A VLSI Combining Network for the NYU Ultracomputer

Susan Dickey, Richard Kenner, Marc Snir and Jon Solsworth

Computer Science Department
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012

ABSTRACT

The NYU Ultracomputer architecture, a shared memory MIMD parallel machine composed of thousands of processing elements, requires a high-performance interconnection network to approximate the ideal behavior of Schwartz's paracomputer model and to implement the fetch-and-add synchronization primitive efficiently. In this paper we describe the routing scheme and protocols for an enhanced message switching network and a VLSI implementation of a network node. Systolic queues at each node combine accesses to the same memory location by different processing elements allowing them to be performed in the same memory cycle. We have had encouraging results to date towards the goal of attaining network latency comparable with memory access times.

1. The Ultracomputer Architecture

In this section we review the architectural model on which the Ultracomputer is based. The reader is referred to Gottlieb, Grishman, et al. [83], Edler et al. [85], and the references therein for further details.

An idealized parallel processor, dubbed a "paracomputer" by Schwartz [80], consists of a number of autonomous processing elements (PE's) each of which are permitted to read or write a shared central memory in a single cycle. In particular, simultaneous reads and writes directed at the same memory location are completed in a single cycle.

The Ultracomputer architecture augments the paracomputer model with the "fetch-and-add" (F&A) operation, a powerful interprocessor coordination primitive that permits highly concurrent execution of operating system algorithms and application programs (see Gottlieb and Kruskal [81]). Fetch-and-add is essentially an indivisible add to memory; its format is F&A(V,e), where V is an integer variable and e is an integer expression. The operation returns the (old) value of V and replaces V by the sum V+e. Concurrent fetch-and-adds are required to have the same effect as if executed in some (unspecified) serial order.

A paracomputer is not physically realizable due to fan-in (and other) limitations. Furthermore, if concurrent fetch-and-add or load operations were to be serialized at the memory of a real parallel computer, the advantages of parallel coordination algorithms would be lost since the critical sections would merely have been moved from the software to the hardware. The NYU Ultracomputer approximates the paracomputer's single cycle access to shared memory with a multicycle connection network. This message switching network, with the topology of Lawrie's [75] Ω-network, connects N = 2^D autonomous PE's to a central shared memory composed of N memory modules (MM's). Routing logic is based on the existence of a (unique) path between each PE-MM pair. Routing conflicts are handled by queues at each switch.

When concurrent loads, stores, and fetch-and-adds are directed at the same memory location and meet at a switch, they can be combined without introducing any delay (Gottlieb, Lubachevsky, and Rudolph [83]). Since combined requests can themselves be combined, any number of concurrent memory references to the same location can be satisfied in the time required for one central memory access from a single PE. This property permits the bottleneck-free implementation of coordination protocols in which many PE's access the same variables ("hot spots"). Since simulations have shown that even moderate hotspot traffic can severely degrade all memory access, not just access to shared coordination locations, logic at the switches to combine memory requests is a crucial part of the design (Pfister and Norton [85]).

For machines with thousands of PE's, the communication network is likely to be the dominant component with respect to both cost and performance. The Ω-network topology and the switch design described below achieve the following objectives (for performance analysis see Kruskal and Snir [83] and Spirakis [84]):

1) Bandwidth linear in N, the number of PE's.
2) Latency, i.e. memory access time, logarithmic in N.
3) O(N log N) identical components.
4) Routing decisions local to each switch; thus routing is not a serial bottleneck and is efficient for short messages.
5) Concurrent access by multiple PE's to the same memory cell suffers no performance penalty.

2. Designing a high-performance network

We believe that the main constraints on network performance are the delays inherent in off-chip
communication to and from each VLSI switching node, rather than the rate at which information can be processed within each node. Therefore, significant amounts of logic can be added to each node without penalty when that logic would help avoid global signaling and reduce bottlenecks within the network.

An important design goal is to attain a bandwidth proportional to the number of PE's, so that network performance will not degrade for large numbers of processors. This has been achieved by a combination of three methods:

1) The interval between messages is the cycle time of a single switch, rather than the transit time of the entire network (the networked is pipelined). Since the number of stages in the network grows logarithmically, non-pipelined networks can have bandwidth at most $O(N/\log N)$.

2) Switch settings are not maintained while awaiting a reply (the network is message switched). The alternative, circuit switching, is incompatible with pipelining.

3) Queues are associated with each switch to allow concurrent processing of requests destined to the same port whenever possible.

In a message-switched network, it may appear that both the destination and return addresses must be transmitted with each message. However, only a single path descriptor field which contains an amalgam of the origin and destination addresses is required. Initially, this field is set to the destination address. At each switch, the high-order address bit selects the port to which the message is to be routed. Each switch replaces this bit with the number of the input port on which the message arrived and rotates the address one bit so that the routing bit for the next switch will be the new high-order bit. When leaving stage J of a D-stage network, the low-order J bits will be the high-order J bits of the origin address and the high-order D−J bits will be the low-order D−J bits of the destination address. Thus, when the message reaches its destination, the destination address will have been replaced by the origin address.

The gate count required to implement this switch does not seem to preclude a one-chip implementation; the main impediment is the high pin count required. Therefore, each message will be split into several packets and one of two methods will be used to transmit these packets across the network. A bit-sliced implementation in which different components are handling different packets of one message can be used (transmission of messages is "space-multiplexed"). Or one can time-multiplex the transmission of successive packets of a message to the same component.

Space-multiplexing enables 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, as routing decisions and combining decisions have a global effect. This is likely to further increase the complexity of such implementation and to slow down the switch cycle. For MOS technologies, the off-chip delays impose an especially high overhead, as the on-chip logic can be quite fast.

If time-multiplexing is used, several cycles are required to transmit each message. However, the internal logic of the switch can be pipelined so that messages do not have to be assembled at each switch, but rather can be handled on a per-packet basis. Thus, when queues are empty, there is only one cycle delay per switch for each packet. Therefore, time-multiplexing adds an additive term to the delay rather than a multiplicative factor. Note however that queuing delays increase multiplicatively with the multiplexing factor, so that the performance of the network under heavy load may be seriously impaired (see Kruskal and Snir[83]). In the current design we have chosen to use time-multiplexing, so that each message is divided into a packet containing the path descriptor, address and opcode, plus one or more data packets.

The interswitch communication protocol is a message-level protocol, not a packet-level protocol. Message transmission occurs at consecutive cycles; the transmission of a message cannot be halted in the middle. A cycle is defined to be "even" for a given switch if the parity of the cycle is the same as the parity of the stage to which the switch belongs, so that cycles which are even for a switch are odd for its predecessors and successors. The reception of a message at each switch starts only at even cycles; the transmission of a message starts only at odd cycles. Note that this protocol implies an even number of packets per message. A switch accepts a new message only if the available space in the corresponding queues guarantees that it will be able to receive the entire message.

Two control signals are associated with each set of data lines. A sender asserts the Data Valid line when it wishes to initiate a message transmission. Independently, a receiver asserts the Data Accept line when it is able to accept a new message. A message transfer starts only if both Data Valid and Data Accept are set and the cycle parity is correct. These control signals can be set ahead of time, since they are ignored during cycles when a message transfer cannot be started. Note that this is not strictly speaking a handshaking protocol: Data Accept is not an answer to Data Valid, nor an acknowledgment, but is issued independently and simultaneously. The sender is asserting the data on the data lines whenever Data Valid is set. If the Data Accept signal was present, it assumes the data has been accepted and proceeds with the next packet. No provision for retry is necessary.

3. Switch structure and components

An individual network switch is a 2x2 bidirectional routing device (see figure 1). Goals in building the switch are:

1) No interference between distinct data paths. A new message can be accepted at each input port provided queues are not full; a message destined to leave at some output port will not be prevented from doing so by a message destined to leave it at another output port.

2) Minimal delay in forwarding messages when queues are empty. A packet entering a switch with empty queues when no other message is destined for the same output port is transmitted to the next stage at the next cycle.
The capability to combine and de-combine memory requests should not unduly slow the processing of requests which are not to be combined.

Associated with each output port in the PE to MM direction is a combining queue capable of accepting a packet simultaneously from each input port. Requests which have been combined with another request are sent to a wait buffer at the same time that the combined request is forwarded to the next stage of the network.

In the MM to PE direction, a reply enters both the wait buffer associated with the input port from which it came and the non-combining queue associated with the output port (towards the PE). An associative look-up is performed in the wait buffer to determine if the reply was to a request that had been previously combined and, if so, the de-combined reply is sent to the appropriate non-combining queue. Each queue has four inputs since messages leaving a given output port in the MM to PE direction may come from either input port and from either wait buffer.

It is clear that a combining queue capable of accepting two simultaneous inputs is significantly more complex than a one-input, one-output queue. Simpler one-input queues can be used in the switch, but since a separate server is required on each input-output path to avoid interference between distinct data paths, four are needed in each switch: one for each input/output pair. In this configuration, combining will not occur in the first stage of the network, and the last stage of combining will be lost unless an additional combining queue is placed in front of each MM.

For simplicity, we will be presenting the design of the one-input, one-output combining queue below.

4. Implementing a combining queue

Our design for a combining queue is an enhancement of the VLSI systolic queue of Guibas and Liang [2]. 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 two 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 goal we resort to an increased number of global control signals. A comparator is added to the basic queue structure to detect requests which are to be combined.

The queue consists of three columns: an IN column, an OUT column, and a CHUTE column (see figure 5). Packets added to the queue enter the IN column and move into the adjacent slot in the OUT column if it is empty. If the slot in the OUT column is full, and the address of the new message doesn’t match the address of the message in the OUT slot, the new message will move up the IN column each cycle until the adjacent slot in the OUT column is empty, and then move over to the OUT column and begin moving down the OUT column. Should the 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.

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 to the OUT port and the appropriate information to allow de-combining to the wait buffer. Packetizing of messages somewhat complicates this logic, since only address packets are compared. See Dickey, Kenner, Solworth & Snir[85] for details.

For combined loads and combined stores the wait buffer need only receive the op-code and the address (including the return address) of one of the requests. For two combined fetch-and-add operations, the sum of the two increments is forwarded to the next stage along with the address from the OUT column, while the increment from the OUT column is stored with the address from the
CHUTE 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 receive the original value plus the increment from the first request.

We note three properties of this queue's structure as they relate to the goals previously discussed:

1) One packet exits the queue at each cycle provided the queue is not empty and the switch in the next stage can receive it.

2) A new packet can be accepted each cycle provided the queue is not full.

3) Packets will transit the queue in one cycle if the queue is empty and the next stage can receive them.

A schematic for a single data bit (containing one slice of the IN, OUT, and CHUTE columns) is shown in Figure 6. 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 computed during the second clock phase. It is used during that phase to indicate whether the IN or CHUTE slots will be marked as occupied.

We have also had a sample of the PE-MM portion of a 2-by-2 combining switch (6-bits wide) fabricated. This switch is composed of four 1-input combining queues described above. These parts have recently arrived and are currently being evaluated.

We plan to convert our designs to a newly available scalable double-metal CMOS process to be provided by MOSIS with minimum feature sizes of 1.6 microns.

References

Susan Dickey, Richard Kenner, Marc Snir and Jon Solworth, "Building the Ultraswitch, a VLSI Network Node for Parallel Processing", in preparation.


Acknowledgement. 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-AC02 76ER03077, and in part by the U. S. Army Research Office, under contract number DAAG29-83-G0009.

5. Results to date

In preparation for the design of a complete combining switch chip, we have designed several chips which have been fabricated in 3- and 4-micron NMOS processes by DARPA's MOSIS facility.

One of these was an 11-bit wide 2-input, 2-output non-combining switch containing approximately 7500 transistors. Samples of these parts operated correctly at a clock speed of 23MHz. Propagation delays from clock to output were measured at approximately 25ns and power dissipation was approximately 1.5W. A 4-input, 4-output switch was constructed using 4 of these parts and functioned as expected. We plan to construct an 8-input, 8-output 33-bit wide bidirectional non-combining omega network using 72 of these parts.