IEEE Computer, 24(1), pages 72--80, January, 1991.

Hector — A Hierarchically Structured Shared Memory Multiprocessor

Z.G. Vranesic M. Stumm D.M. Lewis R. White
Department of Electrical Engineering, University of Toronto

Introduction

This paper describes the architecture of a multiprocessor, called Hector, which exploits current microprocessor technology to produce a machine with good cost/performance tradeoff. A key design feature of Hector is an interconnection backplane that scales well with technology. This is achieved with simple hardware that has short critical paths in logic circuits and short lines in the interconnection network. The result is a system characterized by good performance, reliability and flexibility, which can be realized at a relatively low cost.

An important aim of the Hector project is to develop an architecture suitable for construction of a general-purpose multiprocessor, where the cost is directly proportional to the size so that a low-cost entry-level machine is economically feasible, yet is scalable to larger sizes. In order to easily accommodate configurations with varying numbers of processors, Hector has a hierarchical structure. It features small bus sections interconnected by bit-parallel rings. The buses and rings can transfer data independently of each other so that the aggregate bandwidth increases proportionally to the number of these units.

Hector is suitable for a wide range of applications. It can be effective for single jobs characterized by many parallel tasks, as well as for concurrent execution of multiple jobs consisting of predominantly serial tasks. That is, it is a machine that can be used effectively to run jobs typical of a Unix environment, as well as highly parallel commercial and scientific applications such as transaction systems, finite element analysis, and computer aided design.

Where does Hector fit in the multiprocessor picture

A variety of multiprocessor machines are either commercially available or being designed at the present time. Our goal in the Hector project is to explore a design region where most cost-effective solutions are likely to lie. Figure 1 illustrates where the Hector architecture

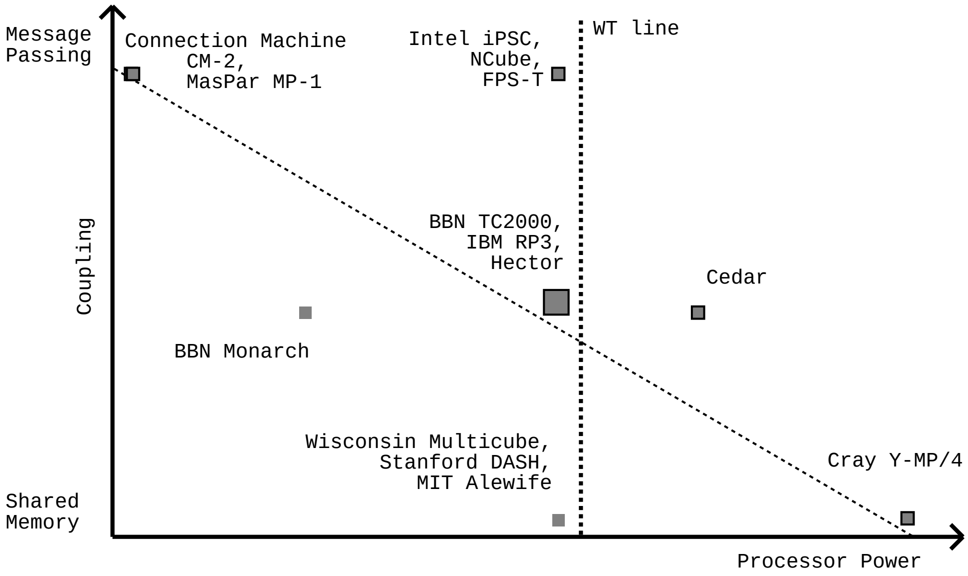

Figure 1: Coupling and Processor Power in Multiprocessors

fits with respect to existing machines and well-known research projects. The figure indicates the relative characteristics of different machines configured to have approximately the same computational power, roughly in the 0.5 to 1 GFLOPS range.

The two axes in the figure represent two important design options available. The horizontal axis represents the power of the individual processors used, with very simple processors at the origin. The less powerful the processors, the larger the number needed to achieve the desired aggregate power of the system. The vertical axis represents the degree of coupling between the processing modules. In loosely coupled systems, processors can directly access only their local memories, but use data passing to communicate with other processors over the interconnection network. In tightly coupled systems, processors communicate through shared memory at speeds normally associated with local (cached) memory access. Between these two possibilities are distributed memory systems that allow processors to directly access shared memory, but with reduced access times to some memory locations.

Loosely coupled systems are easier to build and less expensive, but shared memory systems are considered to be easier to program by many [5]. Given two systems with processors of equivalent power, the more tightly coupled one is considered to be more general. It can typically execute more efficiently all of the applications that its more loosely coupled counterpart can. It can also execute some applications where sharing occurs at a finer granularity that is not suitable for execution on the more loosely coupled systems. Hence, any system represented by a point in the figure is more general than all systems represented by points that are vertically directly above it.

Two extremes in the figure are immediately apparent. On the left side are machines consisting of a large number of simple processors, such as the Connection Machine with its many 1-bit processors. Their interconnection requirements essentially dictate the use of a data passing organization. On the right side are machines that comprise only a few exceptionally powerful processors, such as the Cray YMP-4. These machines can make full use of the shared memory concept. It is clear that the computational power is not simply a function of the architecture, as evident from the fact that an entry using the Connection Machine (running an application executing at a rate of 6 GFLOPS) won the Gordon Bell prize for supercomputing for 1989, while an entry using the Cray YMP-8 (running at 1 GFLOPS) won it for 1988 [2].

A number of multiprocessors have been designed that lie between the above extremes, which rely heavily on advanced microprocessors. These machines tend to fall into three categories, according to the coupling scheme used. A data passing organization has been the natural choice in multiprocessors based on hypercube interconnection backplanes. Well known examples are the Intel IPSC, the Floating Point Systems T-series, and the NCUBE machines.

The shared memory model with full cache consistency is the goal of recently proposed MIT Alewife [1], Wisconsin Multicube [4], and Stanford Dash [7]. Several machines have been built which have the shared memory capability but without hardware-supported cache consistency. The BBN TC2000 , IBM RP3 [9], and our Hector multiprocessor are of this type. In these systems read-only code and data, as well as local data can be cached, but not all shared modifiable data. The Cm* multiprocessor would probably also belong to this class if it were implemented with today's technology. We should also mention the BBN Monarch design [10], which is intended to incorporate a large number of somewhat simpler processors with no caches in a shared memory organization.

In Figure 1 we have drawn a line joining the Connection Machine and Cray extremes. It can be argued that machines that lie above this line are economically feasible with today's technology. Indeed, all of the machines shown in that region of the figure have been built, although not necessarily in full size. On the other hand, machines that lie below the line are more difficult to build, as supported by the fact that none of those mentioned in the figure have been implemented. These systems are challenging to build because it is difficult to provide coherent cached shared data with good performance, without introducing excessive interprocessor traffic.

From a cost point of view, it is apparent that at both ends of the diagonal lie expensive solutions. The most cost-effective solutions may lie along a line in the region where workstation technology is used, indicated by the vertical line WT. The point at the top end of the WT line probably represents the most inexpensive solution, using processors with the best price-performance ratio and a simple interconnection structure. The workstation technology line is rapidly moving to the right; 100 MIPS microprocessors may soon be available. This will make it a challenge to build systems at the point where the WT line

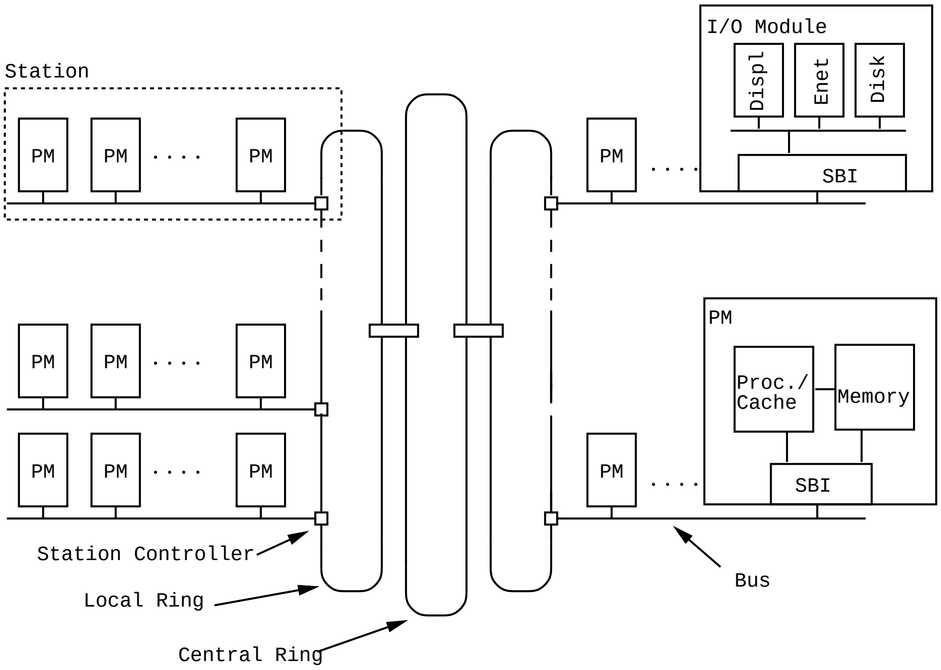

Figure 2: Structure of Hector

and the diagonal intersect, since it appears to be more difficult to scale the speed of the interconnection network than the speed of the processor chips. Hector addresses this challenge by incorporating a novel hierarchical, pipelined backplane that will scale to future technologies.

Organization of Hector

Bus is the simplest structure for interconnecting several processing modules. However, the bus does not scale well in size and saturates quickly if too many modules are connected. With high speed processors, one typically finds that 2-6 processors can be connected to a bus without significant performance degradation. Connecting a larger number of processors to a single bus is feasible only if the request rate for bus transfers by the processors is low.

Hector exploits the natural advantages of the bus by using relatively small bus sections tied together through hierarchical bit-parallel rings. Figure 2 shows the overall organization. A bus section, with the associated processing modules, is called a *station*. Several stations are interconnected by a bit-parallel *local ring*. Local rings are, in turn, interconnected by a *global ring*.

This hierarchical structure is similar to those of Cm* and Cedar. In Cm* [3], multiple processor-memory pairs are connected by a "map-bus" to form a cluster, and multiple clusters are interconnected via "inter-cluster" buses. Rings at higher levels of the hierarchy, as used in Hector, have the advantage that they can be constructed using only short point-to-point transmissions links, allowing for faster signalling. Moreover, the overall bandwidth of the ring increases proportionally to the number of ring segments (although it also increases latency by a corresponding amount). A second difference between Hector and cm* is the manner in which communication is controlled. In cm*, a relatively powerful, horizontally microcoded processor controls communication, whereas in Hector, the control uses simple discrete logic (implemented in PALs in our prototype). This simplicity allows for faster switching and higher throughput.

In Cedar [6] the lowest level consists of several Alliant FX/8 systems, each comprising 8 PE's (with vector units), connected by a crossbar switch to a cache and higher-level communication structures. These, in turn, are connected by another crossbar switch to memory. We believe that the relative simplicity of rings in comparison with crossbars allows Hector to be more easily scaled to larger sizes and faster technology.

Basic Hector Architecture

A station consists of a number of *processing modules* (PM), connected by means of a bus. A typical PM comprises a microprocessor, cache memory, on-board memory, and various I/O interfaces. While we use the term PM to refer to a module (board) plugged into a slot in the backplane, it need not have any processing capability; some may be just memory modules, and other, specialized PM's provide interfaces to various input/output devices.

Hector provides a flat, global (physical) address space, where each PM is assigned a unique range of addresses. All processors can transparently access all memory locations. Information transfer takes place in a packetized format using a synchronized packet transfer scheme. The traffic is controlled by three types of interface circuits. A *station bus interface* (SBI) in each PM handles the communication requirements of the PM. A *station controller* controls on-station transfers as well as the local ring traffic at the station. It contains a set of latches that hold an entire packet and a set of transceivers that isolate the station bus from the local ring. A packet traverses the ring by being transferred from the latches of one station into the latches of the next station. An *inter-ring interface* connects a local ring to the global ring. Within one clock cycle, a packet can be transferred:

- between two PM's within a station,
- between the latches of two adjacent station controllers or inter-ring interfaces,
- from a PM in a station into the latches of the next station controller on the ring, or
- from the latches of a station controller to a PM in the station it controls.

An on-station transfer and a transfer from the latches of that station controller to the latches of the next station can occur simultaneously. The low complexity of each of these operations make the backplane scalable.

By controlling both the station bus and the corresponding ring segment, the station controller can give priority to packets being transferred on the ring; it does not allow a local PM to access the bus when there is a packet in the ring latches addressed to this station (allowing the packet to be latched onto the station bus), and it allows only on-station transfers whenever a valid packet in its latches is to be transferred to the next station. This strategy prevents packets from having to be dropped or queued at the station and local ring levels. The inter-ring interface, which is essentially a 2×2 crossbar switch, requires FIFO buffers in order to store packets when *collisions* occur, where packets coming from both the local and global rings in a given clock cycle have to be routed to the same output.

The addressing scheme used is simple to allow packets to be routed with minimal overhead in a fraction of a clock cycle. Each ring is identified by the most-significant r bits of the (memory) address, the station is identified by the next s bits, and the slot in the station is identified by the least-significant p bits. This allows simple and fast address decoding.

Memory Access and Communication Protocol

Accesses to remote memory modules are transparent to the processors. A non-local memory request is recognized by the SBI control circuitry, which forms a *request* packet that includes a 32-bit destination memory address and a source PM address. In case of a write, the packet also contains 32 bits of data. When the packet arrives at the destination, the destination SBI performs the action requested (read or write) and returns a *response* packet to the source. In case of a read, the response packet contains up to 64 bits of data. In case of a write, the response packet is just the acknowledgement that the write operation has been completed. The response packet is sent to the PM identified by the source address in the request packet.

If a response packet is not received within a timeout period, the source PM retries by retransmitting the request. If no response is received after multiple retries, the operation fails. The requesting PM also retransmits the request if it receives a negative acknowledgment response, which can happen, for example, when a remote request was recognized but could not be serviced. In order to allow a PM to have multiple outstanding requests, a tag is included in each request packet, which is returned with the response packet so that the response can be matched to the correct request. (Our prototype implementation allows only one outstanding request, however.)

Considerable flexibility is provided in the memory access requests. In addition to being able to accommodate requests involving 8, 16, 32 or 64 bit data, a *burst read* of 128

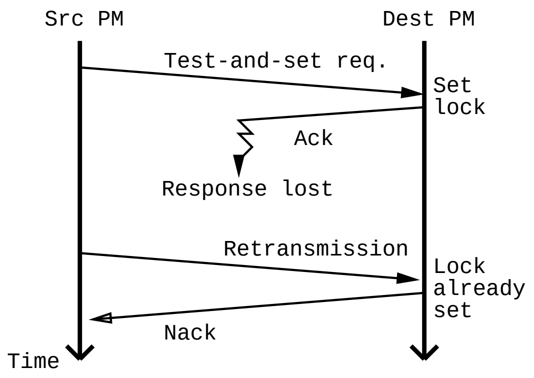

Figure 3: The problem with nonidempotent requests

bits is also available, which is typically used for loading cache lines. In the case of burst reads, multiple response packets are generated automatically by the responder PM, each containing 64 bits of data. The entire operation is retried if any of the response packets does not arrive within the timeout period.

Atomic Operations

Since the transfer of packets is reliable in the common case, we believe that aborting packet transfers without specifically signaling an error condition in some cases will lead to a reduction in the complexity of hardware and cycle time, and hence an increase in performance for the common case. For example, a packet may be dropped when a transmission error is detected (by parity bits) or when a buffer overflows. Requiring the hardware to generate negative acknowledgement packets in these cases would add to the complexity of the hardware. The communication protocol in Hector is capable of detecting aborted transfers by timing out at the source PM, resulting in a retransmission of the request packet.

For read and write operations, the request-response protocol with timeout works correctly because of the idempotent nature of the operations.[1] However, this is not true for all operations. In particular, the read-modify-write operation, needed to implement a processor's `test-and-set` or `swap` instructions, cannot be implemented directly using the same request-response protocol. A difficulty arises if a response to a read-modify-write request is not received within a timeout period, because the source SBI does not know whether its request packet was dropped or whether the corresponding response packet was dropped (in which case all further retries may fail). Consider the example shown in Figure 3, where the response to a test-and-set request is dropped. When the retransmission of the test-and-set

[1] Strictly speaking, the write operation is not idempotent: a retransmission of a write request may result in the write operation occurring twice. This is acceptable behavior for the vast majority of applications; for the others, the write must be made part of a critical region.

request is received at the destination PM, a negative acknowledgement is returned, because the lock is already set. The source PM cannot determine whether the lock was set by its initial request or whether it was set by another processor in the meantime.

To handle this situation, the read-modify-write operation is performed in two separate stages. The first stage entails the sending of a *read-and-lock* request, which causes the responder PM to read the addressed memory location and return its contents in a response packet. When the requester receives the data requested in its read-and-lock packet, it commences the second part of the read-modify-write by sending the data to be written to the destination memory in a *write-and-unlock* packet. To guarantee atomicity, the responding PM must prevent other processors from `read-and-lock`ing the same memory location between these two operations. Each SBI maintains a set of `<proc,addr>` pairs for this purpose, such that an entry is recorded during the read-and-lock operation and cleared during the write-and-unlock operation. These `<proc,addr>` pairs are also used to detect and appropriately handle duplicate requests due to retransmissions.

This protocol allows atomic operations on any memory location and can survive losses of packets, regardless of which packet is lost. If either the read-and-lock or its response is lost, the requester will time out and retransmit the request. If the destination PM received the original request, it will recognize the retransmission (since it stored the identifier of the requester) and respond with an acknowledgement. If either the write-and-unlock or its response is lost, the requester will also time out and retransmit the request. This will result in a write and unlock action if the original write-and-unlock packet was lost and an acknowledgement otherwise.

Comparing Hector's Backplane with other Interconnection Structures

As a hierarchical system, Hector provides for fast local operations over a bus. Most systems with non-uniform memory access times only support memory accesses at two time-cost levels, namely local on-board accesses and remote accesses. For example, in a 16 processor BBN TC-2000, the access time differential between local memory and remote memory is 1:4. In contrast, Hector has multiple levels in its hierarchy, so the cost of accessing data increases incrementally with the distance. In our prototype implementation, the access time differentials between local, on-station, on-ring and off-ring memory are 1:1.2:2:4. The advantage of a hierarchical structure is that much of the communication remains within the lower levels of the hierarchy because of the locality in data accesses, resulting in relatively low average memory access times. Moreover, since a bus is used at the lowest level of the hierarchy, it is possible to provide for cache consistency among the PMs attached to the bus using a simple snoopy protocol.

Other important and distinctive advantages of the Hector backplane are its simplicity, low cost and scalability to higher clock rates. Although it is difficult to make a direct comparison to other systems, an attempt to compare metrics that impact the cost and

	Hector	Banyan	Crossbar	Hypercube
Longest wire length	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Gates in switch path	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Total switch cost	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n \log n)$
Switch data bits	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$
Switch control bits	$O(1)$	$O(1)$	$O(n \log n)$	$O(\log n)$
Switch complexity	$O(1)$	$O(1)$	$O(n^2)$	$O(\log n)$
Fanout	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$

Table 1: The Cost and Complexity of Interconnection Structures

speed of several interconnection structures is shown in Table 1, where $n \leq 256$ is the number of processors. Besides Hector, we consider Banyan networks, crossbars, and hypercubes.

The ability to scale an interconnection network to higher speeds depends largely on the longest wire length needed to connect together parts of the system, because signal quality degrades over long wires at higher clock rates, and because skew between signals in a cable makes it difficult to reduce the clock cycle time. The first line in Table 1 gives the length of the longest wire required by each type of network.[2] Hector features a constant length wire, since each ring only connects to its immediate neighbor. In our prototype, with two levels of rings, the longest wire length is only seven inches. All other systems require wire length that is proportional to the size of the system, in that they require connections to processors at least half way across the entire set of processors, making it difficult to scale to high clock speeds.

The number of gates in the path can also be a significant source of delay. All of the systems have identical orders of delay, although they still may differ by a constant factor that can be quite large (i.e., 50 ns, the time it takes to transfer a packet in Hector vs. 7 ns, the delay through a multiplexer of a crossbar). The third line in the table measures the total cost of the switch for an entire system. Hector maintains $O(n)$ cost, so the cost of the switch is always a small constant fraction of the cost of the entire system. At the other extreme, the cost of a crossbar is $O(n^2)$, which can rapidly dominate the total system cost as n becomes larger.

The remaining lines of Table 1 attempt to measure the complexity of implementing the interconnection network using ASICs. The fourth line measures the amount of data processed by a switch, and the fifth line measures the number of bits required to control the data routing at a switch. These metrics are important because the number of pins that

[2]We assume that these systems can be implemented to fit in a single rack, since a system within a single rack has maximum wire length that is a linear function of processor distance, while in a larger system the wire length might be the square root of the distance.

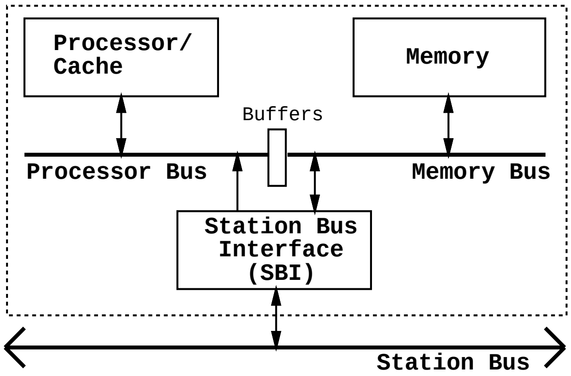

Figure 4: Block Diagram of the Processor Board

can be connected to a chip is limited. The sixth line gives the complexity of the logic in a single switching unit. All networks, with the possible exception of large crossbars, are simple enough that they can be implemented with ASICs. The last line gives the fanout per wire.

Table 1 supports the claim that Hector offers a scalable architecture through the use of short electrical connections, and does so at a lower cost than other interconnection networks.

Implementation

This section describes the implementation of our prototype. Figure 4 shows a block diagram of the PM board. The processor is a Motorola 88100 microprocessor running at 20 MHz. The cache consists of up to four Motorola 88200 16-Kbyte cache chips. The on-board memory comprises up to 16 Mbytes of RAM. It is implemented as part of the processor boards rather than as separate modules, because this reduces bus loading and the average number of bus accesses. The on-board bus is divided into two segments called the processor bus and the memory bus. These two buses are separated by buffers in order to isolate the processor from the memory bus, allowing accesses to this memory by other PM's while the processor is accessing off-board memory.

There are three main memory activities on a processor board that need to be considered: processor on-board requests, processor off-board requests, and requests from the station bus to the memory. The processor accesses on-board memory by first obtaining control of the memory bus and then accessing the memory. Off-board references from the processor use the SBI circuitry, as explained earlier. Memory requests arriving from the station bus are entered in a 2-deep FIFO prior to requesting control of the memory bus. Once control is granted, the SBI performs the memory operation and returns the acknowledgement together with any data. The SBI signals a negative acknowledgement on a distinct bus line if the FIFO is full when a packet arrives.

The Station Bus and Local Ring Interface

Station bus operations are pipelined in the following way. A bus request is sent to the station controller during one cycle. If there is no contention, then the station controller returns the bus grant at the beginning of the next cycle and the packet is placed on the bus during the same cycle the grant is received. In the case of an on-station transfer, the reception of the packet is acknowledged in the next cycle by asserting the Received line of the bus. A separate acknowledgment packet is therefore not necessary for on-station write requests. If the Received line is not asserted, then the source PM will immediately attempt to retransmit the request. The entire process, from the time the source module asserts its Request line to the time it recognizes its Received line, takes three cycles, but it ties up the bus for only one. Independent transfers from different source modules can therefore be placed on the bus in each cycle, allowing the bandwidth of the bus to be fully utilized.

The station controller is responsible for arbitration of the station bus and transfer of data between the station bus and the local ring. It gives the highest priority to packets on the ring addressed to this station. Lower priority is given to requests for off-station transfers by PM's on the station. An off-station transfer is accommodated whenever the ring segment has an empty slot. The lowest priority is given to on-station transfers. Within this priority strategy, the bus is granted in a round-robin fashion when multiple off-station or multiple on-station bus requests are outstanding.

The Inter-ring Interface

The inter-ring controller has a two-deep FIFO buffer. In general, priority is given to packets coming from the global ring; hence, packets that successfully reach the global ring will reach their destination (if no transmission errors occur). Extensive simulations indicate that this is the strategy with the best performance for virtually all data access patterns. Simulation results also indicate that the global ring connecting n rings with interfaces containing two-deep buffers have fewer collisions than an $n \times n$ crossbar with no buffers for the type of data access patterns we expect to be typical.

Instrumentation Support

We have implemented an instrumentation facility that allows nonintrusive monitoring of the hardware. It can operate in a number of modes. For example, the Address Histogram mode allows generation of histograms based on address information, conventional memory use profiles, or inter-reference jump distance distributions. The State Histogram mode

Clock $\rightarrow$ bus grant	7
Bus grant $\rightarrow$ drive bus	18
Drive bus $\rightarrow$ decode address	7
Decode address $\rightarrow$ controller output	10
Setup time	2
Skew	2
Total	46 ns

Table 2: Current 20-MHz Bus Timing

allows analysis of bus cycle and machine state distributions. Two tracing modes are also supported; the Full Trace mode traces address information and the Timestamp mode is used in cases where references to a small number of objects may be distributed over long periods of time. This experimental facility allows us to monitor and measure low level activity on real workloads.

Scaling for Speed

Hector's interconnection network has been specifically designed to be scalable to higher speeds. The cycle time is 46 ns in our 20-MHz prototype, as detailed in Table 2. The relatively large time to drive the bus is due to the large capacitive load offered by the multiple bus drivers and receivers on each station.

Since processor clock speeds will continue to increase, we briefly consider two ways to reduce the clock cycle of the Hector backplane. First, additional pipeline stages in the bus controller can be added. This will lead to better throughput but increased latency. For example, it is possible to split the bus operations of Table 2 into two pipeline stages. In the first stage a packet is enabled onto the bus from one bus driver and clocked into all receivers without being decoded, reducing the electrical loads on the bus from the current three to one per PM. The time needed to accomplish this essentially dictates the cycle time. In the second stage, decoding takes place, where the controller determines in which buffer the packet should be placed. If the packet is addressed to a given module, a multiplexer is selected to pass the packet into an appropriate set of latches.

The second way to reduce cycle time is to use ASIC technology. Table 3 (top half) shows our timing estimates for the pipelined implementations using CMOS standard cells with conventional MSI bus drivers and receivers (due to the low drive capability of CMOS ASICs). The reduced time to drive the bus is due to decreased loading on the bus. Table 3 (bottom half) also shows the same for an implementation using an ECL gate array that includes the bus drivers and receivers on chip.

An implementation using ASICs would eliminate many of the constraints that limit

Technology	Stage 1		Stage 2	
CMOS	Clk → grant	7	Clk → clocked bus data	11
	Grant → drive bus	14	Clocked bus data → ASIC bus	2
	Setup time	3	ASIC bus → decoded address	4
	Skew	2	Decoded address → controller output	5
			Controller output → mux output	3
			Setup time	1
	Total	26 ns	Total	26 ns
ECL	Stage 1		Stage 2	
	Clk → grant	1	Clock → clocked bus data	1
	Grant → drive bus	5	Clocked bus data → decoded address	2
	Bus → ASIC internal bus	1	Decoded address → controller output	2
	Setup time	1	Controller output → mux output	1
	Skew	2	Setup time	1
			Skew	1
	Total	10 ns	Total	8 ns

Table 3: Timing estimates for CMOS and ECL ASIC Technology

performance in our prototype. In particular, the MSI technology used makes it difficult to implement buffering or pipelining for data paths due to the large number of chips required. ASIC technology is capable of implementing many such buffers on a small part of a chip. This would also allow us to significantly reduce memory access times, by interleaving the memory system, pipelining the memory error correction circuitry to deliver corrected data at full memory bus speed, and increasing the size of the data path to 64 or 128 bits. In addition, an 8 packet deep FIFO in the SBI would dramatically reduce the number of retries required. Ultimately, VLSI technology will advance to the point where it may be feasible to implement an entire Hector station on a single chip.

Concluding Remarks

Hector has three important advantages. First, the hierarchical structure allows short transmission lines and construction of a simple and fast backplane. This makes Hector scalable to match the needs of future high speed microprocessors, and leads to increased performance, reliability and flexibility, as well as lower cost. Second, the cost and the overall bandwidth of the structure grow linearly with the number of processing modules. This makes Hector expandable to large sizes, yet allows small configurations at a low cost. Finally, the cost of a memory access grows incrementally with the distance between the processor and memory location. This allows the low cost of localized memory accesses to be exploited by single threaded applications, applications with a small degree of parallelism, and applications with a high degree of locality in their memory accesses.

A possible shortcoming of our current design is the lack of cache consistency across all processing modules. While hardware-based cache consistency mechanisms for larger systems is an active area of research [1, 4, 7], it appears that their complexity and cost will be excessive if consistency is maintained at the granularity of relatively small cache lines. It is not easy to provide consistency at this granularity because of the excessive inter-module traffic and bottlenecks caused by synchronization and serialization requirements. We expect this to be even more difficult as faster processors with multi-level caches are introduced in the future. For this reason, software techniques that keep caches consistent at a coarser granularity [8] may be a viable alternative, although it is not yet clear how effective they can be for different parallel applications.

Non-uniform memory access times impose a challenge in the design of operating systems and parallel applications. In order to exploit the performance potential of a system like Hector, it is necessary to manage memory, I/O and processors in a way that minimizes the average memory access costs by reducing the number of remote memory accesses. The memory management subsystem can replicate or migrate pages to bring them closer to the processes accessing them, but must prevent excessive paging overhead. The scheduler can place processes close to the data they are accessing, while attempting to balance the load on the processors at the same time. Finally, although Hector's raw I/O capacity is large (since each PM has I/O capabilities and since I/O devices can be attached directly to the station bus), this capacity cannot be exploited if all I/O traffic must traverse the system; it is important to localize I/O accesses.

We are presently expanding our Hector prototype which now consists of several stations connected by a local ring. Concurrent with our development of the hardware, two other research groups at University of Toronto have developed operating system support for it, which allows us to run application software. Hector provides an excellent experimental testbed for studying the software issues raised above.

Acknowledgements: The Hector project has been funded by the Natural Sciences and Engineering Research Council of Canada strategic grant $STR0032675$. We wish to thank Kennedy Attong, Yonatan Hanna, Jan Medved, Peter Pereira, and Ron Unrau for their substantial work in helping us implement the prototype. We also wish to acknowledge the help of Ben Bacque, Keith Farkas, Carl Hamacher, Ric Holt, Roman Kiss, Orran Krieger, Chuck Pilkington, Ken Sevcik, Martin Snelgrove, and Safwat Zaky who have participated in the Hector project.

References

[1] A. Agrawal, B.H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *17th Intl. Symp. on Comp. Arch.*, 1990.

[2] J. Dongorra, A.H. Karp, K. Kennedy, and D. Kuck. Special Report: 1989 Gorden Bell Prize. *IEEE Software*, 5(3):100–104, May 1990.

[3] E.F. Gehringer, D.P. Siewiorek, and Z. Segall. *Parallel Processing: The Cm* Experience*. Digital Press, 1987.

[4] J.R. Goodman and P.J. Woest. The Wisconsin Multicube: a new Large-Scale Cache Coherent Multiprocessor. In *14th Intl. Symp. on Comp. Arch.*, 1987.

[5] A.H. Karp. Programming for Parallelism. *IEEE Computer*, 20(5):43–57, May 1987.

[6] D.J. Kuck, E.S. Davidson, D.H. Lawrie, and A.H. Sameh. Parallel suppercomputing today and the Cedar approach. *Science*, pages 967–974, 1986.

[7] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *17th Intl. Symp. on Comp. Arch.*, 1990.

[8] S. Owicki and A. Agrawal. Evaluating the performance of software cache coherence. In *3rd Intl. Conf. on APLOS*, pages 230–242, 1989.

[9] G.F. Pister, W.C. Brantley, and George D.A. The IBM Parallel Research Processor Prototype. In *Proc. ICPP*, pages 764–769, 1985.

[10] R.D. Rettberg, W.R. Crowther, P.P. Carvey, and R.S. Tomlison. The Monarch Parallel Processor Hardware Design. *IEEE Computer*, 23(4):18–30, April 1990.