Network Interface Design for Low Latency Request-Response Protocols

Mario Flajslik
Stanford University

Mendel Rosenblum
Stanford University

Abstract

Ethernet network interfaces in commodity systems are designed with a focus on achieving high bandwidth at low CPU utilization, while often sacrificing latency. This approach is viable only if the high interface latency is still overwhelmingly dominated by software request processing times. However, recent efforts to lower software latency in request-response based systems, such as memcached and RAMCloud, have promoted network interface into a significant contributor to the overall latency. We present a low latency network interface design suitable for request-response based applications. Evaluation on a prototype FPGA implementation has demonstrated that our design exhibits more than double latency improvements without a meaningful negative impact on either bandwidth or CPU power. We also investigate latency-power tradeoffs between using interrupts and polling, as well as the effects of processor's low power states.

1. Introduction

Historically, network card latency has been overshadowed by long wide area links and slow software, both of which easily bring the overall latency into the millisecond range. More recently, datacenter applications have emerged with more rigorous latency requirements, thus inspiring efforts to reach low latency on commodity systems. An example of such datacenter applications are various database and caching services that operate from main memory, such as memcached [6]. Moreover, there are ongoing efforts to build ultra low latency software, such as the RAMCloud project [22]. RAMCloud is a durable storage system boasting latency goals of 5-10 us round trip, inside a commodity datacenter.

There are economic reasons that lead us to believe that the commodity low latency trend will continue for the foreseeable future, and will not be limited to the high performance niche. The ability to do evermore processing and continuously add new features is essential to many software companies' differentiation strategies, thus being the driving force behind generating revenue. These companies' services must run within a small latency budget, demanding commodity technology that provides low latencies.

On the academic front there has been an analogous, and likely correlated, interest in low latency. Some software efforts were already mentioned, but other disciplines also contribute. For example, network and interconnect research yielded ideas for bufferless switching [1] and new network topologies [16]. We position our work between network and software research, right on the interface connecting the host and the network.

We adopt a clean slate approach to the problem and build the lowest latency request-response system that we can. Following the clean slate approach, we developed a very simple, and very fast, minimal object store evaluation application that supports only two operations: GET and SET. The minimal object store application exhibits low absolute latency, but it also has low latency variability. Due to this application choice, the latency focus shifts onto the network interface design, which is our paper's main contribution. We focus on two latency sources in the network interface: 1) control communication and data transfer between the CPU and the network card; 2) processor idle state wakeup times and power management.

Our interface, NIQ (Network Interface Quibbles), was designed after a detailed investigation of reasons behind latency inefficiencies in current network cards, as described in Section 2. We found that one of our key objectives must be to minimize the number of transitions over the PCIe interconnect. This is especially true for small packets, which are prevalent in request-response protocols. In Section 3 we describe how combining existing techniques (e.g. embedding small packets inside descriptors) with new ideas (e.g. custom polling, creative use of caching policies) leads to a low latency interface that does not sacrifice bandwidth.

To evaluate NIQ in detail and compare it to other possible solutions, we built a configurable FPGA-based network card. This network card is configured and controlled by a user-space NIQ driver that provides zero copy capabilities and offers direct application access through bypassing the kernel stack. Evaluation system is described in Section 4, together with bandwidth and latency performance analysis. CPU idle state power measurements and power-latency tradeoffs of interrupts and polling are presented in Section 5.

Ideally, total network latency would be dominated by

wire propagation delay which is limited by the speed of light to around 5 nanoseconds per meter. Assuming total round trip distances of under 200 meters inside the datacenter, total time spent on the wire is under one microsecond. The ultimate challenge is to bring network card latencies into the same range. In the meantime, our NIQ achieved the best round trip latency (client-server-client, with a network cable in between) of 4.65 us. However, much of that time is inherent to the hardware components we had available. In Section 7 we discuss the possibility of round trip latencies under 2.3 us with state of the art components and an ASIC implementation.

2. Network Interface Card Design

When implementing request-response protocols on commodity systems, attempts at low latency often run into a system designed for a wide area network where latency is secondary to achieving high bandwidth. We find current designs to be lacking in latency performance, even though they are well suited for high bandwidth systems that are dominated by software latency. Our motivation are new low latency systems, such as the RAMCloud project [22], that have software overheads in the one microsecond range. As a comparison, one round trip time through an idle linux kernel networking stack measures at 32-37 us. This measurement includes one receive and one transmit path between the NIC driver and the user application for UDP (32 us) and TCP (37 us) packets. In this section we highlight the network controller challenges to achieving low request-response latency on existing systems.

To illustrate the problems addressed in this paper, we examine in detail how one would build a request-response based system on top of a current commodity 10G Ethernet NIC. We assume an Intel 10G x520-DA2 adapter [14] because we have access to one, but other network controllers, such as Broadcom NetXtreme, exhibit similar behavior [11, 4].

A typical network card is connected to the host system over PCI Express (PCIe) and contains these components: DMA engine, ring buffers, Ethernet MAC and PHY, plus additional features (offload engines, QoS, virtualization, etc.). The DMA engine connects directly to the PCIe interface and transfers data between the host memory and the ring buffers on the NIC. Performance of the MAC and the PHY, as well as the PCIe interconnect, contribute to overall performance, but in this paper we focus on the interface between the NIC and the host. This interface is defined by the functionality of the ring buffers and how they are managed by the NIC and the host driver. We describe the interactions between the host and the NIC by stepping through the packet transmit and the packet receive process, shown in Figure 1.

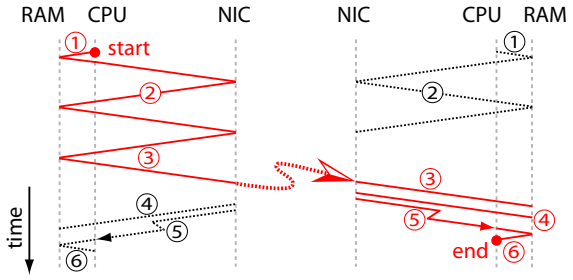

Figure 1: Timing of TX (left) and RX (right) steps.

2.1. Transmit steps

When a client application decides to make a request, it formats a request packet and sends it to the server. The NIC transmit interface requires adding packet metadata (i.e. the packet descriptor) to the main memory descriptor ring shared between the CPU and the NIC (step 1 in Figure 1 left). To inform the NIC of a newly available transmit descriptor, the CPU does an uncached I/O write to the "doorbell" register on the NIC (first part of step 2). The NIC, then, uses its DMA engine to fetch the next transmit descriptor (last part of step 2). The DMA engine is also employed to fetch the contents of the packet that is ready for transmission, based on information obtained in the packet descriptor (step 3). Overall, it takes two and a half PCIe round trips to get from software transmit initiation to the packet being available in the NIC.

Described use of the descriptor ring buffer provides decoupling between the CPU and the NIC, thereby allowing the CPU to get ahead of the NIC in the number of transmitted packets. Bursts of packets from the CPU are effectively queued in the ring buffer which is drained by the NIC as fast as the network allows. This decoupling is key to achieving high bandwidth.

At this point the packet is on the wire and on its way to the server, but there is still some necessary client NIC bookkeeping. When a packet is handed to the NIC for transmission, its memory cannot be reused until the DMA is completed. Upon DMA completion, the NIC sets a flag in host memory indicating that the packet buffer may be reused (step 4). Typically, there is space reserved for this flag in the packet's descriptor entry in host memory. Following step 4, an interrupt is generated (step 5) that triggers the CPU completion handling (step 6) and thus completing the transmit process. Bookkeeping (steps 4-6) is not in the critical latency path of the request-response protocol, but it must be done promptly to prevent the client from running out of resources (e.g. memory space, or flow control credits).

2.2. Receive steps

Before the NIC can receive any packets, it must be loaded with information about available receive packet buffers in main memory. A descriptor ring buffer is used

to keep the available receive descriptors, similar to the transmit descriptor ring. Prior to any packets arriving, the driver allocates multiple receive packet buffers, creates receive descriptors pointing to those buffers, and transfers the descriptors to the NIC (steps 1 and 2 in Figure 1 right).

Once the client's request packet finds its way through the network, it arrives at the server's NIC. Upon packet arrival, the NIC reads the next available descriptor entry to determine where to deposit the packet. After depositing the packet into the host memory using the DMA engine (step 3), the NIC notifies the CPU of the packets arrival. The appropriate packet descriptor ring entry is repurposed as a completion entry, now containing packet length and a flag indicating there is a new valid packet (step 4).

Just as they did in the transmit case, the ring buffer structures allow the NIC to run ahead of the CPU and deposit packets faster than the CPU can process them, at least in bursts. The CPU must read the ring completion entry to discover the location and size of the arrived packet. To avoid dedicating a CPU core to monitoring the completion ring, operating systems prefer to configure the card to interrupt the CPU as a form of a completion notification (step 5). Under high loads, the NIC might generate too many interrupts for the host to handle, thereby putting the receiver at risk of a livelock. One of the mechanism that mitigates this problem is interrupt coalescing in the NIC [26], where the NIC can coalesce several received packets together by deferring receive notifications and eventually triggering only one interrupt for several received packets. Under low load this method adds significant latency (even hundreds of microseconds) because interrupt generation is in the critical receive latency path. As an alternative to interrupts, the host can operate in a polling mode, continuously reading the next expected completion ring entry until it becomes valid. Modern operating systems alternate between polling and interrupt modes depending on the load, thus avoiding the receiver livelock issue [21] regardless of interrupt coalescing.

Finally, the CPU reads the receive descriptor (step 6) and the received packet, which is then forwarded to the application layer for processing. Also, at this point the CPU allocates a new receive buffer and updates the NIC with a new receive descriptor to replace the one that was just used. After processing the request, the server application formats a reply packet and sends it back to the client following the same transmit-receive sequence that was just described, thus completing one request-response round trip. Overall, 16 one-way transitions over PCIe links take place between client initiating a request and receiving a response from the server. Out of the 16 one-way transitions, 12 are synchronous, which means they

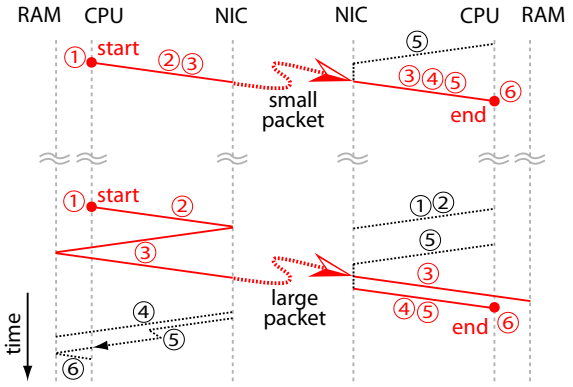

Figure 2: NIQ timing of small and large packets.

have to be completed before the next one can begin, thus affecting the overall latency.

3. NIQ: Interface Design

To resolve our Network Interface Quibbles, we propose a new network interface design, NIQ, that enables low latency implementations of request-response protocols. We achieve that goal by focusing on small packets and reducing the number of PCIe transitions.

The key insight with regard to the nature of request-response protocols is that at least half of the network packets are very small. This is true because either the read request, or the write response contain no data. Another important general insight is that network latency of short packets is generally more critical than the latency of large packets. Whenever large chunks of data are involved, inherent serialization and software processing latencies are higher, lessening the importance of other latency sources.

Network card design described in Section 2 and illustrated in Figure 1 is particularly inefficient for small packets because it requires a total of eight PCIe transitions for send and receive combined. Crossing over the PCIe interconnect is costly in terms of latency (over 0.9 us round trip is our system), therefore we focus on minimizing the number of PCIe transitions. In the best case scenario, only two transitions would suffice: one for transmission and one for reception. We present an interface that accomplishes the best case for small packets, but also provides good results for larger packets.

3.1. Small Packets

NIQ exploits the fact that modern processors are optimized around cache-line size transfers. Minimum size Ethernet packets are 60 bytes in length, thereby fitting into a 64 byte cache line. Note that minimum Ethernet size is usually said to be 64 bytes, but that includes the 4-byte FCS (Frame Check Sequence), although it does not include the preamble and the start delimiter. All mod-

ern network cards generate and strip the FCS in hardware and never expose it to higher layers, reducing the effective minimum packet size to 60 bytes. Observation that minimum size packets fit inside a 64 byte cache line serves as an important guideline to designing the small packet interface. Processors are already optimized for communication using cache lines, so we assume that small packets are minimum size 60 byte Ethernet packets. Expanding the interface to support, for example, two cache-line size packets is very much possible, and we discuss it in Section (Section 7). Timing of the small packet transmit and receive sequences is illustrated in the top part of Figure 2, with step numbers corresponding to Figure 1.

To achieve the ideal goal of one PCIe transition on transmit and one transition on receive, NIQ interface folds all critical steps from Figure 1 into just a single step. Folding of the transmission steps into one is accomplished by embedding the entire small packet within the transmit descriptor. Moreover, we do not employ the DMA engine to transfer the descriptor, but instead the descriptor is transferred by the CPU directly to the network card. Our transmit descriptors are one cache line wide, with flags indicating whether an entire small packet is embedded in it, or it is a traditional descriptor with packet address and length.

On the receive side, folding of the steps is achieved by embedding the entire small packet inside the completion entry. Upon reception, the entire small packet is transferred within a cache line wide completion, instead of copying it into a host buffer via the DMA engine. Assuming the use of polling (discussed later in this section), the completion containing the data also serves as a notification, thus successfully folding all critical receive steps.

As an additional benefit of transferring small packets in this fashion, no host memory buffers are consumed on transmit or receive. Since no host memory is consumed, there is no need to allocate or free any buffers, thus reducing the total amount of software bookkeeping. It is still necessary to exchange flow control credit information between the network card and the host, but that can be batched and done less frequently.

As was already mentioned, both transmit descriptors and receive completions are 64 bytes wide. To efficiently communicate with the network card in cache line size units, we utilize the cache hierarchy and write-gathering buffers. All memory that is written by the CPU is mapped as write-gathering (also called write combining), while the memory that is read by the CPU is mapped as cacheable. This is a departure from standard practices of mapping I/O memory as uncacheable, but it is similar to graphics cards practice of using write-gathering policy for mapping frame buffers.

Any write by the CPU made to a write-gathering address bypasses the cache hierarchy and goes into one of CPU's write-gathering buffers. Once the entire cache line is written, or a memory-ordering instruction (such as *sfence*) is executed, the entire cache line is flushed over PCIe to the network card [12]. Combining the writes in a buffer close to the CPU core improves CPU bandwidth and PCIe bandwidth. In fact, it would not be feasible for the CPU to use uncached writes to write the descriptor over PCIe 8 bytes at a time. Each PCIe packet incurs up to 28 bytes of header overhead across all layers [23], resulting in a 77% overhead. The number of overhead bytes is the same regardless of data payload size, making 64 byte transfers more efficient.

The cacheable mapping of completion entries enables us to issue cache misses to the network card, transferring the entire 64 byte completion in one PCIe packet. Moreover, the method of polling the network card puts the completion entry all the way into the first level cache, where it is ready for immediate processing. In order to force the cache miss, an appropriate *clflush* instruction is executed before the polling read. One of the implications of using a cacheable memory type in this fashion is that read side-effects are not allowed in the NIQ. For example, a NIQ address location might be read multiple times, either speculatively or due to a cache eviction. Therefore, we cannot rely on reads to inform us when the CPU has processed the data, but instead we require explicit flow control notifications.

3.2. Large Packets

For large packets NIQ still uses cache lines for communication, but entire packets no longer fit within a descriptor. Instead, we follow the traditional approach of employing the DMA engine to transfer packet data, as discussed in Section 2. Descriptors and completions are still cacheable and 64 bytes wide, but instead of embedding the entire packet (like in the small packet case), they only contain the first 48 bytes of the packet, which includes headers. Putting the headers inside the descriptor/completion enables an efficient implementation of header splitting on transmit/receive.

Header splitting on transmit is necessary to enable zero-copy techniques often used in low latency system implementations. If the header cannot be split from the payload, it is necessary to copy the packet data into an intermediate memory buffer before transmit. To avoid copying the header and the data into a single buffer, the network card's DMA engine can be programmed to transfer them separately and join them before they leave the card. However, programming the DMA engine to perform two transfers typically requires two separate transmit descriptors. Our interface enables header splitting with just a single transmit descriptor. Splitting the header on reception can also be beneficial, because the

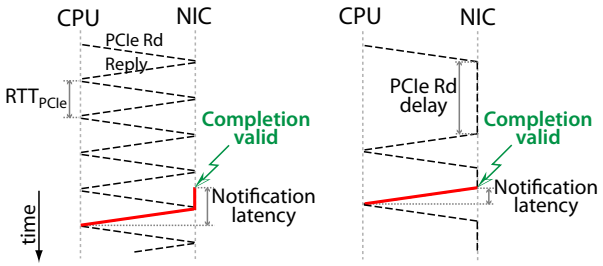

Figure 3: Notification timing diagram without (left) and with (right) PCIe read delay.

header is available for CPU processing as soon as the completion is read, as they are in the same cache line.

We have explored and evaluated the option of using the CPU to transfer entire large packets through the write-gathering buffers, but we found the CPU bandwidth penalties increase significantly with packet size (up to 70% penalty for largest packets). Even though the small packet interface provides lower latency, it is entirely possible to send small packets through the large packet interface. This is recommended when strict packet ordering is important because a small packet that is sent through the small packet interface is allowed to pass a large packet and leave the network card out of order. Allowing small packets to bypass large packets is meant as a feature, since it can easily save over 1 us of transmit time for the small packets. It is entirely possible to enforce strict ordering on the interface, but since the underlying network doesn't guarantee packet ordering, we choose to allow reordering.

There is some necessary bookkeeping of host memory buffers taking place off the latency critical request-response path, much like we discussed in Section 2. On the transmit path, this processing is batched for efficiency reasons and an interrupt scheme employing interrupt coalescing is used to notify the CPU of the necessary bookkeeping. NIQ uses interrupts for bookkeeping notification, but for critical notifications it employs a custom polling scheme, which we describe next.

3.3. NIQ polling

For the NIQ interface we designed a custom polling technique that we refer to as NIQ polling. Instead of polling the host memory, NIQ polling repeatedly issues reads over PCIe to the network card, thus avoiding communicating through the main memory. Additionally, the replies from the network card are put directly into the first level cache, thereby avoiding a cache miss.

The goal of the NIQ polling notification scheme is to minimize *notification latency*. We define *notification latency* as the time between a new valid completion entry being ready in the NIQ and when that completion is ready for processing in the CPU. When repeatedly polling the network card (illustrated on the left of Figure 3), *notification latency* is between half a PCIe round trip and one and a half round trip. The expected *notification latency* value is one PCIe round trip because it may take up to one whole round trip time between the completion being ready and the CPU polling read arriving at the network card.

To lower the expected *notification latency* we introduce a *PCIe read delay* time inside the network card. When there are no new valid completions, the network card holds on to the polling read for one *PCIe read delay* time before eventually replying with an invalid entry, instead of replying immediately. However, if a new completion is ready, a reply is generated immediately, thereby significantly reducing the expected *notification latency* (up to 2x reduction). This process is illustrated on the right of Figure 3. As an added benefit of *PCIe read delay*, fewer invalid entries get transferred over PCIe, also reducing the number of invalid entries the CPU must process.

Choosing the correct value for *PCIe read delay* is a balancing act between minimizing expected *notification latency* and avoiding triggering any deadlock prevention or watchdog mechanisms that could be triggered by a delayed memory read. Expected *notification latency* is inversely proportional to 1+(*PCIe read delay*), and we found that a *PCIe read delay* value of about 20 PCIe round trips gives good latency performance. This delay value is well clear of triggering any operating system watchdog mechanisms, but we have encountered an interesting interaction with Intel's implementation of simultaneous multithreading, known as hyper threading. While a hyper thread is waiting on the polling read to return, one would expect its sibling thread to make full use of the execution units and other shared resources, leading to NIQ polling being an even more attractive solution. However, we found that when using NIQ polling and delaying the read response for longer than around 4 us (or 10000 cycles), the sibling hyper thread makes no forward progress. We attribute this phenomenon to a deadlock/starvation prevention mechanism that detects the polling thread has not made progress for a long time (while waiting on the poll read), and upon detection preventatively stalls the sibling thread. For this reason we do not use hyper threading.

Polling in this way permanently consumes one PCIe read credit, but is unlikely to cause any issues on the PCIe bus because the PCIe spec allows for read delays of at least 10 ms [23]. Next, we discuss alternatives to our NIQ polling technique.

3.3.1. Interrupts vs. host polling vs. mwait

Historically, there was a huge speed mismatch between the I/O devices and the CPU, making interrupt

schemes necessary and efficient. It would be unfeasible for the single core CPU to wait several milliseconds for an I/O device to complete an operation. However, modern network cards are much faster and modern CPUs have multiple cores, thus changing the balance. One would still like the CPU core to be able to do other processing while waiting on the network device, but often there is nothing else to do. This reasoning coupled with poor latency performance of interrupts is what makes polling an attractive option.

We find three main reasons for poor interrupt latency performance. Firstly, we measure a 1.4 us delay between when the interrupt controller is instructed to generate an interrupt, and the linux interrupt handler is executed. The second latency penalty comes from the necessary inter-thread communication between the interrupt handler and the user application thread. The inter-thread communication is necessary because implementing the entire user application (i.e. consuming and generating packets) inside an interrupt handler is impractical at best. The third reason interrupts have a bad latency reputation is power management, and specifically CPU idle states. While waiting for an interrupt the CPU is not busy (unlike when polling) and often reaches a deep idle state with an exit time in tens of microseconds. We investigate the power management tradeoffs further in Section 5.

Polling on the host memory location is a good low latency alternative to interrupts. When using host memory polling, the CPU reads the memory location of the next expected completion entry in a tight loop until that location becomes valid. Because the reads are done in a tight loop, the memory location is cached and reads return quickly for as long as the completion entry is invalid. Due to reads hitting in the cache, the CPU must do a lot of useless work processing invalid entries, realizing they are invalid, and reading them again. When the network card actually updates the completion entry, the cache line gets invalidated and the CPU incurs a cache miss when it tries to read it. To avoid spinning on a memory location in a tight loop, the CPU can issue a *monitor* instruction for that cache line, followed by an *mwait* instruction that halts the processor to save power [12]. As soon as the monitored cache line is invalidated by the network card's write, the CPU is woken up to process the completion. Section 5 provides experimental evaluation of latency-power tradeoffs between interrupts and polling.

4. Object Store Evaluation

In this section we evaluate latency and bandwidth performance of a NIQ implementation. Our implementation is based on a dual socket system with Intel Westmere processors and a NetFPGA [29] board, as shown in the block diagram in Figure 4. The system has a minimum PCIe read round trip latency of 930 ns, which includes

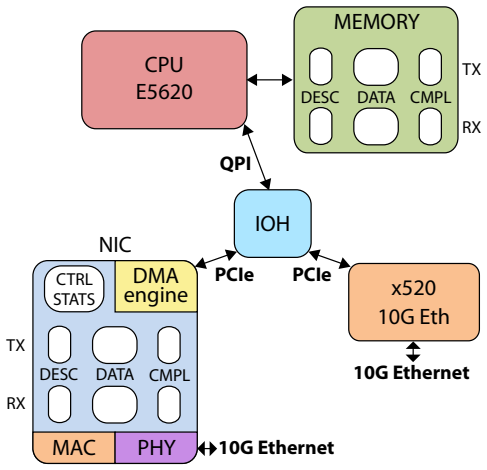

Figure 4: Block diagram of the test system.

the FPGA and the host machine, but is mostly dominated by the FPGA. NetFPGA Ethernet PHY and MAC round trip latency is 868 ns + 0.8 ns/byte, giving a range of 920 ns to 2.08 us for smallest to largest packets. All measurements in this section are based on a request-response roundtrip of a single-threaded minimal object store application. The object store application was built to provide only the essential set of features for conducting the evaluation, thus shifting the evaluation focus onto the NIQ. For latency experiments, client and server are running on different processors, but on the same system, both using the same NetFPGA card with a copper twinax network cable providing a loopback link. Running on the same system, and thus in the same time domain, makes it possible to obtain detailed latency breakdowns. To avoid interference under high load, bandwidth experiments are run on two separate systems, with the client using a commodity network card, while the server still runs on NIQ.

NIQ is directly compared to an Intel x520 network card. To make the comparison fair, the x520 user-space driver was modified and integrated into the object store application so it can be used in the same way as the NIQ driver. For the reasons of implementation simplicity, physical addresses are made available to the application by using a kernel module that allocates 1 MB chunks of contiguous physical memory, which are further fragmented and managed by the application itself. Our application protocol runs on top of Ethernet and IP to correctly emulate a datacenter environment where a packet must travel through switches and IP routers. Most measurements are conducted on GET requests, except where otherwise indicated, because GETs are assumed to be an order of magnitude more frequent [3].

4.1. Latency evaluation

Here we present latency measurements on an unloaded system. In the unloaded system each request is sent in

		A	B	C	D	E	F	G	H	I	J	K
NIC	NIQ	•	•	•	•	•	•	•	•	•		•
	x520										•	
RX	small pkt	•	•	•	•		•	•	•	•		
	niq poll	•				•	•	•	•	•		
	host poll		•								•	•
	interrupt			•								
	mwait				•							
TX	small pkt	•	•	•	•	•	•		•	•		
	hdr split	•	•	•	•	•	•	•	•		•	•
	no DMA						•					
	doorbell								•		•	•

Table 1: Design configuration variants.

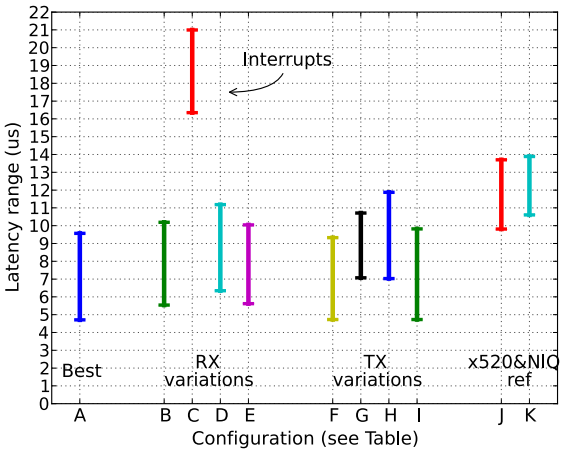

Figure 5: Design configurations' latency ranges for minimum to maximum size packets.

isolation from other requests in an effort to get consistent best case latency breakdown. Experiments with a loaded system are presented in the next subsection. All of the NIQ latency experiments are conducted with both client and server using the NIQ interface. Reference x520 experiments are done with client and server using the Intel x520 network card.

Our NIQ prototype can easily be configured to operate in different modes (e.g. polling, interrupts, header splitting on/off, small packet optimization on/off). We present latency performance of configurations listed in Table 1. For each configuration in Table 1, a matching latency range can be found in Figure 5. The bottom end of each latency range corresponds to a GET request round trip of a small object (4 bytes), while the upper end of the latency range corresponds to a large object (1452 bytes). Configuration A is our best NIQ configuration, as described in Section 3. Reference configurations J and K correspond to the Intel NIC design described in Section 2, with the exception that they use host memory polling instead of interrupts. In configuration K, NIQ prototype is configured to behave the same as the x520 NIC (configuration J) and they both exhibit similar latencies. This validates our choice of comparing our design with the x520 card, since for the same configuration NIQ and x520 perform similarly.

Configurations B through I are all unique and differ from configuration A in only one parameter, enabling us to quantify the benefits of each configuration parameter. In Figure 5 we demonstrate that our NIQ polling technique has the lowest latency, especially compared to an interrupt scheme (configuration C) that exhibits the highest latency. The same Figure 5 shows latency effects of small packet optimizations, as well as the effect of using a doorbell register scheme to initiate a descriptor DMA transfer. Even though, for large requests only, configuration F has a marginal latency advantage over configuration A, we dub configuration A as best. This is because configuration F uses the CPU to transfer data to the NIQ, instead of utilizing the DMA engine, which incurs a CPU bandwidth hit of over 3.5x for large requests. Header splitting provides small latency gains (up to 0.24 us) and it enables zero copy transmit, thereby improving bandwidth by over 20% for the largest requests.

To obtain further insight into the latency breakdown, we instrumented the userspace driver code to timestamp the request-response pairs at various points using the *rdtsc* instruction. For the timestamps to be accurate, we place every *rdtsc* instruction in-between two memory barriers, thus causing a total instrumentation overhead of 0.35 us compared to times presented in Figure 5. We are able to get one way request time measurements because the server and the client run on the same physical machine, but on different processors.

Figure 6 shows latency breakdowns for GET request (subplot a) and SET request (subplot b) running on NIQ. GET and SET breakdowns are similar, with a difference that the server returns a SET reply before touching any object data, leading to a constant server application latency with respect to object size. Replies containing objects that are 12 bytes or smaller, fit into a minimum size packet (after accounting for the 48 byte header), thus allowing the server to use the small packet transmit path and achieve lower latency (shown in the inset of Figure 6a).

While NIQ is optimized for small packets, it also performs well with large packets. A comparison between Figures 6a and 6c illustrates that NIQ and the x520 card latencies scale similarly with object size, but NIQ consistently provides lower latency. Figure 6d illustrates why we choose to transfer large packets using the DMA engine (subplot a), rather than stream writes using the CPU (subplot d). In subplot d, the server driver time dramatically increases with packet size, reducing the server bandwidth by up to 3.5 times. However, streaming CPU

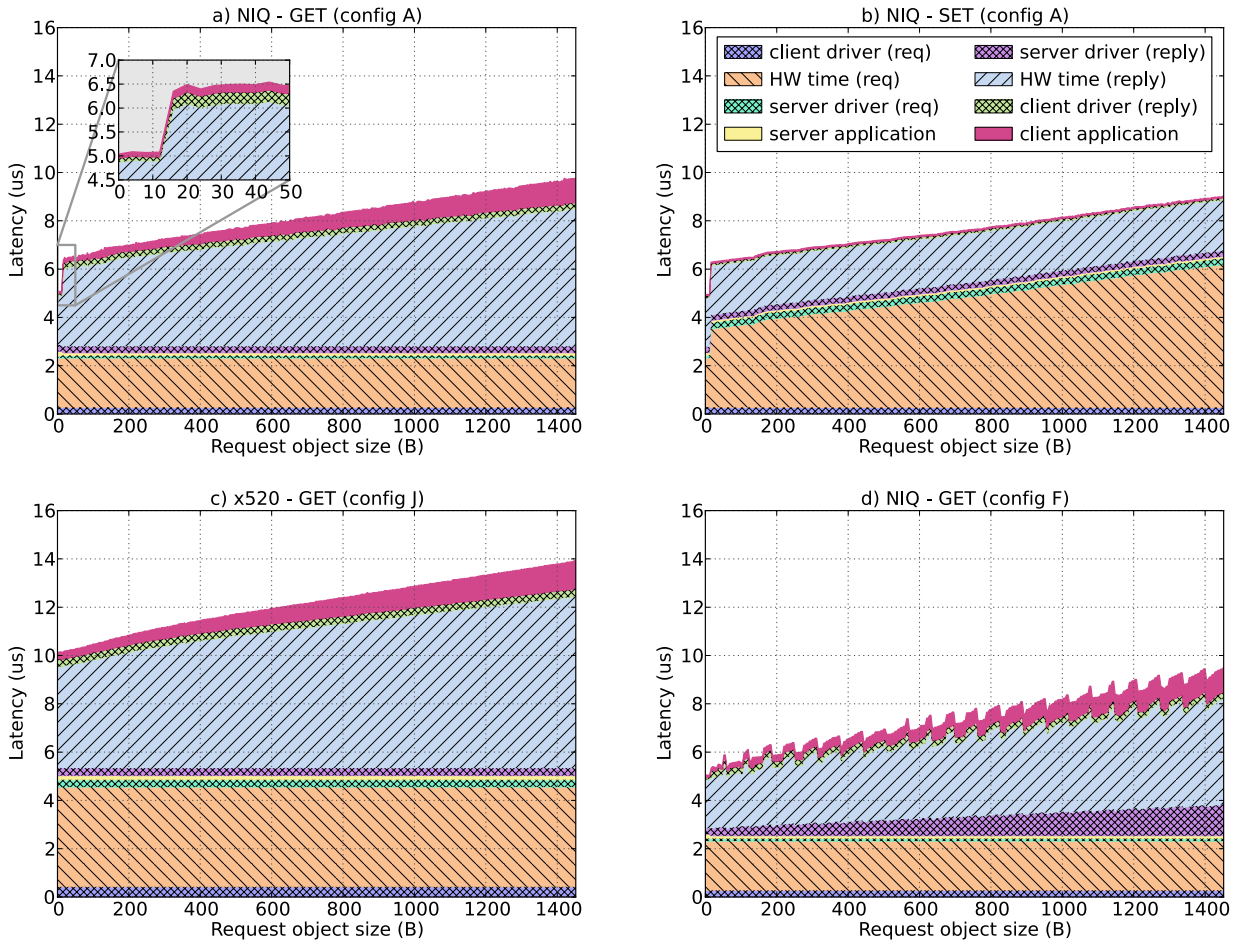

Figure 6: Detailed request latency breakdown.

writes might be a practical solution for transmitting packets that are just over the minimum size, where latency benefit is large and bandwidth penalty low. The saw-like artifacts in subplot d are very specific to our implementation and are the result of the host running out of PCIe credits when the object size is not an integer number of cache lines. Writing a partial cache line is implemented using multiple 8-byte writes, thus requiring more PCIe credits.

4.2. Bandwidth evaluation

Another important performance aspect is how NIQ behaves under heavy load, which we explore next. To generate enough load to saturate the server that is running the NIQ based single-threaded object store application, we use a multithreaded client running on top of the x520 card. This way we are able to make sure the bottleneck is the server running NIQ, therefore measuring NIQ bandwidth and NIQ latency under heavy load.

Figure 7 shows the server throughput on the outgoing link, which is also the bottleneck link since it carries larger packets than the inbound link. For large packets the throughput is limited by the physical link speed of 10 Gbps, but small and medium packet throughput is limited by the server processing power. There is an initial NIQ throughput drop in Figure 7 that occurs when the request object size is over 12 bytes. The drop is caused by the necessary descriptor assembly and additional bookkeeping required by objects that don't fit in small packets. As object size increases further, physical link bandwidth limitation causes a drop in throughput inversely proportional to the object size, for both NIQ and x520. Medium size packets' throughput is roughly constant with variations that are within one cache miss, and thus difficult to account for.

Interestingly, NIQ manifests higher throughput than the x520 card implementation. This observation implies that even though engineering for high bandwidth often compromises latency, the converse is not true; engineering for low latency is very favorable to achieving high bandwidth.

To further explore latency impact of high request loads, we plot the latency vs. load graph in Figure 8. For load generation, we model request arrivals as a Poisson

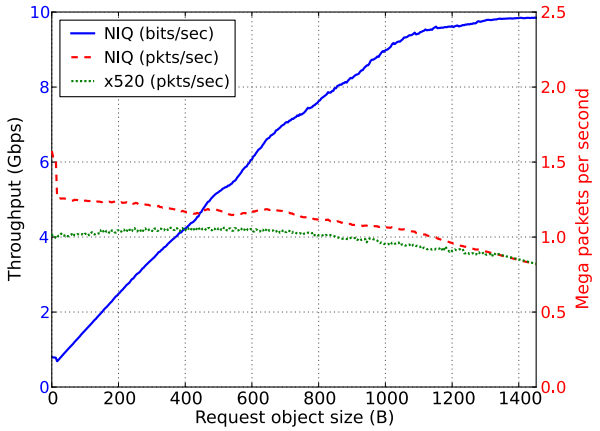

Figure 7: Server throughput results.

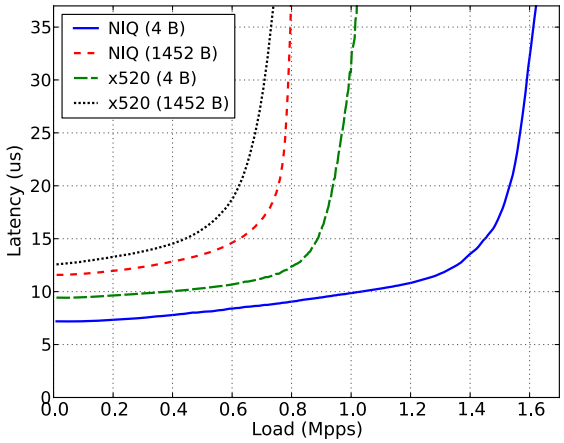

Figure 8: Server latency vs. load.

process by making the time between generated requests follow an exponential distribution. Figure 8 shows load-latency curves for minimum and maximum size objects, for both NIQ and x520 implementations. As predicted by the queueing theory, latency exponentially increases with load, and it does so in a similar fashion for all experimental setups.

5. Latency vs. Power Analysis

In this section we investigate latency and power impacts of processor power management. Our experiments have demonstrated that it is critical to properly use power management states to achieve low latency. We focus on a subset of Intel's power management states [10] that are relevant to our application, namely core idle states (c-states), package idle states (pc-states) and performance states (p-states). These states are Intel's implementation of platform independent mechanisms defined in the ACPI specification [8].

Core and package idle states lower the CPU power consumption during the idle periods at a cost of incurring

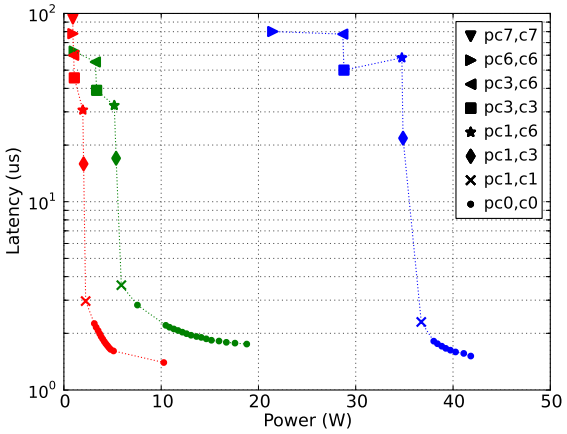

Figure 9: CPU latency-power tradeoffs for: mobile class Ivy Bridge (left/red), desktop class Sandy Bridge (middle/green) and server class Westmere (right/blue).

some wakeup latency when the CPU transitions back from idle to active. Package idle states control the power of the "uncore" logic (e.g. memory controller, shared caches, QPI links), while core idle states manage the cores (c0-active; c1-clock gated; c3-local cache flushed; c6-power gated). Wakeup latencies are on the order of 50 us for the deepest c-states, and up to 100 us for deep pc-states. Processor performance states (p-states) are active states tied to different processor clock frequencies that result from frequency-voltage scaling. Lower frequencies burn less power and incur higher execution time, but they do not add to wakeup latency because the processor can execute instructions in any p-state.

5.1. Experimental evaluation

We experimentally determine wakeup latencies and power consumption for three classes of Intel processors (mobile Ivy Bridge 3610QM, desktop Sandy Bridge 2600K and server Westmere E5620). Power is measured on the 12 volt rail that feeds into the CPU voltage regulators and thus includes any inefficiencies those regulators might introduce (typical regulator efficiency is around 85%). Wakeup latencies are measured from an FPGA board connected to a PCIe slot to avoid any instrumentation code interfering with the measurements. The FPGA generates a ping request to an idle CPU (in low power state) and measures how long it takes to receive a reply. Overall latency displayed in Figure 9, thus, includes one wakeup latency plus some small code execution time and one PCIe round trip overhead (overall overhead is less than 1 us). In this experiment the FPGA generates one ping request every 5 ms, allowing the processor plenty of time to completely reach any idle state. Each point in Figure 9 represents power and wakeup latency for a given (*package pc-state, core c-state*) pair. Power value is the processor idle power in the given idle-state pair,

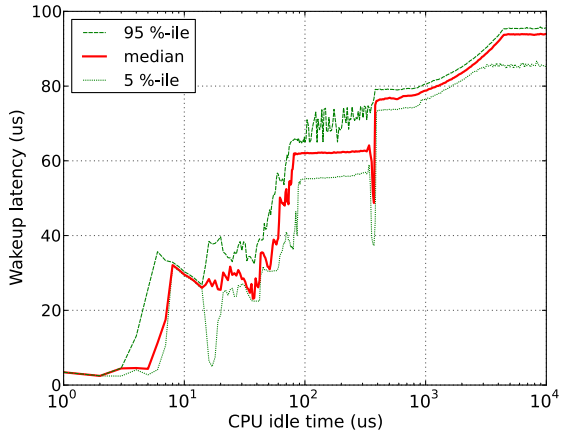

Figure 10: Wakeup latency for a mobile Ivy Bridge processor in transition from active to PC7 state.

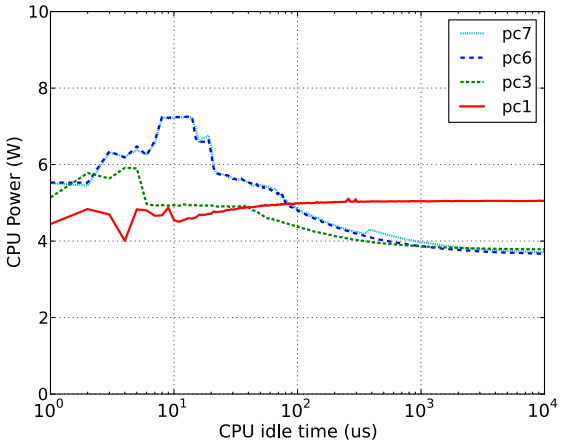

Figure 11: Average CPU power of a mobile Ivy Bridge processor in transition from active to idle state, but being interrupted after *CPU idle time*.

while wakeup latency is the time it takes the processor to exit its idle state and be ready to execute instructions. Multiple active-state points (pc0, c0) correspond to different performance states (i.e. different clock frequencies). Active states have zero wakeup latency, so (pc0, c0) points in Figure 9 show latency measurement overheads and corresponding power consumptions.

It can take several milliseconds for a processor with complex power management (e.g. mobile Ivy Bridge) to go from being active to being steadily in the deepest idle state. Since it can take milliseconds to reach a stable state, we must explore the wakeup latency and power of a processor that is in transition form active to deep idle. In Figure 10, a mobile Ivy Bridge processor is in transition from active state to pc7 (power gated cores, memory controller shut off, shared cache flushed and disabled). The figure shows how the wakeup latency depends on the actual idle time of the processor, as the processor is interrupted after *CPU idle time* into the transition from active to pc7. One can notice several plateaus corresponding to major processor components being shut down, as well as the overall complexity of the low power transition process. It is evident that the wakeup latency is lower if the transition process is interrupted early (e.g. only 4 us wakeup if interrupted 5 us into the pc7 transition). However, the proposition of frequently waking the processor before it completely reaches its intended idle state can be costly in terms of power, as shown in Figure 11. This figure shows the average power of a mobile Ivy Bridge processor that is trying to reach an idle state (pc1, pc3, pc6, pc7), but is always interrupted after *CPU idle time*. One can see from the figure that deep idle states are more efficient only if the CPU is actually idle for longer periods of time. Power numbers presented in Figure 11 were obtained from the RAPL interface registers [12] and each experiment's *CPU idle time* in Figure 10 was randomized to avoid unwanted artifacts due to periodicity.

Effects shown in Figure 10 can also affect DMA latencies on processors with integrated memory controllers. The memory controllers are put into an idle state by the processor, but a DMA read requires them to access main memory. Any delay of over 10 us between the processor starting a transition into pc6 and the DMA engine initiating a read is likely to increase DMA latency by over 10x.

5.2. Application notification performance

Upon new packet reception, the NIC generates a notification to the application that a new packet is available. This notification can be an interrupt, a write to a predefined memory location that is actively polled by the CPU, or a combination of both. The notification mechanism is at the center of power-latency tradeoffs discussed here. To achieve lower power consumption while waiting for a notification, the CPU can enter an idle state. However, if the CPU running the application is idle when the notification is received, wakeup latency is incurred, as discussed previously.

In Figure 12 we present latency and idle power measurements for interrupt and polling notifications on a desktop class system. We consider a case of using interrupts and placing the entire application code inside the kernel interrupt handler. However, this is feasible only for the simplest of cases where the application processing can be done extremely quickly. A more plausible setup is the one where the kernel interrupt handler wakes up a user process that does the bulk of the processing. We find that signaling from the interrupt handler to the user process introduces more than several microseconds of additional latency, even though both interrupt based mechanisms are equally power efficient.

The lowest latency is achieved using *mwait*/polling from inside the user space process. With this mecha-

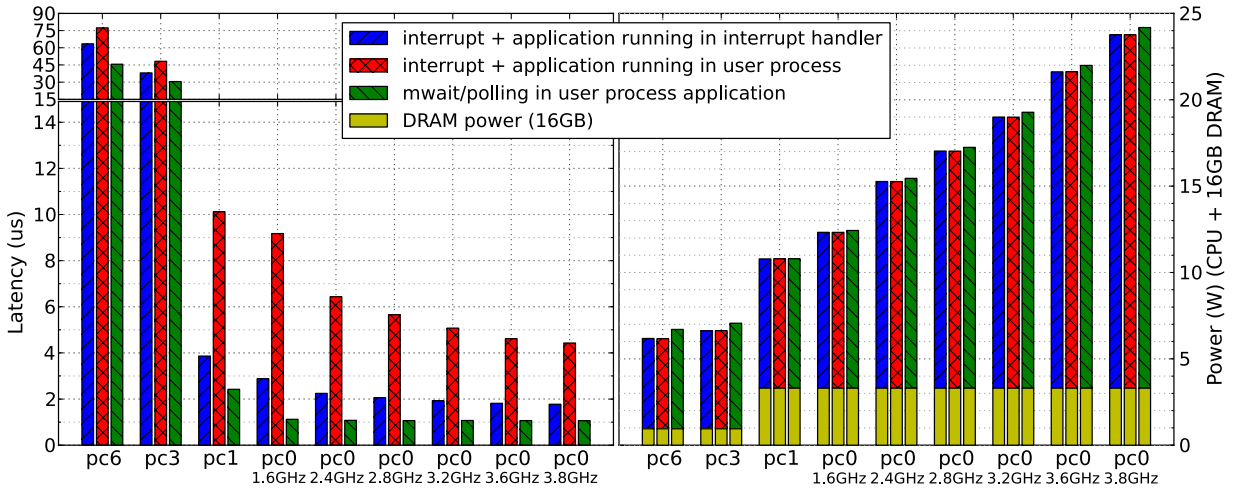

Figure 12: Latency vs. power tradeoffs of different notification mechanisms.

nism, in non-active states (pc6, pc3, pc1 in Figure 12) the user application makes calls to a kernel module to use *monitor/mwait* on a memory address that the NIQ writes to. In the active states (pc0 in Figure 12) the application polls that same memory address in a tight loop. For non-deep idle states (pc0, pc1), the *mwait*/polling approach increases power consumption by less than 2% over interrupt mechanisms, while offering more than 2-4x latency improvements. Deeper idle states (pc6, pc3) offer over 50% reduction in combined CPU and memory power over non-deep idle states (pc0, pc1), but they are unusable due to a huge latency penalty.

One disadvantage of the *mwait* approach is that the application process appears to the operating system as if it is constantly running without voluntarily yielding the CPU. This makes it impossible to take full advantage of the tickless kernel [27], resulting in marginally higher power usage in deep idle states, as shown in Figure 12 for states pc3 and pc6.

In conclusion, *mwait*/polling mechanism combined with the pc1 idle state is a viable way to save power (55% savings compared to highest performance states). However, deeper idle states incur too high of a latency penalty to be used with ultra low latency applications. While there is a power penalty to using *mwait*/polling instead of interrupts, it is not as dramatic as initially expected. We believe that the poor latency reputation of interrupts and the high power consumption reputation of polling are somewhat unjustified. While waiting for an interrupt, processors are generally allowed to enter deep idle states (e.g. pc6), but polling always keeps them busy in an active state (pc0). This means pc6-state power and latency are associated with interrupts, while pc0-state power and latency are associated with polling. However, one can select the CPU idle state for either interrupts or polling, and Figure 12 is meant to help with that choice.

6. Related Work

Engineers have been building low latency RPC systems since long before Ethernet was a dominating link layer protocol, as demonstrated in [32]. In that paper authors identify network controllers as having a significant influence on the overall RPC latency. They also express concern with the trend of NIC latencies not improving nearly as fast as their bandwidth capabilities. These trends have continued with NIC bandwidth increasing 1000-fold since then, but round trip RPC latencies improved only 10-100 times. Ironically, most of the latency improvement came from the decrease in serialization latency, which is directly tied to bandwidth (e.g. 64 byte packet at 10 Mbps has serialization latency of 51.2 us, while at 10 Gbps it is only 51.2 ns). The same authors [32] compared DMA based Ethernet controllers with FIFO-based ATM controllers. They concluded that DMA based controllers work well for large packets, but for small packets they prefer a simple FIFO interface that is directly accessed by the host. Similarly to our approach, they go on to advocate building future network controllers with a hybrid interface to best suit both small and large packets. Twenty years later, with computing shifting into datacenters, we find that many of the good ideas from the past are being adapted to fit current needs.

Even though Infiniband has traditionally been favored in high performance computing, recent work has shown that 10G Ethernet can also achieve good latency performance [24]. To complement this finding, high performance switch and NIC vendors have built 10G Ethernet switches capable of 0.3-0.5 microsecond latencies [7, 2] and network adapters capable of 1.3 microsecond latencies [20]. These proprietary components have promptly found their way into vertically integrated solutions for low latency applications, such as online trading systems [30]. We view our work as an integral part of the over-

all efforts to openly discuss, understand and build low latency datacenter systems on commodity hardware.

Researching the interactions between the cache hierarchy and I/O data transfers [31, 9, 13] has resulted in an implementation by Intel that claims to improve their network controller latency by 10-15% [13]. Previous work on power management of datacenter servers [18, 19] has been promising, but it assumes that latencies in hundreds of microseconds are acceptable. On the software research front, there are efforts that improve the average and tail latencies of existing datacenter applications, such as memcached and web search [15]. With a clean slate approach, the RAMCloud project [22] is driving the total round trip request-response latency into the 5-10 us range, applying significant efforts to reducing the software overheads. Kernel network stack overheads are typically removed by circumventing the network stack completely and accessing the network card through a user-level driver [17, 5, 28]. User-level drivers are typically hardware specific and do not provide the OS protection mechanisms that applications are accustomed to. There is ongoing work on the netmap framework [25] that provides low latency of a user level access in conjunction with the benefits of an operating system. This software research complements our work by accentuating the need for low latency network interfaces.

7. Discussion and Conclusion

Our approach in this paper is a clean slate approach where we assume the applications are written in a way to take advantage of NIQ. For an existing application to benefit from NIQ, it would need to be modified to use the NIQ user space library to send and receive packets. It is also possible to implement the NIQ driver as a kernel module, thus enabling application multiplexing, but we do not explore this option in the paper. Some of the software low latency techniques used in the NIQ implementation can also be used with most other network cards (e.g. polling, kernel bypass, header splitting). To make the evaluation comparison fair, we have implemented those techniques for both NIQ and the Intel x520 card. On the other hand, some of the techniques are unique to NIQ and require support form the driver, hardware and the host system (e.g. delayed PCIe reads, mapping NIQ memory as cacheable).

NIQ performance results presented in Section 4 are limited by the FPGA implementation. We implemented NIQ prototype on an FPGA development board utilizing available standard components, such as MAC, PHY and the PCIe core. However, those available components exhibit relatively high latencies, when compared to the lowest possible with today's technologies. We are able to extrapolate what the overall request-response latency would be, if the system is built with state of the art components. One small packet round trip through our PHY and MAC components measures at 920 ns, while the best ASIC components take only 300 ns, indicating 620 ns of possible improvement in the Ethernet path. For the PCIe estimation we take the minimum PCIe latency seen on the Intel's x520 card (560 ns) and compare it to the minimum NIQ PCIe latency (930 ns). Additionally, we measure an extra 190 ns of possible savings when running on one of the newest available server processors with an on-chip PCIe controller, for a total PCIe round trip savings of 560 ns. Since our minimum request-response latency (4.65 us) includes two Ethernet and two PCIe round trips, we infer possible request-response times of under 2.3 us, thereby doubling our performance.

We demonstrate that it is possible to fit a basic GET request within a minimum size (60 B) packet using binary object keys. Many existing object store systems use string keys instead of binary object keys, which generally makes GET requests bigger than a single cache line. One solution to this would be to use a hash on the string object keys to convert them into binary keys. However, it is also possible to extend the NIQ small packet interface to support larger packets (e.g. two cache line size packets) on receive and transmit. On the transmit side, writing a two-cache-line packet using the write-gathering memory mapping is as simple as writing two cache lines back-to-back. On the receive side, however, the change would be more extensive. One simple solution to support larger packet on the small packet interface is not to communicate complete headers between NIQ and the CPU. Instead, NIQ checks and drops redundant header fields on receive (e.g. destination mac address, type fields, etc.), thus enabling bigger Ethernet packets that still fit into one cache line. Another possible solution is to use two sibling hardware threads (hyperthreads) to issue two polling cache misses, with replies eventually ending up in the same L1 cache ready for processing. A similar solution might involve explicit prefetches (using the SSE instruction), or even automatic adjacent line prefetching, to get more than one cache line at a time from NIQ to the CPU. Finally, one might simply use the host polling technique; especially on a new SandyBridge-EP platform with DDIO [13], where it would perform well.

In its current prototype form, NIQ interface can only be accessed by a single thread at a time, thus limiting possibilities for parallelism. As future work we intend to extend the NIQ interface to support multiple queues, thus supporting multithreading and increasing performance through parallelism.

We also discuss two extreme solutions for building a low latency request-response applications (e.g. object store). One solution is to implement the entire application in the NIQ. This becomes problematic as soon as two or more memory accesses are needed, such as a hash

table lookup and a data read, because memory accesses over PCIe exhibit high latency. The second solution is to completely integrate the network card with the processor. This is an attractive option from the latency point of view, with an open question of how exactly would the integrated controller be connected to the processor. Unless the latency of that interconnect is negligible, such design would still benefit from our findings.

In conclusion, we have designed, implemented and evaluated a network interface solution for low latency request-response protocols. We demonstrate significant latency gains by focusing on minimizing the number of transitions over the PCIe interconnect, particularly for small packets. Moreover, we designed and made a case for a custom polling notification technique by evaluating its latency and power performance. We also investigated processor power management implementation and the impact it has on latency. Finally, our system's latency gains did not come at the expense of bandwidth, but there was an increase in implementation complexity.

References

[1] ALIZADEH, M., KABBANI, A., EDSALL, T., PRABHAKAR, B., VAHDAT, A., AND M., Y. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of USENIX NSDI conference* (2012).

[2] ARISTA NETWORKS. 7100 Series Switches: `http://www.aristanetworks.com/en/products/7100series`, July 2012.

[3] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. *SIGMETRICS Perform. Eval. Rev. 40*, 1 (June 2012), 53–64.

[4] BROADCOM CORPORATION. Broadcom NetXtreme® 57XX User Guide, April 2012.

[5] DERI, L. ncap: wire-speed packet capture and transmission. In *End-to-End Monitoring Techniques and Services, 2005. Workshop on* (may 2005), pp. 47 – 55.

[6] FITZPATRICK, B. Distributed caching with memcached. *Linux J. 2004*, 124 (Aug. 2004).

[7] FULCRUM MICROSYSTEMS. FM4000 Switches: `http://www.fulcrummicro.com/documents/products/FM4000_Product_Brief.pdf`, July 2012.

[8] HP, INTEL, MICROSOFT, PHOENIX, TOSHIBA. ACPI: Advanced Configuration and Power Interface Specification, December 2011.

[9] HUGGAHALLI, R., IYER, R., AND TETRICK, S. Direct cache access for high bandwidth network I/O. In *Proceedings of the 32nd annual international symposium on Computer Architecture* (Washington, DC, USA, 2005), ISCA '05, IEEE Computer Society, pp. 50–59.

[10] INTEL CORPORATION. Intel and Core i7 (Nehalem) Dynamic Power Management, 2008.

[11] INTEL CORPORATION. ixgbe - Intel 10 Gigabit PCI Express Linux driver, 2010.

[12] INTEL CORPORATION. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3, April 2011.

[13] INTEL CORPORATION. Intel® Data Direct I/O Technology (Intel® DDIO): A Primer, February 2012.

[14] INTEL CORPORATION. Intel® Ethernet Converged Network Adapter X520, 2012.

[15] KAPOOR, R., PORTER, G., TEWARI, M., VOELKER, G. M., AND VAHDAT, A. Chronos: predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing* (New York, NY, USA, 2012), SoCC '12, ACM, pp. 9:1–9:14.

[16] KIM, J., DALLY, W. J., SCOTT, S., AND ABTS, D. Technology-driven, highly-scalable dragonfly topology. In *Proceedings of the 35th Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2008), ISCA '08, IEEE Computer Society, pp. 77–88.

[17] KRASNYANSKY, M. uio-ixgbe: `https://opensource.qualcomm.com/wiki/UIO-IXGBE`, July 2012.

[18] MEISNER, D., GOLD, B. T., AND WENISCH, T. F. Powernap: eliminating server idle power. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2009), ASPLOS XIV, ACM, pp. 205–216.

[19] MEISNER, D., SADLER, C. M., BARROSO, L. A., WEBER, W.-D., AND WENISCH, T. F. Power management of online data-intensive services. In *Proceedings of the 38th annual international symposium on Computer architecture* (New York, NY, USA, 2011), ISCA '11, ACM, pp. 319–330.

[20] MELLANOX TECHNOLOGIES. ConnectX-2EN NIC: `http://www.mellanox.com/related-docs/prod_software/ConnectX-2_RDMA_RoCE.pdf`, July 2012.

[21] MOGUL, J. C., AND RAMAKRISHNAN, K. K. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst. 15*, 3 (Aug. 1997), 217–252.

[22] OUSTERHOUT, J., ET AL. The case for RAMCloud. *Commun. ACM 54*, 7 (July 2011), 121–130.

[23] PCI SPECIAL INTEREST GROUP. *PCI Express Base Specification Revision 1.1*. PCI-SIG, 2005.

[24] RASHTI, M., AND AFSAHI, A. 10-gigabit iwarp ethernet: Comparative performance analysis with infiniband and myrinet-10g. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International* (Mar 2007), pp. 1 –8.

[25] RIZZO, L. Revisiting network I/O APIs: the netmap framework. *Commun. ACM 55*, 3 (Mar. 2012), 45–51.

[26] SALAH, K. To coalesce or not to coalesce. *AEU - International Journal of Electronics and Communications 61*, 4 (2007), 215 – 225.

[27] SIDDHA, S., PALLIPADI, V., AND VEN, A. Getting maximum mileage out of tickless. In *Linux Symposium* (2007), vol. 2, Citeseer, pp. 201–207.

[28] SOLARFLARE COMMUNICATIONS. OpenOnload: `http://www.openonload.org`, July 2012.

[29] STANFORD NETFPGA GROUP. NetFPGA 10G, `http://netfpga.org`, July 2012.

[30] SUBRAMONI, H., PETRINI, F., AGARWAL, V., AND PASETTO, D. Streaming, low-latency communication in on-line trading systems. *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops* (2010), 1–8.

[31] TANG, D., BAO, Y., HU, W., AND CHEN, M. DMA cache: Using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance. In *2010 IEEE 16th HPCA* (Jan 2010), pp. 1 –12.

[32] THEKKATH, C. A., AND LEVY, H. M. Limits to low-latency communication on high-speed networks. *ACM Trans. Comput. Syst. 11*, 2 (May 1993), 179–203.