Experiences with the Hector Multiprocessor

Michael Stumm, Zvonko Vranesic, Ron White, Ronald Unrau and Keith Farkas

Technical Report CSRI-276
October, 1992

Computer Systems Research Institute
University of Toronto
Toronto, Canada M5S 1A4

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

Experiences with the Hector Multiprocessor

Michael Stumm, Zvonko Vranesic, Ron White, Ronald Unrau and Keith Farkas

Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada M5S 1A4
Email: stumm@csri.toronto.edu

Abstract

Hector is a shared-memory multiprocessor based on a hierarchy of unidirectional slotted rings. The architecture was designed to be both size scalable and generation scalable, yet remain simple. The Hector system demonstrates that a scalable shared-memory multiprocessor need not be more complex than a distributed-memory multicomputer, yet it can provide for lower latency communication.

This paper describes our experiences with the Hector project. It examines critically both the Hector architecture and the decisions made while building a prototype machine. The key implementation choices are presented and scrutinized with respect to the main objectives of simplicity and scalability. Software implications, based on the development of an operating system and application code, are discussed and evaluated. Performance results obtained in experiments run on the prototype are given.

1 Introduction

Hector is a shared-memory multiprocessor architecture based on a hierarchy of unidirectional slotted rings. Unidirectional rings are used because *i*) with their point-to-point connections, they can be run at high clock speeds; *ii*) it is easy to make full use of their bandwidth; *iii*) they provide a natural broadcast mechanism; and *iv*) they allow easy addition of extra nodes. The hierarchical structure increases the overall bandwidth of the system by supporting concurrent transfers within various subsystems. Hierarchy also reduces the latency of localized transfers. Moreover, it provides a unique path between any two nodes, which can be useful in the implementation of some cache consistency protocols.

The Hector system demonstrates that a shared-memory multiprocessor need not be more complex than a distributed-memory multicomputer, yet it can provide for lower latency communication and still be size and generation scalable. Hector is size scalable because the throughput scales with the size of the system, and also because the cost of the system is directly proportional to the size of the system. Very small configurations with only a few processors do not require an elaborate and expensive backplane in order to be upgraded later to a larger size; large systems can be constructed by simply connecting together smaller systems without the need for complex and expensive connection components. The architecture is generation scalable in that it can support the fast clock rates of future processor generations. This is a direct result of having only point-to-point connections that allow fast communication and having simplicity as the primary principle in our design.

At the University of Toronto, we have implemented a prototype with 32 processors to test the key concepts of the architecture. We have implemented an operating system for it, and run a variety of applications on it. The system is being used daily by students, scientists, and occasionally by local industry.

In this paper, we describe the experiences we have had with the prototype and reevaluate the design decisions made. In Section 2 we briefly review the design of Hector; for a more complete description the reader is referred to an earlier paper [19]. Section 3 describes our implementation and focuses on a number of optimizations that simplify the implementation and reduce the latency of basic operations. In Section 4 we evaluate our design critically and also discuss issues that we consider to remain open. In evaluating our design, we compare Hector to other systems with similar objectives. Section 5 discusses the implications that the Hector architecture has on software.

2 The Hector Architecture

Hector consists of a set of *stations* connected by a hierarchy of rings. Each station comprises a cluster of one or more processing modules (PMs). In our prototype, the PMs are connected to a bus, as depicted in Figure 1. A different implementation could have a station implemented on a single printed circuit board with several PMs directly connected to a shared second-level cache. Each PM may contain a processor, memory (I/O devices behave as memory), or both. Stations are connected to *local* rings, which in turn are interconnected by higher-level

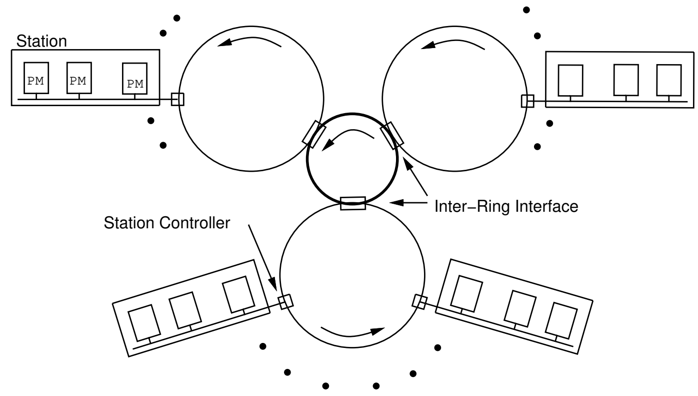

Figure 1: Structure of Hector with two levels in the ring hierarchy.

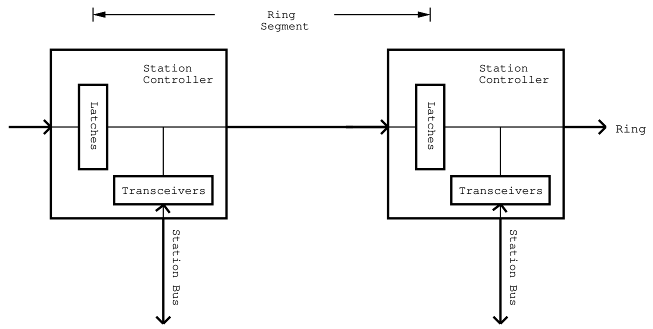

Figure 2: Two station controllers joined by a ring segment.

rings. Hector provides a flat, global (physical) address space, and each station is assigned a unique contiguous portion of that address space, determined by its location. All processors can transparently access all memory locations in the system.

Information transfer in Hector takes place by means of fixed size *packets* in a bit-parallel format. *Station controllers* arbitrate on-station traffic as well as local-ring traffic at the station, while *inter-ring interfaces* control traffic between two rings. Both types of interfaces can be realized with simple circuits. The station controller contains a set of latches that can hold an entire packet. A packet traverses the ring by being transferred from the latches of one station controller into the latches of the next station controller. As shown in Figure 2, the transceivers gate incoming packets from the station latches to the PMs on this station, and they gate outgoing packets to the latches of the next station.[1] The inter-ring interfaces require FIFO buffers in order to store packets if collisions occur, which can happen when, in a given cycle, input packets from both rings are to be routed to the same output.

A request for access to a non-local memory location (i.e. memory address that is not on the same PM) initiates a packet transfer across the backplane, which is done using a request-response protocol. A *request* packet is sent from the requesting PM to the target PM, followed by a *response* packet which is sent by the target PM back to the requesting PM. In the case of a read access, the request packet contains the target memory address and the address of the requesting PM. The response packet contains the read data. Burst reads are supported to decrease cache line fill times. In this case, multiple data packets are returned in response to a single request. Individual response packets are transmitted as soon as their data is available, instead of forming one larger packet. This increases concurrency, because memory is being accessed while some of the data is already being sent. Moreover, the hardware is simplified by only having to handle one packet size. For a write access, the request packet contains the write data in addition to the addressing information, and the response consists of an acknowledgment packet.

It is possible that a request packet cannot be successfully delivered to the target PM, as can happen when the target PM is congested and thus incapable of accepting a further request. In this case, the target station controller generates a negative acknowledgement (NACK) packet, which is sent back to the requesting PM so that it can retry the operation by retransmitting the request packet. The Hector protocol was designed to allow packets to be lost (e.g., due to congestion at some node or by being corrupted in transit). When a packet is lost, the requesting PM will retry the request after a timeout. The read and write operations can be retried when either the request or the response packet is dropped because the operations are *idempotent*; that is, the same effect is achieved whether the operations are executed once or a number of times in succession.[2] In order to make read-modify-write operations such as `test-and-set` or `swap` idempotent, we partition these operations into two subtransactions, *read-and-lock* and *write-and-unlock*, as described in detail in [19].

The entire backplane operates in a synchronous fashion. During a given clock cycle, a packet can be transferred between two adjacent ring segments, between the latches of a station controller and the associated station, or between a station and the next station controller. The addressing scheme uses bit masking and is thus simple; it allows packets to be routed within a fraction of a cycle. The top r bits of the (memory) address specify the ring, the next s bits identify the station, and the PM within the station is identified by the next p bits. Hence, the station controller can correctly route a packet by comparing the top $r + s$ bits with its own address. Similarly, the inter-ring interface can route packets by only comparing the top r bits. Different values for r, s,

[1] Station interfaces become slightly more complex when the rings are clocked faster than the stations. In this case, an additional output buffer is needed between the station and the transceiver, and FIFO buffers are needed to store packets incoming to the station, because over the short term they may arrive faster than they can be consumed.

[2] For write operations this is only true if used for non-shared or synchronized accesses; we discuss this in Section 4.

and p define different configurations of the machine.

Cache consistency in Hector is provided through a basic broadcast mechanism and a software controlled filter mechanism that limits the scope of the broadcasts to make the scheme scalable. The cache consistency scheme is decentralized and hierarchical and thus requires only small additions to the hardware and the transfer protocol described above. It exploits the natural broadcast and ordering properties of unidirectional rings. The broadcast and filter mechanisms can be used to implement both write-through invalidate as well as update protocols, and can support a variety of consistency models. To simplify the discussion, however, we only sketch how processor consistency [9] is achieved using a write-through invalidate protocol. A more complete description and a discussion of alternatives can be found in [6, 7].

A packet is broadcast to all PMs by first propagating it to the highest-level ring, where it traverses the entire ring exactly once and is then removed. At each inter-ring interface, a copy of the packet is made and passed down one level of the ring where it again circulates around the entire ring and is further propagated to all lower-level rings. At the lowest level, the packet is processed at each station.

Using this broadcast mechanism, the protocol for write operations to shared data is as follows. Write operations are initiated by a PM write through the cache, sending a write request to the destination memory. When the destination can service the request, it generates an acknowledgement which is then broadcast to all PMs, including the PM that initiated the write request. The acknowledgement contains the information needed by the PMs to invalidate their cache lines.

A pure broadcast scheme, where every write is transmitted to every station, is obviously not scalable, since broadcast traffic would increase linearly with the number of PMs. A simple filtering mechanism prevents invalidation packets from propagating to those sections of the system where no PM has a copy of the data. Due to locality of data accesses and sharing locality typical of many applications, an effective filter reduces invalidation traffic significantly. The filter mechanism is implemented as follows. Each inter-ring and station controller interface has two filters, one to restrict broadcast packets from going farther up in the hierarchy, and the other to restrict broadcast packets from entering the subsystems below them. Bitmaps are used to implement both filters, with each bit representing a physical page (of cachable memory). The bitmaps are maintained on a per page basis instead of a per cache line basis, because they are then much smaller and because they can be managed in software by the operating system. Since the operating system already manages and controls access to physical memory for each PM through its page tables, it can also manage the bitmaps without much extra effort, thereby eliminating the need for hardware to track memory accesses, and thus reducing the hardware substantially at the cost of some extra invalidation traffic.

Trace-driven simulation studies show that this consistency scheme performs well for common parallel scientific and engineering code [6].

3 Hector Prototype Implementation

We have built a prototype Hector machine to gain insight in the issues of performance, efficiency of design, packaging considerations, and suitability from the software point of view. The prototype consists of 32 PMs organized into 8 stations of four PMs each. Each station may contain up to 8 PMs. Each PM contains a Motorola 88100 processor [14], up to 256K of cache, 16 MBytes of memory, a *station bus interface* (SBI), and I/O interfaces that include serial, SCSI, and Ethernet ports.

Most of the PM logic has been implemented using PAL chips, which uses up much real estate on the printed circuit boards and has caused some constraints. For example, each PM occupies a full triple-height VME board (17" ×14"), which has constrained the attainable clock cycle time to 50 ns. Moreover, in view of scarce space on the board, we limited the PM memory input buffer to a single buffer, instead of a larger FIFO that could store multiple concurrent memory requests. We also decided not to implement hardware-supported cache consistency, but to leave cache consistency for the software to handle.

Our implementation includes a number of optimizations that simplify the hardware, allow a reduction in the clock cycle time, and/or reduce overall memory access latency. First, the station bus is centrally controlled by the station controller so that it can give higher priority to a packet on the ring which is destined for this station, by not granting the bus to a local PM. Whether a packet is destined for the station is determined in a timely fashion by examining the address of the packet in the latches of the *previous* station controller. This optimization is important also because it eliminates the need for extra pipeline stages at each ring-interface,[3] or the need to reduce the ring clock rate.

Second, there are two separate bus request lines from each PM to the station controller: one to request the bus for *on-station* packet transfers, and one to request the bus for *off-station* transfers. Off-station requests are granted only when there is a free slot on the local ring. On-station requests are granted whenever (1) there is no incoming packet on the ring destined for this station, and (2) none of the station PMs has initiated an off-station transfer. With this strategy, the station controllers do not require FIFO queues, and never need to drop packets.

Third, no explicit (N)ACK packet is transmitted over the bus at the destination station in the case of a write operation, or in the case where a read request cannot be received.

[3] Each additional pipeline stage in the ring path at the ring-interfaces reduces the overall bandwidth of the ring.

Instead, a *Received* line on the destination station bus is toggled at the end of the cycle in which the request packet is sent over the bus to indicate whether the packet was received by the target PM. For on-station requests, this line is interpreted by the source PM directly. For requests that come from the ring, this line is interpreted by the *next* station controller, to which it is directly connected. Hence, instead of requiring the target PM to generate and send the acknowledgement packet, the request packet on arrival at the target station interface is also forwarded to the next station on the ring when it is driven onto the station bus. There, the status of the packet is updated to become a (possibly negative) acknowledgement during the next cycle based on this Received signal. But if the packet was a read packet that was accepted by the target PM, then it is removed at the next station (by marking it empty), because the data return packet, generated later, serves as the acknowledgement. This schemes reduces the latency of some operations by at least two cycles.

The size of each packet in the prototype is 93 bits, enough to contain 64 bits of read data, or 32 bits of data and a 32-bit address for write requests. Packets are transferred in bit-parallel fashion. This is made possible by the fact that each station controller only has links to its two ring neighbors, and because the connection between rings involves only two rings.

Our implementation allows each PM to have only one outstanding request at a time, because this simplifies the design considerably. In order to support multiple outstanding requests (for memory prefetching or for processors with multiple data paths) the packets would have to be extended to include tags so that the response packets can be matched to the request packets. Moreover, each PM would require larger buffers in order to be able to accept all response packets.

Hector runs synchronously; that is, all PMs and station controllers are driven by a single master clock. In our prototype, a clock board generates differentially driven ECL clock signals, which are distributed to each board using a coax cable pair cut to proper length. This clock distribution scheme was viewed with skepticism by some experienced hardware designers, but it has worked very well. We have found that careful distribution and proper termination of clock signals within the PM boards is more important.

With respect to packaging, two stations are housed in one card cage with the two station controller boards in the middle. Multiple cages are stacked on top of one another. Adjacent station controllers are either vertically one above another, or (in the top and bottom most cages) next to each other in the same cage. All stations in a ring can therefore be connected by cables that are only 4 inches long.

With respect to performance, Table 1 depicts the cost of basic memory operations, as measured on a 16-processor prototype configured with four PMs per station. Included in the table are the times to access local, on-station and off-station memory. Off-station memory requests must traverse the ring consisting of four segments; in larger rings this time would increase by 1 cycle for each additional station connected to the ring. The right-most column includes the (contention-free) access times for a projected 256 processor system, configured with 4 processors per station, 4 stations per (local) level-1 ring, 4 level-1 rings per level-2 ring, and 4 level-2 rings per global ring, assuming that each request must traverse the entire ring hierarchy for each access. Also shown in the table is the memory access cost differential, ρ, which is the remote/local memory access time ratio. These numbers show that the effects of NUMA (Non Uniform Memory Access) is relatively mild, (although the NUMA effect becomes more severe once contention is considered, as will be discussed in Section 5).

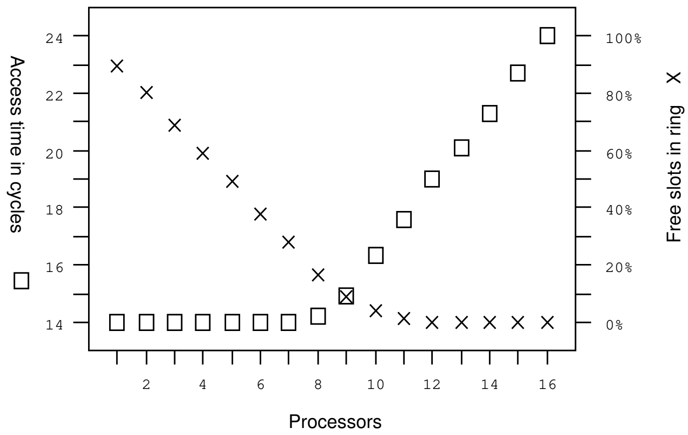

Figure 3: Average memory access times for varying degrees of ring traffic.

In practice, for real applications where instructions and private data can be cached, the extra overhead of traversing the ring is not very large. For example, in a four-station configuration, the execution time of a four processor implementation of a 512×512 point FFT is only 3% longer if the four processors are partitioned across two stations and only 4% longer if they are partitioned across four stations, when compared to the case where all 4 processors are in the same station.

Because in our implementation the rings are clocked at the same rate as the stations, incoming and outgoing buffers are not needed at the station controller. For smaller configurations, this actually leads to faster off-station memory accesses, because each buffer through which packets must pass increases the latency. For large configurations, it is essential to run rings at a faster clock rate, because the rings would otherwise congest and become a bottleneck. A simple experiment shows how quickly a ring can saturate. In this experiment, processors write to off-station memories as quickly as possible in an attempt to saturate the ring. Ring traffic is maximized, because write accesses have the shortest latencies in our implementation, and because each processor writes to a different memory module so that there is no memory contention.

The boxed curve in Figure 3 shows how the access time in-

operation		on-board	on-station	ρ	off-station	ρ	256 procs	ρ
read	(one 4 byte word)	10	15	1.5	19	1.9	35	3.5
cache fill	(16 bytes)	20	26	1.3	30	1.5	46	2.3
write	(one 4 byte word)	10	8	0.8	14	1.4	30	3.0
atomic swap		24	30	1.2	38	1.6	70	2.9

Table 1: Hector memory access time in cycles.

creases as the number of accessing processors is increased from 1 to 16 (in a 4-station configuration), and the crossed curve shows the percentage of free slots in the ring. In this stress test, the ring begins to saturate at 9 processors. Memory access time does not increase substantially up to this point; however, additional processors beyond 9 increase the average memory access time. The shape of the access time curve is flatter than might be expected in the 2 to 9 processor range, because the processors quickly fall into lockstep and do not interfere with each other in this particular experiment.) In the 12 to 16 processor range, the access time is determined only by the total available bandwidth over the number of processors. Note that the bandwidth of the ring does *not* increase as stations are added to the ring, but is dictated entirely by the maximum transfer rate at any of the nodes on the ring.[4] Therefore, the bandwidth of a ring can be increased only by increasing the clock rate.

4 The Good, the Bad, and the Unknown

In this section we review our design decisions and implementation choices, roughly characterizing them as either good decisions, probable mistakes, and as yet unresolved issues.

4.1 Good Decisions

We believe that the structure of Hector and its use of slotted rings was a good choice. Ring-based systems are not entirely new; the CDC Cyberplus [8] and the MIT Concert multiprocessor [5] are early ring implementations. More recently, the KSR multiprocessor [2, 3], which has a structure similar to Hector, has been brought to the market. Other ring architectures have been proposed, but not yet implemented, such as the Express Ring [1]. With the standardization of SCI (Scalable Coherent Interface) [11], we expect ring-based systems to become more common in the future.

The key aspect that sets Hector apart from other systems is the simplicity that permeates the design. Only by keeping the design and implementation simple were we able to successfully implement a prototype with very limited means — our total budget was C$700,000 — and develop a system that costs us on the order of C$5,000 per PM regardless of system size. More importantly, it is the simplicity of the design that results in a system with lower memory access latencies, and will allow very fast implementations (that can be proven correct) in the future. Table 2 contrasts the basic latencies of Hector with those of the Dash [12] and the KSR [4] multiprocessors, and shows that Hector has far lower latencies than the other two systems.

		local	on-station	off-station
Hector	read	10	15	19
	write	10	8	14
Dash	read	15	29	101
	write		16.7	88.7
KSR		18		126

Table 2: Basic Access Latencies for Hector, Dash and KSR. Both the Hector and Dash numbers are for a 16-processor system, configured into 4 stations; the KSR numbers are for a 32-processor system. (The Dash local read time corresponds to the time needed for a second-level cache access).

Figure 4 shows that Hector exhibits stable behavior in the presence of hot spots. In this experiment, p processors iteratively read and write to a single memory location in a tight loop. As can be seen from the figure, the memory access time grows linearly in the number of processors. The figure also gives a comparison with a 32 processor KSR machine as reported in [4].

To keep the design simple, Hector does not have special support for software synchronization beyond implementing the standard atomic memory access operations (`swap` in the case of the M88000). We believe that this has been the correct choice. Hardware support just adds complexity, and we have found that software-based solutions work very well. Figure 5 shows the time needed for a process to obtain and free a lock with $p-1$ competing processes using an algorithm due to Mellor-Crummey and Scott [13]. Similarly, Figure 6 shows the time needed for p processes to enter and exit a barrier, using a tree-based algorithm. These algorithms were run on a 16-processor Hector configured with 4 stations on a single

[4] Several recent papers have incorrectly hinted that ring bandwidth increases with the size of the system.While it is true that at any given time, larger rings can contain more data, the data on average must traverse a correspondingly longer distance.

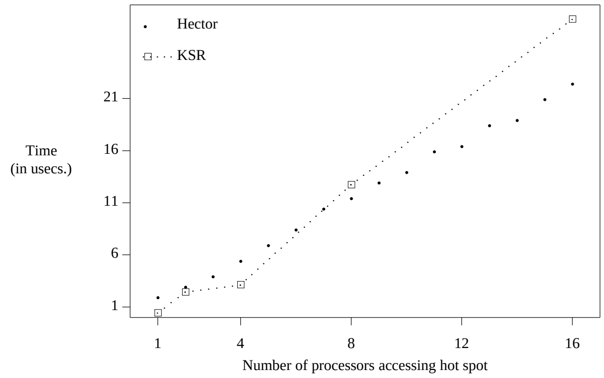

Figure 4: Time to read and write a hotspot shared variable

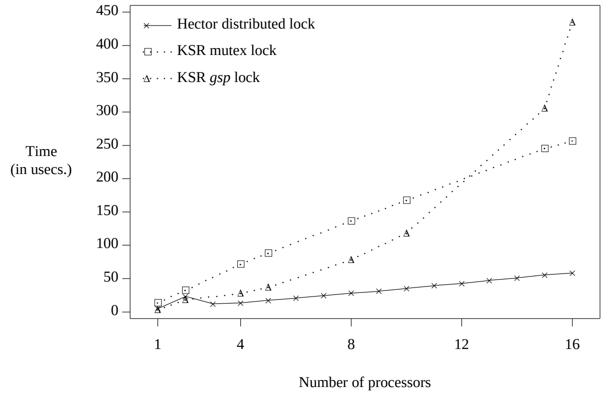

Figure 5: Response time for entering and exiting a null critical region

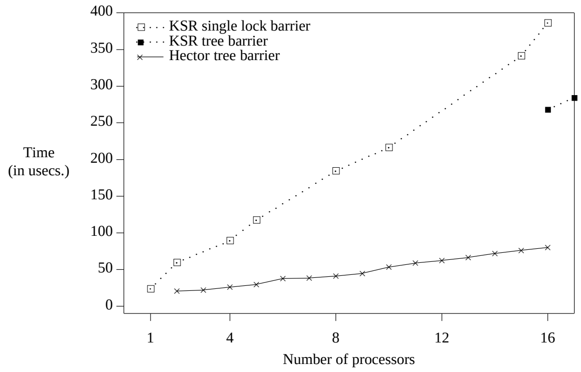

Figure 6: Barrier response time

ring. The figures also show the corresponding times for a 32-processor KSR system, as reported in [4].

Hector does not have a special facility to transfer blocks of data larger than a cache line. In a NUMA system, pages are often replicated and migrated, so a special block transfer support could be effective. We have extensively evaluated such support through simulation [20]. While block transfer support was found to reduce the time needed to transfer a block, this reduction comes at the cost of increased latencies for normal memory accesses. The simplest protocol investigated, namely the slotted ring in which block messages are broken up into individual packets and sent as separate messages (as is done with cache line fills), was found to be the best choice overall. In particular, it was found that concurrently reading from memory while transmitting the data as it becomes available, resulted in much better performance than if these phases occur serially. While we have not implemented such a facility, it would be simple to add, since it is a direct extension of the existing cache-line fetch protocol.

With respect to our implementation, the most difficult choice we had to make (surprisingly), was that of the microprocessor. For a state of the art implementation, a microprocessor that is new to the market must usually be chosen at a time when it is not clear which processor will have sufficient market share in the future. Market penetration is important because it increases the availability of support chips, software tools, and backwards-compatible future generation processor chipsets. In this regard, we consider our choice of Motorola M88100 as having been acceptable. While this chipset has not become as popular as the Sparc and Mips chipsets, it is used in Data General's product line and their work on porting the GNU compilers to the M88000 ensured that we had available a good working compiler.

The interrupt subsystem we implemented has proven to be a success. A processor interrupt can be triggered by swapping a 32-bit word into an interrupt register. The swap is successful only if there is no interrupt pending, and the software can detect whether the swap was successful or not by examining the return value of the swap. Any value may be swapped into the interrupt register and this value is available to the interrupt service routine. We typically swap in the pointer to an interrupt routine so that this function can be called quickly, and devices swap in values that identify the device and the type of interrupt. Unfortunately, we only have one such interrupt register per PM; having a few additional (prioritized) such registers would have helped simplify the implementation of the operating system.

Our PM implementation includes an interface that gives an external processor access to all important system resources. We have found that this interface greatly reduced the development time of both hardware and software. At the hardware level, it allowed the testing of subunits, such as the memory subsystem or the station bus interface, before the rest of the system was operational. For the software, it provided a temporary interface for I/O; initially, we used a 68000-based processor board with an Ethernet interface and Internet protocols to download operating system code, and to provide NFS access. Later, we attached SCSI and Ethernet cards to this interface. (A flaw in the implementation of this interface, however, is that we did not assign it a high enough priority for accessing system resources, causing, for example, our Ethernet board to occasionally time out when transferring (Ethernet) packets to a congested memory.)

Finally, each PM has a microsecond timer. These timers have proven to be invaluable for developing and experimenting with software.

4.2 Probable Mistakes

The original Hector design had tolerance of unreliable communication as a design feature; a simple timeout mechanism was to trigger the retransmission of a request packet if either a request or its response packet is lost (as is done in LAN protocols). The intent was to allow simpler hardware implementations, by allowing packets to be dropped whenever they are either too difficult to handle or at points of congestion, with the expectation that the timeout mechanism would recover. For example, it was envisioned that the inter-ring interfaces would have limited buffer space and that packets could be dropped whenever these buffers overflow.

In retrospect, this design choice is unfortunate for several reasons. First, in larger systems, timeout values are needed that are greater than the maximum round trip delay (including maximum time spent in any queues along the way). If the probability of a packet being dropped is not negligible, then these large timeout values result in high *average* memory access latencies, because much of the memory access latency is spent waiting for the timeout to occur.

Second, the cache consistency mechanism (which was designed later) cannot tolerate the loss of invalidation packets.

Third, if a write acknowledgement packet is dropped, then a data item may be written multiple times as a result of a single `store` instruction. This complicates software, requiring that writes to shared data be contained in critical regions; for example, without locks around writes in queue manipulation code, the queue may become inconsistent because of the non-atomicity of the writes.

Finally, it is not difficult to ensure that packets are not dropped (unless they are corrupted). The ring-interfaces are the only interfaces that might (due to congestion) drop a packet in the current design.[5] With FIFO buffers large enough to hold all packets in a system, it is no longer necessary to drop the packets. Not having packets dropped would allow a simpler implementation of atomic memory accesses, eliminating one of the packet transactions.

As another mistake, we underestimated the importance that memory throughput plays. When the project started, the interconnection network was the target of our focus because at that time bus-based systems were reaching the limits in terms of performance. With the backplane we have developed, the memory subsystem is now clearly the bottleneck, and (in retrospect) we did not pay enough attention to its design. For example, one of the compromises in our implementation is that a PM can service only one request at a time; requests that arrive while memory is servicing an earlier request are not queued at the target PM, but instead are negatively acknowledged so that they can be retransmitted later. Unfortunately, this results in a maximum sustainable memory utilization of only slightly over 50%. Simulation studies confirm that having just a single extra buffer to hold incoming requests could bring memory utilization close to 100%. Having an even larger FIFO buffer would then only reduce the number of retransmissions and hence network traffic, but would not have a further significant effect on memory utilization. Having multiple memory banks per processor would increase the aggregate memory bandwidth more dramatically. Our memory subsystem also has the capability to correct single-bit errors and detect double-bit errors, but this has clearly been unnecessary, using up valuable PCB real estate, and slowing down each memory access by a cycle.

Finally, an implementation detail we would change in a future system is our power distribution. We would put a DC-DC converter on all the boards and distribute a higher voltage to reduce current requirements of each board. This would not only make power distribution easier and smaller, but also decrease noise due to the added isolation.

4.3 Open Issues

Our Hector prototype does not provide for hardware supported cache consistency. On the one hand, this has been a drawback because our software efforts were distracted to some extent from studying the effects of non-uniform memory access times, one of the original goals. Instead, we were forced to concentrate on software-based consistency schemes.[6] On the other hand, this can also be seen as a plus, having found software techniques that are simple and effective in many cases (see the next section). In fact, we are now not convinced that hardware cache consistency is absolutely necessary; a hardware solution adds complexity to the design and hence will most likely add to the latency of all cache misses, and the scalability of any hardware-based scheme is still highly questionable. We continue to study this issue, but find it difficult to compare software techniques to hardware solutions in a direct and fair way on a system without hardware consistency support.

[5] Remember that PMs return negative acknowledgment packets when they cannot handle an incoming packet, and that negative acknowledgement packets can always be received when they arrive at their target.

[6] The latency of a cached and an uncached access differs by at least a factor 10.

Another open issue involves the rate at which unsuccessful memory accesses are retried. As soon as we had an 8-processor 2-station configuration, it became apparent that individual processors could be starved from accessing heavily contended memory where repeated attempts would be rejected. We experimented with several strategies, and have found that it is unsuitable to have processors retry as soon as they receive negative acknowledgements or retry after a random delay. Similarly, an exponential backoff approach (as used in the Ethernet, for example) does not help because that approach only works for lightly loaded resources. The solution we ultimately implemented uses a variable-delay backoff strategy. In the case of a failed request, the PM waits 13 cycles before retrying. If unsuccessful, it repeats this 10 times. Then, it waits 89 cycles before retrying again. The wait of 89 cycles may be repeated 100 times. If still unsuccessful, the PM keeps retrying after every 3 cycles until it succeeds. It gives up after 64,000 attempts and raises an interrupt. This scheme works well in our prototype, but *i*) we do not yet understand why, and *ii*) we think that it is unlikely to work well in larger systems. Processor starvation and backoff strategies are therefore important issues we are still studying.

Hector is designed to be scalable, but our prototype is too small to properly assess its scalability. We have simulated large Hector systems with up to 1024 processors and have found that the average memory access times are low enough to be able to obtain good application speedup, if (and only if) the applications exhibit a sufficient degree of locality [10]. Much scientific and engineering code (FFT, matrix multiplication, Gaussian elimination, Hough transforms, etc.) can be written to have a high degree of locality, and the operating system can increase locality by migrating and replicating memory pages. While the simulator accurately simulates a real system[7] — the Hector prototype was used to validate it — we found it necessary to use synthetic workloads in order to complete a simulation run within a reasonable time. A true test of the scalability of Hector architecture can only be done on a real system with many processors and real applications.

[7] As an interesting aside, we had implemented several Hector simulators during various stages of the design. These simulators were thought to accurately predict the behavior of the system, because they produced results that were plausible even under extreme cases. After having implemented the prototype, however, we found that the behavior of the real system differed in some circumstances quite significantly from the behavior predicted by the simulators. The differences were found to be due to minor programming errors and the abstracting away of what were thought to be unimportant details of the system. (For example, modelling memory refresh can make a difference when attempting to predict the performance of some simple applications.) In the end, a large amount of work was required to modify the simulators to have them more accurately match the behavior of the hardware. As a result of this experience, we now interpret simulation results more skeptically, regardless of where they may come from.

5 Software Considerations

An operating system, called Hurricane [17, 18], has been written for Hector and many parallel programs have been ported to the machine. We have found that parallel programs originally implemented for other architectures (especially those with uniform memory access times), did not perform well without specifically tuning the code for Hector. In particular, three aspects of the system need to be addressed. First, and most importantly (because of the lack of hardware cache consistency), the cachability of as much data as possible must be assured. The operating system automatically caches data pages that are accessed either by a single processor, or only read by multiple processors. During phase changes, the application can instruct the operating system to re-enable caching of individual memory regions at page granularities. (Coherence can be managed at a finer granularity if done by the application directly.)

Second, memory contention must be minimized. The effects of memory contention in our prototype are more pronounced because of the single memory access buffer at each PM, but they exist in any large system. Caching reduces memory contention to some extent, but we still observe serious contention at particular memory modules when parallel programs start up or enter a new phase, because all processes start accessing the same code and the same data. To reduce contention, the operating system automatically replicates read-only pages. Moreover, the application can instruct the operating system to spread the pages of a particular data region evenly over multiple PMs. Contention has the effect of increasing the difference in the cost of accessing local and remote memory, because local accesses are serviced with a higher priority (they never need to retry) than accesses from remote processors that may have to be retransmitted several times.

Finally, because the cost of accessing memory is a function of the distance between the accessing processor and the target memory, the placement of data needs to be managed carefully. The operating system can migrate pages to a particular PM; page replication can also bring a page copy closer to the accessing processors. The application can specify to the operating system where the pages of a particular region are to be located.

To summarize, the placement of the application processes, the cachability of the data, the placement of pages within the system, and the placement of data within pages need to be managed carefully to obtain good speedup. Because each page migration, page replication, or cache flush has a cost associated with it, these operations should only be invoked when their overheads can be amortized over a large number of subsequent, less expensive accesses. There are several reasons for limiting the degree of migration and replication. First, caching reduces the number of actual memory accesses relative to the number of processor load and store operations,

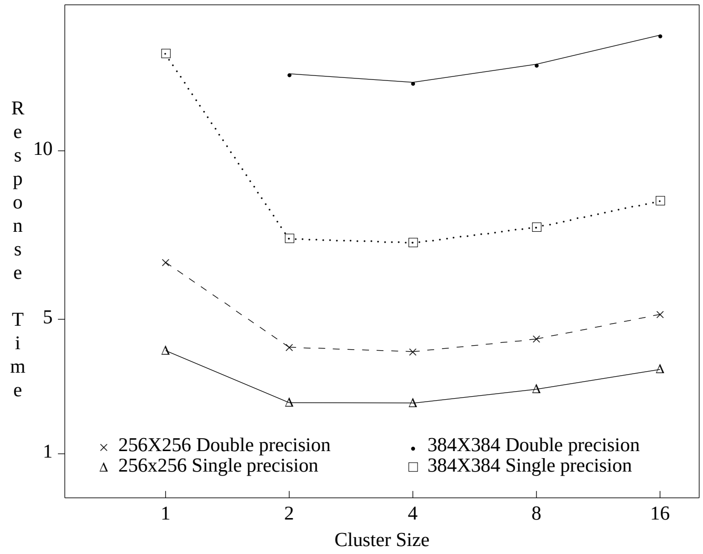

Figure 7: Response time (in seconds) of Matrix Multiply on a 16-processor, 4-station Hector as the operating system cluster size varies from 1 to 16

making it more difficult to amortize a page migration or replication. For example, with a low local/remote memory access cost ratio of only 1.6, and a page for which the cache hit rate is only 80%, the processor would have to execute 22,000 loads and stores to the page before migration starts to become cost effective.

Second, the page fault behavior in many applications is bursty in nature. For example, in a parallel matrix multiply ($Z = X \times Y$), where the computation is partitioned by row, all participating processors access the first row of matrix X at the very beginning of the computation. Contention at the memories containing X would be severe if all pages of X were copied to each PM participating in the computation, effectively increasing the cost to replicate them.

Third, replication increases the total amount of memory required per application (since individual pages appear in several memories), possibly leading to increased I/O to backing store. For example, a 1024×1024 matrix replicated onto each PM uses 8 MBytes on each PM. By reducing the degree of replication, more memory becomes available, lowering the amount of paging to and from backing store, with an attendant improvement in performance even if the number of remote accesses is increased overall.

For these reasons, and because the local/remote memory access cost ratio is reasonably small within a station, our operating system replicates pages only on a per station basis, and it migrates pages only across stations, never within a station. This clustering approach is used throughout much of the operating system, where objects are shared within a cluster but replicated and/or migrated across clusters. For example, the internal data structures of the operating system are managed in this way [17, 18]. Also, processes of an application are scheduled to minimize the number of clusters they span [21]. Figure 7 shows that this clustering can have an effect on application performance. In this figure, a matrix multiplication algorithm is run on a 16-processor Hector configured with 4 stations on a ring. While the hardware configuration remains unchanged, the matrix multiplication algorithm is run with the operating system configured with varying cluster sizes.[8] The size of the operating cluster defines the degree of page replication and migration, and more generally defines the locality in which operating resources are managed. From the figure, it is apparent that having operating system clusters has a direct performance advantage, yet that the cluster size can be made too small.

Although page-based cache consistency, page migration, and page replication occur transparent to the application, the application still needs to participate directly in the management of memory, specifying how memory regions are to be managed. Moreover, since all of the above mechanisms operate at page granularity, the application also needs to minimize false sharing. False sharing occurs when data that is used in different ways is allocated to the same page, resulting in poor performance. For example, if read-only data is allocated on the same page as data that is write shared, then the read-only data cannot be replicated and cannot be cached. Our execution environment provides for a sophisticated memory allocator, allowing the application programmer to specify which data may be co-located on a page. However, in our experience, programmers have difficulties assessing in advance how and by which processes data will be accessed.

Sandhu, et.al. have developed a method, called Romm, to support data replication, migration and coherence at the granularity of arbitrary objects [15]. Romm is based on the observation that accesses to shared data typically occur within a critical region, where a lock is obtained prior to the accesses and freed after the accesses. These synchronization calls are extended to also manage the location, replication and cachability of the objects controlled by their lock. The programmer can chose from (and experiment with) several protocols. For example, setting a read lock (which may be held by several processes simultaneously) may simply enable the caching of the corresponding object, or it may replicate the object into local memory. Freeing the lock then takes whatever action is needed to ensure the consistency of the object when accessed in the future. Experience indicates that this scheme performs well across a large range of applications, and that it typically does not require much effort from the programmer.

Figure 8 depicts the speedup of parallel implementations of LU decomposition with a problem size (600×600) that maximize false sharing when the data is managed at the granularity of a page. Initially LU is run as a UMA program without

[8] Note that the operating system clusters can be varied independently of the hardware configuration.

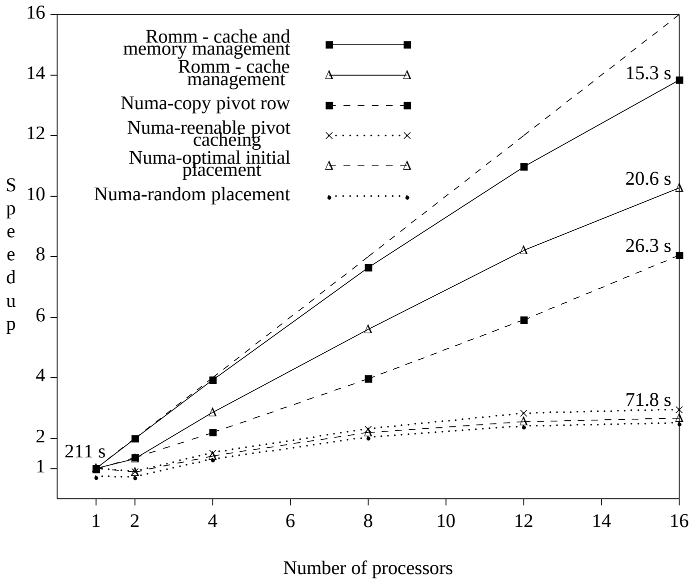

Figure 8: Speedup for 600×600 LU decomposition with various strategies for managing memory

any optimizations. Then, LU is run with pages optimally placed; this improves the performance only slightly. As a further optimization, caching is reenabled (on a per page basis) for pivot rows, which leads to a slight further improvement. Because these optimizations are all page-based, and because there is much false sharing in this example, they do not improve performance by much. (The page-based solutions do work well if instead a nicely lined up 512×512 Matrix is used.) A better improvement is achieved if the program is modified to explicitly copy the pivot row into each processor's memory, but from a programmer's point of view, this is cumbersome. With Romm, the programmer need only properly lock the shared data before accessing it, and identify which protocols to use. The figure shows the speedup obtained when Romm just manages the caches, and when it manages both the caches and the memory.

Overall, the most serious challenge that needs to be addressed on the software side is that of eliminating false sharing. Page-based strategies often do not work well because of false sharing, but even the object-based methods can perform poorly because of false sharing within a cache line. Similarly, Cache Only Memory Architectures (COMA) [16] (such as the KSR), where memory segments are replicated and migrated directly by the hardware, can perform poorly because of false sharing. It is for this reason that we do not believe the complexity introduced for COMA support is warranted. For COMA systems to perform well without programmer intervention, it is necessary to solve the false sharing problem; but if that problem is solved, then a page-based software solution will be just as effective without the attendant complexity of hardware that slows down common memory accesses.

6 Concluding Remarks

As part of the Hector project, we have developed a novel multiprocessor architecture and implemented a prototype machine. We have also developed the Hurricane operating system and other system software, and are now running parallel applications on the machine in various configurations. In this paper, we gave a brief overview of the architecture and implementation, and presented some of our experiences with the project. While we pointed out the mistakes we made, we feel that the positive aspects of the Hector project far outweigh the negative aspects. The project to date has, in our opinion, been very successful, especially in light of the fact that we now have a near production quality prototype in daily use. Building a prototype, even though difficult in a university environment, has proven to be a worthwhile experience. It brought into focus all aspects of the system, something which would have been impossible to accomplish though simulations alone.

The basic Hector architecture is sound, even if its distributed memory makes it challenging for software developers to exploit its performance potential. We strongly believe that non-uniform access times, due to the distribution of shared memory across the system, will be a fact of life. The many parallel applications that run with only a small degree of parallelism, and the many large-scale parallel applications that exhibit a large degree of locality in their data accesses, benefit directly from the faster memory access times available in NUMA systems. The architecture and the prototype we have implemented demonstrate that scalable shared-memory multiprocessors need not be more complex than scalable distributed-memory multicomputers, yet they can only perform better.

While most of our efforts are now focused on further software development at the operating and run-time library level, we have started the process of designing a second generation Hector system. The design of the memory subsystem will receive much attention because of the demands placed on it by newer processor architectures with very high clock rates, concurrent memory accesses and wide data paths. We will provide some hardware support for cache consistency, most probably as part of a hybrid software-hardware solution.

Our primary objective of keeping the architecture and design simple will continue to guide our efforts. We believe that meeting this objective is the only way to attain high performance and scalability. But more importantly, because large-scale shared-memory multiprocessor technology is still immature, it is necessary to iterate on designs and experiences quickly in order to make forward progress. This is only possible if the designs are kept simple.

Acknowledgements

We gratefully acknowledge the contributions of David Lewis who helped design our PM, Kennedy Attong, who helped build the prototype, Orran Krieger and Jan Medved, who designed the I/O modules on our PM. Orran Krieger, Yonatan Hanna, Ben Gamsa and Fernando Nasser who contributed significantly to our system software. The region locks and Figure 8 are due to Harjinder Sandhu, Ben Gamsa and Songnian Zhou. Comments by Orran Krieger and Steve Wilton helped improve the paper.

References

[1] Luiz Barroso and Michel Dubois. Cache coherence on a slotted ring. *Proc. of the International Conference on Parallel Processing*, 1991.

[2] G. Bell. Ultracomputers: A teraflop before its time. *Communications of the ACM*, 35(8):27–47, August 1992.

[3] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR 1 computer system. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, February 1992.

[4] T.H. Dunigar. Kendall Square multiprocessor: Early experiences and performance. Technical Report ORNL/TM-12065, Oak Ridge National Laboratory, May 1992.

[5] R. Halstead Jr. et al. Concert: Design of a multiprocessor development system. In *Proc. 13th Annual International Symposium on Computer Architecture*, pages 40–48, June 1986.

[6] K. Farkas, Z. Vranesic, and M. Stumm. Cache consistency in hierarchical-ring-based multiprocessors. In *Proc. Supercomputing 92*, 1992.

[7] K.I. Farkas. A decentralized hierarchical cache-consistency scheme for shared-memory multiprocessors. Master's thesis, University of Toronto, April 1991.

[8] M. Ferrante. CYBERPLUS and MAP V interprocessor communications for parallel and array processor systems. In W.J. Karplus, editor, *Multiprocessors and Array Processors*, pages 45–54, 1987.

[9] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th International Symposium on Computer Architecture*, pages 15–26, 1990.

[10] M.A. Holliday, D.S. Kindred, and M. Stumm. On the scalability of ring-based, hierarchical sahred memory multiprocessors. Technical report, Computer Systems Research Institute, University of Toronto, 1991.

[11] SCI (Scalable Coherent Interface): An overview. Technical Report P1596: Part I, doc171-i, Draft 0.59, IEEE, February 1990.

[12] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Implementation and performance. In *Proc. 19th Annual International Symposium on Computer Architecture*, pages 92–105, May 1992.

[13] J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[14] Motorola Inc. *MC88100 RISC microprocessor User's Manual*, 1988.

[15] H. Sandhu, B. Gamsa, and S. Zhou. ROMM: Region oriented memory management. Technical Report CSRI-269, Computer Systems Research Institute, University of Toronto, 1991.

[16] P. Stenström, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proc. 19th Annual International Symposium on Computer Architecture*, pages 80–91, May 1992.

[17] M. Stumm, R. Unrau, and O. Krieger. Designing a scalable operating system for shared memory multiprocessors. In *Proc. Usenix Conference on Micro-kernels and Other Kernel Architectures*, pages 285–303, April 1992.

[18] R. Unrau, M. Stumm, and O. Krieger. Hierarchical clustering: A structure for scalable multiprocesssor operating system design. Technical Report CSRI-268, Computer Systems Research Institute, University of Toronto, 1991.

[19] Z. Vranesic, M. Stumm, D. Lewis, and R. White. Hector — A hierarchically structured shared memory multiprocessor. *IEEE Computer*, 24(1):72–80, January 1991.

[20] S.J.E. Wilton. Block transfers in a shared memory multiprocessor. Master's thesis, University of Toronto, 1992.

[21] S. Zhou and T. Brecht. Processor pool-based scheduling for large-scale NUMA multiprocessors. In *Proc. ACM Sigmetrics Conference*, September 1990.