the state of the ghost bit in the saved packet. Output is enabled from all output buffers (as opposed to output latches, which will have output enable/load inhibit signals OFF).

5. Otherwise: propagate the input representing the output of the comparator decision (Port1 < Port0? [This decision takes advantage of convexity for collisions -- Port0 always comes from a higher priority PE.]). If the other input's ghost bit is ON, open both latches (this will flush the ghost, allowing at least a new copy of it to be installed), otherwise, open only the latch of the chosen input.

3.3.2.3 Ghost bit control

The ghost bit control (M in figure 15) takes as input the routing decision of the routing control and the ghost and destination bits of the inputs. The routing decision from the routing control informs the ghost bit control which input will be forwarded. The dest bit indicates for which output the chosen message is meant. The ghost bit is set as follows:

Input ghost bit (the ghost bit for the chosen input) ON - set ghost bits of both output buffers to ON.

Input ghost bit OFF- set ghost bit of the output buffer to which the message is not destined to ON -- set the ghost bit of the buffer to which the message is destined to OFF.

The ghost bits of tokens will be set unnecessarily, but the ghost bit of a token message is unimportant.

3.3.3 Return Path Component Design

The design of the return path component generally follows the same pattern as the FPC. Differences in architecture and operation are due solely to the need for decombining of responses, rather than combining of accesses. Figure 16 presents the RPC architecture.

The main difference between the operation of the FPC and the RPC is that, in order to decombine responses, the RPC must employ a two-level system packet selection. The first level (C,D) selects the next returning packet, while the second level (D,E) decombines, if necessary.

The combine queue (A) is a simple FIFO queue that receives combining information from the switch's FPC. The routing control's decision is the same, with the following addition: after choosing the next response, the routing control checks to see if the response is to a combined access. The routing control then tells the second level of selection components which packet (or combination, for the data packets) to forward.

All latches remain closed in the decombining situation, and the second decombined response is emitted in the next cycle (latches must remain closed in order to save the information from which decombining is performed). This situation necessarily serializes the decombining operation, making it a two-cycle process. Section 3.3.6 discusses ways to improve this operation.

The normal cycle operation of the RPC is as follows:

1. Input (and output) latches [Bs and Hs in figure 16] latch in new packets.
2. Routing control [F in figure 16] compares incoming packets, deciding which to route next. Routing control also checks whether next response was combined at this switch.
3. Routing control disseminates first- and second-level decision data. First level data informs the input MUXs [C,D] which packets to propagate. Second level data informs the routing MUX [D] which packet (between chosen and saved combined) to propagate, and tells the recombine ALU [E] how to proceed (input select, add 0, add negative, add). Routing control also sets load inhibit and output enable signals, and pops enable signal for combining queue [A].
4. D, E propagate packets. Routing control sets ghost bits for each output buffer. B propagates routing control's decision data as a new routing bit for this stage (MM id information).
5. End of cycle. Output latches latch in previous output (in case it is not accepted at the next stage). Input latches latch in new data, if open. Combine queue accepts new data and pops front, if necessary.

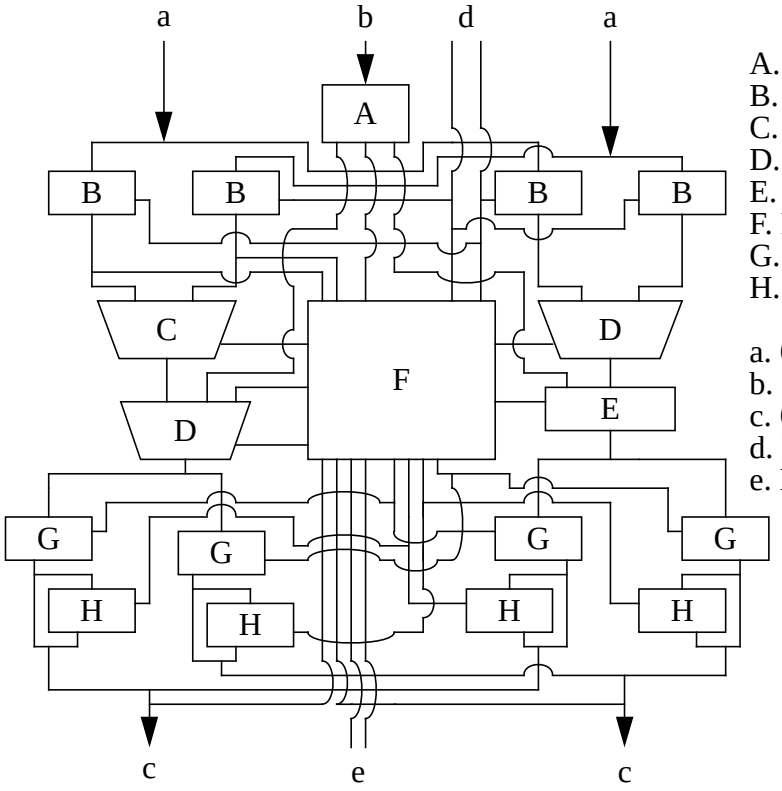


FIGURE 16. Return Path Component architecture

3.3.4 Combining System

The operation of the combining system in this design was chosen for its simplicity, keeping bandwidth considerations in mind. Section 3.3.6 discusses more complex extensions to the combining systems.

Local synchrony ensures that responses returning to a given switch from global memory arrive in the same order in which their respective responses left the switch. Thus, the design of the combining information queue is merely a FIFO queue. In our implementation, we assume that this queue is long enough that it will never overflow. This is possible because there is a maximum number of accesses which, having passed through a given switch, can be outstanding from that switch (have not returned). The equation for CMAX is given in section 3.2.1.7. For this design we have, for the maximum queue length (only those switches in the stage furthest away from the MMs need queues of this length):

$$C_{MAX} = nS + oM + t = 5*(2^6-2) + 1*(2^6) + 2 = 376 \quad \text{(EQ 18)}$$

Each switch (S) could buffer two combined accesses and three combined responses (n). Each MM (M) could buffer one combined access, assuming MMs have one input latch and no output latches (o). The given switch can buffer only combined responses on its two RPC response inputs (t).

The combining information queue holds an entry for each access that leaves the switch, whether it is combined or not. On return, therefore, the RPC need only check the combine queue for each response and decombine if necessary. Since no token is ever combined, we use the token bit to indicate combining in the saved information, and the RPC routing control checks this bit. If an access is not combined, the FPC might send a copy of it to the combining queue for error checking, or the FPC might send nothing at all. Because the FPC will indicate when an access has left, the RPC will reserve a space in the FIFO for that access.

Only true accesses are combined. The routing control ensures that two tokens or two ghost messages cannot be combined, nor can a token be combined with a true access or ghost message. The MM address of a ghost message is guaranteed to be unreachable from any switch where the ghost is sent. No ghost's 'timestamp' will match a true access compared to it, so a ghost message will never be combined with a true access. The routing control also blocks output to the combine information queue when passing a token or a ghost message. The combine information queue records only the transfer of true accesses.

As we implement it here, local synchrony guarantees that all combinable accesses within a given ltu will be combined. Combinable accesses are guaranteed to meet and be combined at the earliest possible point within the network. This is necessary because if two combinable accesses were to meet and not be combined, then later combining might compromise the order of execution of the accesses.

3.3.5 Packaging

The implementation of the switch presented here uses current off-the-shelf components. We are able to package an entire switch, including FPC, RPC, and combine information queue,

using 6 chips. Two of these consist mainly of FIFOs which implement the combine information queue. The other four are Actel A1280 field programmable gate arrays, two each for the FPC and RPC. The A1280 has 140 pins of user I/O and contains 1200 logic gates. In each component, one A1280 handles the routing information (including the routing control) and one handles the data information.

Since the logic involved in the switch is relatively simple, the chip count is limited by the necessary pin counts for I/O. We simplify the design in several places to achieve pin counts low enough to allow implementation with the above-mentioned components. Our main simplification is to implement the switch so that the data being broadcast on either output during a given cycle is the same, with the exception of a single bit. Thus, we can use single buffer/latch combinations to implement both output ports, rather than separate sets of latches for each port. The exception is the ghost bit, which must have separate buffer/latches for each port. In this way, for example, we limit the output from each sub-component of the FPC to 33 bits rather than 64 bits, which allows the pins necessary for the combining information port to be allocated within each sub-component.

Table 4: Pin and logic gate counts for various sub-components

Sub-component	Pin Count (I/O)	Logic Gate Count
FPC Routing	136 (66/70)	~500
FPC Data	132 (68/64)	~250
RPC Routing	138(100/38)	~600
RPC Data	134(102/32)	~600

The FPC as a whole has 130 bit-inputs and 132 bit-outputs (this number takes into account the multiplexing suggested above). The RPC has 195 bit-inputs and 67 bit-outputs. The combining information queue has 66 bit-inputs and 64 bit outputs. The entire switch has 260 bit-inputs and 134 bit outputs. Table 4 shows pin counts and approximate gate counts.

3.3.6 Design Extensions

We present here several methods for extending this design to more varied and useful architectures and situations.

No PE Address—It is possible to eliminate the need for each access to carry its PE address. This will free up those bits for other purposes, like expanding the number of PE/MMs, the size of global memory, or the number of possible operations. In [DiK92], the authors present the design for a switch that replaces the routing bit of the MM address with a bit designating the input port that received the access. When the access reaches global memory, it has a full address for the PE to which it must return.

This approach is feasible in our design, but somewhat complicated because the MM address is necessarily part of the ‘timestamp’ for each message, and the timestamp of a ghost message might change relative to other accesses using this protocol. The MM address and memory location of a ghost message must be changed to all ones or all zeros, depending on which port is sending it out. This does not affect the operation of the system, however, the routing control must explicitly forbid ghost messages from being combined, since this situation may now occur.

Non-Sequential Combining—One bottleneck apparent in our design is that decombining responses is a sequential operation that requires two switch cycles to perform. In fact, by adding more complexity to the RPC we could reduce the decombining operation to a single switch cycle. This would involve splitting the output ports, and providing each with a separate recombining ALU. Although we can achieve the added logic using current components, this would increase the pin count beyond the limits of the A1280, thus creating a split that would separate the RPC data component into at least two and possibly three sub-components.

Space Ahead—Because there is a limit on the number of outstanding accesses that a PE may have during a given ltu, we may be able to eliminate the need to block inputs to switches by using FIFO queues on output ports. Although this will greatly expand the necessary size of the combining information queue, it may be possible to use another combining protocol in tandem with this idea that limits its size.

There is only one message cell per input to the switch. Thus, a token sent by a given switch cannot reach its next destination until, at a minimum, that switch has passed the previous

token to output. At this point, there are only a finite number of true accesses that can pass through the switch before it receives a token on its other input, and the token is passed.

In fact, if the network is operating at capacity (the PEs never send out a ghost message), the number of true accesses and ghost messages that will pass through the switch (N) is no greater than the maximum number of true accesses that might pass through the switch during this ltu. We can use this information to implement FIFO queues on output ports that can buffer a number of accesses equal to N. In this way (again, given that PEs never send out ghost messages), we guarantee that a switch will be able to output during each cycle. This eliminates the need for blocking, along with its added complexity.

The FIFOs generally must be long, however (for this design, assuming that a PE can issue 8 accesses per ltu, the FIFO would have to be able to buffer 256 messages [2^5*8]). The literature [Won86] suggests that buffering more than 2 or 3 accesses at a port is counterproductive.

Scalability—This design scales quite well to larger networks and applications. Adding information to the packet in 32-bit increments allows for 64-bit or larger data words, or larger networks (an addition of another 32 bit sub-packet would allow 2^{20} PE/MMs, with an additional 60 operations, for example). Because the switch is designed to expand ‘sideways,’ enlarging the packet size would not generally affect the cycle time of the switch, except that the necessarily larger comparators and adders would have longer cycle times. Any design would face this same problem in scaling.

Asynchronous Processing—This design can be modified easily for asynchronous processing. The FPC or RPC cycles whenever it has input on both of its ports. A handshake protocol can be used to indicate when full input has been received.

Another extension to either synchronous or asynchronous processing means that the switches need not send a message on each output during each cycle if we were to add memory to the switch itself. This memory would retain the timestamp of the last message sent from a given

input or output. The switch would then need to send a message only when it was able to update the latest timestamp. This would require significant added complexity.

3.4 Minimum Requirements for Deadlock Freedom

In this section we show that, in the case of the Ultracomputer and other equivalent architectures, it is possible to achieve deadlock freedom in a local synchrony implementation without the use of ghost messages. We make the same assumptions about the system and implementation of local synchrony, but assume also that the system does not generate and distribute ghost messages. The only difference in operation, other than the lack of ghost message generation, is in the handling of tokens.

Deadlock freedom without ghost messages relies on general information about the timestamps of messages that might arrive on a given input to a node. This *inherent knowledge*, which can be found in the makeup of the logical timestamps applied to accesses, allows a node in some cases, to feed a token on an output even when tokens have not yet been received on either input.

We begin by considering the DDG as defined previously in this chapter. At the center of any DDG is a deadlock circularity. To break deadlock, we must be able to break this circularity at any point. Consider a deadlock circularity that is part of the DDG. Lemma 3.2 states that any leftmost node in the deadlock circularity must be a type L node.

Lemma 3.5: There are at least two type L nodes in the deadlock circularity.

Proof: By lemma 3.2, there is at least one type L node in the circularity. Assume that there is only one, N_0. By definition of the network topology, the set of nodes to the right of N_0 that are serviced by N_0 is divided into two disjoint subsets, each serviced by one of the outputs. There is no connection between any node in one subset and any node in the other. Thus the circularity that causes deadlock cannot exist unless it passes through another node, N_k, at or left of the stage in which N_0

resides. Since N_0 is a leftmost node, N_k must be in the same stage, and, by lemma 3.2, must be a type L node. Figure 17 illustrates this argument. ■

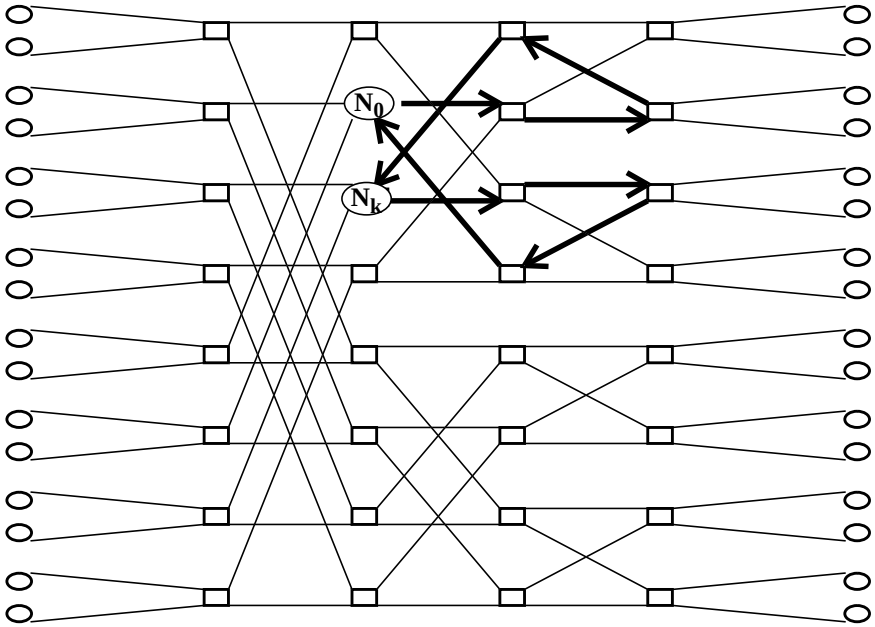


FIGURE 17. An example of deadlock in a banyan network. N_k must be in the same stage as N_0.

Corollary to lemma 3.5: There are at least two leftmost type L nodes in the deadlock circularity.

Proof: Lemma 3.5 shows that at least two type L nodes exist in any deadlock circularity. Let us choose N_0 and N_k such that a subsequence of the dependencies in the deadlock circularity is $\{N_0,..., N_k\}$, and includes no other type L node, where N_0 --> N_1 is a type 2 dependency. The definition of the circularity makes this choice possible. We simply inspect the circularity and choose two leftmost nodes in sequence. ■

> **Lemma 3.6:** If N_0 and N_k are as described above, then N_{k-1} --> N_k is a type 1 dependency.

Proof: Assume the converse. Then N_{k-1} --> N_k is a type 2 dependency, which means that N_{k-1} is left of N_k--by definition of N_k a contradiction. ■

Corollary to lemma 3.6: Let N_l be any other type L node that is part of the deadlock circularity. N_{l-1} -> N_l is a type 1 dependency.

Proof: Let $LAST_i$ be the timestamp of the last access chosen for output by node i. $LAST_i$ includes accesses chosen but not sent, due to blocking ahead. Let $LAST_{i,j}$ be the timestamp of the last access output or chosen for output on line {i, j}. Let a type L' node be defined as a node in the

deadlock circularity with a type one dependency on one input and a type 2 dependency on the other input, where both dependencies are part of the deadlock circularity (and thus none of the node's outputs are part of the circularity). We can show the existence of type L' nodes to be similar to the existence of type L nodes. Since there is no type L node between N_0 and N_k in the deadlock circularity, only one type L' node, N_j, exists between them, and $\{N_0,..., N_j\}$ consists entirely of type 1 dependencies, while $\{N_j,..., N_k\}$ consists entirely of type 2 dependencies. ■

> **Lemma 3.7:** If N_0, N_j, and N_k in the deadlock circularity are as described above, then logical time unit $LAST_m$.ltu <= $LAST_0$.ltu, for any m = {1,k}.

Proof: By virtue of the type 1 dependency on N_0-->N_1, $LAST_1 < LAST_0$ and thus $LAST_1$.ltu <= $LAST_0$.ltu. This same reasoning is applied to N_1-->N_2 and so on through N_{j-1}-->N_j, with the result that $LAST_j < LAST_0$ and thus $LAST_j$.ltu <= $LAST_0$.ltu. By virtue of the type 2 dependency on N_k-->N_{k-1}, $LAST_{k,k-1} = LAST_{k-1}$. Again, this same reasoning is applied to the succeeding nodes between N_k and N_j, with the result that $LAST_j = LAST_{k,k-1}$ and thus $LAST_j$.ltu = $LAST_{k,k-1}$.ltu, and, by substitution, $LAST_{k,k-1}$.ltu <= $LAST_0$.ltu. $LAST_k$.ltu = $LAST_{k,k-1}$.ltu, because a token is sent on each output when the ltu changes, and thus $LAST_k$.ltu <= $LAST_0$.ltu. ■

> **Corollary to lemma 3.7:** $LAST_m$ <= $LAST_0$, for any m = {1,k-1}.

> **Lemma 3.8:** For any type L nodes N_m and N_n in a deadlock circularity, $LAST_m$.ltu = $LAST_n$.ltu.

Proof: Construct the sequence of type L nodes within the circularity from any type L node to itself - $\{N_0, N_1, N_2,...,N_p, N_0\}$. By lemma 5.3, $LAST_0$.ltu >= $LAST_1$.ltu >= $LAST_2$.ltu >=...>= $LAST_p$.ltu >= $LAST_0$.ltu. $LAST_0$.ltu = $LAST_0$.ltu, and thus $LAST_0$.ltu = $LAST_1$.ltu = $LAST_2$.ltu =...= $LAST_p$.ltu. ■

We now make some observations about the network topology and the logical timestamping system used by local synchrony. There are two basic variations to the timestamping system. In either system, the ltu is the first element of the timestamp. When FIFO combining is part of the implementation, the memory address of the access is the second element of the timestamp. When FIFO combining is not in force, the timestamp may be simplified, with the PE id being the second component of the timestamp.

The network architecture presented in this implementation has the characteristic of convexity, both for PEs and MMs. In other words, it is possible to number PEs and MMs to ensure that

the inputs of any switch node come from distinct, contiguous sets of PEs (by id) and the outputs cover distinct, contiguous sets of MMs.

When sorting accesses in the same logical time unit by PE id, we see that a switch will send all the accesses arriving on one input for that ltu before sending any arriving on the other input. This is because all the accesses arriving on the line in question are from PEs with lower ids than the PEs sending all the accesses arriving on the other input. In effect, one input has higher priority than the other at all times during any ltu. By the same logic, when sorting accesses in the same ltu by memory location, we see that a switch will send all the accesses bound on one output during that ltu before sending any bound on the other output.

This may not be the most efficient method of implementing the timestamping system. For example, hashing of addresses or ids might generate a non-serialized use pattern for communication lines. In the absence of a ghost messaging system, however, implementing the timestamping system in this way guarantees deadlock freedom.

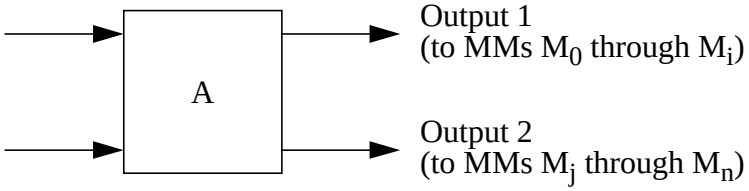


FIGURE 18. Node A as described above. Note: {0, i} < {j, n}

Consider node A in figure 18. Node A is passing accesses based first on ltu, then on the address of the variable accessed. Node A's output 1 covers MMs M_0 through M_i, while output 2 covers MMs M_j through M_n. The MM numbering and the timestamping system has been designed so that {0,i} have higher priority than {j, n}. Thus, during the processing of any given ltu, once node A passes the first access (during that ltu) along output 2, it may pass a token on output 1, because it will pass no more accesses on output 1 during the current ltu.

We restrict the following arguments to the case of FIFO combining, where accesses are sorted first by ltu, then by address of the variable accessed.

> **Lemma 3.9:** Given a deadlock dependence circularity, there exists some type L node, N_k, which can emit a token on its empty output.

Proof: Assume the converse. Choose a leftmost node, N_j, in the circularity. By lemma 3.2, this node is a type L node. By assumption, N_j has a type 2 dependency on its high priority output. By the corollary to lemma 3.5, at least one other leftmost type L node, N_k, exists in the circularity. Choose N_k such that no other leftmost type L node is in the subsequence $\{N_j,...,N_k\}$ of the circularity. Since N_j and N_k are at the same stage of the network, and by virtue of the circularity itself, outputs {j, j+1} and {k, k-1} each cover the same subset of MMs. By assumption, this set of MMs is high priority compared to the set covered by the other output of N_j and N_k. N_k does not have a type 2 dependency on output {k, k-1}, and thus must have a type 2 dependency on its other output, which is the low priority output. The high priority output of N_k is thus empty, and a token can be emitted on that output, since no other accesses will be sent on that output during the current ltu. ■

A key result follows:

> **Theorem 3.2:** Inherent knowledge is sufficient to break any possible deadlock in a local synchrony implementation on a Banyan-type equidistant network.

Proof: We assume without loss of generality that all other dependence in the DDG is related to the deadlock circularity; i.e., that any type1 dependency comes from attempts to pass information through nodes in the circularity, and any type 2 dependency comes from waiting for information to be passed through nodes in the circularity. Lemma 3.9 states that there exists some type L node, N_k, within the deadlock circularity, that can emit a token on its empty output (the one with a type 2 dependency). Choose type L node N_0 such that there is no other type L node and only one type L' node, N_j, between N_0 and N_k (lemma 3.6). By lemma 3.8 and the emission of this token, $LAST_{k,k-1}.ltu = LAST_0.ltu + 1$. This contradicts the corollary to lemma 3.7. Node N_{k-1} now has a token on one input and, by assumption, has input on its other input, so it may progress by sending accesses from the non-token input until a token arrives. At this point, node N_{k-1} may send a token on both outputs. This process is repeated at each preceding node until a token is passed to node N_j on its empty input. N_j will then be able to progress by sending the accesses that have been blocked in type 1 dependencies. Progress continues at each preceding node, including N_0, until the node is able to send tokens on its outputs as well. Each preceding pair of type L nodes is able to progress to the next ltu in sequence. Since this is a circularity, at some point the progress chain will reach back to N_k, which will progress to the true ltu change (of course, N_k will not release another token on {k, k-1}). Thus, the emission of one token, guaranteed by lemma 3.9, will cause the entire deadlock circularity to progress until at least each type L node has increased its ltu by 1. When this has occurred, another token can be released, and so on. ■

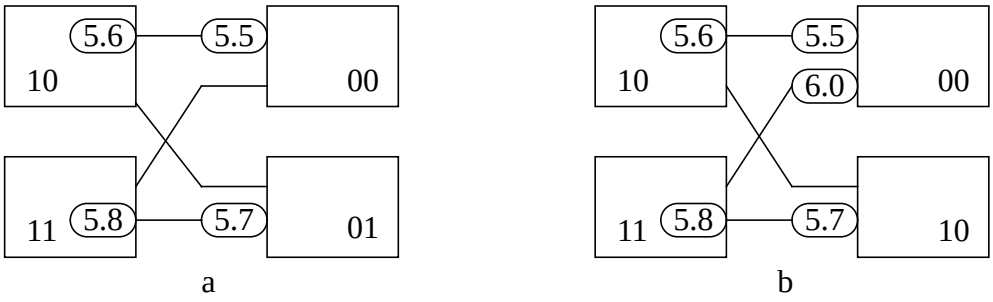


FIGURE 19. a) Deadlock b) Inherent knowledge solution

We give a simple example of this process, using a two-stage network as in figure 19 above. SE 10 sends an access with timestamp 5.5 to SE 00, then blocks trying to send its next access, with timestamp 5.6, to SE 00. SE 11 sends an access with timestamp 5.7 to SE 01, then blocks trying to send its next access, with timestamp 5.8, to SE 01. Deadlock occurs because no SE can progress. This situation is depicted in figure 19a. Because SE 11 has sent an access to SE 01 during ltu 5, however, it can determine that it will not send any more accesses to SE 00 during this ltu. It can thus send a token (timestamp = 6.0) to SE 00. This breaks the deadlock, as SE 00 is then able to progress (figure 19b). The fact that a token has been received by SE00 guarantees that SE 10 will progress until it reached the next ltu, and thus the deadlock is broken.

3.5 Summary

In this chapter we have laid the foundation for later work by providing a motivation for investigating implementations of local synchrony. We first made a simple argument showing that local synchrony can outperform locking systems given certain assumptions, that we will show later are reasonable.

Then we presented an implementation of local synchrony and showed that it is feasible and deadlock-free. Our plan covered most aspects of implementation, including operative algorithms, transfer and routing mechanisms, buffer management and flow control, and the combining system.

Also we presented a low-level hardware design for a switch to be used in the implementation, and discussion of the minimum requirements for deadlock-freedom in convex local syn-

chrony implementations. In the next chapter we show that local synchrony can be implemented efficiently on any shared memory architecture.

4.0 A General Implementation Technique

In this chapter, we show implementability for local synchrony in a correct and deadlock-free way on any shared memory architecture. Our implementation scheme preserves any locality (defined in chapter 3) present in the physical architecture, and provides greater opportunities for parallelism and less overhead than other approaches.

Our presentation proceeds as follows: in section 4.1, we define correctness criteria for local synchrony implementations. In section 4.2, we discuss two different approaches to implementation of local synchrony, based on the logical service time of a switching element (the logical time taken in a single step in the network). Each approach has different problems in implementation. We discuss these problems at length, as well as our reasons for choosing one over the other.

In section 4.3, we define a class of directed graphs, which we call LS graphs, and show how an LS graph may be used to represent the communication interconnections of a given shared memory architecture. In section 4.4, we present a local synchrony implementation for such architectures.

In section 4.5, we prove that the implementation of local synchrony on the LS architecture of section 4.4 is correct and deadlock-free. Finally, in section 4.6, we present an algorithm which, for any candidate shared-memory architecture, will generate an LS graph which is mappable onto the candidate architecture. This algorithm may be used in conjunction with virtual channels [Dal92] to provide a correct and deadlock-free implementation of local synchrony on any shared-memory architecture without compromising any locality (defined earlier as the proximity of all or part of shared memory to a given PE) exhibited by that architecture.

Section 4.7 gives several examples of such translations for interesting architectures. Section 4.8 discusses the operation of virtual switching elements, which are used with virtual channels for local synchrony implementation. Section 4.9 discusses memory requirements for implementation and compatibility with different routing schemes.

4.1 Correctness of a Local Synchrony Implementation

The notion of correctness of a local synchrony implementation is directly related to the goals of local synchrony itself. By abstracting the operation of local synchrony to a logical level, we may say that in executing a given computation, PEs release a sequence of messages, bound for various MMs, with strictly decreasing (among messages from the same PE) and strictly unique (among all messages) priority. The priority system is based on a possible physical ordering of events by execution time in a general system, and thus messages released later in the stream have strictly lower priority than those released earlier.

For a given distributed computation, the goal of a local synchrony implementation is to complete the computation, and ensure that the MMs consume messages in strict order by decreasing priority. Because the goals of local synchrony are met when the MMs consume messages we do not consider here the handling of responses from MMs. Responses may be handled normally, or in a locally synchronous manner. We assume, without loss of generality, that the responses are handled by a separate network within the architecture.

We also assume correct operation of the PEs and MMs. PEs are failure-free and produce a stream of accesses, bound for the MMs and ordered by a given timestamping or priority system which guarantees that each message has a unique priority. MMs are failure-free, have a single input, and consume messages as they arrive.

In order to recognize the completion of a computation, we assume that PEs release a sequence of messages with strictly decreasing and unique priorities, followed by an End-of-Computation (EOC) message which marks the end of that PEs message stream for this computation. The EOC message has lower priority than any computation message released by any PE and has equal priority to the EOC message of any other PE.

In a correct implementation of local synchrony, any deadlock-free computation will execute deadlock-free. A computation will also progress toward a termination state. In the termination state, all computation and EOC messages have been sent by PEs and have been consumed by MMs

in strictly decreasing priority order. The last message sent on all communication lines is an EOC message.

The three elements that determine correctness for a local synchrony implementation, then, are deadlock freedom, the correct order of consumption of messages by the MMs, and a guarantee of progress to the termination state. We will show in section 4.5 that the implementation and LS architectures we describe meet the requirements of correctness and deadlock freedom. We will then show how an LS graph can be generated for any candidate architecture. Such an LS graph can be mapped onto the candidate architectures using virtual channels [Dal92], allowing correct and deadlock-free implementation of local synchrony on the candidate.

In the next section, we outline two approaches to local synchrony implementation, based on the logical time taken by a message in negotiating a single network step. A correct implementation is possible by using the isotach approach (a message's timestamp is incremented by a single ltu at each step), or a 'time propagate' approach (a message has the same timestamp at all times). We discuss each approach and their good and bad qualities as possible implementations for local synchrony. We choose the approach which best suits the goals of local synchrony.

4.2 Approaches to Local Synchrony Implementation

There are two approaches to implementing local synchrony. We shall see that each of the approaches presented has flaws: one because the system performance is inconsistent with the goals of local synchrony; and the other because directed loops within an architecture can cause deadlock unless certain buffer and switch operation requirements, which we feel are unrealistic, are enforced. We choose the approach that best fits the goals of local synchrony, and then show that it can be implemented efficiently and correctly.

In local synchrony, sources (PEs) generate messages with a unique logical timestamp and are required to release messages in timestamp order. Other nodes are also required to release messages in timestamp order.

As logical systems, local synchrony and the Chandy/Misra logical simulation [ChM79] are quite similar. While the method of message generation is different, the results are the same: sources generate and send streams of messages in a timestamp order, and all other nodes are required to process messages conservatively by timestamp (while Chandy and Misra never mention conservatism, it is implied by the algorithms presented). Once a message has been queued for output by its source, it is treated the same in either system.

Chandy and Misra present a convincing proof that their system is correct and deadlock free, yet in fact they neglected to consider some possibilities which are admissible under their requirements but can still deadlock. For example, differences in the service times of nodes in a fork and join loop can cause deadlock unless finite but unbounded buffer cells are in use. Figure 20 depicts such a situation. Consider this to be a queueing network, where jobs arrive at a node, are processed in some finite time called the service time, and then are sent on through the network.

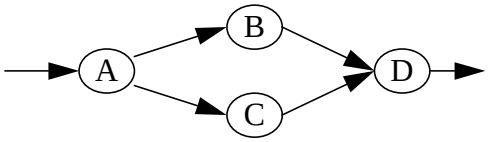


FIGURE 20. Queueing network which could deadlock.

Let the service time at A, B, and D be 1 and the service time at C be 1000, and let each node be able to buffer only 1 job (message). Let jobs be of the form [t, p], where t is the send time from the previous node, and p is the path information that declares which branch the job will take at A. Suppose the input to A is {[1,C],[2,C],[3,C],[4,C],[5,B],...}. Processing will proceed as follows:

1. Node A receives [1,C] and passes [2,C] to C and [2,NULL] to B.
2. Node B receives [2,NULL] and passes [3,NULL] to D. Node C receives [2,C] and passes [1002,C] to node D. Node A repeats step 1 with [2,C].
3. Node D receives [3,NULL] and [1002,C] and sends [4,NULL]. Node B receives [3,NULL] and sends [4,NULL] to node D. Node C receives [3,C] and blocks trying to send [1003,C] to node D. Node A receives [3,C] and blocks trying to send [4,C] to node C, either after or before sending [4,NULL] to node B.
4. If [4,NULL] reaches node B, it is sent on to node D and onward.

At this point, the network is deadlocked. Node D cannot conservatively pass [1002,C], because a job with smaller timestamp could arrive from node B. In fact, such a job ([5, B]) exists, but will not reach node A until node D sends [1002,C]. The only solution to this problem is to provide node D with a buffer queue which is long enough to provide protection from this type of deadlock (in this case, the queue would have to have 998 job slots). This solution is sub-optimal due to low utilization of buffer space and the need for network analysis to generate the queue sizes. In general, finite but unbounded buffer cells would be necessary to guarantee correct and deadlock-free operation.

Because the service time of nodes is uniform over all nodes in our approaches to implementation of local synchrony, this type of deadlock cannot occur in a local synchrony implementation. However, we shall see that the Chandy/Misra system can still deadlock under conditions and implementations which are consistent with the goals of local synchrony.

4.2.1 The Time Propagate Approach

In the time propagate approach, an intermediate node receives a message and passes it on in the same logical instant, i.e. with the same execution priority or logical timestamp. This approach is the most intuitive and simple way to enforce the ordering constraints of local synchrony, which guarantee sequential consistency, version consistency, and atomicity of global memory accesses. As we shall see, however, the time propagate approach is counter to the stated goal of local decision making, which is central to the philosophy of local synchrony itself.

Consider the architecture depicted in figure 21. Input streams of messages are provided. Nodes A, B and C are intermediate nodes in the system. Each conservatively chooses the next lowest priority on its inputs (input streams are guaranteed to arrive in priority order) and passes that message on. Nodes D and E are sink nodes which consume messages as they arrive (in priority order).

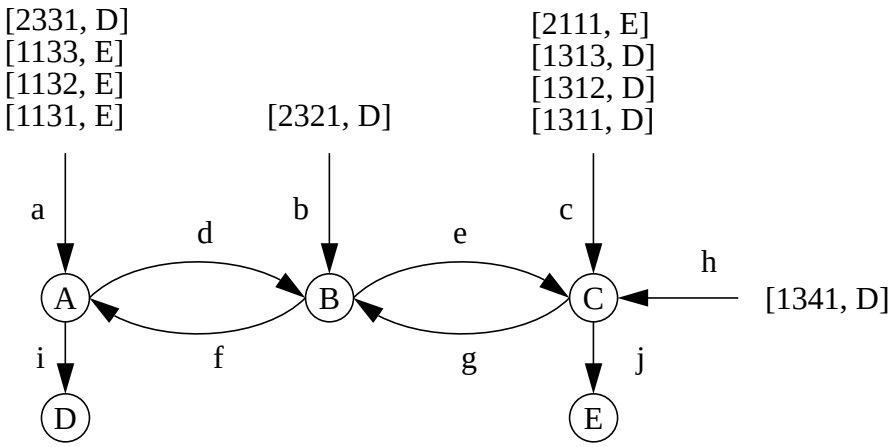


FIGURE 21. Example architecture.

For situations involving directed loops, Chandy and Misra describe some arcs as predictable and require that any directed loop have at least one predictable arc. In the local synchrony implementations, arcs are not generally predictable, but knowledge of the message priority scheme allows intermediate nodes to know what the next lowest possible priority on an incoming line may be. This guarantees that at any time there is at least one predictable arc in a directed loop, even though that arc may not be predictable in general (such a line is only predictable when the sender can generate the next possible timestamp).

In the following example we use logical timestamp information to generate a unique four-digit priority for each message. The third digit of the priority, which represents the source ID of the message, is the key which allows predictability for at least one arc in all circumstances. The other digits of the priority are created from other logical timestamp information: the first digit represents the logical time unit, the second represents destination, and the fourth represents rank. Assume for simplicity that at most nine messages may be ranked by a given source in any given ltu. For example, the source which feeds line c in figure 2 sends messages which always have a 1 in the third digit of their timestamp. This is the only source in the system which uses this key.

We will now consider the operation of the network. Let intermediate nodes be defined as non-source, non-sink nodes in the network. Initially, each intermediate node (A, B, C) will send a message [1000, NULL] on each of its outputs. Processing will then continue as follows:

1. A and B now have [1000, NULL] as their priority input. Having sent the same message already, they block. Node C has simulated up to priority1000, and the next possible priorities are 1111-1119, which can only arrive on input c, so, since the input on c is greater than 1119, node C can send [1119, NULL] on its outputs.
2. Node B now has [1119, NULL] as its priority input. Since the next possible priority is 1121-1129, which can only arrive on input b, and the actual input on b is greater, node B can send [1129,NULL] on its outputs.
3. Node C receives [1129,NULL], but cannot forward any message because it can make no prediction about the next priority to arrive. Node A receives [1129, NULL]. Since the next possible priority is 1131, which is on input a, node A can send [1131,E] on output d, and [1131,NULL] on output i.
4. Node A may continue to process [1132, E] and [1133, E], which progress across the network to node E. Node A then sends [1139, NULL], which is the best prediction it can make.
5. When Node C receives [1139, NULL], it is able to make a prediction, but it must limit itself to sending [1211, NULL], rather than [1311, D], which is the next message to be processed. This procedure continues as nodes B and A make their predictions which finally result in [1239, NULL] reaching node C, which allows it to finally send [1131, D].

While this process does execute correctly and in a deadlock-free way, it is counter to the philosophy and goals of local synchrony in two ways. First and foremost, the reader will note that in the directed loop (C, B, A, B, C), all nodes must communicate to make one message transfer decision. In many current parallel architectures, such as a hypercube or mesh, the node interconnection scheme is such that this type of procedure amounts to global communication. The philosophy of local synchrony calls for local decision making only, without global synchronization.

Another flaw is that the system spends much extra processing and time raising the predicted priority or logical time of each node. In general, the priority may only be raised by a minimum amount at each node, even if that node holds the next message to be processed (see step 5 above). We would like a local synchrony implementation to allow a node to perform useful work if it has useful work to do at any time. For these reasons, we deem this type of implementation suboptimal for local synchrony.

4.2.2 The Time Increment Approach

The time increment approach overcomes the flaws in the time propagate approach by introducing a distance component into the processing of messages. The time increment approach implements an isotach system by adding a constant to the priority of a message each time the message takes a step in the network. By using a positive, uniform, constant service-time, we enforce the velocity invariant for isotach systems described in chapter 2.

Note that messages from a given source (PE) are still consumed in priority order at any given MM, although the order of consumption for messages to the same sink (MM) from different sources is altered depending on the distance from source to sink. Because the sources assign priorities to messages, it is possible to modify the assignment scheme for priorities in such a way that messages will reach network outputs in the same order generated by the time propagate approach. This is accomplished by subtracting the distance a message must travel from the ltu in which it is to be executed. The service time which is added to the priority each step is equivalent to one ltu.

The problem that arises when using this approach is that, much like our earlier example, deadlock is now possible unless large buffers are employed. While the size of these queues is finite and calculable, we will later show that a better solution exists.

Let us consider the example presented in section 4.2.1 (Figure 21). Rather than a service time of 0, nodes now have a service time of 1000, which is added to the priority of an incoming message when the message is sent. Message processing is much more efficiently implemented, as we see below. Initially, each node sends the initialization message [1000, NULL] on each of its outputs (this message will arrive at the front of the input streams as well). Processing continues as follows:

1. Each of nodes A, B, and C now has [1000, NULL] on each input, so [2000, NULL] can be sent on each output.
2. Node B has [2000, NULL] as its priority input, and sends [3000,NULL] on each output. Node A chooses [1131, E] and sends [2131, E] on output d and [2131, NULL] on i. Node C chooses [1311, D] and sends [2311, D] on output g and [2311, NULL] on j.

3. Node B has received [2131, E] and [2311, D], and can choose [2131, E] and send [3131, E] on output e, and [3131, NULL] on f (each of these messages will overwrite the ghost message [3000, NULL] at the inputs to nodes A and C).
4. Node A now sends [2132, E] on output d. Node C is blocked waiting to send [2312, D] on g.
5. Deadlock now occurs. Nodes B and C are each waiting for the other to proceed. The state of the inputs for each node is shown below.

Node A — a = [1133, E], f = [3131, NULL] — A cannot pass input [1133, E] because it is blocked by traffic ahead.

Node B — d = [2132, E], b = [2321, D], g = [2311, D] — B cannot pass [2132, E] because it is blocked by traffic ahead, and cannot pass [2311, D] because it cannot conservatively guarantee that [2311, D] will be the next to be passed on output f. Even if it could make this guarantee (A could guarantee that it will not pass messages destined for node D through node B), once passed, A could not pass [3311, D] further along output i, as one can see that [2331, D] will appear on input a at a later time, and should be sent before [3311, D].

Node C — e = [3131, E], c = [1312, D], h = [1341, D] — C cannot pass [1312, D], because it is blocked by traffic ahead, and cannot pass [1341, D] for the same reason and because it should only be emitted after [1312, D]. C cannot pass [3131, E] because a message could arrive on either other input that should precede it ([2111, E], which will arrive later on input c, is an example of such a message).

Deadlock occurs because the service time for nodes in a directed loop is greater than the difference in timestamp values for messages entering the loop on multiple inputs. This creates a situation where two (or more) nodes may be waiting on each other, but cannot communicate to break the deadlock because of the conservative processing constraints in the system. This situation was overlooked in the Chandy/Misra logical simulation system [ChM79].

While this system can deadlock, it exhibits many traits which make it attractive as a possible implementation of local synchrony. Switching elements are able to operate independently, making decisions based only on local communication. No global communication is required to make progress at any time. Finally, there is no extra work required in advancing the logical clocks of switches up to the point where useful work can occur. Because processing decisions are local, switches can immediately jump to their next ‘useful’ clock value to continue processing.

The deadlock situation only occurs when switching elements arbitrate between messages without reference to the distance those messages have to travel. It cannot arise in equidistant implementations (see chapter 3), because any two messages being compared always have the same

distance to travel. In the general (non-equidistant) implementation, however, messages must travel differing distances to reach their destinations.

There are two solutions to this problem. Using the Chandy/Misra approach would involve reserving enough buffer space to handle all messages that might arrive on a given input during a given ltu, and rewriting the algorithms to allow continual acceptance of inputs even while a switch is blocked waiting to output. This approach is wasteful of buffer space. Our approach solves this problem by eliminating the possibility of comparing dissimilar priorities (the priorities of messages with differing distances to travel in the network), without making messages travel any further than necessary (Ranade [Ran87] achieves a similar result by requiring all message to travel the same distance, compromising locality).

Our approach also provides better space efficiency and more opportunities for parallelism. Several processing ‘threads’ can occur in parallel, so greater efficiency is possible and more opportunities for parallelism exist. Our approach re-maps the architecture of figure 21 as shown in figure 22.

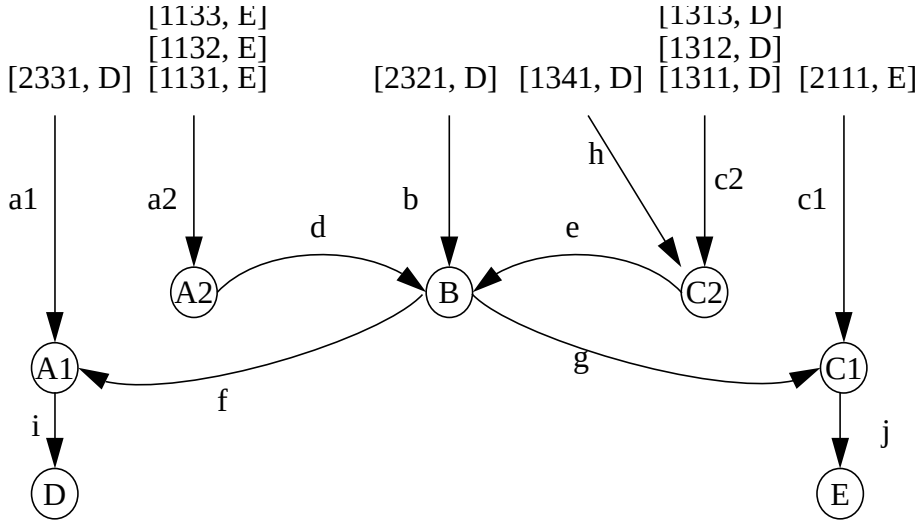


FIGURE 22. Recasting of the network in figure 21.

Nodes A and C (and inputs a and c) are separated into two sub-nodes (and two sub-inputs): A1 and C1 (a1 and c1) to handle messages with one step yet to travel in the network, and A2 and C2 (a2 and c2) to handle messages with three steps yet to travel. Processing occurs as before. Ini-

tially, each node sends the initialization message [1000, NULL] on each of its outputs (this message will arrive at the front of the input streams as well). Processing continues as follows:

1. Each node now has [1000, NULL] on each input, so [2000, NULL] can be sent on each output.
2. Nodes B, A1, and C1 has [2000, NULL] as its priority input, and sends [3000,NULL] on each output. Node A2 chooses [1131, E] and sends [2131, E] on output d. Node C2 chooses [1311, D] and sends [2311, D] on output g.
3. Node B has received [2131, E] and [2311, D], and can choose [2131, E] and send [3131, E] on output g, and [3131, NULL] on f (each of these messages will overwrite the ghost message [3000, NULL] at the inputs to nodes A1 and C1). Node A1 has [2331, D] as priority and sends [3331, D] to sink D. Node C1 sends [3111, E] to sink E in the same manner.
4. Node A2 now sends [2132, E] on output d. Node C2 is blocked waiting to send [2312, D] on g.
5. Node B can choose [2132, E] and pass on [2132, E] to node C1. Processing continues.

Note that deadlock does not occur in this architecture, and that messages are received by sinks D and E which would not be received in earlier examples until B had processed several more messages. If D and E were subnetworks, instead of sinks, this would mean an earlier opportunity for processing in those networks, and thus greater parallelism. In fact, if the example network were not symmetrical around node B, even more possibilities for parallelism would be apparent.

The next section describes a class of graphs which have the general structure of the architecture in figure 22. These graphs are distinguished by the fact that, if considered as shared-memory architectures, they allow efficient and correct implementation of local synchrony. We will later show how to generate such a graph which can be mapped onto any candidate shared-memory architecture.

4.3 LS Graphs

We describe a class of directed graphs called local synchrony (LS) graphs. In section 4.6, we present an algorithm which generates an LS graph from the interconnection topology of a general connected shared memory architecture. A generated LS graph can then be mapped back onto the candidate architecture using virtual channels [Dal87]. LS graphs can be used to implement local synchrony correctly on architectures having directed cycles without compromising locality.

We present LS graphs because they are consistent with the goals of local synchrony and they represent the most space efficient implementation we have found. We do not claim that our implementation is optimal with respect to space efficiency, only that it is better than other known approaches.

An LS graph is a directed graph {V, E}, where V is a set of vertices and E is a set of directed edges among the vertices in V. For simplicity, we consider only connected graphs. V is further subdivided into $\{V_S, V_R, V_I\}$. Vertex subset V_S (source or sending vertices) consists of vertices from which directed edges only originate. Vertex subset V_R (sink or receiving vertices) consists of vertices at which directed edges only terminate. Vertex subset V_I (intermediate vertices) consists of vertices at which directed edges both originate and terminate. By definition, subsets V_S, V_R, and V_I are disjoint (this requirement may be somewhat relaxed under certain circumstances).

It may be seen from the definition of LS graphs that set E for any LS graph is restricted such that edges may only originate at vertices in V_S and V_I, and may only terminate at vertices in V_R and V_I. In this discussion we will concern ourselves with directed paths from vertices in V_S to vertices in V_R. The path from a vertex in V_S to a vertex in V_R may consist of any number of edges and includes only vertices in V_I between the two end vertices. In other words, a path originates at a vertex in V_S, passes through zero or more vertices in V_I, and terminates at a vertex in V_R.

We further restrict E such that sub-paths from a given vertex, q, in V_I may not pass again through q before terminating. This restriction eliminates infinite-length paths in finite-sized graphs. It also eliminates directed loops in the graph, which is vital to our discussion of deadlock freedom in our general implementation technique for local synchrony (see section 4.4). We make no restrictions on the number of distinct paths from a given vertex in V_S to a given vertex in V_R, nor on the lengths of such paths, other than the non-looping constraint given above.

A vertex in V_I may be included in any number of paths from vertices in V_S to vertices in V_R. Let A be the set of sub-paths which connect vertex $q \in V_I$ to all vertices in V_R to which q is

connected. We restrict our definition of LS graphs such that the lengths (i.e., the number of edges) of all sub-paths in set A for any $q \in V_I$ are equal. We will call this length the distance value of the node q. This restriction aids in proving deadlock freedom for resulting architectures while preserving locality constraints when candidate architectures are transformed using LS graph mapping. Note this restriction applies only to vertices in V_I and not to vertices in V_S (the set A for a vertex in V_R has no elements).

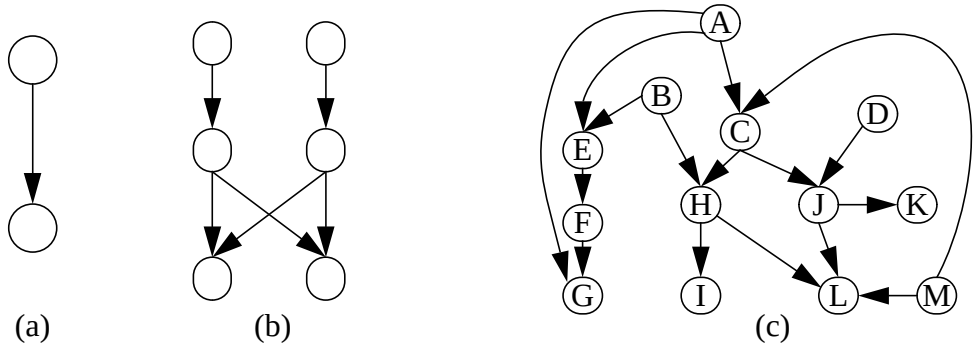


FIGURE 23. LS graphs

Figure 23 shows several examples of LS graphs. Consider graph c. The vertex set for this graph includes subsets V_S = {A, B, D, M}, V_I = {C, E, F, H, J}, and V_R = {G, I, K, L}.

We make the following observations about LS graphs:

1. Consider two paths, a and b, which are chosen from the A sets of two vertices in set V_I of an LS graph, and share some number of vertices. Any difference in the lengths of the two paths must occur in sub-paths a' and b' which begin at the first vertex in each path and end at the first vertex the paths have in common.

Let x be the first shared vertex. By definition of LS graphs, all sub-paths originating at x have the same length, l. Thus the length(a) = length(a') + l and length(b) = length(b') + l, and length(a) - length(b) = (length(a') + l) - (length(b') + l) = length(a') - length(b').

This is a simple observation, but it emphasizes an important point about LS graphs as defined here. When any two paths meet in the graph, their termination vertices are, at the meeting point, the same distance (i.e., number of edges) away. Paths may be of different lengths, but the difference must come before they share vertices. Note also that paths may diverge after sharing vertices, but the divergent sub-paths are of the same length.

2. Let A' be the set of lengths of all paths in set A for a vertex $q \in V_S$. The number of edges originating at q must be equal to or greater than the cardinality of set A'.

By the definition of LS graphs, the first vertex in any path from q is in set V_I or V_R, and all sub-paths originating at that vertex have equal length. Thus, for each length, $l \in A'$, there must be at least one vertex connected to q from which all sub-paths have length l-1.

Observation 2 shows that, in LS graphs as defined above, paths of different length from a vertex $q \in V_S$ to vertices in V_R must leave q by different edges.

In order to limit our discussion to currently 'interesting' cases, we define as a subset of the LS graphs the *regular* LS graphs. The LS graphs that we will generate and discuss as examples of our general implementation technique for local synchrony will come from this group. We define an LS graph to be regular if and only if:

1. Sets V_S and V_R are non-empty.
2. The number of edges originating at vertex $q \in V_S$ is equal to the cardinality of set A' for q.
3. Each vertex $q \in V_S$ is connected by at least one path to each vertex $p \in V_R$.
4. Each vertex $p \in V_R$ has one and only one input edge.

Note that a regular LS graph can be asymmetric with respect to the number of connections to each source or sink vertex. Sources have multiple outputs because the distance from a source to any two sinks may be different. All paths to a given sink have a one-step (from the last intermediate vertex to the sink) sub-path, however, so sinks need only one input, since all these sub-paths may be merged.

Consider an LS graph G = {V, E}. Let the vertices in V represent physical processing elements (PEs, MMs, switches) that communicate along directed physical communication channels (the edges in E) by passing messages or packets. The system is asynchronous, but a processing element cannot discard an output packet until its reception has been acknowledged by the receiving processing element.

We divide the processing elements into three groups: PEs are bound to the vertices in set V_S; MMs are bound to the vertices in set V_R; and switching elements are bound to the vertices in set V_I.

Note that if the original LS graph was regular, as defined, we have created a true shared memory architecture (we consider here only PE to MM message traffic—response traffic may be handled by a separate network). A group of PEs is connected to a group of MMs (representing global shared memory) by a network of switching elements. Each PE is connected to each MM by some path, made up of switching elements. We call an architecture of this type an LS architecture. The next section presents an implementation plan for local synchrony on LS architectures.

4.4 Local Synchrony Implementation on LS Architectures

We now discuss the implementation of local synchrony on architectures based on LS graph communication topologies, as presented in section 4.3. As in [ChM79], we assume a simple communication protocol for messages: a message is sent from node i to node j if and only if node i is ready to send the message and node j is ready to receive the message. We assume that a protocol, e.g. Hoare [Hoa78], is in place that guarantees that messages are transmitted correctly between nodes. All messages generated by nodes are 2-tuples, <p, m>, where p is a priority measure unique to the message and m is a global memory access or the value NULL (a ghost message).

The operation of the architecture is a relatively straightforward implementation of local synchrony. PEs execute a process that makes demands (accesses) on global shared memory. Using a logical timestamping system, PEs provide each global memory access with a unique priority (p) for execution, and release accesses in timestamp order.

The timestamping system that gives messages their unique priority is as follows: p consists of a four-tuple, {*ltu, dest, pid, rank*}; and the isotach velocity invariant is enforced by the use of token messages, generated by PEs, which delineate between ltus. Note that a message does not carry any information about the ltu in which it was released or will be executed. This information is inherent in the position of the message in the access stream.

dest is the destination global memory location. *pid* is the PE id (we assume for simplicity that each PE executes only a single process), and *rank* is the rank of the access among those

released by the same PE in the same logical time unit. Timestamps are compared lexicographically, with a lower timestamp translating to a higher priority for execution.

Switches conservatively pass global memory accesses from inputs to outputs by logical timestamp (rather than by arrival time, as might be the case in a normal system). A switch must wait until it has ‘latest timestamp’ information on each input before it can make an output decision.

Because they have only one input, MMs execute global memory accesses in timestamp order, which, for LS architectures generated from regular LS graphs, is the order of their arrival.

A switch or PE with more than one output also sends ghost messages, which carry timestamp information (but not an actual access) on unused outputs. When a true access is sent on an output, each other output may benefit from the transmission of a ghost message. Ghost messages arriving on a given line are overwritten by ghosts or true accesses arriving on that line at a later time (the conservative nature of local synchrony ensures that a message has equal or greater timestamp than the preceding message), which limits the proliferation of ghost messages within the system.

Because we are concerned here only with the communication of messages within the system and not the internal processes of PEs, which are general parallel programs, we may abstract the operation of the PE in our implementation. The pertinent algorithm of the PEs in a local synchrony implementation is as follows:

1. Execute internal process until it is necessary to generate a global memory access or token.

For a global memory access:

2. Generate a message tuple $<p_k, m_k>$ with non-NULL m_k and $p_k > p_{k-1}$.
3. Select the output channel on which the message will be sent.
4. Generate a message tuple $<p_k, m_k'>$, where m_k' = NULL, for each other output channel.
5. Send each message on its respective output channel.

For a token:

2. Generate a message tuple $<p_k, m_k>$, where m_k = NULL and p_k = T, a special value reserved for tokens, for each output channel.

3. Send each message on its respective output channel.

Switching elements and MMs, in essence, do not initiate actions (excepting on start-up, as discussed below). Their process is driven completely by their inputs from PEs and other switches. The algorithm for a switching element is as follows:

1. Initially, send a token on each output.
2. Wait until a message (true or ghost) has been received on each input channel.
3. Create LOW, the set of all inputs with minimum p-value.
4. In the event that LOW has more than one element, choose the message $<p_k, m_k>$ with non-NULL m-value or, if all have NULL m-value, choose at random (since each is a copy of the others).
5. Select the output channel on which the message will be sent. If the destination of the message is not reachable from this switching element (the message is a ghost or token), choose an output at random.
6. Generate a message tuple $<p_k$, NULL> for each other output channel.
7. Consume (remove from the system) all input messages in LOW.
8. Send each message on its respective output channel.

We note here that step 1 is only necessary for the implementation to be an isotach network [Wil93]. This approach is equivalent to the time increment approach, and PEs generate and send isochrons according to the description in Chapter 2. Since there are no cycles in the LS graph, it may be efficient to implement the time propagate approach. In this case step 1 would not occur, and PEs would queue isochrons differently. Throughout this dissertation, we assume isotach implementation.

MMs simply consume accesses as they arrive. For this implementation, we assume that ghosts are sent to MMs, but adjacent switches may also consume ghosts targeted to MMs. This algorithm assumes implementation on a general LS architecture, where MMs might have more than one input. The specific cases we will discuss later in this chapter come from the set of regular LS architectures, in which each MM has only one input. An MM's algorithm is as follows:

1. Wait until a message has been received on each input channel.
2. Create LOW, the set of all messages on inputs with lowest p-value.
3. Consume (remove from the system) all messages in LOW. The internal process of the MM would execute the global memory access for any true access and would merely discard any ghost or token (If a locally synchronous response network is in place, ghosts would pass directly into that network).

In the next section we will demonstrate that the LS architecture implementation described above meets our definition of correctness for such a system. We will go on in later sections to show that any shared memory architecture can emulate an LS architecture, allowing correct and deadlock-free implementation of local synchrony.

4.5 Correctness of LS Architecture Implementations

As discussed in section 4.1, the three elements which constitute correctness for a local synchrony implementation, then, are deadlock freedom, the correct order of consumption of messages by the MMs, and a guarantee of progress to the termination state. In this section we prove that the implementation presented in section 4.4 for the architectures of section 4.3 meets these criteria.

4.5.1 Correctness of Order of Consumption

The logical timestamps of messages directed to a given sink node (MM) define the correct consumption order. Messages must be consumed by the MM in timestamp order. We assume that the stream of messages output by source nodes (PEs) on each of their outputs is in timestamp order.

Lemma 4.1: The consumption order of messages at any sink (MM) is correct.

Proof. Assume the converse, that some MM does not consume messages in timestamp order. If this is the case, then the sequence of messages arriving on one of the MM's inputs is not in order, because the MM is constrained to choose conservatively the earliest timestamp on any input for consumption, and to block if messages have not arrived on all inputs. Consider the node connected to the input with incorrectly ordered messages. This node cannot be a source, because we assume that all outputs of sources are correct. It must therefore be an intermediate node. Because the node is also constrained to choose conservatively, one of its inputs must be out of order as well. We thus have a chain $\{S, I_1, I_2,...\}$ of nodes, each of which has a faulty input. Because there are no directed loops in the network, and the network is finite, this chain must end at a source node. Contradiction. ■

4.5.2 Deadlock Freedom

We derive this argument in part from the deadlock freedom arguments in [ChM79]. Chandy and Misra define a communication line (i, j) to be a WN line if node i is waiting to output to node j, but j is not waiting for input from node i. NW, NN, and WW lines are defined similarly. Let S be the service time of all nodes in the network. Let T_i be defined as the timestamp of the last message accepted for output by node i (note that when output, this message's timestamp will be increased by S). T_i is then the 'current' logical time, or the logical time up to which node i has progressed.

Lemma 4.2: At any point in the computation,

1. if (i, j) is a WN line then $T_i > T_j - S$, and
2. if (i, j) is an NW line then $T_i = T_j - S$

Proof. Observation 1: If (i, j) is a WN line, then node i has $T_i = t$ and is waiting to send a message with timestamp t+S to node j, while j has not yet processed the previous message from i, which has timestamp t' < t+S. By the conservative nature of the system, $T_j < t'$, and thus the result. Observation 2: If (i, j) is an NW line, then node j has processed the last message sent from i and no more (by the conservative nature of the system), and waits for more input from node i. T_j, is therefore equal to the timestamp of the last message sent from node i, which is by definition T_i+S. ■

We define the system to be deadlocked if:

1. all nodes are either in the termination state or in wait state (either waiting for input or waiting to output), and at least one node is in a wait state, and
2. there are no two nodes waiting for each other; i. e., a waiting to output to b and b waiting for input from a. If this is the case then node a can send a message to node b and condition 1 is not met.

Lemma 4.3: If an LS architecture is deadlocked, then a deadlock circularity exists, and this circularity includes both NW and WN lines.

Proof. Assume the system is deadlocked. Choose any node i which is in a non-termination state. Node i is waiting either for input or to output. Let j be the node i waits upon. Node j cannot be a source or sink node. If node j were a source or a sink, then by assumption, j would continue at some point and no deadlock would exist. Node j cannot be in the termination state, because one of its incident lines has not transmitted an EOC message. J cannot be waiting on i, because i is waiting on it and by the definition of LS architectures there is only one communication line between i and j, so j must wait on another node. Construct a sequence of intermediate nodes $\{N_0, N_1, N_2,...\}$

where N_i is a node in a wait state and each successive N_i waits on N_{i+1}. Since the number of nodes is finite, there exist k and l such that node k and node l are the same. Let k and l be chosen such that $N_{k+1},...,N_l$ are distinct. The sequence N_k->...->N_k must include both NW and WN lines, as there are no directed cycles in the network, the sequence is circular, and a WN dependence represents a step from stage i to stage i-1, while a NW dependence represents a step from stage i to stage i+1. Note that WW and NN lines are not possible within the circularity, as the nodes connected by a WW line can progress, and neither node waits on a NN line. ■

> **Corollary to Lemma 4.3:** In any deadlock circularity in an LS architecture, the number of NW lines is equal to the number of WN lines.

Proof. Choose any node in the circularity. Starting from this node, the circularity makes a number of ‘down’ steps (WN lines - from stage i to stage i-1) and a number of ‘up’ steps (NW lines - from stage i to stage i+1). Since these are the only two possibilities, and the path is circular, it is apparent that the circularity includes the same number of ‘up’ steps as ‘down’ steps. ■

> **Theorem 4.1:** The system is never deadlocked.

Proof. Assume the converse. By lemma 4.3 a deadlock circularity must exist, and may be defined N_k->N_k as above. Without loss of generality, assume that N_{k+1}->N_{k+2} is a WN line, and that the distance value d of node N_{k+1} (defined earlier as the length of all sub-paths from the node to the sink nodes) is equal to the greatest distance value in the sequence. Both N_{k+2} and N_k then have distance value d-1, and N_k->N_{k+1} is a NW line. Construct the sequence of nodes {N_A=N_{k+1}, N_B,..., N_N} which includes, ordered according to their position in the circularity, all nodes within the circularity with both an incident NW and an incident WN line. Note that by selecting this set, the lines in the deadlock circularity that connect any two nodes in the set are all NW or all WN. By lemma 4.2,

$$T_A > T_B - bS$$
$$T_B = T_C + cS$$
$$T_C > T_D - dS$$
$$...$$
$$T_{N-1} > T_N - nS$$
$$T_N = T_A + aS$$

where b is the number of NW lines between nodes A and B, and c is the number of WN lines between nodes B and C, and so on. By back substitution, we reduce this set of equations to:

$$TA > TA + (a - b + c - d +...- n) * S$$

However, by corollary 4.3.1:

$$a - b + c - d +... - n = 0,$$

This contradicts our assumption, and so proves the theorem. ■

We have shown that local synchrony timestamp ordering is deadlock-free on architectures represented by LS graphs. Some of the arguments made (specifically in the proof of the corollary to lemma 4.3) make reference to properties specific to LS graphs. While it is our opinion that this proof can be extended to the class of general directed acyclic graphs, such discussion is beyond the scope of this dissertation.

4.5.3 Termination

Showing deadlock freedom for this implementation demonstrates that the system will not halt prematurely. By definition, information flow is one-way within the system, with no directed loops. Each source generates a finite number of messages, which may generate at most a finite number of other messages (a message may cause the generation of at most g ghost messages, where g is the total number of communication lines in the system). Thus only a finite number of messages need pass through the system in order to complete a computation. Progress follows from deadlock freedom, and the system is guaranteed to reach the termination state in a finite number of steps.

In the next section we present an algorithm which generates an LS graph from the architecture graph of a candidate architecture. The LS architecture which this graph represents can be mapped onto the candidate architecture, allowing correct and deadlock-free implementation of local synchrony on the candidate architecture.

4.6 Mapping LS Architectures onto Candidate Architectures

Consider the NYU Ultracomputer architecture. The interconnection topology of this architecture fits the requirements of the definition of LS graphs. Each node includes only one element: a PE, an MM, or a switching element (a PE or MM might have an associated network interface unit– for our purposes such a configuration represents a single element). PE nodes have only outputs and MM nodes have only inputs. The sub-paths to MMs originating in each switching element all have the same finite length (which also means that there are no directed loops in the network). Thus the

Ultracomputer architecture enables the implementation of local synchrony in a correct and deadlock-free way. This is true for any equivalent equidistant architecture.

This is not the case with non-equidistant architectures like a hypercube or mesh, however. These architectures exhibit locality, meaning that the paths from any point to differing destinations may be of differing lengths. Non-equidistant architectures are also generally composed of complex nodes including PEs and MMs, as well as switching hardware.

We now prove that local synchrony can be implemented on any connected shared memory architecture (as discussed below) by presenting an algorithm for creating an LS graph which can be mapped onto the interconnection topology of the candidate connected shared-memory architecture. Local synchrony can then be implemented on the candidate architecture by using virtual channels [Dal92] to simulate the LS architecture represented by the LS graph, without loss of the locality of the candidate architecture. We have already shown in previous sections that the class of LS graphs represents interconnection topologies for architectures on which local synchrony can be implemented in a correct and deadlock-free way.

We make the following assumptions about connected shared memory architectures:

1. A shared memory architecture consists of a network of interconnected nodes. Each node may have zero or more PEs, MMs, and switches, although a node must have at least one of the three, and a node with more than one input or output must have a switch to handle node I/O (we assume without loss of generality that PEs and MMs perform no switching functions). We further assume, without loss of generality, that all elements within a switch are connected, and that a PE and an MM in the same node are connected through a switch. Note that traditional distributed memory architectures may emulate shared memory.
2. We assume for simplicity that the network is fault-free and that an I/O system is in place to handle communication between nodes.
3. We assume, again for simplicity, that communication channels are one way. A channel allowing two-way communication is considered to be a pair of one-way channels.
4. Each input channel is able to buffer at least one message. Local synchrony requires this (that the network be non-discarding) for implementation.
5. Switches perform operations sequentially, but we abstract the acceptance of input into a buffer cell from the operation of a switch. In other words, a switch may accept input into an empty input buffer cell while waiting to output a message to another switch.

6. Again, for simplicity, we will consider message traffic only from PEs to MMs. We assume that a separate network handles the responses generated by the MMs.
7. We assume that the routing scheme in place does not route messages in such a way that they pass through the same node more than once. This assumption eliminates looping and infinite length paths. This assumption may be relaxed to allow longer finite paths for fault tolerance and load balancing purposes.

We now present the algorithm for generating an LS graph which may be mapped onto the interconnection topology graph for any given shared memory architecture. The candidate architecture may then emulate an LS architecture, in which the physical nodes of the candidate architecture may consist of several virtual nodes, and the physical channels may consist of several virtual channels. By this mapping a correct and deadlock-free implementation of local synchrony is possible on any shared-memory architecture:

1. Begin with the interconnection topology graph, C, of any shared memory architecture.

The goal is to generate an LS graph which can be mapped onto the physical architecture of the candidate, preserving any locality present.

2. Separate all nodes in graph C with > 1 element (MM, PE, or switch) into separate nodes, each with one and only one element. Keep any node interconnections.

There are now three sets of nodes: PE nodes, MM nodes, and SWITCH nodes. Note that a node in the original architecture is now a set of nodes related by their physical proximity in the candidate architecture, and that PE and MM nodes may only be connected to SWITCH nodes.

3. Translate SWITCH nodes to sets of switching nodes. Compute the set A of all possible paths from PE nodes to MM nodes in the graph. Compute further the set A* of all distinct lengths of paths from PE nodes to MM nodes in the graph. Let C be the greatest length in A*. Separate each SWITCH node N' into a set of (virtual) switch nodes, $\{N_1, N_2, ..., N_C\}$, one for each possible sub-path distance.

In step three, the routing scheme employed in the candidate architecture becomes an issue. Obviously, the routing scheme defines the possible paths which might link a PE to an MM. In this discussion we generally consider only static, shortest path schemes, which are guaranteed to decrease the distance to the MM with each step. However, dynamic systems with fault tolerant or load balancing capabilities are completely compatible with this implementation technique, given

that there is a maximum limit on the distance an access will travel before reaching global memory. The cost in buffers of such systems is considered in section 5.1.7 and chapter 6.

4. For each switching node N_i, N_i has a connection from a PE node p if p was connected to N', and p had a path to any MM of length i.

Step 4 provides a separate output from each PE for each of the different distances messages emitted from that PE may travel. All connections made in step 4 are intra-node connections in the candidate architecture.

5. For each switching node N_1, a communication connection from N_1 to an MM node q exists if N' was connected to q.

Step 5 connects MMs to the switch network. Since messages may only arrive at the MM from one step away, the MM generally has only one connection.

6. For each switching node N_i, a communication connection to N_i from M_{i+1} exists if there is a connection between N' and M' in the candidate architecture; and for some sub-path of length i in the candidate architecture, s = {N,...,q}, where q is an MM node, there is a sub-path t = M' $\cup$ s in the candidate architecture also.

Step 6 covers the internode connections in the candidate architecture. A connection is generated for each physical communication channel for each possible distance a message might travel after passing through that channel.

7. Remove any switching nodes which have no communication connections from the graph.

Step 7 removes unnecessary virtual switches created in step 3. Step 7 is unnecessary for most interesting architectures. In section 4.8.3 we present an example of a candidate architecture that makes use of step 7 to simplify the generated LS graph. Let us here restate the requirements which make up the definition of LS graphs.

1. An LS graph is a directed graph {V, E}, where V consists of three distinct subsets of nodes {V_S, V_I, V_R), and E is a set of directed edges.
2. Nodes in V_S originate directed edges only.
3. Nodes in V_R terminate directed edges only.
4. Nodes in V_I both originate and terminate directed edges.
5. There is no directed path from any node to itself.
6. For any node q $\in$ V_I, all directed paths originating at q which terminate at any node r $\in$ V_R have the same length.

We now show that the translation algorithm produces an LS graph.

Lemma 4.4: The translation algorithm creates an LS graph from the candidate architecture's interconnection topology graph.

Proof. Requirement 1 is met by steps 2 and 3 of the algorithm. Let V_S be the PE nodes, V_I be the SWITCH nodes, and V_R be the MM nodes in graph C after step 2. Step 3 merely subdivides the nodes in V_I. Nodes in V_S are given edges only in step 4, and all edges assigned meet requirement 2. Nodes in V_R are given edges in step 5, and all edges assigned meet requirement 3. Nodes in V_I are given edges in step 6. If a node in V_I is assigned an input edge in step 6, it is required by the conditions of step 6 to have at least one output edge. If a node in V_I has no input or output edges, it is eliminated by step 7. Nodes with only output edges are not possible, because the path connecting such a node with a node in V_R is by definition a sub-path of some path from a node in V_S to the node in V_R, and thus a node in V_I with one or more output edges must have one or more input edges.There is no directed path from any node to itself. Nodes in V_S and V_R have only output and input edges, respectively. Input edges to nodes in V_I come only from nodes in V_S or nodes in V_I with higher distance indexes, and output edges go only to nodes in V_R and nodes in V_I with lower distance indexes. Information flow is thus only 'one-way'. This meets requirement 5. Finally, we prove that requirement 6 is met by induction on the number of subsets of nodes in V_I. All nodes $N_1 \in V_I$ are output connected only to MMs by step 5 (and the fact that no nodes $N_0 \in V_I$ exist), and thus all sub-paths to nodes in V_R from nodes $N_1 \in V_I$ are of length 1. All nodes $N_2 \in V_I$ are output connected only to nodes $N_1 \in V_I$ by step 6, and thus all sub-paths to nodes in V_R from nodes $N_2 \in V_I$ are of length 2. Assume all sub-paths to nodes in V_R from nodes $N_i \in V_I$ are of length i. By step 6, all nodes $N_{i+1} \in V_I$ are output connected only to nodes $N_i \in V_I$, and thus all sub-paths to nodes in V_R from nodes $N_{i+1} \in V_I$ are of length i+1. By induction, requirement 6 is met. ■

An LS graph which is mappable onto the candidate architecture results from performing this algorithm. The LS graph represents an equivalent architecture on which local synchrony can be implemented in a correct and deadlock-free way. However, via mapping the locality of the candidate architecture is preserved. Each node in the candidate architecture becomes a set of virtual nodes. Emulating the resultant LS architecture on the candidate architecture requires only the use of virtual channels [Dal92] to create virtual switching nodes within the switching elements of the candidate architecture.

The next section presents several examples of the process of generating an LS architecture from a candidate architecture.

4.7 Examples of Local Synchrony Implementations

This section includes several examples of specific implementations of local synchrony on non-trivial candidate architectures. Implementation is made possible by using the algorithm of section 4.7 to generate an LS architecture which can be mapped onto the candidate architecture.

4.7.1 Implementation on a Torus

We first perform the transformation algorithm on a 3x3 torus (wrapped mesh or 3-ary 2-cube) architecture. Figure 24 presents the candidate architecture and the node architecture for the candidate. We assume a simple routing scheme for accesses. An access first moves along its source ‘row’ until it reaches its destination column, then continues to its destination. This scheme is equivalent to high-order E-routing in binary n-cube architectures.

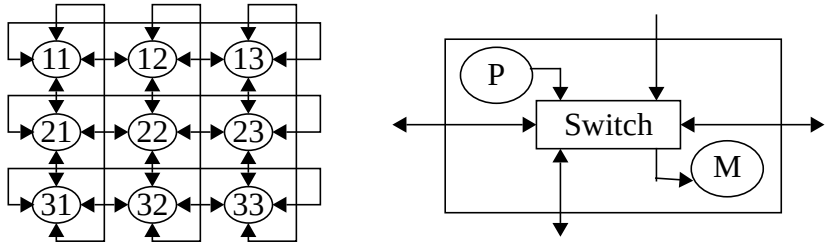


FIGURE 24. 3x3 torus and torus node architecture.

Figure 25 shows the architecture after step 2 of the transformation algorithm. The layout has been modified for readability. Note that all inter- and intra-node connections are intact at this point.

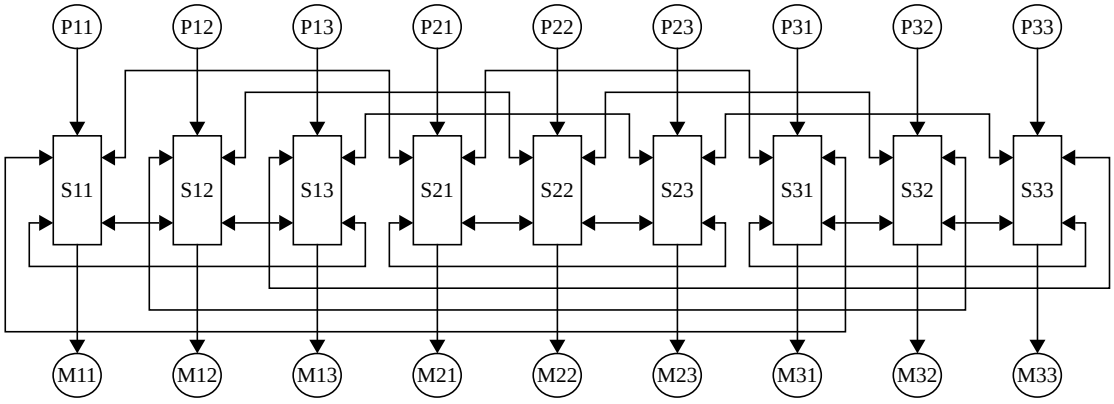


FIGURE 25. 3x3 torus after step 2 of transformation algorithm.

In calculating the set of all paths from switching nodes to MMs, we find the maximum length of any path to be three (note that this maximum length includes the step from a local switch to it's local MM). Each switching node is divided into three separate switching elements, to handle accesses with varying distances to travel to their destination MM. The state of the transformation after step 3 is depicted in figure 26.

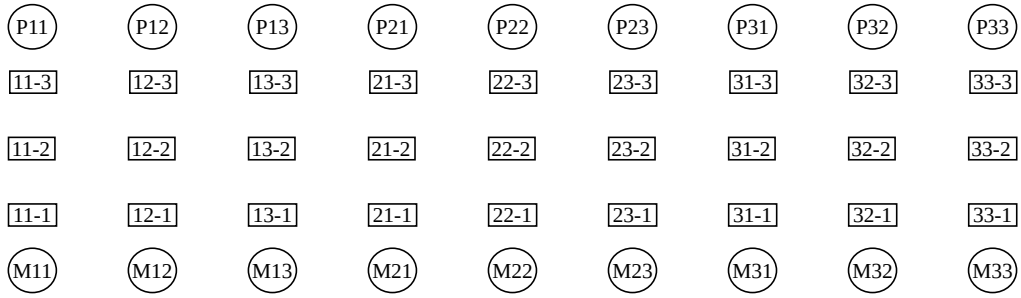


FIGURE 26. 3x3 torus after step 3 of transformation algorithm.

Figure 27 shows the transformation after step 4 of the algorithm. Step 4 connects PEs to switching elements. Note that in the case of the torus, each PE is connected to each of its 'local' switching elements. PEs can only be connected to 'local' switching elements, but need not necessarily be connected to each.

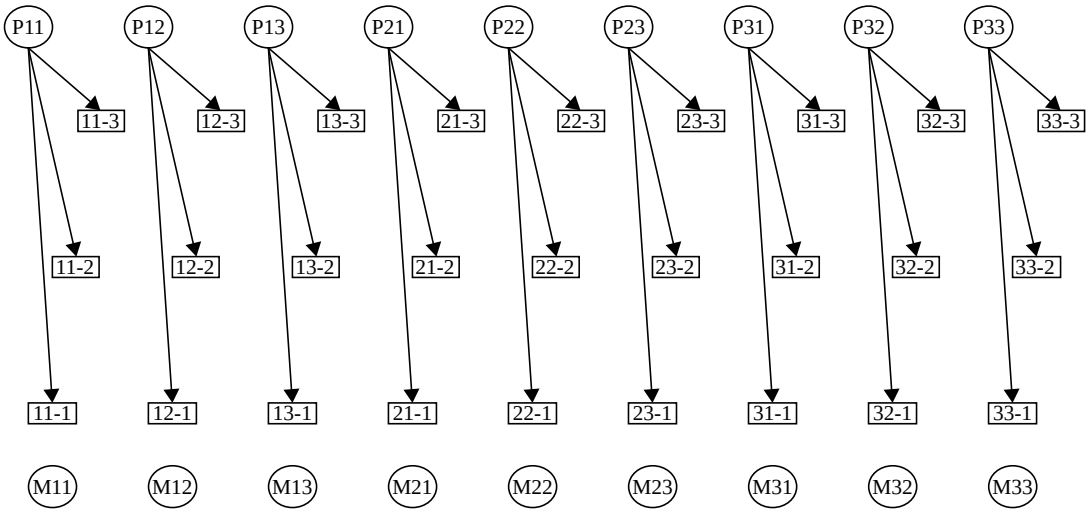


FIGURE 27. 3x3 torus after step 4 of transformation algorithm.

Step 5 of the algorithm connects switching elements to MMs. Note that MMs have only one connection, and it is only to a 'local' switching element. This belies the assumption, made earlier for shared memory architectures, that communication between physical nodes be only through

the switches (a PE or MM cannot have a direct ‘outside’ connection, but must communicate through a switch). Figure 28 shows the transformation after step 5 of the algorithm.

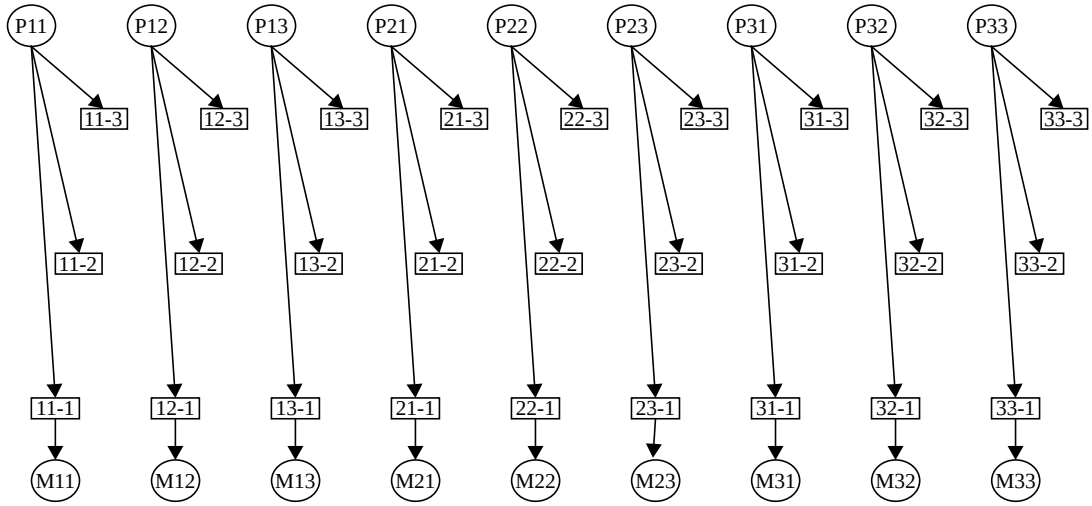


FIGURE 28. 3x3 torus after step 5 of transformation algorithms.

Step 6 of the transformation process forms the switching element interconnections, and is shown in figure 29. Note that switches on a given level are connected only to switches on the level above and the level below, and that the direction of flow is always downward on the graph. This shows the absence of directed loops in the graph.

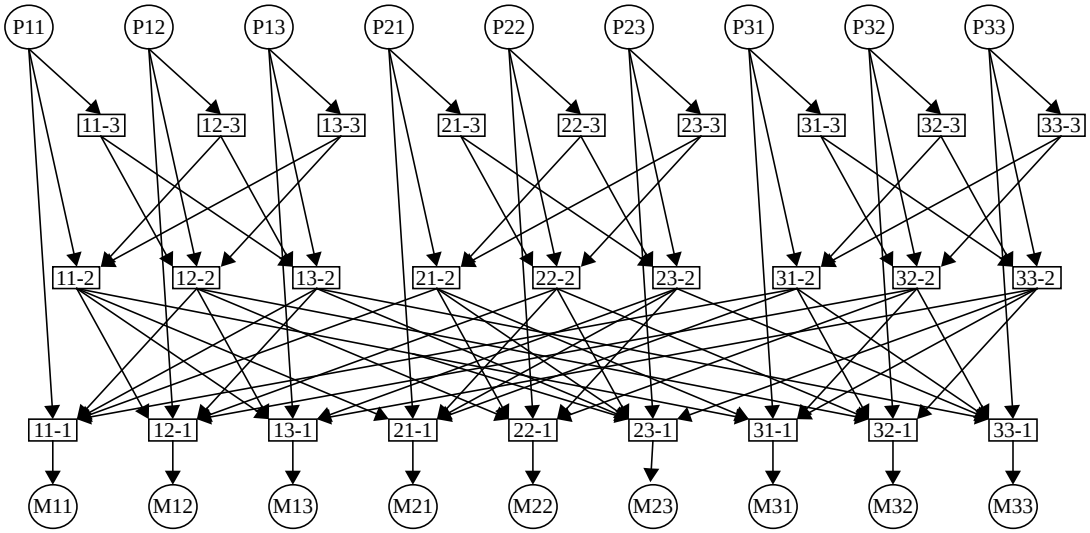


FIGURE 29. 3x3 torus after complete transformation algorithm.

Step 7 has no effect on this transformation, as all switching elements are connected, so figure 28 depicts the torus after complete transformation into an LS architecture.

4.7.2 Implementation on a Binary 3-Cube

The binary n-cube architecture has proven to be a popular parallel computing architecture. We present here a brief example of an LS graph transformation of a binary 3-cube architecture, depicted in figure 30.

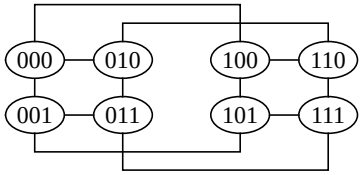


FIGURE 30. Binary 3-cube architecture.

For simplicity, we assume a static, low-order e-routing scheme for global memory accesses. Low-order e-routing sends messages along paths which cross the lowest dimension (as represented by the node ID) which must be crossed in order to reach the destination node. The transformation algorithm will generate an LS graph (which represents an LS architecture, which guarantees correct, deadlock-free implementation of local synchrony) for any routing scheme, given that routing paths have a finite maximum length.

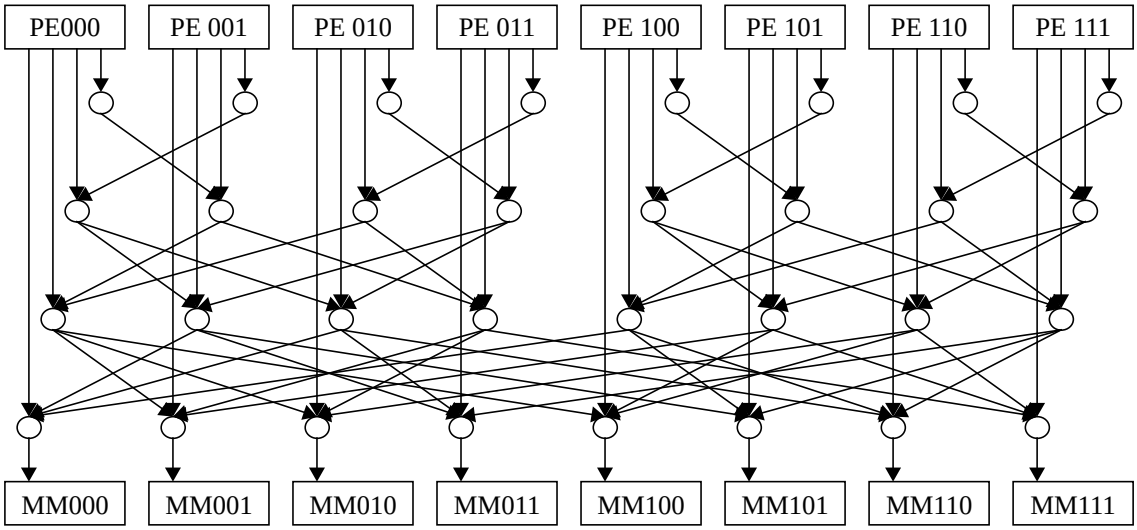


FIGURE 31. Final LS graph transformation for binary 3-cube.

Nodes in the cube are assumed to include a PE, an MM, and a switch. Figure 31 shows the LS graph translation of the binary 3-cube architecture, assuming the low-order e-routing scheme. Note that each node in the original architecture has been divided into 6 sub-nodes: a PE, an MM,

and 4 sub-switches (the longest path allowable by the routing scheme in the candidate architecture is of length 4, including the step from local switch to MM). Note again that in the LS graph translation, there is no connection between any two nodes which cannot be simulated by a true physical connection in the candidate architecture.

4.7.3 Implementation on a Crossbar

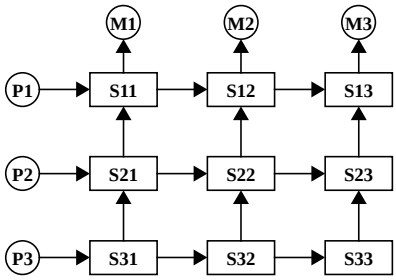


FIGURE 32. 3x3 crossbar.

A crossbar network is an indirect network as shown in figure 32. A square network of switches connects a row of MMs to a column of PEs. This example illustrates how the general implementation technique works for non-equidistant, indirect networks. We will assume for this example that the routing scheme transfers global memory accesses along the origin row to the destination column and then to the destination.

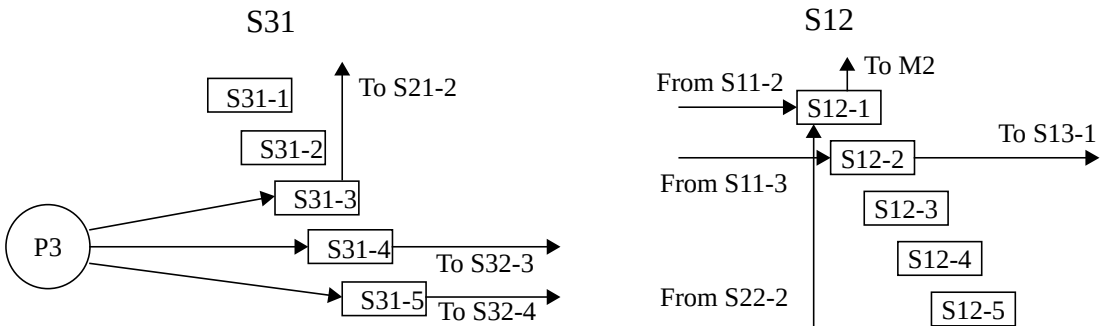


FIGURE 33. Intermediate state of S31 and S12 in transformation.

In the transformation algorithm, step 2 does not change the graph. All nodes already have only one element. In step 3, we compute that the longest path from any switching element to any MM is of length five, so each switching node is divided into five switching elements. Figure 33 shows the results of steps 4, 5, and 6 for nodes S31 and S12. Note that several of the switching ele-

ments have no inputs or outputs and will be discarded in step 7. Note that no PE or MM in the crossbar architecture has any ‘local’ switching elements.

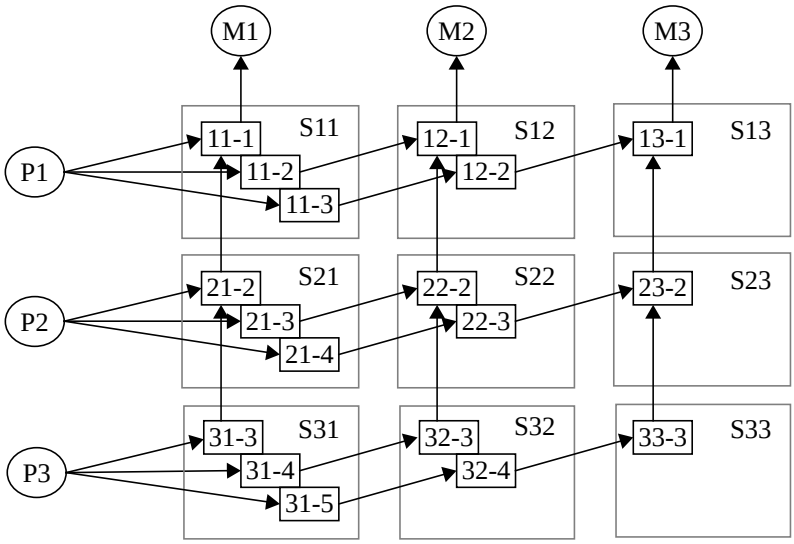


FIGURE 34. Final transformation for 3x3 crossbar.

The result of the transformation algorithm is presented in figure 34. In the case of the crossbar, switch nodes in the candidate architecture are divided into different numbers of switching elements. In each of the previous examples, all switching nodes were redesigned in the same fashion. This is a function of the underlying topology of the candidate architecture.

4.8 Operation of Virtual Switching Elements

In operation, a switching element in the candidate architecture of a local synchrony implementation may consist of several virtual switching elements, each handling global memory accesses with different length sub-paths to travel before execution. This is accomplished by the use of virtual channels [Dal92].

Virtual channels are created by dividing the buffer for a communication channel into several smaller queues, one for each virtual channel to be implemented. The virtual channels are then allocated independently but compete with each other for the physical bandwidth of the physical channel itself.

In this particular application, progress is guaranteed and possible deadlock avoided by giving highest priority to virtual channels handling messages with the smallest distance to travel.

For example, in figure 31 above, switch S31 is separated into three virtual switching elements. Both 31-4 and 31-5 use the physical channel connecting S31 to S32. 31-4 would normally be given priority over 31-5 for use of the channel, as it opens buffer cells closer to global memory. 31-5 would get access to the channel if 31-4 was empty or blocked. This is the simplest algorithm for physical channel allocation to virtual switching elements. Other algorithms might achieve a more fair distribution of access without the possibility of deadlock, given that a blocked virtual switching element cannot continually hold the physical channel while other virtual switching elements that might proceed are locked out.

4.9 Memory Requirements for Implementation

In order to implement the virtual switching elements required for a non-trivial local synchrony implementation, the minimum buffer requirements of the systems must be enlarged. Instead of a minimum of one buffer cell per physical channel, such an implementation would require a minimum of one buffer cell per *virtual* channel.

The number of extra buffer cells required varies depending on the topology of the candidate architecture. This implementation of local synchrony may also be compared to the modified Chandy/Misra approach, which eliminates the LS graph translation technique but requires that each input queue in the system be long enough to buffer all the messages it might receive within a given ltu.

In general, our implementation requires O(d) extra buffer cells per physical channel, where d is the depth, or longest distance an access might travel within the network. In order to avoid deadlock, the time increment approach with directed cycles (applying the approach to non-LS architectures–we will refer to this as the modified Chandy/Misra, or MCM, approach) requires O(nk) extra buffer cells per physical channel, where n is the number of sources (PEs) and k is the number of accesses a source can emit during an ltu. While it is possible for the depth d to be O(n), this is a worst case situation, and most interesting architectures have d of smaller magnitude.

Consider the three examples we presented in section 4.7. The torus has O(n) physical channels total. Since the depth of the torus topology is $n^{1/2}$, implementing local synchrony requires $O(n^{3/2})$ buffer cells total. The memory cost of MCM would be $O(kn^2)$ total buffer cells over all nodes, where k is the maximum number of accesses that might be released by a PE in a given ltu.

The binary n-cube has O(n log n) physical channels total. Since the depth of the hypercube is log n, local synchrony implementation requires $O(n [\log n]^2)$ buffer cells total. This result contrasts with MCM, which requires $O(kn^2 [\log n])$ buffer cells total.

The requirements for the crossbar are more difficult to calculate because of the varying buffer cell requirements for different nodes. The crossbar has $O(n^2)$ physical channels total.Since the depth of the crossbar is O(n), a local synchrony implementation requires$O(n^3)$ buffer cells total. Again, this result contrasts with MCM, which requires $O(kn^3)$ buffer cells total.

Our examples have only considered shortest-path routing schemes. A designer may wish to incorporate fault-tolerance or load-balancing into an implementation by providing non-shortest-path capability, including looping in access paths. This is possible within the confines of a local synchrony implementation, as long as a maximum path length is enforced. The maximum path length determines the depth of the LS graph that will be mapped onto the candidate architecture, and thus must be finite and defined by the routing scheme. Note that any finite routing scheme will be deadlock-free in this implementation, by virtue of the LS graph mapping. Because of buffer costs, we would expect the maximum path length to best be limited to a small number of steps more than the physical maximum enforced by the candidate architecture.

For example, a local synchrony implementation on a binary 10-cube has an architectural maximum path length of 10 steps. The designer might wish to increase this limit artificially to 16, to allow for load balancing or fault tolerance. Instead of 10 virtual switching elements, each switch is divided into 16 elements. Note that, although accesses may be passed back to nodes they have previously passed, this does not affect the non-looping requirement of the LS graph, and thus the deadlock-freedom for the implementation.

4.10 Summary

In this chapter, we presented a general technique that allows local synchrony to be implemented on any shared memory architecture. Our implementation scheme preserves any locality present in the physical architecture, and provides greater opportunities for parallelism and less overhead than other approaches.

First we defined correctness criteria for local synchrony implementations. Then we discussed two different approaches to the implementation of local synchrony, based on the logical service time of a switching element. We showed that each approach has different problems in implementation. Our choice of approach was the time increment approach, which is more efficient and uses less resources than the time propagate approach.

We defined LS graphs, and presented a local synchrony implementation for architectures represented by LS graphs. We proved that this implementation is correct and deadlock-free. The we presented an algorithm which generates an LS graph which is mappable onto any candidate shared-memory architecture, and proved the algorithm correct.

Finally, we gave several examples of LS graph transformations for interesting architectures. Also we discussed the operation of virtual switching elements, which are used in LS graph mapping, memory requirements for implementation, and compatibility with different routing schemes.

In the next chapter, we discuss some of the fault-tolerance aspects of local synchrony implementations.

5.0 Fault-Tolerance Issues

This chapter discusses some issues regarding fault-tolerance in the general local synchrony implementation presented in chapter 4. Because logical synchronization is built into local synchrony, some traditionally difficult fault-tolerance problems are easily implemented in locally synchronous systems. This is in contrast to traditional implementations, where synchronization must be considered in the problem solution.

A local synchrony implementation is able to tolerate certain types of temporary faults without compromising its state. Permanent faults may also be tolerated using redundant path techniques in hardware, which have been proposed in the literature.

We present a local synchrony rollback (LSR) procedure for checkpointing and rollback of computations in the general local synchrony implementation presented in Chapter 4. We define a checkpointing procedure which is guaranteed to save a consistent system state. Then we present a rollback procedure that is simple to implement and does not require processes to suspend execution. We show that the rollback procedure is correct, and that under certain circumstances it is possible to limit the number of saved checkpoints through validation, which guarantees that no rollback will occur beyond a given checkpoint.

5.1 The Preservation of System State in the Presence of Faults

We discuss the operation of local synchrony in the presence of a single, simple fault. The fault in this case will be the failure of a single physical communication channel. The operation of a *fail-stop* [Sch83] system requires that failures be detected before the system may enter any state resulting from the fault. The importance of fail-stop operation is that no information in the system is changed or lost due to a fault. For example, a fail-stop channel fault would not cause the loss of any access from the system. An access being transmitted when the fault occurred would either be fully transmitted or not transmitted at all.

We argue that the normal operation of a locally synchronous system is guaranteed to be tolerant of fail-stop faults, due to its conservative nature. Throughout this chapter, we make the following assumptions about the architecture on which our implementation is operating:

1. The architecture is composed of a packet-switching network, singly buffered on input channels only.
2. The routing scheme for accesses is static. In other words, accesses bound for a given MM will always follow the same path.
3. The communication channels of the network are fail/safe. In other words, the failure of a channel does not cause accesses to be lost or partially transmitted. Incorrectly transmitted accesses are assumed to be detectable.
4. The cessation of computational progress is detectable. The system is able to detect a channel fault, given that the progress of the computation has been halted.
5. Channels or nodes which have failed can be repaired or replaced.

Also we make the following statements concerning an implementation of local synchrony:

> A local synchrony implementation is a conservative (timestamp) ordering system. Thus, an element (PE, switch, or MM) operating in a locally synchronous manner must perform actions (pass or execute accesses) on a given resource (channel, memory location) in a manner which preserves a serializable schedule of global memory access. An element operating in a conservative manner is constrained not to perform any action until it has, with certainty, performed all previous actions, by the ordering constraint.

Consider the failure of a communication channel that handles accesses being transmitted from node A to node B. Channel AB acts as a resource necessary for some of the actions to be performed by node A. Since the fault has made this resource unavailable, node A must hold any actions requiring it. Node A can continue operation only by performing actions that do not utilize channel AB. When node A has nothing but actions requiring channel AB, it will block.

Node B will block within a finite amount of time after it finds the input buffer for channel AB empty. This condition is due to the conservative nature of the operation of a locally synchronous switch. Note that both A and B are ‘blind’ to the failure of channel AB. Both nodes are operating normally under the conservative principle that, being unable to choose (node B) or perform (node A) its next action, an element must block.

Assuming the faulty channel can be replaced/repaired, processing will continue normally. A and B (and their neighbors) might have fallen behind during the interim, but the conservative nature of local synchrony has guaranteed that no action has been performed out of order.

The computation may also be allowed to continue until progress has halted. The other neighbors of nodes A and B will block next, awaiting information from a node which is currently blocked. Again, these nodes will continue operation only to the point at which they cannot conservatively choose or perform their next action. At some later juncture, the blocked condition will have spread to every node, and computational progress will cease. Each switch, MM, and PE will be blocked, either waiting to receive information or waiting to send information to a currently full buffer (a PE might be idle, having finished its program). Once again, note that since conservative operation occurs at each element in the system, no action has been performed contrary to the ordering requirements of local synchrony.

At this point, the state of the computation may be saved to some stable storage or transferred to a non-faulty machine. This includes the states of global memory; all access queues in the network; and the local memory, program, and process state of each PE. When the faulty channel has been fixed, the computation may be reloaded and restarted at the exact point where it blocked, with no compromise of local synchrony.

Of course, the same guarantee can be made for fail-stop node faults within the network. As long as the current state of the node (buffers, MM, PE) is not lost, it can be restored when the node is repaired and the computation restarted without error. Given that information is not lost in the face of component failures, the state in which the network (or part thereof) blocks, even in the face of numerous faults, will not compromise the constructible serial schedule of global memory accesses guaranteed by local synchrony.

We can relax the assumption of fail/safe communication channels so that a simple handshaking protocol is in place on each channel. Such a protocol requires the receiver to send an acknowledgment to the sender when the access in transit has been received. This acknowledgment

is not sent unless the entire, correct access has been received (remember we assume that incorrectly transmitted accesses are detectable), and the sending node does not discard its copy of an access until the acknowledgment is received.

If the fault occurs before a given access is fully transmitted, the receiving node operates as if the access was never received. A node's action in this case is no different than it would be in the fail/safe system described above. If the fault occurs between transmission of an access and transmission of the acknowledgment, then the receiving node will continue to operate (including transmission of the access in question) until it is forced to block, due to the faulty channel. When the channel is restored, the sender, not having received the acknowledgment from the receiver, will send the same access again. The receiving node need only keep a record of the last access it has received on this line. If the newly received access is the same as the last, or if it breaks the sorted transmission rule, then it is discarded, having already been processed.

5.2 A Single-Level Fault-Tolerance System

Locally synchronous implementations can be designed to tolerate the permanent failure of a communication channel. Consider the system described above, without assuming that failed channels can be repaired. For a given computation to complete in the face of a permanent channel failure, without compromising local synchrony, those accesses normally transmitted through the faulty channel must be rerouted so that they still reach the 'other side' of the channel in the correct sequence. We will assume that a node is capable of detecting a faulty communication channel.

Again, this is possible because local synchrony guarantees the preservation of system state in the presence of a fault. The receiving node will block until input from the given (faulty) channel is received. We need only describe a system for passing those accesses along another path, effectively bypassing the faulty channel. We do this by providing an 'escape-hatch' in the buffers and hardware of switches in the network.

We assume that the hardware implementation provides alternate hardware communication paths from a given source to a given destination. For example, if line 00-01 in a binary 3-cube is

faulty, then a suitable alternate communication path would be lines 00-10, 10-11, and 11-01. Much of the literature is devoted to redundant path designs for various architectural topologies [TYZ85][BaB87][IIS82][VaR86].

When a node detects a faulty output channel, it simply redirects accesses that would normally be output on the faulty channel along another transmission path, toward the neighboring node along the faulty channel. These accesses are tagged with a temporary destination (the receiving node at the other end of the faulty channel) and are designated as *special* accesses. Special accesses are not bound by the normal conventions of local synchrony. They have transmission priority over ‘normal’ accesses: a node having a choice between a normal and a special access will route the special access first. Special accesses are segregated from the stream of normal accesses by the allocation of separate buffers in the switching hardware. In this way, special accesses can find their way around the alternate route even if all the normal access buffers along that route are currently full and progress has been halted. A diagram of a switch design is presented in figure 35.

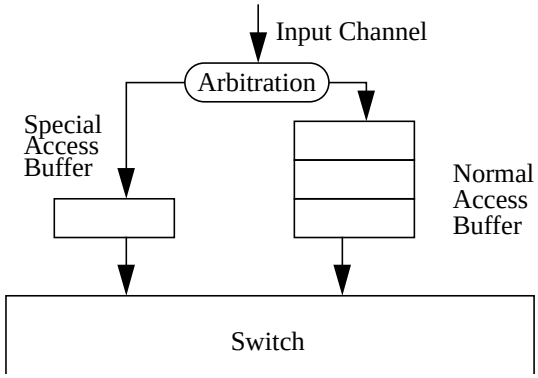


FIGURE 35. Design of a single-level fault tolerant switch input.

Upon receiving a special access, a node first checks its temporary destination. If the temporary destination is not the node itself, the access is sent on unchanged. If the temporary destination names the node itself (the node has a faulty input channel on which the special access was supposed to be transmitted), then the node processes the access in order, changing it back to a normal access when it is transmitted.

The special access is guaranteed to arrive before any later accesses are processed at the temporary destination because the temporary destination requires information on all inputs in order to conservatively choose its next action. As long as a special access has not arrived at its temporary destination, the temporary destination can make conservative processing decisions upon only the information it last received on the now-faulty channel. This information has an earlier timestamp than the special access, so any processing decision made by the temporary destination will also have an earlier timestamp. The special access is then guaranteed to arrive before any processing decision with equal or greater timestamp is made at the temporary destination, and will thus be processed in order there.

The deadlock freedom of the implementation is not affected, because alternate paths do not use resources normally allocated to regular communication paths, and special accesses thus cannot block normal accesses within the network. While the transmission of special accesses may delay the transmission of normal accesses, it cannot do so indefinitely.

In this way, the system can tolerate a minimum of one permanent faulty channel in the network. Such a system could tolerate any number of *non-conflicting* faults. Two faults conflict if their alternate routes share channels. This requirement generally limits the tolerance of faults in input (output) channels that are connected to a given node to half the total number of input (output) channels connected to that node.

As long as faults are non-conflicting, blocking and deadlock cannot occur, as the special accesses using any given path were all generated by a single node. If faults conflict, then the conservatism of the implementation may be compromised (out of order message processing) or deadlock can occur, and a rollback (with fault removal) or restart of the calculation must occur.

5.3 A Rollback Algorithm for Local Synchrony

The domino effect [Ran75] is the main obstacle to implementing rollback recovery in a distributed system. An element’s rollback to a checkpoint can cause other entities to roll back in a potentially unlimited chain. Randell pointed out that the domino effect can be eliminated if it is

possible to synchronize state saving checkpoints. Local synchrony allows absolute synchronization of checkpoints to be implemented without checkpoint communication and with no more synchronization overhead than is already present in the system.

In [TKT92], the authors define a consistent system state as a set of recovery points, one for each node within the system, such that restoring all recovery points will not cause any orphan (lost) messages. Our approach is to define a set of recovery points, or checkpoint, for all nodes within the system in which *no outstanding messages exist*, and thus no orphans are possible.

In this section, we present a procedure for checkpointing and rollback for computations taking place in a locally synchronous environment as described in Chapter 4. Such a procedure can be shown to be correct if it meets these conditions:

1. Checkpointed states are consistent system states.
2. Each system element rolls back to the checkpoint in such a way that local synchrony is maintained (the rollback is synchronized).

We will show that the procedure described below meets each of these conditions. Because each element (other than PEs) within a locally synchronous isotach system increments the logical time of a message by a single ltu when processing the message, we make the following distinction:

> **Definition:** The *current logical time* of a message is its current timestamp.The current logical time of an element is the current logical timestamp of the last message processed by that element.
>
> **Definition:** The *execute logical time* of a message is the current logical time of the message when it is executed at its target MM. The execute logical time of an element is the execute logical time of the last message processed by the element; e.g., if a message with current timestamp l is processed by element A and will be processed by an MM at time $l+i$, the execute logical time of element A is $l+i$.

We measure i as positive if the element processes messages on their way to the MMs, and negative if the element processes messages returning to PEs from the MMs.

We make this distinction in order to simplify later arguments concerning the correctness of our checkpointing and rollback procedure. While current logical time differences between elements or messages are a function of the distance between them, execute logical time differences measure the true logical time differences between elements or messages, based on the serial schedule of global memory accesses guaranteed by the local synchrony implementation. Each ltu mea-

sured in execute logical time (e.g., as processed at the MMs) is a barrier consistent with the sequential consistency and atomicity constraints of local synchrony (see Chapter 2).

Consider a real-time synchronous shared memory processor in which all memory request processing takes place during a certain period of each network cycle, and the response to any access sent during a cycle is guaranteed to return during the same cycle. Such a system reaches a consistent state at the end of each network cycle, because no accesses are outstanding and, therefore, none can be orphaned by saving state at each node. Note that switching elements are all empty at the end of a network cycle.

We define a checkpoint for the general local synchrony implementation (Chapter 4) and show that it describes a consistent system state. Because local synchrony is an asynchronous system (it is only logically synchronous), there is no guaranteed physical time when the state of the entire system is consistent. We can generate a consistent state, however, by synchronizing state saving at checkpoints in execute logical time. The checkpoint is reached at an element when the element processes a pre-specified token, called the checkpoint token. The global state saved at the checkpoint consists of:

1. The process state of each process when its PE has queued the checkpoint token (and before any accesses to be executed after the checkpoint have been generated); and
2. The state of each local memory and of global memory after all accesses queued before the checkpoint have been executed and responses returned.

The checkpoint includes the states of processes and local and global memory. Since no processes or memory reside at switches, the switches have no state and no action in the checkpoint procedure. We describe the state saving process at each element:

1. Switching elements - There is no state to save. There are no outstanding accesses or responses in the entire system at the logical checkpoint, and thus all switches appear to be empty.
2. MMs - Each saves its current state when the checkpoint token arrives on its input.
3. PEs - A PE begins a checkpoint by saving its process state (for simplicity, we assume that PE multiprocessing is not in effect) when the checkpoint token is queued at each PE output. The checkpoint is fully saved by saving the responses to all accesses made prior to the checkpoint (before the execute logical time specified by the checkpoint token). At each PE input, responses are saved until

the checkpoint token returns on that input (when the PE input has reached the checkpoint). In other words, the state includes the execution point of the PE and the responses to all accesses made before the execution point was saved.

We now show our checkpointing procedure meets the Correctness Condition 1 as stated above:

Lemma 5.1: The checkpointed state is logically consistent at all elements within the system.

Proof: By contradiction. Assume the checkpointed state is not consistent. Then one or more of the following is true: 1) some saved process state is inconsistent; 2) some access response that should have been saved in a local memory state was not saved; 3) some access response that should not have been saved in a local memory state was saved; or 4) some saved MM global memory state is inconsistent. We take each case separately. Case 1 is impossible because the processes initiate the checkpoint by queueing the checkpoint token. By assumption, each process saves its state after this action and before any other action. The atomicity and sequential consistency constraints described in Chapter 2 guarantee that a PE will not generate accesses to be executed after the checkpoint until it has, with certainty, generated all accesses to be performed prior to the checkpoint, and sent the checkpoint token. For case 2 to be true, the response to an access that should be saved must return after the checkpoint token. This is not possible because all accesses sent prior to the checkpoint token have execute timestamps less than the checkpoint token, and, by the conservative nature of local synchrony, their responses will arrive back at the PEs before the checkpoint token and thus be part of the saved state. The same argument may be applied to case 3, as any access sent after the checkpoint token has higher execute timestamp, and thus its response will arrive at the PE after the checkpoint token and not be part of the saved state. Case 4 is also not possible. By the conservative nature of local synchrony, the MMs will not accept the checkpoint token and save their state until they have processed all accesses with execute timestamp less than the checkpoint token, and before any access with execute timestamp greater than the checkpoint token. Each case is not possible. Contradiction. ■

The total saved state then includes the process states at the checkpoint and the state of local and shared memory after all accesses made before the checkpoint have been performed and responses received. Of course, because the checkpoint is logical instead of physical, there is generally no physical time when the system is actually in this state. Instead, the effect of the implied serial schedule of shared memory accesses is that a process within the system could observe such a state.

Checkpoints are set to occur at regular intervals of execute logical time rather than physical time, eliminating possible clock synchronization problems. The disadvantages of the traditional synchronized approach are the costs of checkpoint communication and synchronization delays during normal operation [BLL90] [Kim88]. Local synchrony requires no checkpoint communication and has already built synchronization overhead into the system in order to facilitate concurrent access to global memory.

In section 5.2, we showed that local synchrony was compatible with standard link and node failure hardware protocols for redirecting messages around a fault. It is beyond the scope of this work to consider hardware fault detection strategies or the faults that might generate the need for rollback. We assume that such fault detection systems are in place, and that process and memory states described below are saved to non-volatile, recoverable storage, which may be directly accessed by the PE or MM when rollback is necessary. We present here the LSR procedure used to roll back a computation to a previously checkpointed system state when a fault has occurred:

1. A fault is detected.

We assume that the detection strategy provides enough information to determine the latest checkpoint which is uncontaminated by the fault. We will call this the rollback checkpoint. We assume that the rollback checkpoint has been completely saved; i.e., all elements have execute logical time later than the checkpoint. For simplicity, we further assume that the recovery phase in the system is fault-free. If a fault is detected during the recovery phase, the fault detection system may regenerate the rollback request after the current rollback has concluded.

2. Each PE is informed of the rollback request and rollback checkpoint.

We assume that the fault detection system executes this step in one of two ways: (1) either it is able to inform PEs directly of a rollback request, or (2) it is able to place the request in a location in global memory (we assume for simplicity that it is possible to do this at the beginning of any given ltu, before any PE has accessed the location) that is accessed regularly (and regularly accessible) at predetermined logical times by each PE.

3. Each PE queues *rollback tokens* on each output.

After being notified of the rollback request, each PE immediately flushes its output queues of any messages and queues a *rollback token* on each output. Rollback tokens are special tokens which are distinguishable from normal tokens. Any element receiving a rollback token discards all other messages (including normal tokens) until rollback tokens appear on each input. The element then passes rollback tokens on each output, performs its initialization actions, and continues processing normally.

MMs may be apprised of the rollback checkpoint by the recovery system, in which case they immediately restore their state to the checkpoint and ignore all input until the receipt of rollback tokens on each input, at which time they pass rollback tokens on each output and continue processing normally. Alternatively, MMs may receive checkpoint information from the rollback tokens.

4. Migration occurs, if necessary.

We note here that an interval of logical time may be set aside for any migration activity necessary. This is only a general discussion of simple migration, and not an in-depth study of possible migration actions. If processes or memory must migrate, the original rollback request would include the requirement for a migration period. For a simple migration, this period would last at most 2 ltus (in execute logical time—for process migration between PEs, a WRITE followed by a READ; we assume memory migration occurs off line and PEs would need only to read a new memory map, which can be performed in a single ltu). During this period, process execution may be suspended, if necessary.

5. Processes resume execution from the checkpointed state. Other elements resume normal operation upon receipt of the rollback tokens.

The rollback tokens move across the network to the MMs and back to the PEs, acting as a barrier. Pre-barrier actions are ignored, and processing resumes in the checkpointed state after the barrier has passed. We now show our checkpointing procedure meets the Correctness Condition 2 as stated above:

Lemma 5.2: The LSR rollback algorithm guarantees that each system element rolls back to the checkpoint in such a way that local synchrony is maintained (the rollback is synchronized).

Proof: The rollback is synchronized by the rollback tokens. Rollback tokens have, in effect, a greater execute timestamp than any message in the system prior to the rollback request, but earlier execute timestamp than any message sent after rollback has occurred. Thus, all pre-rollback actions occur prior to the rollback, and all post-rollback actions occur after the rollback. Because rollback tokens are treated like tokens (when a rollback token is at the head of each input, all are consumed and rollback tokens are sent on each output), they constitute a barrier which synchronizes the rollback of each element in the system. The rollback meets Correctness Condition 2. ■

Finally:

Theorem 5.1: The LSR rollback algorithm guarantees correct rollbacks of parallel computations.

Proof: We have stated that the checkpointing and rollback procedure is correct if it meets the correctness conditions as stated above. Lemma 5.1 shows that the procedure meets Correctness Condition 1, and lemma 5.2 shows that the procedure meets Correctness Condition 2. ■

5.4 Checkpoint Validation

We now show that under certain constraints a local synchrony implementation using the LSR checkpointing and rollback procedure can limit the number of checkpoints which must be retained by a process. We note that throughout this section, we refer to logical times and time periods in the *execute* sense, as described in the previous section. For example, when we discuss the PE inputs as having all passed logical time *i*, we mean that each has execute logical time greater than *i*.

It is possible for a local synchrony implementation to checkpoint at the beginning of each ltu. Because of the loose synchronization inherent in local synchrony implementations, it is possible to bound the number of checkpoints which must be retained by a process. A checkpoint is *validated* when no rollback will occur to an earlier checkpoint. Once a checkpoint is validated, there is no need to save earlier checkpoints.

In order to discuss checkpoint validation, it is important to understand the notion of time lags within a local synchrony computation. Any element within the system may at any time have

progressed to a different logical time than any other element. If this is the case, a logical time lag exists between the two elements. Because of the conservatism and loose synchronization of local synchrony implementations, there are limits on the amount of lag possible between elements. For example, the limit of the time lag (either current or execute) between any two consecutive cells in an input queue is a single ltu. If each cell holds a token, then by conservatism the tokens must mark consecutive ltus. If each cell holds an access, the accesses are guaranteed to be in the same ltu.

A PE input covers an MM if responses from the MM arrive on the input . Before presenting our checkpoint validation arguments, we define the following:

> In the general local synchrony implementation, the difference between the execute logical times of any two PE inputs both of which cover a given MM is called the *execute input lag* between those PEs.

We make the following observation about execute lags within a local synchrony implementation:

> The execute input lag, L_{input}, between inputs covering an MM of any two PEs at a given physical time cannot exceed ds, where d is the depth of the network and s is the number of cells per input queue and the number of cells per output queue in a switch (presumed equal at all switches).

Consider two PE inputs (into separate PEs) which cover the same MM. There exists a switch between the PEs and the MMs where the streams of responses bound for the two PEs split. At this switch, the execute logical time of the message in the front of each input stream at the switch cannot differ by more than one ltu (one input holds a token beginbning ltu $i+1$ while the other holds a message to be processed during ltu i). By the conservative nature of local synchrony, it is impossible for either PE input to have later execute logical time than the current execute logical time of the switch. Since each message inbound to a PE input can change that input's logical time by at most one ltu (the reception of a token), the PE input's execute logical time cannot be less than the execute logical time of the switch by more ltus than the number of message slots along the shortest direct path between the input and the switch, which is limited by d. The maxi-

mum lag between two PE inputs will be smaller if their ‘split switch’ is fewer than *d* steps away. The argument for the PE outputs is similar.

We assume that a fault during ltu *i* will be discovered by the fault detection system within *k* ltus of the time all PEs have received responses to all accesses made up to and including ltu *i*, and that checkpoints occur every *m* ltus. We further assume that the recovery system can directly inform PEs of rollback requests. A checkpoint, c, is thus validated (guaranteed correct) at the validation ltu, V(c), when all PE inputs have progressed *k+1* ltus beyond the checkpoint.

A PE finishes a checkpoint *c* when it receives the checkpoint token on all of its inputs. A rollback request may be generated at any execute logical time up to ltu *c+k*. Since the PE inputs of any two PEs cannot have a execute lag of more than $max(L_{input})$, a PE knows that checkpoint *c* is validated after the PE’s inputs have passed the validation ltu*:*

$$V(c) = c+k+max(L_{input})+1 \quad \text{(EQ 19)}$$

Ltu *V(c)-1* is the last possible ltu at which a PE could be notified of a rollback request for a checkpoint prior to checkpoint c. When a PE reaches *V(c)* on its inputs, it is assured that all system entities have passed ltu *c+k+1*, which validates the checkpoint. If *m* is the checkpoint interval, then at any time, up to $(k+max(L_{input})+1)/m$ checkpoints will not yet have been validated. Once a checkpoint is validated, all previous checkpoints may be discarded. In this way, the system can guarantee the ability to roll back while keeping no more than two checkpoints in memory at any time, by making the checkpoint interval *m* equal to or greater than $max(L_{input})+k+1$.At least two checkpoints must be saved because an error may occur between the beginning and end of the checkpoint procedure. The number of stored checkpoints required for a given checkpoint interval *m* is $max(2, (k+max(L_{input})+1)/m)$.

Note that PEs can validate checkpoints without any global synchronization. Once a checkpoint is validated, either the PEs or the recovery system can cause the MMs to discard older checkpointed states.

5.5 Summary

In this chapter, we discussed some issues regarding fault tolerance in the general local synchrony implementation presented in chapter 4. Because logical synchronization is built into the local synchrony implementation, some traditionally difficult fault-tolerance problems are easily implemented in locally synchronous systems.

A local synchrony implementation is able to tolerate certain types of temporary faults without compromising its state. Permanent faults may also be tolerated using redundant path techniques in hardware, which have been proposed in the literature.

We presented the LSR procedure for checkpointing and rollback of a computation in the general local synchrony implementation, and proved it correct. Finally, we showed that under certain circumstances it is possible to limit the number of saved checkpoints through checkpoint validation.

In the next chapter, we present analytical models and simulation studies which predict the performance of locally synchronous systems in comparison with conventional systems.

6.0 Performance

In this chapter, we present the results of simulation and analytical studies of local synchrony implementations. We use mean value analysis in modeling local synchrony. Our analytical models predict the theoretical raw power performance of the implementations and agree closely with the raw power results of our simulation studies.

The simulation studies, performed jointly with Craig Williams [RWW92], compare the performance of local synchrony implementations with conventional implementations for different types of workloads, based on various levels of concurrency control requirements.

Section 6.1 presents an overview of the architectures modeled and the internal communication algorithms performed by each. Section 6.2 describes the analytical models we have developed and uses these models to generate performance predictions. Section 6.3 presents the results of our simulation studies. In section 6.4, we compare local synchrony to another approach to logical timestamp ordering for concurrency control, and present simulation data comparing the two approaches.

6.1 Architectures Modeled

The architectural model assumed in this study is a MIMD shared memory parallel processor based on a multi-stage switching network. We simulate two types of local synchrony architectures in the analytical study and simulate these and two similar conventional implementations in the simulation study. Each network is composed of 2x2 switches interconnected in the same baseline network topology. A diagram of a multistage interconnection network with this topology is shown in figure 36. The message transmission protocol is store-and-forward using a send-acknowledge protocol [REF87]. To allow the use of time-stepped simulation, we assume the networks are clocked; i. e., switches begin each cycle simultaneously. We believe the results are also applicable to self-timed networks.

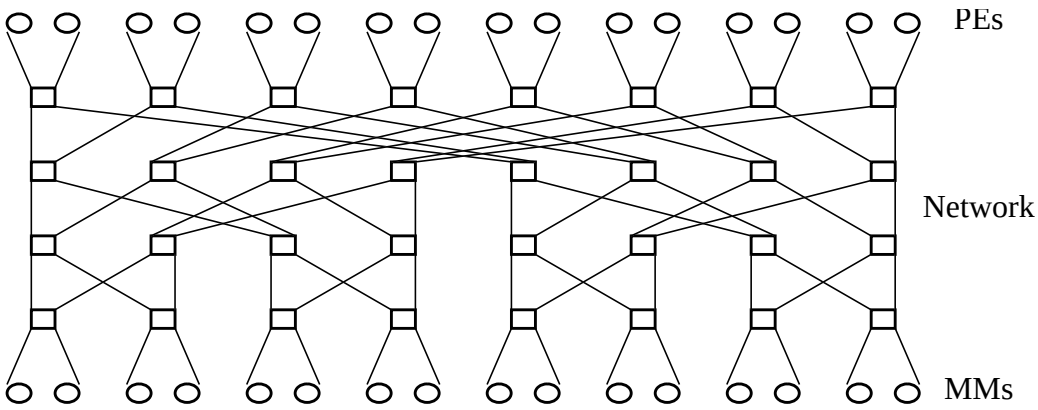


FIGURE 36. A 4 stage interconnection network.

The networks differ only in the design of the individual switches and in the algorithms the switches execute. Networks C1 and I1 are both composed of 2x2 crossbar switches with input and output buffer queues. Networks C2 and I2 use more sophisticated switches, called z-switches, that yield higher throughput at some cost in latency. We developed the z-switch [RWW92] to improve the throughput of the local synchrony network. When it proved successful, we translated it into a conventional switch design to enable us to compare the z-switch version of the local synchrony network(I2) to a conventional network with comparable routing advantages (C2). In the context of a conventional network, the z-switch is similar to switch designs with output buffers [KHM87] or internal buffers [KuJ84].

We assume switches in C1 and I1 have the same cycle time; i. e., we assume that the amount of time required in the absence of conflicts for a message to travel through a switch in C1 is the same as in I1. All performance data is measured in units of this cycle time. The switches in networks C2 and I2 are assumed to have twice the latency of those in C1 and I1.

6.1.1 Conventional network C1

The switches in network C1 consist of two input queues, two output buffers, and a routing unit that can route a message from either of the two inputs to either of the two outputs. Each buffer can hold a single message and each queue consists of one or more buffers. A diagram of the switch architecture appears in figure 37..

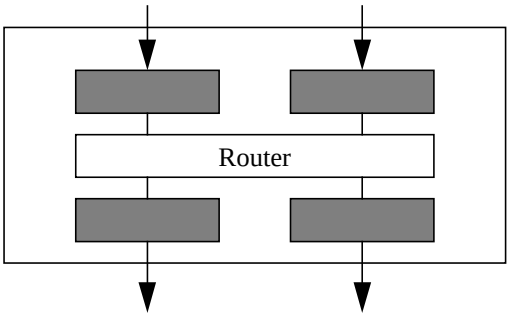


FIGURE 37. Simple switch design (C1 and I1)

The switching cycle of switches in network C1 consists of two steps: an internal communication phase, in which each switch attempts to route messages on its inputs to its outputs; and an external communication phase, in which each switch attempts to send the messages on its outputs to the inputs of switches at the next stage, using a send-acknowledge protocol. The internal communication and external communication phases at each switch are as follows:

Internal Communication Phase:

1. Choose an input randomly (whether its buffer is full or empty) such that each input is chosen with equal probability.
2. If the input holds a message, determine the output on which the message should be routed.
3. If the output is available, route the message; i.e., move it to the correct output.
4. Repeat steps 2 and 3 with the other input. (Note that a switch is capable of routing two messages in a single internal communication phase if each messages is bound for a different output.)

External Communication Phase:

1. For each non-empty output, send a copy of the message on the output to the next stage switch connected to the output.
2. For each input on which a new message arrives, if room in the input queue is available, add the message to the queue and send an ACK to the sender.
3. For each output for which an ACK is received, mark the output empty.

6.1.2 Conventional network C2

Network C2 is a high-throughput conventional network based on the z-switch. In network C2, switch outputs are decoupled from switch inputs to provide better throughput at some cost in network latency. Decoupling inputs from outputs prevents a blocked message from blocking another message merely because it arrives on the same input. In a z-switch, a message whose out-

put buffer is available can make progress even though a message arriving previously on the same input is blocked due to a full output buffer.

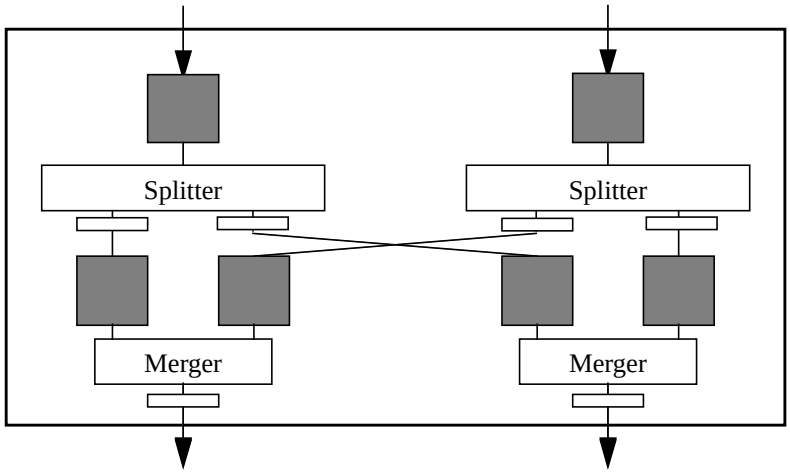


FIGURE 38. Z-switch high-throughput design.

Each switch element consists of four components: a “splitter” for each switch input and a “merger” for each switch output. Each splitter has one input queue and two output buffers, one for each merger. Each merger has two input queues, one for each splitter, and one output buffer. A diagram of the switch design appears in Figure 38.

As in C1, each switch cycle in C2 consists of an internal communication phase and an external communication phase. Each splitter and merger participates in both steps of the cycle:

Internal Communication Phase - Splitter:

1. If the input is empty, skip the internal communication phase.
2. Determine the output on which the incoming message should be routed.
3. If that output is available, route the message.

Internal Communication Phase - Merger:

1. If the output is full, skip the internal communication phase.
2. Randomly choose an input.
3. If the input holds a message, route it.
4. Otherwise, route the message, if any, on the other input.

External Communication Phase - Splitter:

1. For each non-empty output, send a copy of the message on the output to the merger connected to the output.

2. If a new message arrives on the input and room in the input queue is available, add the message to the queue and send an ACK to the sender.
3. For each output for which an ACK is received, mark the output empty.

External Communication Phase - Merger:

1. If the output holds a message, send a copy of the message to the next stage switch connected to the output.
2. For each input on which a new message arrives, if room in the input queue is available, add the message to the queue and send an ACK to the sender.
3. If an ACK is received for the output, mark the output empty.

The latency time for C2 is assumed to be two cycles because the splitter and merger each have one cycle latency time.

The two local synchrony networks simulated, I1 and I2, correspond to C1 and C2, respectively. In each local synchrony network the structure of the switches is similar to that of the switches in the corresponding conventional net, but the switch algorithm is different. The local synchrony switches apply the local synchrony algorithm described in chapter 3. The local synchrony algorithm requires that the switch route messages arriving in the same ltu in a given order. In local synchrony networks I1 and I2, each switch routes operations in non-decreasing order by “route-tag”. The “route-tag” of an operation is the ordered pair “(dest, source)”, where “dest” is the variable accessed by the operation and “source” is the pId of the source PE. Route-tags are lexicographically ordered.

Local synchrony networks carry two types of messages in addition to operations: tokens and ghosts. As described in chapter 2 and elsewhere, tokens are control signals that divide ltus. A ghost [Ran87] is a copy of an operation with a bit set to indicate it is not a real operation. In other respects, ghosts look like operations. In particular, ghosts have route-tags. When a switch sends an operation on one output it sends a ghost with the same route-tag as the operation on the other. A switch receiving a ghost knows that all further operations it receives on the same input in the same ltu will have a larger route-tag than the ghost. This knowledge may enable the switch to route an operation on its other input. Ghosts improve network performance and are necessary in some networks for deadlock freedom [ReW91].

6.1.3 Local synchrony network I1

In network I1 the switches have the same structure as the switches in C1; i.e., each switch has two input queues, two output buffers, and a router.

Each switch in I1 records the route-tag of the last message it routed as “last_tag”. The value of “last_tag” represents the best conservative guarantee the switch can make about the route-tag of the next message it will route. Initially, and at the beginning of each ltu, the value of “last_-tag” is reset to (-1,-1).

Each switch in I1 always chooses the minimum message for routing, i.e., the message with the minimum route-tag. Identifying the minimum message requires that both inputs hold a message. In identifying the minimum message, tokens are treated as having a route-tag greater than that of any other type of message and operations and ghosts are treated identically. If both inputs hold tokens, the minimum message is chosen arbitrarily. If only one input holds a token, the message on the other input is the minimum message. If neither holds a token, the minimum message is the message with the lowest route-tag.

Whenever a switch has no operation or token to route on an output, it routes a ghost. Ghosts take up only unused bandwidth and unused buffers. A ghost can always be overwritten by a newer message. Therefore an output buffer is “available” if it is empty or contains a ghost, and room in an input queue is “available” if the queue is not full or if the message at the tail is a ghost. The internal communication and external communication phases in I1 are as follows:

Internal Communication Phase:

1. Determine the input holding the minimum message.
2. If the minimum message is a token and both outputs are available, remove the tokens from both inputs, place tokens on both outputs, and reset last_tag. (Note that if the minimum message is a token then the other input also holds a token.)
3. Otherwise, set last_tag equal to the route-tag of the minimum message. If the minimum message is an operation, determine the output on which it should be routed. If the output is available, route the operation.
4. Emit a ghost with route-tag = last_tag on each available output.

External Communication Phase:

1. Same as the external communication phase in C1.

Network I1 has low throughput in relation to C1 because the switches in I1 can route only one message per cycle. A switch in a local synchrony network that can see only the operations at the head of each of its inputs has insufficient information to route more than one operation per cycle given the requirement that a switch must route operations in sorted order. The z-switch used in network I2 was designed to overcome this throughput limitation.

6.1.4 Local synchrony network I2

The switches in network I2 have the same structure as those in C2. However, the units called splitters in C2 are called multiplexors in I2, to better represent their actual function. The function of a multiplexor is to copy each operation received on its input onto both outputs. The function of a merger is to route the message streams received from both multiplexors in order by route-tag. Each multiplexor and merger records the last_tag as in I1. The internal communication and external communication phases in I2 are as follows:

Internal Communication Phase - Multiplexor:

1. If the input holds a token and both outputs are available, consume the token, put tokens on both outputs, and reset last_tag.
2. Otherwise, set last_tag equal to the route-tag of the message on the input. If the input holds an operation, determine the output on which it should be routed, and move the message to that output if it is available.
3. Emit a ghost with route-tag = last_tag on each available output.

Internal Communication Phase - Merger:

1. If the output is not available, skip the internal communication phase.
2. Determine the input holding the minimum message.
3. If the minimum message is a token, remove the tokens from both inputs, place a token on the output, and reset last_tag.
4. Otherwise, set last_tag equal to the route-tag of the minimum message and, if the minimum message is an operation, move it to the output.
5. If the output is available, emit a ghost with route-tag = last_tag.

External Communication Phase - Multiplexor:

1. Same as the external communication phase for the splitter in C2.

External Communication Phase - Merger:

1. Same as the external communication phase for the merger in C2.

Local synchrony networks, as simulated, are simple variations of the networks described here, modified to allow piggybacking of tokens. Piggybacking of tokens occurs when a switch knows that a token follows the current message, and sends the message with a ‘token bit’ set, instead of sending the message followed by a token. Piggybacking thus reduces traffic in the network.

Section 6.2 presents analytical models for networks I1 and I2. Section 6.3 presents simulation studies comparing all four networks.

6.2 Analytical Models

We present several analytical models of the performance of local synchrony implementations on both the I1 and I2 networks. Our specific target is the Banyan communication network which connects n processors to n memory modules with a network consisting of $\log_2 n$ stages of n/2 2-input, 2-output switches.

A locally synchronous switch differs from more conventional switches by the way it chooses packets for transmission through the switch. Each packet carries a unique timestamp, which is used to determine the order of transmission through the switch. The input streams of packets into the network are guaranteed to be ordered by increasing (not necessarily monotonically) timestamp, and the switches must guarantee the same ordering for their outputs.

This transmission procedure requires that a switch receive timestamp information on each of its inputs before making an output decision. Such a protocol has no direct analog in queueing theory, as arrival rates have no effect on the order in which packets are serviced. We have chosen to design our models based on a simpler probabilistic approach.This allows us to accurately define the interactions between packets and buffers, although the memoryless nature of the approach and its assumption of independence among buffers within a switch may adversely affect the models’ predictions.

Our work also differs from traditional performance modelling for communication networks because it employs several different types of packets in the operation of local synchrony. Ghost messages carry only timestamp information, which serves to enhance network efficiency; tokens delineate the boundaries between logical time ltus; and combined packet-tokens contain both a packet and a ‘piggybacked’ token.

In order to make our work tractable we assume that the network operates synchronously, with timing measured in switch cycles which consist of an internal communication phase, during which packets are actually routed, and an external communication phase, during which packets are transferred between switches (or, in the case of the I2 design, between switch sub-components). We also limit our approach to measuring the raw power of the networks. Raw power measurements consist of a load of singleton accesses applied on each input. No sequential consistency or atomicity constraints are applied other than those guaranteed by the modeled networks.

We present the equations governing only the simplest model of performance and our final models, which take into account all discussed extensions. This is done for brevity and because each extension is represented in the final models.

6.2.1 A Probabilistic Model for Switch I1

We first present a performance model for a switch with both input and output buffers (in our simulation study, the conventional and local synchrony versions of this switch are denoted C1 and I1, respectively). The switch consists of two inputs with packet buffers for at least one packet, a switch router, and two outputs with packet buffers. Figure 39 shows the switch topology.

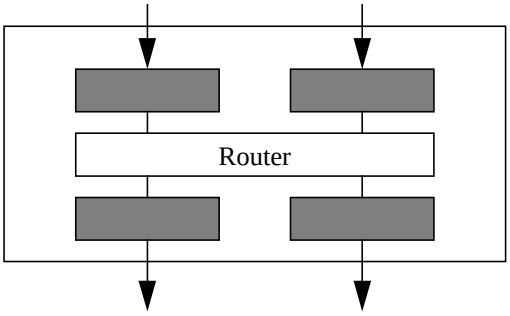


FIGURE 39. Switch I1 architecture.

We can apply Jenq's probabilistic approach [Jen83] to this type of architecture. This is the simplest model for performance of this type of switch, and it does not consider either the packet transfer protocol inherent in locally synchronous operation or the different types of local synchrony packets.

The packet transfer protocol is first-come, first-served (FCFS), with non-deterministic choice in the case of packet collision. A packet will be chosen for forwarding if the other input buffer is empty. If both inputs are full, either packet will be chosen first with equal probability. Once the first packet has been routed, the router attempts to route the second packet. In this way it is possible for the router to route two packets in a single cycle. Jenq presents a probabilistic approach to analytical modeling of equidistant, synchronous, single-buffered packet switching networks. We extend this approach to the C1 architecture. Jenq's initial assumptions are:

1. The packet load on the inputs to the network is uniform.
2. The packets arriving at each input are destined uniformly.

With these simplifying assumptions, the state of a given switch is statistically identical to that of any other switch in the same stage of the network. While the states of input (or output) buffers in the same switch are not generally independent, Jenq found that making this assumption simplified the model with only minimal adverse effects on the ability of the models to predict performance.

The following notation applies to our extension of Jenq's approach to the I1 architecture (note that only 'normal' packets are represented in the model):

n - The number of stages in the communication network.

i - Denotes the state at the beginning of the switch cycle; i.e., prior to the internal communication phase.

f - Denotes the state at the end of the switch cycle; i.e., after the internal communication phase, but before the external communication phase

$in^{i}_{j}[k,t]$ - The probability that a given input buffer at stage k has j packets at the beginning of cycle t.

$out^{i}_{j}[k,t]$ - The probability that a given output buffer at stage k has j packets at the beginning of cycle t.

$in^f_j[k,t]$ - The probability that a given input buffer at stage k has j packets after the internal communication phase during cycle t.

$out^f_j[k,t]$ - The probability that a given output buffer at stage k has j packets after the internal communication phase during cycle t.

Q[k,t] - The probability that a packet is ready to come to an input buffer at stage k at time t. In other words, the load on an input buffer in stage k at cycle t.

R[k,t] - The probability that a packet in an output buffer at stage k will advance at cycle t, given that the buffer is full.

We assume single buffers for this model, j = {0,1}. Q_{in} is the input load applied to each input of the network. In order to measure the power of the network itself, we do not take into account the effects of message queueing prior to network acceptance. We apply Q_{in}=1.0 in this work. Our interest in analytical modelling is twofold: to compare the theoretical performance of the conventional and local synchrony systems, and to validate the simulation data presented later. With these goals in mind, we use Q_{in}=1.0 to measure maximum predicted throughput and delay. We present equations used to predict the performance of the networks we consider. We present equations first, followed by English descriptions of what the equations mean. The following equations describe the state of the switches in each stage of the network.

$$Q[k,t] = \begin{cases} Q_{in} & (k=1) \\ out^f_1[k-1,t-1] & (k>1) \end{cases} \quad \text{(EQ 20)}$$

The load applied at an input to the first stage is the input load, Q_{in}. At any other stage, the load applied is the probability that the output buffer (at the previous stage) which is connected to the input held a packet prior to the external communication phase of the previous cycle.

$$in^i_1[k,t] = in^f_1[k,t-1] + in^f_0[k,t-1]*Q[k,t] \quad \text{(EQ 21)}$$

$$in^i_0[k,t] = 1 - in^i_1[k,t] \quad \text{(EQ 22)}$$

A given input buffer contains a packet at the beginning of the internal communication phase if the buffer contained a packet at the end of the previous internal communication phase (packets cannot move from inputs during the external communication phase), or if the buffer was empty but a new packet was transferred to it during the external communication phase. The buffer is empty if it is not full.

$$out^{i}_{0}[k,t] = out^{f}_{0}[k,t\text{-}1] + out^{f}_{1}[k,t\text{-}1]*R[k,t] \quad \text{(EQ 23)}$$

$$out^{i}_{1}[k,t] = 1 - out^{i}_{0}[k,t] \quad \text{(EQ 24)}$$

A given output buffer is empty at the beginning of a cycle if it was empty before the previous external communication phase, or if it contained a packet that was able to move forward during the previous external communication phase. The buffer contains a packet if it is not empty.

$$in^{f}_{0}[k,t] = in^{i}_{0}[k,t] + in^{i}_{1}[k,t]^{2}*out^{i}_{0}[k,t]*0.75 + in^{i}_{1}[k,t]*in^{i}_{0}[k,t]*out^{i}_{0}[k,t] \quad \text{(EQ 25)}$$

$$in^{f}_{1}[k,t] = 1 - in^{f}_{0}[k,t] \quad \text{(EQ 26)}$$

An input buffer is empty at the end of the internal communication phase (before the external communication phase) if it was empty at the beginning of the internal communication phase (line 1), or if it contained a packet that was able to be routed during the internal communication phase (lines 2 and 3). The chance of a packet advancing if both inputs are full (line 2) is 0.75 because the packet may be chosen first (0.5 chance); or if it is not chosen first, the packets may each be destined for different outputs (0.25 chance), in which case both can advance (if output buffers are free). Since switches in this model exhibit conventional operation, a packet will automatically advance if the other input buffer is empty (and the packet's destined output buffer is free—line 3). The buffer contains a packet at the end of the internal communication phase if it is not empty.

$$out^{f}_{1}[k,t] = out^{i}_{1}[k,t] + out^{i}_{0}[k,t]*(0.75*in^{i}_{1}[k,t]^{2}+2*0.5*in^{i}_{1}[k,t]*in^{i}_{0}[k,t]) \quad \text{(EQ 27)}$$

$$out^{f}_{0}[k,t] = 1 - out^{f}_{1}[k,t] \quad \text{(EQ 28)}$$

An output buffer contains a packet at the end of the internal communication phase (before the external communication phase) if it contained a packet before the internal communication phase, or if the buffer was empty and a packet was sent to it. A packet is sent if both inputs have packets (there is only a 0.25 chance that both packets will be destined for the other output) or if only one input holds a packet (there are two possible states in which this is true) that is destined for the output buffer in question (a 0.5 chance). The buffer is empty if it is not full.

$$R[k,t] = \begin{cases} in^{f}_{0}[k+1,t] & (k < n) \\ 1 & (k=n) \end{cases} \quad \text{(EQ 29)}$$

The probability that a packet in an output buffer at the end of the internal communication phase will move forward during the external communication phase of the current cycle is the probability that the input buffer (at the next stage) that is connected to the output is empty at the same time. The probability is 1 for the last stage, because we assume that network outputs will always accept.

We expect this system of equations (20-29) has a steady state. Given that, values on the left-hand side of the equations will converge upon iteration to time independent values that may be used to generate predictions for the network's throughput and delay for the given network input load. Each of the models we present converges to a steady state. In general, we find that convergence occurs within a small number (<100) iterations of the calculations.

The throughput, s (equation 30), is the probability that a packet remains in a switch output buffer after the internal communication phase and that this packet will advance during the external communication phase. This probability will be the same for each stage of the network at steady state.

$$s = out^{f}_{1}[k]*R[k] \quad \text{(EQ 30)}$$

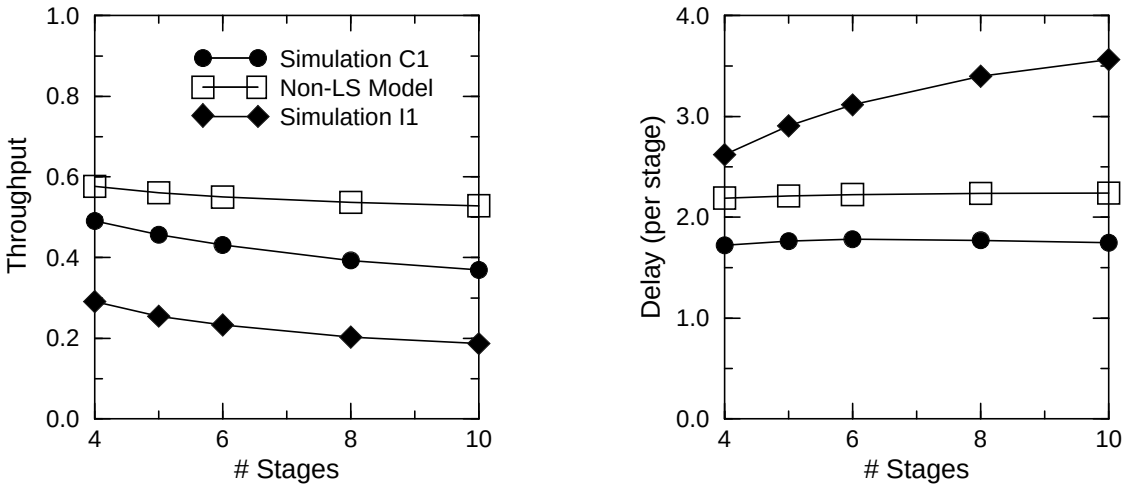
Delay (equation 31) is measured as the average number of cycles a packet or combined packet-token will spend in both input and output buffers at each stage of the network. At steady state, for example, R[k] gives the probability that a packet or packet-token will advance from an output buffer at stage k in one cycle. The inverse of R[k] is the average time a packet or packet-token will spend in an output buffer at stage k in cycles.

The second term (within the brackets in equation 31) is the time spent in the input buffer of the switch. The denominator is the probability that a packet will advance from the input buffer to the output in a given cycle, given that the input buffer holds such input. The output buffer to which the packet is bound must be free. We compensate for the fact that the last cycle spent in an

input buffer is also the first cycle spent in the output buffer by subtracting one cycle from the delay at each stage.

$$d = \frac{1}{n} \times \sum_{1}^{n} \left[\frac{1}{R[k]} + \frac{1}{(in_0^i[k] + 0.75in_1^i[k])\, out_0^i[k]} - 1 \right] \quad \text{(EQ 31)}$$

In graphs 6.2.1, we show the predicted performance of this model against the indicated performance (from simulation studies presented later in this chapter) of switch I1 in both conventional and local synchrony operation. As expected, the model is a better predictor of conventional performance, although it does not predict even the performance of network C1 with great accuracy. Compared to the simulation of C1, the model overestimates throughput by approximately 40% and delay by approximately 13% for a network of ten stages.



GRAPHS 6.2.1 - Non-LS model vs. normal and local synchrony simulations.

6.2.2 Extending the Model to Local Synchrony

In order to extend the model presented in section 6.2.1 so that it can better predict the performance of locally synchronous operation, we must make several extensions. The first and most obvious of these is to rewrite the model so that it considers local synchrony timestamp comparison for packet choice as in architecture I1, instead of FCFS with non-deterministic choice in the case of packet collision (architecture C1). Section 6.2.3 discusses the necessary changes, and presents data gained by applying these changes to the model.

Later extensions include representing tokens and combined packet-tokens in the model and introducing the idea of *information flow*, which limits the model to a more realistic interpretation of how ghost message information enhances performance in the model.

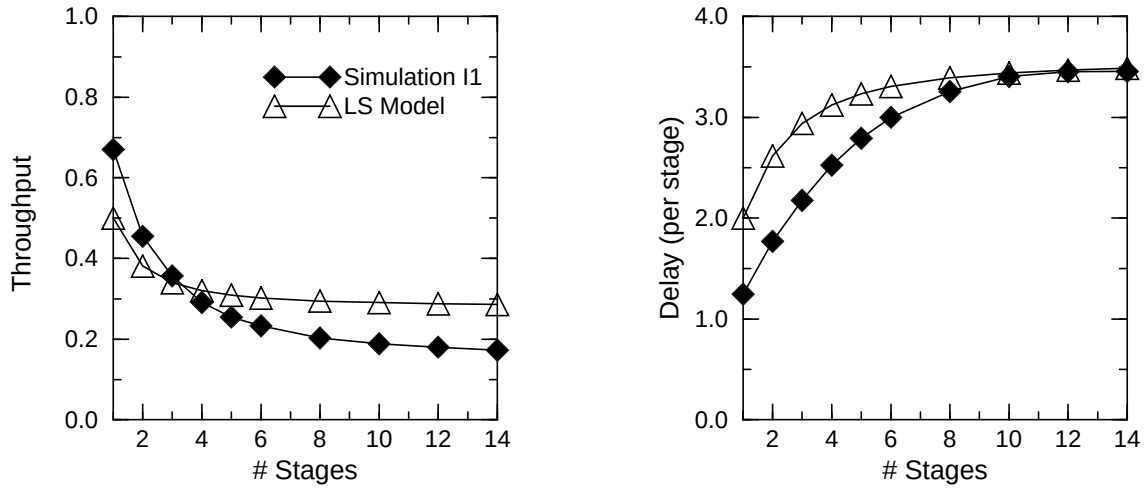
6.2.3 Locally Synchronous Timestamp Comparison

The model presented in section 6.2.1 demonstrates the conventional operation of a switch in network C1. During each cycle, the switch randomly chooses one of its inputs. If the input contains a packet, the switch transfers this packet to the relevant output buffer if that buffer is empty, then tries the same operation on the unchosen input. It is therefore possible for the switch to move two packets from input to output buffers in the same cycle, and a switch may transfer a packet if the other input is empty.

Local synchrony (network I1) requires the switch to order by logical timestamp the stream of packets exiting on each output. This means that the switch cannot pass a packet unless that packet's timestamp is less than the latest timestamp to arrive on the switch's other input. When two packets collide at a switch, therefore, only one can move forward during the current cycle. In the general local synchrony implementation, a packet cannot be forwarded if it is in an input buffer at a switch whose other input buffer is empty. The use of ghost messages to disseminate latest timestamp information, however, allows packets a 50% chance of advancing even if the other input to the switch is empty.

Modelling locally synchronous operation in this way requires a second assumption about the timestamps of packets: in any timestamp comparison, there is an equal chance that either timestamp will win the comparison. This assumption is not generally true, because there is dependence among the timestamps arriving on any given input. Once a packet loses a comparison, for example, it has a higher chance of winning the next comparison based on the difference between the winning and losing timestamps. Our assumption greatly simplifies the model by making it memoryless. We shall demonstrate that we can generate good predictive results even with this assumption.

Ghost message information plays an important role in the LS model. Switches are designed to send ghost messages, which propagate the latest timestamp information, on unused outputs during each cycle. Thus, an ‘empty’ buffer in the LS model actually contains timestamp information that can be compared to the timestamp of a packet in the opposite input buffer. In this way, packets may be transferred through a switch even when no other packet has come to the switch’s other input. Because the ghost message information might win the comparison, however, there is only a 50 percent chance that a packet will advance in this situation, given that the output buffer to which it is destined is empty (the same situation in the conventional network guarantees that the packet will advance).



GRAPHS 6.2.2 - LS-P model vs. local synchrony simulation

The LS-P model takes these differences into account. The equations are modified to reflect the local synchrony algorithm. Since each switch must send ghost message information to any unused output buffer during each cycle, a normally empty input buffer can be assumed to contain ghost message information. This allows a comparison and possible transfer of a packet in the other input buffer. Graphs 6.2.2 show the performance of the LS-P model compared to simulation data for locally synchronous operation of switch I1.The model seems to be a good predictor of delay for larger numbers of stages, but predicts significantly greater throughput for the same delay (greater than a 50 percent difference for a network of fourteen stages).

6.2.4 Information Flow

We would expect the LS model to produce higher throughput and higher delay than the simulation: higher throughput because the LS model does not take into account tokens, which consume buffer space without adding to throughput; and higher delay because a given packet in the analytical model may indefinitely lose the comparison at a given switch. In the raw power simulation of switch I1, a token is inserted into the input stream after every packet. This limits the number of times a given packet can lose the comparison to k-1 at a given switch, where k is the number of processors that can send packets through the switch.

The actual chances of a packet losing more than one or two times at a given switch are small in the simulation. Since each input stream consists of packets with destinations uniformly distributed, we would expect that a uniform distribution would hold over all inputs to some degree. In that case, any given switch would only process a few packets in an ltu, so the chance of losing more than a few times would be small. Of course, at later stages in larger networks, this artificial uniformity of the distribution would be less and less apparent, as accesses which may be uniformly distributed but are still bound for the same MMs converge. This convergence can cause long lines of accesses with which another access might lose the timestamp comparison.

The LS model generally meets these expectations in its predictions, but indicates much higher throughput with almost equal delay for larger networks. The model is, in fact, too optimistic regarding comparisons between packets and ghost message information. Ghost message information that won a previous comparison but was blocked by full output buffers, or that is identical to previous information on the same input, generally will not help a packet advance, but will instead win the comparison itself. Thus, the model will be a better predictor of performance if we can approximate whether an empty input buffer contains ‘new’ or ‘old’ ghost message information.

One way to make this approximation without adding memory to the model is to use *information flow.* The information flow to a stage in the communication network may be defined as the amount of ‘new’ timestamp information the stage receives in a given cycle. New information is

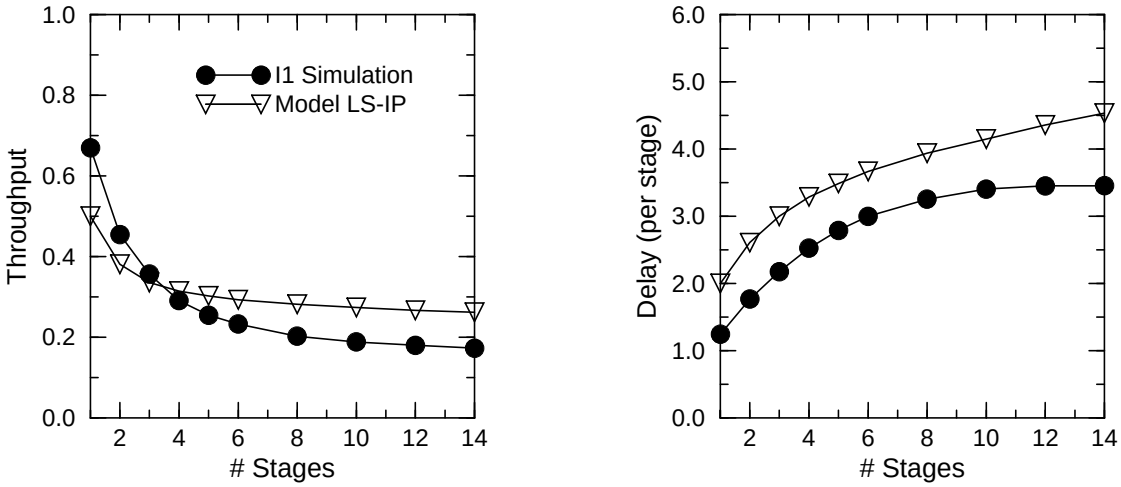
propagated by packets which are routed to output buffers and then forwarded to the next stage during the same cycle. Any ghost messages generated by these packets are guaranteed to contain new timestamp information. Ghost messages generated by switches in which accesses are blocked awaiting transfer or routing are generally ‘old’ information, because they are merely copies of ghost messages sent earlier.

We can approximate the information flow from stage i to stage i+1 in the network by considering the information flow into stage i and its maximum output of new information, given the input flow. The information flow into stage 1 of the network is the input load. Average maximum information flow for stage i can be measured by applying the information flow into stage i-1 to a single-stage network. The steady-state throughput of this network is the average maximum information flow into stage i. Since each packet that leaves a stage also can generate a ghost message on another output, information flow measures both ‘new’ packets and ‘new’ ghost messages. The actual throughput of an n-stage network may be greater than the information flow from any given stage. This is because some of the throughput consists of packets that have been blocked, and thus do not generate any new information.

Information flow modifies the LS model by allowing normal comparison between ghost messages and packets only until ghost message concentration equals the information flow for that stage. If the ghost message concentration (as measured by the probability that an input buffer is empty) is higher than the information flow, then the difference between the two is ‘old’ ghost information. We assume for simplicity that ‘old’ ghost information always wins the comparison with a packet, and thus our information flow factors effectively reduce throughput and increase delay. The use of information flow in this way is effective for high input loads, where the concentration of packets in the network is large. As the probability that a ghost message will be compared to another ghost message increases, the effectiveness of our approach decreases, because the comparison of two ghost messages could generate new information which is not taken into account in

our model. For full loads, which best represent the raw power of the network, we find that the use of information flow increases the precision of the model's predictions.

Graphs 6.2.3 show performance predictions for the LS-IP model (the packet model with information flow factors) against the simulation data for switch I1. Predictions for throughput are slightly reduced, and delay is consistently higher than the simulation. Model LS-IP acts as an upper bound on both throughput and delay for larger networks. Its prediction of throughput is still almost 50 percent too optimistic for a fourteen-stage network, its predicted delay also overestimates by over 30 percent, which is more in line with expected results for the model.



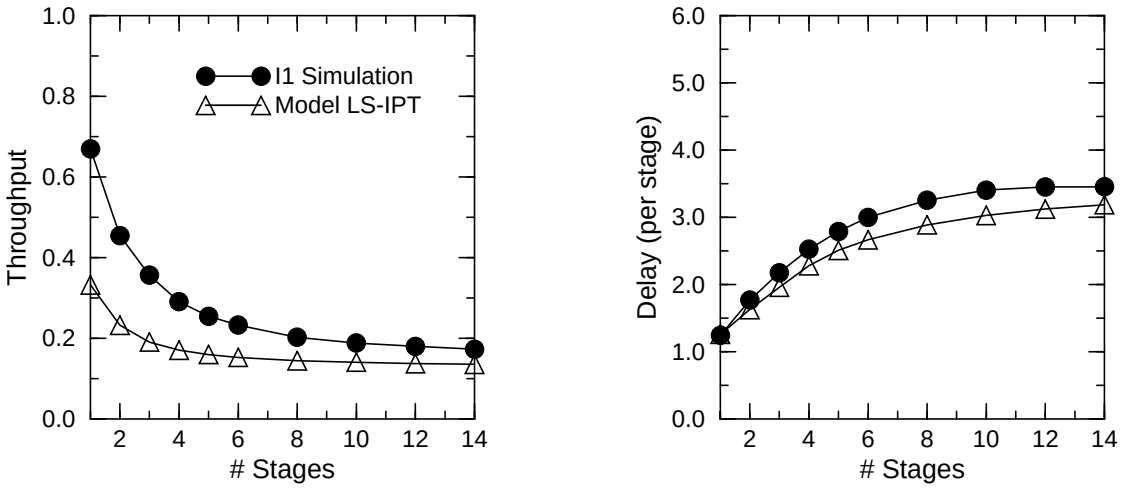
GRAPHS 6.2.3 - LS-IP model vs. local synchrony simulation

6.2.5 Representation of Tokens

In order to generate better prediction of performance, we must further refine our model. The predictions of model LS-IP give upper bounds for throughput and delay. We next generate a model that gives lower bounds (for larger networks) on throughput and delay by explicitly representing the tokens that delineate ltus. These tokens require buffer space in the network that would otherwise be allocated to packets, but they do not contribute to the throughput of the network. As a result we would expect a model with tokens to exhibit throughput that is less than in the simulation. A packet will always be chosen over a token, however, so a lower concentration of packets coupled with the existence of tokens should also significantly reduce delay, because the presence

of tokens in the network virtually eliminates the possibility that a packet will be blocked indefinitely.

In order to represent tokens, we add a third state to the state space for a buffer within the network. In this new model, LS-IPT, a buffer may be empty (ghost information only), contain a packet, or contain a token. The input load is modified to include both the probability that a token will arrive and the probability that a packet will arrive (the sum of these probabilities cannot be more than 1 and both must be between 0 and 1; for this discussion we use $Q_T = Q_P = 0.5$ for a full load, which emulates the simulation load). A packet always wins a collision with a token, and a token cannot advance until both input buffers hold tokens and both output buffers are free, in which case both tokens are removed from the inputs and a token is transferred to each output buffer.



GRAPHS 6.2.4 - Model LS-IPT vs. local synchrony simulation

In graphs 6.2.4, performance data for model LS-IPT is presented with data for the local synchrony simulation of switch I1. This model, in fact, gives a lower bound on throughput and delay for this implementation of local synchrony. Model LS-IPT seems to be a good predictor of lower bound performance of the I1 design for larger networks. The delay predictions are off by at most 14 percent, while throughput is underpredicted by approximately 28 percent for a fourteen-stage network. We expect the delay prediction to always be lower than the simulation data, since a

computation of predicted delay for twenty stages yields a lower delay (nearly identical to the model's prediction for fourteen stages) than the simulation shows at fourteen stages.

6.2.6 A Representative Model

Finally, we present a representative analytical model for the I1 implementation of local synchrony. In order to do this, we must add the existence of combined packet-tokens to the LS-IPT model to form the LS-IPTC model. This model is the closest approximation of the I1 implementation of local synchrony that we have explored using this modelling technique.

A combined packet-token consists of a packet with a 'piggybacked' token. This configuration has a specific ordering: packet then token. The input load applied now consists of combined packet-tokens only (as is the case in the simulation of network I1, which we are modeling here).

A given switch can separate the packet from the token if necessary, or combine a token with a packet in an output buffer to form a combined packet-token. Model LS-IPTC would be expected to closely approximate the performance of an actual implementation of local synchrony on a network of I1-type switches. Note that the use of combined packet tokens can, in some cases, allow the switch to make two transfers during the same cycle. For example, when a packet and a combined packet-token collide, if the timestamp of the combined packet-token (the timestamp of the packet in the combination—the token marks the end of the ltu in which the packet will be executed) wins the comparison, then the switch will split the combined packet-token and transfer the packet. At this point, the switch can also transfer the singular packet, if it is bound for the other output buffer and that buffer is empty.

We now present model LS-IPTC, a representative model for the performance of a local synchrony implementation using switch I1. A buffer may be in any of four states: in state 0 a buffer is empty (only ghost message information is present); in state P a buffer holds a packet; in state T a buffer holds a token; and in state C a buffer holds a message with a piggyback token. We use the following notation:

n - The number of stages in the communication network.

i - Denotes the state at the beginning of the switch cycle; i.e., prior to the internal communication phase.

f - Denotes the state at the end of the switch cycle; i.e., after the internal communication phase, but before the external communication phase

$in^i_j[k,t]$ - The probability that a given input buffer at stage k is in state j at the beginning of cycle t.

$out^i_j[k,t]$ - The probability that a given output buffer at stage k is in state j at the beginning of cycle t.

$in^f_j[k,t]$ - The probability that a given input buffer at stage k is in state j at the end of cycle t (before the external communication phase).

$out^f_j[k,t]$ - The probability that a given output buffer at stage k is in state j at the end of cycle t.

NG[k,t] - The probability that an input buffer contains 'new' ghost information at the beginning of cycle t. This probability is generated using $in^i_0[k,t]$ and information flow data (IF[k]) for this model as discussed in section 7.2.2.

$Q_j[k,t]$ - The probability that j, a packet, token, or combined packet and piggyback token is ready to come to an input buffer at stage k at time t. In other words, the load (of the specified packet type) on an input buffer in stage k at cycle t.

R[k,t] - The probability that a packet or combined packet and piggyback token in a merger output buffer at stage k will advance to the next stage during cycle t, given that the buffer has a packet or combined packet and piggyback token.

The following equations describe the relationships between the buffers in the system:

$$in^i_P[k,t] = in^f_P[k,t-1]*(1-Q_T[k,t-1]) + in^f_0[k,t-1]*Q_P[k,t-1] \quad \text{(EQ 32)}$$

An input buffer contains a packet at the beginning of the cycle (before the internal communication phase) if it contained a packet before the previous external communication phase and the buffer was not sent a token (which would create a combined packet-token), or if the buffer was empty before the previous external communication phase and was sent a packet during that phase.

$$in^i_T[k,t] = in^f_T[k,t-1] + in^f_0[k,t-1]*Q_T[k,t-1] \quad \text{(EQ 33)}$$

An input buffer contains a token at the beginning of the cycle if it contained a token before the previous external communication phase, or if it was empty and was sent a token.

$$in^i_C[k,t] = in^f_C[k,t-1] + in^f_0[k,t-1]*Q_C[k,t-1] + in^f_P[k,t-1]*Q_T[k,t-1] \quad \text{(EQ 34)}$$

An input buffer contains a combined packet-token at the beginning of the cycle if it contained one before the previous external communication phase, or if it was empty and a combined packet-token was sent, or if it contained a packet and a token was sent to form a combined packet-token.

$$in^{i}_{0}[k,t] = 1 - in^{i}_{P}[k,t] - in^{i}_{T}[k,t] - in^{i}_{C}[k,t] \quad \text{(EQ 35)}$$

An input buffer is in the empty (or ghost message) state at the beginning of the cycle if it is not in any other state.

$$NG[k,t] = MIN(in^{i}_{0}[k,t], IF[k]) \quad \text{(EQ 36)}$$

We approximate the concentration of ‘new’ ghost message information in empty buffers by comparing the concentration of empty buffers with the information flow to that stage. We assume that if the concentration of empty buffers is higher than the information flow, then the difference between the two represents ‘old’ ghost message data, which will win any comparison and thus does not help the movement of packets (although the movement of the ghost message information could help packets at later stages in the network). ‘New’ ghost message information has an equal chance of winning comparisons with packets.

$$out^{i}_{0}[k,t] = \begin{cases} 1 & (k=n) \\ out^{f}_{0}[k,t-1] + (out^{f}_{P}[k,t-1]+out^{f}_{C}[k,t-1])*R[k,t-1] + out^{f}_{T}[k,t-1]*(in^{f}_{0}[k+1,t-1]+in^{f}_{P}[k+1,t-1]) & (k<n) \end{cases} \quad \text{(EQ 37)}$$

An output buffer is empty at the beginning of the cycle (before the internal communication phase) if it is in the last stage of the network (the network outputs always accept). Otherwise, the buffer is empty if it was empty before the previous external communication phase, or if it contained a packet or combined packet-token that was able to move forward, or if it contained a token that was able to move forward. A token can be sent to an empty buffer or to a buffer containing a packet to form a combined packet-token.

$$out^{i}_{T}[k,t] = \begin{cases} 0 & (k=n) \\ out^{f}_{T}[k,t-1]*(in^{f}_{T}[k+1,t-1]+in^{f}_{C}[k+1,t-1]) & (k<n) \end{cases} \quad \text{(EQ 38)}$$

An output buffer contains a token at the beginning of the cycle if it is not in the last stage of the network (we assume that network outputs can always accept any output) and if it contained a token prior to the previous external communication phase that was not able to move forward during that phase (a token may only move forward if the buffer to which it is to be sent is empty or contains a packet).

$$out^{i}_{C}[k,t] = \quad out^{f}_{C}[k,t-1]*(1 - R[k,t-1]) \qquad \text{(EQ 39)}$$

An output buffer contains a combined packet-token at the beginning of the cycle if it contained one prior to the previous external communication phase and if the combined packet-token was not able to move forward during that phase.

$$out^{i}_{P}[k,t] = \quad 1 - out^{i}_{T}[k,t] - out^{i}_{C}[k,t] - out^{i}_{0}[k,t] \qquad \text{(EQ 40)}$$

An output buffer contains a packet at the beginning of the cycle if it is not in any other state.

$$\begin{aligned} in^{f}_{0}[k,t] = \quad & in^{i}_{0}[k,t] + \\ & in^{i}_{P}[k,t]*out^{i}_{0}[k,t]*(0.5*(in^{i}_{P}[k,t]+NG[k,t]+in^{i}_{C}[k,t])+in^{i}_{T}[k,t]) + \\ & in^{i}_{P}[k,t]*out^{i}_{0}[k,t]*0.5*in^{i}_{C}[k,t]*0.5 + \\ & in^{i}_{T}[k,t]*(in^{i}_{T}[k,t]*(out^{i}_{0}[k,t]+out^{i}_{P}[k,t])^{2}+ \\ & \qquad in^{i}_{C}[k,t]*out^{i}_{0}[k,t]*(out^{i}_{0}[k,t]+out^{i}_{P}[k,t])) + \\ & in^{i}_{C}[k,t]*in^{i}_{T}[k,t]*out^{i}_{0}[k,t]*(out^{i}_{0}[k,t]+out^{i}_{P}[k,t]) + \\ & in^{i}_{C}[k,t]^{2}*out^{i}_{0}[k,t]^{2}*0.5 \end{aligned} \qquad \text{(EQ 41)}$$

An input buffer is empty after the route phase of the cycle (before the external communication phase) if it was empty before the internal communication phase or if it contained a packet, token, or combined packet-token that was able to be routed during the internal communication phase. A packet is routable if it is chosen over another input and the output where it is going is empty (line 2 of equation 41), or if the packet of a combined packet-token is chosen instead but the packet is bound for the output not taken by the chosen packet (line 3). In such a case, the packet may be routed even if the packet-token is blocked and cannot split. A token is routable if the other input is a token and both outputs are empty or contain packets, or if the other input is a combined packet-token, the destination output of the packet is empty, and the other output is empty or contains a packet (line 4). A combined packet-token is fully routable (splitting the combined packet-

token will not leave the input empty) if the other input is a token, the destination output of the packet is empty, and the other output is empty or contains a packet (line 5), or if the other input is a combined packet-token, both outputs are empty, and each packet is bound for a different output (line 6).

$$
\begin{aligned}
in^{f}_{T}[k,t] = \quad & in^{i}_{T}[k,t]*(in^{i}_{P}[k,t]+in^{i}_{0}[k,t]+ \\
& in^{i}_{T}[k,t]*(1-(out^{i}_{P}[k,t]+out^{i}_{0}[k,t])^{2})) + \\
& in^{i}_{C}[k,t]*(1 - out^{i}_{0}[k,t]) + \\
& in^{i}_{C}[k,t]*out^{i}_{0}[k,t]*(out^{i}_{T}[k,t] + out^{i}_{C}[k,t])) + \\
& in^{i}_{C}[k,t]*out^{i}_{0}[k,t]*(in^{i}_{T}[k,t]*(out^{i}_{T}[k,t]+out^{i}_{C}[k,t]) + \\
& 0.5*in^{i}_{C}[k,t]* 0.5 + \\
& in^{i}_{C}[k,t]*0.5*(1-out^{i}_{0}[k,t]) + \\
& 0.5*(in^{i}_{P}[k,t]+NG[k,t])) \qquad \text{(EQ 42)}
\end{aligned}
$$

An input buffer contains a token after the internal communication phase of the cycle if it contained a token before the internal communication phase that was not able to move forward, or if it contained a combined packet-token before the internal communication phase and only the packet was able to move forward. A token will not move forward unless either the other input holds a token and both outputs can accept tokens (lines 1,2), or the other input holds a combined packet token, the packet can advance, and the other output is able to accept a token (lines 3,4). A combined packet-token will be split if the packet can move forward but the token cannot. This will be the case if: the other input is a token but the other output (the one to which the packet is not directed, since that must be empty for the combined packet-token to split) contains a token or packet-token (line 5); or the other input contains a combined packet-token and either both packets are bound for the same output (line 6), or the output to which the other packet is destined is not empty (line 7); or if the other input contains a packet or a new ghost (the packet from the combined packet-token can only win a comparison with a new ghost, as opposed to any ghost—line 8).

$$
\begin{aligned}
in^{f}_{C}[k,t] = \quad & in^{i}_{C}[k,t]*(0.5*(in^{i}_{P}[k,t]+NG[k,t])+ \\
& (in^{i}_{0}[k,t]-NG[k,t])+ \\
& (0.5*(in^{i}_{P}[k,t])+NG[k,t])*(1 - out^{i}_{0}[k,t]) + \\
& 0.5*in^{i}_{C}[k,t]* 0.5 + \\
& 0.5*in^{i}_{C}[k,t]*0.5*(1-out^{i}_{0}[k,t]) + \\
& 0.5*in^{i}_{C}[k,t]*(1-out^{i}_{0}[k,t]) + \\
& in^{i}_{T}[k,t]*(1-out^{i}_{0}[k,t])) \qquad \text{(EQ 43)}
\end{aligned}
$$

An input buffer contains a combined packet-token after the route phase of the cycle if it contained one before the internal communication phase and no part could be forwarded. The packet-token might lose the comparison to a packet, combined packet-token (both packets are bound for the same output), or ghost message (it automatically loses to an old ghost–line 2) or win the comparison against any type of input, but be blocked by a full output buffer (lines 3,5,6,7).

$$in^{f}_{P}[k,t] = 1 - in^{f}_{C}[k,t] - in^{f}_{T}[k,t] - in^{f}_{0}[k,t] \quad \text{(EQ 44)}$$

An input buffer contains a packet after the internal communication phase of the cycle if it is not in any other possible state.

$$\begin{aligned} out^{f}_{C}[k,t] = \; & out^{i}_{C}[k,t] + \\ & out^{i}_{0}[k,t]*(0.5*2*in^{i}_{C}[k,t]*in^{i}_{T}[k,t]*(out^{i}_{0}[k,t]+out^{i}_{P}[k,t]) + \\ & \qquad 0.5*in^{i}_{C}[k,t]^{2}*out^{i}_{0}[k,t]) + \\ & out^{i}_{P}[k,t]*(in^{i}_{T}[k,t]^{2}*(out^{i}_{P}[k,t]+out^{i}_{0}[k,t])+ \\ & \qquad 0.5*2*in^{i}_{T}[k,t]*in^{i}_{C}[k,t]*out^{i}_{0}[k,t]) \end{aligned} \quad \text{(EQ 45)}$$

An output buffer contains a combined packet-token after the internal communication phase of the cycle if it contained one prior to the internal communication phase, or if it was empty prior to the internal communication phase and a combined-packet-token was sent to it (lines 2,3), or if it contained a packet prior to the internal communication phase and a token was sent to it (Lines 4,5).

$$\begin{aligned} out^{f}_{T}[k,t] = \; & out^{i}_{T}[k,t] + \\ & out^{i}_{0}[k,t]*(out^{i}_{0}[k,t]+out^{i}_{P}[k,t])*in^{i}_{T}[k,t]^{2} + \\ & out^{i}_{0}[k,t]^{2}*2*0.5*in^{i}_{T}[k,t]*in^{i}_{C}[k,t] \end{aligned} \quad \text{(EQ 46)}$$

An output buffer contains a token after the internal communication phase of the cycle if it contained one prior to the internal communication phase, or if it was empty prior to the internal communication phase and a token was sent to it. A token will be sent if both inputs are tokens and the other output is empty or a contains a packet (line 2) or if both outputs are empty and the inputs are a token and a combined packet-token (2 possibilities) and the packet is bound for the other output (0.5 chance–line 3).

$$\begin{aligned} out^{f}_{P}[k,t] = \; & out^{i}_{P}[k,t]*((1-in^{i}_{T}[k,t])^{2}+ \\ & \qquad 2*in^{i}_{T}[k,t]*(in^{i}_{P}[k,t]+in^{i}_{0}[k,t])+ \\ & \qquad 2*in^{i}_{T}[k,t]*in^{i}_{C}[k,t]*0.5+ \end{aligned}$$

$$
\begin{aligned}
& 2*in^i_T[k,t]*in^i_C[k,t]*0.5*(1-out^i_0[k,t])+ \\
& in^i_T[k,t]^2*(out^i_T[k,t]+out^i_C[k,t])) + \\
out^i_0[k,t]*(in^i_P[k,t]^2*0.5+ & \\
& 2*0.5*0.5*in^i_P[k,t]*NG[k,t]+ \\
& 2*0.5*in^i_P[k,t]*in^i_T[k,t]+ \\
& in^i_C[k,t]^2*0.5*0.5*(1-out^i_0[k,t]) + \\
& in^i_C[k,t]^2*0.5*0.5 + \\
& in^i_C[k,t]^2*0.5*(1-out^i_0[k,t])*0.5 + \\
& 2*0.5*0.5*in^i_C[k,t]*NG[k,t] + \\
& 2*0.5*in^i_C[k,t]*in^i_T[k,t]*(out^i_T[k,t]+out^i_C[k,t]) + \\
& 2*0.5*in^i_C[k,t]*in^i_P[k,t] + \\
& 2*0.5*in^i_C[k,t]*in^i_P[k,t]*0.5*0.5) \qquad \text{(EQ 47)}
\end{aligned}
$$

An output buffer contains a packet at the end of the route phase of the cycle if it contained a packet prior to the internal communication phase and no token was sent to it during that phase, or if it was empty prior to the internal communication phase and a packet was sent to it during that phase. The buffer cannot receive a token unless both inputs contain tokens or combined packet-tokens (lines 1,2). Nothing will be sent if the inputs contain a token and a combined packet-token and the packet is bound for this output (line 3), or if the packet is bound for the other output but is blocked there also (line 4). If both inputs contain tokens, a token will be sent unless the other output contains a token or combined packet-token (line 5). The buffer will receive a packet if both inputs contain packets and the chosen packet is bound for this output buffer (line 6), or if a packet bound for this buffer wins the comparison against a ghost (line 7) or a token (line 8). If both inputs contain combined packet-tokens, then a packet will be sent if: the winning packet is bound for this output and the loser is bound for the other output but is blocked (line 9); or if both packets are bound for this output (line 10); or if the losing packet is bound for this output and the winning packet is bound for the other output and is blocked (line 11). A packet component of a combined packet-token bound for this output will be sent if it wins a comparison with a new ghost (line 12), or if the other input is a token but the other output already contains a token or combined packet-token (line 13). If the inputs contain a combined packet-token and a packet, a packet will be sent to this buffer if both packets are bound for this buffer (line 14), or if the packet component wins and is bound for the other output and the packet is bound for this output (line 15).

$$out_0[k,t] = 1 - out^f_P[k,t] - out^f_T[k,t] - out^f_C[k,t] \quad \text{(EQ 48)}$$

An output buffer has only ghost message information at the end of the internal communication phase of a cycle if it is not in any other state.

$$Q_C[k,t] = \begin{cases} Q_{C1} & (k=1) \\ out^f_C[k-1,t] & (k>1) \end{cases} \quad \text{(EQ 49)}$$

$$Q_P[k,t] = \begin{cases} Q_{P1} & (k=1) \\ out^f_P[k-1,t] & (k>1) \end{cases} \quad \text{(EQ 50)}$$

$$Q_T[k,t] = \begin{cases} 1-Q_{C1}-Q_{P1} & (k=1) \\ out^f_T[k-1,t] & (k>1) \end{cases} \quad \text{(EQ 51)}$$

The load of combined packet-tokens, packets, and tokens on an input buffer is generated by the presence of each type of token in the output buffer that feeds the input. The input load distribution describes the load on the first stage of the network.

$$R[k,t] = \begin{cases} in^f_0[k+1,t] & (k<n) \\ 1 & (k=n) \end{cases} \quad \text{(EQ 52)}$$

The probability that a packet or combined packet-token held in an output buffer will be able to advance is based on the ability of the next stage or network output to accept. The network outputs are assumed always to be able to accept.

Upon iteration, the steady-state throughput, s, and delay, d, of the system can be calculated using the following equations. The throughput, s (equation 53), is determined by the concentration of packets and combined packet-tokens in output buffers and whether the outputs will be able to advance. The throughput must be the same for each stage of the network at steady state for the model to be correct.

$$s = (out^f_P[k]+out^f_C[k])*R[k] \quad \text{(EQ 53)}$$

Note that tokens do not contribute to the performance measures of the system. Only packets and combined packet-tokens contribute to throughput. The same is true for the delay measure.

Delay (equation 54) is measured as the average number of cycles a packet or combined packet-token will spend in both input and output buffers at each stage of the network. At steady state, for example, R[k] gives the probability that a packet or packet-token will advance from an

output buffer at stage k in one cycle. The inverse of R[k] is the average time a packet or packet-token will spend in an output buffer at stage k in cycles.

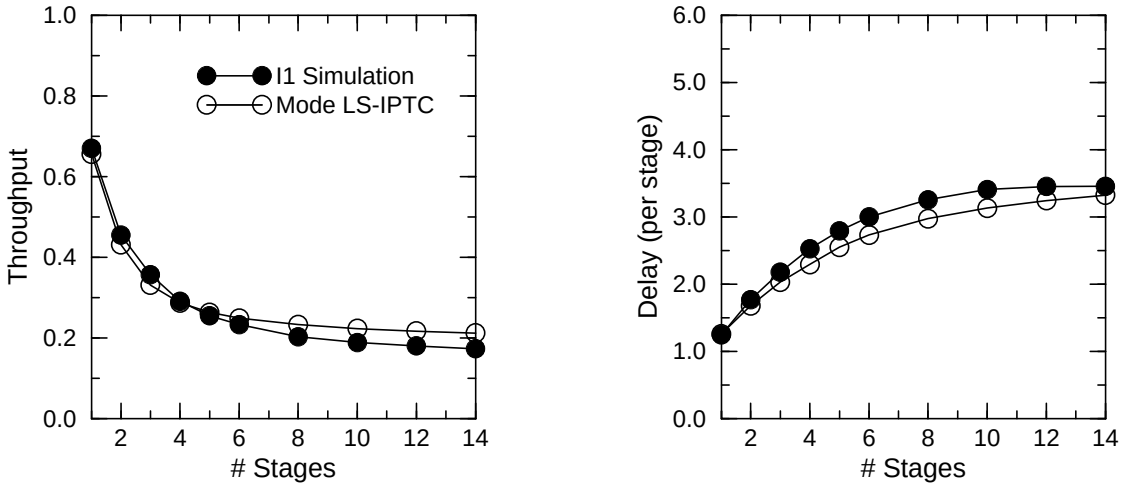
The second term (in the brackets in equation 54) is the time spent in the input buffer of the switch. The denominator is the probability that a packet or a packet component of a combined packet-token will advance from the input buffer to the output in a given cycle, given that the input buffer holds such input. The output buffer to which the packet is bound must be free, and the packet must win a comparison with a ghost, packet, or combined packet-token, lose a comparison with a combined packet-token but be able to be passed anyway (the packet component is bound for the other output), or be compared with a token (a packet always wins such a comparison). We compensate for the fact that the last cycle spent in an input buffer is also the first cycle spent in the output buffer by subtracting one cycle from the delay at each stage.

$$d = \frac{1}{n}\sum_{1}^{n}\left[\frac{1}{R[k]} + \frac{1}{out_0^i[k]\ (0.5\,(in_C^i[k] + in_P^i[k] + NG[k]) + 0.25\,(in_C^i[k]) + in_T^i[k])} - 1\right]$$

(EQ 54)

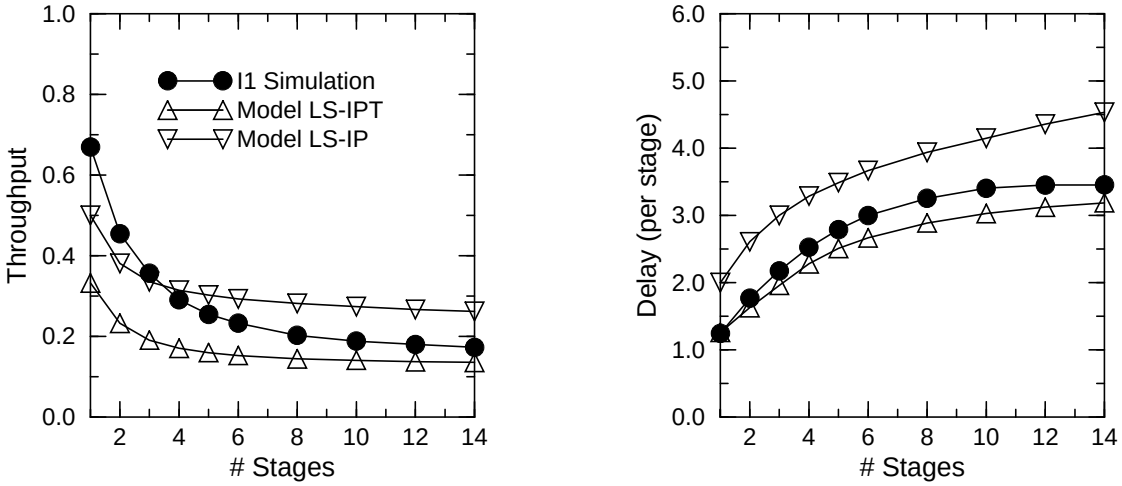
Graphs 6.2.5 plot performance of the LS-IPTC model against the simulation data for locally synchronous operation of switch I1. Model LS-IPTC predicts the performance of the simulation within 23 percent in all cases for throughput and within 9 percent in all cases for delay (within 4 percent at 14 stages).

The differences in throughput and delay for larger networks can be attributed to limitations in the distribution of destinations of input packets in the simulations. Slightly non-uniform distributions will have little effect in smaller networks, because long strings of packets cannot occur (due to the small numbers of processors) and because any given switch at an early stage only has a small number of processors feeding it, limiting the number of packets a switch receives in any given ltu and the number of processors it can affect with a slowdown. In a larger network, non-uniform distributions can cause much longer strings of packets that, at later network stages, are capable of backing up larger numbers of processors.



GRAPHS 6.2.5 - Model LS-IPTC vs. local synchrony simulation for switch I1.

Graphs 6.2.6 again show the predictions of models LS-IP and LS-IPT, which give upper and lower bounds on the throughput and delay of simulation I1. Model LS-IP does not bound the simulation for small networks, because the action of combined packet-tokens allows switches to make more than one decision per cycle. The effect of combined packet-tokens is reduced in larger networks because the number of times combined packet-tokens meet in comparison with the total number of meetings falls significantly at later stages, and the higher concentration of packets in model LS-IP does provide a true upper bound.



GRAPHS 6.2.6 - Local synchrony simulation vs. models LS-IPT and LS-IP to show envelope.

6.2.7 Models for Switch I2

Switch I2 is a high-throughput architecture that utilizes a two-stage design. The multiplexor stage uses two elements to route each input to its correct output (and to send a ghost message to the other output, if necessary). The two elements in the merger stage then merge their two inputs (one from each multiplexor) onto the switch's outputs. The I2 switch design is shown in figure 40.

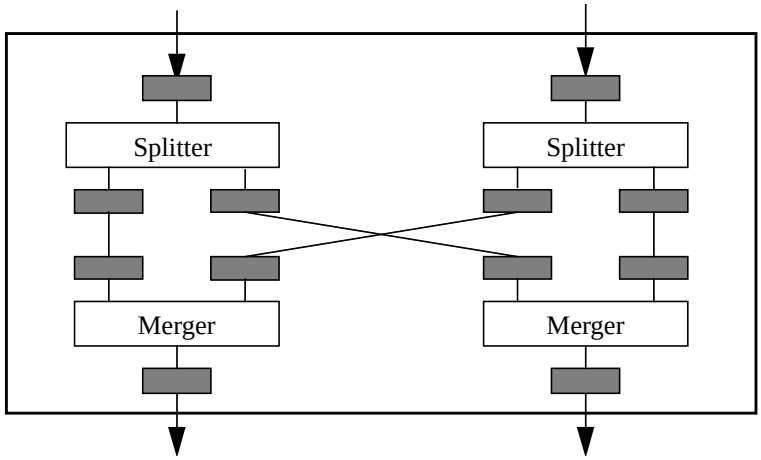


FIGURE 40. Architecture for switch I2.

This design eliminates situations in which a packet that is blocked from moving to its output buffer can hold up several packets (waiting in the other input buffer and at earlier stage switches feeding that buffer) that might otherwise be transferred, and thus considerably boosts throughput. This design, however, necessitates a minimum of two cycles for a packet to be transferred from input to output.

We apply the same approach to modelling the I2 switch networks that we applied to the I1 networks. Because this design uses four levels of buffers (multiplexor inputs and outputs and merger inputs and outputs), we must double the complexity of the model. We expect that the two-stage switch design with four sets of buffers may cause the analytical model's predictions to be less accurate than the I1 models. This is because the probabilistic approach neglects the effects of dependence among buffers. Jenq found that the effects of dependence were small in conventional systems using only input buffers and simple switches. Our contention is that multilevel buffers

increase the effects of dependence, but that our modelling technique still produces reasonable results without the added complexity of considering dependence among buffers.

The I2 switch is a high throughput design. Because information flow is directly related to throughput, the information flow in I2 networks is also high. At full loads, the concentration of empty buffers (holding only ghost message information) in the I2 networks is small, and thus information flow has little or no effect when applied to the I2 analytical models. For this reason, we simplify the I2 analytical models by omitting information flow from them.

We present model LS2-PTC, a representative model for the performance of the I2 switch design networks. A buffer may be in any of four states: in state 0 a buffer is empty (only ghost message information is present); in state P a buffer holds a packet; in state T a buffer holds a token; and finally, in state C a buffer holds a message with a piggyback token.

n - The number of stages in the communication network.

i - Denotes the state at the beginning of the switch cycle; i.e., prior to the internal communication phase.

f - Denotes the state at the end of the switch cycle; i.e., after the internal communication phase, but before the external communication phase.

$S\_in^i_j[k,t]$ - The probability that a given multiplexor input buffer at stage k is in state j at the beginning of cycle t.

$S\_out^i_j[k,t]$ - The probability that a given multiplexor output buffer at stage k is in state j at the beginning of cycle t.

$S\_in^f_j[k,t]$ - The probability that a given multiplexor input buffer at stage k is in state j at the end of cycle t (before the external communication phase).

$S\_out^f_j[k,t]$ - The probability that a given multiplexor output buffer at stage k is in state j at the end of cycle t.

$M\_in^i_j[k,t]$ - The probability that a given merger input buffer at stage k is in state j at the beginning of cycle t.

$M\_out^i_j[k,t]$ - The probability that a given merger output buffer at stage k is in state j at the beginning of cycle t.

$M\_in^f_j[k,t]$ - The probability that a given merger input buffer at stage k is in state j at the end of cycle t.

$M\_out^f_j[k,t]$ - The probability that a given merger output buffer at stage k is in state j at the beginning of cycle t.

$Q_n[k,t]$ - The probability that a packet, token, or combined packet and piggyback token is ready to come to an input buffer at stage k at time t. In other words, the load on an input buffer in stage k at cycle t.

R[k,t] - The probability that a packet or combined packet and piggyback token in a merger output buffer at stage k will advance to the next stage during cycle t, given that the buffer has a packet or combined packet and piggyback token.

Note that the state of the multiplexor inputs at the beginning of a cycle is similar to the state of the inputs in the LS-IPTC model.There is no difference between the operation of the different networks during the external communication phase, other than the internal communication between multiplexor outputs and merger inputs, which takes place during the external communication phase and is unique to the I2 design, but is similar to the communication between output and input buffers in earlier models.

$$S\_in^i_P[k,t] = S\_in^f_P[k,t-1]*(1-Q_T[k,t-1]) + S\_in^f_0[k,t-1]*Q_P[k,t-1] \quad \text{(EQ 55)}$$

A multiplexor input buffer contains a packet at the beginning of a cycle (before the route phase) if it contained a packet prior to the previous external communication phase and did not receive a token (to form a combined packet-token) from the merger output at the previous stage that feeds the input, or if the buffer was empty and a packet was sent.

$$S\_in^i_T[k,t] = S\_in^f_T[k,t-1] + S\_in^f_0[k,t-1]*Q_T[k,t-1] \quad \text{(EQ 56)}$$

A multiplexor input buffer contains a token at the beginning of a cycle if it contained a token prior to the previous external communication phase, or if the buffer was empty and a token was sent by the merger output buffer at the previous stage that feeds the input.

$$S\_in^i_C[k,t] = S\_in^f_C[k,t-1] + S\_in^f_P[k,t-1]*Q_T[k,t-1] + S\_in^f_0[k,t-1]*Q_C[k,t-1] \quad \text{(EQ 57)}$$

A multiplexor input buffer contains a combined packet-token at the beginning of a cycle if it contained a combined packet-token prior to the previous external communication phase, or if it contained a packet and was sent a token, or if it was empty and was sent a combined packet-token.

$$S\_in^i_0[k,t] = 1 - S\_in^i_P[k,t] - S\_in^i_T[k,t] - S\_in^i_C[k,t] \quad \text{(EQ 58)}$$

A multiplexor input buffer is empty at the beginning of a cycle if it is not in another state.

The state of the multiplexor output buffers at the beginning of a cycle is similar to the state of the output buffers in the I1 models or the merger output buffers in this model. While communication between multiplexor outputs and merger inputs, which occurs during the external communication phase, is unique to the I2 models, it is similar to communication between outputs and inputs in the I1 models.

$$\begin{aligned} S\_out^i_0[k,t] = \quad & S\_out^f_0[k,t-1] + \\ & (1-S\_out^f_0[k,t-1])*M\_in^f_0[k,t-1] + \\ & S\_out^f_T[k,t-1]*M\_in^f_P[k,t-1] \end{aligned} \qquad \text{(EQ 59)}$$

A multiplexor output buffer is empty at the beginning of a cycle (before the route phase) if it was empty prior to the previous external communication phase, or if it held a packet, token, or combined packet-token prior to the previous external communication phase and was able to send it to the merger input to which the buffer is connected. Note that a token can be sent to a merger input that holds a packet to form a combined packet-token.

$$S\_out^i_T[k,t] = \quad S\_out^f_T[k,t-1]*(1-M\_in^f_0[k,t-1]-M\_in^f_P[k,t-1]) \qquad \text{(EQ 60)}$$

A multiplexor output buffer contains a token at the beginning of a cycle if it held a token before the previous external communication phase and was unable to forward it during that phase.

$$S\_out^i_C[k,t] = \quad S\_out^f_C[k,t-1]*(1-M\_in^f_0[k,t-1]) \qquad \text{(EQ 61)}$$

A multiplexor output buffer contains a combined packet-token at the beginning of a cycle if it held a combined packet-token prior to the previous external communication phase and was unable to forward it during that phase.

$$S\_out^i_P[k,t] = \quad 1 - S\_out^i_0[k,t] - S\_out^i_T[k,t] - S\_out^i_C[k,t] \qquad \text{(EQ 62)}$$

A multiplexor output buffer contains a packet at the beginning of a cycle if it is not in any other state.

The next sets of equations govern the actions of the multiplexor component of the switch, which occur during the route phase.

$$\begin{aligned} S\_in^f_0[k,t] = \quad & S\_in^i_0[k,t] + \\ & S\_in^i_P[k,t]*S\_out^i_0[k,t] + \\ & S\_in^i_C[k,t]*S\_out^i_0[k,t]*(S\_out^i_0[k,t]+S\_out^i_P[k,t]) + \\ & S\_in^i_T[k,t]*(S\_out^i_0[k,t]+S\_out^i_P[k,t])^2 \end{aligned} \qquad \text{(EQ 63)}$$

A multiplexor input buffer is empty after the internal communication phase (before the external communication phase) of a cycle if it was empty prior to the internal communication phase, if it held a packet that could be forwarded (the output for the packet was empty–line 2), if it held a combined packet-token that could be forwarded (the output for the packet was empty and the other output was empty or held a packet–line 3), or if it held a token that could be forwarded to both outputs (line 4).

$$S\_in^f_P[k,t] = \quad S\_in^i_P[k,t]*(1-S\_out^i_0[k,t]) \qquad \text{(EQ 64)}$$

A multiplexor input buffer contains a packet after the route phase of a cycle if it held a packet prior to the internal communication phase and was unable to forward it (a packet or combined packet-token can be forwarded only to an empty output buffer).

$$S\_in^f_C[k,t] = \quad S\_in^i_C[k,t]*(1-S\_out^i_0[k,t]) \qquad \text{(EQ 65)}$$

A multiplexor input buffer contains a combined packet-token after the internal communication phase of a cycle if it held one prior to the internal communication phase and was unable to forward it.

$$S\_in^f_T[k,t] = \quad 1 - S\_in^f_0[k,t] - S\_in^f_P[k,t] - S\_in^f_C[k,t] \qquad \text{(EQ 66)}$$

A multiplexor input buffer contains a token after the internal communication phase of a cycle if it is not in any other state.

$$\begin{aligned} S\_out^f_P[k,t] = \quad & S\_out^i_P[k,t]*(S\_in^i_P[k,t]+S\_in^i_0[k,t]) + \\ & S\_out^i_P[k,t]*S\_in^i_T[k,t]*(S\_out^i_T[k,t]+S\_out^i_C[k,t]) + \\ & S\_out^i_P[k,t]*0.5*S\_in^i_C[k,t]*(1-S\_out^i_0[k,t]) + \\ & S\_out^i_P[k,t]*0.5*S\_in^i_C[k,t] + \\ & S\_out^i_0[k,t]*0.5*S\_in^i_P[k,t] + \\ & S\_out^i_0[k,t]*0.5*S\_in^i_C[k,t]*(S\_out^i_T[k,t]+S\_out^i_C[k,t]) \end{aligned} \qquad \text{(EQ 67)}$$

A multiplexor output buffer contains a packet at the end of the internal communication phase of a cycle (prior to the external communication phase) if it contained a packet prior to the internal communication phase and it did not receive a token (to create a combined packet-token–lines 1, 2, 3, 4—various possible buffer states which create this situation), or if it was empty prior to the internal communication phase and it received a packet (lines 5, 6). Note that tokens (and the

token component of a combined packet-token) can only be forwarded through the multiplexor if both outputs can accept a token.

$$\begin{aligned} S\_out^f_0[k,t] = \quad & S\_out^i_0[k,t] * (S\_in^i_0[k,t]+ \\ & S\_in^i_P[k,t]*0.5+ \\ & S\_in^i_T[k,t]*(S\_out^i_T[k,t]+S\_out^i_C[k,t])+ \\ & S\_in^i_C[k,t]*0.5*(1-S\_out^i_0[k,t])) \end{aligned} \quad \text{(EQ 68)}$$

A multiplexor output buffer is empty after the internal communication phase of a cycle if it was empty prior to the internal communication phase and no packet, token, or combined packet-token was sent to it. A packet or combined packet-token in the input buffer might be sent to the other multiplexor output, while a token might be blocked from advancing by the existence of a token or packet-token in the other multiplexor output.

$$\begin{aligned} S\_out^f_C[k,t] = \quad & S\_out^i_C[k,t] + \\ & S\_out^i_0[k,t]*0.5*S\_in^i_C[k,t]*(S\_out^i_0[k,t]+S\_out^i_P[k,t]) + \\ & S\_out^i_P[k,t]*S\_in^i_T[k,t]*(S\_out^i_P[k,t]+S\_out^i_0[k,t]) \\ & S\_out^i_P[k,t]*0.5*S\_in^i_C[k,t]*S\_out^i_0[k,t] \end{aligned} \quad \text{(EQ 69)}$$

A multiplexor output buffer contains a combined packet-token after the internal communication phase of a cycle if it contained one prior to that phase, or if it was empty and it received a combined packet token (line 2), or if it contained a packet and it received a token (lines 3, 4).

$$S\_out^f_T[k,t] = \quad 1 - S\_out^f_P[k,t] - S\_out^f_0[k,t] - S\_out^f_C[k,t] \quad \text{(EQ 70)}$$

A multiplexor output buffer contains a token at the end of the route phase of a cycle if it is not in any other state at that time.

The next two sets of equations govern the states of merger input and output buffers at the beginning of the cycle. Their state definitions are similar to those of the multiplexor buffers for the same part of the cycle, because the external communication phase of the cycle has not changed its definition.

$$M\_in^i_0[k,t] = \quad M\_in^f_0[k,t-1]*S\_out^f_0[k,t-1] \quad \text{(EQ 71)}$$

A merger input buffer is empty at the beginning of a cycle (before the internal communication phase) if it was empty prior to the previous external communication phase and nothing was send to it during that phase.

$$M\_in^{i}_{T}[k,t] = M\_in^{f}_{T}[k,t-1] + M\_in^{f}_{0}[k,t-1]*S\_out^{f}_{T}[k,t-1] \quad \text{(EQ 72)}$$

A merger input buffer contains a token at the beginning of a cycle if it contained one prior to the previous external communication phase or if it was empty and a token was sent to it during that phase.

$$M\_in^{i}_{C}[k,t] = M\_in^{f}_{C}[k,t-1] + M\_in^{f}_{P}[k,t-1]*S\_out^{f}_{T}[k,t-1] + M\_in^{f}_{0}[k,t-1]*S\_out^{f}_{C}[k,t-1] \quad \text{(EQ 73)}$$

A merger input buffer contains a combined packet-token at the beginning of a cycle if it contained one prior to the previous external communication phase, or if it contained a packet and was sent a token, or if it was empty and was sent a combined packet-token.

$$M\_in^{i}_{P}[k,t] = 1 - M\_in^{i}_{0}[k,t] - M\_in^{i}_{T}[k,t] - M\_in^{i}_{C}[k,t] \quad \text{(EQ 74)}$$

A merger input buffer contains a packet at the beginning of a cycle if it is not in any other state at that time.

$$M\_out^{i}_{P}[k,t] = M\_out^{f}_{P}[k,t-1]*(1-R[k,t-1]) \quad \text{(EQ 75)}$$

A merger output buffer contains a packet at the beginning of a cycle (before the internal communication phase) if it contained one prior to the previous external communication phase and was unable to send it forward during that phase.

$$M\_out^{i}_{T}[k,t] = \begin{cases} M\_out^{f}_{T}[k,t-1]*(S\_in^{f}_{T}[k+1,t-1]+S\_in^{f}_{C}[k+1,t-1]) & (k<n) \\ 0 & (k=n) \end{cases} \quad \text{(EQ 76)}$$

A merger output buffer contains a token at the beginning of a cycle if it contained one prior to the previous external communication phase and was unable to send it forward during that phase, because the input to which it is connected already contained a token or combined packet-token. Note that a token may be sent forward and combined with a packet to form a combined packet-token.

$$M\_out^{i}_{C}[k,t] = M\_out^{f}_{C}[k,t-1]*(1-R[k,t-1]) \quad \text{(EQ 77)}$$

A merger output buffer contains a combined packet-token at the beginning of a cycle (if it contained one prior to the previous external communication phase and was unable to send it for-

ward during that phase. Although a combined packet-token can be split, the circumstances which make that possible are indistinguishable in this context.

$$M\_out^i_0[k,t] = 1 - M\_out^i_P[k,t] - M\_out^i_T[k,t] - M\_out^i_C[k,t] \quad \text{(EQ 78)}$$

A merger output buffer is empty at the beginning of a cycle if it is not in any other state at that time.

The following two sets of equations govern the operation of the merger sub-components, which perform their actions during the route phase of the cycle. The merger performs the local synchrony timestamp comparison between its two inputs in order to merge a sorted stream onto its single output.

$$\begin{aligned} M\_in^f_0[k,t] = \; & M\_in^i_0[k,t] + \\ & M\_in^i_P[k,t]*(0.5*(1-M\_in^i_T[k,t])+M\_in^i_T[k,t])*M\_out^i_0[k,t] + \\ & M\_in^i_C[k,t]*M\_in^i_T[k,t]*M\_out^i_0[k,t] + \\ & M\_in^i_T[k,t]*M\_in^i_T[k,t]*(M\_out^i_0[k,t]+M\_out^i_P[k,t]) + \\ & M\_in^i_T[k,t]*M\_in^i_C[k,t]*M\_out^i_0[k,t] \end{aligned} \quad \text{(EQ 79)}$$

A merger input buffer is empty after the internal communication phase (before the external communication phase) of the cycle if it was empty prior to the internal communication phase, or if it contained a packet, token, or combined packet-token and was able to forward it during the internal communication phase. Note that tokens must appear on both inputs before a token can be transferred to the output (lines 3, 4, 5).

$$\begin{aligned} M\_in^f_P[k,t] = \; & M\_in^i_P[k,t]*0.5*(1-M\_in^i_T[k,t]) + \\ & M\_in^i_P[k,t]*(0.5*(1-M\_in^i_T[k,t])+M\_in^i_T[k,t])*(1- \\ & M\_out^i_0[k,t]) \end{aligned} \quad \text{(EQ 80)}$$

A merger input buffer contains a packet after the internal communication phase of a cycle if it contained one prior to the internal communication phase and the packet lost the timestamp comparison and was thus unable to move forward (line 1), or if the packet won the timestamp comparison, but was blocked by traffic already in the output buffer (line 2).

$$\begin{aligned} M\_in^f_C[k,t] = \; & M\_in^i_C[k,t]*0.5*(1-M\_in^i_T[k,t]) + \\ & M\_in^i_C[k,t]*(0.5*(1-M\_in^i_T[k,t])+ \\ & \qquad M\_in^i_T[k,t])*(1-M\_out^i_0[k,t]) \end{aligned} \quad \text{(EQ 81)}$$

A merger input buffer contains a combined packet-token after the internal communication phase of a cycle if it contained one prior to the internal communication phase and the packet component lost the timestamp comparison and was thus unable to move forward (line 1), or if the packet component won the timestamp comparison, but was blocked by traffic already in the output buffer (line 2).

$$M\_in^f_T[k,t] = 1 - M\_in^f_0[k,t] - M\_in^f_P[k,t] - M\_in^f_C[k,t] \quad \text{(EQ 82)}$$

A merger input buffer contains a token after the route phase of a cycle if it is not in any other state at that time.

$$\begin{aligned} M\_out^f_P[k,t] = {} & M\_out^i_P[k,t]*(1-M\_in^i_T[k,t]^2) + \\ & M\_out^i_0[k,t]*(M\_in^i_P[k,t]^2 + \\ & \quad 2*0.5*M\_in^i_0[k,t]*M\_in^i_P[k,t] + \\ & \quad 2*M\_in^i_P[k,t]*M\_in^i_T[k,t] + \\ & \quad M\_in^i_C[k,t]^2 + \\ & \quad 2*0.5*M\_in^i_0[k,t]*M\_in^i_C[k,t] + \\ & \quad 2*M\_in^i_C[k,t]*M\_in^i_P[k,t]) \end{aligned} \quad \text{(EQ 83)}$$

A merger output buffer contains a packet after the internal communication phase (before the external communication phase) of a cycle if it contained one prior to the internal communication phase and the merger inputs did not both contain tokens (a token could then be sent to form a combined packet-token–line 1), or if the output was empty before the internal communication phase and a packet could be transferred to it. A packet will be transferred if both inputs hold packets (line 2, or if the inputs are a ghost and a packet and the packet wins the comparison (line 3), or if the inputs are a packet and a token (line 4), or if both inputs are combined packet-tokens (line 5), or if a combined packet token wins the comparison with a ghost (line 6), or if the inputs contain a combined packet-token and a packet (line 7). Note that the merger is able to either split up a combined packet-token to transfer only the packet, or transfer a token to an output buffer containing a packet, in order to form a combined packet-token.

$$\begin{aligned} M\_out^f_T[k,t] = {} & M\_out^i_T[k,t] + \\ & M\_out^i_0[k,t]*M\_in^i_T[k,t]^2 \end{aligned} \quad \text{(EQ 84)}$$

A merger output buffer contains a token after the internal communication phase of a cycle if it contained one prior to the internal communication phase, or if the buffer was empty and a

token was sent to it during that phase. The merger sends a token only when both inputs hold a token.

$$M\_out^f_C[k,t] = M\_out^i_C[k,t] + M\_out^i_P[k,t]*M\_in^i_T[k,t]^2 + M\_out^i_0[k,t]*2*M\_in^i_C[k,t]*M\_in^i_T[k,t] \quad \text{(EQ 85)}$$

A merger output buffer contains a combined packet-token after the internal communication phase of a cycle if it contained one prior to the internal communication phase (line 1), or if it contained a packet and a token was sent to it during that phase (line 2), or if it was empty and a combined packet-token was sent to it (line 3).

$$M\_out^f_0[k,t] = 1 - M\_out^f_P[k,t] - M\_out^f_T[k,t] - M\_out^f_C[k,t] \quad \text{(EQ 86)}$$

A merger output buffer is empty after the internal communication phase of a cycle if it is not in any other state at that time.

As with previous models, the loads on multiplexor input buffers are defined by the input loads (for stage 1) or the probability that the merger output buffer connected to the input contained a given type of message prior to the previous external communication phase.

$$Q_C[k,t] = \begin{cases} Q_{C1} & (k=1) \\ M\_out^f_C[k-1,t] & (k>1) \end{cases} \quad \text{(EQ 87)}$$

$$Q_P[k,t] = \begin{cases} Q_{P1} & (k=1) \\ M\_out^f_P[k-1,t] & (k>1) \end{cases} \quad \text{(EQ 88)}$$

$$Q_T[k,t] = \begin{cases} 1-Q_{C1}-Q_{P1} & (k=1) \\ M\_out^f_T[k-1,t] & (k>1) \end{cases} \quad \text{(EQ 89)}$$

The ability of a packet or combined packet-token to advance to the next stage from a merger output buffer depends on the state of the multiplexor input buffer to which output is connected (that buffer must be empty). The network outputs are assumed to always accept outputs from the n^{th} stage.

$$R[k,t] = \begin{cases} S\_in^f_0[k+1,t] & (k<n) \\ 1 & (k=n) \end{cases} \quad \text{(EQ 90)}$$

After iteration, these quantities reduce to time independent quantities. We predict the steady state throughput, s (equation 91), and delay, d (equation 92), using the following equations.

$$s = (M\_out^f_C[k] + M\_out^f_P[k])*R[k] \quad \text{(EQ 91)}$$

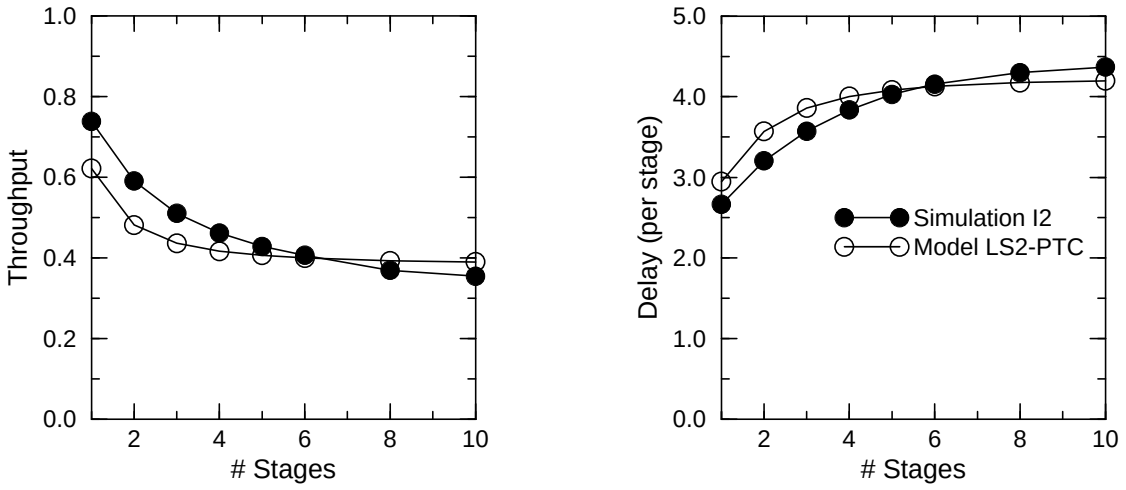
The delay (equation 92) must be calculated using the probable delay for a packet or combined packet-token in each of the four buffer stages in a switch: the first component (within the brackets) is the delay for the merger output buffer, the second is the delay for the multiplexor input buffer, the third is the delay for the multiplexor output buffer, and the fourth is the delay for the merger input buffer. Note that a comparison is made only between inputs in the merger, so the delay is complex only in the fourth component. We compensate for the fact that the last cycles spent in the multiplexor and merger input buffers are also the first cycles in the multiplexor and merger output buffers, respectively, by subtracting two cycles from the delay encountered at each stage.

$$d = \frac{1}{n}\sum_{1}^{n}\left[\frac{1}{R[k]} + \frac{1}{Sout_0^i[k]} + \frac{1}{Min_0^f[k]} + \frac{1}{(0.5\,(1 - Min_T^i[k]) + Min_T^i[k])\,Mout_0^i[k]} - 2\right]$$

(EQ 92)

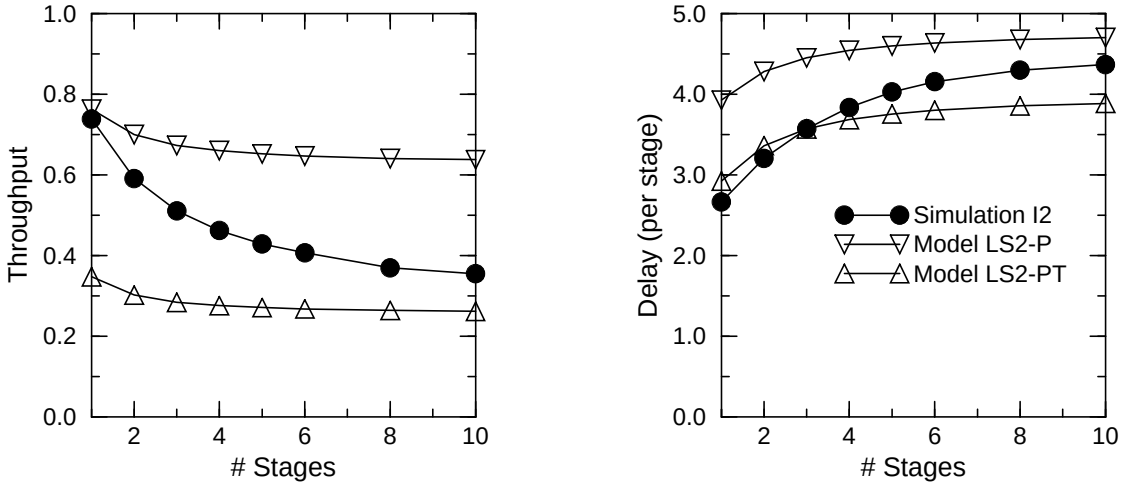
Graphs 6.2.7 plot the performance of simulation I2 against the predictions of model LS-IPT2. The model predicts reasonably well for larger networks, as it is more optimistic about the performance properties of the design. Like the I1 models, this is to be expected: the simulations are prone to slight variations in the distribution of packet addresses, which can cause increasingly large backups in later stages in the network as network size grows. The model predicts throughput to within 12 percent for a network of ten stages, and within 5 percent for delay. Both throughput and delay predictions would be expected to worsen slightly for larger networks, as the simulation data becomes slightly worse (lower throughput and higher delay) for numbers of stages greater than 10.

Graphs 6.2.8 chart the predictions of the simpler LS-P and LS-PT models, which again give upper and lower bounds for the performance of the I2 design for larger numbers (>4) of stages.



GRAPHS 6.2.7 - I2 network simulation vs. Model LS2-PTC.

Graphs 6.2.8 chart the predictions of the simpler LS-P and LS-PT models, which again give upper and lower bounds for the performance of the I2 design for larger numbers (>4) of stages.



GRAPHS 6.2.8 - Network I2 vs. LS-P and LS-PT models for switch I2 to show envelope.

6.3 Simulation Data

This section presents results from simulation studies performed jointly with Craig Williams [RWW92].We present simulation data for each of the four networks described in section 6.1.

Our assumption is that networks C1 and I1 have the same switch cycle time, and that C2 and I2 have latency double that of C1 and I1.

Our simulation environment allows us to present data for each network and to vary a number of operational parameters and workload models. The simulation parameters include the following:

Network Size - The number of PE/MMs.

Variables/MM - The number of shared variables per MM.

Network Switch Queue Size - The maximum number of messages that a switch input queue can hold. In C2 and I2, the queue size applies to the input queues of both the splitter/multiplexer and the mergers.

READ probability - The probability that any given access is a READ, as opposed to a WRITE.

Atomic Action Mean - In workload models that include atomic actions, the average number of operations in an atomic action. The sizes of atomic actions are from an exponential distribution with a mean equal to the mean supplied as a parameter and a range from 1 to (10 * (mean-1)) + 1.

Forced Singleton Probability - A user defined probability *p*. The probability that an atomic action is a singleton, i.e., that it is of size 1, is $p + q(1-p)$, where q is the probability of choosing 1 from the distribution of atomic action sizes described above.

Atomic Action Cap - In workload models that include atomic actions, the cap on the number of atomic actions that any PE may have started and not completed at any given time.

Traffic Model - A set of parameters that describe the distribution of operations on variables. The simulation uses three different types of traffic models: a uniform model, in which all variables are accessed with equal probability; a hot-spot model, in which one variable is accessed with greater probability; and a warm-spot model, in which several variables are accessed with greater probability.

Workload Model - A set of rules governing the production and execution of operations. The simulation uses many different workload models, ranging from a “raw power” workload, in which PEs produce a saturation or probabilistic load of generic operations with no constraints on the order in which operations are executed, to a workload with atomicity, sequencing, and data-dependency constraints among operations.

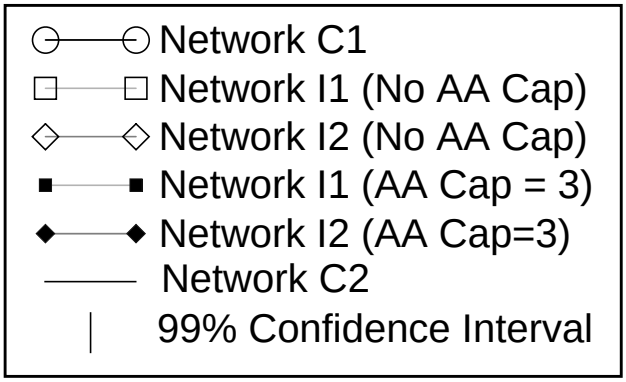
The study is divided into six series of simulations, each focusing on a different workload model, or, in one case, traffic model. For each series, a ‘base case’ is defined giving the default parameter settings. The base case for each parameter and the range of values tested is presented in figure 41. In the remainder of this section, we describe each series and its results.

Parameter	Base Case Value	Other Values Tested
Network Size	5 stages (32 PE/MMs)	4, 6, 8, 10
Variables/MM	32	-
Network Switch Queue Size	1	2, 3, 4, 5, 6
READ Probability	.75	0, .25, .50, .90, 1.0
Atomic Action Mean	3	1-10, 16
Atomic Action Cap	-	1, 3, 6, 12, 16, 20, 24, unlimited
Forced Singleton Probability	0	0, .1, .25, .5, .75
Traffic Model	Uniform	Hotspot, Warmspot

FIGURE 41. Value Ranges for Simulation Parameters

All data points represent results from at least 10 independent runs. For data points at which runs give results with a high variance, we report 99% confidence intervals based on 16 runs.

The following legend applies to all graphs presented. The parenthetical descriptions of the differences between data lines apply only to series 3 and 6.1. In other series, only the first legend for each network will appear.



6.3.1 Raw Power Data

The first series measures the “raw power” of each of the networks. We use the term “raw power” to mean network throughput and delay under a workload of operations with no atomicity or sequencing constraints. The workload model consists of generic, independent operations -- reads and writes are not distinguished and operations may arrive at memory in any order. In the base case the load on the network is a saturation load. A PE generates a new operation whenever its output buffer is empty. Only the forward, i.e., PE->MM, network is simulated in this series. Each MM is a sink for operations. We assume the memory cycle time is the same as the switch cycle time -- each MM can consume one operation per cycle.

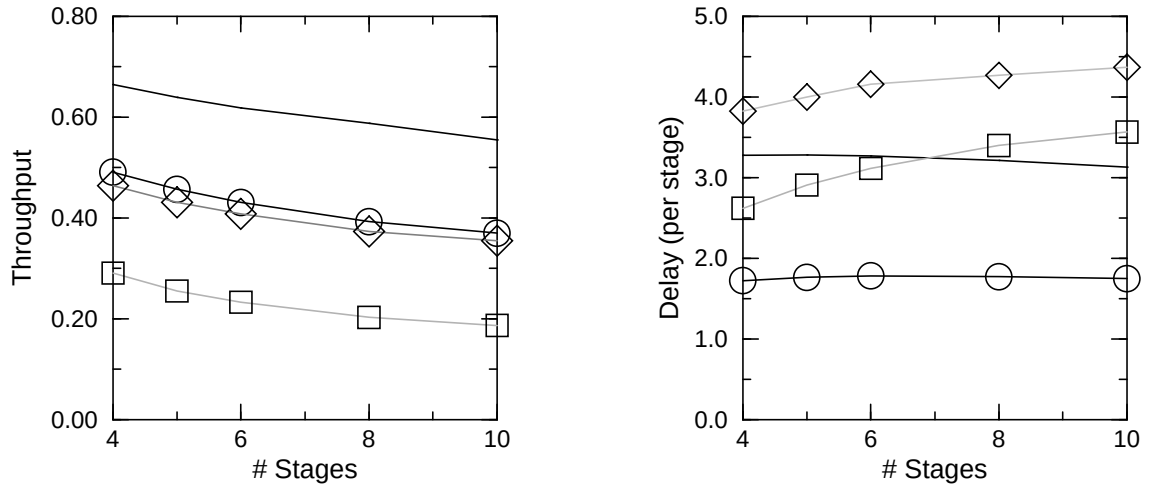
In I1 and I2 enforcement of atomicity is relaxed by specifying an isochron size of one. No comparable way exists in I1 or I2 to relax enforcement of sequential consistency. Thus series 1 actually compares isotach networks that enforce sequential consistency with conventional networks that do not.

Unless otherwise stated, the delay reported in series 1 is the network delay per stage. The delay does not include source queueing or other non-network delay such as delay at the network interface. More specifically, the delay is the average number of cycles between the time an operation is sent from the PE-SIU to the first-stage switch until the time it is sent from the last stage switch to the MM-SIU for the destination MM, divided by the number of network stages. In this series the throughput reported, unless otherwise stated, is the average number of operations arriving at memory per cycle, divided by the number of MMs.

Base Case - Base case performance of the networks is shown in several of the series 1 graphs, e.g. in Graphs 6.3.1-scalability at stages = 5. Each isotach network has lower throughput and higher delay in the base case than the corresponding conventional network. I2’s throughput is two-thirds that of C2, and its delay is one-fourth longer than C2’s. I1’s throughput is slightly over half C1’s and its delay is two-thirds longer than C1’s. Although the difference in the raw power of each pair of corresponding networks is significant, the faster of the isotach networks has delay in

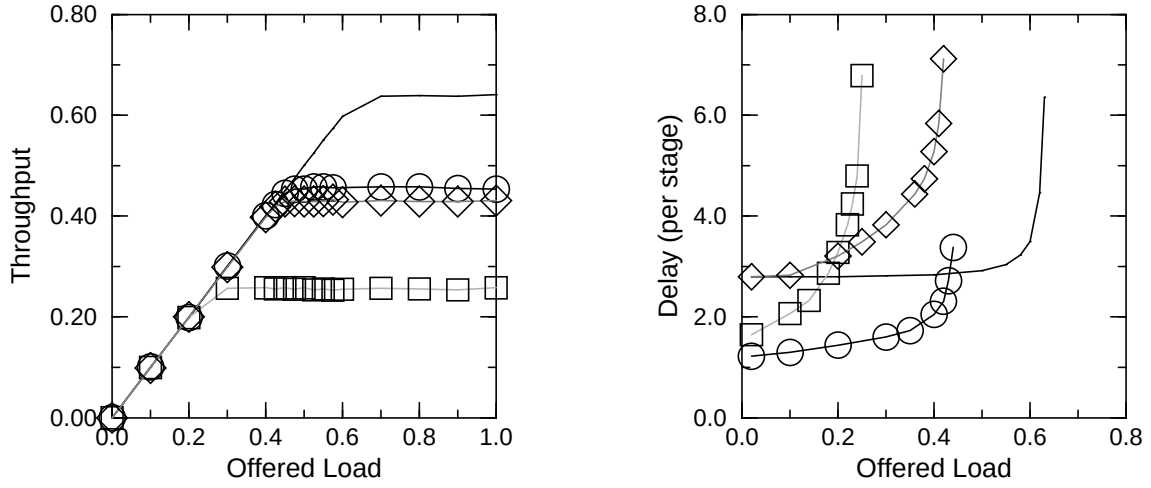
the base case comparable to that of the slower of the conventional networks and the high throughput isotach network has throughput comparable to that of the low throughput conventional network.

In each part of the series, one parameter is varied while each of the other parameters remains at its base case setting.



GRAPHS 6.3.1-Scalability - Data for series 1, variation of network size.

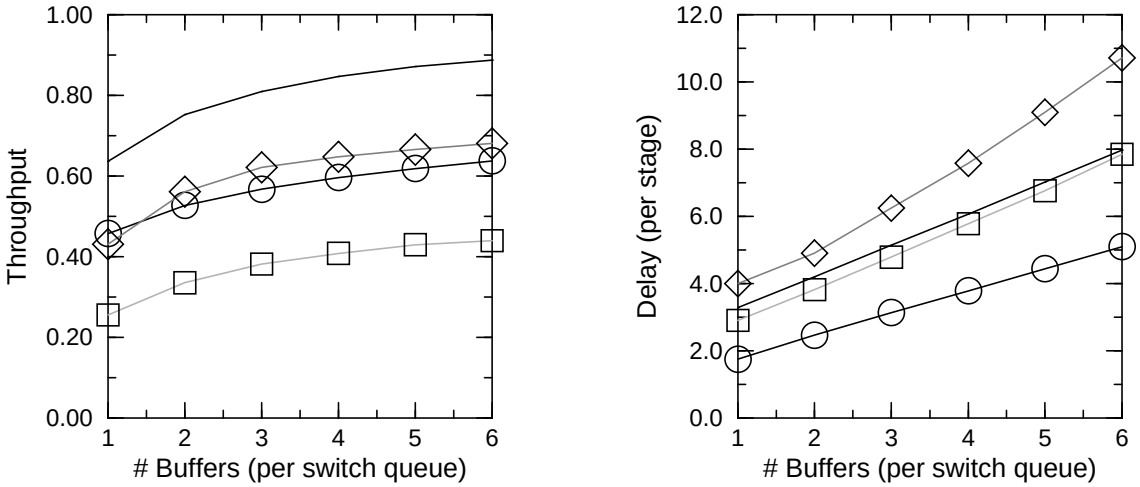
Scalability - Graphs 6.3.1-scalability show the effect of increases in the number of network stages on network throughput and delay. The throughput of each network decreases slowly at about the same rate for each network. The delay per stage increases slowly for I1 and I2 and stays roughly constant for C1 and C2.



GRAPHS 6.3.1-prob_load - Series 1 data for variation in offered load.

Probabilistic Load - Graphs 6.3.1-prob_load show throughput and delay data for specified offered loads. Instead of generating a new operation whenever its output buffer is empty, a PE generates a new operation each cycle with probability *r.* As *r* increases, throughput for each network increases up to the network's saturation point. Delay for each network increases sharply near the saturation point.When load is generated independently of the state of a PE's output buffer, source queueing can account for a significant portion of total delay. Therefore, in this part of series 1 source queueing is included in delay. Delay is measured from the time the operation is generated until it leaves the MM-SIU. Each of the conventional networks can handle a higher load with less delay than the corresponding isotach network. The request rate at which C2 saturates is the highest, 0.65; and I1 the lowest, 0.25. C1 and I2 saturate at about the same request rate, 0.45 and 0.43, respectively.

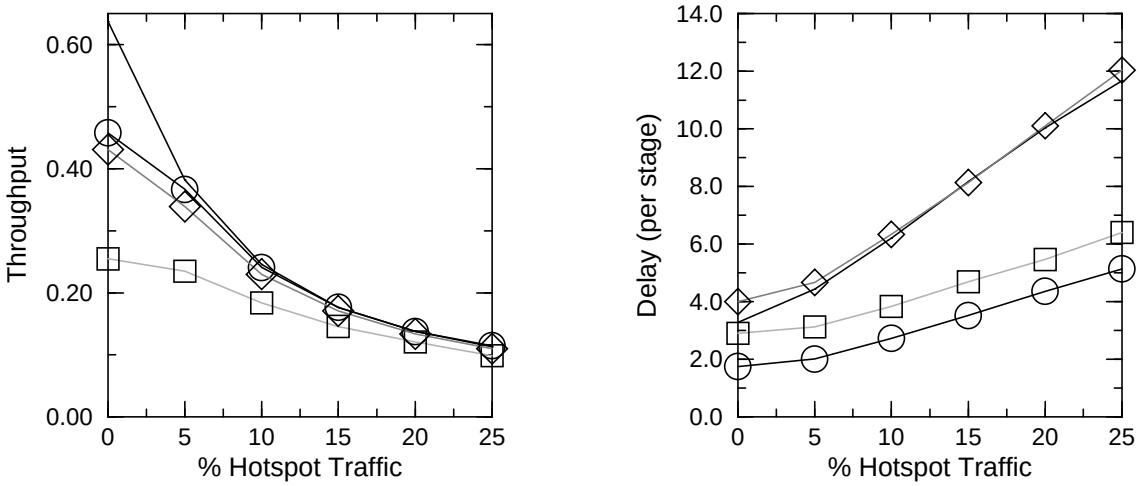
Switch Input Queue Size - Graphs 6.3.1-switch_queues show the effect of increases in the size of the switch input queues. As the number of buffers in each switch input queue increases, throughput improves for all networks, but delay worsens. The graphs indicate that performance is best when the switch input queues are small, able to queue only 1 or 2 operations at most. The result applies only to saturation loads in which source queueing is not a factor.



GRAPHS 6.3.1-switch_queues - Series 1 data for variation of the number of buffers per switch queue.

Hot-Spot Traffic - Graphs 6.3.1-hot_spot give performance results for the hot-spot traffic model. The hot-spot traffic model [PhN85] is a simple variation on the uniform traffic model in

which one variable, the hot-spot, is accessed more often than the other variables. The hot-spot is chosen randomly such that for each trial, each global variable may be the hot-spot with equal probability. The probability that a given operation accesses the hot-spot is the hotspot probability *r* plus *(1-r)* divided by the total number of variables. As the probability of generating operations accessing the hot-spot increases, throughput decreases and the differences in throughput among the networks narrows. The delay of the z-switch networks, C2 and I2, increases at a significantly faster rate than the low-throughput networks. Again, the result applies only to saturation loads in which source queueing is not a factor.



GRAPHS 6.3.1-hot_spot - Series 1 data for various hotspot loads.

Warm-spot Traffic - Table 5 gives results from a warm-spot traffic model. A warm-spot traffic model has several “warm” variables instead of a single hot variable. We defined the warm-spot traffic model in an attempt to identify a model that captures contention for shared variables more realistically than either the uniform or the hot-spot traffic models. The warm-spot traffic model is based on the 80/20 rule (see e.g. [Knu73]), a rule of thumb widely used in describing the pattern of accesses to a pool of shared objects. The 80/20 rule says that 80% of the accesses are to 20% of the variables. The rule is applied recursively, i.e., 80% of the initial 80% of accesses are to 20% of the initial 20% of the accesses, and so on until the number of objects in the set of most accessed objects reaches one. The warm-traffic model we use is a modified version of the standard 80/20 rule that supports ratios other than 80/20. For simplicity, it truncates the recursion at 4 levels

and it assumes that the probability of assessing the most frequently accessed variables at level *i* is the probability of accessing the most frequently accessed variables at level *i-1* cubed. The effect of this modification is to lessen the contention for the most frequently accessed variables. Table 2 shows performance under three different warm-spot traffic models: light contention (60/30 rule); medium contention (70/20); and heavy contention (80/10). The table indicates that increased contention affects the relative performance of the networks in the warm-traffic model in the same way as in the hot-spot model: the differences in throughput narrow and delay in the z-switch networks C2 and I2 grows faster than delay in C1 and I1.

Access Throughput	Low Sharing	Medium Sharing	Heavy Sharing
C1	0.442	0.396	0.156
C2	0.585	0.487	0.163
I1	0.247	0.233	0.124
I2	0.413	0.362	0.147
Normalized Access Delay	Low Sharing	Medium Sharing	Heavy Sharing
C1	1.805	1.964	3.952
C2	3.305	3.970	8.957
I1	3.001	3.164	5.396
I2	4.135	4.546	9.529

TABLE 5. Series 1 Warm-spot data.

The results from Series 1 show that each conventional network outperforms the corresponding isotach networks for workloads with no atomicity, data dependence, or other sequencing constraints. Each of the remaining series compares the performance of the networks under a workload that includes some combination of atomicity and sequencing constraints. In addition to including constraints on execution order, series 2-6 differ from series 1 in other ways:

1. Processes - Each process is independent, executing its own program on its own PE, and each process's program consists of a sequence of atomic actions, each containing one or more operations on shared variables.

2. Reverse network - Each network simulation includes traffic in both the forward (PE->MM) and reverse (MM->PE) directions. All of the reverse networks are conventional networks. I2 uses C2 as the reverse net and the other networks use C1. In each cycle each MM (PE) can both receive one operation (response) from the network and issue one response (operation).
3. Throughput - The throughput reported is the atomic action throughput instead of the access throughput. Unless otherwise stated, the throughput reported is the average number of atomic actions completed per cycle divided by the number of MMs.
4. Delay - The delay reported is the atomic action delay. Unless otherwise stated, delay is the average number of cycles from the time an atomic action is generated until it is completed. Note that delay is not normalized for the number of network stages as it is in series 1.
5. Networks - Results are shown only for C1, I1, and I2. We simulated all four networks, including C2. Although C2 outperforms all the other networks in series 1, it performs more poorly than C1 in the remaining series. The results show that at every data point, C1 provides more throughput with less delay than C2. The reason for C2's poor performance relative to C1 is that in the later series C2 retains its high latency but can't take advantage of its high throughput. Both conventional networks are under-utilized in series 2-6 because higher level synchronization constraints prevent processes from issuing operations fast enough to saturate the network. For simplicity, we omit further discussion of C2.

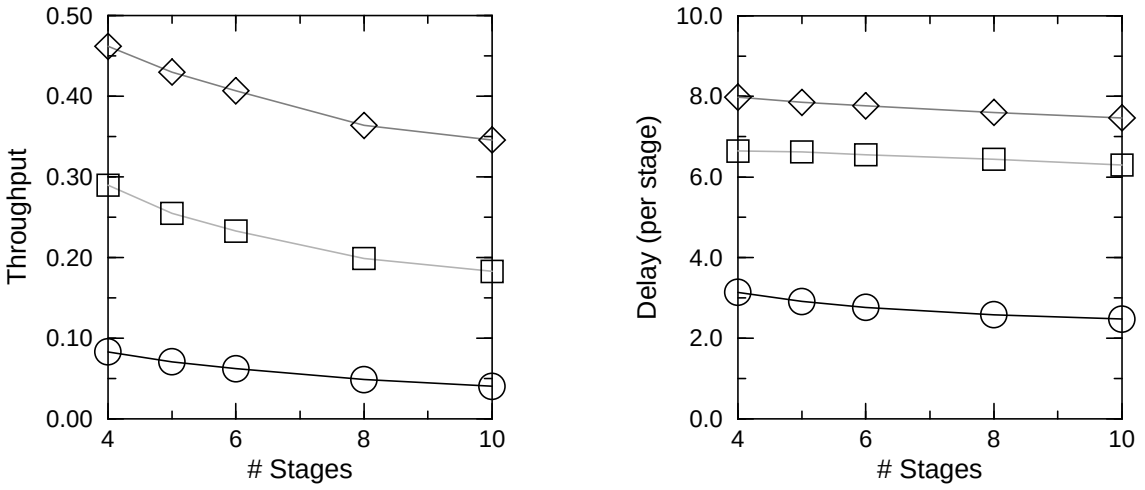
6.3.2 Sequential Consistency Only Data

Series 2 compares the networks under a workload model requiring sequential consistency. In this series, all atomic actions are singletons (size 1) and no data dependencies constrain the issuing of operations, but each process's operations must appear to be executed in order. In I1 and I2, operations can be pipelined without risk of violating sequential consistency, whereas C1 must enforce sequential consistency by limiting each PE to one outstanding operation at a time. In C1, each PE repeatedly issues an operation, waits for a response from memory, and then issues its next operation (i. e., the parameter "aa_cap" = 1). In I1 and I2, each PE issues a new operation whenever its output buffer is empty (aa_cap = unlimited). Note that the isotach network simulations in series 1 and 2 are the same except for the inclusion of the reverse networks and the difference in the way results are reported. Delay in series 2 is the number of cycles between the time an operation is placed in the PE's output buffer until the time the PE receives the response to the operation.

Base Case - Throughput for C1 in the base case is only about 15% of its series 1 throughput, reflecting the high cost of enforcing sequential consistency in a conventional network. For the

isotach networks the throughput in series 1 and 2 is the same. Because they can pipeline operations, the isotach networks outperform C1 in terms of throughput but the delay per operation is longer than C1's delay. In the base case I1 can execute 3.6 times as many operations as C1, but each operation takes 2.3 times as long. I2 can execute 6 times as many operations, but each takes 2.7 times as long. Delay for all networks is higher than in series 1 because delay in series 2 includes delay due to the reverse network and non-network delay.

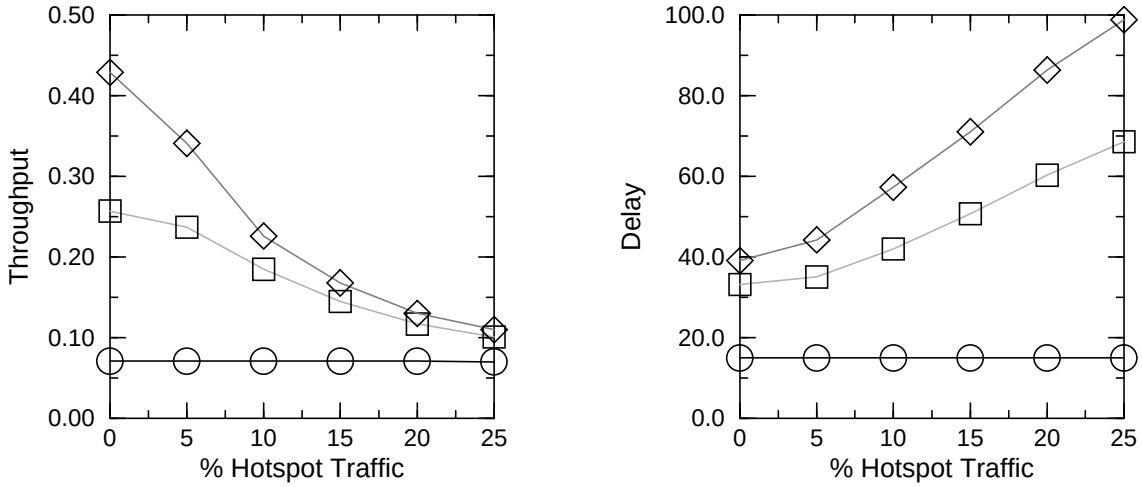
In each part of the series, one parameter is varied while each of the remaining parameters remains at its base case setting.



GRAPHS 6.3.2-scalability - Series 2 data for variation of network size. O - C1; ❑ - I1; ◊ - I2;

Scalability - Graphs 6.3.2-scalability show the effect of varying the network size from its base case setting of 5. As network size increases from 4 to 10 stages, throughput slowly decreases and delay per stage remains roughly constant. Delay per stage in I1 and I2 is relatively constant as the number of network stages increases because the delay due to the forward network is only one component of the delay reported in series 2. The other components, the reverse network and non-network delay either remain constant on a per stage basis or decrease. Delay per stage due to the reverse network is roughly constant since the reverse networks are conventional networks. Non-network delay declines on a per stage basis as the delay is amortized over more stages. Throughput decreases for each network at about the same rate but for different reasons. Throughput per MM decreases in the isotach networks because in isotach networks throughput per MM decreases as the

number of stages increases. Throughput per MM decreases in the conventional networks because a PE can have only one outstanding operation at a time and delay for each operation increases as the number of stages increases.



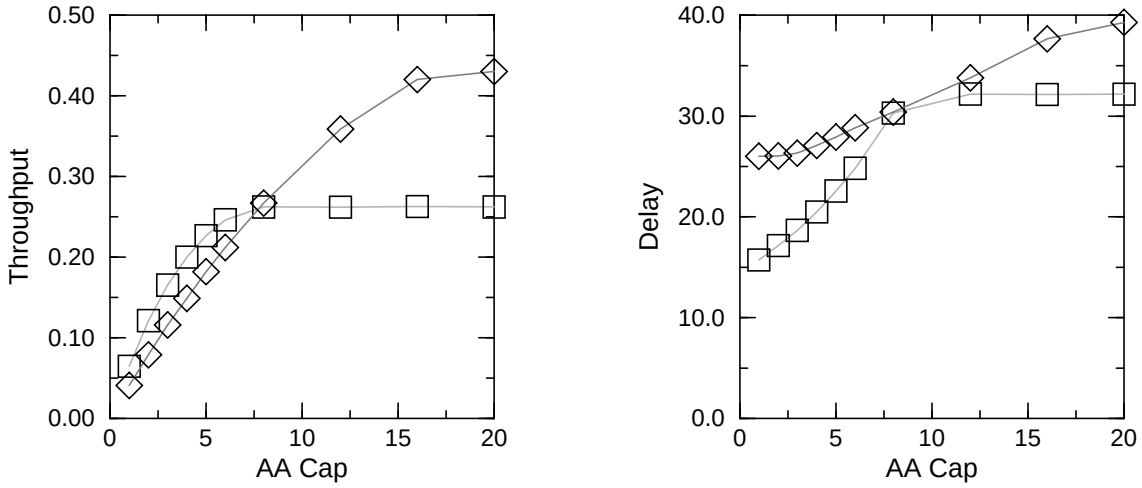
GRAPHS 6.3.2-hot_spot - Series 2 data for various hotspot loads. O - C1; ❑ - I1; ◊ - I2;

Hot-spot and Warm-spot Traffic - Graphs 6.3.2-hot_spot and Table 6 show network performance for the hot-spot and warm-spot traffic models, respectively. As in series 1, networks I1 and I2 show significant decreases in throughput and increases in delay as contention increases. Network C1 has low throughput and low delay. Over the range of traffic model parameters tested, C1 is unaffected by increases in the probability of hot-spot and warm-spot traffic. Throughput and delay are unaffected in C1 because the network is operating at only about 28% capacity due to the restriction on pipelining.

Access Throughput	Low Sharing	Medium Sharing	Heavy Sharing
C1	0.071	0.071	0.071
I1	0.249	0.229	0.128
I2	0.408	0.367	0.254
Access Delay	Low Sharing	Medium Sharing	Heavy Sharing
C1	15.000	15.000	15.002
I1	34.032	36.038	56.861
I2	40.344	42.887	76.587

TABLE 6. Series 2 warm-spot data.

Data Dependencies - Graphs 6.3.2-aa_cap show the effect of changes in the data dependence distance on system performance. Data dependencies diminish throughput of isotach networks by an amount that depends on the distance between data dependent operations. Until now, we have assumed data dependencies either do not exist or are between operations that are so far apart that the data dependencies do not effect pipelining. In this part of series 2 we relax this assumption. With a data dependence distance of 1 (modeled by imposing an aa_cap of 1), the throughput of the isotach networks is lower and the delay higher than that of C1, as can be predicted from the results of series 1. Beginning at a data dependence distance of 2 (aa_cap of 2), the throughput of the isotach systems is higher than C1's although the delay also continues to be higher. C1 requires an aa_cap of 1 in order to maintain sequential consistency and so is unaffected by the data dependence distance. I2 requires a dependence distance of about 16 to take full advantage of its throughput. For I1, a dependence distance of about 5 is sufficient. I2 performs less well than I1 at small dependence distances because I2 cannot take advantage of its higher throughput but is hurt by the higher response turnaround time due to its higher latency.



GRAPHS 6.3.2-aa_cap - Series 2 data for variation of data-dependence distance. ❑ - I1; ◊ - I2;

6.3.3 Atomicity Only Data

Series 3 compares the performance of isotach and conventional systems under a workload requiring atomic access to multiple shared variables. The atomic actions are assumed to be *flat*,

i.e., no data dependencies exist among operations from the same atomic action, and independent, i.e., no data dependencies among different atomic actions inhibit pipelining. The atomic actions must appear to be executed as an indivisible step, but may appear to be executed in any order, i.e., the workload does not require enforcement of sequential consistency. Although sequential consistency is not required, the isotach networks guarantee sequential consistency by their design.

Techniques for enforcing atomicity in isotach systems were described in Chapter 2. Isotach networks enforce atomicity by issuing all the operations in an atomic action in the same isochron. The conventional systems we simulate enforce atomicity using 2PL. A process does not release any lock acquired in an atomic action until it acquires all locks for the atomic action. To avoid deadlock, each process acquires the locks it needs for each atomic action in a predetermined linear order. Since execution need not be sequentially consistent, a process may execute more than one atomic action concurrently.

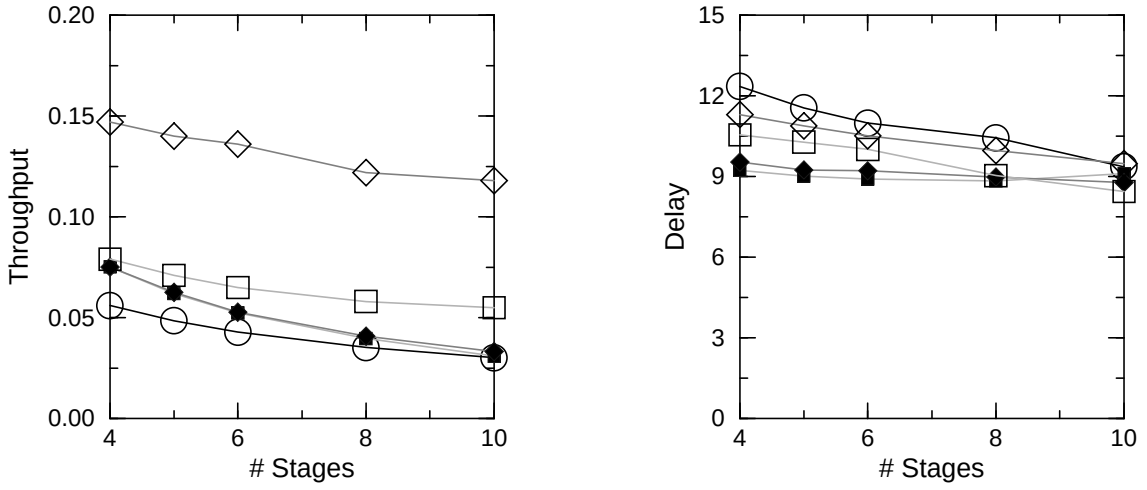
The algorithm for lock acquisition and release simulated is more efficient than that used in most systems. Instead of spinning on a lock, processes queue lock requests at memory. The algorithm distinguishes read locks from write locks and only the latter are exclusive. Several processes can concurrently hold a read lock on the same variable. 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 round-trip 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.

Since execution need not be sequentially consistent, locks are not obtained for operations from singleton (size 1) atomic actions. The MMs execute operations from singleton atomic actions as soon as they are received even if another process holds an exclusive lock on the variable

accessed. In the base case, with an atomic action mean size of 3, about 22% of atomic actions are of size 1.

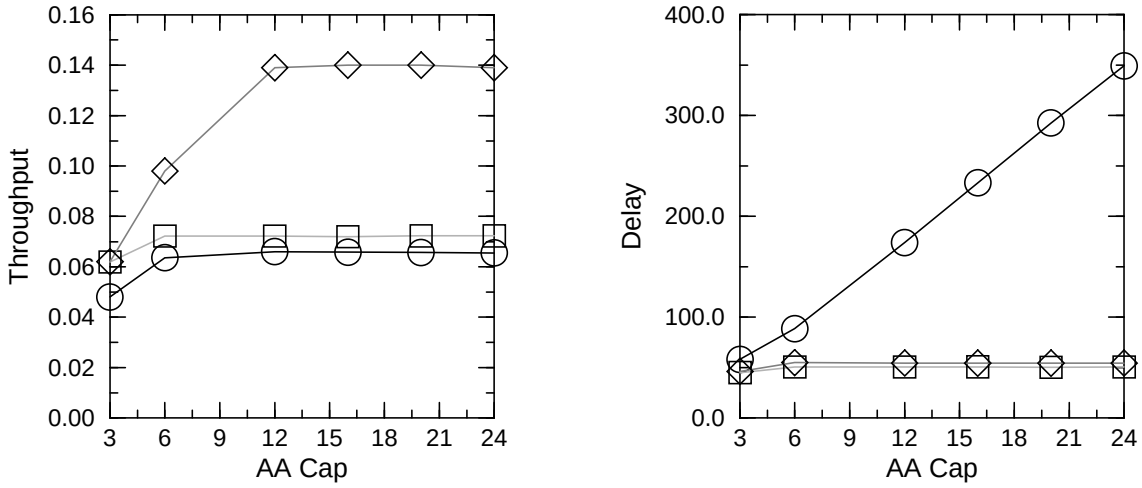
Delay in series 3 is the number of cycles from the time the atomic action is generated until the last response is received. Note that for conventional networks, the time required to release locks is not included in delay. A process generates a new atomic action whenever its output buffer is empty and the number of atomic actions it has started but not completed is fewer than a given number, aa_cap, supplied as a parameter. Initially we set aa_cap = unlimited for both isotach and conventional systems to allow unrestricted pipelining. We found however that allowing unlimited production of atomic actions in the conventional networks is counterproductive due to contention for locks. Latency rises steeply with no gain in throughput. This effect does not occur in the isotach systems, where locking is not necessary. We present data for C1 with aa_cap = 3. For I1 and I2, we show results for aa_cap = 3 and for aa_cap = unlimited.

Base Case - When each network is subject to an aa_cap of 3, throughput in the isotach networks is slightly higher in the base case and delay slightly lower than in C1. The performance of the uncapped isotach networks is better than the capped networks. When I2 is uncapped, its throughput is almost 3 times that of C1 and its delay remains lower than C1's.



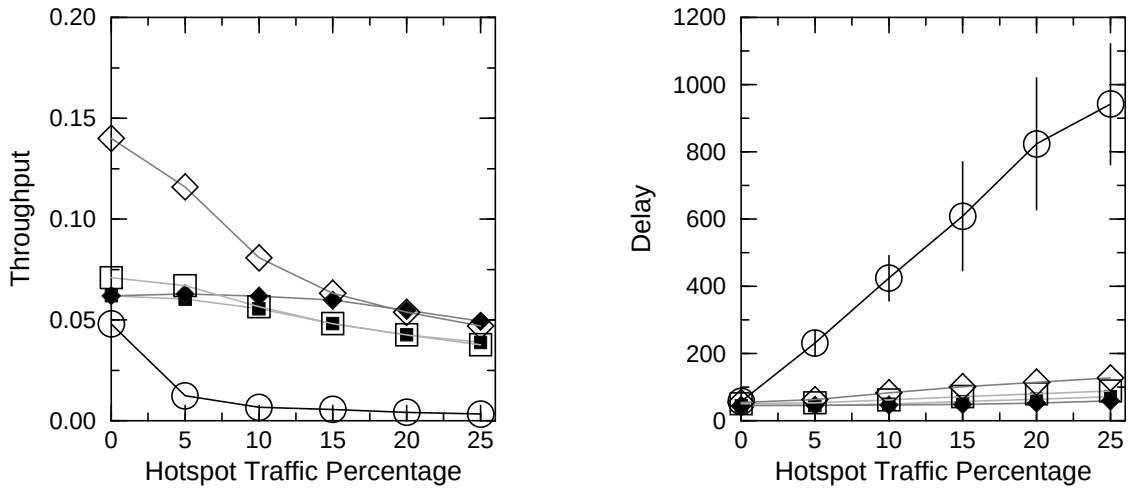
GRAPHS_6.3.3-scalability - Series 3 data for variation of network size.
O - C1; ❑ - I1; ◊ - I2; ■ - I1 (AA Cap = 3); ◆ - I2 (AA Cap = 3);

Scalability - Graphs 6.3.3-scalability show the effect of varying the number of network stages. With aa_cap = 3 for all networks, the graphs indicate that the performance of I1 and I2 becomes worse than that of C1 for large systems (stages > 9), but that the uncapped isotach networks continue to outperform C1. The graphs show that as the number of stages increases, throughput and delay gradually decrease. The delay reported in this part of series 3 is the delay normalized for the number of stages. The decrease in throughput and delay is attributable to the factors discussed under the scalability part of series 2. The graphs show an anomalous result for delay in large isotach networks: delay becomes worse when a cap is placed on the number of active atomic actions.



GRAPHS 6.3.3-AA_cap - Series 3 data for variation of network size. O - C1; ❑ - I1; ◊ - I2;

Atomic Action Cap - Graphs 6.3.3-AA_cap show the effect of varying the cap on the number of atomic actions a PE can have active at any one time. The graphs show C1 performs very poorly with a large aa_cap. C1 does not benefit from increasing the aa_cap because increasing the aa_cap also increases contention for locks. This contention is reflected in the delay for C1, which rises steeply as the aa_cap increases. Throughput for C1 levels off at about 6. For I1 and I2, throughput and delay rise as the aa_cap increases until the network is saturated (at about 6 for I1 and about 12 for I2) and thereafter remain constant**.**



GRAPHS_6.3.3-hot_spot - Series 3 data for variation of hotspot load.
O - C1; ❑ - I1; ◊ - I2; ■ - I1 (AA Cap = 3); ◆ - I2 (AA Cap = 3);

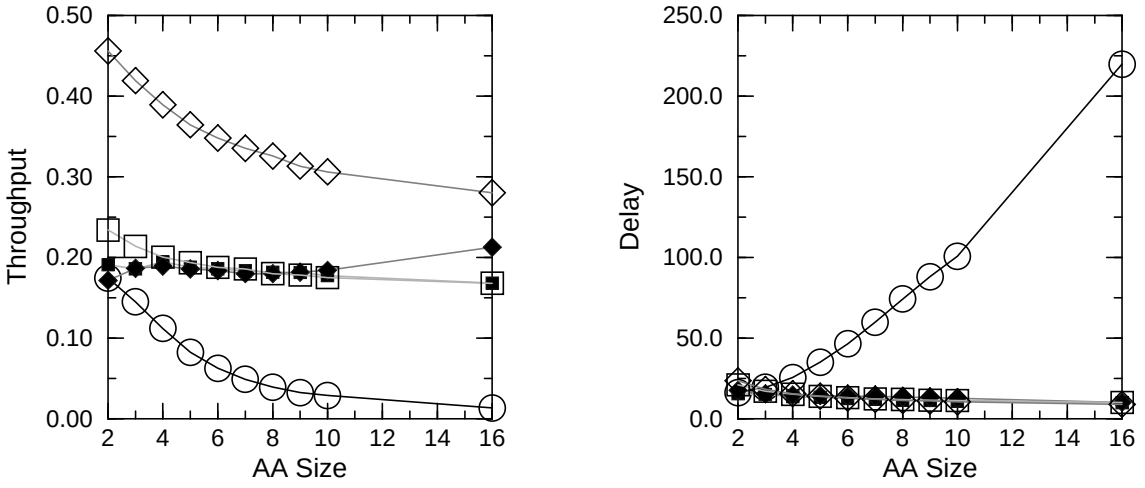
Hot-spot and Warm-spot Traffic - Graphs 6.3.3-hot_spot and Table 7 show network performance for the hot-spot and warm-spot traffic models, respectively. The results show that C1 performs increasingly poorly relative to I1 and I2 as the percentage of traffic going to hot and warm variables increases. C1 is more adversely affected by nonuniform access patterns because the use of locking as the means for enforcing atomicity magnifies the effect of contention. The throughput graph shows that until the percentage of hotspot traffic is about 15%, throughput for I1 and I2 is lower when aa_cap = 3 than when aa_cap = unlimited, indicating that the cap itself is the limiting factor. Thereafter the capped and uncapped networks have similar throughput, indicating that the hotspot traffic has become the limiting factor. We attribute the large confidence intervals exhibited by C1 to the fact that placement of the hot-spot variable changes with each trial, while the order in which locks are acquired remains the same.

Atomic Action Throughput	Low Sharing	Medium Sharing	Heavy Sharing
C1 (AA Cap=3)	0.0269 (.0267 - .0271)	0.0127 (.0125 - .0129)	0.0035 (.0034 - .0036)
I1 (AA Cap=3)	0.0611 (.0606 - .0616)	0.0599 (.0595 - .0603)	0.0415 (.0405 - .0427)
I2 (AA Cap=3)	0.0625 (.0615 - .0635)	0.0622 (.0615 - .0629)	0.0561 (.0539 - .0583)
I1 (No AA Cap)	0.0693 (.0682 - .0705)	0.0661 (.0649 - .0673)	0.0390 (.0371 - .0409)
I2 (No AA Cap)	0.1359 (.1341 - .1376)	0.1238 (.1214 - .1263)	0.0565 (.0527 - .0602)
Atomic Action Delay	Low Sharing	Medium Sharing	Heavy Sharing
C1 (AA Cap=3)	108.393 (107.537 - 109.250)	232.193 (228.971 - 235.414)	849.973 (817.718 - 882.228)
I1 (AA Cap=3)	45.751 (45.460 - 46.042)	46.711 (46.252 - 47.170)	67.186 (65.508 - 68.864)
I2 (AA Cap=3)	46.312 (45.635 - 46.989)	46.558 (46.004 - 47.111)	52.123 (49.918 - 54.328)
I1 (No AA Cap)	52.291 (51.664 - 52.918)	54.602 (51.664 - 52.918)	86.775 (82.979 - 90.570)
I2 (No AA Cap)	55.702 (55.156 - 56.249)	60.092 (59.208 - 60.976)	112.505 (105.842 - 119.168)

TABLE 7. Series 3 warm-spot data. 99% confidence intervals are shown in parentheses.

Atomic Action Size - Graphs 6.3.3-AA_size show the effect of varying the average size, aa_mean, of atomic actions. Throughput as reported in this part of series 3 is normalized for atomic action size, i.e., reported throughput is atomic action throughput times the average number of operations per atomic action. Thus reported throughput is the average number of operations arriving at each MM. Delay is also normalized. Delay is the atomic action delay divided by the average number of operations per atomic action. The results show C1 performs poorly for large atomic actions. As the atomic action mean size increases, C1's throughput drops and its delay rises steeply. The isotach networks, both capped and uncapped, outperform C1. The drop in throughput is more gradual and, instead of increasing, normalized delay actually decreases. The decrease is

attributable to the fact that operations in the same atomic action are delivered and executed concurrently in an isotach system, so delay per operation declines as the number of operations per atomic action increases. For large atomic actions, the isotach systems perform markedly better than the conventional systems. When the average atomic action size is 16, the delay in the conventional systems is greater than the delay in the isotach systems by a factor of about 22 and the throughput is less than the throughput in the isotach systems by a factor of about 12 (I1) to 16 (I2). The steep rise in delay in the conventional systems is attributable to the locking protocol. A process waiting to acquire a lock retains the locks it has already acquired for operations in the same atomic action. 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.

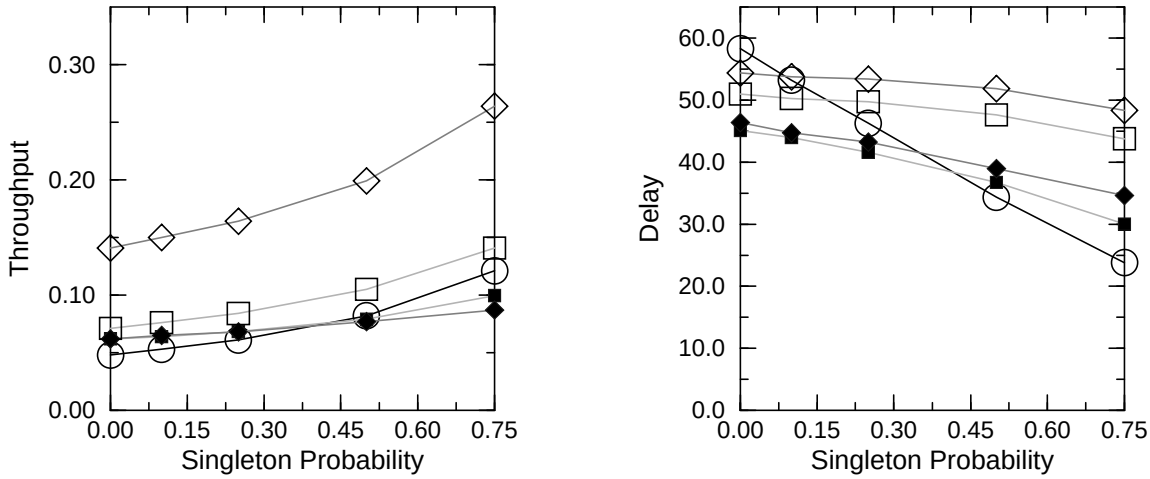


GRAPHS_6.3.3-AA_size - Series 3 data for variation of atomic action size.
O - C1; ❑ - I1; ◊ - I2; ■ - I1 (AA Cap = 3); ◆ - I2 (AA Cap = 3);

The capped versions of I1 and I2 have lower throughput than the uncapped versions when the atomic actions are too small to saturate the network.

Forced Singletons - Graphs 6.3.3-force_sing show the effect of varying the probability of forced singletons. Recall that the actual percentage of singleton atomic actions, i.e., atomic actions of size 1, is larger than the forced-singleton probability because some singleton atomic actions may be drawn from the exponential distribution of atomic action sizes. With an atomic action mean of 3, a forced singleton probability of 50% corresponds to a total probability of singleton

atomic actions of about 61%. The results show that network C1 performs well when the percentage of singletons is high. Since C1 does not enforce sequential consistency in series 3, or acquire any locks for singleton atomic actions, as the percentage of singletons increases, its performance becomes increasingly similar to its performance in series 1. The isotach networks are less sensitive to the percentage of singleton atomic actions. As the singleton percentage increases, their performance improves, but the rate of improvement in delay is less than that of C1 and, except for the uncapped version of I2, the rate of throughput is about the same as C1's. As a result, the isotach networks, with the exception of the uncapped version of I2, perform less well than C1 when the percentage of singletons is high.

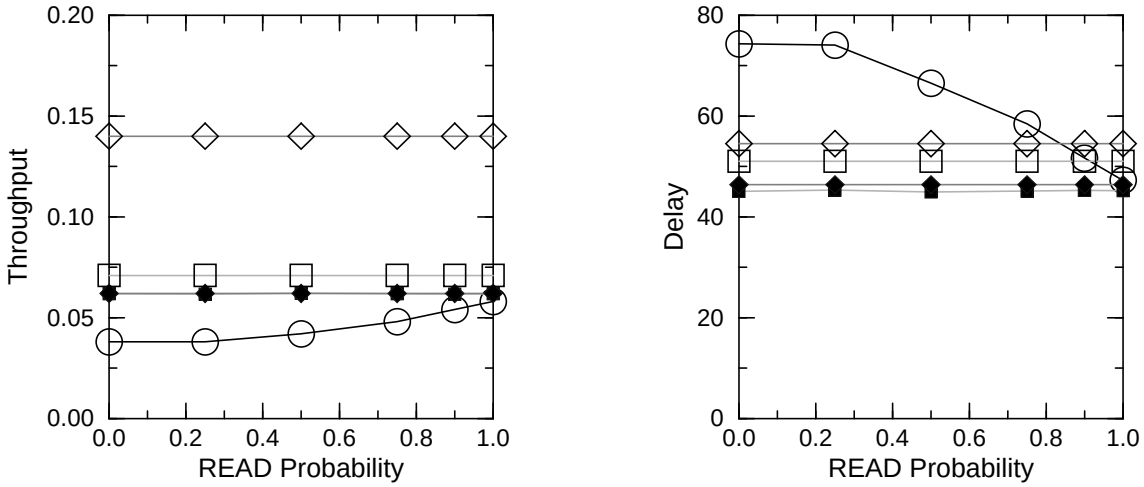


GRAPHS_6.3.3-force_sing - Series 3 data for variation of singleton probability.
O - C1; ❑ - I1; ◊ - I2; ■ - I1 (AA Cap = 3); ◆ - I2 (AA Cap = 3);

Read/Write Probability - Graphs 6.3.3-read_prob show the effect of varying the percentage of accesses that are READ operations from the base case setting of 75%. The conventional networks perform better as the percentage of read accesses increases because the read-locks required for READ's can be shared, whereas WRITE's require exclusive locks. The isotach networks are unaffected by the percentage of reads.

Series 3 shows that for a workload model that requires atomicity but not sequential consistency, the isotach networks outperform the conventional networks. Even with the handicap of also

insuring sequential consistency, the isotach networks are able to perform better than the conventional networks.



GRAPHS_6.3.3-read_prob - Series 3 data for variation of READ probability.
O - C1; ❑ - I1; ◊ - I2; ■ - I1 (AA Cap = 3); ◆ - I2 (AA Cap = 3);

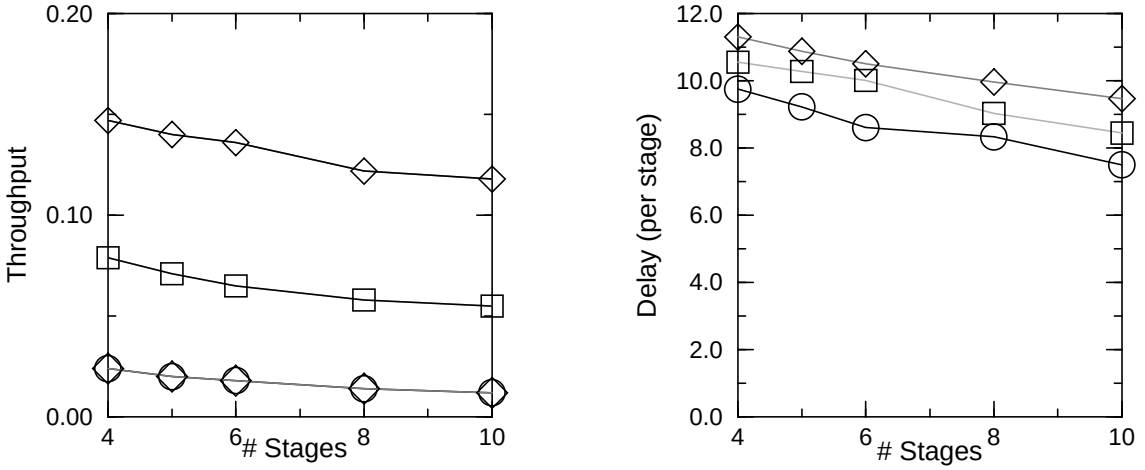
6.3.4 Flat Atomic Action Data

Series 4 compares the performance of the networks under a workload which requires both atomicity and sequential consistency. As in series 3, the atomic actions are assumed to be independent and flat, but in this series, the atomic actions must not only be executed atomically but also must appear to be executed in order.

The conventional networks ensure atomicity through the locking protocol discussed above. Sequential consistency is enforced by limiting the number of outstanding atomic actions per PE to one (aa_cap = 1) and by requiring processes to obtain locks for singleton atomic actions. In the isotach systems, by contrast, a PE may have any number of active atomic actions and it need not obtain locks.

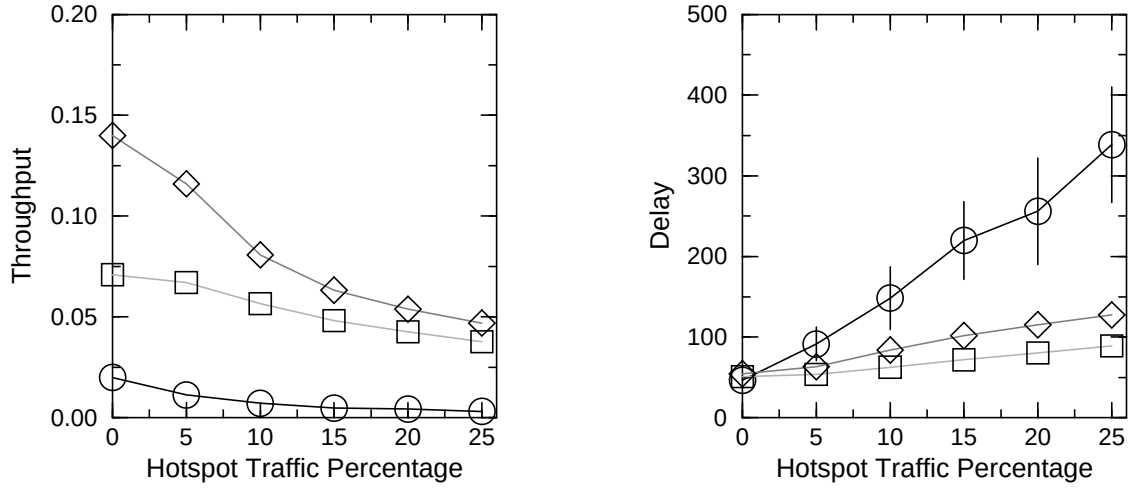
This series is very similar to series 3. Throughput and delay are measured and reported in the same way as in series 3. In terms of the simulation parameters, the only difference is in the setting of the aa_cap. For the conventional networks, aa_cap is reduced from 3 to 1. For the isotach networks, only the uncapped versions are relevant in this series. The results reported for I1 and I2 are identical to those reported in series 3 as the results for the uncapped versions of I1 and I2. Our report for series 4 is limited to the differences between series 3 and series 4.

Base Case - C1 has lower throughput and lower delay in series 4 than in series 3 because the cap on the number of atomic actions is lower. The lower cap reduces delay by reducing the load on the network and by reducing contention for locks. As a result, C1's delay is better in relation to I1 and I2 in this series than in series 3 and its throughput is worse. C1's per stage delay in the base case is slightly better than that for I1 and I2 (9.2 for C1 as opposed to 10.3 for I1 and 10.9 for I2), but its throughput is much worse: in the case of I1, by a factor of 3.5 and in the case of I2 by a factor of 7.



GRAPHS 6.3.4-scalability - Series 4 data for variation of network size. O - C1; ❑ - I1; ◊ - I2;

Scalability (Graphs 6.3.4-scalability) - As in series 3, throughput and delay ease gradually in both the isotach and conventional systems as the number of stages increases from 4 to 10.



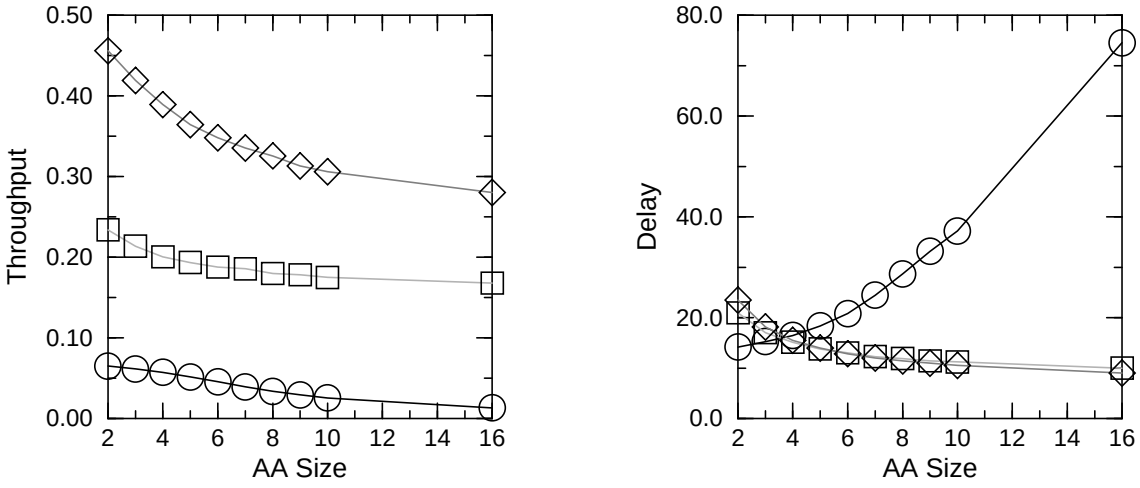
GRAPHS 6.3.4-hot_spot - Series 4 data for variation of hotspot load. O - C1; ❑ - I1; ◊ - I2;

Hot-spot and Warm-spot Traffic (Graphs 6.3.4-hot_spot and Table 8) - As in series 3, the results show that the advantage of isotach networks over conventional networks grows as the probability of hot spot and warm spot traffic increases.

Atomic Action Throughput	Low Sharing	Medium Sharing	Heavy Sharing
C1	0.0175 (.0175 - .0176)	0.0115 (.0115 - .0116)	0.0032 (.0031 - .0033)
I1	0.0693 (.0682 - .0705)	0.0661 (.0649 - .0673)	0.0390 (.0371 - .0409)
I2	0.1359 (.1341 - .1376)	0.1238 (.1214 - .1263)	0.0565 (.0527 - .0602)
Atomic Action Delay	**Low Sharing**	**Medium Sharing**	**Heavy Sharing**
C1	54.086 (53.907 - 54.264)	83.755 (83.185 - 84.325)	308.015 (296.555 - 319.475)
I1	52.291 (59.208 - 60.976)	54.602 (53.784 - 55.421)	86.775 (82.979 - 90.570)
I2	55.702 (55.156 - 56.249)	60.092 (59.208 - 60.976)	112.505 (59.208 - 60.976

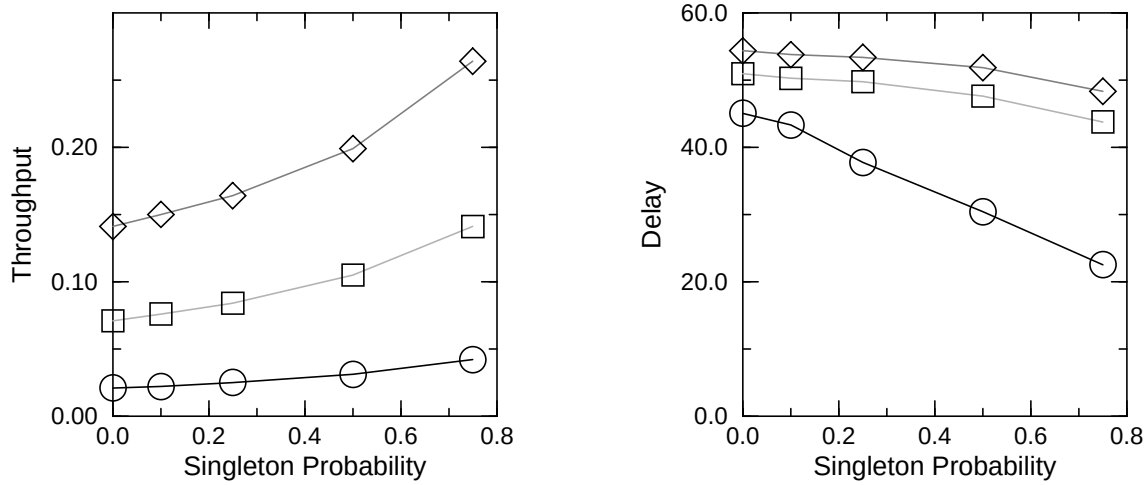
TABLE 8. Series 4 warm-spot data. 99% confidence intervals are shown in parentheses.

Atomic Action Size (Graphs 6.3.4-AA_size) - The delay in C1 is lower than is series 3, so the difference in delay between the conventional and isotach systems is less in this series. The delay in C1 is higher than that of I1 and I2 by a factor of about 7.5 instead of 22 for aa_size = 16.



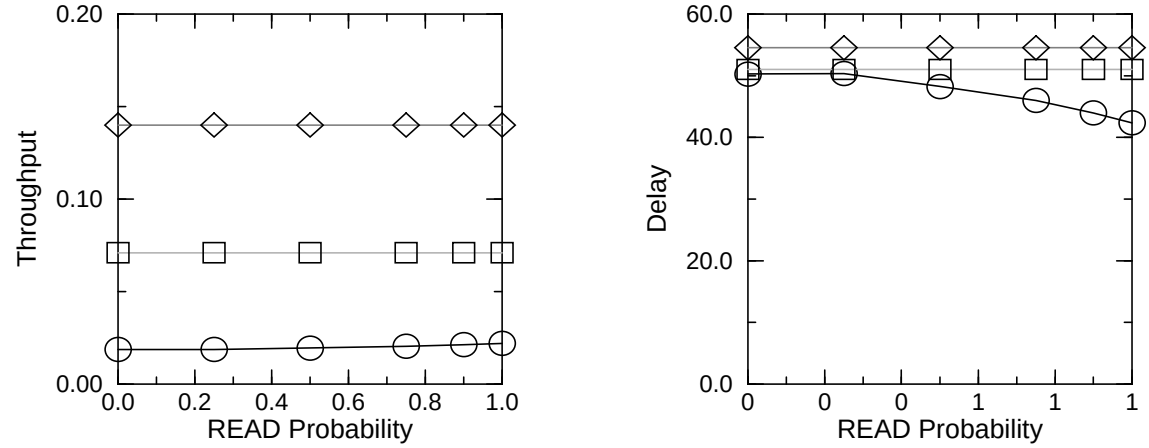
GRAPHS 6.3.4-AA_size - Series 4 data for variation of atomic action size. O - C1; ❑ - I1; ◊ - I2;

Forced Singletons. (Graphs 6.3.4-force_sing) - C1 does not show as great an improvement as the percentage of singletons increases in this series as in series 3. The reason C1 derives less benefit from singletons in series 4 is that singletons are subject to more restrictions. In series 4, singletons cannot be pipelined by C1 and must respect locks in the same way as ordinary accesses.



GRAPHS 6.3.4-force_sing - Series 4 data for variation of singleton probability. O - C1; ❑ - I1; ◊ - I2;

Read/Write Probability (Graphs 6.3.4-read_prob) - C1 is less sensitive to the proportion of reads and writes than in series 3. Reducing the aa_cap from 3 to 1 means fewer operations are contending for access to the same variable. In the base case of series 4 (25% WRITES) the percentage of lock requests that are granted immediately is 97 as opposed to 91% in series 3. Concurrently active operations access the same variable so infrequently in this series that increasing the percentage of writes has little effect on performance.



GRAPHS 6.3.4-read_prob - Series 4 data for variation of READ probability. O - C1; ❑ - I1; ◊ - I2;

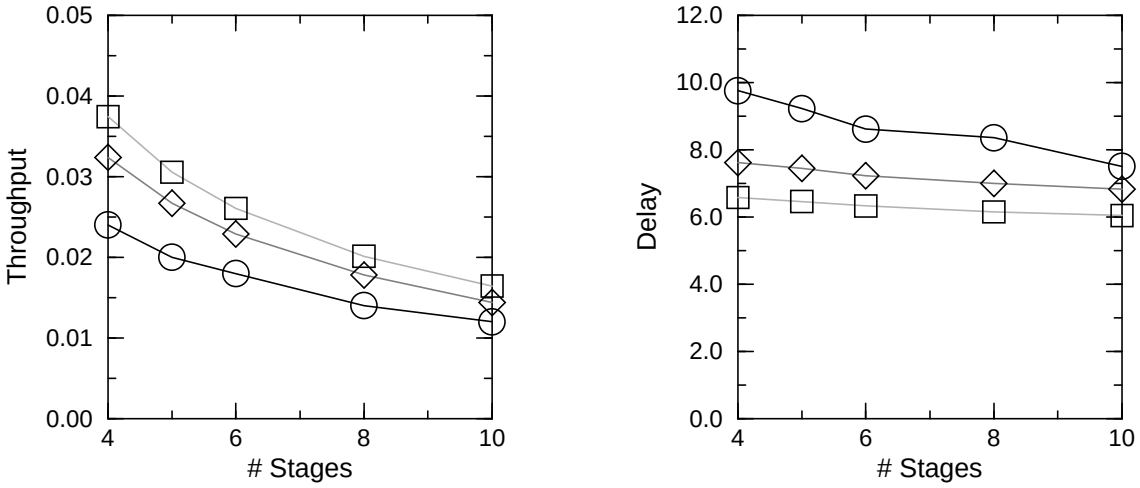
This series shows that the isotach systems perform very well and the conventional systems very poorly when each process's workload consists of a sequence of flat, independent atomic actions. The conventional systems cannot take full advantage of the greater raw power of their networks because the rate at which operations can be issued is severely limited by the locking protocol and the restriction on pipelining. The performance of the isotach systems, by contrast, is limited only by the performance of the raw power of the networks. Though the raw power of the 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.

6.3.5 Data Dependency Data

This series compares the performance of conventional and isotach systems for a workload model in which data dependencies exist both within and among atomic actions. The existence of data dependencies among atomic actions means that a process cannot issue an atomic action until the proceeding atomic action is complete. To simulate this type of data dependence, we set aa_cap =1 for all networks.

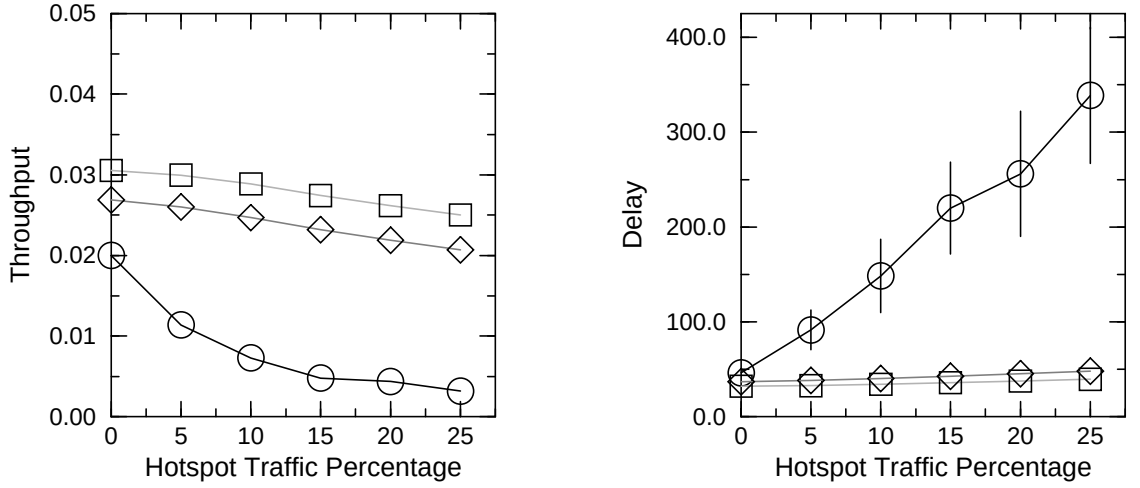
The data dependencies within atomic actions are assumed to be of a simple type: each WRITE depends on all the READ's in the same atomic action. As a consequence, a process cannot issue a WRITE until it has received responses to all the reads in the same atomic action. The introduction of data dependencies within atomic actions should make C1 very slightly less efficient than in series 4, but we assume for simplicity that C1's performance remains the same. The data for C1 in this series is copied from the data from series 4. For an isotach system the impact of data dependencies is much more significant since they prevent the isotach system from taking advantage of its ability to pipeline while maintaining sequential consistency. In the case of I1 and I2, the data dependencies within atomic actions are modelled by using the scheduling isochron technique in section 2 (access sequences and split operations) for every atomic action containing a WRITE.

Base Case - Both the throughput and delay are lower for the isotach systems than in series 4. Delay is lower because the networks are more lightly loaded. As noted above, the data for C1 is copied from series 4. I1 and I2 perform only slightly better than C1 in the base case. Throughput for C1 is lower (.020 for C1 instead of about .030 and .027 for I1 and I2) and delay per stage is higher (9.2 for C1 instead of 6.6 and 7.5 for I1 and I2).



GRAPHS 6.3.5-scalability - Series 5 data for variation of network size. O - C1; ❑ - I1; ◊ - I2;

Scalability (Graphs 6.3.5-scalability) - The throughput of C1 and I2 diminish gradually and at about the same rate. I1 scales slightly less well than I2 under this series's workload. The throughput of I1 drops faster than I2's and, instead of diminishing, the delay per stage for I1 increases very slightly.



GRAPHS 6.3.5-hot_spot - Series 5 data for variation of hotspot load. O - C1; ❑ - I1; ◊ - I2;

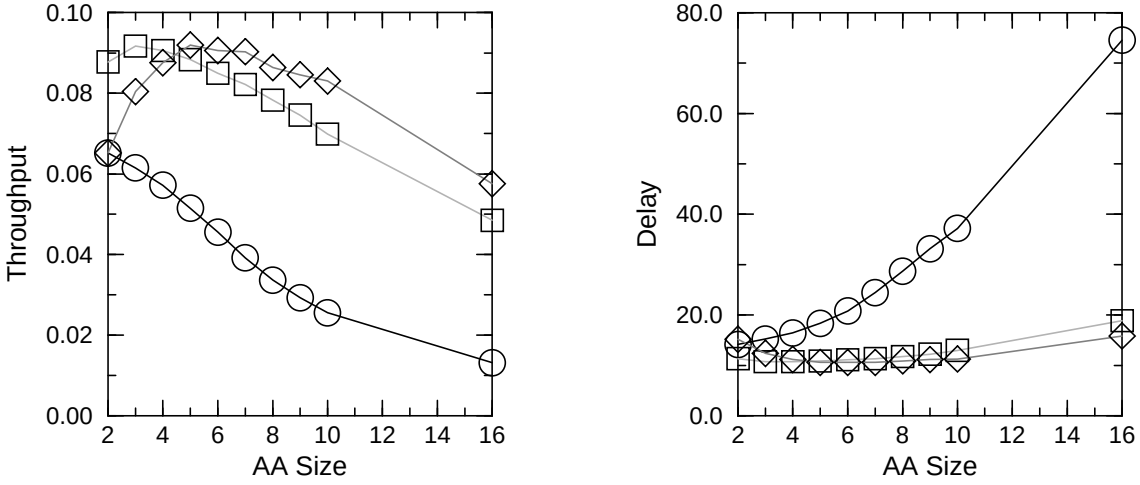
Hot-spot and Warm-spot Traffic (Graphs 6.3.5-hot_spot and Table 9) - Although the throughput for the isotach networks is much lower than in series 4, it remains higher than the throughput of C1. Delay for the isotach networks improves significantly over the delay in series 4 and is much lower than the delay for C1. When the probability of accessing the hot-spot is 10%, the throughput for I1 and I2 is about 3.5 times higher than C1's and delay is about 3.5 times lower. When the probability of hot spot access is high (0.25), throughput for I1 and I2 is higher than C1 by a factor of about 7.8 and 5.25 respectively and delay is lower by a factor of about 8.5 and 6. The warm-traffic data show the same pattern.

Atomic Action Throughput	Low Sharing	Medium Sharing	Heavy Sharing
C1	0.0175 (.0175 - .0176)	0.0115 (.0115 - .0116)	0.0032 (.0031 - .0033)
I1	0.0301 (.0297 - .0304)	0.0288 (.0287 - .0289)	0.0207 (.0202 - .0212)
I2	0.0260 (.0258 - .0262)	0.0245 (.0243 - .0247)	0.0163 (.0159 - .1067)
Atomic Action Delay	**Low Sharing**	**Medium Sharing**	**Heavy Sharing**
C1	54.086 (53.907 - 54.264)	83.755 (83.185 - 84.325)	308.015 (296.555 - 319.475)
I1	32.869 (32.259 - 33.479)	34.356 (33.844 - 34.868)	47.973 (47.331 - 48.615)
I2	38.246 (38.011 - 38.482)	40.642 (40.260 - 41.024)	61.089 (59.694 - 62.48

TABLE 9. Series 5 warm-spot data. 99% confidence intervals are shown in parentheses)

Atomic Action Size (Graphs 6.3.5-aa_size) - As in series 3 and 4, throughput and delay for this part are normalized for the atomic action size. As the atomic action mean size increases, the data show that throughput for the isotach networks initially increases, and then decreases, whereas normalized delay initially decreases and then increases. Throughput increases initially because increasing the atomic action size increases the number of operations presented to a network that is underutilized because of the cap of 1 on the number of active atomic actions. The subsequent decrease in throughput can be attributed to several factors:

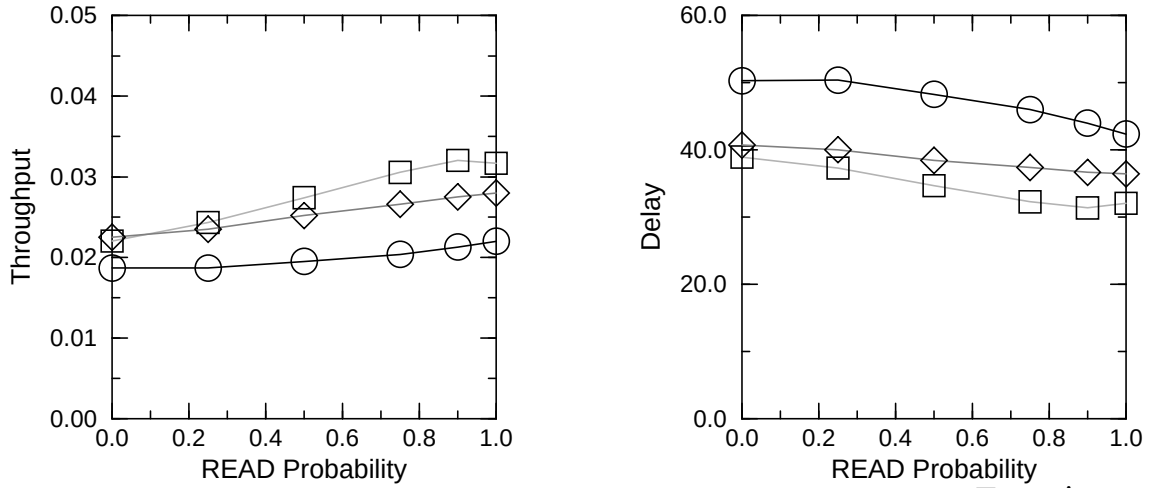
1. the network becomes less efficient as the size of the atomic actions increases;
2. the percentage of atomic actions that include a WRITE increases with the result that a greater percentage of atomic actions must be executed in 2 stages; and
3. more operations are accessing the same number of variables with the result that the probability a READ waits on an unsubstantiated SCHED increases.



GRAPHS 6.3.5-AA_size - Series 5 data for variation of atomic action size. O - C1; ❑ - I1; ◊ - I2;

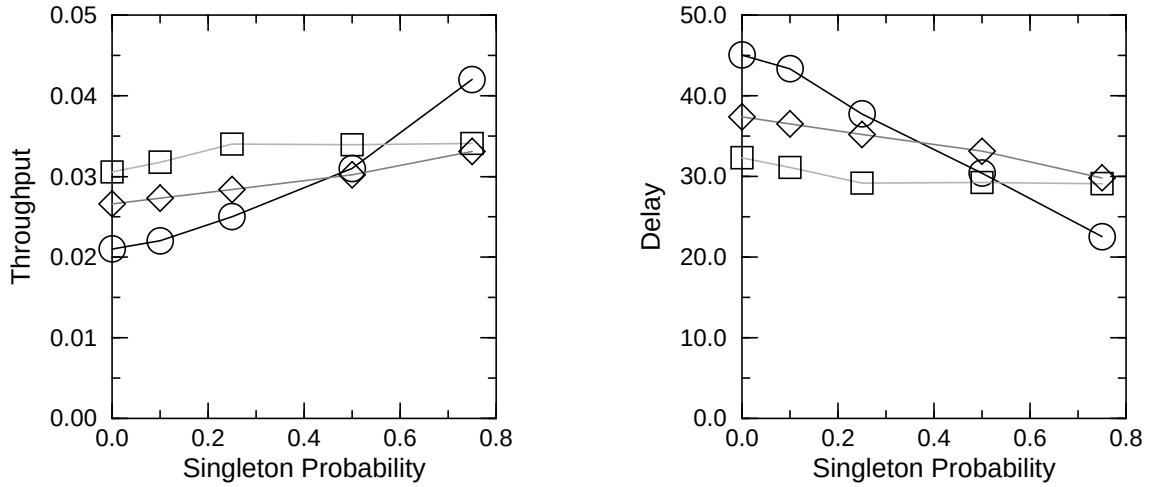
Delay increases for the reasons throughput decreases. The initial very small decrease in delay occurs because operations in the same atomic action are issued and executed concurrently so delay per operation tends to decrease as the number of operations per atomic action increases. Although the efficiency of I1 and I2 is much lower than in series 3 and 4, the isotach networks still outperform the conventional networks by a margin that increases as the atomic action size increases. When the atomic action mean size is 16, throughput is about 3.5 to 4.5 times higher for I1 and I2 than for C1 and delay about 4 to 5 times lower.

Read Probability (Graph 5-read_prob) - The efficiency of each of the systems increases gradually as the probability of a read access increases. None of the systems is very sensitive to the percentage of reads because the probability that operations concurrently access the same variable is low, given the cap of 1 on the number of active atomic actions per PE and the uniform distribution of accesses.



GRAPHS 6.3.5-read_prob - Series 5 data for variation of READ probability. O - C1; ❑ - I1; ◊ - I2;

Forced Singletons. (Graphs 6.3.5-force_sing) - This data shows that isotach networks continue to outperform conventional networks even under the performance limiting effects of this series until the forced-singleton probability reaches about .5 (meaning the percentage of singleton accesses is about 60%).



GRAPHS 6.3.5-force_sing - Series 5 data for variation of singleton probability. O - C1; ❑ - I1; ◊ - I2;

Series 5 shows that isotach systems can outperform conventional systems even for workloads in which the existence of data dependencies negates one of the principal advantages of isotach systems, the ability to pipeline operations while maintaining sequential consistency. The series indicates that the scheduling isochron technique is more efficient than 2PL for structured

atomic actions of the type simulated, especially for large atomic actions or nonuniform access patterns.

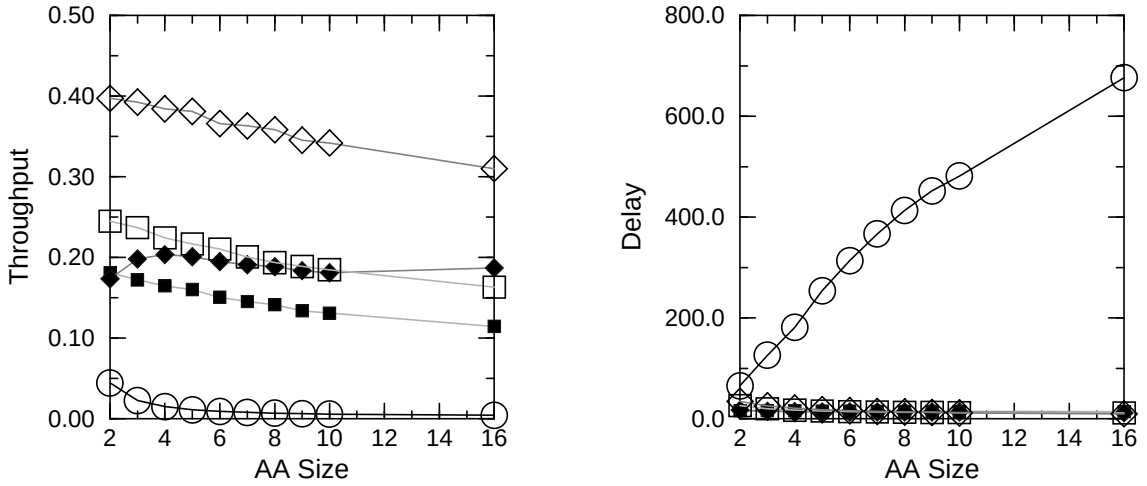
6.3.6 Warm Traffic Data

Series 6 compares the performance of the networks when accesses are not uniformly distributed. This series repeats parts of earlier series using an 80/20 rule warm traffic model in place of the uniform traffic model. We include this series to explore the effect of contention for shared variables in conventional and isotach systems.

This series also differs from earlier series in that it uses a switch buffer size of 2 for the isotach networks. C1 is slightly more efficient with a switch buffer size set at 1 instead of 2 and the isotach networks are slightly more efficient at 2 instead of 1. In all previous series, the switch buffer sizes are all set at 1. In this series we compare the conventional and isotach systems when each has its optimal switch buffer size: 1 for C1 and 2 for I1 and I2.

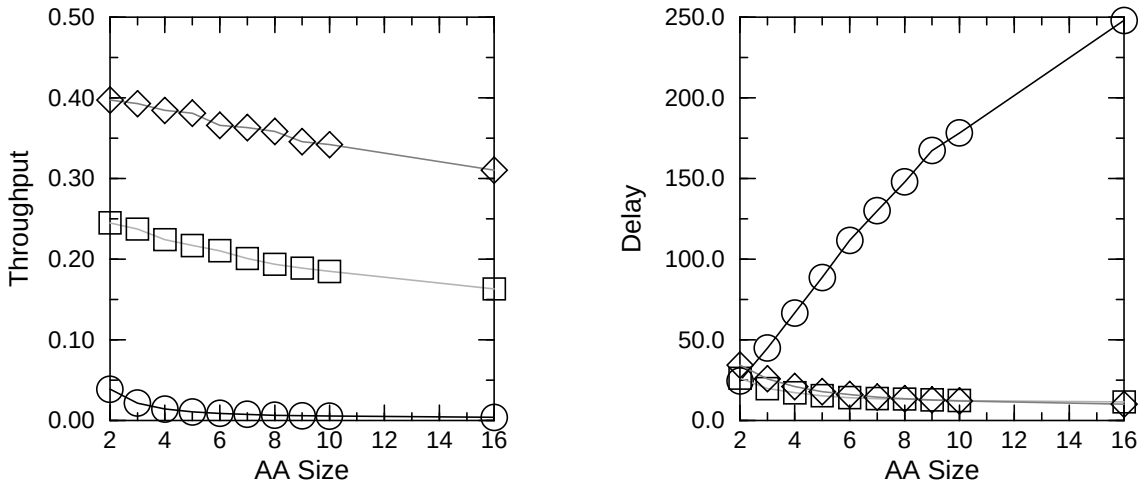
In the first part of series 6 (series 6.1), the workload model is the *atomicity only* model from series 3. In the second part (series 6.2), the workload model is the *flat atomic action* model from series 4.

Base Case - In the base case for series 6.1 (atomicity only), the throughput of the capped version of I2 is higher than C1's by a factor of about 9 and delay is about 8 times lower. In the base case of series 6.2 (flat atomic actions) throughput of I2 is 18 times higher than C1's and delay about half that of C1. When accesses are not uniformly distributed the margin by which isotach networks outperform conventional networks increases. The performance of the isotach networks in series 6.1 and 6.2 is similar to their performance in series 3 and 4. The throughput is essentially the same and the delay only slightly higher. The increase in the buffer size helps offset the decrease in throughput caused by the non-uniform distribution of accesses. The performance of C1, by contrast, is considerably worse in series 6. When the probability of conflicting operations is low, locks have less impact on performance than when it is high. The use of locks to enforce atomicity means C1's performance suffers when accesses are not uniformly distributed.

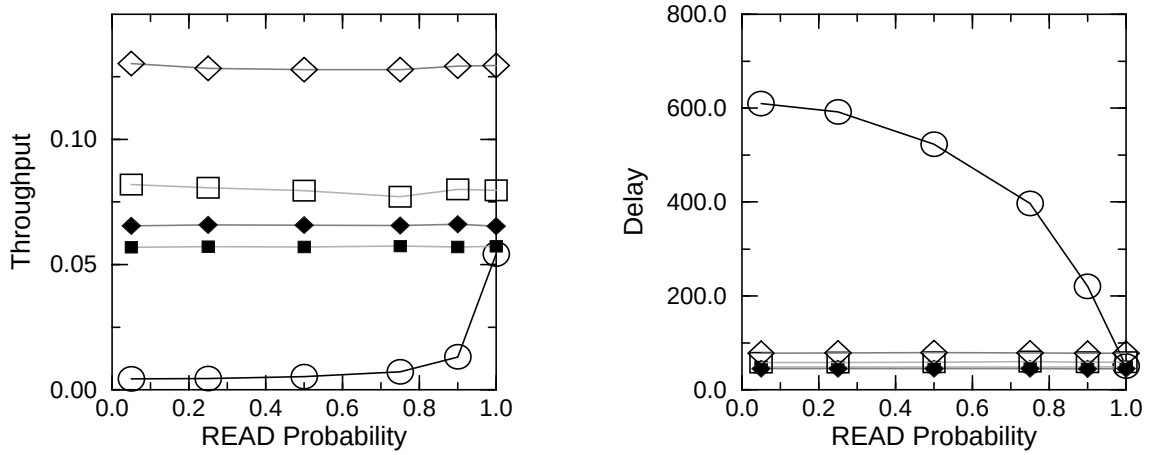


GRAPHS 6.3.6.1-AA-size - Series 6.1 data for variation of the atomic action size. O - C1; ❑ - I1; ◊ - I2; ■ - I1 (AA Cap = 3); ◆ - I2 (AA Cap = 3);

Atomic Action Size - Graphs 6.3.6.1-AA_size and 6.2-AA_size show the effect of varying the atomic action size under a warm-traffic model. These graphs correspond to Graphs 6.3.3-AA_size and 4-AA_size, respectively. When atomic actions are large and traffic is non-uniform, C1's performance is very poor relative to the isotach networks. For large atomic actions (aa_size = 16), the throughput in series 6.1 of the capped version of I2 is about 43 times higher than C1's and delay about 57 times lower. In series 6.2, I2's throughput for large atomic actions is about 78 times higher than C1's and its delay about 24 times lower.

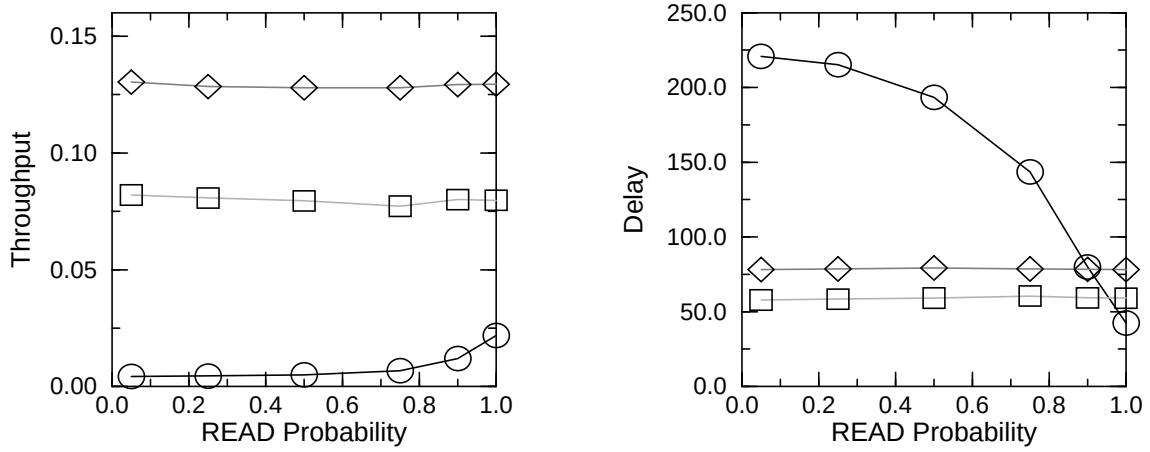


GRAPHS 6.3.6.2-read_prob - Series 6.2 data for READ probability variation. O - C1; ❑ - I1; ◊ - I2;



GRAPHS 6.3.6.1-read_prob - Series 6.1 data for variation of the READ probability. O - C1; ❑ - I1; ◊ - I2; ■ - I1 (AA Cap = 3); ◆ - I2 (AA Cap = 3);

Read Probability - Graphs 6.3.6.1-read_prob and 6.2-read_prob show the effect of varying the probability that an access is a READ. The isotach networks are not sensitive to the READ probability, but C1 is, and to a much greater extent than under the uniform traffic model. The effect on C1 is much greater under a warm traffic model because the probability that two operations conflict is significantly higher than when accesses are uniformly distributed. In the base case (read_-prob = 75%), the percentage of locks granted immediately declines from 91% in series 3 to 81% in series 6.1 and from 97% in series 3 to 83% in 6.2. When the READ probability is low (25%), the throughput in series 6.1 of the capped version of I2 is higher than that of C1 by a factor of about 15 and its delay is lower by a factor of about 13. The comparable figures for series 6.2 are 29 for throughput and 2.7 for delay.



GRAPHS 6.3.6.2-read_prob - Series 6.2 data READ probability variation. O - C1; ❑ - I1; ◊ - I2;

Series 6 indicates that isotach systems outperform conventional systems by a wider margin when accesses are not uniformly distributed.

6.3.7 Conclusions from the Simulation Study

We have presented results from a simulation study, performed jointly with Craig Williams [RWW92], comparing the performance of locally synchronous isotach and conventional networks. The study shows conventional networks have higher raw power than isotach networks, but that under a workload of operations with atomicity and sequencing constraints, isotach networks outperform conventional networks. Isotach networks perform best in relation to conventional networks under workloads with the following characteristics:

1. Execution is required to be sequentially consistent;
2. The distance between data dependent operations within the same process's program is large enough to allow operations to be pipelined;
3. Atomic actions are large; and
4. Contention for shared variables is high.

When the workload has two or more of these characteristics, isotach networks perform markedly better than their conventional counterparts. In some cases, the improvement in both throughput and delay is more than ten-fold.

The locally synchronous systems outperform the conventional systems in spite of the lower raw power of the isotach networks. A conventional system cannot take advantage of the higher throughput and lower delay of its network when the synchronization techniques it uses work by restricting throughput and imposing delays. In conventional systems, the limiting factor on execution speed is not the network, but restrictions and delays imposed by the synchronization techniques. In isotach systems, the limiting factor is the network itself. Atomicity and sequencing constraints have markedly less effect on execution speed in isotach than in conventional systems.

The simulation results indicate that network throughput is of more importance relative to network latency in its effect on system performance in isotach systems than in conventional systems. Isotach systems are better able to take advantage of high throughput and are better able to use pipelining to mask latency. The high-throughput z-switch isotach network (I2) in most cases

outperforms the lower throughput isotach switch (I1) in spite of the assumption that I2's cycle time is twice that of I1. In designing conventional networks, however, the simulation indicates that accepting worse delay for better throughput is a poor decision. The high-throughput z-switch C2 outperforms all other networks in raw power, but consistently performs worse than the simpler conventional switch C1 under a workload with synchronization requirements.

The most important assumptions made by the study are as follows: 1) the cycle time for each isotach network switch is the same as that of the corresponding conventional switch and the cycle time for a z-switch is twice that of a single-cycle switch -- in particular, that the latency of the isotach network which performs best (I2) is twice that of the conventional network that performs best (C1); 2) the workload models simulated are relevant to actual parallel computations; and 3) the conventional system simulated is a good basis for a fair comparison of conventional with isotach systems. We believe that these assumptions are realistic.

6.4 Alternate Implementations - Sorting at the MM

In a local synchrony implementation, an access is executed only when there can exist no unexecuted accesses to the same memory location with earlier timestamps. Accesses bound for the same memory location in the same ltu must be executed in *pid* and *tock* order to meet this implementation requirement. The *pid* ordering guarantees atomicity for isochronic accesses, while the *tock* ordering guarantees sequential consistency for accesses to the same memory location, from the same PE. We assume here a system using explicit tokens, and thus accesses are already sorted by the *ltu* component of their timestamps.

The *pid* and *tock* ordering can generally be accomplished in one of two ways. The first is for the MMs to buffer all accesses in a given ltu, and then sort them before execution. This method requires that only the PEs and MMs act in a locally synchronous manner. The second is for the PEs to sort the accesses of a given ltu by *pid* and *tock* before emission. In this case, the switches would then merge sorted streams of accesses from each PE. We consider the second alternative to be simpler and more efficient, for reasons discussed below. Note that local synchrony requires that

accesses may not be aborted once they have been emitted from a given PE. Therefore, any architecture implementing local synchrony must include at least single access buffering on the input channels of each switch and MM.

Sorting at memory nodes requires that MMs be able to buffer the maximum number of accesses that might be received in any given ltu. If PEs and switches do not keep accesses sorted within ltus, an access to a given memory location might arrive during the processing of other accesses in its ltu. Since accesses from different PEs to the same memory location arriving during the same ltu must be ordered, the MM must have all the accesses which arrive during a given ltu buffered in order to perform its sort. Meeting this buffering requirement would involve setting a limit on the number of accesses allowable in a given isochron, or on the number of accesses per isochron. For example, if one limited the number of accesses emitted per isochron per PE to *k* then each memory node would require *k*a*n* bytes of buffer space, where *a* is the size of an access in bytes, and *n* is the number of processing nodes. While the architecture can only generate *kn* accesses per ltu, *all* of those accesses might be bound for the same MM, generating the buffer requirement stated above. This does not include considerations for multiple processes operating on a single processing node.

Note that scalability of a system is compromised in this case. To keep the same ability to process isochrons requires that buffer sizes be increased at each MM when adding PEs to the system. Since each PE can generate up to *k* accesses bound for a given MM in any ltu, *ka* bytes additional buffer space would be needed at *each* MM for each PE added to the system.

In the case of sorted order transmission, each processing node would have to be able to buffer only the accesses that the node itself was generating in a given logical time unit. Compile-time processing could be used to generate most accesses in order. Memory units would perform a sorted merge among input channels, needing to buffer only one access from each input at a time. The unsorted transmission system discussed above used *kan* bytes of buffer space per MM (a total of *k*a*n*m* across all MMs, *m* representing the number of MMs) to allow PEs to generate *k*

accesses per ltu. The sorted transmission scheme would allow the same *k* accesses per PE per ltu while requiring at most *ka* bytes of buffer space per PE (a total of *k*a*n* across all PEs). In the event that isochronic accesses can be issued in order, a PE would require only a single cell output buffer.

In other words, situating the same amount of buffer space at the PEs would result in significantly better utilization of buffer space. If accesses could be generated in order, the number of accesses per logical time unit, and thus per isochron, would be theoretically unlimited. Scalability is maintained, as adding PEs requires only that each new PE be supplied with the same amount of buffer space (*O(k)*) as each other PE already in the system. Single cell input buffers are required at each switch node in either case.

A second point in favor of sorted transmission of accesses relates directly to the first. While the complexity of the sorting operation is unchanged, a larger number of computational elements may be applied to the problem by using the communications network and the sorted transmission scheme. As might be expected, the use of a larger number of computational elements means more opportunities for parallelism.

The third point in favor of sorted transmission is that combining of operations can be made much more efficient. If the *combining* component is added to the timestamp, as discussed in section 2.8, combinable accesses are guaranteed to meet at the heads of their respective input queues during the sorted merge operation at the switch nodes and MMs. This enhancement would come without compromising the goal of local synchrony: to provide sequentially consistent, atomic serial schedules for the global memory accesses of parallel programs. Combining would be enhanced by eliminating the need for queue search in order to find combinable operations, and ensuring that combinable accesses in a given logical time unit would be combined.

These arguments, especially those relating to scalability, indicate that sorted transmission would be the best solution for a local synchrony implementation, assuming that either approach could be implemented with relatively comparable time efficiency. Clearly, if the objective is to

maximize k, the number of accesses emitable per ltu per PE, without scalability limitations, performing a sorting operation at the PEs and a sorted merge at the switch nodes would be the better approach. While we do not intend to abandon research into systems which sort accesses at memory, we believe that our buffer space and scalability arguments make true local synchrony a better alternative.

6.4.1 Simulation Results for MM Sorting

Atomicity and sequential consistency are guaranteed in current systems and architectures by using locking protocols and limiting access pipelining by processes. The implementation of local synchrony presented here is an alternative to these solutions which provides better performance when the need for atomicity and sequential consistency is significant relative to the number of shared memory accesses performed.

Using the paradigm of logical timestamps to enforce atomicity and sequential consistency, there are several ways to implement an access ordering system which will provide a serial schedule of accesses to global memory in the same way as local synchrony. In this discussion we compare local synchrony to a somewhat different access ordering approach, which might be thought to be able to achieve better performance.

Better network performance can be achieved by doing some or all of the access ordering at the memory modules themselves. Although this would not be a scalable or efficient solution with respect to memory, it would result in faster transfer of accesses through the communications network. However, the time costs of buffering and sorting accesses at the MMs outweigh the network speedups as isochron size increases. We make the following assumptions in order to simplify the discussion:

1. MMs are capable of performing one global memory access per network switch cycle.
2. MMs are capable of doing an insert (of an access into the sorted list of accesses currently held) in one switch cycle. Note that an MM cannot generally receive more than a single access per switch cycle, so faster sorting has no benefit.
3. MMs are capable of performing accesses for a given ltu while receiving the accesses for the next ltu.

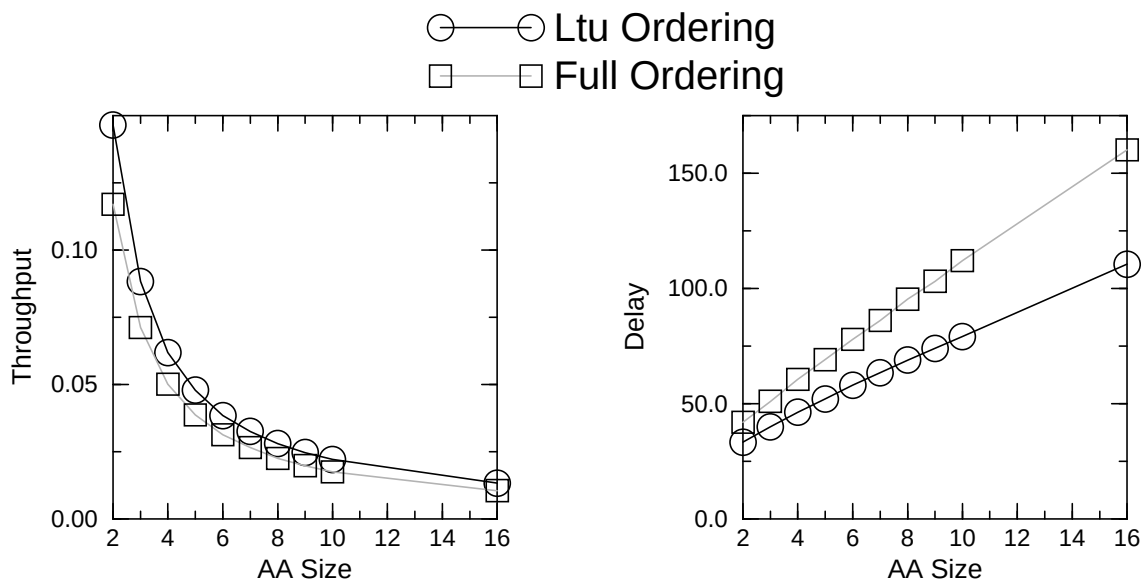
We feel that assumption 2 above is highly optimistic. It provides an implementation which utilizes MM sorting with a significant advantage.

Consider an implementation in which accesses are sorted by ltu only as they cross the network. In such an implementation, a switch would pass accesses without regard to timestamps, subject only to the caveat that the switch cannot pass a token until it holds tokens on each of its inputs, at which time it would pass a token on each of its outputs and flush all tokens on inputs. Accesses arriving at a given MM would be inserted into a sorted list as they arrived (1 per switch cycle). The MM could begin processing accesses for a given ltu only when all accesses due to be performed during that ltu were received (the last received access might still be the first to be executed). We will call this implementation *Ltu Ordering.*

Graphs 6.4.1 show simulation results for a five stage network architecture using ltu ordering in I1-like switches, in comparison with the data for our I1 implementation, which we call *Full Ordering*. Note that figure 35 presents only network performance data, without regard to actual delays due to accesses being held by the MM.

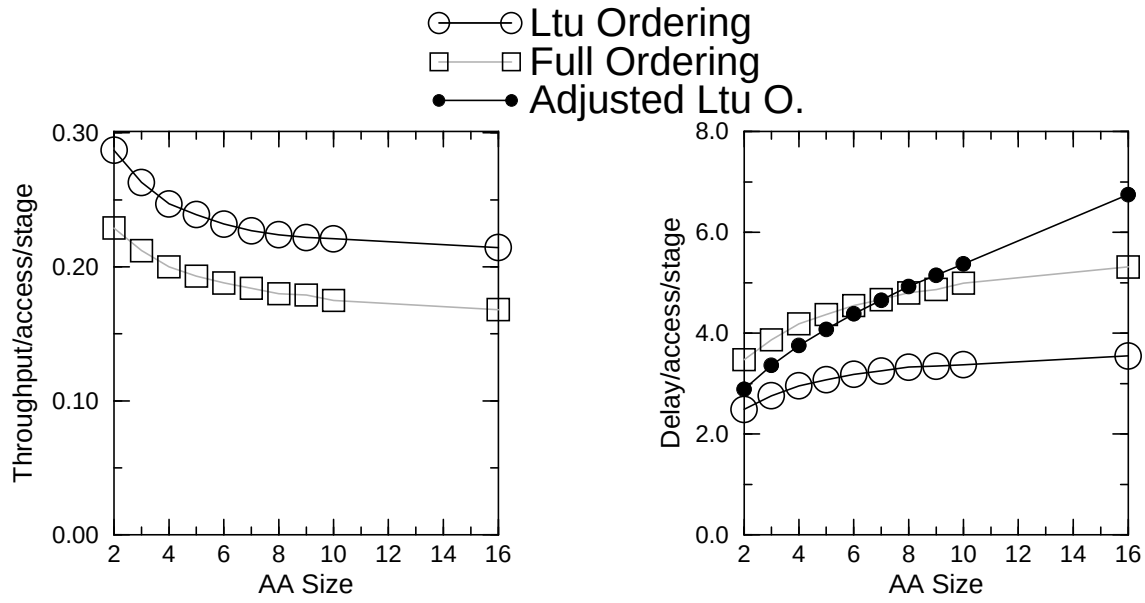
Ltu ordering shows a throughput improvement of about 15% for small sized isochrons, which decreases somewhat as isochron size increases. Isochron delay grows at a slower rate for ltu ordering, which shows approximately a 30% advantage at isochron size 16.

Graphs 6.4.1 show a marked improvement in network performance for ltu ordering over full ordering. However, if we take into account the waiting times necessary for accesses to be executed in ltu ordering, we see different results. Because each access within a given ltu must be held until all the accesses to be executed during that ltu have arrived, an extra delay is imposed on each access, on average equal to the number of accesses that the MM will process during the current ltu. In the simulations, which use a random distribution of global memory locations for accesses, this number averages out to the average number of accesses released by each PE during each ltu, or the average isochron size.



GRAPHS 6.4A - Network performance comparison between ltu ordering and full ordering.

Graphs 6.4.2 factor this additional delay into the performance data presented in graphs 6.4A. In graphs 6.4.2, the data is presented relative to each access, rather than each isochron (throughput is multiplied and delay is divided by the average number of accesses per isochron). The extra delay associated with ltu ordering is divided across all the network stages.



GRAPHS 6.4B Performance data with adjusted delay.

With the assumption that an MM is able to perform the accesses of a given ltu while receiving the accesses for the next, throughput is not affected by holding delays at memory. However, the average delay of an access from the time it leaves the PE until the time it is executed rises

much more steeply than full ordering as the isochron size increases, showing approximately a 28% increase for an average isochron size of 16.

This data does not take into account the benefits of FIFO combining, which is implementable in the sorted transmission approach but not in the memory sorting approach. We expect that in high concurrency situations, where hotspots are likely, FIFO combining will further benefit the performance of the sorted transmission model.

6.5 Summary

In this chapter, we presented the results of simulation and analytical studies of local synchrony implementations. We first described the network and switch architectures which would be used in the studies.

For analytical models, we used mean value analysis to model the raw power of local synchrony implementations. Our analytical models extend the probabilistic method to more complex systems involving multiple message types. Our analytical models agree closely with the raw power results of our simulation studies.

The simulation studies, performed jointly with Craig Williams [RWW92], compare the performance of local synchrony implementations with conventional implementations for different types of workloads, based on various levels of concurrency control requirements. The study shows conventional networks have higher raw power than locally synchronous isotach networks, but that under a workload of operations with atomicity and sequencing constraints, isotach networks outperform conventional networks.

Also we discussed local synchrony in comparison with MM sorting, another approach to concurrency control through logical timestamp ordering. We presented resource requirements, efficiency concerns, and simulation data which show that local synchrony is more efficient than MM sorting when atomic actions are large.

7.0 Conclusions

7.1 Research Review

We have studied the implementation of a low-level concurrency control mechanism, local synchrony, on general-purpose multi-processing architectures. We have further investigated performance characteristics of specific implementations, as well as some fault-tolerance aspects of the general implementation.

Our research goals in this work were threefold. Our first goal was to develop a correct, deadlock-free method for implementing local synchrony on a variety of interesting, asynchronous multi-processing systems. We first presented a low-level implementation plan for the NYU Ultracomputer architecture. Our plan included implementation details and algorithms for operation, as well as a proof of correctness of deadlock freedom. Second, we presented a low-level switch design for such an architecture that is realizable with current technology, and an alternate proof of deadlock freedom using inherent knowledge rather than ghost messages.

Finally, we developed a general implementation plan, which allows local synchrony to be implemented on any connected architecture (as defined in chapter 4). Our approach was to define a class of directed graphs, LS graphs, which represent architectures on which local synchrony can be correctly implemented, and then to present an algorithm which creates an LS graph from any shared memory architecture. The target architecture then uses virtual channels [DaS87] to emulate the LS graph topology. Our general implementation uses less memory and provides greater opportunities for parallelism than other approaches, and is proven correct and deadlock-free. The proof of correctness of the general implementation of local synchrony is a significant addition to the theory of parallel processing.

Our second goal was to assess some of the fault-tolerance capabilities of the general local synchrony implementation. We discussed the preservation of coherent system state within a local synchrony implementation in the presence of faults, and presented a simple system by which such an implementation could tolerate faults by using redundant hardware communication paths. We

presented and proved correct the local synchrony checkpointing and rollback procedure, which can be used to perform logically synchronized rollbacks of the general local synchrony implementation. Finally, we showed that under certain conditions it is possible to limit the number of saved checkpoints.

Our third goal was a simulation and analytical study of performance of local synchrony implementations. Extending the probabilistic method of Jenq [Jen83], we developed several analytical models of local synchrony implementations that predict the raw power of the locally synchronous communication networks studied (switch architectures I1 and I2—section 6.1). These predictions enable us to validate the performance predictions of our simulations. The analytical models also extend the probabilistic method to more complex types of systems than previously investigated.

Our simulation study (performed jointly with Craig Williams [RWW92]) compares the performance of locally synchronous isotach [WiR91] networks to similarly constructed conventional networks. We present data for a conventional single-stage switch architecture (switches C1 and I1—section 6.1), and for a two-stage, high-throughput architecture (switches C2 and I2). The study considers in detail the performance of the various implementations under sequencing and atomicity constraints. We contend that our assumptions about cycle times and workload models are valid, and that the conventional systems simulated constitute a good basis for fair comparison between conventional and locally synchronous isotach systems.

As expected, the raw power of the isotach networks is less than that of conventional networks. However, under a workload of operations with atomicity and sequencing constraints, the locally synchronous networks outperform the conventional networks by significant factors. Isotach networks perform best in relation to conventional networks under workloads with the following characteristics:

1. Execution of global memory accesses is required to be sequentially consistent;
2. The distance between data dependent operations within the same process's program is large enough to allow pipelining of operations;

3. Atomic actions are large; and
4. Contention for shared variables is high.

When the workload has two or more of these characteristics, the locally synchronous implementations perform markedly better than the conventional implementations. In some cases, the improvement in both throughput and delay is more than ten-fold.

The locally synchronous systems outperform the conventional systems in spite of their lower raw power. A conventional system cannot take advantage of the higher throughput and lower delay of its network when the synchronization techniques it employs work by restricting throughput and imposing delays. In conventional systems, the limiting factor on execution speed is not the network, but restrictions and delays imposed by synchronization techniques. In locally synchronous isotach systems, the limiting factor is the network itself. Atomicity and sequencing constraints have markedly less effect on execution speed in locally synchronous implementations.

The simulation results indicate that, relative to network latency, network throughput is more important in its effect on system performance in locally synchronous isotach systems than in conventional systems. Isotach systems are better able to take advantage of high throughput and are better able to use pipelining to mask latency. The high-throughput z-switch isotach network (I2) in most cases outperforms the lower throughput isotach switch (I1) in spite of the assumption that I2's cycle time is twice that of I1. In designing conventional networks, however, the simulation indicates that accepting worse delay for better throughput is a poor decision. The high-throughput z-switch C2 outperforms all other networks in raw power, but consistently performs worse than the simpler conventional switch C1 under a workload with synchronization requirements.

The final conclusions we draw from this work are as follows:

1. Local synchrony is implementable on a wide range of parallel architectures without unreasonable hardware costs.
2. Local synchrony is compatible with current trends in hardware fault-tolerance techniques. Local synchrony allows the implementation of simple algorithms for fault-tolerance and computation rollbacks.
3. Local synchrony presents a viable alternative to conventional methods of synchronization for parallel processors. Although the unit cost of a global memory access under local synchrony is greater, local synchrony offers significant advantages over conventional techniques under more realistic workloads.

7.2 Future Work

Local synchrony represents a new approach to concurrency control in parallel architectures. Although timestamping systems have been popular in the database literature, the use of logical time and access timestamps in parallel processing architectures has not been thoroughly explored. The requirements of a database and those of an operating environment are significantly different. The approach to timestamping systems for parallel architectures must, therefore, be different as well.

We believe this field has many opportunities for further study. For example, Williams' delta-cache protocols [Wil93] show that isotach systems can be used to develop an entirely new method of cache coherency techniques. A significant area of future research would be to explore other applications for local synchrony implementations. For example: from the area of synchronization, more complex dependence requirements (e.g., dependence between the accesses of different processes) and broadcast/multicast operations; from fault-tolerance, load-balancing and traffic control.

We look forward to further development of several areas that we have begun to explore in this work. One major emphasis will be on further refinements to implementation techniques for local synchrony. Actual implementation of the local synchrony switch presented in Chapter 3 would allow us to measure true switch performance and develop experimental architectures that implement local synchrony. Further research will aim to refine the implementation and explore alternative methods to improve efficiency and performance.

Another goal of future research would be the low-level design of a high-throughput z-switch for local synchrony. Our assumptions about the cycle time of the high-throughput switches are pessimistic; further research should allow us to develop lower latency versions of the z-switch architecture. Further enhancements to our designs would include providing for multi-input switches (as opposed to the two-input design presented here).

Another implementation goal would be to assess whether other routing techniques would enhance the performance of local synchrony. Virtual Cut-Through [KeK79] and Wormhole Routing [DaS87] are both compatible with local synchrony implementations, and it may be possible to use these approaches to increase the efficiency of the local synchrony implementation, perhaps through timestamp look-ahead at the switch level.

One drawback to local synchrony implementations is that sequencing constraints are not always necessary for every access in a general computation. When concurrency control requirements are low, conventional networks outperform local synchrony. We intend to explore hybrid implementations of local synchrony in which accesses that need not be subject to concurrency control requirements are able to pass more quickly through the network. This approach is similar to that proposed for the IBM RP3 [Pfi85]. For example, a singleton READ access sent by a PE may not have sequential consistency restraints (as a singleton, there are no atomicity restraints). Such an access might have higher priority than normal accesses in the hybrid approach.

Another area where we intend further research is the analytical models presented in Chapter 6. Our main goal is to extend the analytical modeling technique to cover general implementations of local synchrony. The general implementation is a multi-stage architecture, but the numbers of inputs and outputs at each stage can vary significantly within and among architectures. We intend to develop general techniques that allow analytical modeling for general implementations of local synchrony using our implementation technique.

A second goal in this area would be to consider different types of access traffic. The models presented in Chapter 6 consider only uniform traffic distributions, and thus are only able to model the raw power of the networks. We intend to develop models based on more realistic traffic models that include sequencing constraints. Non-uniform distributions of access destinations would also be a goal, so that hot-spot performance could be predicted analytically.

Further goals in this area include covering multi-buffering architectures and combining systems. Each of these enhancements presents significant problems in implementation, in large

part due to the large number of possible buffer states. We intend to explore ways to make such enhancements to our models while keeping the models tractable. We also intend to refine our notion of information flow as presented in Chapter 6 and to explore other modeling techniques.

In the area of simulation, we intend to expand our simulation data by considering more complex sequencing constraints (e.g., pointer following, data dependence between the accesses of different PEs). Another goal is to generate simulation data using access streams based on execution traces of real parallel programs, which would provide evidence of the actual performance of real local synchrony implementations.

In both simulation and analysis, we intend to pursue significant comparative analysis in the future. This type of analysis has three distinct areas:

The first area is the comparison of various enhancements to local synchrony implementations to the basic implementation and to each other. An example of work in this area would be analyzing the effect of combining in local synchrony implementations and collecting data on the performance of enhanced switch designs for local synchrony architectures.

The second area includes comparison of the various approaches to logical timestamp ordering. We intend to compare local synchrony implementations to the Chandy/Misra [ChM79] timestamp ordering system, and to make a thorough comparison of pulse ordering implementations as presented in Chapter 6. We intend further to investigate possible optimistic implementations of logical timestamp ordering systems for concurrency control.

The final area for comparative analysis is between local synchrony and other concurrency control approaches. In Chapter 6 we performed this type of analysis with locking implementations.

This dissertation and [Wil93] have presented strong evidence that logical timestamping systems and local synchrony are a viable alternative to conventional methods of concurrency control. That the study of logical timestamping methods for concurrency control is a field rich with opportunities. This dissertation has taken a significant step in the direction of establishing that fact.

References

[AbP89] S. Abraham and K. Padmanabhan, Performance of the Direct Binary n-Cube Network for Multiprocessors, *IEEE Trans. on Computers 38,7* (July 1989), 1000-1011.

[Abr90] S. Abraham, Issues in the Architecture of Direct Interconnection Schemes for Multiprocessors, in Ph.D. Dissertation University of Illinois, Urbanna, IL, March 1990.

[AdS82] G. B. Adams, III and H. J. Siegel, The Extra Stage Cube: A Fault-Tolerant Interconnection Network for Supersystems, *IEEE Transactions on Computers C-31,5* (May 1982), 443-454.

[Agr83] D. P. Agrawal, Graph Theoretical Analysis and Design of Multistage Interconnection Networks, *IEEE Transactions on Computers*, July 1983, 637-648.

[Awe85] B. Awerbuch, Complexity of Network Synchronization, *J. ACM 32,4* (October 1985), 804-823.

[BaJ88] M. Balakrishnan and R. Jain, On Array Storage for Conflict-Free memory Access for Parallel Processors, in *Proc. 1988 International Conference on Parallel Processing Vol. 1*, IEEE, 1988, 103-107.

[BaB87] V. Balasubramanian and P. Banerjee, A Fault Tolerant Massively Parallel Processing Architecture, *Journal of Parallel and Distributed Computing 4* (1987), 363-383.

[BaD89] P. Banerjee and A. Dugar, The Design, Analysis and Simulation of a FaultTolerant Interconnection Network Supporting the Fetch-and-Add Primitive, *IEEE Trans. on Computers 38,1* (Jan 1989), 30-46.

[BeG80] P. A. Bernstein and N. Goodman, Timestamp Based Algorithms for Concurrency Control in Distributed Database Systems, in *Proc. 6th International Conf. on Very Large Data Bases*, October 1980.

[BeG81] P. A. Bernstein and N. Goodman, Concurrency Control in Distributed Database Systems, *ACM Computer Surveys 13* (1981), 185-222.

[BGCS89] Y. Birk, P. Gibbons, J. Sanz, and D. Soroker, A Simple mechanism for Efficient Barrier Synchronization in MIMD Machines, *IBM Research Report 7078* (67141), October 1989.

[BNR89] R. Bisiani, A. Nowatzyk and M. Ravishankar, Coherent Shared Memory on a Distributed Memory Machine, in *Proc. 1989 International Conference on Parallel Processing*, IEEE, 1989, I-133-141.

[ChM79] K. Chandi and J. Misra, Distributed Simulation: A Case Study in Design and Verification of Distributed Programs, *IEEE Trans. on Software Engineering SE-5, 5* (Septembe 1979), 440-452.

[ChM87] K. Chandy and J. Misra, Conditional Knowledge as a Basis for Distributed Simulation, *CIT CS tech. Rep. 5251:TR:87*, 1987.

[ChS89] M. Chen and K. G. Shin, Fault-Tolerant Routing in Hypercube Multicomputers Using Depth-First Search, in *International Computer Science Institute Tech. Rep.-89-006*, February 1989.

[Dal87] W. Daly, Wire-Efficient VLSI Multiprocessor Communication Networks, Proc. Stanford Conference on Advanced Research in VLSI, MIT Press, March 1987, 391-415.

[Dal90] W. Dally, Virtual-Channel Flow Control, *Proc. 17th IEEE International Symposium on Computer Architecture*, May 1990, 60-68.

[DaS87] W. J. Dally and C. L. Sietz, Deadlock-Free Message Routing in Multiprocessor Interconnection Networks, *IEEE Transactions on Computers C-36,5* (May 1987), 547-553.

[DGKL86] S. Dickey, A. Gottlieb, R. Kenner, and Y. Liu, Designing VLSI Network Nodes to Reduce Memory Traffic in a Shared Memory parallel Computer, *NYU Ultracomputer Note #125*, August 1986.

[DiJ81a] D. M. Dias and R. Jump, Analysis and Simulation of Buffered Delta Networks, *IEEE TOC C30,4* (April 1981), 273-282.

[DiJ81b] D. M. Dias and J. R. Jump, Packet Switching Interconnection Networks for Modular Systems, *IEEE Computer 14,12* (December 1981), 43-53.

[DiK92] S. Dickey and R. Kenner, A Combining Switch for the NYU Ultracomputer, June 1992.

[DSB88] M. Dubois, C. Scheurich and F. Briggs, Synchronization, Coherence, and Event Ordering in Multiprocessors, *IEEE Computer 21,2* (February 1988).

[EGL76] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, The Notions of Consistancy and Predicate Locks in a Database System, *Comm. ACM 19,11* (November 1976), 624-633.

[Fly66] M. J. Flynn, Very High Speed Computers, *Proceedings of the IEEE 54* (December 1966), 1901-1909.

[GGH91] K. Gharachlorloo, A. Gupta, and J. Hennessy, Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors, *Proc. ACM APLOS-IV*, April 1991, 245-257.

[Gel81] D. Gelertner, A DAG-based algorithm for Prevention of Store-andForward Deadlock in Packet Networks, *IEEE Transactions on Computers C-30* (October 1981), 709-715.

[Gib89] P. B. Gibbons, The Asynchronous PRAM: A Semi-Synchronous Model for Shared Memory MIMD Machines, in *International Computer Science Institute Tech. Rep.-89-062*, December 1989.

[GLR81] A. Gottlieb, B. D. Lubachevsky and L. Rudolph, Coordinating Large Numbers of Processors, in *Proc. 1981 International Conference on Parallel Processing*, 1981.

[GGK83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, The NYU Ultracomputer Designing a MIMD Shared Memory Parallel Computer, *IEEE Transactions on Computers 32,2* (February 1983), 175-189.

[GLR83] A. Gottlieb, B. Lubachevsky and L. Rudolph, Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processes, *ACM Trans. Prog. Lang. and Systems 5,2* (April 1983), 164-189.

[Gun81] K. D. Gunther, Prevention of Deadlocks in Packet-Switched Data Transport Systems, *IEEE Transactions on Communications COM-29,4* (April 1981), 512-524.

[Jef83] D. Jefferson, Virtual Time, in *Proc. 1983 International Conference on Parallel Processing*, 1983, 384-394.

[Jef85] D. R. Jefferson, Virtual Time, *ACM Trans. Prog. Lang. and Systems 7,3* (July 1985), 404-426.

[Jen83] Y. Jenq, Performance Analysis of a Packet Switch Based on Single-Buffered Banyan Network, *IEEE Journal on Selected Areas in Communications SAC-1,6* (December 1983), 10141020.

[KHM87] M. Karol, M. Hluchyj, and S. Morgan, Input Versus Output Queueing on a Space-Division Packet Switch, *IEEE Trans. on Communications Com-35, 12* (December 1987), 1347-1356.

[KeK79] P. Kermani and L. Kleinrock, Virtual Cut-Through: A New Computer Communication Switching Technique, *Computer Networks 3* (1979), 267-286.

[KrS83] C. P. Kruskal and M. Snir, The Performance of Multistage Interconnection Networks for Multiprocessors, *IEEE Transactions on Computers C-32,12* (December 1983), 1091-1098.

[KRS88] 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.

[Kum86] D. Kumar, Simulating Feedforward Systems Using a Newtork of Processors, *Proceedings of the Annual Simulation Symposium*, 1986, 127-144.

[KuR85] V. P. Kumar and S. M. Reddy, Design and Analysis of Fault-Tolerant Multistage Interconnection Networks With Low Link Complexity, 1985.

[KuJ86] M. Kumar and J. R. Jump, Performance of Unbuffered Shuffle-Exchange Networks, *IEEE TOC C-35,6* (June 1986), 573-578.

[Lam78] L. L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, *Comm. ACM 21,7* (July 1978), 558-565.

[Lam79] L. Lamport, How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs, *IEEE Transactions on Computers C-28,9* (September 1979), 690-691.

[Lee86] G. Lee, Some Issues in General-Purpose Shared Memory Multiprocessing: Parallelism Exploitation and Memory Access Combining, Ph.D. Thesis, CS Dept. Univ. of Illinois at Urbana-Champaign, 1986.

[LiK91] T. Lin and L. Kleinrock, Performance Analysis of Finite-Buffered Multistage Interconnection Networks with a General Traffic Pattern, *Proc. 1991 ACM SIGMETRICS Conf.*, 68-78.

[Mer91] A. Merchant, A Markov Chain Approximation for the Analysis of Banyan Networks, *Proc. 1991 ACM SIGMETRICS Conference*, 60-66.

[MeS80] P. M. Merlin and P. J. Schweitzer, Deadlock Avoidance in Store-and-Forward Networks 1: Store-and-Forward Deadlock... 2: Other Deadlock Types, *IEEE Transactions on Communications COM-28,3* (1980), 345360.

[Mil79] M. Milenkovic, Update Synchronization in Multiaccess Systems, University of Massachusetts, Amherst, May, 1979.

[OwL82] S. Owicki and L. Lamport, Proving Liveness Properties of Concurrent Programs, *ACM Trans. Prog. Lang. and Systems 4,3* (July 1982), 455-495.

[Pad90] K. Padmanabhan, Cube Structures for Multiprocessors, *Comm. ACM 33,1* (January 1990), 43-52.

[Pat81] J. H. Patel, Performance of Processor-Memory Interconnections for Multiprocessors, *IEEE Transactions on Computers C-30,10* (October 1981), 771-780.

[Pfi85] G. F. Pfister, The IBM Research Parallel Processor (RP3): Introduction and Architecture, *Proc. Int. Conf. on Parallel Processing*, 1985, 764-771.

[PrV81] F.P. Preparata and J. Vuillemin, The Cube-Connected Cycles: A Versatile Network for Parallel Computation, *Comm. ACM 24,5* (May 1981), 300-309.

[RAK89] U. Ramachandran, M. Ahamad and M. Khalidi, Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer, in *Proc. 1989 International Conference on Parallel Processing*, IEEE, 1989, II-160-169.

[Ran75] B. Randell, System Structure for Software Fault Tolerance, *IEEE Trans. on Software Engineering SE-1* (June 1975), 220-232.

[Ran87] A. G. Ranade, How To Emulate Shared Memory, in *Annual Symp. on Foundations of Computer Science '87*, IEEE, 1987, 185-194.

[RBJ88] A. G. Ranade, S. N. Bhatt and S. L. Johnsson, The Fluent Abstract Machine, *Yale University Department of Computer Science Tech. Rep.-573*, January 1988.

[Ree83] D. P. Reed, Implementing Atomic Actions on Decentralized Data, *ACM Transactions on Computer Systems 1,1* (February 1983), 3-23.

[ReT86] R. Rettberg and R. Thomas, Contention is No Obstacle to Shared-Memory Multiprocessing, *Comm. ACM 29,12* (December 1986), 1202-1212.

[ReW91] P. F. Reynolds, Jr. and R. R. Wagner, Jr., A Local Synchrony Implementation: Banyan Networks, *UVA CS Tech. Rep.-91-38*, December 1991.

[RWW89] P. F. Reynolds, Jr., C. Williams and R. R. Wagner, Jr., Parallel Operations, *UVA CS Tech. Rep.-89-16*, December 1989.

[RWW92] P. F. Reynolds, Jr., C. Williams and R. R. Wagner, Jr., Empirical Analysis of Isotach Networks, *UVA CS Tech. Rep.-92-19*, June1992.

[RSL78] D. Rosenkrantz, R. Stearns and P. Lewis, System-Level Concurrency Control for Distributed Data Bases, *ACM Trans. Database Systems 3,2* (1978), 178-198.

[Rud81] L. Rudolph, Software Structures for Ultraparallel Computing, Ph.D. Dissertaion, NYU, 1981.

[SaS88] Y. Saad and M. H. Shultz, Topological Properties of Hypercubes, *IEEE Transactions on Computers C-37,7* (July 1988), 867-872.

[Sch83] F. B. Schneider, Fail-Stop Processors, in *Digest of Papers Spring Compcon '83*, IEEE Computer Society, San Francisco, CA, March 1983.

[Sei85] C. L. Seitz, The Cosmic Cube, *Comm. ACM 28,1* (January 1985), 22-33.

[ShS88] D. Shasha and M. Snir, Efficient and Correct Execution of Parallel Programs that Share Memory, *ACM Trans. Prog. Lang. and Systems 10,2* (April 1988), 282-312.

[TAF88] Y. Tamir and G. Frazier, High-Performance Multi-Queue Buffers for VLSI Communication Switches, *Proc. 15th IEEE Symposium on Computer Architecture*, May 1988, 343-354.

[TKT92] Z. Tong, R. Kain, and W. Tsai, Rollback Recovery in Distributed Systems Using Loosely Synchronized Clocks, *IEEE Trans on Parallel and Distributed Systems 3, 2* (March 1992), 246-251.

[TuR88] L. W. Tucker and G. G. Robertson, Architecture and Applications of the Connection Machine, *IEEE Computer 21,8* (August 1988), 2638.

[TYZ85] N. Tzeng, P. Yew and C. Zhu, A Fault-Tolerant Scheme for Multistage Interconnection Networks, 1985.

[TYZ86] N. Tzeng, P. Yew and C. Zhu, Fault-Diagnosis in a Multiple-Path Interconnection Network, *IEEE Transactions on Computers*, 1986, 98-103.

[VaR86] A. Varma and C. S. Raghavendra, Fault-Tolerant Routing in Multistage Interconnection Networks, *IEEE Transactions on Computers*, 1986, 104-109.

[Wag87] R. R. Wagner, Jr., Parallel Operations in Shared Memory, Master's Thesis University of Virginia, 1987.

[WiE90] D. Willick and D. Eager, An Analytical Model of Multistage Interconnection Networks, *Proc. 1990 ACM SIGMETRICS Conf.*, 192-199.

[Wil90] C. Williams, Concurrency Control in Asynchronous Parallel Computations: A Research Proposal, University of Virginia, June 1990.

[Wil93] C. Williams, Concurrency Control in Asynchronous Parallel Computations, PhD Dissertation, University of Virginia, January 1993.

[WiR89] C. Williams and P. F. Reynolds, Jr., On Variables as Access Sequences in Parallel Asynchronous Computations, *UVA CS Tech. Rep.-89-17*, December, 1989.

[WiR91] C. Williams and P. F. Reynolds, Jr., Combining Atomic Actions in a Recombining Network, *UVA CS Tech. Rep.-91-33*, November, 1991.

[Won86] M. C. Wong, A Combining Omega Network: Performance vs. Implementation, *IBM Research Report RC 11977 (#53952)*, 6/24/86.

[Wu80] C. L. Wu and T. Y. Feng, On a Class of Multistage Interconnection Networks, *IEEE Transactions on Computers*, August 1980, 694-702.

[WuL92] C. Wu and M. Lee, Performance Analysis of Multistage Interconnection Network Configurations and Operations, *IEEE Trans. on Computers 41, 1* (January 1992), 18-27.

[YLL87] H. Yoon, K. Y. Lee and M. T. Liu, Performance Analysis and Comparison of Packet Switching Interconnection Networks, in *Proc. 1987 International Conference on Parallel Processing*, August 1987, 542-545.

[YLL90] H. Yoon, K. Lee and M. Liu, Performance Analysis of Multibuffered Packet-Switching Networks in Multiprocessor Systems, *IEEE TOC 39,3* (March 1990), 319-327.

[YoL89] H. Yoon and K. Lee, B-Banyan and B-Delta Networks for Multiprocessor Systems, *Journal of Parallel and Distributed Computing 7* (1989), 570-582.

[YYF85] W. C. Yen, D. W. L. Yen and K. Fu, Data Coherence Problem in a Multicache System, *IEEE Transactions on Computers C-34,1* (January 1985), 56-65.