FORTH-ICS / TR-127 December 1994

Pipelined Memory Organization for High Performance Switching and Buffering

M. Katevenis, P. Vatsolaki, A. Efthymiou

ABSTRACT: Switch chips are building blocks for high speed present and future computer systems. Switches need internal buffering, to temporarily hold packets when output contention occurs. Contrary to a widespread impression, a single shared buffer is *not* more expensive than multiple input buffers (FIFO or non-FIFO), while it performs better and its control is simpler. We present a novel organization for such a shared buffer and the switching and cut-through circuitry around it, which is simpler and smaller than previous organizations, and which is particularly suitable for future high speed IC technologies. It is based on multiple memory banks, addressed in a pipelined fashion. The first word of a packet is transferred to/from the first bank, followed by a "wave" of similar operations for the remaining packet words in the remaining banks. An FPGA-based prototype has been built, while standard-cell and full-custom designs are in progress. Simulation of the full-custom design indicates that the 64 Kbit shared buffer of an 8x8 switch will operate at 1 Gbps/link (worst case) in a conservative 1 μm CMOS technology, and will fit in 45 mm^2 including crossbar and cut-through.

Pipelined Memory Organization for High Performance Switching and Buffering

Manolis Katevenis, Panagiota Vatsolaki, and Aristides Efthymiou

Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)
P.O.Box 1385, Science and Technology Park, Heraklio, Crete, GR-711 10 Greece
E-mail: katevenis@ics.forth.gr, Tel: +30 (81) 391664, Fax: +30 (81) 391661

Technical Report FORTH-ICS / TR-127 – December 1994

Work performed under ESPRIT Contract 6253 "SHIPS"

ABSTRACT: Switch chips are building blocks for high speed present and future computer systems. Switches need internal buffering, to temporarily hold packets when output contention occurs. Contrary to a widespread impression, a single shared buffer is *not* more expensive than multiple input buffers (FIFO or non-FIFO), while it performs better and its control is simpler. We present a novel organization for such a shared buffer and the switching and cut-through circuitry around it, which is simpler and smaller than previous organizations, and which is particularly suitable for future high speed IC technologies. It is based on multiple memory banks, addressed in a pipelined fashion. The first word of a packet is transferred to/from the first bank, followed by a "wave" of similar operations for the remaining packet words in the remaining banks. An FPGA-based prototype has been built, while standard-cell and full-custom designs are in progress. Simulation of the full-custom design indicates that the 64 Kbit shared buffer of an 8x8 switch will operate at 1 Gbps/link (worst case) in a conservative 1 μm CMOS technology, and will fit in 45 mm^2 including crossbar and cut-through.

KEYWORDS: *crossbar switch, shared buffer switch architecture, multiported buffer, pipelined memory, gigabit VLSI switch buffer.*

M. Katevenis and A. Efthymiou are also with the University of Crete, Department of Computer Science.

CONTENTS

This text is available in Postscript form, by ftp, from server “ftp.ics.forth.gr”
Login as “anonymous”, give your e-mail address as password
Directory: “tech-reports/1994”
Files: “94.TR127.PipelinedMemory.README”, “94.TR127.PipelinedMemory.ps.Z”
Source Directory: ~kateveni/telegraphos/tr127

1. Introduction: Switch Buffers

Switches are basic building blocks for present and future computer and communication systems. They are used to build the interconnection networks of large-scale parallel computers [AtSe88]. They are needed for building gigabit local area networks that will form the backbone of high performance distributed computer systems [Kung92]. Switches become useful even inside small-scale or uniprocessor systems at high clock rates, when point-to-point connections become necessary because buses run out of steam due to arbitration and turn-around delays [SCI92]. Switches are also used to build wide area communication networks. Yet, the architecture of switches is nowhere near a convergence to a certain ‘‘style’’, analogous to the convergence of processor architectures to RISC, for example. We feel that there is much need and much room for innovation and improvement in switch architecture, today. This work aspires to contribute in that direction.

A switch has a number of incoming and a number of outgoing links, connecting it to other switches or, through interface devices, to processors, memories, I/O, etc. The function of a switch is to receive packets of information from its incoming links, make routing decisions for each packet, and forward each of them along the appropriate outgoing link(s). Since the destinations of the incoming packets are usually not known or scheduled in advance, *contention* (collisions) will often arise: multiple packets arriving (almost) simultaneously through different incoming links may have to be forwarded along the same outgoing link. When contention occurs, some of the incoming packets have to *wait* inside the switch, in some form of *buffer memory,* and go out at a later time.

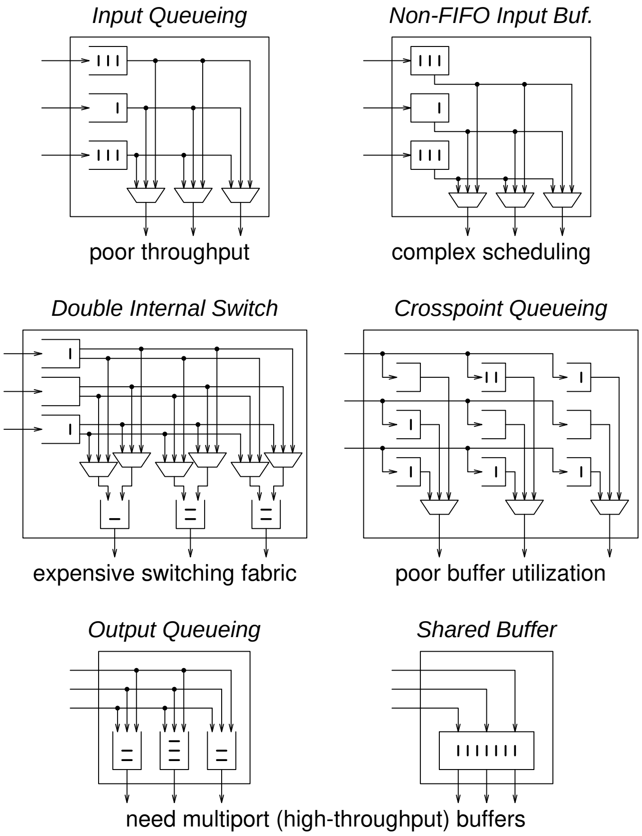
The subject of this paper is the efficient implementation of such buffer memory. Performance-wise, the best buffer architecture is *shared buffering,* which, however, many designers consider to be expensive to implement. In this paper, we present a novel organization, which we call *pipelined memory,* that can be used to implement, among others, shared buffering at low cost. We begin by reviewing the most important switch buffer architectures.

1.1 Switch Buffer Architectures

Figure 1 shows the basic options for the organization of the buffer memory in a switch, using 3 by 3 switches as example. *Input queueing* is the simplest but worst performing architecture. At each occasion, only the front (head-of-line) packet of each input queue is considered for possible routing to its destined output; if it collides with the front packet from another queue, then it has to wait, and so do all the packets behind it in its queue, even if the latter are destined to currently idle outputs. This is known as *head-of-line blocking;* because of it, a switch with equal input and output throughput, and with independent, randomly destined packet traffic, saturates at about 60% of the link capacity [HlKa88]. When the traffic is bursty and the bursts are larger than the buffers – for example with multi-flit packets in wormhole routing – saturation occurs even sooner: in [Dally90], with 20-flit messages and 16-flit buffers, simulation showed saturation at about 25% of link capacity ([Dally90]: fig. 8, case of 1 lane).

To improve the performance of input queueing, one may drop the requirement that the buffers behave like FIFO queues, thus eliminating head-of-line blocking. In this *non-FIFO input buffering* architecture, we maintain the property that the input buffers have a single read port. This organization provides quite better performance than input queueing [HlKa88] [AOST93] [TaCh93] [LaSe94], although output throughput may be less than optimum with practical schedulers, and latency is worse than optimum whenever some output remains idle because all buffers that contain packets for it are busy with forwarding other packets to other outputs, each. Buffer implementation is more complicated than with input queueing [TaFr88]. Also, a more complicated scheduler is needed, because now the scheduling of each output depends on the scheduling of the other outputs: only one output port is allowed to use each buffer at any given time. Schedulers for this architecture have been studied e.g. in [AOST93], [TaCh93], [LaSe94].

Another method to improve the performance of input queueing is to provide an internal *switching fabric of higher throughput* than that of the incoming links [PaBr93]; figure 1 shows an example with a double internal switch. By emptying the input queues faster than they get filled, we get the equivalent of input queueing operating at a reduced incoming load, which yields an acceptable performance. Since the outgoing links will usually have the same throughput as the incoming links, packets will in general accumulate after the internal switching fabric, so this organization also needs output queues. The cost of this architecture is in the additional queues, in the requirement for the queues to be three-ported (rather than two-ported as above), and in the increased throughput of the switching fabric.



Input Queueing

poor throughput

Non-FIFO Input Buf.

complex scheduling

Double Internal Switch

expensive switching fabric

Crosspoint Queueing

poor buffer utilization

Output Queueing

Shared Buffer

need multiport (high-throughput) buffers

Figure 1: Switch queueing and buffering architectures 4

To reach maximum throughput, one can go to *crosspoint queueing:* each arriving packet is placed into the single queue that corresponds to its incoming and outgoing link combination. Every outgoing link can now be kept busy, by selecting and forwarding a packet to it, independent of what the other links do. This architecture achieves optimum utilization of link throughput, but has the disadvantage of needing many buffer memories, and of requiring a total memory capacity that is considerably higher than the other architectures in order to reach a similar level of performance.

Each buffer memory in the above architectures needs a throughput twice or three times the link throughput. The two next architectures need higher buffer throughput, but do not suffer from the disadvantages of the former architectures. In *output queueing,* each buffer is associated with one outgoing link, and must be able to accept, in the worst case, packets arriving simultaneously from all inputs. Output queueing achieves optimum utilization of link throughput, like crosspoint queueing, but it yields higher buffer memory utilization. Still, its memory utilization is lower than what shared buffering achieves.

Shared buffering uses only one buffer memory for the entire switch; this must have a throughput at least as much as the aggregate throughput of all incoming and all outgoing links.

The shared buffer architecture yields the same optimum utilization of link throughput as crosspoint and output queueing, and, in addition, it achieves the best buffer memory utilization. For example, according to [HlKa88], a 16×16 switch with incoming link load of 0.8 (uniformly distributed destinations), needs the following buffer sizes in order to achieve packet loss probability of 0.001: *(i)* 86 packets under shared buffering (5.4 per output); *(ii)* 178 packets under output queueing (11.1 per output); and *(iii)* 1300 packets under "input smoothing" (80 per input) – input smoothing has considerable similarities to non-FIFO input buffering. Concerning latency, the simulations in [AOST93, fig. 3] showed output queueing (or equivalently shared buffering) to be about twice faster than input buffering (with the particular scheduling algorithm that that paper uses) for loads between 0.6 and 0.9. While the shared buffer architecture is performance-wise the best, many designers consider it as expensive because of the high throughput requirements on the buffer – see e.g. [TaFr88, p. 344], the implicit assumption behind [AOST93], [LaSe94], and the considerable number of real switches using input queueing/buffering, like e.g. [Arno89], [HoLa93], [C104], [Souz94].

As mentioned earlier, in this paper we present a novel *pipelined memory* organization, that can be used to implement shared buffering at low cost. More generally, it can implement high-throughput (multi-port) buffer memories for switches at low cost. Thus, it is also appropriate for output queueing, or for a – more interesting – mixture of crosspoint queueing and shared buffering. We can call this mixture *block-crosspoint buffering;* it consists of a number of shared buffers, each dedicated to service a certain subset of incoming and outgoing links. One goal of this paper is to show that the silicon area needed to implement shared buffering will usually be smaller than the area needed to implement input buffers with complex non-FIFO scheduling. The next subsection reviews the existing techniques for implementing high-throughput multi-port memories.

1.2 Existing Techniques for Multi-Port Buffer Memories

True multi-port memory is very expensive, because each storage bit must have multiple word lines and bit-lines, thus raising significantly the memory cost per bit. As a result, for almost every application other than the smallest memories (e.g. processor register files), multiple *interleaved banks* of single-ported memory are used in place of multi-port memory. The buffers in the "Knockout Switch" [YeHA87] use this technique, in an output queueing architecture (in that particular case, the bank addresses differ by at most 1 from each other in any given enqueue time slot). The "Prelude" switch [DeCS88] shared buffer memory operates as an interleaved memory when cells are extracted from it (in that case, neighboring bank addresses differ by 1 in succeeding clock cycles).

When an interleaved memory is used as FIFO queue or as buffer for multi-word packets, most of the accesses to it are sequential. This eliminates most or all of the bank conflicts, and also makes it possible to have a single address register for all memory banks, effectively merging the multiple memory banks into a single *wide memory.* This works best when the size of the items accessed – the packet size in switches – is a multiple of the wide memory word size, so that all accesses are aligned on wide-word boundaries. A wide memory is particularly feasible and attractive inside a very large scale integrated (VLSI) chip, because storage arrays inside chips are naturally wide: their aspect ratio is usually 1:1 or 1:2, thus e.g. a 32 Kbit memory is usually organized as a 256 by 128 rectangular array. The IBM "Vulcan" switch [Stun94] uses a wide memory shared buffer. The "Prelude" switch [DeCS88] buffer memory operates as a wide memory when cells are stored into it.

We have used the wide memory organization in an earlier switch chip design [KaSC91], in a shared buffer architecture. Figure 2 shows that buffer, with its associated input and output latches, drivers, and cut-through circuitry, for a simple case of a 2×2 switch. We assume that packets are transferred on the links in the form of multiple words of w bits each, one word per cycle time. The memory cycle time is assumed to be equal to the link cycle time. In the simple example of figure 2, the link throughput is 1 word/cycle, the packet size is 4 words (or a multiple thereof), and the width of the buffer memory is 4 words ($4w$ bits) – hence the memory throughput is 4 words/cycle, which is equal to the aggregate incoming and outgoing link throughput.

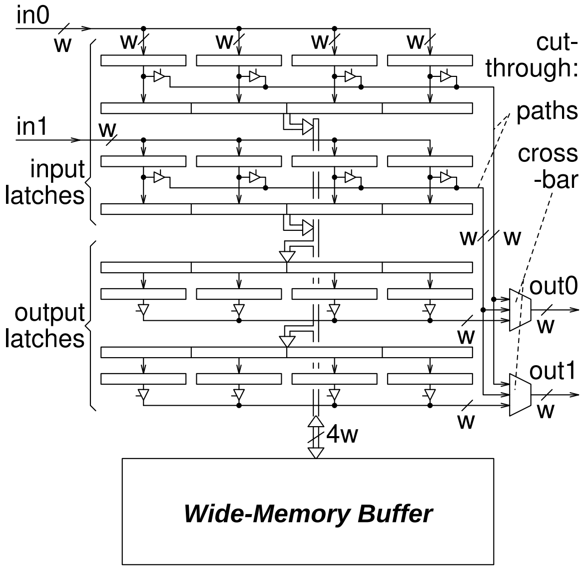


Figure 2: Input, output, and cut-through overhead in a 2x2 shared-buffer wide-memory switch

The relative complexity of the I/O circuitry required with the wide-memory organization stems from the fact that each operation has to be performed on an entire packet, and thus it can only be performed after the entire packet has been assembled. Just after an entire packet has been assembled in the first row of input latches, a new packet may start arriving on the same link, and it, too, has to be held somewhere. Hence, *double buffering* is needed on the input side, because it is *not* possible to guarantee that the wide memory will be available for storing the packet into it in precisely the desired cycle (since packet arrivals are not synchronized on the various links: they are not guaranteed to always start and finish on different cycles for the different links). Double buffering was also used in the outgoing link circuits in [KaSC91], as shown in figure 2, in order to allow transmissions of packets on the different links to start in the same cycle, if desired; for the output, this is a feature – not a requirement.

Cut-through refers to the capability of the switch to start forwarding a packet along its desired outgoing link after the header of the packet has been received, but before the entire packet has arrived; it applies when the outgoing link was idle when the header arrived. Cut-through capability is essential for modern low-latency switches. Since a packet cannot be stored into the wide memory before all of it has arrived, and since cut-through must start before that time, it follows that cut-through must be performed through paths *other than* the wide memory or its data bus. Figure 2 shows the paths that are necessary for cut-through in the wide-memory architecture: additional tri-state drivers, bus wires, and an output crossbar are needed. Notice that the paths provided in this design do not suffice in order to be able to initiate cut-through between the time a packet is transferred into the second row of input latches and the time when it is transferred onto the wide memory bus.

1.3 This Paper

As discussed in section 1.1, shared buffering is the best switch buffer architecture (and block-crosspoint buffering comes close), but many switch designers have avoided it by fear that the high-throughput buffer that it requires would be too expensive. However, there exist techniques for implementing such high-throughput buffers, and section 1.2 reviewed them. Notice that the required width of the wide memory can*not* be considered as excessive: it is equal to the sum of the widths of the buffer memories required in a switch that follows the more popular input-queueing or non-FIFO input-buffering architectures. Indeed, an $n \times n$ switch with throughput 1 word/cycle/link needs n input buffer memories that are either 2-ported or 2-word wide each. If all these input buffer memories are placed next to each other, a (single-port) wide memory of width $2 \cdot n$ words results, which is precisely the width required by the shared buffer architecture.

One source of cost for the wide memory technique is its peripheral circuitry: input, output, and cut-through latches, drivers, and paths (figure 2). The contribution of this paper comes precisely here: a novel buffer memory organization is proposed, which is something intermediate between interleaved and wide memory, significantly reduces the cost of that peripheral circuitry, and also simplifies control. Section 2 describes this *pipelined memory* organization, and section 3 lists the additional reasons why this organization is particularly advantageous for current and future VLSI technologies.

Switch designers that opt for input queueing/buffering architectures may also be afraid of the requirement placed by shared buffer implementations that packet size must be a multiple of the (wide) memory width (section 2.3 shows how packet sizes of half the memory width can also be accommodated). We believe that these fears are groundless, or that their reason will disappear with the evolution of high-speed networks. In fact, we believe that high-speed networks will converge to using fixed-size packets (cells, or flits), just like processors converged to using fixed-size instructions (RISC). ATM, with 53-byte fixed-size cells (packets), was a big step in that direction. Concerning actual packet size and memory width numbers, let us note that packets as small as 256 to 512 bits (ATM cells are 424-bits wide) work fine with memories as wide as 256 to 1024 bits (using the technique of section 2.3 for 1024 bits). With an (on-chip) memory cycle time of 5 ns, which is reasonable in CMOS today (see also section 3), this yields a shared buffer throughput of 50 to 200 Gbits/s (12 to 25 GBytes/s) – enough for 16 incoming and 16 outgoing links in the Giga-*Byte* per second range, each. If more links or more throughput is desired, one can always go to block-crosspoint buffering.

We have used the pipelined memory organization for the switch of our Telegraphos project [Kate94]. Three prototypes of this switch, in different technologies, have been or are being implemented; these are described in section 4. Finally, section 5 compares the pipelined memory organization with non-FIFO input buffering, and presents our conclusions.

2. The Pipelined Memory Organization

The wide memory (figure 2) is a simplification of the interleaved memory organization, and this simplification is applicable when most accesses are sequential. Figure 3 shows the *pipelined memory* organization proposed in this paper, which is a different and better simplification, applicable in the same cases. The pipelined memory is like an interleaved memory where the address used by a bank is the same as that used by the previous bank in the previous cycle. The simplification of using the same address for subsequent banks in subsequent cycles is important because *(i)* it simplifies control relative to the interleaved case, *(ii)* it makes the VLSI implementation faster than wide memories (see section 3), and *(iii)* it greatly reduces the size of the peripheral circuitry relative to the wide memories, as will be seen below.

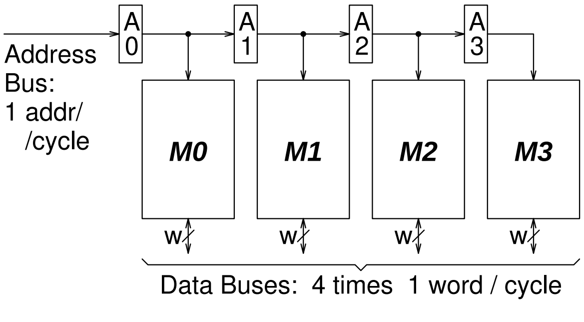


Figure 3: Pipelined memory organization

2.1 Switching and Cut-Through with the Pipelined Memory

The wave-like manner in which accesses to a pipelined RAM proceed matches well the serial character of packet arrivals and departures in switches. The multiple simultaneous accesses to a pipelined RAM, at various stages of completion each, match well the multiple packets arriving at and departing from a switch, all at the same time. As a result, the pipelined memory organization is better suited to switch buffer operation than the wide memory organization. This is illustrated in figure 4, which shows the same 2x2 shared-buffer switch example as figure 2, this time using pipelined memory. To explain the operation of this simple example, let us make the same assumptions as in figure 2: the memory cycle time is equal to the link cycle time, the link throughput is one word (w bits) per cycle, the number of pipeline stages is equal to the number of incoming plus outgoing links (four), and the packet size is four words. An incoming packet arrives in four consecutive cycles. As it arrives, it is written into the corresponding input buffer registers, one word at a time. Imagine this as a *wave* of new packet words entering into the input buffer registers (and overwriting the old data in them). During *some* cycle after the arrival of the first word of this packet, and before the arrival of the first word of the next packet, this first word is written into the first memory stage, $M0$. In the three subsequent cycles, the next three words of the packet will be written into the next three memory stages, $M1$, $M2$, and $M3$, at the rate of one word per cycle. Think of this as another wave of words being written into the buffer memory from left to right. All words of the same packet are written into the same address in each of the memory stages. The tail of the packet may be written into the last memory stage after the head of a new packet has entered into the first input buffer register. This does *not* lead to any loss of information, since the wave of storing the old packet into the buffer memory had started before the new packet wave started overwriting the input registers, and since both waves proceed at the same rate from left to right. Thus, *no double buffering* is needed here, unlike the [KaSC91] wide memory organization.

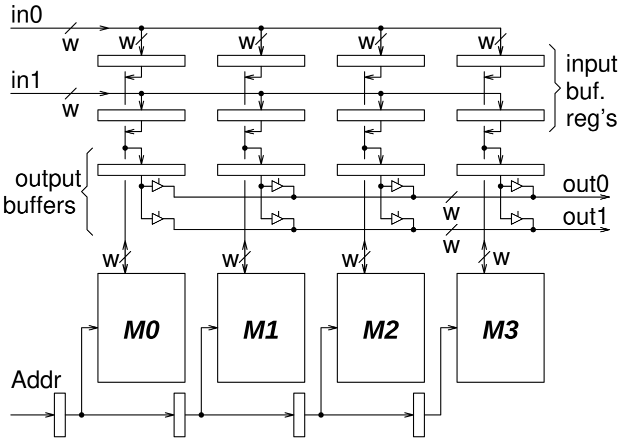


Figure 4: The 2x2 shared-buffer switch of figure 2, using pipelined memory

On the output side, again, the pipelined memory needs one level of buffering less than the wide memory organization. As noted earlier, the wide memory organization can operate with only one level of output buffer registers per outgoing link, although the [KaSC91] design uses two levels. Figure 4 shows the simpler and more economical case: there is only one row of output buffer registers *shared* among all outgoing links. The restriction here is that no two outgoing links can start sending out packets in the same cycle. Output operation is as follows. The first word of the outgoing packet is read out of $M0$ and placed in the leftmost output buffer register. In the next cycle, this register drives the desired outgoing link (it can also drive more than one links, thus providing multicasting); at the same time, the second word of the packet is read out of (the same address of) $M1$. The wave of reading from the memory stages and transmitting successive words proceeds left to right, again at the rate of one word per cycle. Every cycle, a new operation wave can start from $M0$: we can start reading another outgoing packet and send it on another outgoing link, or we can start writing a new incoming packet into the buffer memory.

In the pipelined memory organization, *cut-through* is straightforward – *no additional* circuitry is required. This represents the major area savings relative to wide memory, since the latter needs an extra set of buses on the input side and an extra crossbar in order to implement cut-through. Any cycle after the first word of a packet has arrived and was written into its leftmost input buffer register, this register can drive the data bus of $M0$, as is the normal procedure for storing the word into $M0$. At the same time, this word can also be loaded into the leftmost output buffer register, and in the very next cycle it can be sent out on an outgoing link. All this may well happen before the tail of the packet has arrived into the switch, and thus it constitutes "cut-through". Since the wave of packet arrival started before the wave of packet departure, and since they both proceed left to right at the same rate, transmission of the packet's tail will only be attempted after that tail has arrived into the switch and has been written into the rightmost input buffer register.

2.2 *Control of the Pipelined Memory Switch*

Control of the pipelined memory and associated input and output registers and drivers is simple and straightforward. The first reason is that each pipeline stage performs exactly the same operation as the previous stage in the previous cycle. Thus, the control circuit only needs to generate the control signals for the *first* memory stage. For an $n \times n$ switch, these signals are: n load enable's for the input buffer registers; n output enable's for these same registers (at most one of these n signals can be active at a time); write enable, read enable, and address for memory $M0$; the load enable for the output buffer register (one signal); and n tri-state enable's for the output buses. The control signals for each subsequent stage are delayed versions of the control signals of the previous stage.

When the control section of the switch sees a new packet of $2n$ words arriving on an incoming link, it activates the load-enable signal of the corresponding register of stage 1. Since new packets may arrive simultaneously on any number of incoming links, the n load-enable signals of stage 1 must be controllable independently from each other. During one of the $2n$ cycles after the first word of the packet was placed in the first input register, that register will have to drive the bus of its stage, so that this word is transferred to $M0$, or to some output bus(es), or both. The precise cycle when this is done depends on the resolution of contention for access to the buffer memory by the incoming and the outgoing links. Normally, higher priority is given to the outgoing links, because any delay to supply data to an outgoing link will lead to idle time on that link, while delays to store incoming packets into the buffer memory have no direct consequence on the switch performance. Since there are n outgoing and n incoming links, and since each of them may initiate at most one read or write access to the buffer memory during every $2n$-cycle interval, it follows that enough resources exist to service all links.

During a cycle when the contention resolution circuit decides to service a certain incoming link, the control circuits activate the driver of the corresponding input register and the write-enable of $M0$, and also provide the address where the packet should be written (the way in which this address is generated is unrelated to the pipelined memory architecture). When the contention resolution circuit decides to service an outgoing link, the control circuits activate the read-enable of $M0$, they provide the address from which the packet should be read, and they activate the output buffer load enable; in the next cycle, the proper output bus driver is enabled.

Owing to the pipelined organization, *cut-through* operation is straightforward, without any extra control. The $M0$ read that initiates the transmission of a packet can be performed any cycle after the *first* word of the packet was written into the data buffer, even if not all other words of the packet have been written yet. It is also possible to transmit the packet at the same time that it is being written into the memory; in this case, writing the packet into the memory is optional.

2.3 Operation with Half-Width Packets

Although in the straightforward pipelined memory organization the packet size must be an integer multiple of the total width of all memory stages, packets of half this size can also be handled. This is achieved as follows. Consider a shared-buffer $n \times n$ switch, with $2n$ pipelined memory stages, as above. Let us consider the shared buffer operation when the packets are of size n words each. The peak throughput of each link is one packet every n cycles. Since there are n incoming and n outgoing links, the shared buffer must be able to initiate one write operation of one incoming packet *and* one read operation of one outgoing packet in each and every cycle. The shared buffer will consist of *two* pipelined memories, with n stages each (for a total of $2n$ stages, as mentioned above). Each packet is stored into one or the other of these two memories. In each and every cycle, one read operation of one outgoing packet is initiated from one of the two memories – whichever one the desired packet happens to be in. In the same cycle, one write operation of one incoming packet must also be initiated; this will be initiated into the *other* one of the two memories. The only restriction is that empty space for incoming packets must exist in both halves of the shared buffer, since it is not known a priori into which of the two halves each incoming packet will go.

3. Pipelined Memory Optimizations for VLSI

Figure 5(a) shows the floorplan of memory in integrated circuits. The storage cells are arranged in a rectangular array, and are connected to the grid of word line and bit line wires that run through the array. Memory accesses can be made to one or more of the bits in a row at a time. The row decoder selects the desired row, based on the address provided to it. Since we are interested in wide memories, we will assume, for simplicity, that accesses refer to the entire selected row (but this is not a prerequisite for the pipelined memory organization). Wide memories (as in figure 2) will have long word lines. However, in high-speed memories, the word lines that are longer than a certain limit present performance problems, due to their "*RC*" delay (the Ohmic resistance, *R*, of a long metal wire is non-negligible, and, in connection with the parasitic capacitance, *C*, of itself and of the transistor gates that it drives, it causes considerable delays that increase with the square of the length). When the width of a memory is beyond a certain limit, its cycle time can be reduced by organizing the memory as several distinct arrays, each of them having its own address decoder, as illustrated in figure 5(b); the same (encoded) address is supplied to all the decoders.

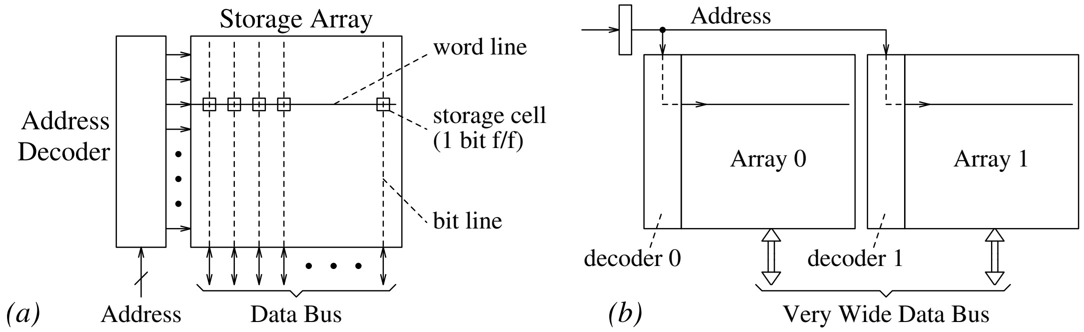


Figure 5: Wide memories in VLSI: (a) general layout, (b) floor-plan of very wide memory

Figure 6(a) shows how traditional RAM arrays can be arranged in order to implement the pipelined memory of figure 3 in VLSI. Comparing figures 5(b) and 6(a), we see that the cost of the pipelined RAM array is similar to the cost of the wide RAM: both must have additional address decoders – either to avoid word-line *RC* delays, or to implement pipelined accesses. In many cases, a further optimization of the pipelined RAM implementation is possible, which is *not* possible for the wide RAM. This is shown in figure 6(b), and its key idea is that, since the arrays touch each other, the decoded address can be transferred from one stage to the next *directly* from word-line to word-line, without being re-decoded. Effectively, the word lines of all the stages that are on the same horizontal are connected through pipeline flip-flops into one long word line, which is activated in a "wave" fashion. If this long word line were a single wire, then its large *RC* delay would limit the memory cycle time. However, the wave-like operation that is characteristic of the pipelined memory turns this drawback into an advantage by controlling the rate of this delay, using pipeline flip-flops, and making it equal to the rate of transmission over the switch links. The organization of figure 6(b) replaces all but the first row address decoders in figure 6(a) with master-slave registers, which have one bit per word-line, each. This is advantageous when the flip-flop array is narrower than the decoder array, or when its power consumption is smaller; both of these will often be true, especially for memories with many rows (see section 4.3).

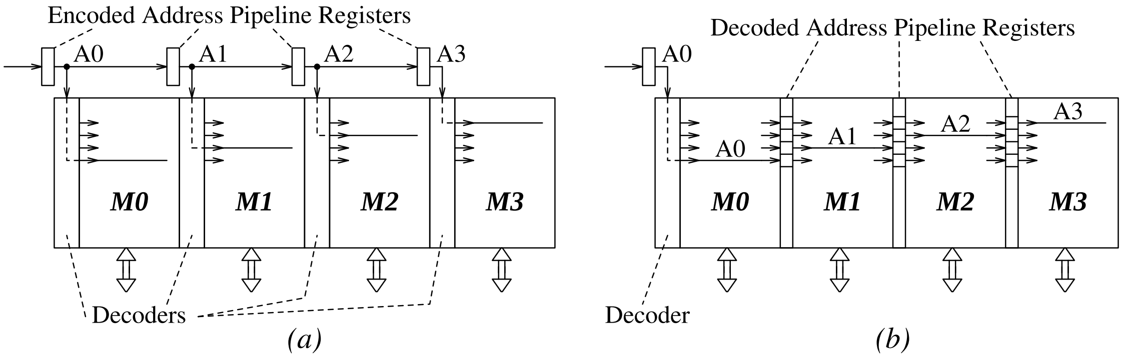


Figure 6: Pipelined RAM in VLSI: (a) traditional, (b) novel organization and floor-plan

Figure 7 shows a variation of the VLSI pipelined memory that will be appropriate in future very-high-speed IC technologies, when on-chip clock skew will be considerable relative to clock period, and when the buses around the shared buffer and the bit lines will have high *RC* delays. Even though these incoming and outgoing link buses have a limited capacitive load on themselves (compared e.g. to the RAM word lines), they will also suffer from *RC* delays in future high-speed technologies, because of the continued technology trend for shrinking geometries and increasing clock frequencies. At that point, the solution will be to pipeline these buses too, as shown in figure 7, "extra pipeline stage (a)". In this figure, only one of the input and one of the output buses that will be present in a shared buffer are shown. Each of these buses is divided into two pipeline stages in figure 7; more stages are also possible. The number of stages in all input buses must be the same, and it must also equal the number of stages in all output buses. At the point where each I/O bus is split by a pipeline register, one extra decoded-address pipeline register must be inserted into the pipelined RAM. Any word of a packet that is stored in memory stages $M0$ through $M3$, in figure 7, passes through 2 pipeline registers on its way in, and it passes through 3 pipeline registers on its way out. On the other hand, words stored in memory stages $M4$ through $M7$ pass through 3 pipeline registers on their way in, and 2 registers on their way out. Consequently, the total number of clock cycles spent in going through the I/O pipeline is the same (5 cycles) for all words of every packet; this observation is especially useful in understanding the timing of cut-through operation.

Another feature of the VLSI floorplan of figure 7 is that data *always flow in a single direction,* and hence control and clocking signals can also be made to always flow in the same direction. To

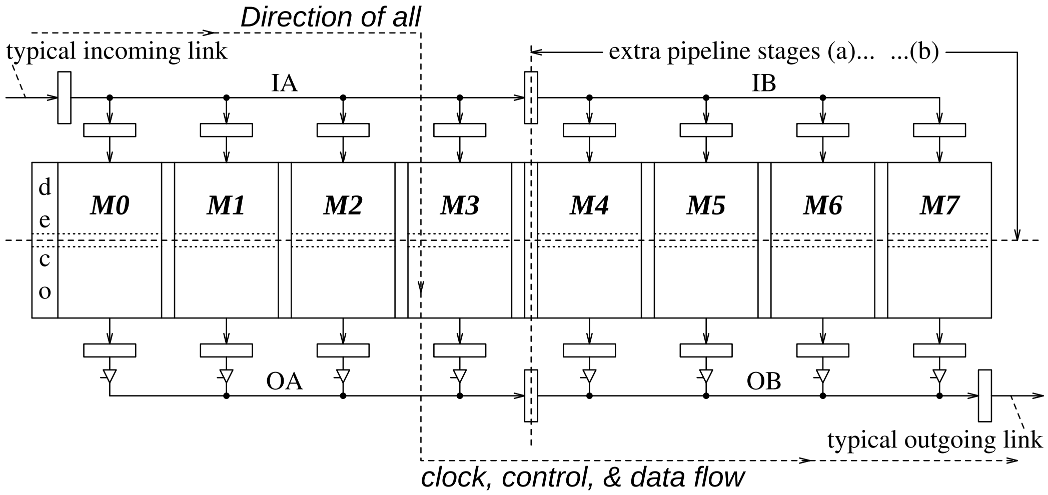


Figure 7: Pipelining the input and output buses of figure 4 when these are very long

achieve this, the outgoing and incoming links were placed on opposite sides of the memory arrays. By making all signals flow in the same direction and along paths of similar length, it is possible to match (to some degree) the *RC* delays of the wires that carry these signals, so that their *relative skew* is minimized, which will be crucial in the high-speed technologies of the future.

Lastly, if the memory arrays have many rows and the bit-line drive time becomes too long, bit lines can also be split into multiple pipeline stages, as shown in figure 7, "extra pipeline stage (b)", provided that all memory stages as well as the address decoder are also split in the same manner. This arrangement adds extra pipeline stages in the vertical flow; all packet words passing through the switch do so through an equal number of stages, thus preserving their relative timing. During memory write or read accesses, only one of the bit-line stages performs the access, while the others merely pass the information from top to bottom.

4. Example: the Telegraphos Switch Buffers

The work presented in this paper is being carried out in the context of the *Telegraphos* project, in which we design and implement switches and processor-network interfaces that can enable parallel processing on workstations clustered through gigabit LAN's [Kate94]. We are working in parallel on three prototypes of our Telegraphos system, using different technologies for each – FPGA, semi-custom ASIC, full-custom VLSI – and being at different stage of completion for each. The next three sub-sections describe the pipelined memory shared buffer that forms the core of the switches in these three prototypes.

4.1 Telegraphos I: FPGA-based Prototype

The Telegraphos I switch is a 4×4 crossbar implemented with field-programmable gate arrays (FPGA) and SRAM chips. It operates at 17 MHz, and each link carries 8 bits of packet data per clock cycle, i.e. 133 Mbps/link. The packet size is 8 bytes (64 bits), and the shared packet buffer consists of 8 pipelined stages; it is built out of 8 SRAM chips. The logic for access arbitration among the incoming and the outgoing links, as well as for control signal generation for the first pipeline stage, is contained in one Xilinx 3130PC84 FPGA, and it is approximately equivalent to 500 gates. The peripheral circuitry of the shared buffer (input and output registers and drivers, and control signal pipeline registers) can be thought of as constituting an 8-bit wide datapath; this datapath has been sliced into four 2-bit-wide slices, and was implemented in four 4 Xilinx 3164PC84 FPGA's, each of them containing the equivalent of 1500 gates. The PCB wiring is quite dense in the area of the shared buffer, requiring 4 signal layers with 0.2mm wide traces to route all the interconnections. Telegraphos I has been built and is currently under testing in our lab. Figure 8 is a photograph of the central part of its switch. The FPGA at the center is the shared buffer controller; the four FPGA's that form a cross around it are the 4 slices of the shared buffer datapath; and 2 SRAM chips in each of the four corners of the cross form the 8 pipelined memory stages.

Figure 8: Photograph of the Telegraphos I switch shared buffer

4.2 Telegraphos II: Standard-Cell ASIC Design

The Telegraphos II switch is a single-chip version of the Telegraphos I switch, operating at higher speed. This ASIC uses the 1 μm CMOS standard-cell process of European Silicon Structures Inc. (ES2), and its design is currently in the stage of placement and routing. Like Telegraphos I, the Telegraphos II switch is also a 4×4 crossbar, but each link is expected to operate at 400 Mbps. The expected clock frequency is 25 MHz, and each link is 16-bit wide inside the chip (the off-chip links are double-cycled, carrying 8 bits every 20 ns). The packet size is 16 bytes (128 bits). The shared packet buffer has eight pipelined stages; each of these is one 256×16 compiled SRAM megacell. The eight SRAM megacells occupy an area of 20 mm^2 (8×2.5), and their peripheral circuits (input/output registers/drivers) occupy approximately another 20 mm^2 (8×2.5).

4.3 Telegraphos III: Full-Custom Pipelined-SRAM Shared Buffer

A pipelined memory shared buffer is a very regular VLSI structure. In a semi-custom ASIC, it is a pity to only be able to exploit this regularity for the SRAM portions of it (as compiled megacells) and not for the entire buffer with its peripheral circuits. To show the feasibility and advantages of a full-custom layout of the entire buffer, we are making such a design, for the same fabrication technology as that of the Telegraphos II switch. This full-custom shared buffer with peripheral circuits will soon be fabricated inside a test chip, and it will later form the core of a Telegraphos III switch chip. As we will see below, by going from standard-cells to full-custom, the peripheral circuits of the shared buffer gain approximately a *factor of 12* in speed, capacity, and area: full-custom has twice the number of links, the clock is 2.5 times faster (62.5 MHz worst case)[1], and the peripheral circuit area is 2.4 times smaller (8.5 mm^2).

Figure 9 shows the floor plan of this Telegraphos III pipelined memory shared buffer, which we have fully laid out and extensively simulated using HSPICE. It will form the core of an 8×8 crossbar switch with worst-case throughput of 1 Gbps/link (typical case: 1.6 Gbps/link). On-chip, each link consists of 16 wires, carrying 16 bits every clock cycle (10 ns typical, 16 ns worst case). The buffer consists of 16 pipelined stages, and it provides storage for up to 256 packets of 256 bits each.

The buffer is designed in the same technology as the Telegraphos II switch, which is a 5 Volt, single poly, double metal CMOS process with 1 μm minimum drawn feature size, and 0.8 μm effective channel length. For a 1 μm CMOS technology, this is *not* a very fast process. To characterize its performance, consider an inverter with a 6.5/1.0 pull-up transistor and a 2.5/1.0 pull-down transistor, driving four identical inverters, and being driven by a signal with 0.5 ns rise/fall time. This inverter has a rise time of 0.7 / 1.5 ns (typical/worst case), and a fall time of 0.7 / 1.3 ns. The mid-point delay for the above circuit is 0.4 / 1.0 ns for the falling output, and 0.4 / 1.1 ns for the rising output. In our simulations, the worst case conditions are: slow transistor

---

[1] the speed gain refers to the entire buffer – peripheral circuits *and* SRAM.

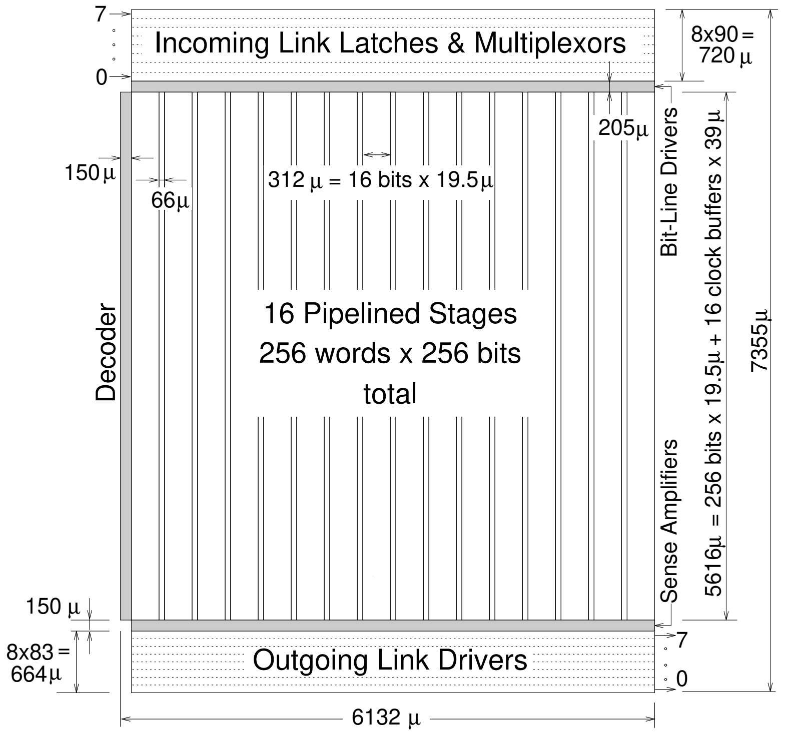
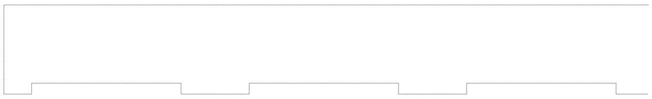


Figure 9: Floor plan of 16 Gbps, 64 Kbit pipelined buffer in 1 µm full-custom CMOS

models, high parasitics, supply voltage of 4.5 V, and 125 °C junction temperature.

As we see from the floor plan of figure 9 (which is drawn to scale), a decoded address pipeline register is 2.3 times smaller than the normal address decoder for the SRAM. The largest gains relative to the standard-cell version come from the use of dynamic latches for the incoming links, the use of precharging for all long wires that have multiple drivers (e.g. outgoing link buses), and the overlap of the peripheral circuitry with the link wire routing. The peripheral circuitry area is just about 1.5×6.0 mm^2 (the width is pitch-matched to the memory cells), compared to 2.5×8.0 mm^2 for the half-sized (4×4) standard-cell design. Given that the peripheral circuitry area grows with the square of the number of links, an 8×8 standard-cell design would be about *9 times* larger than this same configuration in full-custom.

Figure 10 shows the layout of a part of an incoming link, with 6 dynamic latches and the corresponding drivers (pull-down's) for the (vertical) input multiplexors; the outputs of these multiplexors (not shown in the figure) feed the bit-line drivers that write into the memory arrays. The pattern of one of these six bits, repeated 256 times horizontally and 8 times vertically, forms the main part of the incoming-link peripheral circuits. As we see, all active circuits are laid out *under* the horizontal link wires (in0 through in15). Thus, the area of this block approaches the minimum possible area of a crossbar, since every crossbar has to have at least the data wires. The only deviation from that minimum area are the three control wires and the two (shared) power



□

Performance measurement	typical	slow
input mux: fall time	1.0 ns	2.5 ns
word line rise time	1.0 ns	2.2 ns
memory write: bit line fall time[2]	3.5 ns	8.0 ns
bit-line equalization after write	3.5 ns	4.5 ns
memory read: bit line slew rate	0.5 V/ns	0.1 V/ns
latched memory output: drive time	2.5 ns	4.0 ns
outgoing link: fall time	3.0 ns	7.0 ns

5. Comparisons and Conclusions

In section 1.1 we saw that non-FIFO input buffering is the best switch buffer architecture among those that require relatively low-throughput buffer memories, while shared buffering, which requires high-throughput buffer memories, is the best among all buffer architectures. For switches with many links and with small packet sizes, one can use block-crosspoint buffering instead of a single shared buffer. In this section, we will compare the cost of non-FIFO input buffering and shared buffering, when using full-custom CMOS technology like the one of section 4.3.

Figure 11 presents a simplified but useful comparison of these two architectures, from the point of view of silicon area. We assume a switch with n incoming and n outgoing links; let T be the memory cycle time and the clock period, and assume that the individual link throughput is w bits$/T$, i.e. each link enters and leaves the block as w wires. Under input buffering, each buffer memory needs a write throughput of w bits$/T$, plus a read throughput of w bits$/T$. Consequently, it must either be dual-ported and have a width of w bits, or single-ported with $2w$ width; in either case, its horizontal size is approximately $2w$, in units of single-ported bit cells. Figure 11 assumes that the n input buffers are organized as tall and skinny memory arrays that are placed next to each other, so that their outputs come out of their bottom. Under this arrangement, the total buffer memory width of the switch is $2nw$. The shared buffer has *exactly the same width,* since its throughput must equal the aggregate throughput of all switch links, which is equal to the aggregate throughput of all input buffers. The total input buffer width includes n address decoders, while the shared buffer width includes one address decoder and $2n-1$ pipeline registers; to a first order of approximation, these are about the same. Let H_i be the height of the input buffers, and H_s the height of the shared buffer. If H_i and H_s are measured in units of bit cells, and if the input buffers are single-ported, then the input buffer size is $2wH_i$ bits per input or $2nwH_i$ total, while the shared buffer size is $2nwH_s$ bits. To achieve a similar level of switch

---

[2] the memory cell flips state much faster (1ns after the word line rise)

performance, fewer total buffer bits suffice under shared buffering (section 1.1), so we can let H_s be (significantly) smaller than H_i. This represents a net advantage of the shared buffer.

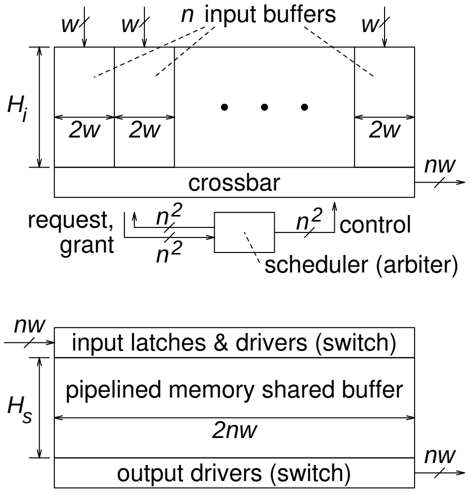


Figure 11: VLSI floor plan comparison: input versus shared buffering

Under (non-FIFO) input buffering, a w-bit-wide $n \times n$ crossbar fabric is needed. The area complexity of this could be $w \cdot n^2$ (w 1-bit-wide switches), in units of 1-bit transmission gates, but this does not take into account the routing of the n^2 control wires that each crossbar needs; taking the latter into account, the complexity would probably grow to $w \cdot n^3$. Practically, however, we believe that it is best to *pitch-match* the crossbar fabric to the input buffers; this will avoid spending considerable area in routing the $2nw$ wires from the buffer memories' outputs to the crossbar fabric. If this approach is followed, then we estimate the size of the crossbar as roughly $2nw \times nw$, where the length $2nw$ is in units of storage cells, and the height nw is in units of wires (n w-bit crossbar outputs); we assume that the n^2 control signals can pass vertically through the empty space that is left by the relatively large storage cell pitch. For the shared buffer case, as we saw in section 4.3, the area occupied by the input and output buffer latches and drivers is dominated by the incoming/outgoing link wires, vertically, and it is limited by the storage cell pitch, horizontally. Thus, the area of each one of these two blocks is again roughly $2nw \times nw$, measured in the same units as above, i.e. approximately the same as for the crossbar under input buffering. In fact, these input and output latch/driver blocks are two $n \times 2n$ w-bit crossbar switches, since they can route any one of the n incoming links to any one of the $2n$ memory stages, and any one of the latter to any one of the n outgoing links. The advantage of these switches, relative to general crossbars, is that they only require $2n$ or n external control wires each; the rest of the $2n^2$ control signals are generated internally, as delayed versions of the external signals, according to the pipelined mode of operation.

As we see, the input buffers and the shared buffer have the same width ($2nw$). The input buffers need more storage bits for similar performance, so $H_i > H_s$. The input buffering scheme needs only one crossbar, of approximate area $2nw \times nw$, while the shared buffer needs two such blocks, one for input and one for output, for a total area of approximately $2nw \times 2nw$. On the other hand, non-FIFO input buffering needs a quite complex scheduler (arbiter), whose cost is difficult to estimate here. Based on [TaCh93], the basic cell of such an 8×8 arbiter might be $180\ \lambda \times 150\ \lambda$ (it needs 4 more request and 4 more grant wires than fig. 23 of [TaCh93]); then, the entire 8×8 arbiter would be $1440\ \lambda \times 1200\ \lambda$, or $720\ \mu \times 600\ \mu$ for a 1µm technology ($\lambda = 0.5\mu$). This has about the same height but is much narrower than the extra "switch" needed with the shared buffer; however, we have not accounted for the 64 request, 64 grant, and 64 control wires that must connect the arbiter to the input buffer controllers and to the crossbar.

Concerning latency (delay from packet header entrance to exit), the pipelined memory shared buffer has the disadvantage that it cannot initiate the transmission of two packets in the same clock cycle. Under heavy traffic load, this is not important, because every packet has to wait for other packets to go out in front of it, anyway. Under light load, the above effect will increase the cut-through latency of all but one of the packets that happen to enter into the switch in the same clock cycle. Probabilistically, the expected value of this delay is approximately 0.25 clock cycles times the link load; for 40% load, this would be one tenth of a clock cycle, i.e. negligible. On the other hand, non-FIFO input buffering always suffers from the extra delay of executing the scheduling (arbitration) algorithm. Under heavy load, shared buffering has considerably less latency than non-FIFO input buffering, as discussed in section 1.1.

Conclusions

In high speed switch chips, input buffering (FIFO or non-FIFO) and shared buffering cost about the same per bit of storage, to a first-order approximation. However, by its nature, shared buffering performs better than input buffering for a given total number of buffer capacity. Thus, contrary to a widespread impression, shared buffering should be the architecture of choice for gigabit VLSI switches. We have described a novel "pipelined memory" organization, which is particularly suitable for VLSI shared buffers, and we have shown a few prototype implementations of it. Relative to interleaved or wide memories, the pipelined memory greatly simplifies control, input and output latches and drivers, and cut-through circuitry. Additionally, the pipelined memory lends itself to very high speed IC technologies, because all clock, control, and data signals always flow in the same, fixed direction, thus allowing the designer to minimize their relative skew by matching their delay times.

Acknowledgements

Telegraphos is being developed within the ESPRIT project 6253 “Supercomuter Highly Parallel System” (*SHIPS*), funded by the European Union, through DG III of its Commission, HPCN Unit. We deeply appreciate this financial support, without which Telegraphos would have not existed.

Telegraphos is a collective effort: many people have contributed to the design of the architecture, the prototypes, and the systems software. The authors wish to acknowledge in particular George Kornaros and Nicolas Houstis for porting the Telegraphos I switch design into the Telegraphos II ASIC technology, Christoforos Kozyrakis and Miltos Grammatikakis for contributing to the switch simulation, and George Kalokerinos, Apostolos Dollas, Manolis Stratakis, Chara Xanthaki, and George Papadourakis for implementing the rest of the Telegraphos I system prototype. Useful comments on an early draft were provided by Miltos Grammatikakis and Dimitrios Serpanos. The help of all of them is greatly appreciated.

References

[AOST93] T. Anderson, S. Owicki, J. Saxe, C. Thacker: “High-Speed Switch Scheduling for Local-Area Networks”, ACM Trans. on Computer Systems, Vol. 11, No. 4, November 1993, pp. 319-352.

[Arno89] E. Arnould, F. Bitz, E. Cooper, H.T. Kung, R. Sansom, P. Steenkiste: “The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers”, Proceedings of ASPLOS-III Conference, ACM SIGARCH Vol. 17, No. 2, April 1989, pp. 205-216.

[AtSe88] W. Athas, C. Seitz: “Multicomputers: Message-Passing Concurrent Computers”, IEEE Computer Magazine, Vol. 21, No. 8, August 1988, pp. 9-24.

[C104] “IMS C104 Packet Routing Switch – Preliminary Datasheet”, INMOS Ltd, 1000 Aztec West, Almondsbury, Bristol, BS12 4SQ. UK, June 1993.

[DaSe87] W. Dally, C. Seitz: “Deadlock-Free Message Routing in Multiprocessor Interconnection Networks”, IEEE Trans. on Computers, Vol. C-36, No. 5, May 1987, pp. 547-553.

[Dally90] W. Dally: “Virtual-Channel Flow Control”, Proceedings of the 17th Int. Symposium on Computer Architecture, May 1990, pp. 60-68.

[DeCS88] M. Devault, J-Y. Cochennec, M. Servel: “The "Prelude" ATD Experiment: Assessments and Future Prospects”, IEEE Journal on Selected Areas in Communications, Vol. 6, No. 9, December 1988, pp. 1528-1537.

[HlKa88] M. Hluchyj, M. Karol: “Queueing in High-Performance Packet Switching”, IEEE Journal on Sel. Areas in Communications, Vol. 6, No. 9, Dec. 1988, pp. 1587-1597.

[HoLa93] M. Homewood, M. McLaren: "Meiko CS-2 Interconnect Elan - Elite Design", Meiko Ltd., 650 Aztec West, Bristol, UK, May 1993.

[KaSC91] M. Katevenis, S. Sidiropoulos, C. Courcoubetis: "Weighted Round-Robin Cell Multiplexing in a General-Purpose ATM Switch Chip", IEEE Journal on Selected Areas in Communications, Vol. 9, No. 8, October 1991, pp. 1265-1279.

[Kate94] M. Katevenis: "Telegraphos: High-Speed Communication Architecture for Parallel and Distributed Computer Systems", Technical Report FORTH-ICS/TR-123, Institute of Computer Science, FORTH, Heraklio, Crete, Greece, May 1994, 38 pages; available on-line from FTP server ftp.ics.forth.gr, anonymous login, directory tech-reports/1994, file 94.TR123.Telegraphos.ps.Z

[Kung92] H. T. Kung: "Gigabit Local Area Networks: A Systems Perspective", IEEE Communications Magazine, Vol. 30, No. 4, April 1992, pp. 79-89.

[LaSe94] R. Lamaire, D. Serpanos: "Two-Dimensional Round-Robin Schedulers for Packet Switches with Multiple Input Queues", to appear in IEEE/ACM Trans. on Networking, late 1994 or early 1995.

[PaBr93] A. Pattavina, G. Bruzzi: "Analysis of Input and Output Queueing for Nonblocking ATM Switches", IEEE/ACM Trans. on Networking, Vol. 1, No. 3, June 1993, pp. 314-328.

[SCI92] ANSI/IEEE Standard 1596-1992 "Scalable Coherent Interface", IEEE, 1992.

[Souz94] R. Souza, P. Krishnakumar, C. Ozveren, R. Simcoe, B. Spinney, R. Thomas, R. Walsh: "GIGAswitch System: A High-Performance Packet-Switching Platform", DEC, Digital Technical Journal, Vol. 6, No. 1, Winter 1994, pp. 9-22.

[Stun94] C. Stunkel, D. Shea, B. Abali, M. Denneau, P. Hochschild, D. Joseph, B. Nathanson, M. Tsao, R. Varker: "Architecture and Implementation of Vulcan", Int. Parallel Processing Symposium, April 1994.

[TaFr88] Y. Tamir, G. Frazier: "High-Performance Multi-Queue Buffers for VLSI Communication Switches", Proceedings of 15th Int. Symposium on Computer Architecture, ACM SIGARCH Vol. 16, No. 2, May 1988, pp. 343-354.

[TaCh93] Y. Tamir, H-C. Chi: "Symmetric Crossbar Arbiters for VLSI Communication Switches", IEEE Trans. on Parallel & Distributed Systems, Vol. 4, No. 1, January 1993, pp. 13-27.

[YeHA87] Y. Yeh, M. Hluchyj, A. Acampora: "The Knockout Switch: A Simple, Modular Architecture for High-Performance Packet Switching", IEEE Journal on Selected Areas in Communications, Vol. SAC-5, No. 8, October 1987, pp. 1274-1283.