Optical Waveguides in General Purpose Parallel Computers

A Thesis
Presented to
The Faculty of the Division of Graduate Studies

By

Martin H. Davis, Jr.

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy
in Computer Science

Georgia Institute of Technology
November 1992

Optical Waveguides in General Purpose Parallel Computers

Approved:

Umakishore Ramachandran, Chairman

Mostafa H. Ammar

Richard M. Fujimoto

Carl M. Verber

Sudhakar Yalamanchili

Date Approved by Chairman

Acknowledgements

The work needed to earn a Ph.D. degree seems, at times, quite daunting to the one who has undertaken this task. And, most certainly, the colleagues, friends, and family of the one earning the degree must face the quite daunting task of working with and living with that person! Therefore, the task of my having earned the Ph.D. cannot be complete until I acknowledge the help, advice, encouragement, and trust that those people have placed in me.

Most integral to the Ph.D. process is one's advisor: thank you Kishore for seeing me through the arduous task (and thanks to Vasanthi for calling Kishore **every** time I met him in his office!). One's reading committee ensures that the reported work constitutes a worthwhile endeavor: thank you Dr. Ammar, Dr. Fujimoto, Dr. Verber, and Dr. Yalamanchili for your time and comments. I also thank Dr. Venkateswaran and Dr. Ahamad for their words of encouragement and advice from time to time.

One's peers are also necessary to the Ph.D. process; without their constant wit, sarcasm, encouragement, and, most importantly, concomitant trials and tribulations the process would be difficult. Ajay Mohindra always gave me much feedback on every aspect of my work; he was also a good personal friend. Gautam Shah helped in understanding certain details. Vibby Gottemukkala did . . . , well, he just usually goaded and provoked me constantly. Sreenivas Gukal was a quiet office mate. Sathis Menon always had a good joke or story, and he was quite willing to discuss the 1992 Presidential political campaign at any time! Dr. Margaret Francel, during her summer sojurns, provided much needed relief from the heat and school topics—her daily summaries of talk show topics and discussions gave an unusual perspective to life!

Without one's family's constant, enduring, and patient love and support, the Ph.D. process would be lonely and difficult. I cannot thank my parents, Martha B. Davis and Martin H. Davis, Sr., enough for the love and support they have shown me in so many ways. I hope that Martin III will some day realize that, although at times his demands for my time seemed to make my life more difficult, those same demands gave me perspective on what really is important in life. And to my wife Carol: I do not think you knew what you were getting yourself in for, I know that these past few years have not always been easy, I know some of your goals have been delayed or irrevocably altered, but I appreciate, more than I can ever truly express satisfactorily, your love, and I hope I can help you achieve your dreams.

Quotations

I list here several quotations which I have found useful while working on my dissertation.

1.

Spock: I seem to have a body which stretches into infinity.

Scotty: Body? Why, you have none!

Spock: Then, what am I?

Bones: You are a disembodied brain.

Spock: Fascinating; it could explain much, Doctor.

A short time later…

Jim: Spock, you're in a black box tied in with light rays into a complex control panel.

Spock: Fascinating.

from *Star Trek* episode entitled "Spock's Brain"

2.

Once you've been there, it's easy.

–Fred W. Lennon, IBM Manager

3.

There are only three types of research curves: those that go up, down, or flat.

–Sudha Yalamanchili, Georgia Tech EE professor

4.

You think you have it made when you teach your child to wipe his bottom, but, then, you have to teach him to become a civilized human being.

–Kenneth B. McKenzie III, Presbyterian Minister

5.

So what's the big deal Dad? The light is just going round and round and round...

–Martin H. Davis III

6.

If you don't understand something, explain it to someone else. If you still don't understand it, teach a class on it. And if you still *don't understand it, write a book on it!*

–R. S. Jin, FIT Physics and Space Science professor

7.

We are scientists!

–Kishore Ramachandran, Georgia Tech CoC professor

Contents

List of Tables

List of Figures

Summary

This thesis examines how optics can be used in general purpose parallel computing systems. Two basic assumptions are made. First, optical waveguide communications technology will continue to mature and become more and more prevalent in smaller and smaller scale environments. Second, electronic computational capabilities will continue to increase for at least the next decade. Thus, this research explores ways in which optical waveguide communications can be combined with traditional electronic computing elements to support general purpose parallel computing. The specific question asked is, "How can the properties of optical waveguides give rise to architectural features useful for general purpose parallel computing?" The answers to this question are developed in the context of a distributed shared memory computing design called OBee. This work defines the OBee design, a specific implementation, based on optical waveguides, of a previously developed, more abstract architecture named Beehive. The basic building block of OBee's physical optical architecture is an Optical Broadcast Ring (OBR). The thesis defines how one or more waveguides (or wavelengths) are arranged in varying topologies; it also defines several different access protocols. Together, a particular combination of topology and access protocol define a given OBR's properties. The OBee design employs a particular OBR to define a specific implementation of Beehive's reader initiated cache coherency protocol. The OBee design uses two different OBRs to define two distinct implementations of Beehive's sole synchronization primitive, locks. As improvements to Beehive, OBee adds two more synchronization primitives, barriers and Fetch-and-OP. The OBee design uses two different OBRs to define two distinct implementations of barriers; similarly, it uses two different OBRs to define two distinct implementations of Fetch-and-OP. Analytical evaluations of the performance of the raw architectural primitives are presented which show the primitives can be executed in reasonable amounts of time. The thesis concludes that optical waveguides can provide more than just high speed data transmission since the OBee design demonstrates that command primitives can be directly built from OBRs' properties. Several questions for future research pertinent specifically to OBee and generally to optics in computing are enumerated.

Chapter 1

Introduction

Over the years there has been much research into the role that optics can play in computing [5, 7, 23]. For certain specialized *analog* computations (e.g., signal processing or matrix-vector multiplication), optics is clearly well suited. However, the role of optics in *digital* or *general purpose* computing has not been so clear. Initial attempts have been made at constructing elementary computers from optical computational elements [14, 35, 36, 44, 45, 66], but these efforts cannot compete with the power of current electronic computing elements such as the DEC Alpha chip [21].

In contrast to the primitive state of optical computational elements, the state of optical communications is much more mature. The use of optical communications has become quite commonplace and practical in large-scale transmissions systems (e.g., long-haul and intracity voice and data transmission) [6, 47]. Researchers also believe that because of the physical properties of electrical buses and interconnects [20, 24] that limit their bandwidth and interconnect distance, optical interconnects will eventually be necessary for board to board, module to module, chip to chip, and even intrachip communications in computer systems [11].

There are several observations regarding communication and computation which have led to this research:

- the present-day, commonplace large-scale communications systems rely upon the well developed technology of optical waveguides;
- the bandwidth potentially available on a single waveguide, because of the high frequency of light, is several tens of Tbits/sec;
- as optical waveguide communications technology continues to mature, it is being incorporated into small-scale transmission environments such as university campuses and individual buildings;
- electronic computing elements such as CPUs are quite advanced and continue to improve in their capabilities.

Therefore, because of these observations, this research has two motivating assumptions. First, optical waveguide communications technology will irreversibly continue to replace electrical communications in smaller-scale environments for reasons such as the high bandwidth and favorable engineering properties (e.g., low crosstalk and immunity to electromagnetic interference). Second, electronic computing capabilities will continue to increase immensely. For example, Intel Corp. has a "Project 2000" in which it expects to place 100 million transistors on a single one square inch chip by the year 2000.

Previous researchers' work, as will be elaborated upon in Chapter 3, for the most part, has been concerned with how high speed data rates (greater than Gbit/sec) can be engineered. However, such research has ignored other potentially useful properties of optics. Therefore, the research in this thesis takes a different direction. On the basis of the two motivating assumptions given above, this research examines how properties of optical waveguides (other than just high speed data rates) can be combined with the mature computational capability of electronics in the design of a general purpose parallel computer architecture. Thus, this researches question can be summed up as:

> *If optical waveguides are used in interconnecting electronic computational elements, how can the properties of the optical waveguides be translated into architectural features for general purpose parallel computers?*

In order to answer this question, we have kept three goals in mind:

1. the purpose or application of the machine;
2. an appropriate logical architecture;
3. the manner in which the hardware can support the first two goals.

The answer to our first goal comes from our earlier stated interest in general purpose parallel computing, i.e., in the scope of this research, we have not been interested in designing a computer architecture for specific applications such as image understanding, neural networks, weather and climate modeling, or quantum chromodynamics calculations. Rather, our goal has been an architecture to support computing for any application which can be parallelized.

The remainder of the thesis deals with the second and third design goals, and is organized as follows. Chapter 2 briefly reviews various logical parallel architectures and describes the specific logical parallel architecture for which our optical designs are aimed. The properties of the physical optical communications architecture that we have developed are described in Chapter 3. In Chapters 4 and 5 we show how our optical hardware provides support for cache coherency and various synchronization primitives which are useful in parallel programming. Chapter 6 presents some evaluations of the proposed architectural features. Chapter 7 gives the conclusions of this research and directions for future research.

By satisfying the second and third design goals, this research has generated several answers to the earlier stated question of how to use optical waveguides in constructing architectural features:

- the bandwidth of the optical waveguide is divided into multiple, logical channels;
- channels are pre-assigned their function in the architecture;
- low channel utilization is not frowned upon;
- the physical optical communications architecture can provide direct hardware support for useful command primitives.

These answers should be viewed by the reader as the themes which tie together the thesis. As the reader progresses through the thesis, he will find these themes developed and explained.

Chapter 2

Architectural Description

As stated in Chapter 1, the computer architect must keep in mind an appropriate logical architecture. In this chapter we describe a logical architecture called *Distributed Shared Memory* (DSM), a specific DSM architecture called *Beehive*, and our proposed optical DSM architecture called *OBee* (for Optical Beehive).

2.1 Distributed Shared Memory (DSM)

Since, as explained in Chapter 1, our stated application is for general purpose parallel computing, the most appropriate logical architecture is the *Multiple Instruction, Multiple Data* (MIMD) organization. The two classical types of MIMD architectures are *distributed memory* (message passing) and *shared memory* structures.

In the distributed memory architecture the nodes work relatively independently (loosely coupled). Each node has its own memory which no other node can access, as illustrated in Fig. 1. In order for a node to coordinate and communicate its computation with other nodes, explicit messages are sent via an interconnection network. Proponents of this architecture believe that it can accommodate a large number of processors. This contention is based on the fact that in distributed

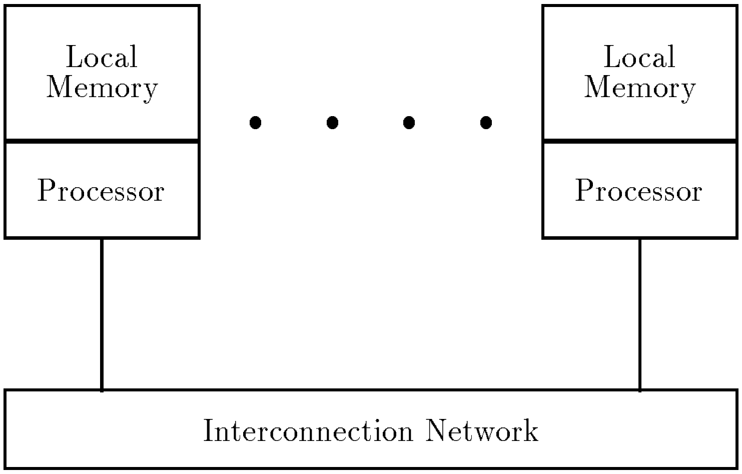

Figure 1: Block diagram of the Distributed Memory architecture.

memory architectures, as processors are added, memory *and* the communications capability of the interconnection network are increased proportionally. The Intel iPSC/2 and Intel Touchstone Delta machines are well known examples of the distributed memory architecture.

In contrast, the nodes in a shared memory architecture work together much more closely (tightly coupled). The processors all share a common, global memory which is not associated with any particular one of the processor nodes, as illustrated in Fig. 2. Processor nodes may also have their own private, local memory which other nodes cannot access. Coordination and communication among processors is easily accomplished by their sharing the appropriate data structures in the global memory. Strictly speaking, in this definition, processor nodes all have uniform memory access, i.e., any processor's access to any part of the global memory is the same as any other's access. The shared memory architecture is favored by

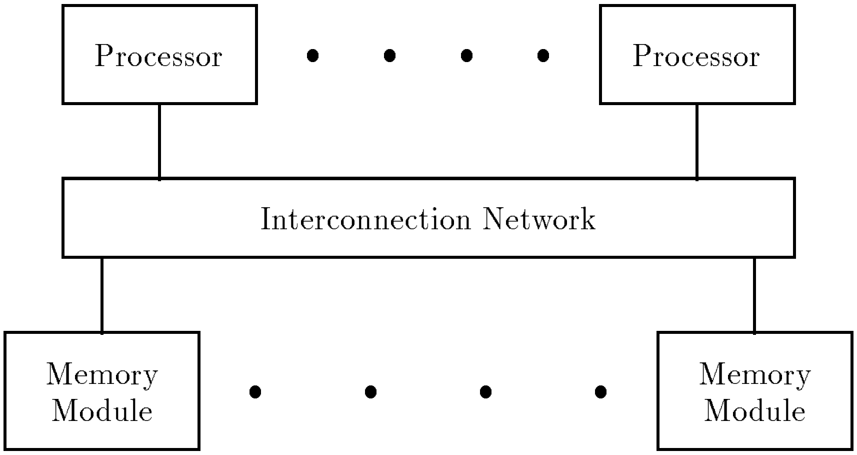

Figure 2: Block diagram of the Shared Memory architecture.

those who maintain that it is much easier to program than a distributed memory architecture. The shared memory architecture proponents base this contention on the fact that the global memory makes the parallel machine look very similar to a standard uniprocessor computing model: programmers do not worry about decomposing the computation into parts which must send messages from one part to another. A well known example of the shared memory architecture is the Sequent Symmetry machine.

Thus, the criticism of distributed memory is that it is hard to program efficiently, and the criticism of shared memory is that it does not support a large number of processors well. Recent attempts to remove both of these criticisms in one architecture have resulted in a parallel architecture known as *Distributed Shared Memory* (DSM). In the DSM architecture the global memory is partitioned among all the computing nodes, i.e., the global memory is physically distributed

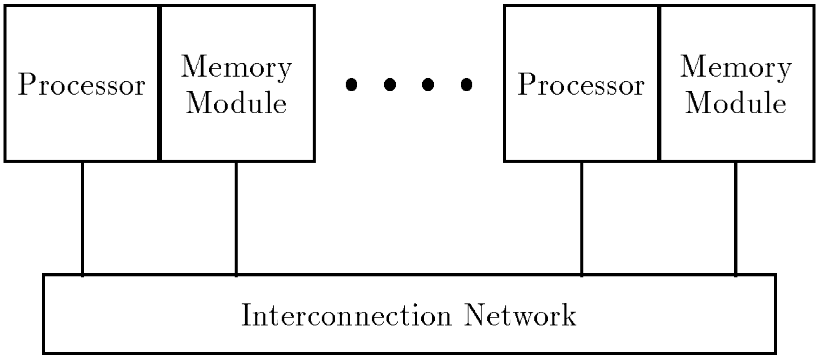

Figure 3: Block diagram of the Distributed Shared Memory (DSM) architecture.

among the nodes, as illustrated in Fig. 3. Each computing node has access to *both* the part of global memory physically associated with it *and* to all the other parts physically associated with the other nodes. The part of global memory physically associated with a node is called that node's *Nearest Shared Memory* (NeSM), and the parts physically associated with the other nodes are called the *Remote Shared Memory* (ReSM).

Even though the DSM architecture provides a *logical* global memory, a node's access to memory is non-uniform. This non-uniform memory access occurs because a node has preferential (in a restricted sense) access to its NeSM. The node's access to its NeSM is preferential in the sense that its requests of that memory module can be made via a dedicated connection, whereas other nodes' requests to that memory module (which they see as ReSM) must be made via an interconnection network. A node's access to its NeSM is *not* preferential in the sense of having higher priority than remote nodes, just preferential in terms of physical access. A commercial example of the DSM architecture is the BBN

Butterfly. The Dash project at Stanford [57] is an academic DSM machine.

2.2 Beehive

In Section 2.1 only a very high logical level of abstraction for the DSM architecture was described. The next level down of logical abstraction needs to define some more specific characteristics. The *Beehive* definition (proposed by Lee and Ramachandran [55]) specifies some of these details.

Beehive's overall logical architecture follows the organization shown in Fig. 3 for a DSM architecture. The Beehive definition does *not* specify what the interconnection network should be—it may be any arbitrary network (e.g., a bus, a multistage network, a hypercube, a mesh, etc.). There are some other details which the Beehive definition does define. Some of these details are strictly hardware (as illustrated in Fig. 4), and some are definitions of how the architecture should behave.

The "CPU" block in Fig. 4 represents the computational capabilities of a Beehive node, but the Beehive definition does not specify what kind of computational capabilities exist or how they are implemented.

From the discussion in Section 2.1 the reader should realize that a cost of providing a global memory when the memory modules are physically distributed among the nodes is the possible long latency of a ReSM access. The hardware solution for alleviating this problem is to provide each node with a private cache (as illustrated by the "Cache Controller" and "Cache" blocks in Fig. 4). The cache stores local copies of accesses to both its NeSM and ReSM. Thus, the

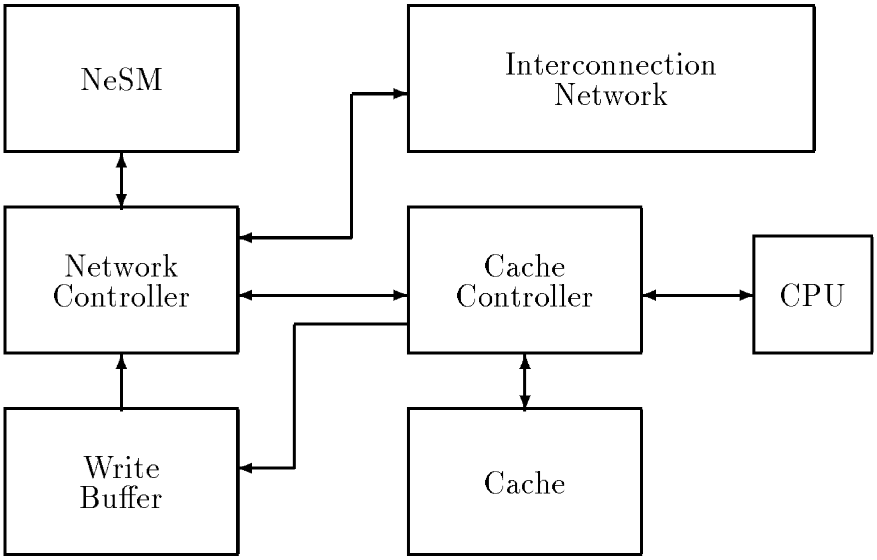

Figure 4: Block diagram of the Beehive node architecture.

CPU issues a normal memory reference and does not know where that reference resides. The Cache Controller intercepts the memory reference; if the reference can be satisfied in the Cache, it is returned immediately to the CPU. If the memory reference cannot be satisfied by the Cache, the Cache Controller asks the "Network Controller" to act upon the memory reference. The Network Controller's responsibility is to determine whether the memory reference can be satisfied by the node's NeSM or whether the request must be put on to the "Interconnection Network" and sent to the appropriate ReSM.

Although the local Cache can reduce the long latency of ReSM accesses, the cost for this solution is the introduction of the *cache coherency* (or *memory consistency*) problem [75]. Since multiple copies of the same global memory location may exist simultaneously in an arbitrary number of the local, private caches, some mechanism must be used to enforce a consistent view of global memory for the programmer. Without such a consistent view the programmer cannot write a parallel program that will execute correctly (i.e., give the expected results).

From the programmer's standpoint the simplest memory consistency model is *sequential consistency* as proposed by Lamport [53]. Sequential consistency states that the multiprocessor execution of a parallel program should have the same effect as a sequential execution of any arbitrary, allowed interleaving of the memory operations of all the parallel tasks of the program. An allowed interleaving is one that preserves the program order of the memory operations of each independent parallel task. As shown in Fig. 5, from the viewpoint of the task's execution, any memory operation (read/write) must become visible everywhere else in the system before the next memory operation may be issued. Given the sequential consistency model, the programmer can use ordinary read and write operations to achieve synchronization among the parallel tasks. Thus, we say that sequential consistency imposes a *strong* ordering upon *all* memory operations regardless of their type.

Consider that memory operations may be divided into two types: shared data accesses and synchronization accesses. This division may be used to define a *weak memory consistency* model. In a weak memory model shared data accesses are not necessarily consistent until a synchronization point is reached. Therefore, multiple copies (located in different local caches) of shared data can be temporarily inconsistent with each other. A synchronization point is used to bring the copies back into a consistent state. We say that the synchronization points are strongly ordered. The reason for using a weak memory model is to improve the performance

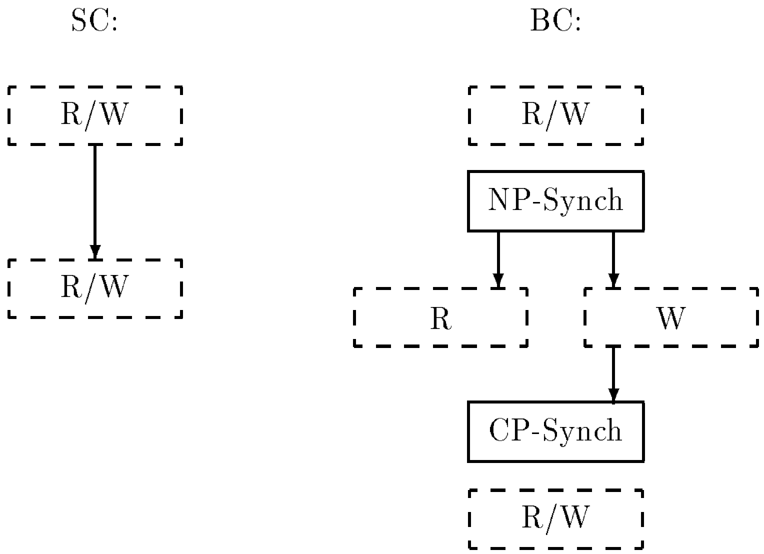

Figure 5: The sequential consistency (SC) and buffered consistency (BC) memory model constraints. Arrows indicate dependencies.

of the program since the physical architecture can implement features to take advantage of the weak memory model.

Various weak memory models have been proposed: weak consistency [22], release consistency [33], and *buffered consistency* (BC) [55]. Beehive first defined the BC model. The BC model, as shown in Fig. 5, has considerably different constraints on the dependencies of memory accesses as compared to sequential consistency. First, memory operations are divided into two classes, reads/writes and synchronization points. Reads/writes are themselves divided into *private* and *shared* accesses. Private reads/writes are assumed to be to data owned and accessed by only one task. Shared reads/writes are assumed to be to data shared by multiple tasks. Synchronization points are themselves divided into *NP-Synch* and *CP-Synch* points. A NP-Synch (Non-consistency Preserving) point defines when a *Synchronization Epoch* (SE) begins. The matching CP-Synch point defines when the SE ends. The salient feature of a SE is that while inside a SE, only the task executing the SE has a consistent view of the shared data; no guarantee is made to other tasks about the state of the shared data. At the end of the SE the CP-Synch point is executed. Execution of the CP-Synch point guarantees that all other tasks now have a consistent view of the shared data contained within that SE. When the CP-Synch point has completed execution, then (and only then) may the task continue its execution.

The second constraint of the BC memory model is that no guarantee is made about the consistency of reads/writes outside a SE. The programmer normally would be accessing private data that do not require any consistency (since our assumption is that private data are "owned" by only one task and can only be accessed by that task) in such regions. However, the Beehive definition does not disallow the programmer from accessing shared data outside a SE; the programmer must keep in mind that a read to such shared data might not return the latest (consistent) value and that a write might result in incorrect program operation.

The BC model's third constraint is that when a CP-Synch point is reached, only writes to shared data must be completed before the CP-Synch operation is deemed completed. The CP-Synch point need not wait on reads to complete; the programmer may improve program execution efficiency by "hoisting" some reads into the SE before issuing the CP-Synch instruction. Keeping in mind the previous discussion of correctness, the reader sees that such reads should be to private data or to shared data for which a possible lack of consistency is not a problem.

The “Write Buffer” block in Fig. 4 is the additional hardware needed to implement the BC memory model. Private reads/writes and shared reads are handled by the Cache Controller as previously described. Shared writes, however, are placed into the Write Buffer by the Cache Controller and are eventually propagated by the Network Controller over the Interconnection Network. This propagation of the shared writes eventually brings the caches into a consistent state; a more detailed discussion of the particular cache coherency protocol is given in Chapter 4. Because the shared writes are placed into the Write Buffer, as far as the CPU is concerned, the shared writes have completed, and the CPU may continue its execution stream. When the CPU reaches a CP-Synch point in its execution stream and issues the CP-Synch instruction, it stall until all the pending shared writes in the Write Buffer have been propagated to all other nodes and until all other appropriate copies of that data have been updated. Once these actions have taken place, the CP-Synch point is deemed to be completed.

The reader can now see how the program’s performance is improved by allowing the hardware to exploit the weak memory model. When a task issues a shared write, the task no longer necessarily has to wait until the shared write has been communicated everywhere. The latency of the shared write is being overlapped with the task’s computation in the SE. In addition, if the structure of the program permits, private reads which logically would occur after the CP-Synch point of the current SE may be placed (“hoisted”) inside the SE. That is, shared write latency can be overlapped with subsequent private reads. Therefore, in the ideal SE, when the CP-Synch instruction is issued, the Write Buffer has already been flushed (because the shared writes have been propagating to the other nodes while the task performed other instructions), and the CP-Synch instruction can complete without delay.

2.3 OBee

OBee (Optical Beehive) is our proposed hybrid electronic/optical DSM architecture which builds upon the Beehive definition given in Section 2.2. The OBee definition specifies some more details which Beehive leaves undefined.

The first additional detail that OBee specifies is that its interconnection network is not the arbitrary interconnection network of Beehive. The interconnection network consists of multiple *Optical Broadcast Ring*s (OBRs) which are based on optical waveguides; OBRs will be described in detail in Chapter 3. For now, suffice it to say that by having a specific interconnection network, OBee can effectively support additional features which Beehive cannot.

A second detail in OBee that differs from Beehive is the implementation of the cache coherency protocol. Although the *reader initiated* protocol has the same logical behavior in both Beehive and OBee, the implementation is different because of the use of OBRs in OBee. The details of reader initiated cache coherency and the OBee implementation are given in Chapter 4.

Finally, the third detail that differentiates OBee from Beehive is the additional hardware support for *synchronization primitives*. Beehive defines how *locks* are implemented, but as illustrated in Fig. 6, the OBee architecture utilizes specialized hardware for the *lock*, *barrier*, and *F&OP* synchronization constructs. This specialized hardware is both electronic and optical, depending upon the particular

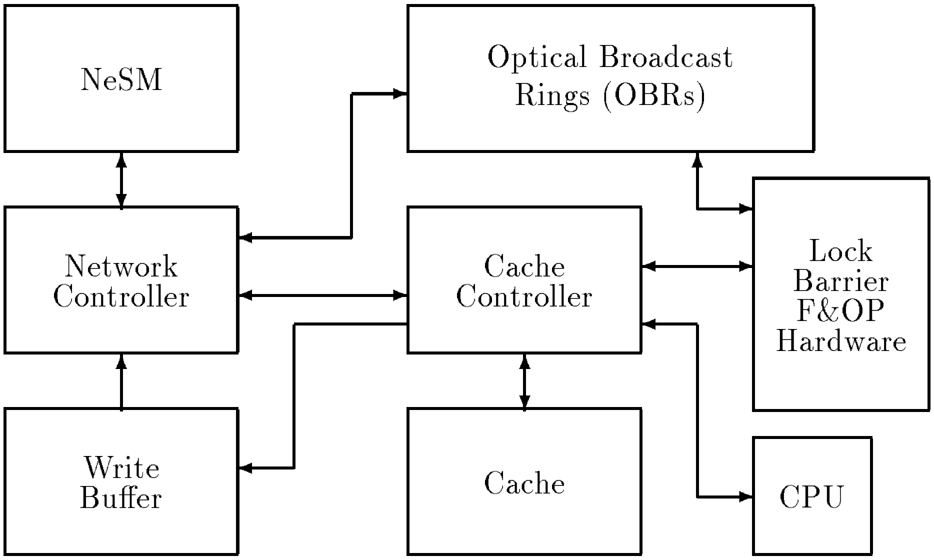

Figure 6: Block diagram of the OBee node architecture.

implementation. In addition, OBee utilizes the properties of the OBRs to implement these three synchronization primitives. The details of these synchronization primitives' implementation are discussed in Chapter 5.

Chapter 3

The Optical Broadcast Ring (OBR) Architecture

As described in Section 2.3, the OBee architecture uses a specific interconnection network rather than the arbitrary interconnection network of Beehive. The OBee interconnection network is based on an optical waveguide architecture that we call a *Optical Broadcast Ring* (OBR). This chapter presents a very brief description of optical waveguides (with references for more detail), reviews the pertinent previous work in optical waveguide architectures, and then discusses our proposed OBR architecture.

3.1 Description of an Optical Waveguide

Light can be transmitted in free space or via a *dielectric waveguide* [47, 65]. An optical fiber is an example of a light conducting waveguide and is normally cylindrical in form (see Fig. 7). When the waveguide is constructed properly and the light enters at an allowed angle, the waveguide confines the optical electromagnetic energy within the core and guides the light in a direction parallel to its axis. Since the index of refraction of the cladding is less than the index of refraction of the core ($n_{clad} < n_{core}$), then one can understand the light's being confined in the core

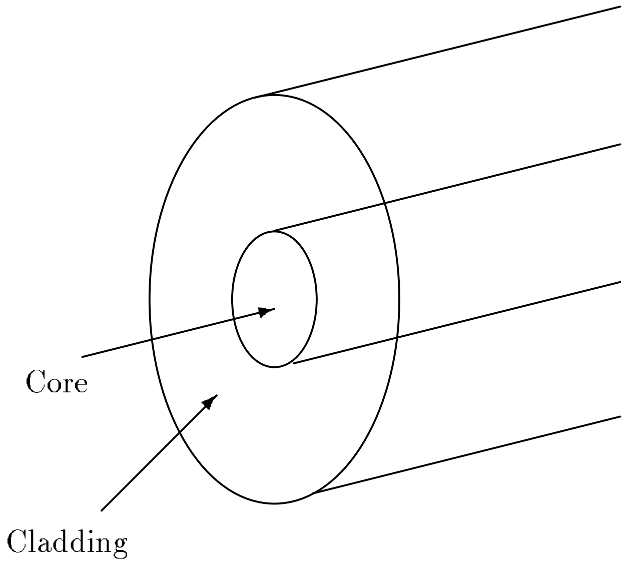

Figure 7: The structure of an optical fiber.

by applying Snell's law of total internal reflection at the core and cladding interface. However, to understand fully the details of how light propagates through the waveguide, Maxwell's equations, for the particular geometry of the waveguide, must be solved. The interested reader should refer to texts such as by Keiser [47] or Midwinter and Guo [65] for more information about the physical nature of optical waveguides. For our purposes in the rest of this thesis, it is sufficient to know that light of many different wavelengths (each wavelength being a logical channel in our usage) can be propagated via optical waveguides.

3.2 Prior Optical Waveguide Architectures

A number of optical waveguide architectures have been previously proposed. Some of these architectures are "paper designs;" others have progressed to the prototype stage. We classify these architectures into two types: those designed as an optical substitution of a computer's "system backplane" and those designed for Local Area Network (LAN) communications.

3.2.1 System Backplane Architectures

The Heidelberg Polyp [60, 61, 81, 82] is a tightly coupled multiprocessor shared memory machine in which the original multiple global electrical buses are replaced by optical buses. Each optical bus is implemented with a star topology. In the optical bus the control signals (such as bus lock, address strobe, read/write, address acknowledge, etc.) and the address and data lines that appear in parallel in the electrical version are serialized and packetized for transmission on the optical fiber. The recipient processes these signals and in a similar manner asserts/deasserts the appropriate signals and lines in response. This design was used so that the optical bus could be made plug compatible with the original electrical bus hardware. The initial version has a transfer rate of 0.85 Mbyte/sec per bus.

A prototype optical bus was built with a "light guiding plate" [38] at Duisburg University. This implementation uses a "metal-clad glass plate" (or a plate made of some other optically transparent material). The optical signals are propagated through this plate and are received through small coupling windows at the destinations. The prototype achieved a transfer rate of 72 Mbit/sec.

Two more recent optoelectronic parallel bus systems have been developed and tested at Duisburg University [41, 42, 49]. One system is called the "Optical Parallel Plate Stack" (OPPS). It consists of a set of circular optically transparent plates which are covered with a cladding of lower refractive index than the plates. The edges of the plates do not have the cladding material. The plates are placed in a stack and optically isolated from each other. Boards which are to communicate with each other via these plates are placed perpendicular to the edges of the plates; thus, the optical bus signals are coupled in and out of the plates through the narrow peripheral edges of the plates. The second system is called the "Optical Parallel Strip Plate" (OPSP). It consists of a rectangular board on to which are placed a number of optically transparent strips running the length of the board. These strips are covered with with a cladding of refractive index less than that of the strips. A number of strips, all running parallel to each other, are placed on to the board and optically isolated from each other. Boards which are to communicate with each other via the optical strips are connected perpendicular both to the optical board's surface and to the direction in which the strips run. The points on the optical strips at which the communicating boards are coupled have the cladding removed so that light may be transmitted and received into and from the strips. The prototype experiments on these systems indicate that data rates in the vicinity of 1 Gbit/sec are feasible for plates 100 mm in diameter (the OPPS system) and strips of length 500 mm (the OPSP system).

The Delft Parallel Processor is an ambitious multiprocessor project that aims ultimately to interconnect fully 1000 processors with each other [17, 18, 19, 26, 27, 28, 29, 30, 31]. The designers' goal is for each processing element to transport every 50 nsec (one clock cycle) one 64-bit piece of data to every other processing element (including itself). That is, the input buffer of every processor contains a copy of every 64-bit output buffer of every other processor. This interconnection method is labeled the "newspaper" broadcast concept and is equivalent to a fully interconnected network of processors. A naive implementation would consist of $64 \times N^2$ fibers (one fiber for every bit; N the number of processors). The designers maintain that CAD/CAM techniques would allow this many fibers to be wired. However, the designers have also been working on developing a free space "kaleidoscope" optical multiplexer which takes the 64 fibers corresponding to the output buffer of a node and creates an optical image that can be transmitted to all the processors' input buffers. An initial laboratory model of the kaleidoscope has been built which fully interconnects 9 nodes. The laboratory model requires 576 input fibers (9 nodes × 64-bit words) and 5184 output fibers (9 nodes each receiving 9 64-bit words).

The Melbourne University Optoelectronic Multicomputer Project has been started by one of the researchers on the Delft project [72, 73, 74]. This project aims to provide "virtual" full interconnection between processors by using a homogeneous topology of sub-networks. Each sub-network is a "fiber trunk," a bundle of fibers which provides full interconnectivity among all the nodes on that trunk. Nodes in different sub-networks may communicate by having their messages relayed across the sub-networks.

Honeywell and Thinking Machines Corp. investigated the use of optical fibers in the Connection Machine (CM-2) [46, 54]. A 64K processor Connection

Machine involves connecting 4K processing elements (each processing element contains 16 processor cells) via a 12-dimensional hypercube network. In the physical structure of the Connection Machine, the board level implements dimensions 0–4 via traces on the board, the cardcage implements dimensions 5–8 via a backplane connecting the boards in the cardcage, and the cabinet implements dimensions 9–11 via ribbon cables connecting the cardcages. Dimensions 0–8 define a *subcube*, which consists of 512 processing elements (which equals 8K processors) in one cardcage. In this joint project, optical fibers were used to replace the ribbon cables in dimensions 9–11, i.e., to connect the 8 subcubes that make up a 64K Connection Machine. The initial optical implementation was reported to run at 400 Mbit/sec. The next step is for the optical link to run at over 1 Gbit/sec, but those results have not been reported yet.

The Nectar system is a fiber optic "network backplane" to connect heterogeneous computers [4, 50, 51, 52, 64]. It is designed to exploit coarse level (task level) parallelism by connecting different types of machines together. A design goal was that a user process on one node be able to send a message to another user process on another node in 100 μsec (excluding the transmission delay on the fibers). The current system uses fibers with a data rate of 100 Mbit/sec. Future work includes upgrading the links to Gbit/sec transmission rates.

International Computers Limited (ICL) of the United Kingdom uses an interconnection network called MACROLAN in their ICL Series 39 Level 30 mainframe system [13, 79, 80]. The MACROLAN is a logical token-passing ring system which connects processing nodes together. It is implemented with optical fibers and a central "port switch" unit. Each node is connected via dual fibers (one for transmitting, one for receiving) to the port switch, which is an active unit, rather than to each other. The logical ring is implemented as follows. Suppose station 1 sends a “go ahead” (GA) message to the port switch. The port switch responds by sending this GA message to the *next* station in sequence, station 2. If station 2 has nothing to transmit, it returns the GA message to the port switch, which then sends the GA message to station 3, the next in the sequence. When a station receives the GA message and has something to transmit, it returns the “start of frame” (SOF) message to the port switch. The port switch now realizes that the station has a message to transmit, and the port switch activates its broadcast mode. The message from the transmitting station is broadcast to all the other nodes. Upon finishing its message, the transmitting station sends the GA message back to the port switch, and the port switch discontinues its broadcast mode and resumes its polling operation. Thus, the GA message is the equivalent of a token which the stations pass around the ring and hold on to as necessary. The initial MACROLAN speed was 50 Mbit/sec. The name MACROLAN is significant because the designers consider the connection among the processor nodes to be a Local Area Network. The designers wanted the flexibility of locating nodes some distance away from each other but also wanted a better transmission speed than LANs of the time.

An optical implementation of a conventional byte-wide time division multiplexed bus is being developed [15, 67]. This project is part of the ESPRIT II OLIVES program. Parallel optical fibers contained in a ribbon assembly are used to transmit the data bits in parallel. The fiber ribbons connect into multiple passive fiber star couplers such that the data transmitted on fiber #1 of a node is broadcast to the receiving fiber #1 of every node. Therefore, parallel data words which are transmitted from any one node are broadcast to all the other nodes. This system acts as simple high speed optical bus since only one node can transmit at a time. There are 12 fibers in each ribbon, one for the clock signal, one for parity checking, and 10 for data. Simulation studies indicate that each ribbon would have a 32 Gbit/sec transmission rate (3.2 Gbit/sec per data line). A demonstration prototype of this architecture is being constructed.

The SYMFONET physical architecture [15, 48, 88] can be used to implement a fully interconnected topology. Each node is connected by a fiber to a central passive star. Each node transmits on a wavelength unique to itself; thus, for N nodes in the system, there are N wavelengths used. Each node also has N receivers, each one tuned to a different one of the transmitting wavelengths. Since each node broadcasts on a different wavelength and receives all wavelengths, each node hears every nodes' message (including its own). Furthermore, all the nodes are synchronized so that all the messages are received simultaneously by all the nodes; each set of N messages (one message broadcast by each node) forms one *frame*. Messages within a frame may be arbitrarily ordered by, say, wavelength, if this property is useful. Ref. [88] reported that an initial implementation would be expected to have eight nodes each broadcasting at 1.2 Gbit/sec; a subsequent implementation would be expected to have 32 nodes broadcasting at 2.4 Gbit/sec each.

An architecture of optical buses called MONET (Multidimensional Optical Network) based upon "D-fiber" has been proposed [39, 58, 59]. The reported D-fiber technology allows, through the combination of a "D" fiber geometry and erbium doped distributed amplification, a large number of taps (Ref. [59] computes 170 based upon current technology). This physical architecture is then proposed as a solution for the MONET system architecture. In the MONET system architecture, three multiplexing techniques are used. First, data can be transmitted over different physical waveguides—space multiplexing. Second, on a given waveguide, different wavelengths can be used—wavelength multiplexing. And third, for a given wavelength on a given waveguide, time multiplexing from different sources can occur. Prototype hardware has been built which demonstrates the feasibility of the D-fiber technology, but no actual MONET system architecture demonstration has apparently been built yet.

A different technique to allow a large number of taps was reported by Prucnal et al. [68]. The authors report that the tapping ratio in their optical taps can be continuously varied and that the tapping ratio can be made extremely small, which allows the tapping of an optical signal with high impedance. Therefore, the combination of low loss and high impedance allow a large number of these taps to be connected to a fiber waveguide. The researchers' theoretical results show that a very large number (several thousand) of taps could be supported on one fiber; their initial engineering experiments demonstrate that approximately 50 taps could be attached to the fiber, but the authors expect the engineering capabilities to increase.

The Multiple Channel Architecture (MCA) was proposed by Wailes and Meyer [86, 87]. In this proposed architecture, the high bandwidth of an optical fiber is divided into multiple channels; each channel is called a virtual bus. The researchers claim that current technology can support up to a 1000 channels on a single fiber, each channel having a data rate of 100–200 Mbit/sec; they believe that not-too-distant technology improvements would allow each channel to transmit at 1 Gbit/sec. In the MCA scheme, each virtual bus transmits its data serially using the well known CSMA/CD arbitration protocol. The MCA scheme proposes that channels be treated as just another resource in the parallel architecture; a parallel program should be assigned a certain number of channels when it is executed just as it is assigned processors, memory, and I/O units.

3.2.2 LAN Architectures

Because of the high bandwidth of optical fibers, much interest has been expressed in adapting this technology to Local Area Networks (LANs). Various strategies have evolved for accessing the bandwidth. Since the IEEE 802.3 standard protocol (commonly known as Ethernet or CSMA/CD) has been popular, attempts have been made to implement it in optical fibers; Fibernet [76] and ISOLAN [71] are two examples. Maxemchuk [62] outlines several variations on "random access" protocols for optical fibers. These variations are based upon ALOHA, CSMA, and CSMA/CD.

Another strategy is represented by FDDI [69, 70], which is a token ring protocol. The DQDB protocol [83] is a reservation access scheme employing two unidirectional channels (data on one channel flows in the opposite direction from the other channel).

A protocol strategy called *attempt and defer* is most closely related to our work. These protocols (examples are U-Net [32], Expressnet [84], and D-Net [85]) make explicit use of the *unidirectionality* of the optical fiber. The unidirectionality property comes naturally in optical fiber because the light (coming from the transmitter) is easily coupled into the fiber in one direction.

The unidirectionality property is used by these protocols as follows. Stations monitor the activity on the waveguide (knowing that the signal comes from only one direction). A station is allowed to *attempt* to access (i.e., transmit) the waveguide when it detects no activity (no signal), also called the "end of carrier." However, an upstream station's transmission can reach the station after it has begun its transmission. The upstream station's transmission will then collide with the given station's transmission. This collision is the given station's signal to *defer* (or cease) any further transmission. Since the collision between the two stations' transmissions makes both transmissions unintelligible, then this technique necessitates that each transmission have preamble bits (carrying no useful information) which can be purposely garbled if need be. The station then resumes monitoring the waveguide and waits for the next quiet time. These protocols can be termed as having *passive* permission in accessing the waveguide: a station is not explicitly told by an upstream station that it now has permission to access the waveguide. Rather, the station keeps attempting to transmit (and defers as necessary) until no upstream station's transmission collides with its own transmission. These protocols are also known as *implicit token* passing protocols since, rather than passing an explicit token signal from station to station, the stations use the "end of carrier" event as an implicit token. These protocols are also characterized as being *demand access* because stations attempt to access the waveguide only when they have something to transmit; the bandwidth of the channel is allocated to a station only when it requires it.

Melhem, Chiarulli, and Levitan [12, 63] looked at various LAN protocols, but they felt that none were entirely appropriate for a system backplane. Therefore, they have developed a protocol (which they do not name, but which we designate the MCL protocol after the authors' names) which they feel is appropriate for a system backplane. Although the researchers did not make the connection, the MCL protocol is nearly identical in philosophy to the D-Net protocol (which is why we discuss the MCL protocol in this section). The major difference is that whereas the D-Net protocol uses what we have called the passive permission (attempt and defer) access technique, the MCL protocol specifies that when a station has finished its transmission, it will notify the downstream stations of that fact. A station knows that it may access the waveguide without conflict when all upstream stations have explicitly notified it that they are finished with their transmissions. We call this technique an *active permission* access control method.

3.3 The Optical Broadcast Ring (OBR)

The major problem with the previously developed optical waveguide architectures described in Section 3.2 is related to one of the themes of this thesis. Specifically, all of the previous architectures are used as simple substitutions for the electrical backplane. In all these architectures, the optical backplane is used just to transmit the data faster. Even in the MCA scheme, in which the multiple channels are to be allocated like any other system resource, the channels are just expected to be high speed substitutions for electrical buses. One of the results of this thesis is that we show that the physical optical architecture can be used not only as a high speed transmission channel, but also that the optical architecture itself can be used to implement useful parallel programming features. From our qualitative study of system backplane and LAN architectures (as related in Section 3.2) and our detailed simulation studies of TDMA, CSMA/CD (Ethernet), and CDMA LAN protocols (previously reported in [16]), we concluded that existing optical architectures were not appropriate to building useful general purpose parallel programming architectural features. In this section we will describe our new physical optical architecture; this description is fundamental to understanding how the parallel programming features described in Chapters 4 and 5 are implemented.

Several features of our *Optical Broadcast Ring* (OBR) architecture make it different from previously proposed optical waveguide architectures in several ways. First, the OBR architecture does form a logical ring. The ring property allows signals and messages to be ordered by the placement of nodes on the waveguide. Second, however, unlike a normal ring, the OBR architecture is not a point to point topology. Each attached station taps the signal going by it and uses that signal as it sees fit. A station does not "intercept" a signal on the OBR and decide whether it should be rebroadcast to the next station. Third, the physical topology (subsequently explained in detail) is such that every signal transmitted by a station is automatically broadcast (if desired) to every station (including itself).

There are actually two OBR architectures. One OBR architecture implements an wired-`OR` transmission channel only; no explicit messages consisting of a series of bits are transmitted. The second OBR architecture uses a control channel, similar to the wired-`OR` OBR architecture, to arbitrate access to a high speed channel on which explicit messages are actually transmitted. Thus, in the first OBR architecture, the wired-`OR` transmission channel forms one logical OBR channel whereas the second OBR architecture uses two channels, a control channel and a transmission channel, to form one logical OBR channel.

3.3.1 Wired-`OR` OBR

In the wired-`OR` OBR architecture no explicit bit messages are transmitted, i.e., no series of bits is sent over the waveguide. Instead, a message is represented by the raising of the signal level. The transmitter is turned on and held on until the "message" needs to be turned off, at which time the transmitter is turned off. Therefore, the optical transmitters do not have to be modulated; they just have to be able to inject some level of light into the waveguide. The optical receivers need to be capable of detecting only the presence or absence of light; they do not have to produce a modulated electrical signal to represent a stream of bits.

The first characteristic of this wired-`OR` OBR architecture is its topology, shown in Fig. 8. Each node can have three taps into the channel. Depending upon the architectural feature being implemented, not all three taps are necessary. The "U" tap, which is optional (depending upon the architectural feature), is the sum of the Upstream stations' signals, i.e., the signal at the "U" tap is the wired-`OR` of the Upstream stations' signals. The "T" tap, which every node must have, is the Transmit tap of the node via which its own signal is injected into the waveguide. The reader should note that the optical signal is injected into the waveguide in

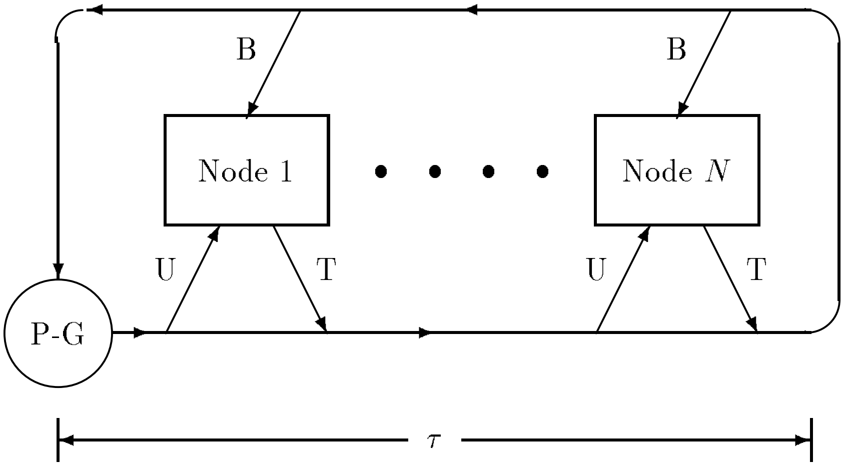

Figure 8: The wired-`OR` OBR architecture topology.

one direction only. In order to give this architecture a broadcast capability, the waveguide loops back so that signals travel in the opposite direction from which they originated. The "B" tap, which, like the "U" tap, is optional, is the wired-`OR` of *all* of the nodes' signals (including its own); thus, the "B" tap is the Broadcast sum of all the nodes' transmissions. The reader should realize that detecting the precise level of the wired-`OR` optical signal is unimportant. The detector only needs to determine if the input ("U" or "B") tap's optical signal is either zero (dark) or at some level. The final component of the topology is the Pulse-Generator, which is optional, depending upon the architectural feature being implemented. Its role is to create a pulse of light periodically; more details are given below.

The wired-`OR` OBR architecture's second characteristic is the access protocol. The access protocol determines how nodes competing to inject a signal into the waveguide are allowed to access the waveguide. For the wired-`OR` OBR architecture there are two access protocols; the choice between the two is dependent upon the architectural feature being implemented (as explained in Chapters 4 and 5).

The first access protocol is very simple: whenever the nodes are permitted to access the wired-`OR` OBR at all, no mediation among them is needed. A node turns its signal on and off at will without regard to when any other node is turning its signal on or off. From a physical viewpoint, this protocol works because the OBR is acting like an `OR` gate. Since the level of the optical signal is unimportant, multiple signals can arrive at an input tap simultaneously, i.e., signals do not interfere with each other. From an architectural feature viewpoint, this protocol is useful if the architectural feature does not need to know what upstream stations are raising their signal or just needs to know when the "B" tap goes dark. The Pulse-Generator is not needed for this protocol.

The second access protocol imposes an order on when nodes can raise and lower their signal (via the "T" tap). Intuitively, the nodes act as follows. A node is not allowed to raise its signal until it sees some signal level on its "U" input tap. Once the node's signal has been raised, the node cannot lower the signal until it sees no signal level (darkness) on its "U" input tap. Thus, as long as some upstream station is transmitting its signal, a node may turn on its signal (we say that a node has *jump-on permission*, a property not present in the MCL protocol). However, the earliest a node may turn off its signal is after all upstream nodes have turned off their signals (turning off the signal is giving *active permission* to the downstream node, a property not present in the D-Net protocol). The nodes' behavior can be summed up by the following rules:

$$\Uparrow\mathrm{T} \quad \text{WHEN} \quad \mathrm{U} \tag{1}$$

$$\Downarrow\mathrm{T} \quad \text{WHEN} \quad \overline{\mathrm{U}} > \text{drop time} \tag{2}$$

We use the following notation: $\Uparrow$T indicates turning on the "T" tap, $\Downarrow$T indicates turning off the tap, U indicates the input signal at tap "U" is at some level, and $\overline{\mathrm{U}}$ indicates the input signal at tap "U" is zero (dark). The "drop time" specified in Rule 2 comes about for the following reason. Suppose that a node did not initially begin transmitting on "T" when it first saw "U" go high, but the node later needs to begin transmitting. Rule 1 allows the node to transmit as long as "U" is high. But, further suppose that just after the node saw that "U" was high and started the physical process of turning on its transmitter, "U" went low. By the time the node (call it i) gets its signal injected into the waveguide, a small amount of darkness has already gone by i and started toward $i+1$. Therefore, node $i+1$ will see its "U" input tap go low (for a short period), then go back up to some level (because it receives the raised signal from node i). The circuitry to implement Rule 2 needs to ignore this drop time (via, e.g., an integrator).

The worst case length of the drop time can be determined by considering that if the node used the "U" input signal to make its decision to turn on "T" just as the tail end of the signal from node $i-1$ passes by "U", then the tail of that signal will have traveled some distance past "T" by the time node i gets its signal turned on through "T". By using the optical geometry (shown in Fig. 9) and the physical response time of the node, we can develop the following equation for the

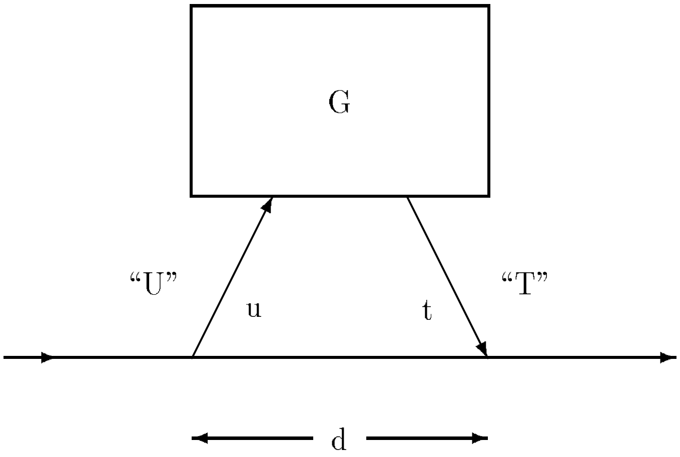

Figure 9: The wired-`OR` OBR drop time geometry.

drop time (in units of time):

$$T_{drop} \leq G + \frac{u+t}{c_w} - \frac{d}{c_w}.$$

The time G is the time for the electrical signal to be generated by the "U" detector and propagate through the electrical circuitry plus the time to turn on the transmitter at "T". The distance u is the length of the tap "U" from the node to the waveguide itself; the distance t is the corresponding length of the tap "T". The distance d is the separation on the waveguide between the input tap "U" and the output tap "T". The parameter c_w is the speed of light in the optical waveguide. If we assume the parameters have values of $G = 20$ nsec, $u = t = 0.01$ m, $d = 0.02$ m, and $c_w = 0.2$ m/nsec (corresponding to an index of refraction in the waveguide of 1.5), then only the first term of the equation is significant, and $T_{drop} \leq 20$ nsec. Lengthening the separation between the "U" and "T" taps (increasing the parameter d) to a large enough value would eliminate the the drop time, but there are two problems with this solution. First, for $G = 20$ nsec, d would have to be approximately 4 m, a length of waveguide which might not be easily engineered. Second, and more importantly, signals would always be delayed at *each node* by the 20 nsec just to account for and eliminate the worst case drop time; adding such a delay at every node would add an unacceptable propagation delay penalty for the entire system.

For simplicity in the notation, we assume for the rest of this thesis that the drop time has been implicitly accounted for in the access protocol rules. Therefore, Rule 2 becomes:

$$\Downarrow\mathrm{T} \quad \text{WHEN} \quad \overline{\mathrm{U}} \tag{3}$$

The Pulse-Generator is needed in this access protocol in order to "restart" the access cycle. The Pulse-Generator sends out a pulse of light when it sees its input tap "I" go dark (for longer than the drop time) after having seen it go high:

$$\text{P-G} \quad \text{WHEN} \quad \mathrm{I}\!\downarrow \tag{4}$$

We use the I↓ notation to mean the falling edge of the input since the Pulse Generator should not send out a new pulse any time it sees a dark input. After sending out a pulse, it will see its "I" tap go to some optical signal level for some length of time. The length of time will depend upon the behavior of the nodes accessing the OBR during this cycle. After seeing its "I" tap go to some level, it cannot send out the next pulse until it sees the "I" tap go dark again (the falling edge). In effect, the Pulse-Generator's "I" tap acts as a Broadcast wired-`OR` of the pulse and the "T" taps of the nodes. The length of the pulse needs to be just long

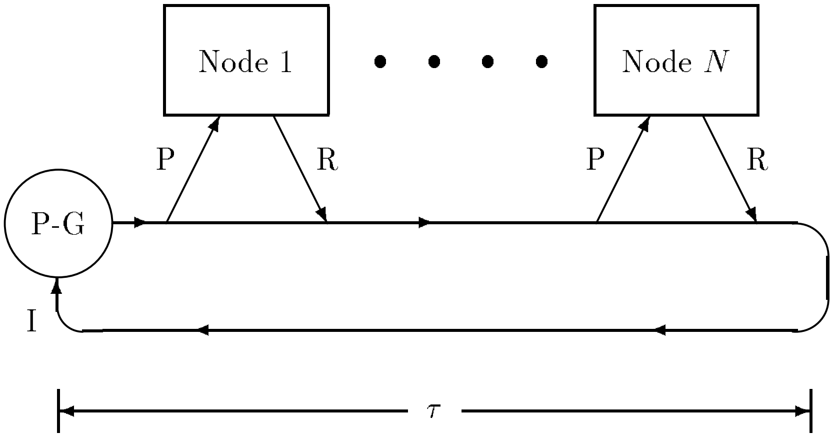

Figure 10: The control channel topology of the explicit message OBR architecture.

enough to trigger the "U" tap's detector; the detector's output then drives the circuitry which implements Rule 1. We call use of the Pulse Generator the OPAC (Optical Pulse Access Control) protocol.

3.3.2 Explicit Message OBR

Two physical channels are used in the explicit message OBR architecture to form one logical OBR channel. One of the physical channels is used as a *control* channel to arbitrate access to the *transmission* channel. The control channel is very similar to the previously described wired-`OR` OBR architecture; an optical signal is raised or lowered on the channel. The transmission channel is a high speed channel on which explicit messages (a message being defined as a set of bits) are sent.

The control channel's topology is shown in Fig. 10. It is the same as the wired-OR OBR topology except that there is no need for the nodes to receive the broadcast sum of the control signals. The “P” input tap is the sum (wired-OR) of the upstream nodes’ Priority request signals. The “R” output tap is used by the node to inject its Request signal (in one direction) into the optical waveguide. As in the wired-OR OBR architecture, the detectors need only detect the presence of some level of optical signal or zero level (darkness), and the transmitters do not have to be modulated. The Pulse-Generator is used, as before, to create pulses of light periodically. It receives via its “I” input tap the broadcast sum of the Request signals and its own pulse.

As with the wired-OR OBR architecture, there is an access protocol which the nodes must follow in order to use the control channel. The nodes follow nearly the same rules as the ones for the wired-OR OBR channel, but now the rules are also linked with the actions the node takes on the transmission channel. The first rule specifies the action the node takes on the control channel when a node wants to request time to put a message on the transmission channel:

$$\Uparrow\mathrm{R}\quad\text{WHEN}\quad\mathrm{P}\tag{5}$$

Thus, as long as any upstream station is continuing to request access to the transmission channel by asserting its “R” signal, a given node will see some optical signal level on its “P” tap and can then assert its own “R” signal (*jump-on permission* again). Note that asserting the “R” signal is *not* the same as transmitting a message on the transmission channel; asserting the “R” signal is *requesting* time on the transmission channel.

In order to understand when time on the transmission channel is granted, we

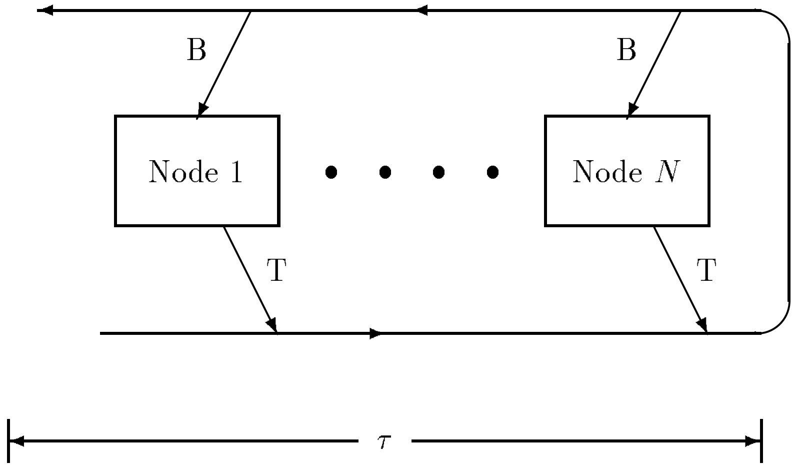

Figure 11: The transmission channel topology of the explicit message OBR architecture.

must specify the topology of the transmission channel. This topology is illustrated in Fig. 11. Each node has two taps. The output tap "T" is used to inject (in one direction) the message into the optical waveguide. Since the input tap "B" is on the loop-back portion of the waveguide, it is used by a node to receive every message every node sends out (including its own messages). The transmitter and detector at a node must be capable of transmitting and receiving (respectively) a modulated light signal which represents a series of bits. The rule governing when a node is granted its transmission time on the transmission channel is tied to the "P" tap on the control channel (again, the drop time is implicitly accounted for):

$$\Uparrow\mathrm{T} \quad \text{WHEN} \quad \overline{\mathrm{P}} \tag{6}$$

This rule is also an expression of the *active permission* property described earlier. The "drop time" condition comes about for the same reason as it did in the wired-`OR` OBR architecture. Meanwhile, the "R" signal on the control channel should stay asserted during the transmission:

$$\text{R} \quad \textsc{while} \quad \text{T} \tag{7}$$

Finally, when the node has finished its transmission, it should drop its “R” output signal on the control waveguide:

$$\Downarrow\text{R} \quad \textsc{at} \quad \Downarrow\text{T} \tag{8}$$

By dropping its “R” signal, the node gives permission (active permission in terms of Section 3.2.2) to the next downstream node that has requested transmission time to begin its transmission. Once the last requesting node in the access cycle has lowered its “R” signal, the Pulse-Generator will see the falling edge at its “I” input tap, which signals it to generate another pulse to begin the next cycle:

$$\text{P-G} \quad \textsc{when} \quad \text{I}\downarrow \tag{9}$$

which is identical to Rule 4 in the wired-`OR` OBR architecture for generating pulses as needed. And as in the wired-`OR` OBR architecture, the length of pulse needs to be long enough to trigger the “U” tap’s detector, whose output then drives the electronic circuitry. These rules define the OPAC protocol for the combination control and transmission channels.

By following these rules of access to the waveguide, the nodes generate a “train” of messages during each access cycle, as illustrated in Fig. 12. Note that the ordering imposed upon the messages, as heard by each node at its “B” input tap, is the physical order of the nodes on the waveguide. Note also that the optical waveguide acts as a pipeline of messages; a node does not wait until its message

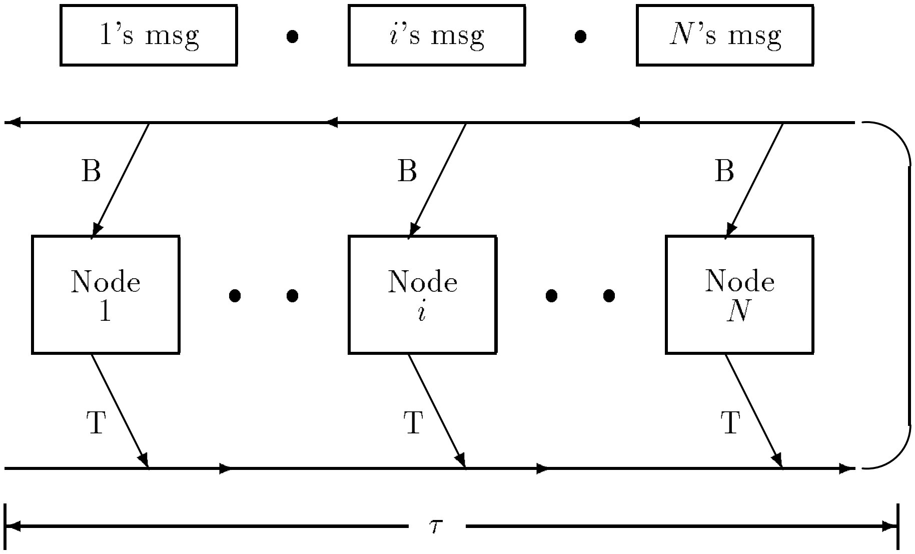

Figure 12: A "train" of messages on the explicit message OBR architecture's transmission channel.

has been received by every node. The reader will recall that a node waits to put its message on the transmission channel until all upstream nodes have finished their transmissions; in terms of message trains, this action is interpreted as meaning that a node adds its message to the end of the train as the end of the train passes by its "T" transmit tap.

Two questions arise as to the overhead (or throughput efficiency) of trains. The first question regards the wasted space between messages of a train. If Rule 8 is strictly followed, then the "R" signal is not dropped until the message transmission at "T" has quit. When the dropped "R" propagates from node i to the next downstream station $i+1$, there will be a finite delay before node $i+1$ can start its transmission. This delay results from three components: (1) the detector's response time to seeing the "P" tap go dark; (2) the gate delay of the circuitry implementing Rule 6, including the delay to see if the darkness lasts longer than the drop time; and (3) the transmitter's response time. Let us call the amount of this delay Δ. Without any compensation the distance between adjacent messages in a given train will be this Δ. However, we can compensate for this delay Δ by adjusting Rule 8 to be:

$$\Downarrow \mathrm{R} \quad \text{AT} \quad \downarrow \mathrm{T} - \Delta \tag{10}$$

so that the "R" signal is turned off by the amount Δ time *before* the message transmission is turned off. Since the enumerated components of the delay Δ may not be precise because of variations in the optical and electronic hardware, Δ must be computed for the worst case. Nevertheless, the separation between messages in a train can be made arbitrarily small by using Rule 10.

The second question regarding the overhead of trains concerns the separation between successive trains, i.e., the overhead between successive access cycles to the OBR. Because Rule 9 specifies that the Pulse Generator must wait until its "I" input tap goes dark (for longer than the drop time), it will take at least τ time for the drop in the "R" signal at the last node to reach the "I" tap of the Pulse Generator. If either the message length is variable or the number of nodes attached to the OBR is unknown, no improvement can be made. However, if the message length from a node is fixed and the number of nodes is known, then Rule 9 can be rewritten:

$$\text{P-G} \quad \text{WHEN} \quad (\mathrm{I}\downarrow) \ \texttt{.OR.} \ (\text{ Timer expires }) \tag{11}$$

When the pulse is generated, the Timer is set to expire $N \times (L + \delta)$ time units later, where N is the number of nodes, L is the (fixed) message length, and δ is the (unavoidable) arbitrarily small separation between messages in a train. The Timer, in effect, dictates the maximum length of the train and if maximum length trains are being generated, causes a new pulse so that there will be minimal wasted time between trains. Because of the geometry of the control channel, if the Timer condition causes a new pulse to be generated, there will not be a subsequent false drop in the signal level at "I" which would incorrectly cause a new pulse. When the Timer is used as specified in Rule 11, we call this *scheduled permission*, a feature not present in either the D-Net or MCL protocols.

3.3.3 Analysis of the OBR Architectures

There are two performance metrics of interest for the wired-`OR` OBR and explicit message OBR architectures. They are the analytical worst and best case access times and the propagation time over the OBR. Access time means the time from when a node decides it wants to start a transmission to when it is actually allowed access to the OBR. Propagation time is defined as the time it takes a message to traverse the OBR to its destination once transmission has begun; it includes both the bit transmission time (how long to put out the bits, e.g., 1 Gbit/sec) and bit reception time (assumed to be the same as the transmission time) and the time to travel the physical distance over the OBR. Thus, response time, in this context, is *not* the same as either access time or propagation time or their sum. Understanding this analysis is necessary to understanding the subsequent analysis (presented in Chapter 6) of the architectural features built from the wired-`OR` OBR

Protocol	*Minimum*	*Maximum*
electrical	$\Delta + \tau + \Delta$	$\Delta + \tau + \Delta$
wired-`OR` – no protocol	Δ	Δ
wired-`OR` – OPAC	2Δ	$2\tau + \Delta + 2\Delta$
em-OBR (var)	2Δ	$2\tau + \Delta + 2\Delta$
em-OBR (fixed)	2Δ	$\min(2\tau, NT_m) + \Delta + 2\Delta$

Table 1: The minimum and maximum access times for the OBR architectures under idle load.

and explicit message OBR physical architectures.

Table 1 shows expressions for the minimum and maximum access times under the OBR architectures (a normal electrical bus is also shown for comparison) under idle load conditions. Several assumptions have been made in deriving these expressions. First, as shown Figs. 8, 10, and 11, the propagation distance from node 1 to node N is τ and vice versa. Second, Δ represents the various component and circuitry delay times of the optical protocols all lumped into one value (this is the same Δ shown in Rule 10). Examples of such delays include logic gate delays and detector and transmitter response times. For the purposes of this discussion, we also represent the component and circuitry delay time of the electrical bus by Δ. Third, idle load means that only one node is attempting access to the OBR channel. Fourth, both explicit message OBR OPAC protocols are discussed, namely, the variable length message version and the fixed length message version (recall that the latter allows a timer to control partially when the Pulse Generator generates a pulse).

We derive the expressions shown in Table 1 as follows. For an electrical bus, the node takes logic time Δ to put its request out. It then takes propagation time τ for that request to go out. Another logic time Δ elapses before the node realizes it controls the bus and can transmit its data. Since on an electrical bus the propagation time τ is position independent, the minimum and maximum access times are the same. For the wired-`OR` OBR with no access protocol, by definition, there is no need to arbitrate for access, so the only delay is always the logic time Δ. For the wired-`OR` OBR using the OPAC protocol and the explicit message OBR (which always uses the OPAC protocol), the different access times arise because of where the pulse might be. In the best case (minimum time), the node decides to access the OBR *just before* the pulse arrives; therefore, the only delay is the logic time 2Δ it takes to recognize the pulse and turn on the signal or message. In the worst case (maximum time), the pulse has gone by the node *just after* the node decided to access the OBR; thus, for the wired-`OR` OBR OPAC and the explicit message OBR with variable message length, the node waits $2\tau + \Delta$ (round trip propagation time plus the logic time for the Pulse Generator) for the next pulse to come around, and then it takes the logic time 2Δ for the node to recognize the pulse and turn on the signal or message. The term of 2τ is the separation between message trains (or access cycles). For the explicit message OBR with fixed length messages, since the timer for the Pulse Generator is in use, the node waits $\min(2\tau, NT_m) + \Delta$ (the minimum of the round trip propagation time or the full message train length, depending upon the design parameters of the system, plus the logic time for the Pulse Generator) for the next pulse to come around, and then it takes the node time 2Δ to recognize the pulse and turn on the signal or message.

In Table 2 we show the expressions for the minimum and maximum access

Protocol	*Minimum*	*Maximum*
electrical	$\Delta + \tau + \Delta$	$N(T_m + \tau + \Delta) + \Delta$
wired-`OR` – no protocol	Δ	Δ
wired-`OR` – OPAC	2Δ	$NT_m + 2\tau + \Delta + 2\Delta$
em-OBR (variable msg length)	2Δ	$NT_m + 2\tau + \Delta + 2\Delta$
em-OBR (fixed msg length)	2Δ	$NT_m + \Delta + 2\Delta$

Table 2: The minimum and maximum access times for the OBR architectures under high load.

times under the condition of high load. The same assumptions as for the idle load are made with several additions. First, it is assumed that the average duration of a node's signal (under the wired-`OR` OBR with OPAC protocol) and the average length of a message (under the explicit message OBR) is T_m. Second, for clarity in the discussion, the value T_m includes the unavoidable small separation between messages (or signals) as applicable; for the wired-`OR` OBR using OPAC, this separation is the drop time; for the explicit message OBR (which always uses OPAC), this separation is the small δ value. Third, high load means that all the nodes are contending for access to the OBR; if a large number of nodes are in the system, then the other $N - 1$ nodes contending for access can be approximated by N for clarity.

We now derive the expressions shown in Table 2. For the electrical bus, if the node happens to be the next one in the arbitration order, its minimum access time is the same as under the idle load condition: a logic time Δ to put out its request, propagation time τ for the request to go out, and another logic time Δ for the node to realize it controls the bus and begin transmitting the message. For the worst case, since the node will be the last one to gain access, there will be the data transmission time plus arbitration grant and response time (T_m and $\tau + \Delta$ respectively) for each of the prior nodes; then, the node will have its own response time Δ. For the wired-`OR` OBR with no access protocol, as under the idle load condition, by definition, there is no need to arbitrate for access, so the only delay time is always the logic time Δ. For the wired-`OR` OBR with OPAC and the explicit message OBR (which always uses the OPAC protocol), the difference between the minimum and the maximum access times depends upon where the pulse might be when the node initiates access. In the best case (minimum time), the node decides to access the OBR *just before* the pulse arrives; therefore, the access time is only the logic time 2Δ it takes to recognize the pulse and begin transmitting the signal or message. For the wired-`OR` OBR (with OPAC) and the explicit message OBR (with variable message length), the worst case (maximum time) occurs when the pulse has gone by the node *just after* when the node has decided to access the OBR. There will be N messages of average length T_m transmitted before the pulse comes back around; in addition, the nodes waits $2\tau + \Delta$ (round trip propagation plus logic time for the Pulse Generator) before the next pulse actually arrives at the node; then, the node takes 2Δ to recognize the pulse and turn on its signal or message. The 2τ term is the separation between message trains (i.e., separation between access cycles). As discussed previously, incorporating a Timer in the Pulse Generator can eliminate this overhead; this improvement is reflected in the maximum access time for the explicit message OBR with fixed message lengths: if the pulse just misses the node, the node only has to wait until the other N nodes have transmitted their messages of length T_m.

Protocol	*Minimum*	*Maximum*
electrical	$T+\tau$	$T+\tau$
wired-`OR`	T	$T+2\tau$
em-OBR	T	$T+2\tau$

Table 3: The minimum and maximum propagation times for the OBR architectures.

Table 3 shows the minimum and maximum propagation times for the OBR architectures under any load conditions. For this analysis, we assume that T is the bit transmission time. For the electrical bus, this term is usually bit-parallel. In the wired-`OR` architecture, this term represents turning on the signal at the source and its reception at the destination (or, the turning off and subsequent detection of the signal. For the explicit message OBR architecture, this term is the bit-serial transmission of the message. As before, the propagation distance from node 1 to node N is τ and vice versa. The reason that the propagation times are the same under any load conditions is that once a node has access to the medium, the propagation time is dependent solely upon the bit transmission/reception rate and the physical distance. The expressions in Table 3 are derived, then, by considering the geometry of the network. For the electrical bus, after the signal is generated (time T), it must propagate out to all the nodes (time τ). The minimum and maximum time are the same since the electrical signal must "settle" on the wire, i.e., propagate (at a minimum) the time τ and be stable. In the wired-`OR` architecture, the minimum time occurs when node N receives its own transmission, i.e., there is essentially no physical distance for the signal to traverse (see Fig. 8). The maximum time occurs when the first node receives its own transmission. Since the geometry of the explicit message OBR is the same as that of the wired-`OR` (see Fig. 11), the expressions for the minimum and maximum propagation times are also the same.

Chapter 4

Cache Coherency

As mentioned in Section 2.2, there is a memory access cost associated with providing a global memory when the memory modules are physically distributed among the nodes. This cost is the possibility of the long latency of a ReSM access. There are three factors which contribute to the latency [78]. The first factor is that a ReSM access must be made over the interconnection network. Use of the interconnection network always implies more latency since the interconnection network connects nodes which are not physically adjacent. That is, there is some physical distance over which data must travel when making ReSM accesses, much more distance than when a node is accessing its NeSM to which it is directly connected. The second factor is contention for the interconnection network. Since all nodes making ReSM accesses use the common interconnection network, the nodes must contend with each other for use of the interconnection network. In contrast, when a node makes a NeSM access, our model has assumed the use of a dedicated connection to its NeSM, so there is no contention latency for a NeSM access. The third factor is memory contention. If several nodes simultaneously present requests to a memory unit, the unit will not be able to satisfy those requests simultaneously, so some of the requesting nodes must wait for other nodes' requests to be satisfied.

Insofar as possible, a goal of the DSM architecture is to alleviate the difference between the NeSM and ReSM access latencies. The solution for doing this, as described in Section 2.2, is to introduce a local, private cache at each node. When a node makes a memory reference, the cache is first searched for the reference. If the search is successful, then the reference is returned. If the reference is not in the cache, then the cache asks the network controller (as shown in Fig. 4) to obtain the reference for it, regardless of whether the reference will be in the NeSM or in a ReSM. When the reference is returned, then the cache, as in a uniprocessor system, keeps it for future use. By this means, the cost of memory accesses (both NeSM and ReSM) is lessened if the locality of the reference is enough, and the difference between NeSM and ReSM access latencies becomes smaller.

The trouble with introducing local, private caches at each node is the well known *cache coherence* problem [78]. The problem is that since several nodes may request the same memory reference, then multiple caches may each hold a copy of the memory location; when one node needs to make a change to its copy of the memory location, how are the copies existent in the other caches affected? Those other copies cannot be ignored or else their respective nodes can access "stale" data, a situation which can cause incorrect program results.

The previous standard hardware solutions fall into two categories: *write-invalidate* and *write-update* [78]. Both of these solutions assume that some arbitrary number of copies of a memory location can exist in the local caches. The idea behind the write-invalidate scheme is that when one node needs to change a memory location (held in its cache), then the other copies are invalidated; the node can then make changes without being concerned with maintaining the copies since there are no other copies. When another node needs to reference its (now) invalidated copy of the location, it will have to obtain the fresh value. In the write-update scheme, when a node makes a change to its cached copy, then all the other copies are updated to reflect the change. Other nodes now automatically have the correct copy in their caches of the memory reference.

There are various ways to implement the write-invalidate and the write-update coherence policies. Archibald and Baer [3] reviewed different *snoopy* implementations of both policies. Snoopy implementations are thusly named because they require that each cache listen hear and process appropriately all the coherence commands issued by all the caches. Bus based parallel computers lend themselves well to using snoopy implementations because of the broadcast nature of the bus, but they cannot support a large number of processors because of the bandwidth limitation of the bus. Agarwal et al. [1] and Chaiken et al. [9] reviewed various *directory based* schemes. Directory based schemes rely upon maintaining, in some manner, where the different copies of a given memory location reside. This directory information is then used to implement whichever coherence policy is desired. Directory based implementations are suitable for systems with arbitrary interconnection networks in which it is assumed that broadcast is either unavailable or very expensive. The *full-map* scheme [8] is the original directory method, but the size of the directory can present a problem. A full-map directory is unwieldy, in a large system, because each directory entry of a memory line must be able to track whether every node has a copy of that line. Since the Stanford DASH machine [56] distributes its memory among the nodes, the directory itself is also distributed. Since each node maintains which remote nodes have a copy of memory locations in that node's portion of the global memory, this scheme is also a full-map directory. Recent work for the MIT Alewife machine [10] has resulted in a *limited* directory scheme in which the hardware can maintain a certain number of entries (the assumption being that regardless of the number of nodes in the system, only a limited number normally would have copies of the memory reference). If necessary, however, their LimitLESS scheme can use software storage to emulate a full-map directory. In both the full-map and limited schemes, the directory information for a particular memory reference is centralized so that ascertaining where invalidation or update messages need to be sent is easily done. In contrast, the Scalable Coherent Interface protocol [40] uses a *chain* directory scheme which relies upon a distributed chain, or linked list, of nodes possessing copies of the memory reference; since the elements of the linked list are distributed among the nodes, this property allows the size of the directory information to grow (and shrink) as necessary. The drawback to this scheme is that coherence information must be passed from element to element in the chain since the directory information is not centralized.

4.1 Cache Coherency in Beehive

Common to all the previously described cache coherence schemes is their being oriented from the *writer's* point of view: the schemes all assume that if a copy of a memory location exists, the writer must do something (invalidate or update) to it. The opposite orientation is termed *reader initiated* cache coherence, as proposed by Lee and Ramachandran [55]. In reader initiated cache coherence, the reader

`head-of-list`

Figure 13: The memory line's directory entry in Beehive.

`update-bit`	$d_1 d_2 \ldots d_k$	`prev`	`next`

Figure 14: A cache line's directory entry in Beehive.

of a memory reference has the responsibility for deciding whether to be informed of changes to his cached copy of the memory reference. Contrast the reader's responsibility in the Beehive scheme to the writer's responsibility in the previously described schemes in which the writer assumes that all readers of a copy of the memory reference want to know about changes.

The Beehive architecture's reader initiated cache coherency is a version of the chain directory scheme since Beehive's interconnection network is arbitrary and unspecified. The Beehive scheme works as follows. Each line of memory has an associated directory entry (see Fig. 13), and each cache line (in every node's cache) also has an associated directory entry (see Fig. 14) (for simplicity in this discussion, we have omitted the fields used for lock synchronization—these fields will be shown in Chapter 5). Together, the memory and cache line directory entries form a chain of which caches contain a copy of a memory location *and* (to implement the reader initiated point of view) are interested in knowing about changes to the memory location.

The memory line's directory entry contains a pointer to the first cache which contains a copy of that memory line. If no caches contain a copy, then that pointer is `NIL`. When the first cache requests a copy, then the memory's directory entry is modified to point to that cache. The cache line's directory entry (for that memory line) is updated so that the `prev` and `next` pointers form the links in the chain. If the cache is interested in knowing about future changes to the memory location, then the `update-bit` is set. When the cache is no longer interested in future updates, then the `update-bit` is cleared, and the cache line's directory entry is removed from the doubly linked list. The `update-bit` is also cleared and the cache line's directory entry removed from the chain when the cache line is replaced.

Beehive also addresses the issues of false sharing (different variables being located in the same memory/cache line) and unrelated shared variables located in the same memory/cache line) by incorporating the *dirty bits* $d_1d_2\ldots d_k$ in the cache line's directory entry. When a cache makes changes to locations in a cache line, the appropriate dirty bits are set so that when the line is written back to memory, only the locations corresponding to the set dirty bits are actually written back.

In order to understand how the Beehive programming model is implemented, the cache coherency policy and buffered consistency commands must be considered together. Table 4 gives these command primitives (and their semantics). These primitives are what the software (generated by the compiler and/or programmer) use to utilize the cache hardware and the write buffer (described in Section 2.2).

The `READ` and `WRITE` commands are used to access private data. Since the data is, by definition, non-shared, there is no need for maintaining coherence. In other words, the cache acts on private data as a uniprocessor cache would.

Read and Write primitives in Beehive.	
Primitive	*Semantics*
`READ`	Read data without cache coherence.
`WRITE`	Write data without cache coherence.
`READ-UPDATE`	Read data with cache coherence by requesting future updates.
`RESET-UPDATE`	Cancel desire for future updates.
`WRITE-UPDATE`	Write data with cache coherence by propagating data globally.
`FLUSH-BUFFER`	Stall until `WRITE-UPDATE`s in Write Buffer have been propagated and completed.

Table 4: The Read and Write command primitives for cache coherency with buffered consistency in Beehive.

The `READ-UPDATE` command is used to read shared data and ask for future changes to the data (thereby maintaining coherence on the data). If the data is not present in the cache, then the request for the data is sent to the memory module containing that memory line. The appropriate pointers in the chain of caches holding this memory line are updated, and the line is returned to the requesting cache. The `update-bit` in the cache line's directory entry is also set to indicate that the requested should be kept up to date. When the `READ-UPDATE` command is issued and the data is present in the cache, then the cache can immediately return the value to the CPU as a local operation.

The `RESET-UPDATE` command allows the node to clear the `update-bit` in that cache line's directory entry. When that bit is cleared, this signifies that the node is no longer interested in maintaining the coherence of that line. The cache line's directory entry is also removed from the doubly linked list. The `update-bit` is also cleared when the cache line must be replaced, and the directory entry is removed from the chain.

The `WRITE-UPDATE` command is used to update shared data and propagate the change to interested nodes. The write is placed into the node's Write Buffer (as shown in Fig. 4). From the node's viewpoint, the update has now occurred. The Write Buffer, through the Network Controller, sends out the update to the appropriate memory module when it can gain access to the network. The memory module then uses the chain to send the updated value out to all caches which want the updated value. Thus, the updates are sent in a point-to-point fashion as each cache receives the updated value and passes it to the next one in the chain.

The reader will recall from the discussion in Section 2.2 that a task reaches the end of its Synchronization Epoch (SE) when the CP-Synch (Consistency Preserving) point is encountered. Although the task considers `WRITE-UPDATE`s to be finished, while in its SE, after issuing the command, when the task reaches the CP-Synch point, the task must issue the `FLUSH-BUFFER` command in order to obey the buffered consistency model. The `FLUSH-BUFFER` command stalls the task until all the `WRITE-UPDATE`s stored in the Write Buffer have been completed. A `WRITE-UPDATE` is deemed completed when the updated value has propagated to the memory module and all interested caches *and* when an acknowledgement saying all the updates have occurred is received by the Write Buffer. Then, and only then, can the Write Buffer remove that `WRITE-UPDATE` as being finished. The acknowledgement is required in the Beehive architecture since the interconnection network is arbitrary and unspecified. The Write Buffer must have an acknowledgement that the propagation of the update is complete since it cannot rely upon any property of the interconnection network to know this fact.

Although the task must stall (via the `FLUSH-BUFFER` command) when it reaches the CP-Synch point, one should remember that the Write Buffer has not been waiting to send out the `WRITE-UPDATE`s until the `FLUSH-BUFFER` command is issued. Rather, the Write Buffer has been sending out the `WRITE-UPDATE`s as soon after the task issued them and it can gain access to the interconnection network. Therefore, the work to maintain the cache coherence can proceed in parallel with the rest of the computation in the task's SE. If there is enough work in the SE that can be structured properly, then by the time the CP-Synch point is reached, there will be no `WRITE-UPDATE`s left in the Write Buffer, and the `FLUSH-BUFFER` command will not cause the task to stall.

4.2 Cache Coherency in OBee

Since the OBee architecture is a derivative of the Beehive architecture, OBee uses the reader initiated cache coherency policy also. However, since OBee uses the explicit message OBR interconnection network described in Chapter 3, the implementation of the reader initiated cache coherence policy is simpler and can be optimized.

Because the OBR (we implicitly are referring to the explicit message OBR in the rest of this section) has the broadcast capability, a *snoopy* approach to reader initiated cache coherency can be adopted. The snoopy approach means that every node listens to the cache coherency commands, just as in the snoopy implementations of the write-invalidate and the write-update protocols. Therefore, there is no need for a memory line directory entry since its only purpose in Beehive

`update-bit`	$d_1 d_2 \ldots d_k$

Figure 15: A cache line's directory entry in OBee.

(as shown in Fig. 13) is to record the start of the list of interested caches. Similarly, a cache line directory entry is now simplified (as shown in Fig. 15) in OBee since it does not need to store the link information. The cache line directory entry only needs the `update-bit` and the dirty bits.

Thus, initially (some modifications will be described below) the command primitives for OBee are the same ones listed for Beehive in Table 4 and work as follows. The `READ` and `WRITE` commands are exactly as in Beehive for reading and writing private, non-shared data. Since OBee has multiple OBRs available, one explicit message OBR (or more) can be dedicated to this traffic. This pre-assigning of an OBR to a particular kind of traffic is an example of one of the themes listed in Chapter 1. The `READ-UPDATE` command is for reading shared data on which coherence is to be maintained. If the data is not in the cache, then the cache requests the value from the appropriate memory module through the Network Controller and sets the `update-bit`. Again, one dedicated explicit message OBR (or more) can be dedicated to this traffic. Unlike Beehive, there is no chain of caches in which the requesting cache needs to be inserted. If the data is in the cache, then it can immediately satisfy the CPU's request as a local operation. The `RESET-UPDATE` command clears the `update-bit` so that the cache will remember that it does not care about using future changes to the cache line; if the cache line is replaced, then the `update-bit` is also cleared. As in Beehive, the `WRITE-UPDATE` command inserts the write into the Write Buffer and lets the write be sent out as the network becomes available. In OBee, however, the updated value only needs to be sent (explicitly) to the memory module containing that memory line. For the time being, assume that only one dedicated explicit message OBR is used for this traffic. Since the update is sent via the OBR and is therefore broadcast to every node indiscriminately, each node's cache listens to the `WRITE-UPDATE`s so broadcast. Each cache determines for itself whether updates are of interest by the following algorithm:

```
REPEAT FOREVER:
  Snoop on OBR and grab WRITE-UPDATE;
  if (WRITE-UPDATE is in cache) .AND. (update-bit is Set) then:
     Update cache line;
     endif.
```

Thus, as in a standard snooping protocol, it is each cache's responsibility to monitor the broadcast coherence information. The OBee cache coherency protocol is not a write-invalidate or write-update scheme in which invalidates/updates are blindly broadcast *and* applied. The reader initiated point of view comes into play because each cache decides, based upon the `update-bit` status, whether to apply the update to the cache line, i.e., the updates are not necessarily used by every cache.

As in Beehive, the `FLUSH-BUFFER` command stalls the processor until the Write Buffer has been emptied. This command can be optimized as described below.

The OBee implementation of the reader initiated cache coherency can be compared against the Beehive specification in several ways. First, consider the number of messages generated by each method when a `WRITE-UPDATE` is sent out:

Beehive	*OBee*
$K+1$	1

K represents the number of caches that have their `update-bit` set for the particular cache line. In Beehive, the first message is generated when the `WRITE-UPDATE` is initially sent to the appropriate memory module. The memory module then forwards the update to the first cache in the chain, which then forwards the update to the next cache in the chain, etc. The $K+1$ messages must occur serially since the updates must propagate link by link through the chain. In contrast, OBee only sends 1 message, which is explicitly sent to the appropriate memory module, but since messages on the OBR are always broadcast, each cache listens to the messages and uses them as appropriate.

A second comparison is that of the transaction time for the `WRITE-UPDATE` command. The transaction time consists of three components: (1) network access latency, (2) network transmission and propagation time, and (3) the cache processing time. In Beehive, because of the arbitrary interconnection network, no guarantees can be made about either the first or second component. The first and second components, coupled with the fact that Beehive requires $K+1$ messages transmitted serially, implies that a `WRITE-UPDATE` in Beehive could take quite some time to propagate to all the necessary nodes. For the OBee implementation, remember that the `WRITE-UPDATE` traffic occurs on one explicit message OBR. From the analysis presented in Chapter 3 (refer to Tables 1, 2, and 3), there is an upper bound on the first and second components. In Beehive, the third component of the transaction time is just the time for the cache to perform the update. No decision is made about whether the cache line needs updating—the update would not have been sent to the cache had it not been part of the update chain. In contrast, in OBee, the cache must determine if the updated line is stored in the cache and whether the copy in the cache has requested updates. Therefore, the third component will take longer in OBee than in Beehive.

The third comparison regards optimization of the `FLUSH-BUFFER` command in OBee. This optimization can occur because of the use of the OBR interconnection network. The key to this optimization is the fact that messages sent out over the OBR in a particular order retain their transmitted order when received by any node. To understand the optimization in OBee, we consider an example in which the processor first places two entries W_A and W_B (in that order) in the Write Buffer via the `WRITE-UPDATE` command, continues its computation in the Synchronization Epoch (SE), then finally issues the `FLUSH-BUFFER` command because it has reached the CP-Synch point that marks the end of the SE. Consider first what happens under Beehive. There are two possibilities as to how the W_A and W_B entries are transmitted. The first possibility (and most restrictive) is that the W_B entry cannot be transmitted until the W_A entry has been transmitted to the memory module, propagated down the list of caches, and the last cache has sent an acknowledgement message to the Write Buffer. Upon receipt of the acknowledgement message, the Write Buffer removes the W_A entry since the `WRITE-UPDATE` transaction has completed. The second, and better, possibility is to let the Write Buffer transmit the W_B entry after transmitting the W_A entry. Since the Beehive interconnection network is arbitrary, this action implies that the W_B entry might complete before the W_A entry (and therefore be removed from the Write Buffer first), but this does not violate the BC memory model semantics. To continue the example, assume that at least one of the entries is still in the Write Buffer when the processor issues the `FLUSH-BUFFER` command. Why must the Write Buffer wait until receiving the acknowledgement from the last cache before it can remove the entry? By definition, when the task leaves the SE, it must know that all `WRITE-UPDATE`s have been *globally* performed, i.e., the memory module and all interested caches have updated their copies. If the processor were to leave the SE by, for example, issuing an UNLOCK (which is a CP-Synch type of command), then, because the Beehive interconnection network is arbitrary, it is conceivable that the next processor to acquire the lock might not have received the update on the entry in the Write Buffer before acquiring the lock. The task newly acquiring the lock would mistakenly assume that the variable was up to date, which assumption would probably lead to an incorrect program result. Therefore, to prevent the BC memory model semantics from being violated, the last cache in the chain must send an acknowledgement to the Write Buffer before it removes the entry. In turn, the `FLUSH-BUFFER` command must stall the processor until all the entries in the Write Buffer have been removed in order to guarantee that when the CP-Synch command is issued, consistency of the updated variables has indeed occurred. As a important reminder to the reader, from Fig. 5, the processor does *not* have to stall while waiting for the CP-Synch command to complete (i.e., be propagated to the other nodes).

Now consider how this example is handled under OBee. We still assume that the `WRITE-UPDATE` traffic is occurring on one explicit message OBR. If we further make the not unreasonable assumption that nodes process received messages in the order they are received, then from the preceding discussion, we can see that the acknowledgement to the Write Buffer is not necessary in OBee because messages sent via an OBR are received in *transmission order* by the nodes. Thus, the W_A entry in the Write Buffer will be received before the W_B entry (assuming nodes process received messages in the order received), and, more importantly, if the task issues the appropriate CP-Synch command over the same OBR, it is guaranteed that the `WRITE-UPDATE`s will be performed by a receiving node *before* it can process the CP-Synch command. Therefore, the scenario, outlined above, of a node receiving an UNLOCK before receiving all the updates, cannot occur. If the Write Buffer stores only `WRITE-UPDATE` commands, then the `FLUSH-BUFFER` command, as in Beehive, translates to stalling the processor until the Write Buffer has emptied. The time to empty the Write Buffer is much shorter since once the entry is transmitted, it is immediately removed from the Write Buffer.

The issuance of the `FLUSH-BUFFER` and the CP-Synch commands can be further optimized. If the appropriate CP-Synch command can be stored in the Write Buffer with the normal `WRITE-UPDATE` entries, then there is no need any more for the `FLUSH-BUFFER` command. Instead, when the task issues the CP-Synch command, it goes into the Write Buffer, and the task can proceed with its code. Since the updates are ahead of the CP-Synch command in the Write Buffer, they will be transmitted first and processed first by the receiving nodes, thereby preserving the BC memory model semantics.

Another optimization can be made if some additional hardware resources are used. So far in this discussion, we have assumed that *all* `WRITE-UPDATE` and

CP-Synch command traffic is transmitted on one explicit message OBR. Considering the previous example, two separate explicit message OBRs could be used for this traffic. One of the OBRs would be the "background" channel: when the Write Buffer contains just `WRITE-UPDATE` commands, it uses that channel. The background OBR is termed thusly because tasks still have work to perform in their SE, and it is not critical that the updates be performed immediately. When a CP-Synch command is put into the Write Buffer, then the Write Buffer uses the "priority" OBR. Even though the task, given the previous optimization, is not stalled, there is a good probability that some other task waiting for the CP-Synch command so that it can proceed into its next SE. In other words, the high priority OBR only carries traffic from tasks ending their SE; therefore, access to and propagation over that OBR is faster, which means that the task waiting for that SE to end can resume its work sooner. We point out that using two OBRs, one of which may have very little traffic, is an example of one of the themes in Chapter 1. The use of two OBRs implies that some `WRITE-UPDATES` in a SE could go over the normal OBR and that some could go over the priority OBR when the SE was finished. Some care has to be taken to ensure that the BC memory model semantics are not violated, i.e., the CP-Synch command is not processed before all the updates are. One way to process the updates and CP-Synch correctly is to label uniquely the `WRITE-UPDATES` and the CP-Synch for a specific SE of a task. Such a labeling could be implemented by letting each node have a simple hardware counter to identify the SEs occurring on that node. As part of the initialization of each SE, the node could issue a command to increment the counter and obtain the resulting value. The CP-Synch command also needs to include how many `WRITE-UPDATE`s are to be performed in that SE. Each receiving node performs the following two algorithms, which requires additional electronic hardware. The first algorithm is executed by the receiving hardware associated with the background OBR, and the second algorithm is executed by the receiving hardware for the priority OBR:

Algorithm 1: (background OBR)

```
REPEAT FOREVER
  Do in Parallel:
    Receive transmission;
    Process WRITE-UPDATE (count how many for each SE);
  Enddo.
```

Algorithm 2: (priority OBR)

```
REPEAT FOREVER
  Do in Parallel:
    Receive transmission;
    if (transmission is WRITE-UPDATE):
       Process WRITE-UPDATE (increment counter for that SE);
    else: //  transmission is CP-Synch
       Wait until the WRITE-UPDATE Counter reaches proper value;
       Perform the  CP-Synch  command;
    Enddo.
```

We could provide the compiler (or programmer) with an additional hardware primitive which would put the `WRITE-UPDATE`s and corresponding CP-Synch on the priority OBR immediately. If the `WRITE-UPDATE`s in a SE were always close to the CP-Synch command with little or no other code able to be put between the `WRITE-UPDATES` and the CP-Synch command, then this primitive would be useful. With all these modifications, the command primitives for OBee are now as shown in Table 5. The `FLUSH-BUFFER` command is no longer needed; its function is now included in the (generic) CP-Synch command.

Primitive	*Semantics*
`READ`	Read data without cache coherence.
`WRITE`	Write data without cache coherence.
`READ-UPDATE`	Read data with cache coherence by using future updates.
`RESET-UPDATE`	Cancel usage of future updates.
`WRITE-UPDATE`	Write data with cache coherence by propagating data globally.
`WRITE-UPDATE-PRIORITY`	Write data with cache coherence via Priority OBR.
CP-Synch	Put CP-Synch in Write Buffer; send over Priority OBR; include number of `WRITE-UPDATE`s in the SE.

Table 5: The Read and Write command primitives for cache coherency with buffered consistency in OBee.

Chapter 5

Synchronization

This chapter describes the direct architectural support in OBee for three different types of synchronization primitives: locks, barriers, and combining F&OPs. Beehive defines one implementation of locks and no implementation for either barriers or combining F&OPs. In contrast, OBee defines two different implementations, based upon the OBR interconnection network itself, for all three synchronization constructs.

5.1 Locks

Locks are a mechanism by which the programmer can access shared data in a controlled and regulated fashion. There are two types of locks: *read* (or shared, denoted as *R/S*) and *write* (or exclusive, denoted as *W/E*). By definition, when a programmer uses a R/S lock, he is guaranteed that the data's value will not change for the duration of the lock. Similarly, by definition, when he uses a W/E lock, he is guaranteed that no one else can change the data's value. Thus, when the programmer abides by the convention of accessing shared data only via locks, he is assured that his parallel program will maintain the shared data in a consistent and correct manner. It is important to note that the hardware itself does not force the

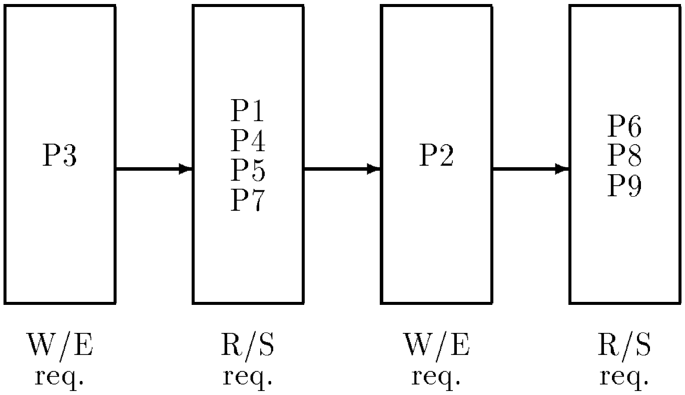

Figure 16: Lock request peer groups showing processors that have requested a particular type of lock.

programmer to follow this convention—the hardware only provides the primitives with the proper semantics to implement the convention. If the convention is to be followed, it must be at some other (higher) level, e.g., the programmer's own self-discipline or the language.

An important concept about locks is the *peer group*, as illustrated in Fig. 16. W/E lock requests, by definition, form a peer group of size one. R/S lock requests made without an intervening W/E lock request form one peer group. The reader might ask why the R/S requests in the fourth peer group in Fig. 16 are not coalesced into the second peer group. The answer is that in order to prevent starvation of W/E lock requests, when a W/E lock request is made, it marks the end of a R/S peer group. A R/S peer group may contain an arbitrary number of requests. From a performance standpoint, if a R/S peer group is the currently serviced peer group

Primitive	*Semantics*
`R/S-LOCK`	Request Read/Shared lock (implicitly on one cache line).
`W/E-LOCK`	Request Write/Exclusive lock (implicitly on one cache line).
`UNLOCK`	Release either type of lock.

Table 6: The lock and unlock command primitives in Beehive.

and no other peer groups are pending, then another R/S lock request can join the current R/S peer group and be granted the R/S lock immediately.

The final comment to make about locks is that when a W/E lock is released, then it must be ensured that the next holder of the lock (either R/S or W/E) must have received the updated values of any shared data protected by the lock before being granted the lock.

5.1.1 Locks in Beehive

The Beehive architecture defines locks in a form called *cache-based locks* [55]. The hardware supports an implicit lock for each cache line. When a lock request is issued and subsequently granted, the request and grant are actually implemented for a specific cache line. The consequence is that the grant of the lock is combined with the return of the data associated with the lock (assuming the data fits in one cache line).

Table 6 lists the Beehive lock (and unlock) primitives and their semantics. The `R/S-LOCK` and `W/E-LOCK` commands are requests for shared and exclusive locks respectively. The `UNLOCK` command releases either lock. The Beehive definition uses a distributed linked-list directory scheme (similar to its scheme for the reader

`usage-bit`	`head-of-list` or `tail-of-queue`

Figure 17: The memory line's directory entry (with cache based locks) in Beehive.

`update-bit`	$d_1 d_2 \ldots d_k$	`lock-status-bits`	`prev`	`next`

Figure 18: A cache line's directory entry (with cache based locks) in Beehive.

initiated cache coherence) to store and grant the requests. The basic structures of the memory line directory entry (shown in Fig. 13) and a cache line directory entry (shown in Fig. 14) are augmented as shown in Figs. 17 and 18 to accommodate maintaining the requests for a lock (remember that a lock is implicitly for an entire cache line, not a specific memory location).

The lock commands in Beehive use these structures in the following manner. The `usage-bit` in the memory line directory entry is used to track whether the memory line is participating in the reader initiated coherence or a lock request. The second field's interpretation depends upon the value of the `usage-bit`. If the line is participating in the reader initiated coherence, then the second field is used as a `head-of-list` pointer as described in Section 4.1. If the line is participating in a lock request, then the second field is used as a a `tail-of-queue` pointer. A cache line directory entry has the added field of `lock-status-bits`, which field stores two pieces of information: whether the cache line is locked and which type of lock has been requested. When a cache requests of memory that it be granted a lock on a line, the memory directory entry, since it points to the tail of the queue of lock requests for that line, supplies the information needed to add the cache to the request queue. The former tail of the queue is given a message to update its `next` pointer in order to maintain the linked list of requests. When a cache releases a R/S lock, since, by definition, no changes could have been made to the cache line, the memory line does not need to be updated. The cache can immediately pass the unlock message to the next cache in the chain and remove itself from the request queue. When the cache releases a W/E lock, then, assuming changes were made to the cache line, the cache updates the memory line (if no changes were made, then the cache can proceed just as if it had a R/S lock). Because the unlock and BC memory model are combined and because the unlock is a CP-Synch point, the releasing cache must wait for an acknowledgement from memory that it has received the update. The releasing cache does not need to have the update propagate to other caches since they will, in turn, receive any changes with the lock grant. After receiving the acknowledgement from memory, the cache transmits the updated cache line to the next requester in the chain, which message serves as the unlock message, and then removes itself from the chain.

The final detail concerning Beehive's locks regards shared data that do not fit into one cache line but which need to be protected by locks. In this case, the implicit cache line lock is used to protect *all* the shared data. The implicit cache line lock directly protects only the portion of the data which fits in the one cache line; the rest of the data, spread out over other cache lines, is indirectly protected as long as the one implicit cache line lock is held. Requests for R/S and W/E locks are as before: make the request and be put into the request queue. When a R/S lock is released, the cache's actions are as described before: pass the lock to the next requester in the queue. However, before a W/E lock can be released, the

`FLUSH-BUFFER` command must be issued so that all the `WRITE-UPDATE`s pertaining to all the shared data spread over multiple cache lines are performed; after the `FLUSH-BUFFER` completes, then the W/E lock itself may be passed to the next requester in the chain.

Therefore, if the locked shared data fits in one cache line, the Beehive locking mechanism allows the coherency of the shared data to be maintained at the same time that the lock is passed from one holder to the next requester. If the locked shared data spans several cache lines, then the cache coherency linked list mechanism described in Section 4.1 must be used to keep the shared data coherent before the lock is released and passed to the next requester.

5.1.2 Locks in OBee

There are two different methods in OBee for using the OBR interconnection network to implement R/S and W/E locks. One method uses the wired-`OR` OBR, and the other method uses the explicit message OBR. The former method we designate as the *purely optical* scheme, and the latter method we call the *hybrid optical/electronic* scheme. We give below the details of these two schemes and then present some comparisons.

5.1.2.1 Purely Optical Scheme

The purely optical scheme is so named because it uses only the presence and absence of optical signals to implement the lock semantics—there are no explicit bit messages used to represent the locks. Thus, this scheme naturally employs the wired-`OR` OBR architecture. Two wired-`OR` OBRs are assigned per logical lock. One of the OBRs is used to maintain R/S lock requests; the other one is used for W/E lock requests. As the reader will recall from Section 3.3.1, there are two different access protocols available on the wired-`OR` OBR topology. The wired-`OR` OBR for the R/S lock requests uses the first access method: when the medium is accessible, no arbitration among the nodes is necessary. The wired-`OR` OBR for the W/E lock requests uses the second method: the ordered access method in which the arbitration among nodes for access to the medium depends upon their physical ordering on the OBR.

With the appropriate combination of these two wired-`OR` OBRs, queues of R/S and W/E lock requests can be built. Since the two OBRs must work in tandem to implement the R/S and W/E lock semantics, the two OBRs must physically interact; they provide signals to each other as shown in Fig. 19. Note that this physical topology is modified in three ways from Fig. 8 in order to account for the necessary physical interactions between the two OBRs. One modification is that the R/S OBR receives its input "U" signal on one channel and puts its output "T" signal on another. The second modification that the W/E OBR puts its output "T" on two channels simultaneously. The third modification is that the W/E OBR Pulse Generator receives two input signals "I" and "B" and that it transmits its pulse simultaneously into two channels. Although this configuration seems quite different from that in Fig. 8, the underlying purposes of the signals are the same. The decisions which each node uses in determining when it makes a particular lock request, receives the lock, and releases the lock are shown in Fig. 20; the rules each of the two Pulse Generators follows are also given.

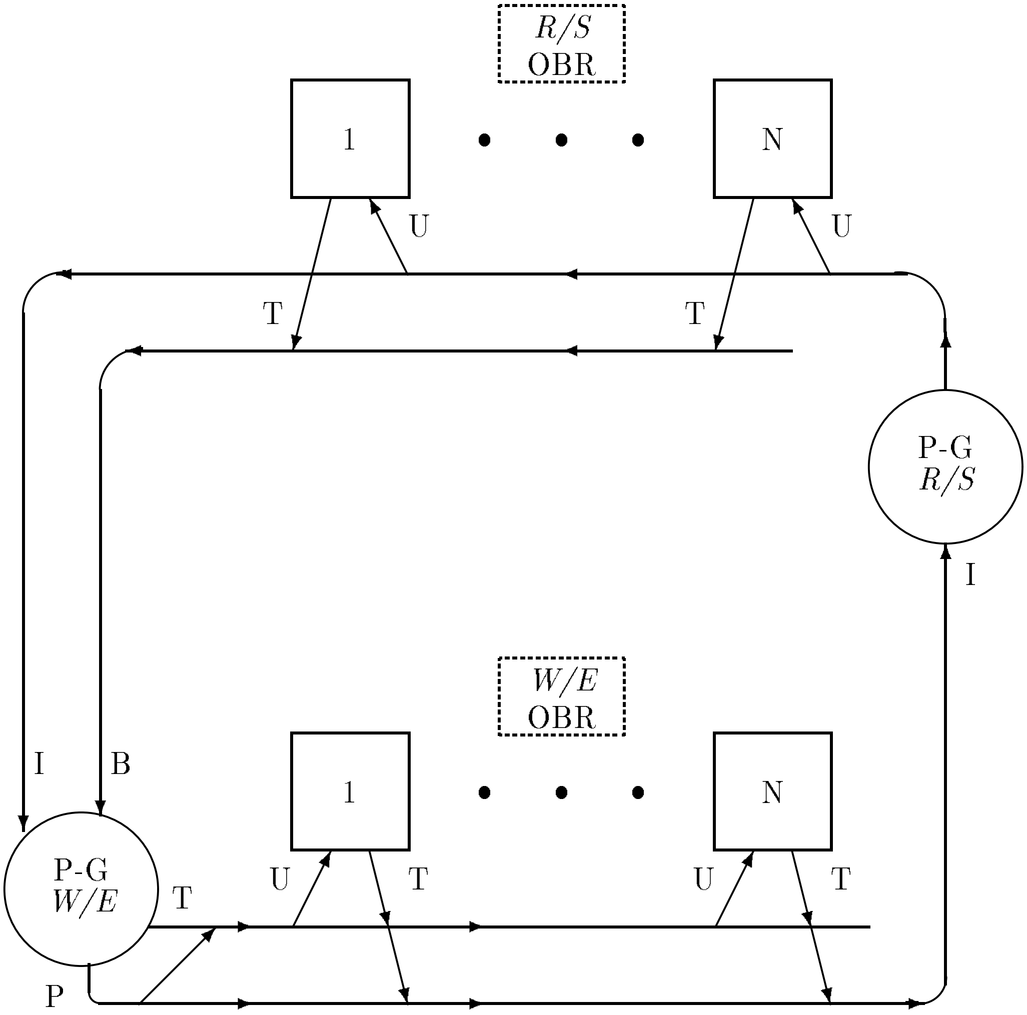

Figure 19: Two wired-`OR` OBRs are physically interconnected in a purely optical scheme to support W/E and R/S lock requests.

W/E Request:	$\Uparrow$W/E–T	WHEN	W/E–U
W/E Grant:	Granted lock	WHEN	$\overline{\text{W/E–U}}$
W/E Release:	$\Downarrow$W/E–T		
R/S Request:	$\Uparrow$R/S–T	WHEN	R/S–U
R/S Grant:	Granted lock	WHEN	$\overline{\text{R/S–U}}$
R/S Release:	$\Downarrow$R/S–T		
$P\text{-}G_{W/E}$ rules:	$\left\{\begin{array}{l}\text{generate Pulse;}\\ \texttt{if B : } \Uparrow\text{T}\texttt{ , else: } \Downarrow\texttt{T.}\end{array}\right\}$	WHEN	I$\downarrow$
$P\text{-}G_{R/S}$ rules:	generate Pulse	WHEN	I$\downarrow$

Figure 20: The algorithms the nodes and Pulse Generators follow for using locks in the purely optical OBee implementation.

To understand the entire sequence of actions on the two wired-`OR` OBRs, consider an example cycle in which there are both W/E and R/S requests. We initially assume that there are no requests for the lock, and we start with the W/E Pulse Generator sending out its pulse on its output "P" and "T" taps. Because the Pulse Generator is receiving nothing on its input "B" tap, it generates no extra signal on the "T" tap. As the pulse travels down the "P" channel, nodes wanting to make a W/E request are triggered to raise their output "T" signals when they see their input "U" signal go high. Since only a short pulse was put on the "P" channel, the first upstream node to request the W/E lock will immediately be granted the lock. When a node holds the lock, it keeps the "T" output high; when the node releases the lock, it lowers its "T" output, which then signals the next downstream requester that it has been granted the lock. This sequence of signals is the second ordered access method described in Section 3.3.1.

Meanwhile, the pulse (either with its original length or in lengthened form because of raised "T" signals by W/E requesters) has traveled to the R/S Pulse Generator. When the last downstream W/E requester has released the lock, the R/S Pulse Generator will be triggered to send out a pulse. This pulse arrives at the "U" input taps of the R/S OBR and causes any nodes wanting the R/S lock to raise their "T" signals. As soon as this pulse has gone by a node, the node immediately gains the lock and shares it with any other nodes that requested it in this R/S OBR cycle. This pulse (the output of the R/S Pulse Generator) then travels to the "I" input of the W/E Pulse Generator. Note that the "B" input of the W/E Pulse Generator is the `OR` of the nodes requesting and holding the R/S lock. By the rules of Fig. 20, the W/E Pulse Generator will first generate another pulse on the "P" and "T" output channels; next, because the "B" input is high (meaning that nodes hold the R/S lock), it will raise the "T" signal and keep it high. The pulse allows nodes to make their W/E requests, but because they will see their "U" input high, they cannot gain access to the lock. Each time the W/E Pulse Generator sees its "I" input fall, it generates another pulse and checks to see whether it can lower its "T" output. When it finally can lower its "T" output, this action will cause the W/E lock to be granted to the first upstream W/E requester.

The just described example covered a cycle in which both W/E and R/S requests are made. Other combinations of requests are possible. Suppose neither type of request is made. The W/E Pulse Generator's pulse travels through its OBR without triggering any "T" signals and arrives at the R/S Pulse Generator. The R/S Pulse Generator's pulse then travels through its OBR and arrives at the

W/E Pulse Generator, without having triggered any “T” signals, to start the cycle again. Suppose that only W/E requests are made. The W/E Pulse Generator’s pulse travels through its OBR, triggering “T” signals; after the requesters are serviced in physical order, the R/S Pulse Generator’s “I” input falls low. The R/S Pulse Generator’s pulse will travel through its OBR without triggering any “T” signals and then arrive at the W/E Pulse Generator to start the cycle again. A similar behavior occurs if only R/S requests are made, but the rules and physical topology of the OBRs allow additional R/S requests to be granted after the initial batch of requests if no W/E requests are meanwhile pending. When the R/S Pulse Generator’s pulse reaches the “I” input of the W/E Pulse Generator, it produces a pulse as before and, because its “B” input is high, raises its “T” signal. Since W/E “T” signals are triggered, then the R/S Pulse Generator, after receiving the pulse, generates its next pulse. The R/S Pulse Generator’s pulse then allows additional R/S requests, via the “U” input, to be granted. Meanwhile, because the W/E Pulse Generator’s “T” signal is high, W/E requests may be made, but they will not be granted until all the current R/S requests are satisfied. In order to prevent a race condition, the W/E Pulse Generator is allowed to lower its “T” signal only when it sees its “I” input fall (which corresponds to the tail end of a pulse) even if the “B” input falls first.

5.1.2.2 Hybrid Electronic/optical Scheme

The hybrid electronic/optical scheme for locks uses explicit messages (for both lock requests and releases) transmitted on one logical channel called the *Lock Channel* to implement the lock semantics. The electronic part comes from using the explicit messages to represent lock commands, and the optical part comes from using an explicit message OBR for the Lock Channel.

For the present discussion, we assume that only one explicit message OBR implements the Lock Channel, i.e., no matter how many different locks exist in the program, *all* traffic for *all* locks is carried on the one explicit message OBR. The explicit message OBR used for locks has exactly the same control and transmission topologies shown in Figs. 10 and 11; it also uses the same access rules enumerated in Section 3.3.2.

The essential idea behind the hybrid scheme is that each node continuously monitors the Lock Channel for lock command traffic. When a node puts a lock request message on the Lock Channel, from the previous traffic, it can determine where in the request queue it should be placed (including in what peer group). After making the lock request, the node, from subsequent traffic, can deduce when it is granted the lock. When the node releases the lock, it puts a message to that effect on the Lock Channel; that message is processed by all nodes to determine which peer group will next be serviced. Note that this scheme makes use of the inherent ordering of messages transmitted on the explicit message OBR (this fact will become quite apparent as the details are developed below). To understand the details of the hybrid scheme we first look at the information which must be stored for each lock and the structure of lock commands as they are transmitted via the Lock Channel; then we examine the monitoring algorithms.

Fig. 21 shows the data structure needed for each lock under the hybrid scheme (for convenience in the following discussion, the various fields are grouped into four parts). This data structure is replicated at every node and modified

`LOCK-ID`	`my-type`	`my-id`	`my-status`

`prior-type`	`prior-id`	`prior-count`

`most-recent-type`	`most-recent-id`	`most-recent-count`

`next-recent-type`	`next-recent-id`	`next-recent-count`

Figure 21: The data structure associated with each lock for the hybrid electronic/optical scheme under OBee.

independently by every node. The first group contains the fields to identify the lock and define the peer group into which the lock request is placed. The `LOCK-ID` field stores a unique identifier for the lock and is the same across all nodes. The `my-type` field stores what kind of lock request is made (R/S, W/E, or NONE). The `my-id` keeps a unique identifier to track the peer group into which the lock request is placed. The `my-status` field tracks whether the request is PENDING or GRANTED. The second group of fields defines the peer group which will be serviced immediately before the lock request's own peer group. The `prior-type` field stores what kind of request the prior peer group is: R/S, W/E, and NONE. The value of NONE indicates that the prior peer group is finished or non-existent. The `prior-id` field contains the unique identifier for the prior peer group. The `prior-count` stores how many requests are in the prior peer group. The `most-recent` and `next-recent` groups have the same fields as the `prior` group and serve a similar purpose. The `most-recent` group defines the most recent peer group to appear on the Lock Channel, and the `next-recent` group defines the peer group that appeared before the most recent. The purpose of these groups is to allow each node, as it observes the lock and unlock commands on the Lock Channel, to know into what peer group its own request should go and when its request should

`SENDER`	`lock-id`	`command-type`	`lock-type`	`REQUEST-id`

Figure 22: The format of lock command messages transmitted over the Lock Channel in the hybrid electronic/optical lock scheme in Obee.

be granted. The fields in each group are modified according to the algorithms presented in Figs. 23, 24, 25, and 26, which will be explained below.

The structure of the lock command messages transmitted over the Lock Channel is shown in Fig. 22. The `SENDER` field is the ID of the node sending the lock command message. The `lock-id` field identifies the lock to which the lock command is applicable. The `command-type` field denotes whether the command is a LOCK or UNLOCK. The `lock-type` field says whether the lock is a W/E or R/S. The meaning of the `REQUEST-id` field depends upon the `command-type` field. If the lock command is of type LOCK, then the `REQUEST-id` field is a unique identifier formed by the node for that lock request; its formation and use are explained in the following discussion on the lock command message monitoring algorithms. If the lock command is of type UNLOCK, then the `REQUEST-id` field identifies the peer group in which the unlock command belongs.

Before explaining specific parts of the monitoring algorithms, we note the two underlying principles for each algorithm. First, each node monitors its "B" (broadcast) input tap. Therefore, each node sees all the traffic both preceding and succeeding its own lock commands. Second, as a reminder, each node sees the same sequence of lock commands as every other node. Therefore, even though each node is independently processing the Lock Channel traffic, there will be no inconsistencies among the nodes in deciding what requests form which peer groups

```
if (COMMAND ==  REQUEST):
  if (SENDER !=  SELF):  //  Just update  most-recent and
                             next-recent peer groups.
    switch REQUEST_TYPE:
     case  W/E: //  Automatically starts new Peer Group.
        next-recent-type   := most-recent-type;
        next-recent-id     := most-recent-id;
        next-recent-count  := most-recent-count;

        most-recent-type   :=  W/E;
        most-recent-id     := REQUEST-id;
        most-recent-count  := 1;

        break;

     case  R/S:
        if (most-recent-type == ( NONE .OR.  W/E)):
           //  Starts a new Peer Group.
           next-recent-type   := most-recent-type;
           next-recent-id     := next-recent-id;
           next-recent-count  := next-recent-count;

           most-recent-type   :=  R/S;
           most-recent-id     := REQUEST-id;
           most-recent-count  := 1;

        else:  //  Since  most-recent Peer Group is R/S, this request
                   joins it.
           most-recent-count++;
           endif.

        break;
     endswitch.
    endif.
  endif.
```

Figure 23: The algorithm which nodes follow for processing Lock Requests from other nodes under the hybrid electronic/optical scheme in OBee.

```
if (COMMAND == REQUEST):
  if (SENDER == SELF):  // Must define my and prior
                           peer groups, as well as update most-recent
                           and next-recent peer groups.
    switch REQUEST_TYPE:
     case W/E: // Automatically starts new Peer Group.
        // First create prior group from most-recent group:
        prior-type   := most-recent-type;
        prior-id     := most-recent-id;
        prior-count  := most-recent-count;

        // Update next-recent and most-recent groups:
        next-recent-type   := most-recent-type;
        next-recent-id     := most-recent-id;
        next-recent-count  := most-recent-count;

        most-recent-type   := W/E;
        most-recent-id     := REQUEST-id;
        most-recent-count  := 1;

        // Create my group:
        my-type  := most-recent-type;
        my-id    := most-recent-id;

        if (prior-type == NONE): my-status := GRANTED;
        else:                    my-status := PENDING; endif.

        break;
```

First part (of three) of Fig. 24.

Figure 24: The algorithm which nodes follow for processing Lock Requests from themselves under the hybrid electronic/optical scheme in OBee.

```
case R/S:
   if (most-recent-type == ( NONE .OR. W/E):
     // Form new Peer Group:
     // First create prior group from most-recent:
     prior-type    := most-recent-type;
     prior-id      := most-recent-id;
     prior-count   := most-recent-count;

     // Update next-recent and most-recent groups:
     next-recent-type    := most-recent-type;
     next-recent-id      := most-recent-id;
     next-recent-count   := most-recent-count;

     most-recent-type    := R/S;
     most-recent-id      := REQUEST-id;
     most-recent-count   := 1;

     // Create my group:
     my-type  := most-recent-type;
     my-id    := most-recent-id;

     if (prior-type == NONE): my-status := GRANTED;
     else:                    my-status := PENDING; endif.
```

Second part (of three) of Fig. 24.

```
    else: //  Since  most recent Peer Group is R/S, this request
          //  joins it.
          //  First create  my group:
      prior-type   := next-recent-type;
      prior-id     := next-recent-id;
      prior-count  := next-recent-count;

      //  Update  most recent group:
      most-recent-count++;

      //  Create  my group:
      my-type := most-recent-type;
      my-id   := most-recent-id;

      if (prior-type ==  NONE): my-status :=  GRANTED;
      else:                     my-status :=  PENDING; endif.

    endif.
   break;
  endswitch.
 endif.
endif.
```

Third part (of three) of Fig. 24.

```
if (COMMAND ==  W/E-UNLOCK):
 if (REQUEST-id == next-recent-id): next-recent-type :=  NONE; endif.

 if (REQUEST-id == most-recent-id): most-recent-type :=  NONE; endif.

 if (REQUEST-id == prior-id):
   prior-type :=  NONE;
   if (my-type == ( W/E .OR.  R/S):
     my-status :=  GRANTED;
     endif.
   endif.

 if (REQUEST-id == my-id): //  Change  my-type to indicate that node
                           //  no longer holds this lock—there is no
                           //   my-count field to update.
   my-type : =  NONE;
   endif.

 endif.
```

Figure 25: The algorithm which nodes follow for processing W/E UNLOCK commands under the hybrid electronic/optical scheme in OBee.

```
if (COMMAND ==  R/S-UNLOCK):
  if (REQUEST-id == next-recent-id):
    next-recent-count--;
    if (next-recent-count == 0): next-recent-type :=  NONE; endif.
    endif.

  if (REQUEST-id == most-recent-id):
    most-recent-count--;
    if (most-recent-count == 0): most-recent-type :=  NONE; endif.
    endif.

  if (REQUEST-id == prior-id):
    prior-count--;
    if (prior-count == 0):
      prior-type :=  NONE;
      if (my-type == ( W/E .OR.  R/S): my_status :=  GRANTED; endif.
      endif.
    endif.

  if (REQUEST-id == my-id): //  Change  my-type to indicate that node
                            //  no longer holds this lock—there is no
                            //   my-count field to update.
    my-type : =  NONE;
    endif.

  endif.
```

Figure 26: The algorithm which nodes follow for processing R/S UNLOCK commands under the hybrid electronic/optical scheme in OBee.

and which peer groups are serviced when. As a final comment about the monitoring algorithms, we point out that it is implicit in each algorithm's specification that the data structure to be modified is the one specified by the `lock-id` field in the lock command message.

We now examine each of the monitoring algorithms. The monitoring algorithm for a node to process lock requests from all nodes except itself is shown in Fig. 23. The essential idea behind this algorithm is that when a node sees either type of lock request from a node (except for itself), the node needs to update the `most-recent` and `next-recent` peer group information. Specifically, if the lock request is of type W/E, then a new peer group is automatically formed; thus, the `next-recent` peer group becomes the former `most-recent` peer group, and the `most-recent` peer group is this new W/E lock request. Note that the `most-recent-id` field is set by reading the value of `REQUEST-id`, a variable which was sent with the lock request command on the Lock Channel. The `REQUEST-id` is formed by the lock requester and consists of two fields, the requesting node's own identifier and a simple counter maintained by the node. Every time the node makes a new lock request, it obtains the counter's value and then increments the counter for the next use. Since each node, via this algorithm, pulls the `REQUEST-id` out of the lock request message, then a given peer group will have the same identifier across all nodes. If the lock request is of type R/S, there are two possibilities. First, the `most-recent` peer group is either non-existent (meaning the lock is not being held) or of type W/E; if this is the case, then the new R/S request must form a new peer group. If the `most-recent` peer group is of type R/S, then the new R/S request can join the existing `most-recent` peer group, which is represented by updating the `most-recent-count` field. Note that if the R/S request joins the `most-recent` peer group, the `REQUEST-id` part of the R/S request's message is ignored. In the case of R/S peer groups of size more than one, the peer group's ID is only the first R/S request's ID.

The monitoring algorithm shown in Fig. 24 is for nodes processing their own lock request messages. A separate algorithm from the previous monitoring algorithm is needed because a node, when processing its own request, must not only update the `next-recent` and `most-recent` peer groups, but it must also record the `prior` peer group information, record its request's peer group information (in the `my` fields), and determine whether it can immediately acquire the lock. First consider if the lock request is of type W/E. As in the previous monitoring algorithm, a W/E request automatically forms a new peer group. Before the `next-recent` and `most-recent` peer group information fields are updated however, what was the `most-recent` peer group becomes the `prior` peer group. Then, the `next-recent` and `most-recent` peer group information fields are updated. Finally, the `my` fields are formed; the node determines whether it immediately acquires the lock by looking at the `prior-type` field to see if the immediately prior lock does not exist. Next, consider if the node is processing its own R/S lock request. This case is broken into two subcases. The first subcase occurs when the `most-recent-type` peer group is either non-existent or a W/E request, in which case the R/S request forms a new peer group. The `prior` peer group information is formed from the `most-recent` peer group information, the `next-recent` and `most-recent` peer group information is updated, and then the node's own request is stored in the `my` peer group. The R/S request can immediately be granted to the node if the `prior-type` field shows that the lock was not being held prior to this request. The second subcase occurs when the `most-recent` peer group is also a R/S request, in which case this R/S request joins the `most-recent` peer group. Note that, unlike the previous parts of the algorithm, the `prior` peer group information is formed from the `next-recent` peer group. Note also that the `REQUEST-id` field of the lock request message is ignored since the ID of the peer group comes from the first R/S request in the peer group. Finally, the R/S request can immediately be granted to the node if the `prior-type` field shows the lock was not being held previously.

In contrast, the monitoring algorithms for the both the W/E and R/S unlock commands are fairly simple. The monitoring algorithm for the W/E unlock command is shown in Fig. 25. If the `REQUEST-id` field of the command message matches either the `next-recent` or `most-recent` peer groups observed by the node, then since a W/E peer group is size one, that peer group is finished and is changed to type NONE. If the `REQUEST-id` field of the command message matches the `prior` peer group, that peer group is also finished and changed to type NONE; in addition, if the node currently has a lock request as indicated by the `my-type` field, it acquires its lock. If the `REQUEST-id` field of the command message matches the node's own lock, then the `my-type` field is changed to NONE to indicate the node has finished with its own request. The monitoring algorithm for the R/S unlock command, shown in Fig. 26, is nearly identical to the W/E unlock monitoring algorithm except that since R/S peer groups can be larger than size one, an additional test must be made to determine if the particular peer group being examined is finished.

5.1.2.3 Discussion

We first point out how the purely optical and hybrid electronic/optical OBee lock schemes employ the capabilities of the OBR. Both schemes rely upon the inherent ordering capability of the access protocol. The queues of the lock requests come directly about when the nodes access the OBR. Recall that in the purely optical scheme, a node releases the lock to next downstream requester; thus, the passing of the lock takes the minimum amount of time since the requests have been queued in the order by which the nodes are physically attached to the OBR. The purely optical method uses the broadcast sum capability of the wired-`OR` OBR to signify when R/S requests may join the current R/S peer group. Besides using the inherent ordering of requests, the hybrid scheme uses the broadcast capability of the explicit message OBR so that nodes rely on counting the unlock messages to determine lock acquisition rather than needing an explicit lock grant. An explicit grant is undesirable for releasing R/S lock requests since, if the peer group is larger than one, extra work must be performed to determine which requester in the R/S peer group takes responsibility for releasing the lock to the next peer group. An explicit grant is also undesirable when a W/E lock holder releases the lock to a R/S peer group since the latter can be arbitrarily large, and it would be unnecessary bookkeeping for the W/E lock holder to track the members of the R/S peer group.

We next compare the OBee lock implementations to the Beehive lock implementation in several ways. The first comparison is between peer groups in OBee and peer groups in Beehive. The peer groups in the purely optical OBee implementation are organized as shown in Fig. 27 as opposed to Beehive's arbitrary peer

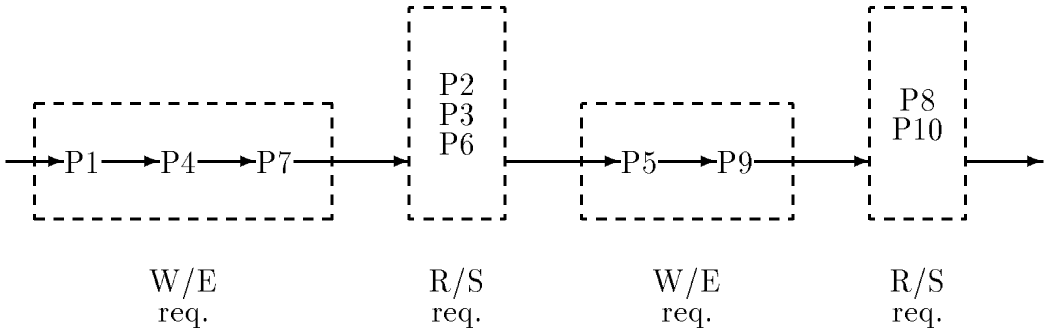

Figure 27: The organization of peer groups in the purely optical OBee lock implementation.

groups of Fig. 16. For a given cycle on the two interconnected wired-`OR` OBRs, W/E requests are batched in one group, and R/S requests are batched in another group. The W/E requests (if any) are then serviced according to the physical order by which the nodes are attached to the OBR (from node 1 to node N). Then, the R/S requests are serviced together (note, though, that the R/S requesting nodes are not notified of the R/S lock grant simultaneously; the notification proceeds from node N to node 1). Any W/E or R/S request which misses a given OBR cycle must wait for the next cycle. Therefore, the purely optical OBee implementation does not form and service lock requests in the strictly FIFO manner that Beehive does. However, the purely optical OBee implementation does not starve either type of lock requests; the two types of lock requests are just batched separately and, in the case of W/E requests, are re-ordered. From the programming standpoint, this difference in implementation does not affect program correctness if the program was written using the BC memory model correctly. To complete this peer group comparison, we make two final observations. One, the peer groups in the hybrid electronic/optical OBee lock implementation are formed and serviced identically to peer groups in Beehive. Two, for both the purely optical and hybrid schemes, if the current lock request is of type R/S and if the most recently formed peer group is of type R/S, then the current R/S lock request is allowed to join the most recently formed peer group, just as in Beehive.

The second comparison between the OBee schemes and the Beehive implementation concerns the combining of the lock grant with the lock's associated data. As described thus far, the two OBee schemes do not associate any data (e.g., a cache line) with the lock. Rather, use of the OBee locks forces the cache coherency to be maintained explicitly since the lock acquisition is not integrated with the cache coherency. We will discuss first how the purely optical OBee lock can be integrated with the cache coherence maintenance, then discuss the issues related to the hybrid scheme. Recall that the purely optical scheme relies upon "high" or "low" optical signals, i.e., the signals themselves do not carry any information—their meaning comes from the physical geometry of the wired-`OR` OBRs. Thus, the wired-`OR` OBRs themselves cannot carry any additional information such as a `WRITE-UPDATE` when a W/E lock is released. In order to send these `WRITE-UPDATE`s, an additional data transmission channel is necessary (call it the W/E `WRITE-UPDATE` channel). To maintain the BC memory model, the `WRITE-UPDATE` must be received and processed by the next requester of the lock *before* the W/E lock release is recognized by that requester. Thus, the wired-`OR` OBR representing the W/E lock requests and the W/E `WRITE-UPDATE` channel must be coordinated via additional decision logic so that when a node receives the W/E lock release on the wired-`OR` OBR, it does not consider itself to have acquired the lock until it has received and processed the

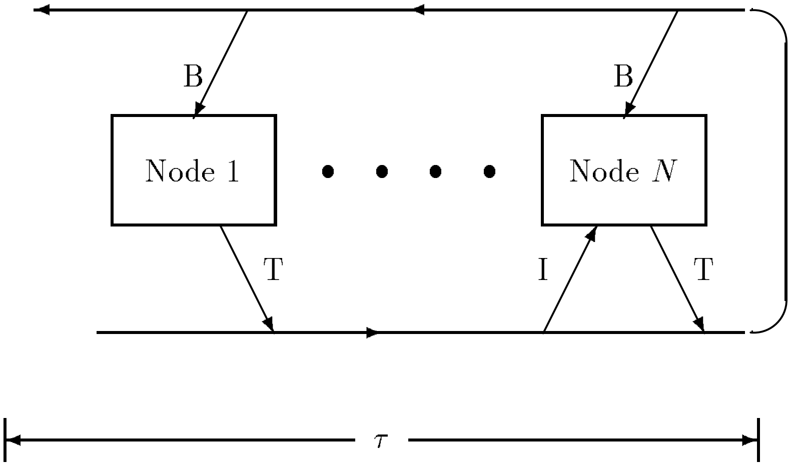

Figure 28: The transmission topology of the additional W/E `WRITE-UPDATE` channel extension to the purely optical OBee lock scheme.

`WRITE-UPDATE` messages on the W/E `WRITE-UPDATE` channel. One implication of this addition to the purely optical scheme is that if the W/E `WRITE-UPDATE` channel uses a dedicated transmission channel (an example of the themes of pre-assigning a channel's function and not worrying about a low utilization of a channel), then when nodes use this channel, they do not need to use any medium access protocol to arbitrate for this channel since holding the W/E lock is, in effect, controlling access to the W/E `WRITE-UPDATE` channel. Thus, the W/E `WRITE-UPDATE` channel has the transmission topology shown in Fig. 28. A second implication of this addition is that data structures of arbitrary size may be associated with the lock since messages on the W/E `WRITE-UPDATE` channel can contain an arbitrary number of `WRITE-UPDATE` commands. A third implication is that when a node releases a W/E lock and has transmitted the associated `WRITE-UPDATE`s, it does not have to wait for an acknowledgement from the memory modules storing the permanent copies

`SENDER`	`lock-id`	`command-type`	`lock-type`	`REQUEST-id`	`WRITE-UPDATE`s

Figure 29: The format of lock command messages that integrate cache coherency as transmitted over the Lock Channel in the extension to the OBee hybrid electronic/optical lock scheme.

of the variables. Recall from Section 5.1.1 that in Beehive, the cache must get an acknowledgement from memory because of the arbitrary delivery order characteristic of the Beehive interconnection network. In this OBee implementation, when Node i sends a `WRITE-UPDATE`, then the memory module is guaranteed to receive later any `WRITE-UPDATE`s which are sent later. Finally, we must consider that if data are associated with the purely optical locks, then, for both R/S and W/E requests, if the data is not present in the node's cache, additional decision logic is necessary between the wired-`OR` OBR and the cache so that the lock is not actually granted until the data has been obtained.

The hybrid electronic/optical OBee lock scheme is easily extended to incorporate the cache coherence maintenance with the acquisition of locks. The structure of lock commands transmitted over the Lock Channel can be modified as shown in Fig. 29 to include the appropriate `WRITE-UPDATE` commands. Note that the `lock-id` field now represents the appropriate cache line. Since the lock release and the `WRITE-UPDATE`s are integrated into one message and sent over one logical channel, then unlike the extension to the purely optical scheme described above, no coordination between the lock release and the `WRITE-UPDATE`s is needed except that a node must apply the `WRITE-UPDATE`s before recognizing the lock release.

As with the purely optical extension, since the lock command can contain an arbitrary number of `WRITE-UPDATE`s, then a data structure of arbitrary size can be associated with the lock. Also, as with the purely optical extension, the node does not need to wait for an acknowledgement from memory regarding the propagation of the `WRITE-UPDATE`s. The problem of the associated data not being present in the cache when the lock request is made can be dealt with by having both R/S and W/E unlock commands include the data in the transmitted message. An additional slight problem is if a node makes a R/S request, and if the most recent peer group is R/S and is currently being serviced, then if the requesting node does not have the data in the cache, it would have to stall until it could obtain the data and then reissue the R/S request.

The third comparison of the OBee scheme with the Beehive implementation concerns the hardware requirements, which are summarized in Table 7. The Beehive implementation requires one transmission network, overhead fields per cache line (since a lock is implicitly associated with a cache line), and moderately complex logic to maintain the queues of lock requests. The purely optical OBee scheme requires two wired-`OR` OBRs per lock in the program (the number of required pairs of OBRs can be reduced by reassigning pairs to different locks dynamically as the program progresses rather than assigning statically one pair per lock). This scheme also requires simple decision logic to interpret when optical signals are present and absent to maintain the queues. The hybrid OBee scheme requires one explicit message OBR to carry all the traffic for all the lock commands. Each lock requires storage for the various peer group fields used to maintain the queues. Each node needs the logic necessary to interpret the explicit bit messages on the

Lock Scheme	*Requirements*
Beehive	1 transmission network overhead queue fields/cache line moderately complex logic
OBee–purely optical	2 wired-`OR` OBRs/lock simple decision logic
OBee–hybrid	1 Lock Channel (explicit message OBR) (peer group storage fields)/lock message interpretation logic
OBee–purely optical with cache coherence	(2 wired-`OR` OBRs + `WRITE-UPDATE` channel)/lock simple decision logic
OBee–hybrid with cache coherence	1 Lock Channel (explicit message OBR) (peer group storage fields)/cache line message interpretation

Table 7: Comparison of hardware requirements among the Beehive and OBee lock schemes.

Lock Channel which represent the lock commands and then apply the monitoring algorithms. When integrated cache coherence is added to the purely optical OBee scheme, the additional W/E `WRITE-UPDATE` transmission channel per lock is needed. Additional logic is also needed to coordinate the wired-`OR` OBRs and the W/E `WRITE-UPDATE` channel. Since the `WRITE-UPDATE`s must be applied regardless of whether the cache coherency is integrated with the locks, no additional logic is needed in the integrated scheme. When integrated cache coherence is added to the hybrid OBee lock scheme, the storage requirement for the various peer group fields increases considerably since the requirement is per cache line rather than per lock. The message interpretation logic is the same as without the cache coherency integration since the logic to apply the `WRITE-UPDATE`s existed even without the

Lock Scheme	`W/E-LOCK`	`W/E-UNLOCK`	`R/S-LOCK`	`R/S-UNLOCK`
Beehive	1 or 2	$2+R$	1 or 2	$3U+R$
OBee	1	1	1	U

Table 8: The number of messages for the various lock commands under the Beehive and OBee lock implementations.

integration. Finally, we note that if the traffic on the hybrid Lock Channels warrants, several Lock Channels can be used to service lock commands. The use of several Lock Channels implies a replication of the message interpretation hardware and also implies, for correct semantics, that all the traffic for any given lock (or cache line with lock) be carried on one Lock Channel only.

The final comparison between the Beehive and OBee lock schemes concerns the number of messages that must be sent for each lock command; this comparison is shown in Table 8. A message in the OBee purely optical scheme corresponds to either the raising or lowering of the optical signal by a node. A message in Beehive or the OBee hybrid scheme refers to the sending of an explicit group of bits. In this comparison we make the assumption that when the lock is requested, the data associated with it already exists in the cache so that the message counts reflect only those pertaining to the locks themselves. In Beehive, either a W/E lock request or R/S lock request takes one message if no other requests are outstanding or two if there are some requests. The W/E unlock takes $2+R$ messages; two messages result from updating memory and receiving an acknowledgement; R messages result from sending the unlock to the next peer group of size R. The R/S unlock (for a peer group of size U) takes approximately $3U+R$ messages. Each node holding the

R/S lock usually takes 3 messages to remove itself from the peer group list (the exact number depends upon just where in the list the node is). It takes R messages to notify the members of the next peer group of size R that they have acquired the lock. For the OBee implementations (both purely optical and hybrid), both the W/E and R/S lock requests take only one message. The W/E unlock takes one message, and the R/S unlock takes U messages, where U is the size of the peer group releasing the lock. Note that the number of messages in the OBee implementation is the minimum number necessary for each operation.

5.2 Barriers

The *barrier* is another useful synchronization construct for parallel programming and was first proposed by Jordan [43]. A barrier is a rendezvous point for some set of processes. The semantics of a barrier are that after initialization of the barrier, the independent nodes participating in the barrier are free to reach the barrier at their own pace. When a node has reached the barrier, it is said to be in the *arrival* phase. After all the participating nodes have arrived at the barrier, then *barrier completion* is said to have occurred. The last phase of the barrier operation is called *notification*, in which all the participating nodes recognize barrier completion and are permitted to proceed past the barrier.

Table 9 shows the three command primitives used to implement the barrier semantics. The `INIT-BARRIER` command initializes the barrier before the participating nodes are allowed to rendezvous at it. A node issues the `REACH-BARRIER` command when it has arrived at the barrier. The `BARRIER-WAIT` command forces

Barrier Command	*Semantics*
`INIT-BARRIER`	Initialize barrier with the participating nodes.
`REACH-BARRIER`	Node has reached the barrier.
`BARRIER-WAIT`	Node waits until barrier completion.

Table 9: The Barrier command primitives available under OBee.

a node to wait until it recognizes barrier completion.

In the subsequent sections we describe two OBee implementations of the barrier construct, a purely optical and a hybrid electronic/optical scheme (recall that Beehive does not provide direct hardware support for the barrier).

5.2.1 Purely Optical Scheme

In the purely optical barrier scheme, as with purely optical locks, only the presence and absence of optical signals are used to implement the barrier semantics. The key ideas behind this representation are that: (1) when the barrier is initialized, the participating nodes raise a signal which is seen by all other nodes; (2) when a node reaches the barrier, it drops its signal; and (3) when a node sees no signals from any participating node, it recognizes barrier completion. Therefore, since this is an `OR` representation of the barrier, the wired-`OR` OBR architecture can be used to implement the barrier. One wired-`OR` OBR is assigned per barrier. The topology of the wired-`OR` OBR is exactly that of Fig. 8. Both access protocols enumerated in Section 3.3.1 are used depending upon which barrier command is being implemented.

The `INIT-BARRIER` command is implemented by using the second access protocol of Section 3.3.1, i.e., the Pulse Generator sends out a pulse to trigger, via the "U" input tap, the participating nodes to raise their "T" output taps. A node knows whether to participate in the barrier from the application program itself. For example, each node could have a bit register, which we designate a *participation barrier register* (PBR), associated with every wired-`OR` OBR. This register indicates whether the node participates in the associated barrier. We assume that when the program begins and before any barriers are initialized with the `INIT-BARRIER` command, each node's PBRs are appropriately set. Thus, the rule governing the raising of the "T" output is:

$$\Uparrow\mathrm{T} \quad \textsc{when} \quad (\mathrm{PBR}\ \texttt{.AND.}\ \mathrm{U}\uparrow).$$

After raising its "T" signal, a node keeps that signal high until it is ready to execute the `REACH-BARRIER` command:

$$\mathrm{T} \quad \textsc{until} \quad (\text{barrier is reached}).$$

The `REACH-BARRIER` command is implemented by the first access protocol described in Section 3.3.1, i.e., when a node reaches the barrier, it drops its "T" signal regardless of which other nodes have or have not dropped their "T" signals. Actually, the node cannot drop its "T" signal until it recognizes that the barrier has been initialized, which fact is indicated by the node's seeing a signal on its "B" input tap. Thus, the rule governing the dropping of the "T" signal (to indicate reaching the barrier) is:

$$\Downarrow\mathrm{T} \quad \textsc{after} \quad ((\text{reached barrier})\ \texttt{.AND.}\ \mathrm{B}).$$

Since a node sees at its "B" input tap the wired-`OR` of all the participating nodes' raised "T" signals, as long as any signal is detected at the "B" tap, the node knows that at least one node has not reached the barrier. Therefore, when a node issues the `BARRIER-WAIT` command, it stalls until it sees no signal on the "B" input tap:

$$\texttt{Node stalls}\ \ \textsc{until}\ \ \overline{\mathrm{B}}.$$

What happens when the last node to reach the barrier drops its "T" signal? Up until this point, the Pulse Generator, via its "I" input tap, has seen a continuously high signal, just as the individual nodes have seen such a high signal on their "B" input taps. The Pulse Generator can take one of two actions. First, it can ignore the dropping of its "I" signal and wait until the program issues an `INIT-BARRIER` command. The second action is for the Pulse Generator to issue another initializing pulse as soon as it sees the "I" signal drop. This action immediately reinitializes the barrier for its next use. We will discuss in Section 5.2.3 when which action is appropriate.

5.2.2 Hybrid Electronic/optical Scheme

The hybrid electronic/optical barrier scheme, in a fashion similar to the hybrid scheme for locks, uses explicit messages (for the appropriate barrier commands) transmitted over one logical channel called the *Barrier Channel*. The idea behind the hybrid scheme is that each node maintains a copy of a data structure which represents the barrier and uses the Barrier Channel traffic to update the barrier's status. The broadcast capability is needed since each node must receive all the

`ID`	`participation`	`counter`

Figure 30: The data structure which captures the state of a barrier in the hybrid electronic/optical scheme under OBee.

messages and independently track the barrier's status. This scheme naturally maps to one explicit message OBR over which *all* the traffic for *all* the barriers is carried. The explicit message OBR for barriers uses the same control and transmission topologies shown in Figs. 10 and 11 and uses the same access rules enumerated in Section 3.3.2.

To understand how the hybrid scheme works we first specify the data structure associated with each barrier and the format of messages broadcast over the Barrier Channel; then we examine the algorithms each node follows to implement the barrier commands. Fig. 30 depicts the data structure which captures the state of the barrier. Each node has a copy of this data structure, and each node modifies its copy independently of other nodes' actions. The barrier is uniquely identified by the `ID` field. The `participation` field stores how many nodes are participating in the barrier, and the `counter` field tracks how many nodes have reached the barrier.

The format of a message sent over the Barrier Channel is shown in Fig. 31. The `ID` field identifies which barrier is being addressed. The *command* field specifies what kind of barrier command should be performed. This field includes the operator and associated operand(s) to be performed on the lock's data structure (the contents and usage of this field will become clear when the monitoring algorithms are subsequently explained).

`ID`	*command*

Figure 31: The format of messages sent over the Barrier Channel in the hybrid electronic/optical scheme under OBee.

```
switch  command:
  case  INITIALIZE:
      participation := $N$;
      counter       := 0;
      break;

  case  REACH:
      counter++;
      break;

      endcase;
   endswitch.
```

Figure 32: The monitoring algorithm that each node follows in processing traffic on the Barrier Channel.

The monitoring algorithm that each node uses in processing the Barrier Channel traffic is shown in Fig. 32. When an `INIT-BARRIER` command message is received, the node uses the message's `ID` field to identify which data structure is being addressed. The proper barrier's `participation` field is set to the number of nodes participating in the barrier and the `counter` field to 0. When a node processes a `REACH-BARRIER` command (including its own), it increments the `counter` field of the barrier specified by the message's `ID` field.

After a node has issued the `REACH-BARRIER` command, it must perform a `BARRIER-WAIT` command, the algorithm for which is shown in Fig. 33. This algorithm is very simple: the node must stall until the number of nodes reaching

```
while (counter != participation): stall; endwhile;

counter := 0;
 resume program execution.
```

Figure 33: The algorithm a node follows to implement the `BARRIER-WAIT` command in the hybrid barrier scheme.

the barrier equals the number participating in the barrier. The hardware for processing Barrier Channel messages must be such that while the node waits for the `BARRIER-WAIT` command to complete, it can process Barrier Channel messages; otherwise it could not complete. After the barrier is reached by all participating nodes, the node resets its copy of the `counter` to reinitialize the barrier for its next use. Note that each node independently ascertains when barrier completion has occurred since each node counts for itself how many nodes have reached the barrier; therefore, there is no need for an explicit command to notify nodes of barrier completion.

5.2.3 Discussion

We first make some comments about how the purely optical and hybrid electronic/optical implementations of barriers employ the properties of the OBR architecture. The purely optical scheme uses the Pulse Generator (and its subsequent ordering of accesses to the OBR) only to control when the barrier is initialized before use. More importantly, since nodes are allowed to turn off their signals without regard to when other nodes do so, the hardware allows an arbitrary number of nodes to issue the `REACH-BARRIER` simultaneously. Since the wired-`OR` OBR broadcasts to every node the sum of the nodes' signals, no one node takes responsibility for asserting barrier completion. The hybrid scheme makes use of the broadcast capability of the explicit message OBR so that nodes independently determine barrier completion. The hybrid scheme also relies upon messages being received in transmission order to prevent a race condition from one use of the barrier to another (discussed in more detail below).

As mentioned earlier, there are two policies for initializing the purely optical barrier (recall that the initialization is performed by having the Pulse Generator send out a pulse). In the first policy, only when the `INIT-BARRIER` command is explicitly issued does the barrier become initialized. The second policy says that when the "I" input to the Pulse Generator drops, another pulse is sent out to initialize the barrier. If, at program initialization, the barrier hardware is statically associated with program barriers, then the second initialization policy is permissible and allows the barrier to be reused as soon as possible. The reader should realize that an explicit `INIT-BARRIER` command is required before the first use of the barrier; also, the participation barrier registers (PBRs) at each node need to be set appropriately. The Pulse Generator, by following the rule:

$$\text{send Pulse} \quad \textsc{when} \quad I\downarrow,$$

in effect recognizes barrier completion and initializes the barrier for its next use. No node takes responsibility for preparing the barrier for its next use. If, however, the hardware barriers are used to implement different program barriers, then a barrier cannot be initialized for reuse except by the `INIT-BARRIER` command.

This dynamic assignment of the barrier hardware also requires resetting the PBRs appropriately whenever the barrier hardware is reassigned to a different program barrier. Therefore, this policy requires that some node take responsibility for initializing the barrier hardware between each use.

The design of the barrier must ensure that no race conditions can result when the same structure is going to be reused immediately: every participating node must be able to recognize barrier completion before the hardware is initialized for the next use. We first discuss the purely optical scheme in the context of this design issue. If the policy of issuing the `INIT-BARRIER` command to initialize the barrier hardware each time before use is followed, then to prevent this race condition, the last node physically to recognize barrier completion must have the responsibility of issuing the `INIT-BARRIER` command. Following the notation of our previous figures (such as Fig. 8), this last node will be the lowest numbered one participating in the barrier. If the Pulse Generator automatically initializes the barrier hardware each time barrier completion occurs (by seeing its "I" input drop), then since it recognizes barrier completion after all the nodes, the race condition cannot occur. In the hybrid scheme (assuming that the data structures are statically assigned to the program barriers), the race condition is avoided because the explicit message OBR guarantees that messages transmitted later in time are received later in time. Therefore, a node is guaranteed to compute and recognize barrier completion before it resets the data structure for the next use and before it can receive `REACH-BARRIER` messages from other nodes that are already using the barrier again. If the barrier hardware data structures are dynamically assigned to program barriers, then the last node physically to recognize barrier completion must take responsibility for issuing the `INIT-BARRIER` command to reset the `participation` field when the barrier hardware is reassigned.

As described so far, the semantics of the barrier operations are that reaching the barrier and waiting for barrier completion are inseparable, i.e., when a node reaches the barrier, it can do nothing but wait for the other participating nodes also to reach the barrier. These semantics are represented in our schemes by issuing the `REACH-BARRIER` command immediately followed by the `BARRIER-WAIT` command. A potentially useful extension to the normal barrier semantics is that of the "fuzzy" barrier proposed by Gupta [37]. In the fuzzy barrier, after the node has reached the barrier, the node potentially has code which it can execute before waiting for barrier completion. Conceptually, the node issues the `REACH-BARRIER` command, executes any code which does not depend upon barrier completion, and then issues the `BARRIER-WAIT` command to wait for barrier completion.

Implementation of the fuzzy barrier semantics in both schemes requires special care that the barrier hardware not be initialized for the next use before all the nodes have a chance to recognize barrier completion. In the purely optical scheme, if the Pulse Generator automatically resets the hardware (by generating another pulse) upon barrier completion, a node might not issue the `BARRIER-WAIT` command soon enough to see barrier completion for itself. If the `INIT-BARRIER` command is to be issued every time the barrier hardware is reset, the question arises under the fuzzy barrier semantics of which node recognizes barrier completion last. This problem can be solved by having a *completion-flag register* (CFR) associated with the wired-`OR` OBR. To implement the fuzzy barrier semantics, the node issues the `BARRIER-WAIT` command immediately after the `REACH-BARRIER` command, but then the node is allowed to execute any code not dependent upon barrier completion. While the node executes that code, there is separate logic monitoring the "B" input tap; when the "B" tap goes low, the logic sets the CFR. After the node finishes executing the code not dependent upon barrier completion, it stalls on the CFR until the CFR is set; the node then clears the CFR and proceeds past the barrier. Since the CFR records for the node when barrier completion occurs, the wired-`OR` OBR can be reset immediately for the next barrier use by either initialization scheme, and other nodes can even reach the barrier again before the node recognizes barrier completion. The hybrid scheme can be extended in a similar manner by adding a `completion-flag` field to the data structure shown in Fig. 30. The monitoring algorithm shown in Fig. 32 is modified as shown in Fig. 34 to have the `completion-flag` cleared when an `INIT-BARRIER` command is received and to set the `completion-flag` and reset the `counter` to 0 when the number of nodes reaching the barrier equals the number of participants. The `BARRIER-WAIT` command algorithm is modified as shown in Fig. 35 to stall on the `completion-flag` and clear it after recognizing barrier completion.

The hardware requirements for the two OBee barrier schemes are shown in Table 10. The purely optical scheme requires one wired-`OR` OBR per hardware barrier; in addition, one participation barrier register (PBR) per node per barrier is required (if the fuzzy barrier semantics are implemented, then one completion-flag register per node per barrier is also required). Simple decision logic is also needed to interpret when optical signals are present and absent and to implement the rules presented in Section 5.2.1. The hybrid scheme requires one explicit message OBR

```
switch  command:
  case  INITIALIZE:
      participation := N;
      counter       := 0;
      CLEAR completion-flag;
      break;

  case  REACH:
      counter++;
      if (counter == participation):
        SET completion-flag;
        counter := 0;
        endif;
      break;

      endcase;
   endswitch.
```

Figure 34: The monitoring algorithm that each node follows in processing traffic on the Barrier Channel for fuzzy barrier semantics.

```
while (completion-flag ==  CLEAR):  stall; endwhile;

CLEAR completion-flag;
 resume program execution.
```

Figure 35: The algorithm a node follows to implement the `BARRIER-WAIT` command for fuzzy barrier semantics in the hybrid barrier scheme.

OBee Barrier Scheme	*Hardware Requirements*
purely optical	1 wired-`OR` OBR/barrier 1 PBR/node/barrier simple decision logic
hybrid	1 Barrier Channel (explicit message OBR) 1 data structure/node/barrier message interpretation logic

Table 10: Comparison of hardware requirements for the OBee barrier schemes.

OBee Barrier Scheme	`INIT-BARRIER`	`REACH-BARRIER`	`BARRIER-WAIT`
purely optical	N	N	0
hybrid	1	N	0

Table 11: The number of messages for the various barrier operations under the two OBee barrier schemes.

(the Barrier Channel) to carry all the traffic for all the barriers. Each barrier also requires a data structure which is replicated at every node. We have not previously specified the implementation of this data structure. The most straightforward implementation is to have a Barrier Table, constructed of a fast associative memory, located at each node. Each named entry in the Barrier Table corresponds to one of the data structures shown in Fig. 30 (we assume the name is assigned by the program itself and the operating system). The message interpretation logic applies a barrier operation (as specified in the monitoring algorithms) as received over the Barrier Channel to the appropriate Barrier Table entry. If the barrier traffic warrants, the traffic could be split over several Barrier Channels, which would require the appropriate duplication of the message interpretation logic. In order to preserve most easily the barrier operations' semantics when the traffic is so split, we require that all the traffic pertaining to any given barrier be restricted to one Barrier Channel.

The OBee barrier schemes may also be compared in terms of the number of messages required for the various barrier operations, as shown in Table 11. A message for the purely optical scheme is defined (as in the discussion for locks) to be either the raising or lowering of the optical signal. In the purely optical scheme, whenever the barrier hardware is initialized, each node must raise its "T" output signal. Therefore, by definition, if N nodes participate in the barrier, N messages are required; however, these messages will all be transmitted in one cycle of the wired-`OR` OBR. The hybrid scheme requires only one message since the message is broadcast to all the nodes. An additional difference between the two schemes occurs when the barrier is initialized for each use. The purely optical scheme requires, as previously discussed, that either an explicit `INIT-BARRIER` command be given or that the Pulse Generator automatically start the initialization upon barrier completion. Either way, N messages are required every time to reset the barrier hardware before its next use. In contrast, in the hybrid scheme, each node stores in the `participation` field the contents of the `INIT-BARRIER` command; thus, unless the number of participating nodes changes, no messages are required to reset the barrier before its next use. When a node reaches the barrier, both schemes require that the node send a message (lowering a signal or a `REACH-BARRIER` message); therefore, N messages (the minimum number) are required for the N participating nodes to reach the barrier. Since each node, in either scheme, recognizes barrier completion either from observing the state of the OBR or from monitoring the OBR traffic, the `BARRIER-WAIT` command is strictly a local operation and requires no messages on the OBR.

5.3 F&OP

The F&A (Fetch-and-Add) synchronization operation was introduced by Gottlieb [34]. Various coordination algorithms for barriers, queues, and semaphores using

F&OP Primitive	*Semantics*
`INIT-FOP`	Initialize the appropriate F&OP hardware.
`F&OP(V,e)`	Atomically perform `(V := V OP e)` ; return old `V`.

Table 12: The semantics of the F&OP synchronization primitive in OBee.

F&A have been proposed [2]. The F&OP (Fetch-and-OP) primitive is a generalization of the F&A primitive. Table 12 shows the format and semantics for the F&OP command. The operator represented by `OP` can be any associative, commutative operator. When `F&OP(V,e)` is issued by a node, the command returns the *old* value of `V` to the node and atomically replaces `V` with `(V OP e)`. The `INIT-FOP` command is used to initialize the F&OP hardware; its exact specification depends upon the implementation.

A fundamental property of the F&OP command is its potential combining capability. This capability is expressed by saying that the F&OP must satisfy the serialization principle: if `V` is a shared variable and many tasks issue `F&OP(V,e)` simultaneously to the same `V`, then the effect of the many F&OP commands issued in parallel is exactly what it would be if they had occurred in some (unspecified) serial order. That is, the final value of `V` due to the parallel `F&OP(V,e)` commands is the result of applying all the operators `OP` and operands `e` to the original `V`, and each task receives an intermediate value of `V` depending upon where its own particular `F&OP(V,e)` command happens to fall in the arbitrary serial order. The designer of an algorithm using the F&OP semantics must realize he *cannot* design the algorithm to rely upon any particular value of `V` that the F&OP operation returns.

The key problem in implementing the F&OP command is providing hardware which can support the combining capability. In the following sections we will describe a *restricted* form of F&OP that can be implemented with the wired-`OR` OBR and a general, unrestricted F&OP that can be implemented with the explicit message OBR.

5.3.1 Purely Optical RF&OP Scheme

A restricted form of F&OP (denoted RF&OP) was proposed by Sohi et al. [77]; their description includes proposed electrical bus-based hardware support. The RF&OP imposes the following constraint for implementing the potential F&OP combining: in any given bus cycle all participating nodes must perform the same `OP` with the same operand `e` on the same target `V`. Their scheme works as follows. Suppose that M nodes issue the `RF&OP(V,e)` command during one bus cycle and that every node knows which other nodes have issued this exact command. If the nodes know their place in some pre-assigned ordering, then each of the M participating nodes can locally compute the correct number of `RF&OP(V,e)` commands to determine what value of `V` they obtain from the command (given either the original or final value of `V`). Some mechanism is needed to ensure that participating nodes have the original (or final) value of `V`; Sohi et al. suggest having memory (where the target `V` resides) be responsible for supplying `V` (and updating it) to the participating nodes.

Our purely optical RF&OP scheme is similar to the proposal of Sohi et al. As with the purely optical locks and barriers schemes, this scheme relies only upon the presence and absence of optical signals to convey the needed information (actually, as will be explained below, the ability to detect *discrete drops* in the signal levels is needed). The key ideas behind this scheme are: (1) during a well defined access cycle, nodes wishing to participate in a `RF&OP(V,e)` operation so signal their desire; (2) as the access cycle progresses, nodes remove themselves from participation in the `RF&OP(V,e)` operation; and (3) all nodes observe each node removing itself from participation, thereby allowing each node to calculate what its `RF&OP(V,e)` command returns to it as well as the final value of `V` for that cycle. This scheme employs the wired-`OR` OBR architecture in which one wired-`OR` OBR is assigned per `V`, `OP`, and `e` combination in the program. The topology of the wired-`OR` OBR is exactly that of Fig. 8, and the access protocol is the second one enumerated in Section 3.3.1 which prescribes an ordered access to the medium.

The access cycle to the wired-`OR` OBR begins by the Pulse Generator sending out a pulse. A node is allowed to raise its "T" output signal whenever it sees its "U" input signal high:

$$\Uparrow \mathrm{T} \quad \text{WHENEVER} \quad \mathrm{U}.$$

The raising of the "T" output signal signifies that the node wishes to perform the `F&OP(V,e)` command that has been assigned to this wired-`OR` OBR hardware. The nodes use the ordering that results from the access protocol of the wired-`OR` OBR to determine their place in the serial ordering of the `F&OP(V,e)` commands issued during the access cycle. Therefore, after raising its "T" output, each node, one by one, drops its "T" output, which causes a discrete drop in the optical signal level

`V-current`	`V-mine`	`OP`	`e`

Figure 36: The structure of the F&OP Definition Table in the purely optical RF&OP scheme.

that will be seen at each downstream node's "U" tap and at all nodes' "B" taps:

$$\Downarrow\mathrm{T} \quad \text{AFTER} \quad \overline{\mathrm{U}}.$$

In order for each node to know how many nodes upstream of it are issuing the `F&OP(V,e)` command, the wired-`OR` OBR needs an additional capability: each node's input tap (the "U" and "B" taps) must be able to detect discrete drops in the optical signal level. Thus, by counting the number of drops at the "U" tap, the node determines how many upstream nodes have issued the `F&OP(V,e)` command. When the node knows its place in the serial order, then, given the old value of `V`, it can calculate the value of `V` it obtains by having issued the `F&OP(V,e)` command.

We now consider how the node obtains the old value of `V` so that it can calculate the value of `V` it obtains from issuing `F&OP(V,e)`. Associated with each wired-`OR` OBR is a data structure called the *F&OP Definition Table* (FDT), whose layout is shown in Fig. 36. The FDT is replicated at every node. The `V-current` field stores the value of `V` when the current access cycle begins. The `V-mine` field stores the value of `V` the node receives from having issued the `F&OP(V,e)` command. The `OP` field and `e` fields designate what operation and operand respectively are associated with this F&OP wired-`OR` OBR hardware. For this implementation, the `INIT-FOP` command takes the form `INIT-FOP(V,OP,e)` in order to initialize the FDT (realize that the wired-`OR` OBR is not used to transmit the initialization of the FDT). Thus, the `INIT-FOP` command supplies the FDT with the original value of `V`.

To determine the value of `V` its `F&OP(V,e)` returns, each node uses the following algorithm:

```
if (node is participating in access cycle):
  C :=  number of drops at “U” tap;

  V-mine := V-current;
  for i = 1 to (C - 1) do:
     V-mine := (V-mine OP e);
     enddo.
```

Note that the node applies `(OP e)` to the current value of `V` one less time than the number of drops at the “U” tap. Lessening the drop count by one is necessary because the first drop each node sees is due to the pulse from the Pulse Generator going by. If this extra drop is not ignored, then the semantics of the F&OP command are violated since the first node in the serial order should obtain the current value of `V` back as its own value.

In order for the nodes to know the current value of `V`, each node, regardless of whether it issues the `F&OP(V,e)` command during a given bus cycle, must monitor its “B” input tap to calculate the new value of `V` after each bus cycle. Each node uses the following algorithm for monitoring the “B” tap:

```
regardless of participation during access cycle:
C :=  number of drops at “B” tap;

for i = 1 to (C - 1) do:
   V-current := (V-current OP e);
   enddo.
```

Note the count of drops is again reduced by one since the pulse from the Pulse Generator is an extra pulse. The adjusted number of drops is the number of nodes that issued the `F&OP(V,e)` during that access cycle. Since each node sees the same number of drops at the "B" tap and follows the same algorithm, each node keeps its copy of `V-current` up to date for the next access cycle.

Up until now, we have said that a node may lower its "T" signal any time after its "U" input tap goes low. The node is actually constrained not to lower its "T" signal any faster than the counting logic of downstream nodes can detect the signal drops and count them. In addition, the logic that implements the "B" tap counting should be constrained not to modify `V-current` until the node (if applicable) has calculated its `V-mine` value. Finally, when can the Pulse Generator send out another pulse to start the access cycle again? The Pulse Generator cannot send out another pulse until it is assured that every node has had a chance to update the `V-current` value. Since Node 1 will be the last one to perform this calculation, then if the Pulse Generator is controlled by Node 1, it will send out pulses at the correct time.

As a simple illustration of this scheme, assume that `V-current` is 23, `OP` is addition, and `e` is 1. Assume that in a 10 node system, for the access cycle under consideration, nodes 2, 5, and 7 issue the `F&OP(V,+,1)` command. A pulse from the Pulse Generator starts the access cycle. The pulse allows Node 2 to raise its "T" signal; when the pulse goes by, since no other upstream nodes requested access, then Node 2 sees one drop at its "U" tap, which, by the "U" tap algorithm, means that its `V-mine` takes the `V-current` value of 23. When the pulse gets to Node 5, the node is triggered to raise its "T" signal; Node 5 will see two drops as its "U" tap goes dark, which means it applies `(+ 1)` once to `V-current` to yield a value of 24 for `V-mine`. Similarly, Node 7 will see three drops at its "U" input, which

`V-current`	`V-mine`

Figure 37: The data structure associated with each target `V` in the hybrid F&OP scheme.

means it applies `(+ 1)` twice to `V-current` to yield 25 for its value of `V-mine`. All ten nodes apply the "T" tap algorithm to keep their copies of `V-current` up to date: `(+ 1)` is applied three times so that `V-current` acquires the value 26.

5.3.2 Hybrid Electronic/optical Scheme

The hybrid scheme removes the restrictions of the purely optical RF&OP scheme. The hybrid scheme allows nodes to issue any arbitrary `F&OP(V,e)` command (meaning that `OP`, `V`, and `e` are all arbitrary) at any time they are given access to the OBR. This scheme uses explicit messages, which represent the arbitrary `F&OP(V,e)` commands, transmitted over one logical channel called the *F&OP Channel*. The idea behind this scheme is that each node maintains an appropriate data structure to represent each target `V` and that by monitoring the traffic on the F&OP Channel, the node can determine the value of `V` it obtains from its F&OP command. As with the hybrid lock and barrier schemes, the broadcast capability is needed so that each node can independently modify the data structure as it listens to all the traffic. This scheme naturally maps to one explicit message OBR for which the control and transmission topologies are as shown in Figs. 10 and 11 and for which the access rules are as enumerated in Section 3.3.2.

The data structure associated with each target `V` is shown in Fig. 37. As in

SENDER	V-target	OP	e

Figure 38: The format of a message transmitted over the F&OP Channel.

```
if (SENDER ==  SELF):
  V-mine := V-current;
   notify CPU that the F&OP has completed by returning V-mine;
  endif;

V-current := (V-current OP e).
```

Figure 39: The monitoring algorithm each node follows in the hybrid F&OP scheme.

the purely optical scheme, the `V-current` field stores the value of `V`, and the `V-mine` field stores the value of `V` the node obtains from having issued the **`F&OP(V,e)`** command. However, unlike the purely optical scheme, the data structure does not need to store the `OP` or `e` parameters because they are transmitted over the F&OP Channel as part of the message that represents each **`F&OP(V,e)`** command. As with the purely optical scheme, this data structure is replicated at each node.

In this implementation, the `INIT-FOP` command has the form `INIT-FOP(V)` so that the data structure is initialized with the proper value of `V-current`. When a node issues a `F&OP(V,e)` command, a message of the form shown in Fig. 38 is transmitted over the F&OP Channel. The `SENDER` field identifies which node transmitted the message. The `V-target` field represents which `V` is the object of the operation, and the `OP` and `e` fields specify the rest of the F&OP operation.

As mentioned, each node monitors the traffic on the F&OP Channel. The monitoring algorithm that each node follows is shown in Fig. 39. The algorithm can be understood by considering that each node sees a stream of `F&OP(V,e)` commands going by and that each node processes each command in the order received (recall that because of the OBR's topology, each node will see the same message order as every other node). If a node sees that the F&OP command is from itself, then it realizes that it needs to supply the value of `V` that its F&OP should return. This value will be the current value of `V` *before* the F&OP command is applied to the current `V`. Regardless of whether the command is from itself, the node, to keep the value of `V` current, applies the F&OP operation specified in the message to `V-current`.

5.3.3 Discussion

We first comment on how the two F&OP schemes make use of the characteristics of the OBR architecture. The purely optical scheme uses the ordering and broadcast features of the wired-`OR` OBR. The ordering feature allows a natural ordering of the issued F&OP commands to be formed, which, in turn, allows each node to calculate what it should receive from the F&OP command. The broadcast property allows every node to monitor all the F&OP commands and thus independently keep track of the current value of the F&OP target `V`. The hybrid scheme makes use of the ordering and broadcast features of the explicit message OBR in a similar fashion.

We point out that unlike traditional F&OP combining hardware, in which the combining takes place in the interconnection network, these two schemes are completely distributed in their combining, i.e., the nodes are responsible for monitoring the F&OP commands, performing the combining locally, determining the proper return value of their F&OP, and maintaining the up to date value of `V`. Insofar as we have described these schemes, each node processes the F&OP commands (be they represented by optical signal drops or explicit bit messages) serially. But, in the case of the purely optical scheme, the processing hardware needed to implement the "U" or "B" tap algorithms could be easily extended in two ways. First, after counting how many drops occur, the hardware could combine all the `(OP e)` operations into one operation. For example, if `OP` is the addition operator, then a combined multiplicative operation can be substituted. Second, the hardware used for processing those algorithms could be made parallel, e.g., a pipelined processing unit can be used to implement the needed number of `(OP e)` operations. These two techniques could also be used in the hybrid scheme, although perhaps not with as much success since the hybrid scheme allows the intermixing of F&OP commands on the F&OP Channel. If, however, there was enough F&OP traffic to warrant splitting the traffic over several F&OP Channels, then perhaps parallel message processing hardware would be worthwhile.

Another optimization in the F&OP processing hardware can be made if it was decided to support only a certain number of `V`, `OP`, and `e` combinations (Freudenthal and Gottlieb [25] discussed implementations of various algorithms in which `OP` is either the increment or decrement operator and `e` is always the constant 1). In the purely optical scheme, each wired-`OR` OBR and associated drop count processing hardware could then be optimized to implement only the prescribed `V`, `OP`, and `e` combination. Such an optimization would eliminate the need for the `OP` and `e` fields in the FDT shown in Fig. 36. This optimization would also be useful in the hybrid scheme if the F&OP traffic were split over several channels;

OBee F&OP scheme	*Hardware Requirements*
purely optical	1 wired-`OR` OBR/`F&OP(V,e)` combination 1 FDT/node/`F&OP(V,e)` combination simple decision and update logic
hybrid	1 F&OP Channel (explicit message OBR) 1 data structure/node/target `V` message interpretation logic

Table 13: The hardware requirements for the two OBee F&OP implementations.

the message processing hardware associated with a given channel would need to be capable of processing only those types of messages transmitted via that particular channel.

The hardware requirements for the two F&OP schemes are summarized in Table 13. Under the assumption of a static assignment of F&OP hardware to program needs, the purely optical scheme requires one wired-`OR` OBR per `F&OP(V,e)` combination. One FDT per node per `F&OP(V,e)` combination is required so that the simple decision and update logic can perform the proper processing of the optical signals. If the designer wants to reduce the number of required wired-`OR` OBRs and associated FDTs and processing hardware, then the operating system must be able to make dynamic assignment of the F&OP hardware to the program needs. The hybrid scheme requires one explicit message OBR to implement the F&OP Channel. The message interpretation logic also needs one data structure per node per target `V`. The message interpretation logic is fairly complex since, unless the design is restricted *a priori*, the F&OP commands received on the F&OP Channel can contain arbitrary `OP` and `e` parameters. As with the hybrid barrier design, the required data structures are stored in a F&OP Table, which is implemented

OBee F&OP scheme	`INIT-FOP`	`F&OP(V,e)`
purely optical	1	2
hybrid	1	1

Table 14: The number of messages used by a node in the two F&OP schemes.

with an associative memory. Each named entry in the F&OP Table corresponds to one of the data structures shown in Fig. 37 (we assume that the name is assigned by the program itself and the operating system). If the F&OP traffic warrants, several F&OP Channels (each corresponding to one explicit message OBR) can be used, which requires a duplication of the message interpretation logic. If the traffic is so split over several channels, then, to preserve most easily the semantics of the F&OP operations, we require that all the traffic for a particular target `V` be carried on only one channel.

The number of messages required in the two F&OP schemes is shown in Table 14. As has been defined before, a message in the purely optical scheme corresponds to either the raising or lowering of the optical signal. Although the `INIT-FOP` command is not issued over the wired-`OR` OBR in the purely optical scheme, for completeness, we show that it requires one message (sent in some other way) to initialize the FDT; the hybrid scheme also requires one message to initialize the appropriate entry in the F&OP Table. In the purely optical scheme, strictly speaking, a node sends two messages to complete its `F&OP(V,e)` operation. One message is the raising of its "T" signal to signify participation in this `F&OP(V,e)` operation, and the second message is the dropping of its "T" signal to inform downstream nodes of its specific place in the command issuance order. In the hybrid scheme, a node sends only the one explicit message indicating what specific `F&OP(V,e)` operation it is issuing.

Chapter 6

Evaluation

In this chapter we present some analytical evaluations of the proposed architectural features. These evaluations are built from the basic analysis of the OBR architecture presented in Section 3.3.3. In this analysis we are interested in the fundamental performance of the architectural features. That is, we are not considering in this analysis the performance of the program as a whole, just the raw, basic performance characteristics of the architectural features.

We evaluate the architectural features in terms of the best and worst case execution times. Execution time is defined to be the time from when a node wants to issue a command to the time when it gets the command's result back (this definition makes execution time the same as response time). Since the OBR's topology and access protocol are deterministic, these two metrics are also deterministic. In each of the following sections we give the appropriate definitions and assumptions pertinent to the derivation of the best and worst case execution time for each architectural feature. After deriving the equations, we use four different types of graphical plots of the equations to make some observations about their behavior. The plots are constructed to give information about the equations' dependencies upon small and large system sizes and small and large distances for coupling the nodes together. Specifically, one type of plot assumes a small fixed system size

(50 nodes) and shows the equations' dependence upon the propagation distance. The second type shows the equations' dependence upon the propagation distance for a large fixed system size (500 nodes). The third type of plot assumes a room size environment for coupling the nodes together (fixed propagation distance of 10 m) while showing the equations' dependence upon the system size. The fourth type of plot shows the equations' dependence upon the system size when a building size environment (fixed propagation distance of 100 m) is assumed.

6.1 Cache Coherency

In considering the execution time for the reader initiated cache coherency protocol, we focus on the `WRITE-UPDATE` command. For that command, we are interested in knowing how long it takes for its effect to be seen by all the nodes. Recall that the `WRITE-UPDATE` command is transmitted over an explicit message OBR (and we assume fixed length messages are used on the OBR). The execution time E_C for the `WRITE-UPDATE` command from a given node is defined as:

$$E_C = T_{queue} + T_{request} + T_{deliver}.$$

The T_{queue} term represents how long the `WRITE-UPDATE` command stays in the node's Write Buffer before becoming the top entry in the Write Buffer; we exclude this term from our analysis since it is dependent upon the program's behavior. The $T_{request}$ term represents the time it takes, once the command is the top entry in the Write Buffer, for the command to put its message on the explicit message OBR. The $T_{deliver}$ term is the time it takes for the message to be delivered to all

Load	*Minimum*	*Maximum*
idle	$(2\Delta + L) + (\tau + D)$	$[\min(2\tau, NL) + \Delta + 2\Delta + L]$ $+(2\tau + D)$
high	$(2\Delta + L) + \min(2\tau + D, \tau + ND)$	$(NL + \Delta + 2\Delta + L)$ $+\max(2\tau + D, \tau + ND)$

Table 15: The minimum and maximum execution times for the `WRITE-UPDATE` command under idle and high load conditions.

the nodes in the system.

There are two situations to be analyzed for the execution time for the `WRITE-UPDATE` command, *idle* load and *high* load. Idle load means that only one node is trying to put a message on the OBR. High load implies that all the nodes are attempting to put messages on the OBR. Table 15 gives the formulas for the minimum and maximum E_C times for the idle and high load situations for the `WRITE-UPDATE` command. The Δ term represents the component and circuitry delay times of the optical protocols (as used in Section 3.3.3), e.g., the time to recognize the drop in an optical input tap and turn on an optical transmitter. The symbol τ is the propagation time from the first node to the last node and vice versa. The number N represents the number of nodes in the system. Since the cache coherency is implemented only with the hybrid scheme, the value L is the (average) length of a message, and the value D is the time it takes to decode a message.

These formulas are derived as follows. Note that each formula contains two terms (a parenthetical expression counts as one term) which correspond directly with the $T_{request}$ and $T_{deliver}$ terms being analyzed for the E_C execution time. Also, note that we must determine when the command message is heard by all nodes, i.e., when does the physically first node on the OBR hear and decode the command message? We first consider the idle load situation in which the given node is the only one accessing the OBR. In the minimum execution time case, the control pulse arrives at the given node just as it wants to issue the `WRITE-UPDATE` command, so the node takes 2Δ to recognize the pulse and begin its transmission, which lasts time L. If the given node is the physically last node on the OBR, then the message travels the propagation distance τ, and it takes time D to be decoded. In the maximum execution time case, the control pulse passes by the given node just after the node issues the command. Therefore, the node waits $\min(2\tau, NL) + \Delta$ for the pulse to be regenerated and come back again. Then, the node recognizes the pulse and begins its transmission 2Δ time later. The transmission lasts L time. If the given node is the physically first node on the OBR, then the command message travels 2τ to reach itself, and it takes time D to be decoded.

We next consider the high load situation in which all the nodes have messages to put on the OBR. In the minimum execution time case, the control pulse arrives at the given node just as it issues the `WRITE-UPDATE` command, so it takes 2Δ for the node to recognize the pulse and begin its transmission, which lasts for L time. Now, how long does it take for the node's command message to be seen by the physically first node? If the given node is the physically first node, then the message travels 2τ to reach itself and is decoded in time D. If the given node is the physically last node, then the message travels only τ, but since it is the last message in the message train, it is not decoded until time ND. Thus, the minimum delivery time is $\min(2\tau + D, \tau + ND)$. In the maximum execution time, the node issues the `WRITE-UPDATE` command just after the pulse passes by it. The node then waits $NL + \Delta$ for the pulse to be regenerated and come back to it. The node then recognizes the pulse and turns on its message in time 2Δ. The message lasts for time L. For this situation, what is the worst time for the command message to be seen by the physically first node? If the given node is the physically first one, then the message travels 2τ to reach itself and is decoded in time D. If the given node is the physically last one, then the message only travels τ, but since it is the last message in the message train, it takes time ND before it is decoded. Thus, the maximum delivery time is $\max(2\tau + D, \tau + ND)$.

We now present various plots of these equations in order to understand the scale of the execution time of the `WRITE-UPDATE` command. We assume that the average message length L is 75 nsec (16 bytes of cache line plus overhead bits for a total of 150 bits at 2 Gbit/sec), the decoding time D is also 75 nsec, and Δ is 10 nsec. Figs. 40, 41, 42, and 43 show various plots for the idle load situation, and Figs. 44, 45, 46, and 47 show various plots for the high load situation. Note that the propagation distance τ is presented in terms of time. Since light propagates at approximately 5 nsec/m in a waveguide, a 500 nsec propagation time corresponds to 100 m.

In Figs. 40 and 41 the number of nodes has been fixed and the propagation distance allowed to vary under idle load. These plots show that since the minimum E_C time does not depend upon N, the plots for the minimum E_C are the same in the two figures. The plots for the maximum E_C are also identical since for these two value of N, the $\min(2\tau, NL)$ is 2τ. We note from these two figures that the worst case E_C time under the idle load is less than 2.5 μsec even for 500 nodes

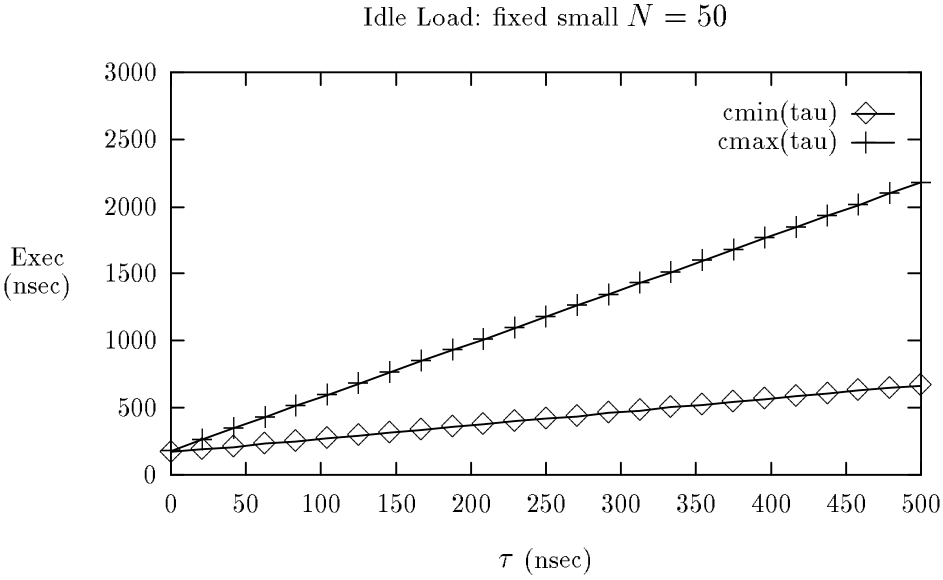

Figure 40: The minimum and maximum E_C times for idle load for small fixed $N = 50$.

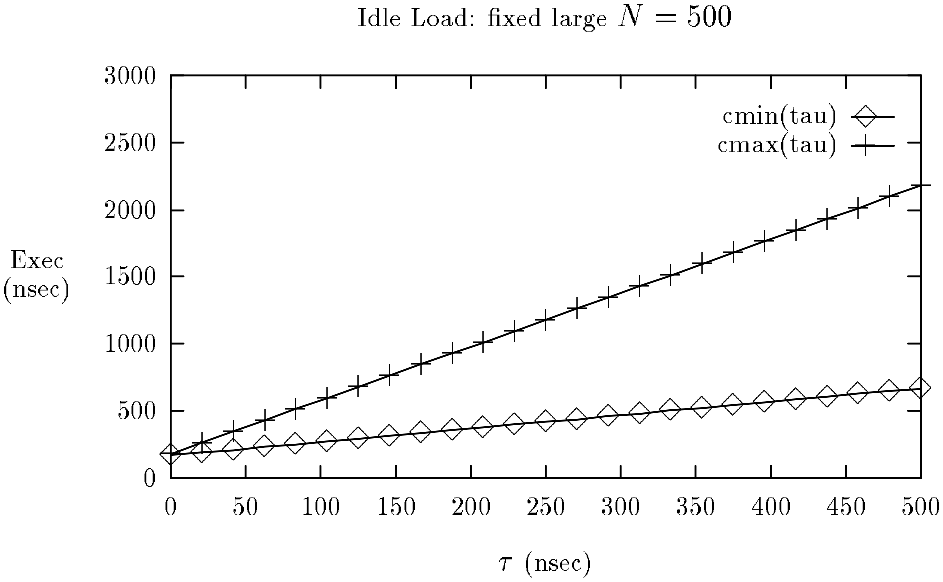

Figure 41: The minimum and maximum E_C times for idle load for large fixed $N = 500$.

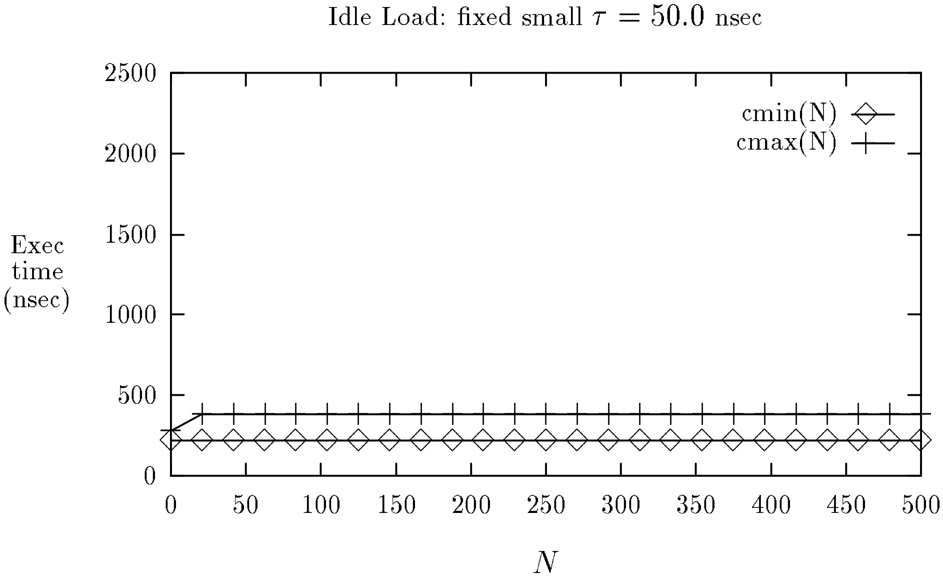

Figure 42: The minimum and maximum E_C times for idle load for small fixed $\tau = 50.0$.

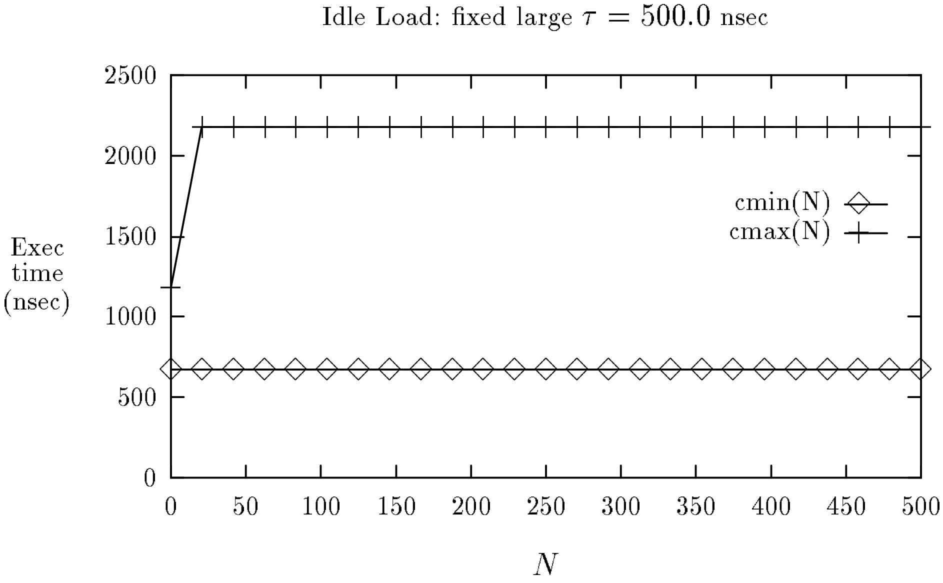

Figure 43: The minimum and maximum E_C times for idle load for large fixed $\tau = 500.0$.

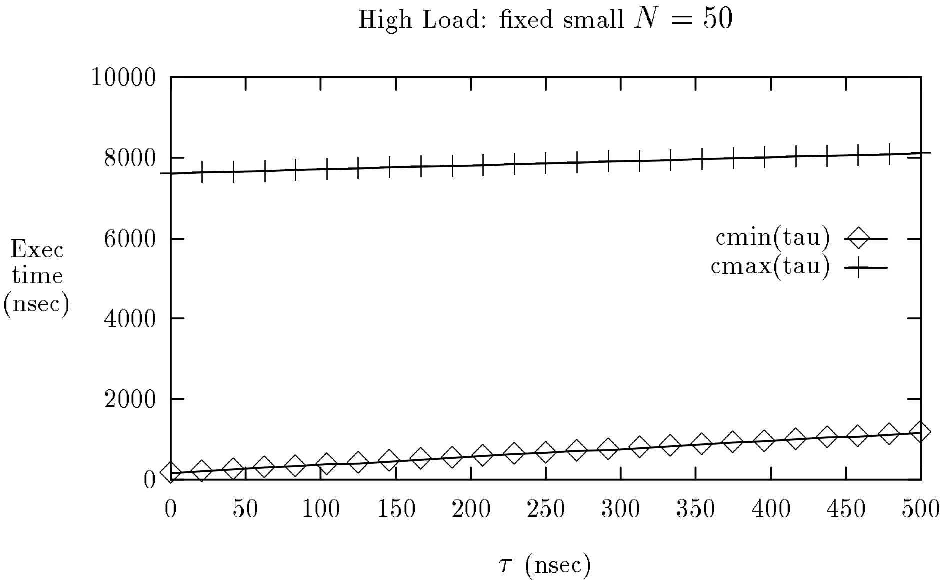

Figure 44: The minimum and maximum E_C times for high load for small fixed $N = 50$.

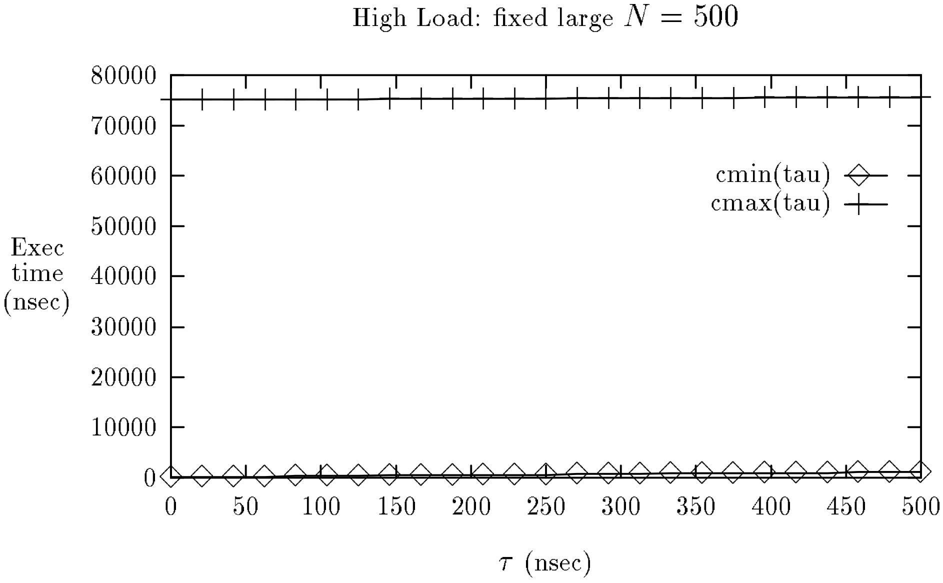

Figure 45: The minimum and maximum E_C times for high load for large fixed $N = 500$.

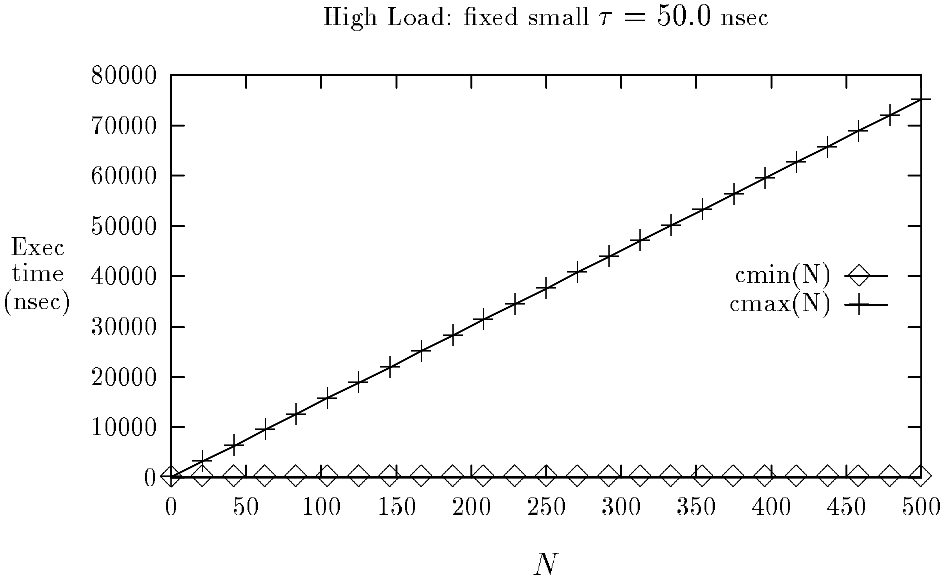

Figure 46: The minimum and maximum E_C times for high load for small fixed $\tau = 50.0$.

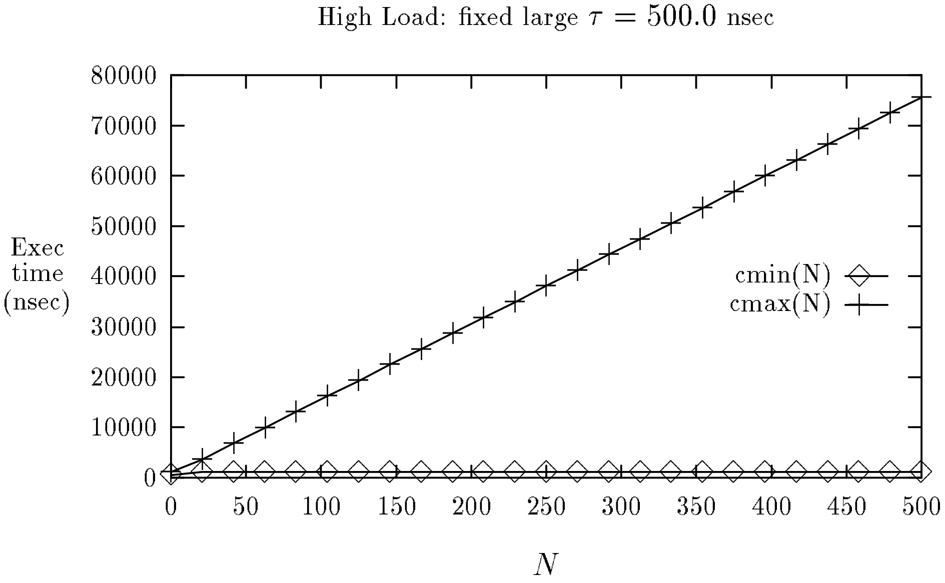

Figure 47: The minimum and maximum E_C times for high load for large fixed $\tau = 500.0$.

100 m apart.

In Figs. 42 and 43 the propagation distance has been fixed and the number of nodes allowed to vary under idle load. In both figures, the minimum E_C is constant (though not the same numeric value) since it does not depend upon N. The maximum E_C becomes constant in both figures (again, though, not the same numeric value) after reaching a threshold point. The threshold point affects the value of the $\min(2\tau, NL)$ term; when N is small enough, the propagation distance dictates when the Pulse Generator sends out a new pulse; when N is large enough, the Pulse Generator sends out a new pulse based upon the maximum train length. We note from Fig. 42 with small τ that regardless of N, the maximum E_C is less than 0.4 μsec; from Fig. 43 with large τ, regardless of N, the maximum E_C is less than 2.5 μsec.

In Figs. 44 and 45 the value of N has been fixed and the propagation distance allowed to vary under high load. Because of the scales of the figures, it is not readily apparent, but the minimum E_C is the same in both figures since for these values of N, the $\min(2\tau + D, \tau + ND)$ term reduces to $2\tau + D$. Also note that the minimum E_C time only grows linearly with the propagation distance τ. The maximum E_C plots can be understood by looking at the formula qualitatively. The first term says that the node has to wait one message train length (NL) before it puts its message out; the second term says that since the given node for the maximum case is the physically last node, its message is at the end of the message train in which it participates. Thus, the execution time is approximately the length of two message trains (since $L = D$). In the small fixed value of N case (Fig. 44), the propagation distance has some effect upon the plot of the equation, but in the

N	τ (nsec)	Execution Time (nsec)	
		min	max
50	50	220	380
50	500	670	2180
500	50	220	380
500	500	670	2180

Table 16: Some specific Idle Load `WRITE-UPDATE` execution times.

N	τ (nsec)	Execution Time (nsec)	
		min	max
50	50	270	7655
50	500	1170	8105
500	50	270	75155
500	500	1170	75605

Table 17: Some specific High Load `WRITE-UPDATE` execution times.

large fixed value of N case (Fig. 45), the propagation distance's effect disappears. Note that even in the large fixed N case, the worst case E_C is less than 80 μsec.

In Figs. 46 and 47 the propagation distance has been fixed and the value of N allowed to vary under high load. Since the minimum E_C time is linearly dependent upon τ, at these scales, the minimum E_C times are nearly the same. For the maximum E_C times, the reasoning of the previous paragraph applies: the execution time is approximately the length of two message trains. Again, note that for these parameters, the worst case of E_C is less than 80 μsec.

As a summary, Tables 16 and 17 give some specific execution times for the `WRITE-UPDATE` command under idle and high load conditions respectively for small and large system sizes and for small and large system scales.

6.2 Locks

In considering the execution time for locks, we are interested in knowing how long it takes for a node to be granted a lock. The execution time E_L for locks is defined for a given node as:

$$E_L \quad = \quad T_{request} + T_{usage} + T_{grant}.$$

The $T_{request}$ term denotes how long it takes for the given node to get its request placed into the queue of lock requests. The T_{usage} term represents how long other nodes which have previously requested and been granted the lock hold the lock before the given node is granted the lock. The T_{grant} term is the time that the immediate prior lock holder takes to release and grant the lock to the given node. The first and third terms ($T_{request}$ and T_{grant}) are dependent upon the hardware supporting lock requests releases and grants. Since the T_{usage} term is dependent upon the program's execution characteristics and not the lock hardware, we exclude its effect from this discussion by considering it to be a constant.

There are two situations for which to analyze the execution time for locks, *idle* locks and *busy* locks. An idle lock is one not being held by any node by the time the given node's request is placed into the lock request queue, i.e., the T_{usage} term is 0. A busy lock is one previously requested and being held by some node by the time the given node's request is placed into the lock request queue, which implies that T_{usage} can be non-zero. Table 18 shows the formulas for the minimum and maximum E_L times for the idle and busy situations for the different lock schemes. The symbols in these formulas are the same as for the cache coherency analysis: Δ is the component and circuitry delay time; τ is the propagation distance from

Scheme	*Minimum*	*Maximum*
purely optical (idle)	$\Delta + 0 + \Delta$	$(2\tau + 2\Delta + \Delta) + 0 + \Delta$
purely optical (busy)	$\Delta + 0 + \Delta$	$(2\tau + 2\Delta + \Delta) + T_{usage}$ $+(\tau + \Delta)$
hybrid (idle)	$(2\Delta + L) + 0 + D$	$[NL + \Delta + 2\Delta + L]$ $+0 + \max(2\tau + D, ND)$
hybrid (busy)	$(2\Delta + L) + 0$ $+\min(2\tau + D, PD)$	$[NL + \Delta + 2\Delta + L] + T_{usage}$ $+[NL + \Delta + 2\Delta + L$ $+\max(2\tau + D, \tau + ND)]$

Table 18: The minimum and maximum execution times for idle and busy locks for the two OBee lock schemes.

the first node to the last node and vice versa; N is the number of nodes in the system; and L and D are the average message length and message decoding time respectively in the hybrid scheme. Additionally, the value P is the number of nodes contending for a given lock in the hybrid scheme.

These formulas are derived as follows. Note that each formula has three terms (a parenthetical expression counts as one term) that directly correspond to the definition of E_L. We first consider the purely optical scheme under the idle load condition. Recall that for idle load the T_{usage} term is zero. For the minimum execution time, the node issues the lock request just *before* the pulse triggers the node. It takes one Δ for the node to recognize the pulse and make its request. Since no other node holds the lock, it takes one more Δ for the node to be granted the lock. In the maximum execution time case, the node issues the lock request just *after* the pulse passes the node. The pulse must propagate for 2τ, and it takes 2Δ for the pulse to be regenerated (since there are two Pulse Generators). It then takes one more Δ for the node to recognize the pulse and make its request. Again, since no other node holds the lock, it takes one more Δ for the node to be granted the lock.

Under the busy load condition, the purely optical scheme has the same minimum execution time as under the idle load condition. This result happens in the two following situations. One situation regards W/E requests: the node makes its request just before the prior lock holder releases the lock; the request takes Δ time. If the prior lock holder is the node physically preceding the given node and it gives up the lock just as the given node finishes its request (the absolute best case), then T_{usage} is zero, and the given node takes one Δ to be granted the lock. This situation also assumes that the given node receives the lock release signal just as it is finished making its request. The other situation regards R/S requests: the given node makes its request just before the signal denoting giving permission to its peer group reaches it. Again, it takes one Δ to make the request, and another Δ to be granted the lock (and again, T_{usage} is zero). In the maximum execution time case, the given node issues the lock request just after the control pulse passes the node. It takes $2\tau + 2\Delta$ for the next pulse to come to the node plus one more Δ for the node to make its request. The node must wait some non-zero time T_{usage} before the lock is released to it. Once the physically prior lock holder releases the lock, it takes maximum time τ for the signal to travel to the given node and another Δ for the node to recognize the lock has been granted to it.

For the hybrid scheme analyses, we assume that fixed message lengths are used so that the Pulse Generator puts out a pulse when either its input drops low or when the maximum length of a message train has occurred (NL). We first consider the idle load situation. In the minimum execution time case, the only traffic on the Lock Channel is the given node's lock request. If the given node makes its request just as the control pulse comes to the node, it takes 2Δ for the node to recognize the pulse and then begin the transmission. If the given node is the last node (node N in our previous topology figures), then the only propagation delay is the transmission length L since the last node immediately hears its own transmission. Since this is the idle situation, the T_{usage} term is zero. It takes time D for the node to decode its message and be granted the lock. In the maximum execution time case, all N nodes are transmitting lock requests on the Lock Channel even though the one given node is the only one requesting the given lock. The control pulse arrives just after the node issues the lock request. It takes $NL + \Delta$ for the pulse to come around and be regenerated and 2Δ for the node to recognize the pulse and begin the transmission, which lasts time L. Again, the T_{usage} term is zero. There are two worst case possibilities to consider for how long it takes the node to receive its message and decode it. One possibility is that the given node is the first node (node 1 in our previous figures). Its message travels 2τ and takes time D to be decoded. The other possibility is that the given node is the last node (node N in our previous figures), which implies there is essentially no propagation delay for its message to reach itself. The node's message is then last in the message train, and it takes time ND before its own message is decoded. Therefore, the worst case time for the given node to receive its message, decode it, and be granted the lock is $\max(2\tau + D, ND)$.

For the busy lock situation under the hybrid scheme, the analysis is somewhat more complicated. In the minimum execution time case, the only traffic on the Lock Channel is from the P given nodes contending for the given lock. The control pulse arrives at the given node just as it issues the lock request, which requires 2Δ to recognize the pulse and begin the transmission, which lasts for time L. In the absolute best case no other nodes hold the lock by the time it processes its own message, so T_{usage} is zero. Now, how much time elapses before the node gets processes its own message and is granted the lock? If the given node is the physically first node, then the message must propagate distance 2τ and be decoded in time D. If the node is the last one, then its message is the last one in the message train, so there are P messages to decode for a total time of PD. Therefore, the minimum time to be granted the lock is $\min(2\tau + D, PD)$. In the maximum execution time case, all N nodes put messages on the Lock Channel, even ones not contending for the given lock. The control pulse arrives just after the given node makes the lock request. It takes $NL + \Delta$ for the pulse to come around and be regenerated plus 2Δ for the node to recognize the pulse and begin its transmission, which lasts time L. Since this is the worst case, some other node is holding the lock, and the T_{usage} time is some non-zero time. When the last prior lock holder is ready to release the lock, it could just miss the the control pulse and therefore have to wait $NL + \Delta$ for the pulse to be regenerated and arrive again. The releasing node then takes 2Δ to recognize the pulse and begin its transmission which lasts for time L. The worst case total propagation and decoding time for the releasing message is one of four possibilities. The first possibility is if the releasing node and the given node are among the physically first nodes, then the message is first in the message train, propagates distance 2τ, and is decoded in time D. The second possibility is if the releasing node is among the physically last nodes and the given node among the first nodes, then the message is last in the message train; it propagates distance τ and takes time ND to be decoded. The other two possibilities (releasing node among the first and given node among last yielding $\tau + D$; releasing and given nodes among the last yielding ND) are better than the other two and need not be considered. Therefore, the worst case total propagation and decoding time for the releasing message to take effect is $\max(2\tau + D, \tau + ND)$.

We now discuss various plots of these equations in order to understand the scale of the execution time of the lock operations. We assume that the average message length L is 25 nsec (50 bit messages at 2 Gbit/sec), the decoding time D is also 25 nsec, and Δ is 10 nsec. Figs. 48, 49, 50, and 51 show various plots for the idle lock situation, and Figs. 52, 53, 54, and 55 show various plots for the busy lock situation. As with the `WRITE-UPDATE` command execution time plots, the propagation distance τ is depicted in terms of time rather than distance.

In Figs. 48 and 49 the number of nodes has been fixed and the propagation distance allowed to vary under the idle lock situation. Both the purely optical and the hybrid schemes are shown in the figures. Since the purely optical and hybrid schemes' minimum E_L times do not depend upon either τ or N, their plots are constant (with the same numeric values) in both figures. In addition, the difference between the purely optical and hybrid schemes' minimum E_L times is just the message length and decoding times. The purely optical maximum E_L time is the same in both figures since it depends only on τ (this may be difficult to see because of the different scales of the figures). Note that the purely optical maximum E_L grows linearly with τ. The hybrid maximum E_L time is nearly constant (with different numeric values) in the two plots. This can be understood by considering

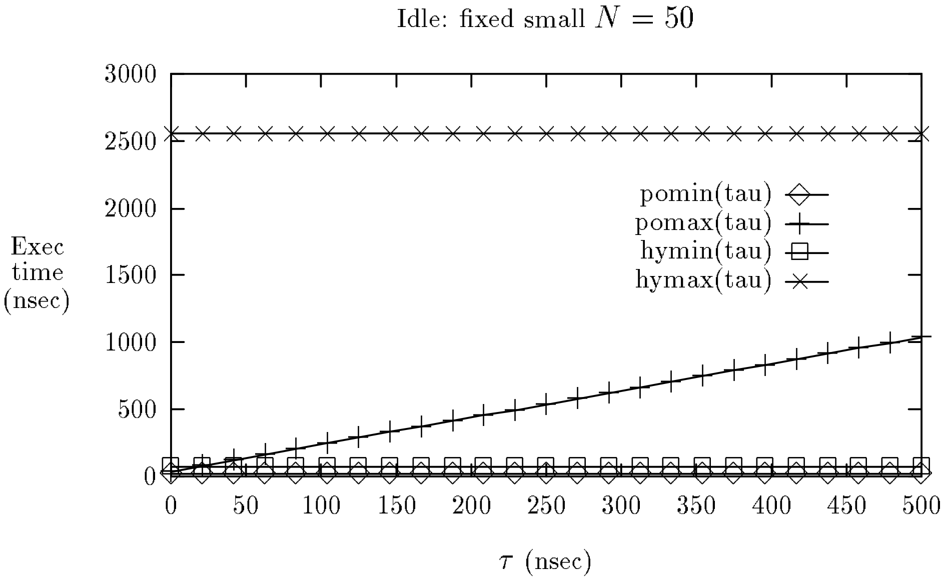

Figure 48: The minimum and maximum E_L times for idle locks for small fixed $N = 50$. The “po” and “hy” represent the purely optical and hybrid schemes respectively.

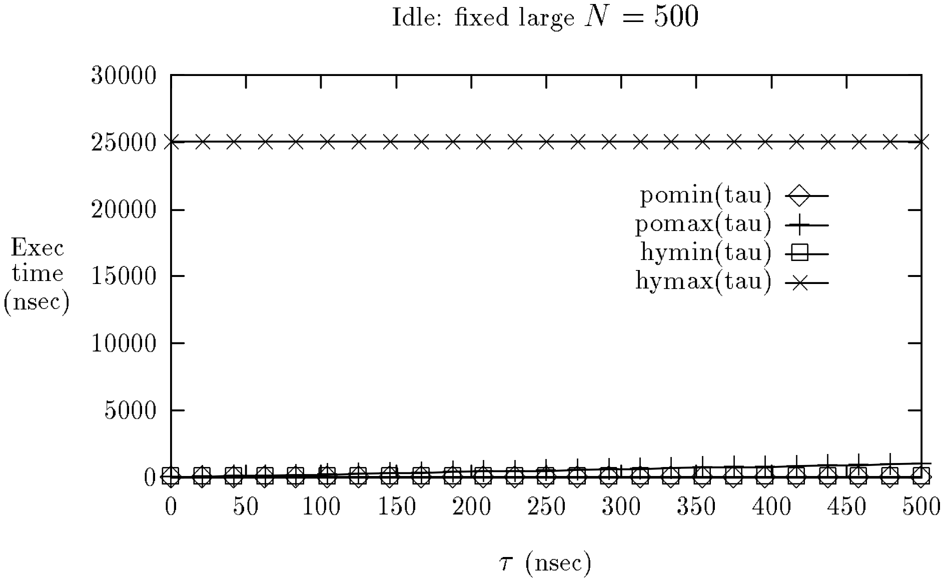

Figure 49: The minimum and maximum E_L times for idle locks for large fixed $N = 500$. The “po” and “hy” represent the purely optical and hybrid schemes respectively.

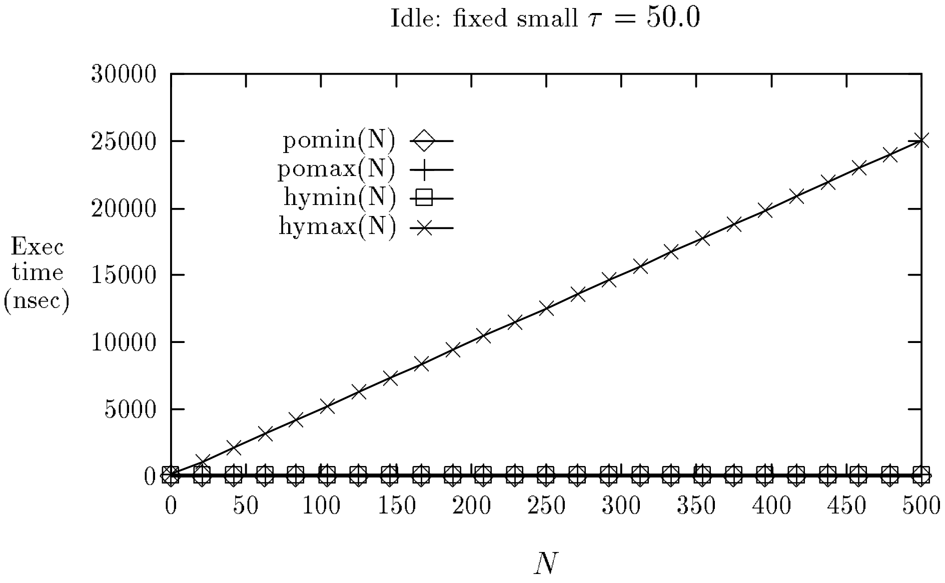

Figure 50: The minimum and maximum E_L times for idle locks for small fixed $\tau = 50.0$. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

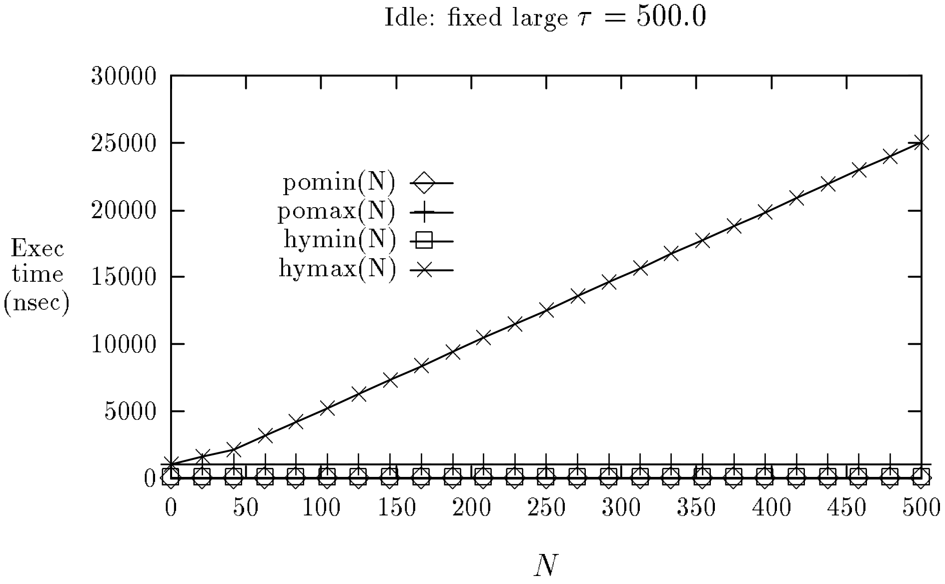

Figure 51: The minimum and maximum E_L times for idle locks for large fixed $\tau = 500.0$. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

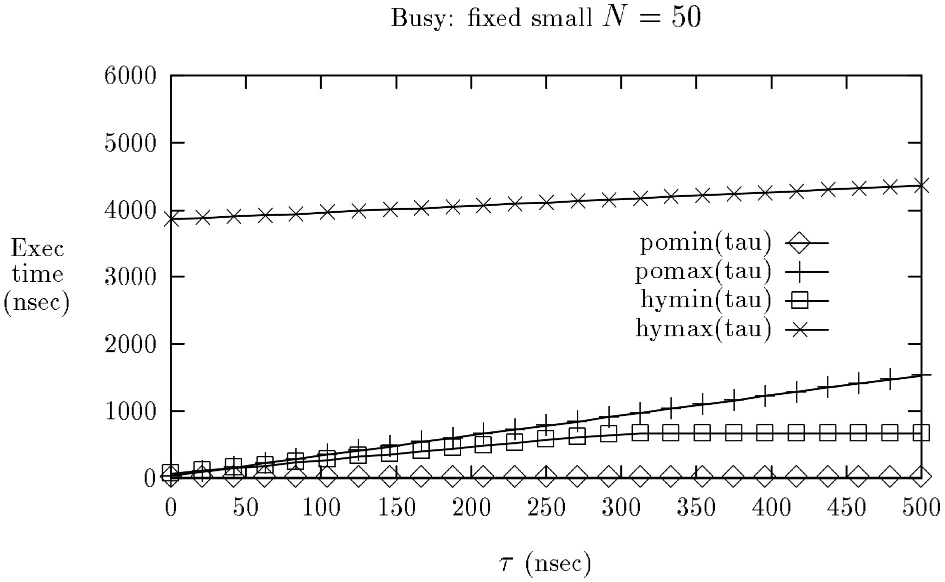

Figure 52: The minimum and maximum E_L times for busy locks for small fixed $N = 50$. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

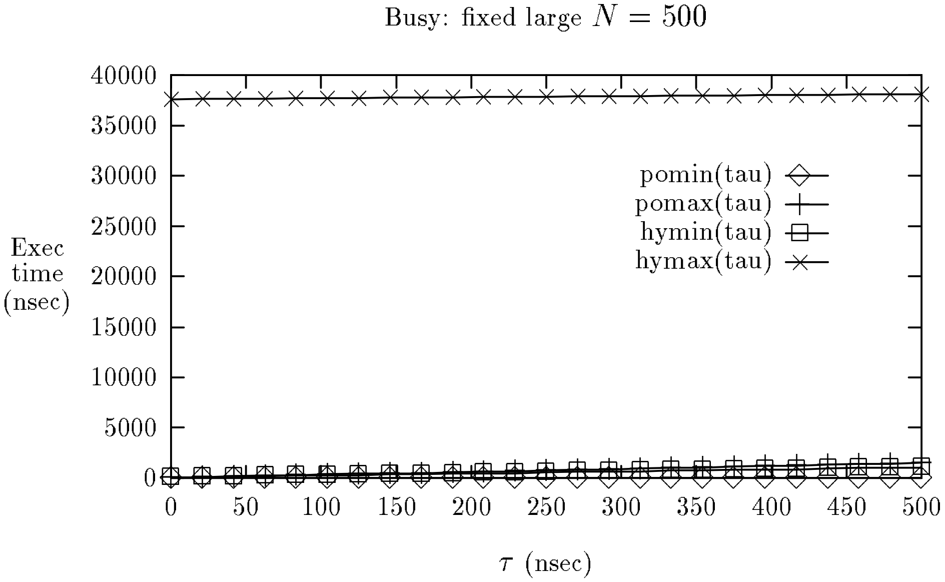

Figure 53: The minimum and maximum E_L times for busy locks for large fixed $N = 500$. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

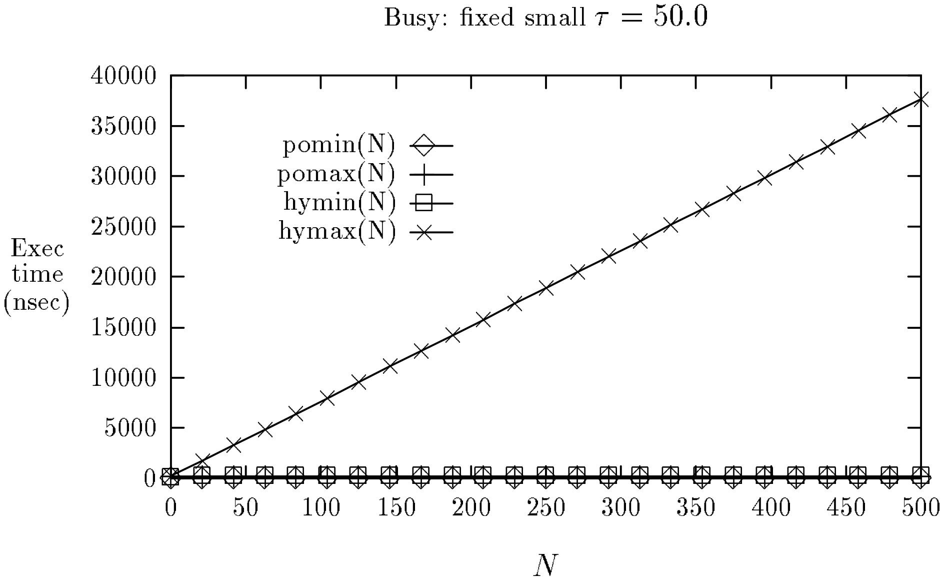

Figure 54: The minimum and maximum E_L times for busy locks for small fixed τ = 50.0. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

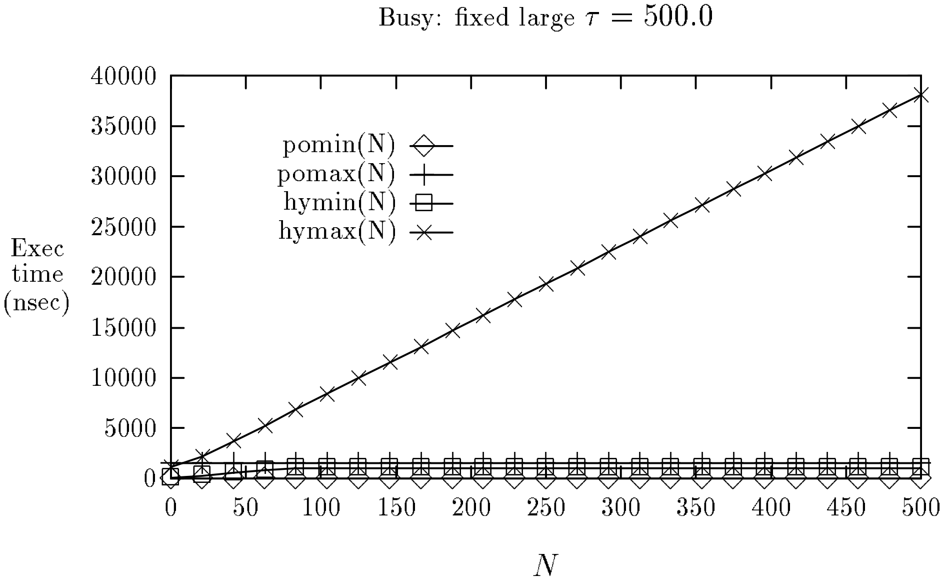

Figure 55: The minimum and maximum E_L times for busy locks for large fixed τ = 500.0. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

the qualitative meaning of the formula. The first term in the formula says that the given node has to wait one full message train before it puts its message out. The second term says that since the given node is the physically last node, its message is at the end of its message train. Thus, it takes approximately two message trains (since we assume $L = D$) for the execution time of the lock operation. For large enough N (as in both of these figures), the effect of N overshadows the effect of τ. We finally note that for the small $N = 50$, the worst case E_L is less than 3.0 μsec and that for the large $N = 500$, the worst case E_L is less than 30 μsec.

In Figs. 50 and 51 the propagation distance τ has been fixed and the number of nodes N allowed to vary under the idle lock situation. Both the purely optical and hybrid schemes are shown in the figures. Again, since the purely optical and hybrid schemes' minimum E_L times do not depend upon either τ or N, they are constant (with the same numeric values) in both figures. In the small fixed $\tau = 50.0$ plot, since τ is so small and because of the scale of the plot, the purely optical maximum E_L is nearly the same as the minimum E_L. In the large fixed $\tau = 500.0$ plot, the purely optical maximum E_L stands out more clearly. More importantly, note that with τ being fixed, the purely optical maximum is constant since it does not depend upon N. For the hybrid maximum E_L time, the strong dependence on N clearly shows in the two plots. In the large fixed $\tau = 500.0$ plot (Fig. 51), one can see a small knee in the maximum E_L time curve where, for small N, the value of τ does influence the shape; otherwise, for the two figures, the two maximum E_L time curves are identical. For the parameters considered here, the purely optical maximum time for the lock operation is approximately 1 μsec, and the hybrid maximum time is less than 30 μsec.

In Figs. 52 and 53 the number of nodes N has been fixed and the propagation distance τ allowed to vary under the busy lock situation. Both the purely optical and hybrid schemes are plotted in the figures. Since the purely optical minimum E_L is not dependent upon either τ or N, it is a constant in both figures. The hybrid minimum E_L is dependent upon either τ or P (the number of nodes contending for the given lock—assumed to be $N/2$ for these plots), depending upon the particular values of the parameters. This behavior is clearly seen in the small fixed $N = 50$ plot (Fig. 52) as the hybrid minimum E_L curve grows linearly with τ and then flattens out to a constant value. In the large fixed $N = 500$ plot (Fig. 53), since N is so large, this behavior is masked out. When considering the maximum E_L times, some value needs to be assumed for T_{usage}; rather than assign some arbitrary value to this term which is program dependent, we have assumed it has a value of zero for this analysis. Therefore, the maximum E_L curves in both figures are the lower limit on the maximum E_L time. The purely optical maximum E_L depends solely upon τ in a linear fashion; this behavior is clearly illustrated in the small fixed $N = 50$ plot (Fig. 52). Because of the scale of the large fixed $N = 500$ plot (Fig. 53, this behavior is not as clearly seen in that plot. The hybrid maximum E_L is nearly independent of τ for both the small and large N values. This can be seen by examining the qualitative meaning of the formula. If the given node just misses a message train, it has to wait one message train before putting out its message. The lock releaser could also just miss a message train, so it too has to wait one message train before putting out its message. If the lock releaser is the physically last node, then its message, being at the end of the message train, will not be decoded until last. Therefore, since we assume $L = D$, it takes approximately three message trains before the lock is granted. Finally, note that the lower limit on the execution time of the lock operations is less 5.0 μsec for the small fixed $N = 50$ case and less than 40.0 μsec for the large fixed $N = 500$ case, even for large values of τ.

In Figs. 54 and 55 the propagation distance τ has been fixed and the number of nodes N allowed to vary under the busy lock situation. Both the purely optical and hybrid schemes are shown in the two figures. Again, since the purely optical minimum E_L time is independent of both τ and N, it is a constant in both figures. And again, the hybrid minimum E_L is dependent upon either τ or P, depending upon the particular values of the parameters. For the small fixed $\tau = 50.0$ (Fig. 54), it is essentially constant over all values of N. For the large fixed $\tau = 500.0$ (Fig. 55), a slight knee in the curve's shape develops, but then the curve flattens out and becomes constant. Since the purely optical maximum E_L is not dependent upon N, then it is constant (with different numeric values) in the two figures. And since the hybrid maximum E_L exhibits such a strong dependency upon N (as explained in the previous paragraph), its plot is essentially the same in both figures even though τ is different in the two figures. Finally, consistent with the discussion in the previous paragraph, the lower limit on the purely optical maximum E_L is approximately 1.5 μsec, and on the hybrid maximum is less than 40.0 μsec for the chosen values of the parameters.

As a summary, Tables 19 and 20 give some specific execution times for idle and busy locks respectively for small and large system sizes, for small and large system scales, and for the two optical implementations.

N	τ (nsec)	Execution Time (nsec)			
		pomin	pomax	hymin	hymax
50	50	20	140	70	2555
50	500	20	1040	70	2555
500	50	20	140	70	25055
500	500	20	1040	70	25055

Table 19: Some specific Idle Lock execution times.

N	τ (nsec)	Execution Time (nsec)			
		pomin	pomax	hymin	hymax
50	50	20	190	170	3910
50	500	20	1540	670	4360
500	50	20	190	170	37660
500	500	20	1540	1070	38110

Table 20: Some specific Busy Lock execution times.

6.3 Barriers

In considering the execution time for barriers, we are interested in knowing how long it takes for some given node to recognize barrier completion if there is some number of nodes which have not yet reached the barrier. The execution time E_B for barriers (assuming fuzzy barrier semantics) is defined for a given node as:

$$E_B = T_{depend} + T_{arrive} + T_{indep} + T_{recognize}.$$

The T_{depend} term represents the execution time of code dependent upon leading up to the barrier, i.e., the code that must be executed before the `REACH-BARRIER` command is issued. Similarly, the T_{indep} term represents the execution time of code which can be executed after the barrier is reached but independent of barrier completion, i.e., code between the `REACH-BARRIER` and `BARRIER-WAIT` commands. Since these two terms are dependent upon the program's structure, we exclude these terms from our analysis. The T_{arrive} term is the time it takes for the `REACH-BARRIER` command to be issued and be seen by the nodes. The T_{arrive} term also includes the time it takes for the node to compute (as opposed to recognize) barrier completion. The $T_{recognize}$ term is the time it takes for a node to recognize barrier completion; this term can be considered constant since it is implemented in either the purely optical or hybrid scheme by checking the value of the completion register or flag. Therefore, this analysis simplifies to considering just the T_{arrive} time.

We assume that of the N participating nodes in the barrier operation, some number $N - P$ have previously reached the barrier, leaving P nodes yet to arrive. There are two situations for which to analyze the execution time of barriers, a *last* arrival and *simultaneous* arrival. A last arrival means that $P = 1$: all but one of the nodes have previously arrived at the barrier and are waiting for the last node to arrive. A simultaneous arrival implies that some arbitrary number P will arrive at the barrier simultaneously. Table 21 shows the formulas for the minimum and maximum E_B times for the last and simultaneous arrival scenarios for the different barrier schemes. The symbols in these formulas are the same as for the lock execution time formulas: Δ represents the component and circuitry delay times; τ is the propagation time from the first node to last node and vice versa; N is the number of nodes in the system; P is the number of nodes simultaneously arriving at the barrier; and L and D are respectively the average length of a message and

Scheme	*Minimum*	*Maximum*
purely optical (last)	$\Delta + \Delta$	$\Delta + 2\tau + \Delta$
purely optical (simultaneous)	$\Delta + \Delta$	$\Delta + 2\tau + \Delta$
hybrid (last)	$2\Delta + L + D$	$NL + \Delta + 2\Delta + L$ $+ \max(2\tau + D, \tau + ND)$
hybrid (simultaneous)	$2\Delta + L + PD$	$NL + \Delta + 2\Delta + L + 2\tau + ND$

Table 21: The minimum and maximum execution times for last and simultaneous arrivals for the two OBee barrier schemes.

the decoding time for a message in the hybrid scheme.

These formulas are derived as follows. Since, from the previous discussion, only the T_{arrive} component of the barrier execution time needs to be computed, each formula corresponds to that term only. We first consider the purely optical scheme with one, last arrival. Recall that a node can lower its signal on the wired-`OR` OBR without regard to what any other node is doing. Therefore, in the minimum execution time case, the node takes one Δ to lower its signal. If the node is the last node, there is essentially no propagation time of the lowered signal to itself. It takes one more Δ for the node to compute and record barrier completion. In the maximum execution time case, the node also takes one Δ to lower its signal. If the node is the first node, then it takes 2τ for the lowered signal to propagate back to itself. It then takes one more Δ for the node to compute and record barrier completion.

We next consider the simultaneous arrival case in the purely optical scheme. As before, a node can lower its signal at any time without regard to what any other node is doing. In the minimum execution time case, the nodes lower their signals in time Δ. The best case time occurs for the physically last node: its lowered signal has essentially no distance to travel before reaching itself. The node then takes one more Δ to compute and record barrier completion. In the maximum execution time case, the nodes also take one Δ to lower their signals. The worst case time occurs for the physically first node: its lowered signal travels 2τ before reaching itself. The node then takes one more Δ to compute and record barrier completion.

We now discuss the hybrid scheme analyses, starting with the one, last arrival case. In the minimum execution time case, the only traffic on the Barrier Channel is the one given node arriving at the given barrier. The node reaches the barrier just as the control protocol gives it access to the Barrier Channel; therefore, the node takes 2Δ to recognize the pulse and begin transmission. The transmission takes L time. If the given node is the last node, there is no additional propagation delay since it immediately hears its own transmission. The node then takes the decoding time D to compute and record barrier completion. In the maximum execution time case, all N nodes are putting traffic on the Barrier Channel (even though only the one given node is arriving at the given barrier). The node just misses obtaining access to the Barrier Channel, so it must wait $NL + \Delta$ for the pulse to be regenerated and come back around. The node then takes 2Δ to begin its transmission which lasts L time. There are two possibilities to consider for how long it takes the given node's message to propagate and be decoded. If the given node is the first node, then its message travels distance 2τ to reach itself and takes time D to be decoded. If the given node is the last node, the message travels distance τ to reach the first node (remember, this is the worst case) and there are N total messages to be decoded, for a time of ND. Therefore, the worst case time for barrier completion to be computed and recorded is $\max(2\tau + D, \tau + ND)$.

Finally, we have the simultaneous arrival case in the hybrid scheme. In the minimum execution time case, the only traffic on the Barrier Channel is from the P nodes simultaneously arriving at the given barrier. As the given nodes issue the `REACH-BARRIER` command, the control pulse reaches the physically first node. That node takes 2Δ to recognize the pulse and begin its transmission, which lasts for time L. If the given nodes are among the last nodes on the OBR, then there is essentially no propagation delay for them to hear their own messages. Since P nodes are arriving simultaneously, there are P messages to be decoded in time PD before barrier completion is computed and recorded. In the maximum execution time case, all N nodes are transmitting messages (regardless of whether the messages pertain to the given barrier). The control pulse just misses the given node, so it must wait $NL + \Delta$ for the pulse to be regenerated and come back to it. It then takes 2Δ to recognize the pulse and begin its transmission, which lasts time L. If the given node is the physically first node, then its message travels 2τ before reaching itself. Since all nodes are transmitting messages, it takes time ND for the messages to be decoded and barrier completion computed and recorded for the given node.

We next present various plots of these formulas in order to understand the scale of the execution time of the barrier operation. As with the lock operation plots, we assume the average message length L is 25 nsec, the average decoding time D is 25 nsec, and Δ is 10 nsec. Figs. 56, 57, 58, and 59 depict the equations for the last arrival case; Figs. 60, 61, 62, and 63 show the plots for the simultaneous arrival case. As in the cache coherency and lock figures, the propagation distance τ is expressed in terms of time.

In Figs. 56 and 57 the number of nodes N has been fixed and the propagation distance τ allowed to vary under the last arrival situation. Both the purely optical and hybrid schemes are shown. Both the purely optical and hybrid minimum E_B times are independent of τ and N, so their plots are constants (with the same numeric values) in both figures. The purely optical maximum E_B is linearly dependent solely upon τ. This behavior is clearly shown in the small fixed $N = 50$ plot (Fig. 56). In the large fixed $N = 500$ plot (Fig. 57), the behavior is masked because of the scale of the plot. The hybrid maximum E_B time is not strongly dependent upon τ. This behavior can be understood by considering the qualitative meaning of the equation. The given node must wait one message train before putting its message on the OBR. If the given node is the physically last node, then its message is the last in its message train. Thus, it takes approximately two message trains for its message to take effect. We note that for the small fixed $N = 50$, the worst case E_B is less than 3.5 μsec; for the large fixed $N = 500$, the worst case E_B is less than 30.0 μsec.

In Figs. 58 and 59 the propagation distance τ has been fixed and the number of nodes N allowed to vary under the last arrival situation. Both the purely optical and hybrid schemes are shown in the figures. Again, since both the purely optical and hybrid minimum E_B times are independent of both τ and N, their plots are constant (with the same numeric value) in both figures. Since the purely optical maximum E_B time is independent of N, its curve is constant (though with

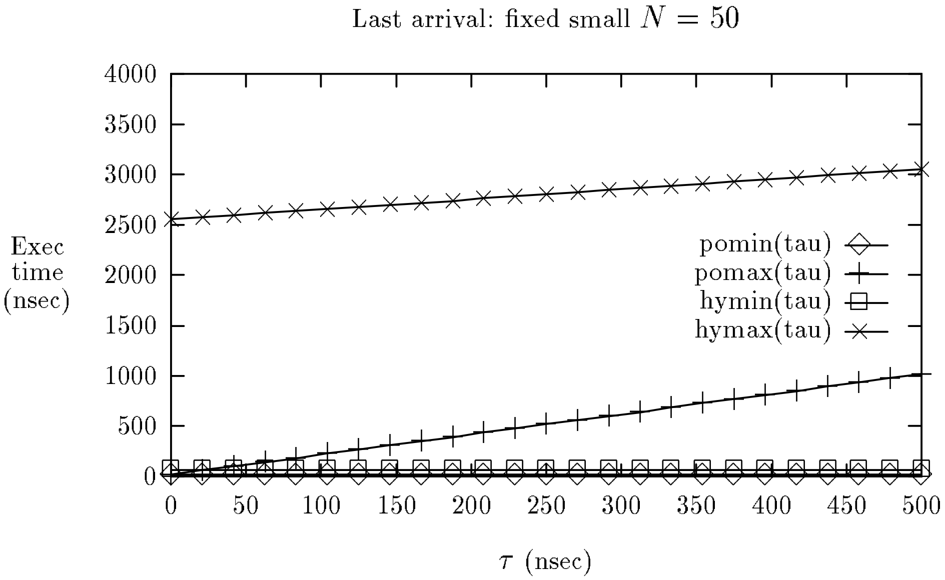

Figure 56: The minimum and maximum E_B times for last arrival for small fixed $N = 50$. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

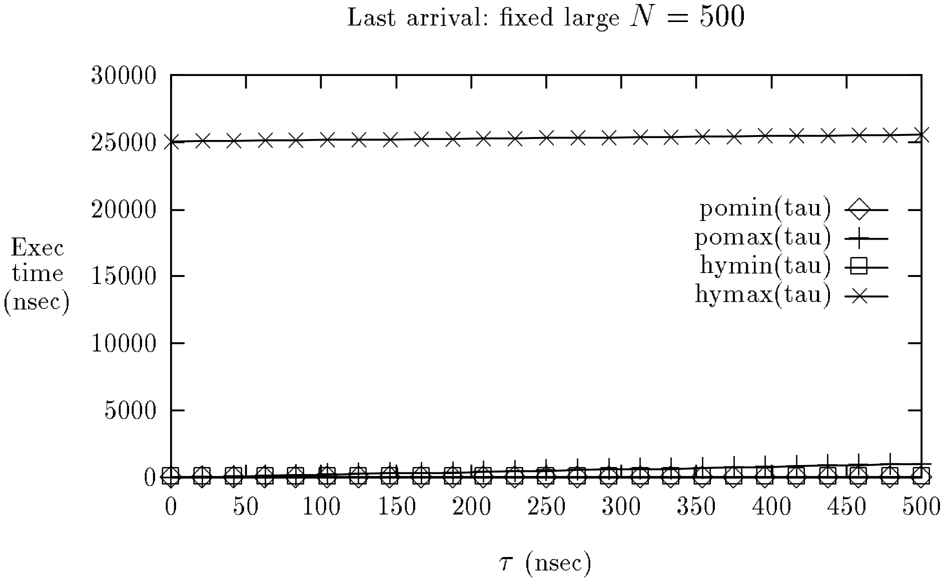

Figure 57: The minimum and maximum E_B times for last arrival for large fixed $N = 500$. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

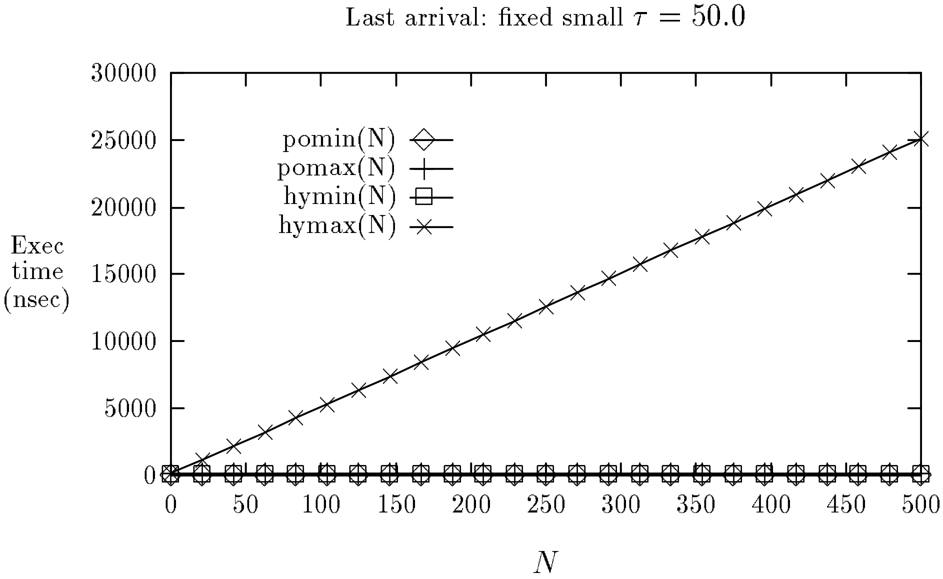

Figure 58: The minimum and maximum E_B times for last arrival for small fixed $\tau = 50.0$. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

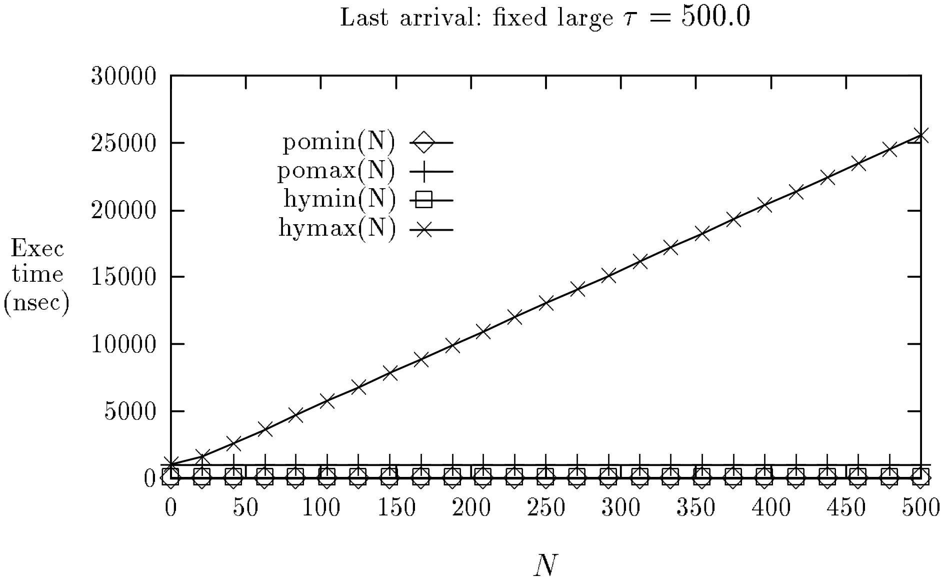

Figure 59: The minimum and maximum E_B times for last arrival for large fixed $\tau = 500.0$. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

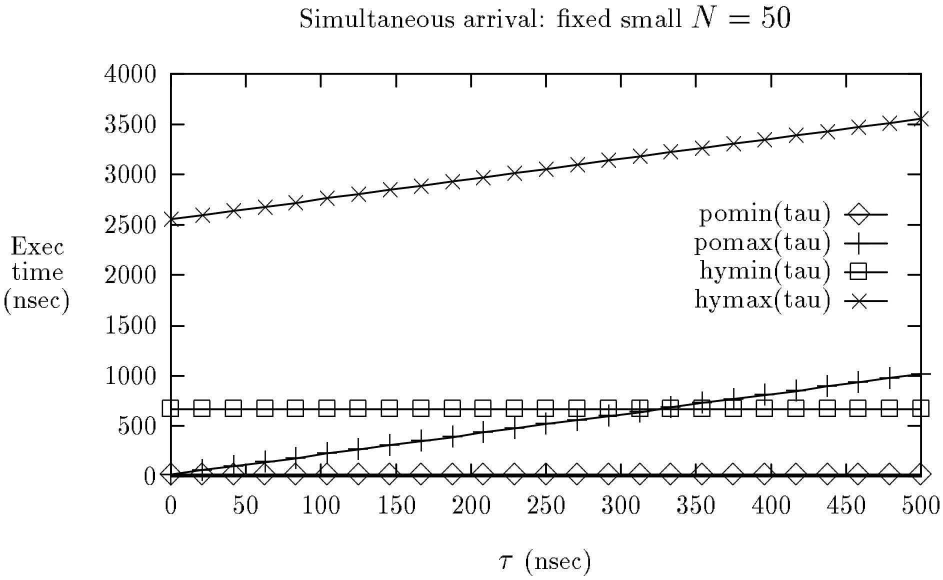

Figure 60: The minimum and maximum E_B times for simultaneous arrival for small fixed $N = 50$. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

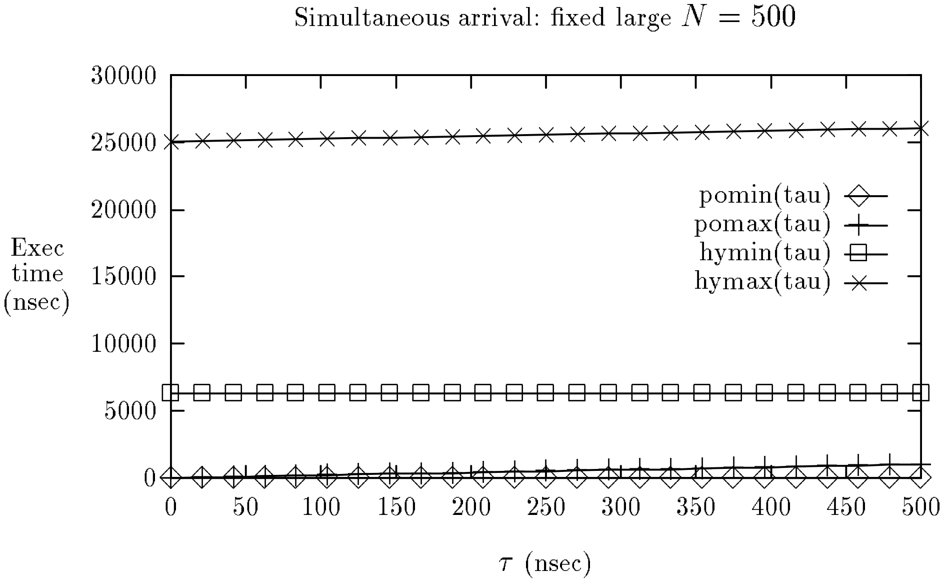

Figure 61: The minimum and maximum E_B times for simultaneous arrival for large fixed $N = 500$. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

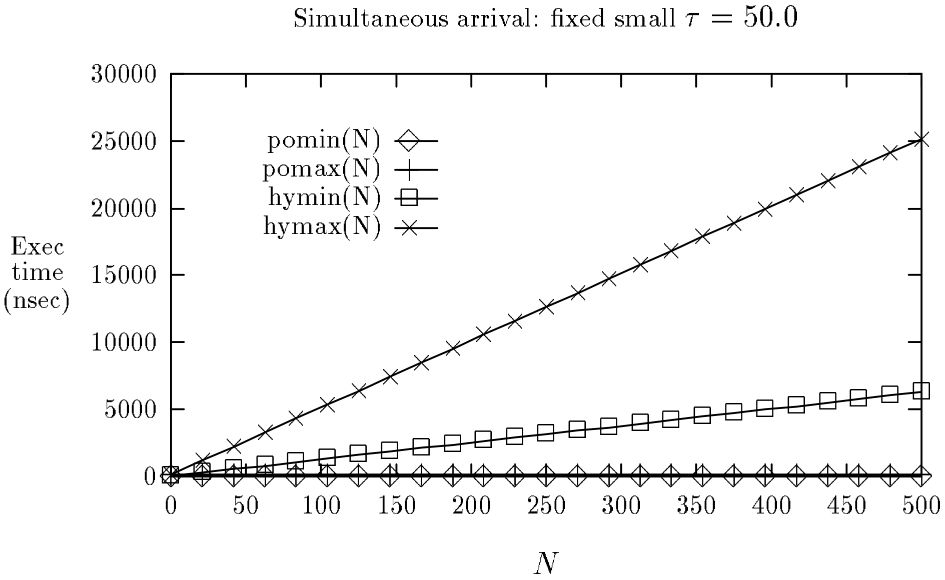

Figure 62: The minimum and maximum E_B times for simultaneous arrival for small fixed $\tau = 50.0$. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

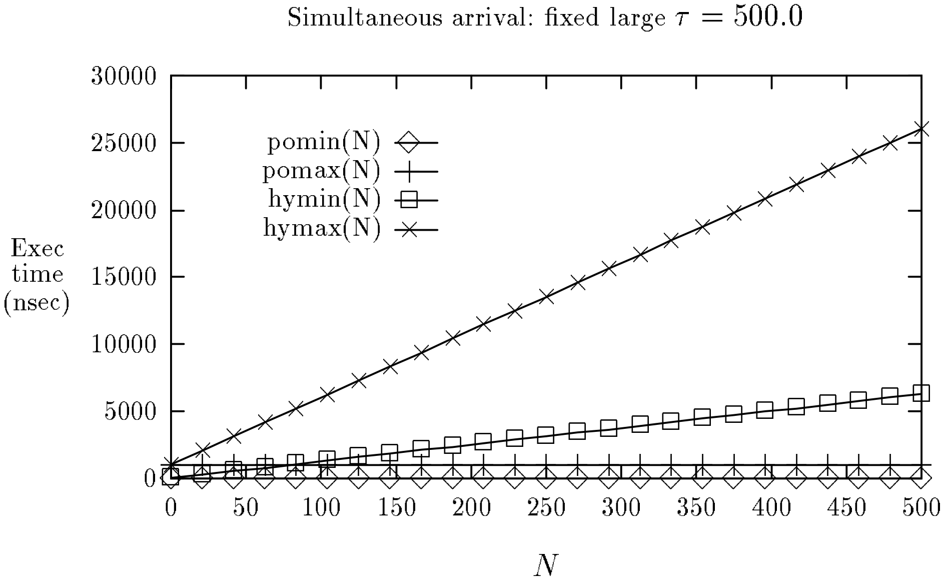

Figure 63: The minimum and maximum E_B times for simultaneous arrival for large fixed $\tau = 500.0$. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

different numeric values) in both figures. The two figures show the strong linear dependency that the hybrid maximum E_B time has on N. For the small fixed $\tau = 50.$ (Fig. 58), the effect of the value of τ is nearly negligible. For the large fixed $\tau = 500.0$ (Fig. 59), there is a slight knee in the curve for small N where the value of τ has some influence. We note that for the chosen parameters, the worst case purely optical E_B is approximately 1 μsec and the worst case hybrid E_B is less than 30.0 μsec.

In Figs. 60 and 61 the propagation distance τ has been fixed and the number of nodes N allowed to vary under the simultaneous arrival situation. Both the purely optical and hybrid schemes are shown. The purely optical minimum E_B time is independent of both τ and N, so its curves are constant (with the same numeric value) and identical to the plots in the minimum E_B case in both figures. The hybrid minimum E_B time depends solely upon the number of nodes P (assumed to be $N/2$ in these figures) simultaneously arriving at the barrier. Therefore, its curves are constant (with different numeric values) in both plots. The purely optical maximum E_B is dependent only upon τ (the equation has the same value as for the minimum E_B); its linear growth in τ is shown in both figures (though masked in the large fixed $N = 500$ plot because of its scale). The hybrid maximum E_B time has some dependency upon τ, but it is nearly constant because of its stronger dependency upon N. As with the hybrid minimum E_B time, it takes approximately two message trains for the messages to take effect. For the chosen parameters, the worst case E_B is less than 4.0 μsec for the small fixed $N = 50$ case and less than 30.0 μsec for the large fixed $N = 500$ case.

Figs. 62 and 63 show the plots for the propagation distance τ being fixed

N	τ (nsec)	Execution Time (nsec)			
		pomin	pomax	hymin	hymax
50	50	20	120	70	2605
50	500	20	1020	70	3055
500	50	20	120	70	25105
500	500	20	1020	70	25555

Table 22: Some specific Barrier Last Arrival execution times.

and the number of nodes N being allowed to vary under the simultaneous arrival situation. Both the purely optical and hybrid schemes are shown. Again, since the purely optical minimum E_B time is independent of both τ and N, its curves are constant (with the same numeric value) in both figures. The hybrid minimum E_B time's linear dependency on P is clearly shown in both figures; the curves are the same in both figures since it has no dependency on τ. Since the purely optical maximum E_B depends upon only τ, then its curves are constant (with different numeric values) in both figures. The strong linear dependency upon N of the hybrid maximum E_B is illustrated in both figures. The slopes of the curves in both figures are the same; the y-intercepts are slightly different because of the different values of τ. For the chosen parameters, the worst case purely optical E_B is approximately 1 μsec, and the worst case hybrid E_B is less than 30.0 μsec.

As a summary, Tables 22 and 23 give some specific execution times for the last and simultaneous arrival cases respectively for small and large system sizes, for small and large system scales, and for the two optical implementations.

N	τ (nsec)	Execution Time (nsec)			
		pomin	pomax	hymin	hymax
50	50	20	120	670	2655
50	500	20	1020	670	3555
500	50	20	120	6295	25155
500	500	20	1020	6295	26055

Table 23: Some specific Barrier Simultaneous Arrival execution times.

6.4 F&OP

In considering the execution time for the F&OP command, we want to know how long it takes for a node to receive its value of `V` after issuing the command. The execution time E_F for the F&OP command is defined for a given node as:

$$E_F = T_{request} + T_{compute}.$$

The $T_{request}$ term represents how long it takes for the node to place its F&OP request on the OBR. Once the request has been made, the $T_{compute}$ term denotes how long it takes before the node computes the value of `V` it receives from its F&OP command.

There are two situations for which to analyze the F&OP command, the *single* and *multiple* command situations. In the single command situation, only one node issues the F&OP command for a particular target `V` whereas in the multiple command situation some number of nodes P are issuing the F&OP command for the target `V`. Table 24 shows the formulas for the minimum and maximum E_F times for the single and multiple situations for the different F&OP schemes. As with the lock and barrier analyses, the following symbols are used: Δ is the component and

Scheme	*Minimum*	*Maximum*
purely optical (single)	$\Delta + \Delta$	$(2\tau + \Delta + \Delta) + \Delta$
purely optical (multiple)	$\Delta + \Delta$	$(\tau + \Delta + \Delta) + (\tau + P\Delta)$
hybrid (single)	$(2\Delta + L) + D$	$[NL + \Delta + 2\Delta + L]$ $+\max(2\tau + D, ND)$
hybrid (multiple)	$(2\Delta + L) + \min(2\tau + D, PD)$	$[NL + \Delta + 2\Delta + L]$ $+\max(2\tau + D, ND)$

Table 24: The minimum and maximum execution times for the single and multiple command situations for the two OBee F&OP schemes.

circuitry delay times; τ represents the propagation delay time from the first node to the last node and vice versa; N is the number of nodes in the system; P is the number of nodes participating in the F&OP operation in the hybrid scheme; and L and D are the average message length and decoding times respectively in the hybrid scheme.

These formulas are derived as follows. Note that each formula has two terms (a parenthetical expression counts as one term) that directly correspond with the two terms in the definition of E_F. We first consider the purely optical scheme under the single command situation. In the minimum execution time case, the control pulse arrives just as the given node issues the command, so it recognizes the pulse and raises its signal in time Δ. Since the given node is the only one participating in the F&OP command, it immediately sees the signal drop and computes its value of `V` in one more Δ. In the maximum execution time case, the control pulse has just passed by the given node as it issues the command; therefore, the node waits $2\tau + \Delta$ for the pulse to be regenerated and come back to it and then takes one more Δ to recognize the pulse and raise its signal. Again, since the given node is the only one participating in the F&OP command, it immediately sees the signal drop and computes its value of `V` in one more Δ.

Next we consider the multiple command situation under the purely optical scheme. In the minimum execution time case, the given node is the physically first node in the group of nodes issuing the F&OP commands. The control pulse for the wired-`OR` OBR arrives just as the given node makes its request, so it takes one Δ for the node to recognize the pulse and raise its signal. Since the given node is the physically first node, it immediately sees the pulse drop and computes its `V` in one more Δ. In the maximum execution time case, the given node is the physically last node in the group of nodes issuing the F&OP commands. The node issues the command just after the control pulse goes past it. Thus, the physically first node in the group receives the new pulse after time $\tau + \Delta$ and takes one more Δ to recognize the pulse and raise its signal. The pulse arrives at the given node after time τ. Since the given node is the physically last in the group, it will have P signal drops to process before obtaining its value of `V`, for a time $P\Delta$.

We now consider the hybrid scheme analyses, starting with the single command situation. In the minimum execution time case, the only traffic on the F&OP Channel is from the one given node issuing its F&OP command. The control pulse arrives at the given node just as it issues the F&OP command, so the node takes 2Δ to recognize the pulse and begin its transmission, which lasts time L. If the given node is the physically last node, then it immediately hears its own message; it decodes its message in time D to obtain its value of `V`. In the maximum execution time case, all N nodes are putting messages on the F&OP Channel, even though there is only one given node issuing the given F&OP command. The given node issues the F&OP command just after the control pulse passes by it; therefore, the node waits $NL + \Delta$ before the pulse is regenerated and comes back by it. The node then takes 2Δ to recognize the pulse and begin its transmission, which lasts for time L. How long does it take for the given node to gets its message and compute its value of `V`? There are two possibilities. First, the given node is the physically first node, so its message travels distance 2τ and is decoded in time D. Second, the given node is the physically last node, so its message propagates essentially zero distance to reach itself, and there are N messages to decode, for a time of ND. Therefore, the worst case for the given node takes $\max(2\tau + D, ND)$ time for its message to reach itself and be decoded to yield its value of `V`.

Finally, we consider the multiple command situation under the hybrid scheme. In the minimum execution time case, the traffic on the F&OP Channel consists solely of the messages from the P nodes participating in the F&OP commands for the sole target `V`. The control pulse reaches the physically first node in the group just as it issues the F&OP command, so it takes 2Δ to recognize the pulse and begin its transmission which takes time L. How long does it take for the given node to receive and process its own message? If the given node is the physically first on the OBR, then its message travels 2τ to reach itself and is decoded in time D. If the given node is physically last on the OBR, then there is no propagation distance to itself, and there are P messages to decode, for time PD. Therefore, the minimum time for the node to compute its value of `V` is $\min(2\tau + D, PD)$. In the maximum execution time case, all N nodes are transmitting messages on the F&OP Channel, even nodes not participating in the given F&OP target `V`. The control pulse just misses the given node, so it must wait $NL + \Delta$ for the pulse to be regenerated and come back to it. It then takes time 2Δ to recognize the pulse and begin its transmission, which lasts for time L. Now, how long does it take for the given node to receive and process its own message? If the given node is the physically first on the OBR, then its message travels 2τ to reach itself and is decoded in time D. If the given node is physically last on the OBR, then there is no propagation distance to itself, and there are N messages to decode, for time ND. Therefore, the maximum time for the given node to compute its value of `V` is $\max(2\tau + D, ND)$.

We next discuss various plots of these equations in order to understand the raw performance of the F&OP command. As in the previous analyses, we assume that the average message length L is 25 nsec, the decoding time D is 25 nsec, and Δ is 10 nsec. Figs. 64, 65, 66, and 67 depict the equations for the single command situation, and Figs. 68, 69, 70, and 71 show the plots for the multiple command situation. And, as before, the propagation distance τ is expressed in terms of time.

In Figs. 64 and 65 the number of nodes N has been fixed and the propagation distance τ allowed to vary under the single command situation. Both the purely optical and hybrid schemes are shown. Both the purely optical and hybrid minimum E_F times are independent of both τ and N; therefore, their curves are constant (with the same numeric value) in both figures. The purely optical maximum E_F time is solely dependent in a linear fashion upon τ, so its curves have the same slope in both figures (though the different scales distort the slope). The hybrid maximum E_F time is nearly independent of τ. This behavior can be understood by considering qualitatively what each term in the formula means. The first

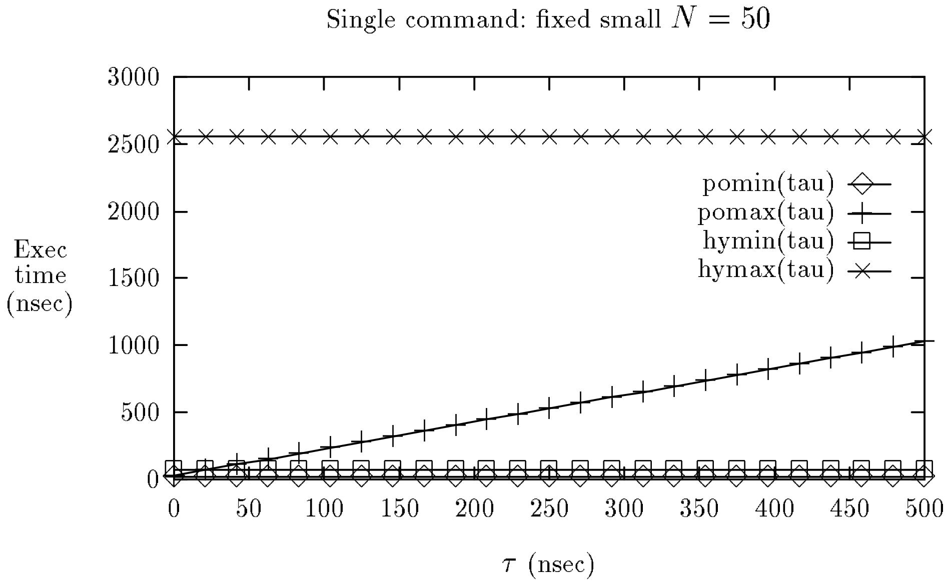

Figure 64: The minimum and maximum E_F times for single command small fixed $N = 50$. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

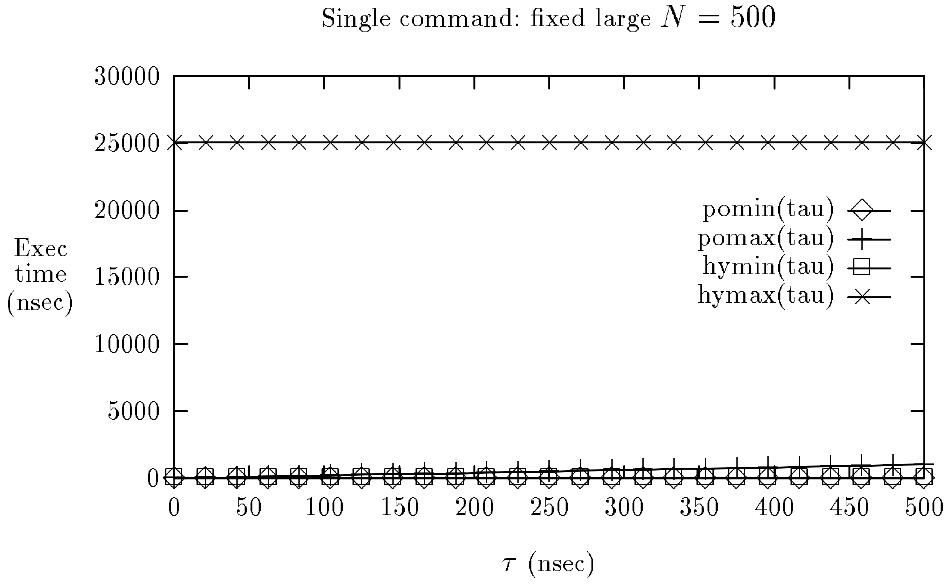

Figure 65: The minimum and maximum E_F times for single command large fixed $N = 500$. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

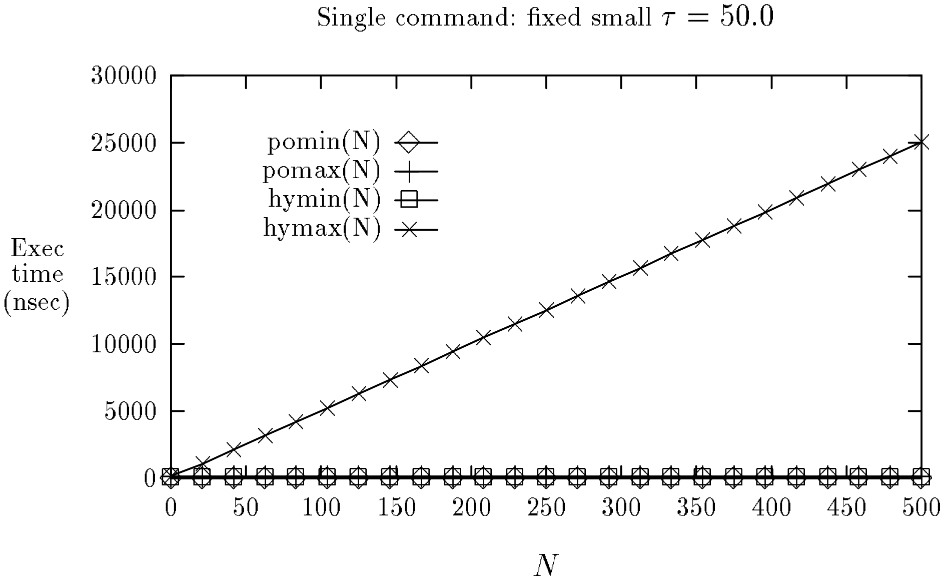

Figure 66: The minimum and maximum E_F times for single command small fixed $\tau = 50.0$. The “po” and “hy” represent the purely optical and hybrid schemes respectively.

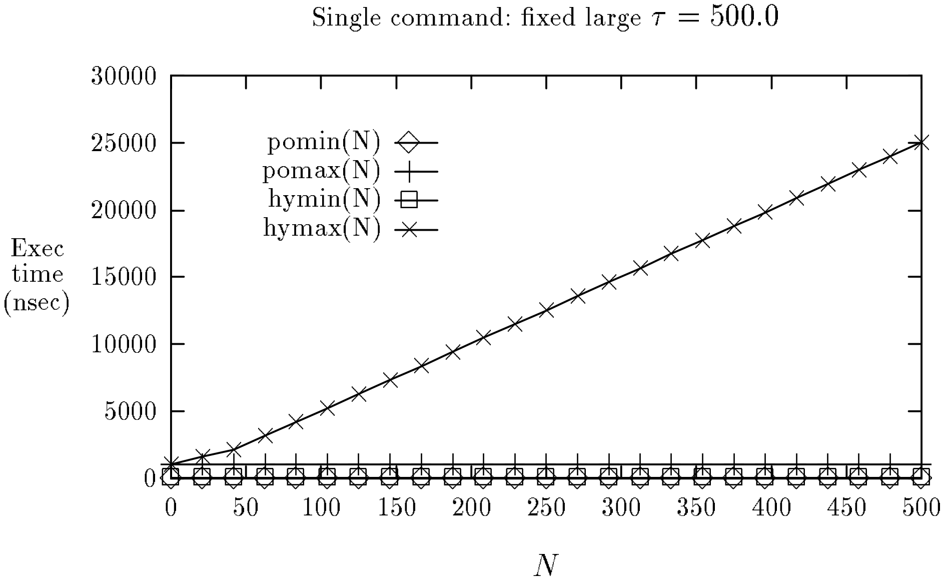

Figure 67: The minimum and maximum E_F times for single command large fixed $\tau = 500.0$. The “po” and “hy” represent the purely optical and hybrid schemes respectively.

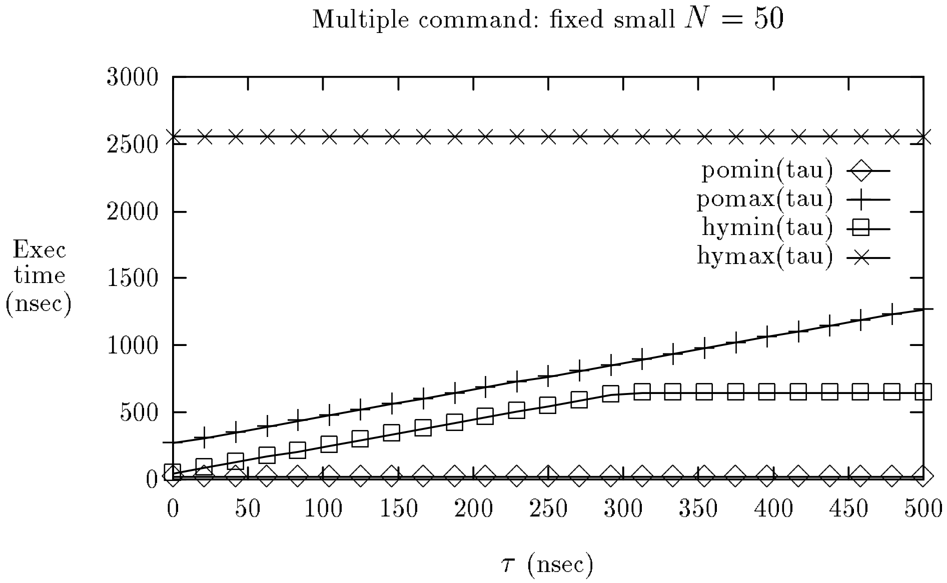

Figure 68: The minimum and maximum E_F times for multiple command arrival for small fixed $N = 50$. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

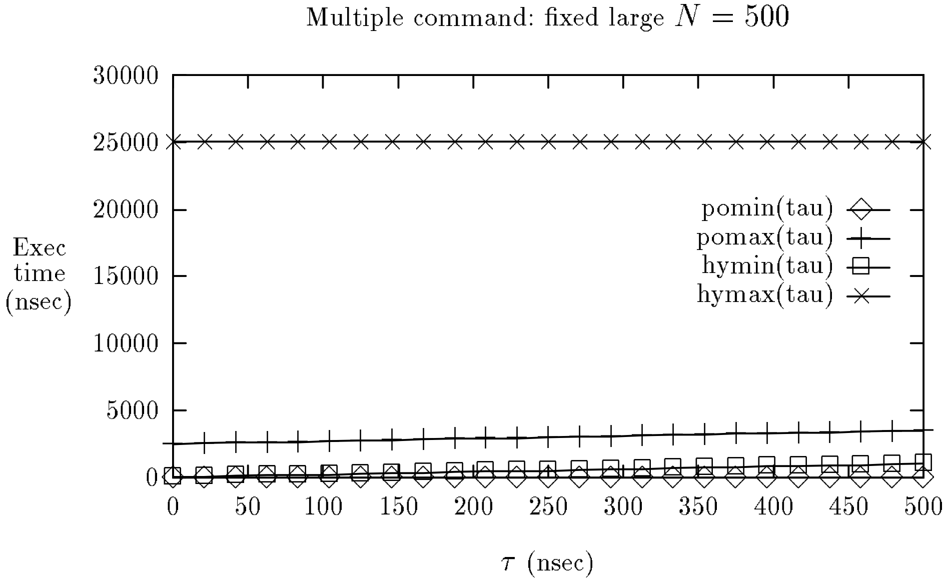

Figure 69: The minimum and maximum E_F times for multiple command arrival for large fixed $N = 500$. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

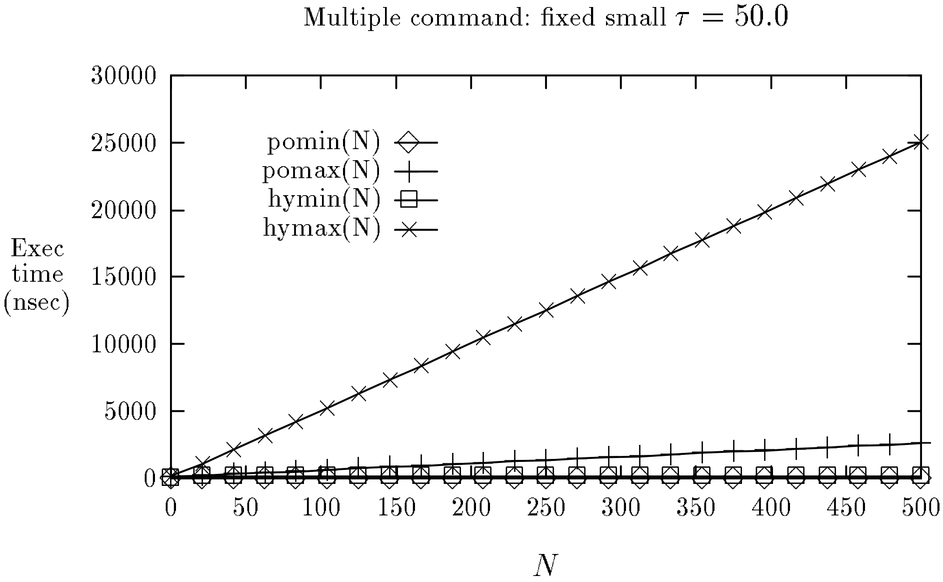

Figure 70: The minimum and maximum E_F times for multiple command arrival for small fixed $\tau = 50.0$. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

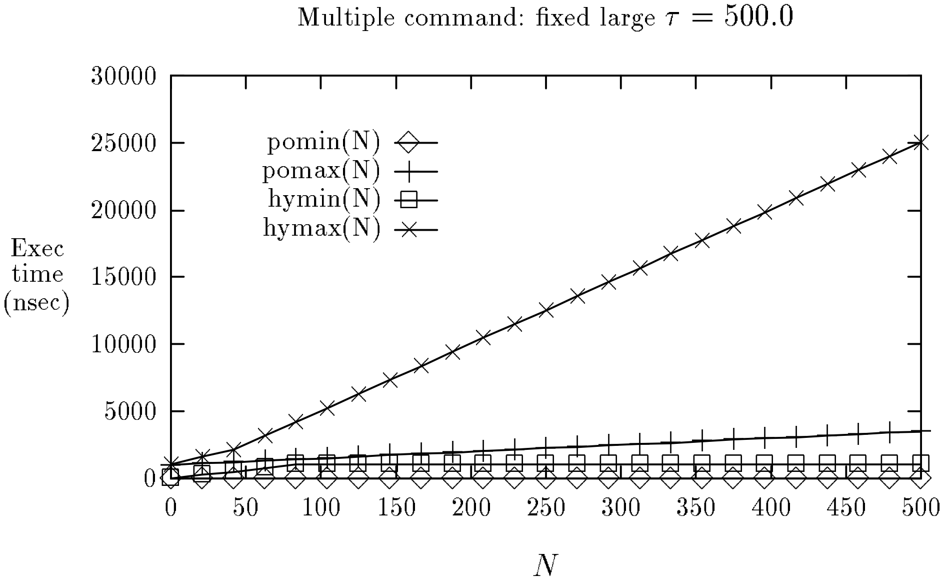

Figure 71: The minimum and maximum E_F times for multiple command arrival for large fixed $\tau = 500.0$. The "po" and "hy" represent the purely optical and hybrid schemes respectively.

term says that the given node waits one message train to put out its message. The second term says that if the given node is the physically last node, then its message is at the end of its message train. Thus, it takes approximately two message trains for the message to take effect. For large enough N (as in these figures), τ has little influence upon the execution time. For the chosen parameters, the maximum E_F for the small fixed $N = 50$ case is less than 3.0 μsec and for the large fixed $N = 500$ case less than 30.0 μsec.

Figs. 66 and 67 show the single command equations for the number of nodes N having been fixed and the propagation distance τ having been allowed to vary. Both the purely optical and hybrid schemes are shown. Again, since the purely optical and hybrid minimum E_F times are independent of both τ and N, their curves are constant (with the same numeric value) in both figures. The purely optical maximum E_F time is independent of N, so its curves are constant (with different numeric values) in the two figures. The hybrid maximum E_F is strongly dependent on N, as illustrated in the two plots. For large fixed $\tau = 500.0$ (Fig. 67), the value of τ does have some influence on the execution time as can be seen in the small knee for small values of N in that figure. Once N is large enough, the hybrid maximum E_F is essentially linearly dependent on N. For these parameters, the worst case purely optical E_F is approximately 1 μsec and less than 30.0 μsec for the worst case hybrid E_F.

In Figs. 68 and 69 the number of nodes N has been fixed and the propagation distance τ allowed to vary under the multiple command situation. Both the purely optical and hybrid schemes are plotted. The purely optical minimum E_F is independent of both τ and N so its curve is constant (with the same numeric value) in both figures. The hybrid minimum E_F time, for small fixed $N = 50$ (Fig. 68), is initially linearly dependent upon the number of nodes P (assumed to be $N/2$) simultaneously issuing the F&OP commands; once τ is large enough, it dominates this equation. This effect is not very prominent in the large fixed $N = 500$ plot (Fig. 69). The purely optical maximum E_F time is linearly dependent upon both τ and P. For the small fixed $N = 50$ plot (Fig. 68), the value of τ has more effect than N, and vice versa for the large fixed $N = 500$ plot (Fig. 69). The two figures illustrate how the maximum hybrid E_F is nearly constant for a fixed N; the qualitative explanation of the equation's two terms, presented previously, explain this behavior. For the chosen parameters, the worst case E_F is less than 3.0 μsec for the small fixed $N = 50$ case and less than 30.0 μsec for the large fixed $N = 500$ case.

Figs. 70 and 71 show the simultaneous command equations wherein the propagation distance τ has been fixed and the number of nodes N has been been allowed to vary. Both the purely optical and hybrid schemes are shown. Again, the purely optical minimum E_F time is independent of both τ and N, so its curves are constant (with the same numeric value) in both figures. The hybrid minimum E_F time is essentially constant, depending upon the particular values of τ or P chosen. For small fixed $\tau = 50.0$ (Fig. 70), it is constant for all but the smallest values of N; for large fixed $\tau = 500.0$ (Fig. 71), it is linearly dependent upon P for the initial values of P (as the knee in the curve shows) and then becomes constant for large enough P. The purely optical maximum E_F depends upon both τ and P. The curves in the two figures have the same slope (since τ has been fixed) with different y-intercepts (because of the different values of τ). The hybrid maximum

N	τ (nsec)	Execution Time (nsec)			
		pomin	pomax	hymin	hymax
50	50	20	130	70	2555
50	500	20	1030	70	2555
500	50	20	130	70	25055
500	500	20	1030	70	25055

Table 25: Some specific Single Command F&OP execution times.

N	τ (nsec)	Execution Time (nsec)			
		pomin	pomax	hymin	hymax
50	50	20	370	170	2555
50	500	20	1270	670	2555
500	50	20	2620	170	25055
500	500	20	3520	1070	25055

Table 26: Some specific Multiple Command F&OP execution times.

E_F time has a strong dependence upon N as illustrated in the two figures. For the large fixed $\tau = 500.0$ (Fig. 71), the value of τ has some influence for small N (as seen by the knee in the curve), but this influence is quickly dwarfed by the dependency on N. For large enough N, the curves are essentially the same in the two figures.

As a summary, Tables 25 and 26 give some specific execution times for single command and multiple command F&OP respectively for small and large system sizes, for small and large system scales, and for the two optical implementations.

Chapter 7

Conclusions and Future Work

7.1 Concluding Remarks

This thesis has been predicated on the assumption that optical waveguide communications technology will become more and more prevalent in smaller and smaller scale environments. Given this assumption, the research question has been whether the properties of optical waveguides can provide more than just high speed data transmission in general purpose parallel computing systems formed from electronic computing elements interconnected with optical waveguides. The answer to this question has provided several contributions.

First, we developed new combinations of optical waveguide topologies and medium access protocols which form several variants of a physical optical communications architecture we call the Optical Broadcast Ring (OBR). The OBR architecture is structured such that optical signals and messages can be both inherently ordered among the nodes and broadcast to all the nodes. OBRs with these two properties form the building blocks for supporting certain general purpose parallel computing needs.

Second, we showed how the OBR building blocks can provide direct hardware support for a specific implementation derivative of an earlier proposed abstract distributed shared memory multiprocessor design called Beehive. This specific implementation derivative of Beehive that we developed is called OBee (for optical Beehive). We described how the OBee design uses OBRs to provide simple hardware support for Beehive's reader initiated cache coherency protocol. We showed how two different types of OBRs can be used to build two distinct implementations of Beehive's only synchronization primitive, locks. We also presented OBR based implementations of two more useful synchronization primitives (not present in the Beehive design), barriers and F&OP. We described two distinct implementations of barriers based upon two different OBRs and described two distinct implementations of F&OP based upon two different OBRs.

Third, we presented an analytical evaluation of the raw performance characteristics of each of the four command features (cache coherency, locks, barriers, and F&OP). This evaluation was composed of the best and worst case execution times of the relevant commands. After deriving the equations representing these two performance metrics under varying conditions, we showed and discussed various plots of the equations. These plots gave quantitative meaning to the equations under small and large system sizes and small and large interconnection distances. For a large system size of 500 nodes connected over the distance of a building (100 m), the worst case execution times were on the order of several tens of μsec.

Thus, we have answered our research question affirmatively: optical waveguides can be structured so that they provide, in addition to the standard high speed data transmissions, direct support for useful general purpose parallel computing features. In addition, our analytical evaluation showed that these features will execute on large systems interconnected over building size distances in a reasonable amount of time.

7.2 Further Questions

The questions that any research work answers are important; just as important are the new (unanswered) questions that arise from the original question and its subsequent answers. We briefly describe here several such issues.

One question is the OBR architecture. Is there another configuration (either topology or medium access protocol) which provides better service? For example, in the current configuration, when a signal (or message) travels from node N back to the first node, could that signal be used productively to trigger access on a parallel OBR on which the signals travel in the opposite direction from the original OBR? Such an OBR architecture would imply that half the time the nodes are physically ordered from 1 to N and half the time from N to 1.

A second question concerns the implicit assumption throughout this thesis that one task is mapped to a node and runs to completion on that node. How do the OBR architectures, associated data structures, and monitoring algorithms (as applicable) need to be modified if multiple tasks are mapped to one node?

A third issue regards how the analytical evaluation of the command primitives' raw performance is translated into measuring the performance of "real" parallel programs. The use of execution-driven system simulations would allow one to examine the behavior of real programs. Real programs' behavior need to be evaluated to answer questions such as: How many OBRs are needed? Which type of OBR architecture (wired-`OR` versus explicit message) is more appropriate for a given command primitive? If multiple explicit message OBRs are allocated for a given command primitive, how is the message traffic to be allocated effectively? Does the physical architecture of the OBR encourage a particular mapping strategy of tasks to nodes? As technologically dependent parameters change (e.g., optical waveguide data transmission speeds, CPU computing performance, multiple CPUs per chip, increasing on-chip cache sizes, increasing memory sizes, decreasing memory and cache speeds, etc.), how are the number, type, and allocation of OBRs changed? Another facet of the technologically dependent parameters is the appropriate computation granularity of programs running on systems employing OBRs. As an example, Fig. 72 shows the granularity ratio of computation to the lock acquisition time versus RISC CPU processor speed. A ratio of 1.0 implies that one instruction can be executed in one lock acquisition time. We assume that the RISC CPU can execute one instruction per clock cycle and that, on average, the lock acquisition time is 100 nsec. As a point of reference in the figure, the DEC Alpha chip currently (in 1992) runs at 200 MHz, equivalent to a granularity ratio of 20.0. This result implies that (depending on how "real" programs behave as determined by the simulations mentioned above) systems using OBRs might be able to use a small computation granularity even over large system sizes and/or scales.

The fourth question asks what other architectural features would be useful. For example, would data prefetching or code and data migration from one

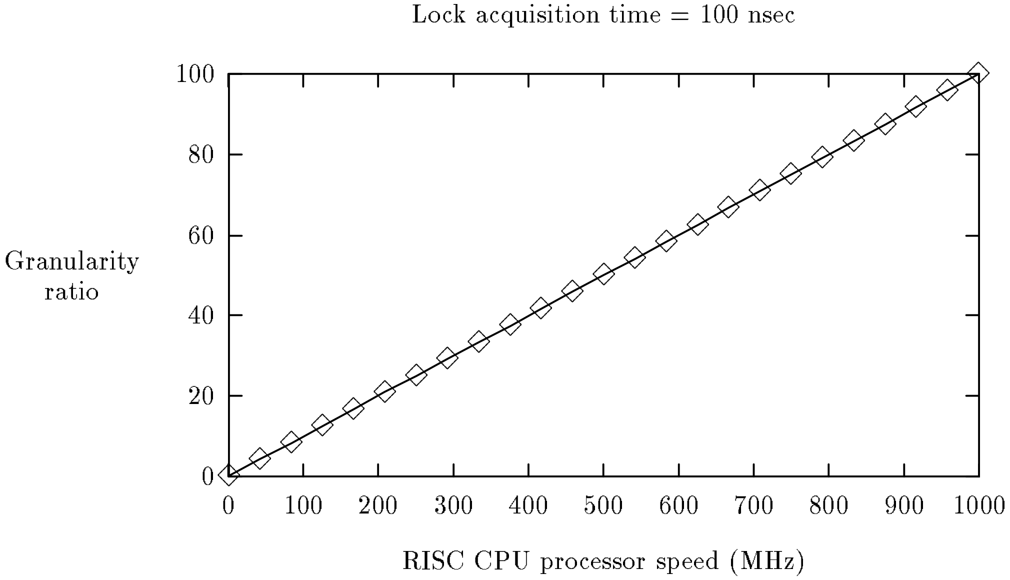

Figure 72: The granularity ratio of computation to lock acquisition time as a function of RISC CPU processor speed.

node to another be worthwhile uses of the optical waveguide's bandwidth? Can other command primitives (whether computational or synchronization in nature) be developed which use the OBR's properties to good advantage?

The fifth question deals with the engineering issues of using our proposed OBR architecture. How many wavelengths (one wavelength being equivalent to one logical channel) can be made available for the wired-`OR` OBRs (which are low bandwidth) and how many for the explicit message OBRs (which are high bandwidth)? Should the transmitters and receivers be tunable or fixed wavelength devices? How fast can the explicit message OBR data transmission channels be reasonably made? Should multiplexing techniques be used to combine a number of explicit message OBR data transmission channels into one physical channel? How many taps (which limits the number of nodes that can be physically attached to an OBR) can be placed on a single physical optical waveguide?

The final issue is much more broad. What other physical optical communication architectures (either in waveguides or in free space) are useful for supporting programming needs? The OBR structure provides inherent ordering and broadcast. Are there other properties (in either the OBR structure or other optical architectures) which would be useful? What other existing computing models are well suited to being supported by optical communication architectures? Can a new computing model be developed which fits naturally with optical communication architectures? And finally, if the computational needs are implemented via optics, can the optical computational implementation be effectively integrated with an optical communications architecture?

Bibliography

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the* 15^{th} *Annual International Symposium on Computer Architecture*, pages 280–9, 1988.

[2] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc., 1989.

[3] J. Archibald and J. Baer. Cache coherence protocols: Evaluation using a multiprocessor model. *ACM Transactions on Computer Systems*, 4(4):278–98, November 1986.

[4] Emmanuel A. Arnould, Francois J. Bitz, Eric C. Cooper, H. T. Kung, Robert D. Sansom, and Peter A. Steenkiste. The design of Nectar: A network backplane for heterogeneous multicomputers. In *Proceedings of Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–16, April 1989.

[5] Henri H. Arsenault and Yunlong Sheng. *An Introduction to Optics in Computers.* SPIE Optical Engineering Press, 1992.

[6] Donald G. Baker. *Monomode Fiber-Optic Design with Local-Area and Long-Haul Network Applications.* Van Nostrand Reinhold Company, 1987.

[7] W. Thomas Cathey, Kelvin Wagner, and William J. Miceli. Digital computing with optics. *Proceedings of the IEEE*, 77(10):1558–72, October 1989.

[8] L. M. Censier and P. Feautrier. A new solution to coherence problems in multiprocessors. *IEEE Transactions on Computer*, C-27(12):1112–8, December 1978.

[9] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, June 1990.

[10] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Fourth International Conference*

on Architectural Support for Programming Languages and Operating Systems, pages 224–34, April 1991.

[11] Ray T. Chen, Huey Lu, Daniel Robinson, Michael Wang, Gajendra Savant, and Tomasz Jannson. Guided-wave planar optical interconnects using highly multiplexed polymer waveguide holograms. *Journal of Lightwave Technology*, 10(7):888–97, July 1992.

[12] Donald M. Chiarulli, Steven P. Levitan, and Rami G. Melhem. Optical bus control for distributed multiprocessors. *Journal of Parallel and Distributed Computing*, 10:45–54, 1990.

[13] Richard D. Cooke. MACROLAN—the design philosophy and trade-offs. In *SPIE Vol. 630 Fibre Optics '86*, pages 65–70, 1986.

[14] Nicholas C. Craft and Michael E. Prise. Processor does light logic. *Laser Focus World*, pages 191–200, May 1990.

[15] W. A. Crossland, P. A. Kirkby, J. W. Parker, and R. J. Westmore. Some applications of optical networks in the architecture of electronic computers. *Optical Computing and Processing*, 1(3):199–207, 1991.

[16] Martin H. Davis, Jr. and Umakishore Ramachandran. Using an optical bus in a distributed memory multicomputer. In *The Sixth Distributed Memory Computing Conference Proceedings, DMCC6*, pages 524–31, April 1991.

[17] L. Dekker and E. E. E. Frietman. Optical link and processor clustering in the Delft parallel processor. In *Proceedings of 1988 International Conference on Supercomputing*, pages 25–38, 1988.

[18] L. Dekker and E. E. E. Frietman. Optical interconnects in high bandwidth computing. In *SPIE Vol. 1505, Optics for Computers: Architectures and Technologies*, pages 148–157, March 1991.

[19] L. Dekker, E. E. E. Frietman, W. Smit, and J. C. Zuidervaart. Optical link in the Delft parallel processor—an example of MOMI-Connection in MIMD-supercomputers. *Future Generations Computer Systems*, pages 189–203, 1988.

[20] D. Del Corso, H. Kirrmann, and J. D. Nicoud. *Microcomputer Buses and Links*. Academic Press, 1986.

[21] Digital Equipment Corporation, Maynard, Massachusetts. *DECChip 21064-AA RISC Microprocessor Preliminary Data Sheet*, April 1992.

[22] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *Proceedings of the* 13^{th} *Annual International Symposium on Computer Architecture*, pages 434–442, 1986.

[23] Dror G. Feitelson. *Optical Computing: A Survey for Computer Scientists.* The MIT Press, 1988.

[24] Michael R. Feldman, Sadik C. Esener, Clark C. Guest, and Sing H. Lee. Comparison between optical and electrical interconnects based on power and speed considerations. *Applied Optics*, 27(9):1742–51, May 1988.

[25] Eric Freudenthal and Allan Gottlieb. Process coordination with Fetch-and-Increment. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 260–8, April 1991.

[26] E. E. E. Frietman, A. de Vette, L. Dekker, and L. Tassakos. Optical interconnects in a multi-computer environment. In *SPIE Vol. 1281, Optical Interconnections and Networks*, pages 33–40, March 1990.

[27] E. E. E. Frietman, L. Dekker, E. H. Nordholt, and D. Chr. van Maaren. Optical interconnects facilitate the way to massive parallelism. In *SPIE Vol. 991 Fiber Optic Datacom and Computer Networks*, pages 152–60, 1988.

[28] E. E. E. Frietman, L. Dekker, W. van Nifterick, P. Demeester, P. van Daele, and W. Smit. Current status and future research of the Delft 'supercomputer' project. In *SPIE Vol. 1390, International Conference on Advances in Interconnection and Packaging*, pages 434–53, November 1990.

[29] E. E. E. Frietman and A. B. Ruighaver. An electro-optic data communication system for the Delft parallel processor. *Computer Architecture News*, pages 2–8, December 1987.

[30] E. E. E. Frietman, W. van Nifterick, L. Dekker, and T. J. M. Jongeling. Parallel optical interconnects: Implementation of optoelectronics in multiprocessor architectures. *Applied Optics*, 29(8):1161–77, March 1990.

[31] Edward E. E. Frietman and Wim van Nifterick. Optoelectronic ICs for high-speed parallel processing. *Lasers & Optronics*, pages 69–71, August 1987.

[32] Mario Gerla, Paulo Rodrigues, and C. W. Yeh. Token-based protocols for high-speed optical-fiber networks. *Journal of Lightwave Technology*, LT-3(3):449–66, June 1985.

[33] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the* 17^{th} *Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[34] A. Gottlieb and C. Kruskal. Coordinating parallel processors: A partial unification. *Computer Architecture News*, pages 16–24, October 1981.

[35] Peter S. Guilfoyle and Richard V. Stone. Digital optical computer II. In *SPIE Vol. 1563 Optical Enhancements to Computing Technology*, pages 214–22, July 1991.

[36] Peter S. Guilfoyle and W. Jackson Wiley. Combinatorial logic based digital optical computing architectures. *Applied Optics*, 27(9):1661–73, May 1988.

[37] Rajiv Gupta and Michael Epstein. High speed synchronization of processors using fuzzy barriers. *International Journal of Parallel Programming*, 19(1):53–73, 1990.

[38] Klaus-Rüdiger Hase. Computer-internal optical bus system with light-guiding-plate. In *Proceedings of 1985 European Conference on Optical Communications*, pages 597–600, 1985.

[39] Peter Healey, Stephen Cassidy, and David W. Smith. Multi-dimensional optical interconnection networks. In *SPIE Vol. 1215, Digital Optical Computing II*, pages 191–7, January 1990.

[40] David V. James, Anthony T. Laundrie, Tein Gjessing, and Gurindar S. Sohi. Distributed directory scheme: Scalable Coherent Interface. *Computer*, 23(6):74–7, June 1990.

[41] Jie Jiang and Udo Kraemer. Two new developments for optoelectronic bus systems. In *SPIE Vol. 1505, Optics for Computers: Architectures and Technologies*, pages 166–74, 1991.

[42] Jie Jiang and Peter Laws. Analysis and realization of the optical backplane system using circular light-guiding plates. In *SPIE Vol. 1773, Optical Enhancements to Computing Technology II*, 1992. Paper 1773B-14.

[43] Harry F. Jordan. A special purpose architecture for finite element analysis. In *Proceedings of the 1978 International Conference on Parallel Processing*, pages 263–6, August 1978.

[44] Harry F. Jordan. Digital optical computers at Boulder. In *SPIE Vol. 1505 Optics for Computers: Architectures and Technologies*, pages 87–98, March 1991.

[45] Harry F. Jordan and Vincent P. Heuring. Time multiplexed optical computers. In *Proceedings of Supercomputing '91*, pages 370–8, November 1991.

[46] Brewster O. Kahle, Edward C. Parish, Thomas A. Lane, and Jerry A. Quam. Optical interconnects for interprocessor communications in the Connection Machine. In *Proceedings of 1989 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 58–61, 1989.

[47] Gerd Keiser. *Optical Fiber Communications (Second Edition).* McGraw-Hill Book Company, 1991.

[48] P. A. Kirkby. SYMFONET: Ultra-high-capacity distributed packet switching network for telecoms and multiprocessor computer applications. *Electronics Letters*, 26(1):19–21, January 1990.

[49] Udo Kraemer and Peter Laws. High-speed optical interconnect for backplane applications. In *SPIE Vol. 1773, Optical Enhancements to Computing Technology II*, 1992. Paper 1773B-15.

[50] H. T. Kung. Advances in multicomputers. *Computing Systems in Engineering*, 1(2–4):153–62, 1990.

[51] H. T. Kung. High-speed networks for high-performance computing. In *COMPCON Spring 1990*, pages 68–72, 1990.

[52] H. T. Kung, Robert Sansom, Steven Schlick, Peter Steenkiste, Matthieu Arnould, Francois J. Bitz, Fred Christianson, Eric C. Cooper, Onat Menzilcioglu, Denise Ombres, and Brian Zill. Network-based multicomputers: An emerging parallel architecture. In *Proceedings of Supercomputer '91*, pages 664–73, November 1991.

[53] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), September 1979.

[54] Thomas A. Lane, Jerry A. Quam, Brewster O. Kahle, and Edward C. Parish. Gigabit optical interconnects for the Connection Machine. In *SPIE Vol. 1178 Optical Interconnects in the Computer Environment*, pages 24–35, September 1989.

[55] Joon Lee and Umakishore Ramachandran. Synchronization with multiprocessor caches. In *The* 17^{th} *Annual International Symposium on Computer Architecture*, pages 27–37, May 1990.

[56] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the* 17^{th} *Annual International Symposium on Computer Architecture*, pages 148–59, 1990.

[57] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash multiprocessor. *Computer*, 25(3):63–78, March 1992.

[58] F. MacKenzie, S. A. Cassidy, P. Healey, D. W. Smith, and D. L. Williams. Waveguide interconnects for optical processing. *Optical Computing and Processing*, 1(2):169–74, 1991.

[59] F. MacKenzie, T. G. Hodgkinson, S. A. Cassidy, and P. Healey. Optical interconnect based on a fibre bus. *Optical and Quantum Electronics*, 24(4):S491–504, April 1992.

[60] Reinhard Maenner, Richard L. Shoemaker, and Peter H. Bartels. The Heidelberg Polyp system. *IEEE Micro*, pages 5–13, February 1987.

[61] R. Männer and O. Stucky. The Polyp multiprocessor: Architecture and applications in nuclear physics. *Computers in Physics*, pages 267–74, May 1990.

[62] Nicholas F. Maxemchuk. Twelve random access strategies for fiber optic networks. *IEEE Transactions on Communications*, 36(8):942–50, 8 1988.

[63] R. G. Melhem, D. M. Chiarulli, and S. P. Levitan. Space multiplexing of waveguides in optically interconnected multiprocessor systems. *The Computer Journal*, 32(4):362–9, 1989.

[64] Onat Menzilcioglu and Steven Schlick. Nectar CAB: a high-speed network processor. In *Proceedings of the* 11^{th} *International Conference on Distributed Computing Systems*, pages 508–15, May 1991.

[65] J. E. Midwinter and Y. L. Guo. *Optoelectronics and Lightwave Technology.* John Wiley & Sons, 1992.

[66] Miles J. Murdocca, Alan Huang, Jurgen Jahns, and Norbert Streibl. Optical design of programmable logic arrays. *Applied Optics*, 27(9):1651–60, May 1988.

[67] James W. Parker. Optical interconnection for advanced processor systems: A review of the ESPRIT II OLIVES program. *Journal of Lightwave Technology*, 9(12):1764–73, December 1991.

[68] Paul R. Prucnal, Edward E. Harstead, and Stuart D. Elby. Low-loss, high-impedance integrated fiber-optic tap. *Optical Engineering*, pages 1136–42, September 1990.

[69] Floyd E. Ross. FDDI—a tutorial. *IEEE Communications Magazine*, 24(5):10–7, May 1986.

[70] Floyd E. Ross. An overview of FDDI: The fiber distributed data interface. *IEEE Journal on Selected Areas in Communications*, 7(7):1043–51, September 1989.

[71] D. A. A. Roworth and N. Howe. ISOLAN—a fibre optic network conforming to IEEE 802.3 standards. In *SPIE Vol. 630, Fibre Optics '86*, pages 88–95, 1986.

[72] A. B. Ruighaver. A modular network for dense optical interconnection of processing elements. *Computer Architecture News*, pages 69–75, June 1990.

[73] A. B. Ruighaver. The Melbourne University optoelectronic multicomputer project. *Supercomputer*, 8(6):22–32, November 1991.

[74] A. B. Ruighaver. From a very long instruction word architecture to a decoupled multicomputer architecture. In *Proceedings of the 1992 International Conference on Parallel Processing*, pages I–188–91, August 1992.

[75] Christoph Scheurich and Michel Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the* 14^{th} *Annual International Symposium on Computer Architecture*, pages 234–43, 1987.

[76] Ronald V. Schmidt, Eric G. Rawson, Robert E. Norton, Jr., Stephen B. Jackson, and M. Douglas Bailey. Fibernet II: A fiber optic Ethernet. *IEEE Journal on Selected Areas in Communications*, SAC-1(5):702–11, November 1983.

[77] G. Sohi, J. Smith, and J. Goodman. Restricted Fetch&Φ operations for parallel processing. In *Proceedings of 1989 International Conference on Supercomputing, Crete, Greece*, pages 410–6, June 1989.

[78] Per Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, June 1990.

[79] R. W. Stevens. MACROLAN: a high-performance network. *ICL Technical Journal*, pages 289–96, May 1983.

[80] R. W. Stevens. MACROLAN: a high-performance network. In *SPIE Vol. 468 Fibre Optics '84*, pages 88–93, 1984.

[81] O. Stucky, R. L. Shoemaker, R. Männer, and P. H. Bartels. Optical interconnection for multiprocessor computer bus systems. *Optical Engineering*, pages 1185–92, November 1989.

[82] Oliver Stucky, Reinhard Männer, Richard L. Shoemaker, and Peter H. Bartels. Multiprocessor communication and control by synchronous and asynchronous optical resource-sharing interconnection networks. In *Proceedings of the Third International Symposium on Computer and Information Sciences*, pages 571–8, October 1988.

[83] Technical Committee on Computer Communications of the IEEE Computer Society, USA, New York. *IEEE Standards for Local and Metropolitan Area Networks: distributed queue dual bus (DQDB) subnetwork of a Metropolitan Area Network (MAN)*, July 1991.

[84] Fouad A. Tobagi, Flaminio Borgonovo, and Luigi Fratta. Expressnet: A high-performance integrated-services Local Area Network. In Karl Kümmerle, Fouad A. Tobagi, and John O. Limb, editors, *Advances in Local Area Networks*, chapter 11, pages 171–89. IEEE Press, 1987.

[85] Chong-Wei Tseng and Bor-Uei Chen. D-Net, a new scheme for high data rate optical local area networks. *IEEE Journal on Selected Areas in Communications*, SAC-1(3):493–9, April 1983.

[86] Tom S. Wailes and David G. Meyer. Multiple channel architecture. In 3^{rd} *Symposium on the Frontiers of Massively Parallel Computation*, pages 315–23, October 1990.

[87] Tom S. Wailes and David G. Meyer. Multiple channel architecture: A new optical interconnection strategy for massively parallel computers. *Journal of Lightwave Technology*, 9(12):1702–16, December 1991.

[88] R. J. Westmore. SYMFONET: Interconnect technology for multinode computing. *Electronics Letters*, 27(9):697–8, April 1991.

Vita

Martin H. Davis, Jr. was born in Chattanooga, Tennessee. After attending The McCallie School for Boys in Chattanooga, he attended the Florida Institute of Technology in Melbourne. At F.I.T. he earned a B.S. in Physics and a B.S. in Space Science. Mr. Davis next attended the University of Tennessee Space Institute in Tullahoma. Because of his interest in Remote Sensing and computers, he earned a M.S. in Engineering Science by writing a Master's thesis entitled "A Geographic Information System for the Big South Fork Area." Finally, having decided that he wanted to learn more about computers, Mr. Davis began attending the Georgia Institute of Technology in Atlanta under the auspices of the School of Information and Computer Science, later to become the College of Computing. One of Mr. Davis' strong beliefs is that as Computer Science evolves and matures, it will become a highly interdisciplinary field. He particularly believes this will be true of Optical Computing, i.e., Optical Computing will come into its own only when physicists, electrical engineers, computer scientists, and others study in tandem optical technology and devices, system architectures, and computing models to determine the appropriate combination. Of course, some would say that Mr. Davis is somewhat biased given his particular background.