Designing VLSI Network Nodes to Reduce Memory Traffic in a Shared Memory Parallel Computer

Susan Dickey, Allan Gottlieb, Richard Kenner, Yue-Sheng Liu[1]

Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012
Ultracomputer Note #125
August, 1986

ABSTRACT

Serialization of memory access can be a critical bottleneck in shared memory parallel computers. The NYU Ultracomputer, a large-scale MIMD (Multiple Instruction stream, Multiple Data stream) shared memory architecture, may be viewed as a column of processors and a column of memory modules connected by a rectangular network of enhanced two by two buffered crossbars. These VLSI nodes enable the network to combine multiple requests directed at the same memory location. Such requests include a new coordination primitive, fetch-and-add, which permits task coordination to be achieved in a highly parallel manner. Processing within the network is used to reduce serialization at the memory modules.

To avoid large network latency, the VLSI network nodes must be high-performance components. Design tradeoffs between architectural features, asymptotic performance requirements, cycle time and packaging limitations are complex. This report sketches the Ultracomputer architecture and discusses the issues involved in the design of the VLSI enhanced buffered crossbars which are the key element in reducing serialization.

1. Introduction

Highly parallel computers composed of thousands of processors and gigabytes of memory have the potential to solve problems with vast computational demands. With 10-20 MIPS and a megabyte of memory soon to be available on a few chips, such highly parallel machines can be built with roughly the same component count as the current generation of supercomputers.

Effective utilization of such a high-performance assemblage demands an integrated hardware/software approach. Several thousand processors must be coordinated in such a way that their aggregate power is applied to useful computation. Classic techniques for interprocessor coordination use ‘‘critical sections’’ — serial procedures in which one processor accesses a shared resource while the others wait. In any highly parallel architecture, such an approach is inadequate, since the cost of critical

[1]This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U. S. Department of Energy, under contract number DEA-C0276-ER03077-V, and in part by the National Science Foundation, under grant number DCR-8413359.

sections rises linearly with the number of processors. Serial procedures become bottlenecks that drastically reduce the power obtained and thus must be eliminated.

Our group has proposed [7] that the hardware and software design of a highly parallel computer should meet the following goals.

- *Scaling.* Effective performance should scale upward to a very high level. Given a problem of sufficient size, an n-fold increase in the number of processors should yield a speedup factor of almost n.
- *General purpose.* The machine should be capable of efficient execution of a wide class of algorithms, displaying relative neutrality with respect to algorithmic structure or data flow pattern.
- *Programmability.* High-level programmers should not have to consider the machine's low-level structural details in order to write efficient programs. Programming and debugging should not be substantially more difficult than on a serial machine.
- *Multiprogramming.* The software should be able to allocate processors and other machine resources to different phases of one job and/or to different user jobs in an efficient and highly dynamic way.

The next section reviews the MIMD shared memory computational model on which the Ultracomputer is based and outlines a hardware design closely approximating this model. Section three discusses selected issues in the design of the network using custom VLSI switches. We conclude with a brief report of the current status and future plans of our VLSI effort.

A series of technical reports, referred to as Ultracomputer Notes, have been prepared by researchers. Readers wishing more information should write to Michael Passaro at 251 Mercer Street, New York, NY 10012 (passaro@nyu.arpa).

2. Ultracomputer Architecture

In this section we review the parallel computation model on which the Ultracomputer is based. Although this idealized machine is not physically realizable, we show that a close approximation can be built.

2.1. The Model

An idealized parallel processor, dubbed a ''paracomputer'' by Schwartz [23] and classified as a WRAM by Borodin and Hopcroft [1], consists of a number of autonomous processing elements (PEs) sharing a central memory (see also Fortune & Wylie [5] and Snir [24]). Every PE is permitted to read or write a shared memory cell each cycle. In particular, simultaneous reads and writes directed at the same memory cell are all acomplished in a single cycle.

The *serialization principle* (see Eswaran *et al.* [4]) states that the effect of simultaneous actions by the PEs is as if the actions had occurred in some (unspecified) serial order. Thus, for example, a load simultaneous with two stores directed at the same memory cell will return either the original value or one of the two stored values. The returned value may be different from the value that the cell finally comes to

contain. In this model, simultaneous memory updates are in fact accomplished in one cycle; the serialization principle makes precise the effect of simultaneous access to shared memory but does not prescribe its implementation.

In an actual hardware implementation, single cycle access to globally shared memory cannot be achieved. For any technology there is a limit, say b, on the number of signals that one can fan in at once. Thus, if N processors are to access even a single bit of shared memory, the shortest access time possible is $\log_b N$. Hardware achieving this logarithmic access time, even when many processors simultaneously access a single cell, has been designed, but does not use off the shelf components. A custom VLSI design is needed for the switching components used in the processor to memory interconnection network.

This network adds significantly to the size of the machine and to its replication costs. For N processors and N memory modules, $N \log N$ switching components are required. This results in an inherently lower peak performance than that of a design of equivalent size in which the processors themselves act as switching components without globally shared memory. For any metric (dollars, cubic feet, BTU's, etc.) the shared memory design with a connection network will contain fewer processors or memory cells than a private memory design with only wires connecting the processors.

We believe that the increased flexibility and generality of shared memory designs adequately compensates for their lower peak performance, but this issue has not been settled. Most likely the answer will prove to be so application dependent that both shared and private memory designs will prove successful.

2.2. The Fetch-and-add Operation

We augment the shared memory model described above with the ''fetch-and-add'' (F&A) operation. This operation is an indivisible add to memory; its format is F&A(X,e), where X is an integer variable and e is an integer expression. The operation is defined to return the (old) value of X and to replace X by the sum $X + e$.

Concurrent fetch-and-adds are required to satisfy the serialization principle. Fetch-and-add operations simultaneously directed at X cause it to be modified by the sum of the increments. Each operation yields the intermediate value of X corresponding to some order of execution. The following example illustrates a typical use of fetch-and-add: Consider several PEs concurrently executing F&A(I,1), where I is a shared variable used as an index into a shared array. Each PE obtains an index to a distinct array element (although one cannot predict which element will be assigned to which PE), and I is incremented by the number of PEs executing the F&A.

Fetch-and-add is a powerful interprocessor synchronization operation that permits highly concurrent execution of both operating system primitives and application programs (see Gottlieb and Kruskal [9]). Using the fetch-and-add operation, we can perform many important algorithms in a completely parallel manner, i.e. without using any critical sections. For example, as indicated above, concurrent executions of F&A(I,1) yield consecutive values that may be used to index an array. If this array is interpreted as a (sequentially stored) queue, the values returned may be used to perform concurrent inserts; analogously

F&A(D,1) may be used for concurrent deletes. The complete queue algorithms contain checks for overflow and underflow, collisions between insert and delete pointers, etc. (see Gottlieb *et al.* [10]). We are unaware of any other completely parallel solutions to this problem. To illustrate the nonserial behavior obtained, we note that given a single queue that is neither empty nor full, the concurrent execution of thousands of inserts and thousands of deletes can all be accomplished in the time required for just one such operation.

2.3. Hardware Realization

The direct single cycle access to shared memory characteristic of paracomputers is approximated in the NYU Ultracomputer by indirect access via a multicycle connection network. A message switching network with the topology of Lawrie's [18] Ω-network (equivalently, the SW Banyan of Goke and Lipovsky [6]) is used to connect $N = 2^D$ autonomous PEs to a central shared memory composed of N memory modules (MMs). Figure 1 gives a block diagram of the machine.

The Ultracomputer design places few constraints on the processors and memory modules. Naturally, the fetch-and-add instruction is needed. In addition, the presence of a memory which has, when viewed through the network, a high bandwidth and non-negligible latency strongly favors processors that permit prefetching of instructions and operands. Furthermore, memory should be interleaved to prevent clustering of references to a given module, and should be augmented with an adder to perform the fetch-and-add operation atomically.

Implementing a cache at each PE permits rapid access to frequently-used instructions and data and reduces network traffic. Unfortunately, caching of read-write shared variables presents a coherence problem among the various caches. An obvious method of eliminating this problem is to simply not cache read-write shared variables and have the software distinguish between shared and private variables, typically by grouping them into separate memory-management segments. A more elaborate scheme is based on the observation that if, during a particular code segment, a shared variable is accessed read-only, or accessed only privately, then this variable may be cached during execution of that segment (McAuliffe [20]).

The enhanced Ω-network connecting processors to memory that we proposed for the Ultracomputer achieves the following objectives:

- Bandwidth linear in N, the number of PEs.
- Latency (memory access time) logarithmic in N.
- Only O(N log N) identical components.
- Routing decisions local to each switch; thus routing is not a serial bottleneck and is efficient for short messages.

Details of the network design are given in the following section and in Gottlieb [7]. See Figure 2 for a diagram of an Ω-network. Note that there exists a unique path connecting each PE-MM pair; the

method for using such a network to implement memory loads and stores is well-known.

Although issued by the processor, fetch-and-add operations are effected in the MMs. Since memory modules operate sequentially, only one request may be satisfied in each cycle. If concurrent fetch-and-add or load operations were to be serialized at the memory of a real parallel computer, we would lose the advantage of parallel coordination algorithms, having merely moved the critical sections from the software to the hardware. Instead, memory requests to identical locations are combined when they meet at a switch (see Klappholz [13] and Gottlieb *et al* [10]).

Combining (merging) requests reduces communication traffic and thus decreases the lengths of the queues within the switches, leading to lower network latency (i.e. reduced memory access time). Furthermore, it can help to prevent saturation of the network when ‘‘hotspot’’ traffic directs many requests to a single location (Pfister & Norton [22] and Lee *et al.* [19]).

Enhanced switches permit the network to combine fetch-and-adds with the same efficiency as it combines loads and stores. When two fetch-and-adds referencing the same shared variable, say F&A(X, e) and F&A(X, f), meet at a switch, the switch forms the sum $e+f$, transmits the combined request F&A(X, $e+f$), and stores the value e in its local memory. When the value Y is returned to the switch in response to F&A(X, $e+f$), the switch transmits Y to satisfy one request (F&A(X, e)) and transmits $Y+e$ to satisfy the other request (F&A(X, f)). Assuming that the combined request was not further combined with yet another request, memory location X is properly incremented, becoming $X+e+f$. If other fetch-and-add operations updating X are encountered, the combined requests are themselves combined. The associativity of addition guarantees that this procedure gives a result consistent with the serialization principle. Note that other associative operations can be combined in a similar manner.

3. Network Design

Designing a network having the above functionality and providing adequate performance for a given configuration involves many tradeoffs. The asymptotically best organization may not necessarily attain the best performance when constructed in a given VLSI technology. The following four subsections discuss in turn message transmission protocols and the effect of packaging limitations, network performance analysis, the structure of buffers internal to the switch, and the cost of combining.

3.1. Message Transmission and Packaging Limitations

Bandwidth proportional to the number of PEs has been achieved by providing queues within each switch. These allow concurrent processing of requests routed to the same port whenever possible. The alternative adopted by Burroughs (Hauck & Dent [12]) of killing one of the two conflicting requests limits bandwidth to $O(\frac{N}{\log N})$; see Kruskal & Snir [14].

In addition, the network is pipelined. Paths through the network are not maintained while awaiting memory responses. Thus the rate at which the processor may issue memory requests is limited by the switch cycle time rather than the network transit time.

The delay inherent in off-chip communication between VLSI switching nodes is likely to be a greater constraint on network performanance than the rate at which information can be processed within each node. Significant additional logic can be included in each node with advantage when that logic would help avoid global signaling or reduce queuing delays by combining messages. On the other hand, for a lightly loaded network the basic cycle speed of the switch is the most important factor in determining the latency of the network. If processing within a node is overlapped with data transfer between nodes, an increase in internal complexity may be tolerated without lengthening the cycle. The degree to which this can be done depends on the current state of technology.

The number of chips required to implement each switching node appears likely to be determined by the pin count required at each node, rather than the silicon area of the switching logic. Logic density per chip has been increasing much faster than pin density per chip. The number of pins available per chip is still too small to construct even a 2×2 bidirectional switch which processes single packet messages (address, control and data) for a 32 bit machine. Therefore, messages must be split into multiple packets and one of two methods can be used to transmit these packets through the network. The first is a bit-sliced implementation in which different components are handling different packets of one message (transmission of messages is ‘‘space-multiplexed’’). Or the transmission of successive packets of a message can be time-multiplexed to the same component.

Space-multiplexing provides a higher bandwidth than time-multiplexing at the expense of more components. However, a large amount of ‘‘horizontal’’ communication and coordination must then take place between the different components of a switch. The communication between components required to match addresses and add data when combining is especially costly in both the complexity of such implementation and the switch cycle time. For MOS technologies, the off-chip delays impose an especially high overhead. Furthermore, module packaging limitations are such that, even if several chips are used to implement a single node, some degree of time-multiplexing may be required in order to package a network node on a given module.

Several cycles are required to transmit each message if time-multiplexing is used. However, the internal logic of the switch can be pipelined so that messages can be handled on a per-packet basis and do not have to be assembled at each switch. Thus, there can be as little as a one cycle delay per switch when queues are empty and hence time-multiplexing contributes an additive term to the latency rather than a multiplicative factor. However, queuing delays increase quadratically with the multiplexing factor, so that the performance of the network under heavy load may be seriously impaired (Kruskal *et al.* [15]).

For any given state of technology, packaging constraints do not determine a unique design for the network. For the same number of pins per chip, it is possible to replace 2×2 switches by $k \times k$ switches, time multiplexing each pin by a factor of $\frac{k}{2}$. Dividing a message into more packets may or may not involve an increase in cycle time, depending on the nature of the detailed VLSI design. Breaking up parts of the address or the data into different packets will increase the internal complexity required to handle matching and addition, but fewer bits per packet will shorten certain global control lines. The increased

logic to perform $k \times k$ switching rather than 2×2 switching is likely to increase cycle time, but possibly not by a significant amount. In the following section, we present performance analyses of various networks in order to indicate the tradeoffs involved.

3.2. Network Performance Analysis

A particular configuration is characterized by the values of the following parameters:

k the size of the switch. Recall that a $k \times k$ switch requires $4k$ ports.

m the time multiplexing factor, i.e. the number of switch cycles required to input a message. (To simplify the analysis we assume that all the messages have the same length.)

t the switch cycle time.

Note that for any k a network with n inputs and n outputs can be built from $\frac{bn}{k}$ $k \times k$ switches and a proportional number of wires, where b is $\log_k n$ and n is a power of k. (If n is not a power of k, some redundancy or unused connections may be required to use identical parts.) Since our network contains a large number of identical switches, the network's cost may be considered proportional to the number of switches.

In order to obtain a tractable mathematical model of the network the following simplifying assumptions are made for the remainder of this section:

- Requests are not combined.
- Messages have the same length.
- Queues are of infinite size.
- Requests are generated at each PE by independent, identically distributed, time-invariant random processes.
- MMs are equally likely to be referenced.

Let p be the average number of messages entered into the network by each PE per time unit. If the queues at each switch are large enough (''infinite queues'') then the average switch delay at the first stage is

$$t + t\left[\frac{m^2 p(1-1/km)}{2(1-mp)}\right]$$

and the average switch delay at later stages is approximately

$$t + t(1 + \frac{4mp}{5k})\left[\frac{m^2 p(1-1/k)}{2(1-mp)}\right]$$

(see Kruskal *et al.* [15]) To compute the average network traversal time T (in one direction), sum the individual stage delays plus the setup time for the pipe, i.e. $(m-1)\ t$.

Note that the network has a capacity of $\frac{1}{m}$ messages per switch cycle per PE. That is, each PE cannot enter messages at a rate higher than one per m cycles, and, conversely, the network can accommodate any traffic below this threshold. Thus, the global bandwidth of the network is theoretically proportional to the number of PEs connected to it.

The initial t in the expressions for the switch delay corresponds to the time required for a message to be transmitted through a switch without being queued (the switch service time). The second term corresponds to the average queuing delay. This term decreases to zero when the traffic intensity p decreases to zero and increases to infinity when traffic intensity p increases to the $\frac{1}{m}$ threshold. The surprising feature of this formula is the m^2 factor, which is explained by noting that the queuing delay for a switch with a multiplexing factor of m is roughly the same as the queuing delay for a switch with a multiplexing factor of one, a cycle mP times longer, and m times as much traffic per cycle [8].

We have plotted in Figure 3 the graphs of T, the network transit time, as a function of the messages per microsecond for different values of k and m with t equal to 50 nanoseconds. Note that the performance advantage of having fewer stages in the network (k greater than 2) is easily outweighed by the increased queuing delay due to more packets per message (m large). This is true even when the cycle time is the same; in practice one would expect the more complex $k\times k$ switches to have a longer cycle time.

Figure 4 shows the effects of increasing and decreasing cycle time. A variety of different switch structures can give roughly the same performance. For a fixed offered load coming from the processor, increasing the switch cycle time has the effect of increasing the traffic intensity p as seen by the switch as well as increasing the service time. Thus, decreasing the cycle time by 20 percent (from 50 to 40 nanoseconds) can improve performance more than going from a 2×2 to a 4×4 network. A comparatively fast cycle time can make a switch with 8-packet time-multiplexing attractive. (Such a switch is likely to be faster only for a non-combining switch, however, due to the difficulties of matching across packets.) On the other hand, the cycle time can increase from 50 to 100 nanoseconds for a 4×4 switch without losing performance if m is cut from 4 to 2.

The above discussion of performance has ignored the cost of constructingh the network. Since the component count for 64 PE network using 2×2 nodes is 192, for 4×4 is 48, and for 8×8 is 16, Figure 5 compares networks of comparable component cost, assuming 50 nanosecond switch times per component. The transit time at traffic intensity p for a message when there are d copies of a given network is the same as for one copy of that network with a traffic intensity $\frac{p}{d}$.

Limitations on offered load due to processors waiting for outstanding memory requests make it unlikely that more than one copy of the network will be desirable (except for fault tolerance), unless more than one PE feeds into each network port (see Norton & Pfister [21]). Results of simulations for a 64-PE network with small buffer sizes at each node (shown in Figure 6) show that an actual load on the network of much more than 1 message per microsecond is unlikely for processors with no prefetch capability. The

processor's rate of issuing requests is slowed by waiting for a response from memory even at the minimum network transit time. The effective throughput is considerably below even the slow memory's maximum bandwidth of 3.3 messages per microsecond. Extra copies of the network provide little help when traffic intensity is that low.

A final determination of an optimal configuration requires more accurate assessments of the technological constraints and the traffic distribution. The pipelining delays incurred for large multiplexing factors, the complexity of large switches, and the heretofore ignored cost and performance penalty incurred with interfacing many network copies, will probably make the use of switches larger than 8×8 impractical for a 4K PE parallel machine.

The previous discussion assumed a one chip implementation of each switch. By using the two chip implementation described in the next section one can double the bandwidth of each switch while doubling the chip count. As delays are highly sensitive to the multiplexing factor *m*, this implementation would yield a better performance than that obtained by taking two copies of a network built of one chip switches. (It would also have the extra advantage of decreasing the silcon area required on each chip.)

We now return to the five assumptions listed above. The first two assumptions, that all messages traverse the entire network and are of equal (maximal) length, are clearly conservative. In practice, combined messages do not each traverse the entire network, and messages that do not carry data (load requests and store acknowledgements) could be shorter.

For the last three assumptions, simulations have shown that queues of modest size can give essentially the same performance as infinite queues (see Figure 8 and also Norton & Pfister [21]). Interleaved memory may make the patterns of access to MMs essentially uniform. PE processes cooperating on the same task will certainly not be independent, but their patterns of memory access, as seen by the network after mediation through a cache, may not be significantly correlated.

3.3. Switch Structure

In the current design we have chosen to use time-multiplexing, with each message divided into one packet containing the path descriptor, address and opcode, plus one or more data packets[2].

The protocol used to transmit messages between switches is a message-level rather than packet-level protocol. That is, packet transmission cannot be halted in the middle of a message. A switch will accept a new message only if the available space in its queues guarantees that it will be able to receive the entire message.

The switch is designed to meet the following goals:

[2] At the expense of a severe increase in complexity, the address can also be transmitted in more than one packet (Snir & Solworth [26]).

- Distinct data paths do not interfere with each other. Therefore, a new message can be accepted at each input port provided queues are not full. In addition, a message destined to leave at some output port will not be prevented from doing so by a message routed to a different output port.
- A packet entering a switch with empty queues, when no other message is destined for the same output port, leaves the switch at the next cycle.
- The capability to combine and de-combine memory requests does not unduly slow the processing of requests that are not to be combined.

Figure 7 shows a block diagram of a switching node. The ‘‘PE port’’ connects to either a PE or to an MM port of a preceding network stage and the ‘‘MM port’’ connects to either an MM or a PE port of a subsequent network stage.

Associated with each MM port is a combining queue capable of accepting a packet simultaneously from each PE port. Requests that have been combined with other requests are sent to a wait buffer at the same time as the combined request is sent to the MM port.

From each MM port, a reply enters both the associated wait buffer and the non-combining queue associated with the PE port to which the reply is routed. The wait buffer inspects all responses from MMs and searches for a response to a previously combined request. When it finds a response to such a request, it generates a second response and deletes the request from its memory. Each non-combining queue has four inputs since messages may come from both MM ports and from both wait buffers.

If a full 2×2 bidirectional switch cannot be constructed on a single chip, packaging alternatives include dividing each switch into a forward path component (FPC), consisting of the two combining queues, and a return path component (RPC) consisting of the wait buffers and non-combining queues. Data forwarded to a wait buffer from a combining queue are transmitted from the FPC to the RPC via two wait buffer output and input ports on the FPC and RPC, respectively. Furthermore, the wait buffers may be constructed on chips separate from the non-combining queues. In addition, pin limitations may require multiplexing the wait buffer data from the FPC to the RPC over a single port.

We do not actually implement two-input combining queues or four input non-combining queues. Such designs, while possible (see Snir [25]), involve a considerable increase in complexity. Instead, to achieve the goal of a switch where there is no interference between distinct data paths, we include a separate one-input queue on each input-output path, in all 4 queues for a 2×2 FPC. Simulations indicate (see Figure 8) that performance of our scheme (four one-input queues, two of which are multiplexed at each output port with a selector that alternates when both queues have data, but always presents data if only one queue has data) is very close to that of a scheme with 2-input queues and equivalent total queue capacity. Performance appears to be superior to that of a design with multiplexing of input data into two 1-input queues associated with each output port (see Kumar & Jump [16]). A disadvantage of our scheme is that messages from distinct input ports cannot be combined. They would, however, leave via the same output port and this may combine one stage later. On the return path, the four-input non-combining queue can be replaced with four one-input queues. Depending on the protocol used, the queues fed by the wait

buffers may only require space for one message.

Details of the current switch design and a description of an implementation for a planned 32-PE prototype can be found in Dickey *et al.* [2, 3] and Gottlieb [7].

3.4. Combining and its implementation cost

Our design for combining and non-combining queues is an enhancement of the VLSI systolic queue of Guibas and Liang [11]. They present a FIFO buffer where an insertion or deletion can be performed every four cycles, and where no global control signals are used, other than the clock signals used by the two-phase logic. We use a modified version of this structure, where insertions and deletions can be made at each cycle. To achieve this we resort to an increased number of global control signals. In combining queues, comparators are added to the basic queue structure to detect requests which are to be combined.

A combining queue consists of three columns: an IN column, an OUT column, and a CHUTE column (see Figure 9). Packets added to the queue enter the IN column and move up the column each cycle until the adjacent slot in the OUT column is empty. In the absence of combining, they then move over to the OUT column and begin moving down. Should a packet reach the end of the IN column without being able to move to the OUT column, the queue is declared full, and no more messages can be accepted.

The above horseshoe movement pattern ensures that, as a new message moves up the IN column, it passes all messages in the queue at the time of its arrival. If the address of a message in the IN column matches the address of a message in the adjacent slot of the OUT column, the item in the IN column is shunted over to the CHUTE, where it proceeds down the CHUTE in tandem with the corresponding message in the OUT column. The two messages exit the queue at the same time. Combining logic then sends a combined request (described below) to the OUT port and appropriate information to allow de-combining to the wait buffer. Packetizing of messages and the blocking of the queue somewhat complicates this logic, since only address packets are to be compared.

For loads, the combined request contains the op-code (load) and the address (including PE address) of one of the requests; for stores, the same plus one of the values to be stored[3]. In each case, the wait buffer must receive the op-code and the two PE addresses of the combined requests, plus the memory address or some other unique identifier for the message that has been combined[4]. For fetch-and-add operations, the combined request containing the sum of the two increments is forwarded to the next stage while the increment from the OUT column is stored in the wait buffer. Upon de-combining, the request that arrived first will receive the original value of the memory location, while the second request will

[3]The serialization principle permits us to discard the other value.

[4]If the memory address is used, a restriction must be added to the logic of the PE to have only one combinable request pending to a single memory location.

receive the original value plus the increment saved from the first.

A schematic for a single data bit (containing one slice of the IN, OUT, and CHUTE columns) is shown in Figure 10. FI, HI, FO, and HO are active during the first clock phase and are computed during the previous clock phase from global *queue full* and *queue blocked* status signals. OTRV, OTRH, CTRV and CTRH are active during the second clock phase and computed during the previous clock phase from the empty status of the OUT and CHUTE slots. The MATCH line is precharged during the first clock phase and is evaluated during the second clock phase. It is used during that phase to indicate whether the IN or CHUTE slots will be marked as occupied. In CMOS technology the additional cost of combining in a given queue cell amounts to 27 transistors out of a total 55.

This design allows only 2-way combining of messages. If a third message to a location arrives at a stage where two requests to that location are queued, only two of them will be combined. Recent work indicates that 2-way combining may not be sufficient; according to Lee *et al.* [19], 3-way combining is required to avoid saturation of the network by hotspot requests for networks with a large number of stages. The above design could be modified for three-way combining by the addition of another CHUTE column. This would involve an increase in complexity of control logic for the combining queue as well as for the wait buffer.

Assuming the same total amount of buffering in nodes on the forward and return paths, our current designs indicate that the silicon area required for an implementation of a 2×2 network node that does combining will be slightly less than double that of a non-combining switch. This is based on assumptions that a combining queue will be roughly twice the size of a non-combining queue with equal message capacity, and that the wait buffer will be approximately 75 percent the size of a non-combining queue. Due to the overlap of the computation of control information with data transmission, we estimate an increase in cycle time of only 10 to 20 percent.

4. VLSI Design Status

In preparation for the design of a complete combining switch node, we have designed several chips which have been fabricated by DARPA's MOSIS facility.

We have received functional 11-bit wide 2x2 non-combining forward path chips containing approximately 7500 transistors and fabricated in 3-micron NMOS. These parts operate at a clock speed of 23MHz with propagation delays from clock to output of approximately 25ns. Power dissipation is approximately 1.5W. A 4x4 test network was constructed using four of these parts and functioned as expected.

We have also had a 6-bit wide portion of the FPC (without the adder) for a 2x2 combining switch fabricated in 4-micron NMOS. This switch is composed of four 1-input combining queues. These parts also operate as expected and have performance and power dissipation similar to the non-combining switches.

Since the final combining switches must be at least 32-bits wide and air-cooled, we have converted our design effort to MOSIS's newly available scalable double-metal CMOS process, which promises minimum feature sizes as small as 1.4 microns. We have submitted, and are awaiting the fabrication of, a 35-bit non-combining forward path using this CMOS technology.

We are currently completing the design of the remaining components (a 32-bit adder and the associative wait buffer) and hope to breadboard a complete (albeit narrow) combining switch node later this academic year.

Acknowledgements

This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U. S. Department of Energy, under contract number DE-AC0276ER03077, and in part by the National Science Foundation, under grant number DCR-8413359.

References

[1] A. Borodin and J.E. Hopcroft, ''Routing, merging and sorting on parallel models of computation'', in *Proc. 14th Annual ACM Symp. on Theory of Comp.*, May, 1982.

[2] S. Dickey, R. Kenner, M. Snir, and J. Solworth, ''A VLSI Combining Network for the NYU Ultracomputer'', *Proc. Intern. Conf. on Comp. Design*, 1985.

[3] S. Dickey, R. Kenner, and M. Snir, ''An Implementation of a Combining Network for the NYU Ultracomputer'', Ultracomputer Note #93, Courant Institute, New York University, 1985.

[4] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, '' The Notion of Consistency and Predicate Locks in a Database System,'' *CACM*, pp. 624-633, November 1976.

[5] S. Fortune and J. Wyllie, ''Parallelism in Random Access Machines'', *Proc. 10th ACM Symp. on Theory of Comp.*, pp. 114-118, 1978.

[6] L.R. Goke and G.J. Lipovsky, ''Banyan Networks for Partitioning Multiprocessor Systems'', *Proc. First Annual Symp. on Comp. Arch.*, 1973.

[7] A. Gottlieb, ''An Overview of the Ultracomputer Project,'' Ultracomputer Note #100, Courant Institute, New York University, 1986.

[8] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, ''The NYU Ultracomputer — Designing an MIMD Shared Memory Parallel Computer'', *IEEE Trans. Comp.* pp. 175-189, Feb. 1983.

[9] A. Gottlieb and C.P. Kruskal, ''Coordinating Parallel Processors: A Partial Unification'', *Comp. Arch. News*, pp. 16-24. Oct. 1981.

[10] A. Gottlieb, B.D. Lubachevsky, and L. Rudolph, ''Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors'', *ACM TOPLAS* **5**, pp. 164-189, Apr. 1983.

[11] L.J. Guibas and F.M. Liang, ''Systolic stacks, queues and counters'', in *Proc. Conf. Advanced Research VLSI*, Jan. 1982.

[12] E.A. Hauck and B.A. Dent, ''Burroughs' B6500/B7500 Stack Mechanism'', *AFIPS 1968 SJCC*, pp. 245-251. Also in D.P. Siewiorek, C.G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, 1982, pp. 244-250.

[13] D. Klappholz, ‘‘Stochastically Conflict-free Data-base Memory Systems’’, *Proc. Intern. Conf. on Parallel Processing*, pp. 283-289, 1980.

[14] C.P. Kruskal and M. Snir, ‘‘The Performance of Multistage Interconnection Networks for Multiprocessors’’, *IEEE Trans. Comp.* **C-32**, pp. 1091-1098, 1983.

[15] C. P. Kruskal, M. Snir and A. Weiss, ‘‘The Distribution of Waiting Times in Clocked Multistage Interconnection Networks,’’ *Proc. 1986 Intern. Conf. on Parallel Processing*.

[16] M. Kumar and J. R. Jump, ‘‘Performance Enhancement in Buffered Delta Networks Using Crossbar Switches and Multiple Links,’’ *Journal of Parallel and Distributed Computing*, Vol. 1, No. 1, August 1984, pp. 81-103.

[17] M. Kumar and G. F. Pfister, ‘‘The Onset of Hot Spot Contention,’’ *Proc. 1986 Intern. Conf. on Parallel Processing*, pp 28-34.

[18] D.H. Lawrie, ‘‘Access and Alignment of Data in an Array Processor’’, *IEEE Trans. Comp.* **C-24**, pp. 1145-1155, Dec. 1975.

[19] G. Lee, C. P. Kruskal and D. J. Kuck, ‘‘The Effectiveness of Combining in Shared Memory Parallel Computers in the presence of ‘Hot Spots’,’’ *Proc. 1986 Intern. Conf. on Parallel Processing*, pp. 35 - 41.

[20] K. McAuliffe, Ph.D. thesis, Courant Institute, New York University, 1985, in preparation.

[21] A. Norton and G. Pfister, ‘‘A Methodology for Predicting Multiprocessor Performance,’’ *Proc. 1985 Intern. Conf. on Parallel Processing*, pp. 772 - 781.

[22] G.F. Pfister and V.A. Norton, ‘‘ ‘Hot Spot’ Contention and Combining in Multistage Interconnection Networks’’, *Proc. 1985 Intern. Conf. on Parallel Processing*, pp. 790 - 797.

[23] J. T. Schwartz, ‘‘Ultracomputers’’, *ACM TOPLAS* **2**, pp. 484-521, 1980.

[24] M. Snir, ‘‘On Parallel Search’’, *Principles of Distributed Computing*, Aug. 1982.

[25] M. Snir, ‘‘Switch Digest,’’ Ultracomputer Hardware Note #27, September 1984.

[26] M. Snir and J. Solworth, ‘‘The Ultraswitch — A VLSI Network Node for Parallel Processing’’, Ultracomputer Note #39, Courant Institute, New York University, 1984.