Characterization of Direct Cache Access on Multi-core Systems and 10GbE

Amit Kumar Ram Huggahalli Srihari Makineni
Intel Corporation
{amit.kumar, ram.huggahalli, srihari.makineni}@intel.com

Abstract

10GbE connectivity is expected to be a standard feature of server platforms in the near future. Among the numerous methods and features proposed to improve network performance of such platforms is Direct Cache Access (DCA) to route incoming I/O to CPU caches directly. While this feature has been shown to be promising, there can be significant challenges when dealing with high rates of traffic in a multiprocessor and multi-core environment. In this paper, we focus on two practical considerations with DCA. In the first case, we show that the performance benefit from DCA can be limited when network traffic processing rate cannot match the I/O rate. In the second case, we show that affinitizing both stack and application contexts to cores that share a cache is critical. With proper distribution and affinity, we show that a standard Linux network stack runs 32% faster for 2KB to 64KB I/O sizes.

1. Introduction

Improvements in computational capability attained by Moore's law and microprocessor architectural innovations are over time balanced by corresponding improvements in I/O capability (Figure 1). In general purpose systems, innovations to balance compute and I/O have largely been in the form of platform technologies such as PCI Express® or HyperTransport™ extending to higher rate external interconnects such as Ethernet. When the raw I/O capability is associated with computation as in network protocol processing, platform technologies must be complemented by software architectural and micro-architectural innovations to ensure that the capability is efficiently realized.

The introduction of 10GbE and PCI Express® shifted the focus of platform optimizations towards inefficiencies in both network stack software as well as in the ability of modern processors to handle network I/O related data movement overheads [1,2,3,4]. The advent of Intel® Core™ micro-architecture [5] and improvements such as Transmit

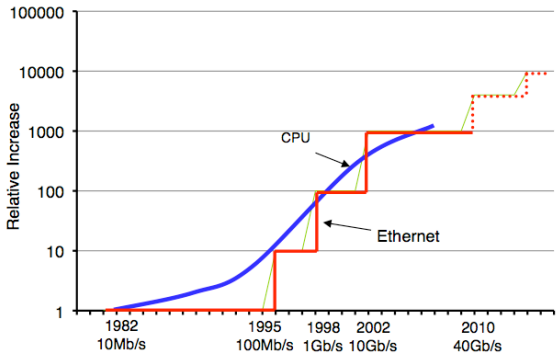

Figure 1. Relative increase in CPU and Network capability

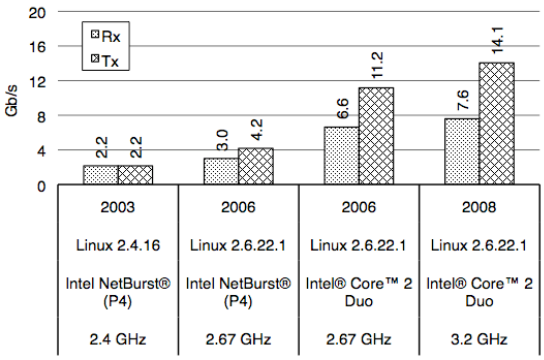

Figure 2. Normalized TCP/IP throughput of a Single Core

Segmentation Offload (TSO) and Large Receive Offload (LRO), have resulted in significant net improvements as shown in Figure 2. However, what increasingly stands out is the memory dependency of the processor when handling packet data. This is particularly the case in Receive-side network protocol processing where the overhead of copying has long been observed as a significant problem [4, 7].

978-1-4244-2932-5/08/$25.00 ©2008 IEEE

One of the recent proposals to address the memory dependency of packet processing has been the concept of Direct Cache Access (DCA) [6, 7]. DCA is a coherence protocol optimization that delivers inbound data from a network interface controller (NIC) directly into processor caches dramatically reducing stalls due to memory access of descriptor, packet header and packet payload data structures. The biggest impact of DCA was earlier demonstrated to be in the reduction of time to copy data from a kernel buffer used by the NIC for DMA into an application buffer [6, 7].

I/O related micro-architectural enhancements have been limited to a few areas such as Message Signaled Interrupts (MSI) [9] and faster access to uncacheable memory mapped regions [10,11]. We believe that the reason for limited innovations is not because of the lack of opportunity but due to the highly challenging complexity of relating micro-architectural changes to system level improvements. Observation of opportunities in existing platforms need to be followed up by realistic full system simulations, FPGA prototypes or silicon prototypes to confirm that any idea really produces a system level benefit. In the case of DCA, previous results from real platform measurements have indicated that DCA produces tangible system level benefits. These promising results are suggestive of how microprocessor architecture can be adapted to future I/O subsystem needs to achieve new levels of platform or system efficiency. However, network protocol processing is a challenging area where all major platform components (processors, memory, chipset, I/O controllers) intersect the system software components (drivers, kernel, network stack and application). Therefore additional layers of complexity exist when we modify our view of the system with higher rates of traffic, assume NUMA configurations, comprehend various levels of component integration, different OS scheduling policies and so on. In this paper, we stretch the platform with 10GbE NIC and exercise multi-core aspects of the platform.

The key contributions of this paper are as follows:

- Measured performance results with DCA at 10Gb/s on quad-core, dual processor platforms
- A detailed characterization of the conditions under which data 'overflows' the cache due to a fundamental mismatch between the flow control policy of the end-to-end network protocol and typical cache sizes
- A review of the coordination required between interrupt distribution and application task scheduling to realize DCA's benefit in a multi-core system
- Preliminary directions in reducing shared cache pollution effects, NUMA and Non-Uniform I/O Access platform configurations

The remainder of this paper is organized as follows. Section 2 reviews prior work in both measurement results as well as architectural direction. In Section 3, we present TCP/IP performance at 10Gb/s for one isolated core with a few software and hardware features. This is followed by data and analysis of DCA's performance and issues with single core and multi-core in the next two sections. In Section 6, architectural issues beyond cache injection and options for future architectural enhancements are reviewed. We conclude with a summary in Section 7.

2. Prior Work

The benefit of directing memory traffic from a NIC directly into caches comes from reduction in memory bandwidth and reducing data access latency from the processor's perspective. In the context of modern general purpose server platforms, the concept of DCA was introduced using simulation based characterization of network traffic by Huggahalli et al [6]. In their work, network traffic generators as well as realistic benchmarks such as SPECWeb99 were used to show that a tight temporal and spatial relationship exists between inbound network traffic and its processing on the CPU. This initial work was followed-up by measurements on a real system using a prototype implementation of DCA by Kumar [7]. The prototype was used to (1) show system level sensitivities across different I/O sizes (2) compare DCA with traditional hardware prefetching and (3) demonstrate the benefit of DCA in the presence of concurrently running realistic applications. The primary result of this paper was that a relatively simple implementation of DCA results in 15 to 43% speed-up on standard operating system network stacks in a real server system. This speed-up reduced the CPU utilization due to network processing allowing concurrently running applications to proportionally benefit from the additional CPU utilization.

In [8], the authors have analyzed cache injection performance benefits with a HPC application using simulators and proposed that the injection policy must have a feedback loop. They observed that OS, compiler and/or application need to help determine which cache to inject and furthermore, the application must be able to consume the data in time to benefit from it. Unless these two requirements are met, cache injection is likely to cause performance degradation.

In [7], the authors also described a Prefetch Hint based protocol to implement Direct Cache Access. We use the same protocol in the experimental setup

described in Section 3. The Prefetch Hint protocol is a hardware complexity trade-off that leverages existing system bus protocols. In this approach, for an incoming packet, as the NIC does DMA transfers to memory via the memory controller, snoops corresponding to each cache line are tagged with hints to the CPU. The hints are intended to pass the snoop address to a prefetching hardware structure in the CPU. The NIC is preprogrammed by the driver to direct hints to CPU that it is interrupting. If the interrupt affinity is changed during execution, the driver updates the hints target as well. Once hints are received by the CPU, it queues them into a prefetch queue and after a certain delay, launches the read requests. This delay is to ensure that data is not requested before it's available in the coherent domain and thus it does not need to be changed once a correct delay is determined for the platform. Since data is read into CPU caches prior to execution of interrupt handler, we effectively achieve cache injection with prefetch-hint protocol. This approach does not reduce memory bandwidth and focuses on processor to memory latency. This paper builds on prior work by examining the interaction between system software and platform data movement through the network and the cache hierarchy.

3. 10GbE and Single core performance

TCP/IP performance is a function of a large number of system parameters. The two common metrics used to characterize TCP/IP performance are throughput and CPU utilization. By normalizing throughput to the same CPU utilization, the throughput capability for a unit of computation can be gauged. In all of the tests in this section, only a single core was used by the benchmark and other cores in the system were kept idle. If the single core was not fully saturated, then the throughput reported in this section was scaled to match 100% utilization of the single core.

For all experiments discussed in this paper, we used the system under test (SUT) as described in Figure 3. Wherever configuration variations are used, like using one core vs. all eight cores, Hardware Prefetchers (HWP) on vs. off and LRO on vs. off, the differences are mentioned along with the analysis. The SUT contains two quad-core Intel® Xeon® processors with Intel® Core™ micro-architecture, making a total of 8 cores available to Operating System (OS). As shown in Figure 3, each pair of cores shares a 4MB L2 (last level) cache. For 10Gb/s Ethernet adapter, we have used Intel® 10Gb/s XF server adapter - a PCI Express® (PCIe®) based discrete NIC. The NIC has multiple receive and transmit queues and each queue is allocated a separate interrupt number. Multiple queues are essential for distribution of flows to cores. Since the analysis in this paper is primarily for receive I/Os and there are eight cores in the system, we enabled eight receive queues and one transmit queue.

For performance measurements, we used a modified version of the publicly available Iperf tool [12]. Iperf is a software tool to generate TCP and UDP packets on a Windows or Linux system. In experiments with 10Gb/s data rate, Iperf's worker threads (that transmit or receive) were throttled by a reporting thread. The reporter thread keeps track of traffic statistics like data rate. We modified Iperf's reporting structure and ensured that worker threads are always free running to achieve maximum performance. Additionally, we wanted to ensure that the throughput across multiple TCP connections in a test run was similar. This functionality was added to Iperf server by invoking *sched_yield()* from each transmit worker thread after transmitting a fixed quantum of data on one connection. The explicit yield ensured a balanced throughput across all connections, making Iperf independent of OS scheduling policy to ensure that. We also chose not to turn on the Nagle's algorithm [13] – an algorithm that opportunistically coalesces multiple I/O segments belonging to same TCP connection into a single packet. By turning this parameter off, small I/O (less than 1500 bytes) transfers are transmitted as small packets incurring all the overheads of the network stack without amortization.

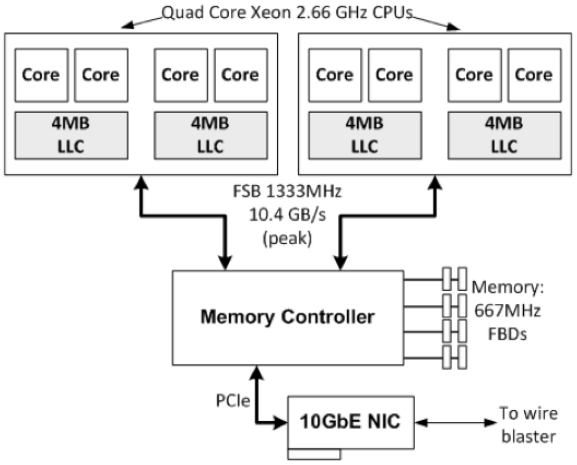

Figure 3. System Under Test

Figure 4 shows results for small I/Os in three different configurations and Figure 5 shows the same for large I/Os (equal to or more than 1500 bytes). The first bar in the graphs is referred to as "baseline" since it captured without LRO, HWP or DCA. The bar shows peak throughput achieved on a receive side test

with 8 TCP connections running simultaneously. Dynamic interrupt moderation is turned on but NAPI [14] is not. The middle bar in Figure 4 and 5 is with LRO enabled. In LRO, a driver tries to coalesce incoming packets on the same connection before passing it to protocol stack in the OS. This coalescing reduces header processing overheads resulting in a higher throughput than the baseline.

Lastly, the third bar from left in the graphs is with both LRO and HWP. Traditional hardware prefetchers employ a memory access stride based predictive algorithm. By prefetching data from memory ahead of time into CPU caches, HWP reduce effective memory access latency and improves performance as a result. However, since prefetches are speculative, they can stream in redundant data that pollutes cache and create greater contention for Front-Side Bus (FSB) and memory bandwidth. Cache pollution and bandwidth contention effects of HWP are often exposed only when multiple cores are engaged by the workload. In addition, HWP cannot hide the access latency of first few accesses as they are triggered by cache misses. Depending on a workload's memory access patterns and other applications sharing the same cache, HWP can improve or hurt the workload performance. Later in section 4 and 5 we show how DCA prefetches just the packet data and achieves better performance by avoiding the loading of redundant data into caches.

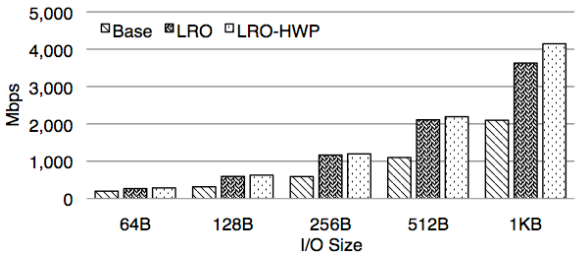

Figure 4. Small I/Os RX performance (1 core)

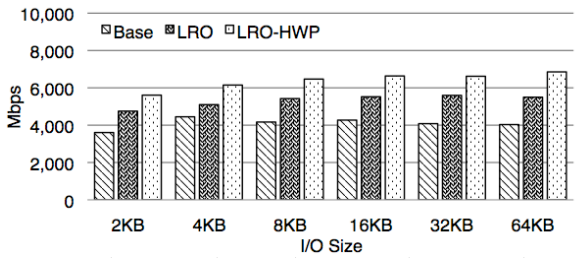

Figure 5. Large I/Os RX performance (1 core)

Average percent improvement of single core's network throughput with LRO over baseline for small I/Os is a significant 77% and for large I/Os it is 30%. The improvement is calculated as explained at the beginning of this section. For small I/Os, since the header processing overhead per byte of payload is very high, LRO significantly improves performance by reducing header processing overhead. HWP percent improvement over LRO is 7% for small I/Os and 20% for large I/Os.

4. 10GbE Single core performance with DCA

An overview of DCA implementation for the SUT was provided in Section 2 with details in [7] Rest of this section is focused on experiment analysis.

4.1. Measured performance and analysis

Since DCA hints are programmed by the driver to follow interrupt affinity to cores, it is important that the entire stack for the packet executes on a core that is interrupted or on a core that shares the L2 cache with the core that is interrupted. We will discuss how this is achieved when multiple cores are used for network processing in Section 5. To test single core performance, Iperf and all interrupt vectors from NIC were affinitized to Core 0 in the system. From Figures 4 and 5, we observe that a single core is not capable of receiving 10Gb/s data rate and thus performance is CPU limited.

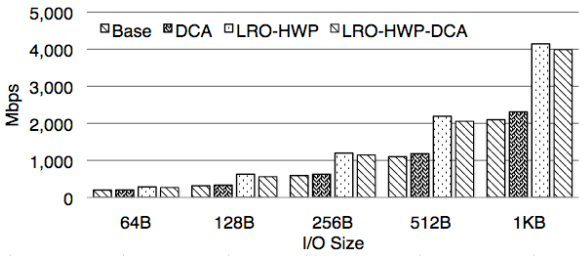

Figure 6. Small I/O single core receive w/ DCA

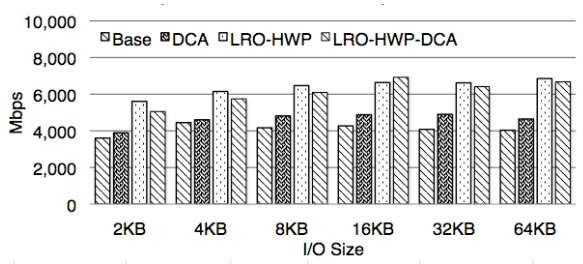

Figure 7. Large I/O single core receive w/ DCA

Figure 6 and 7 show small and large I/O performance with DCA. The first bar in the figures shows base performance without LRO or HWP and

second bar is performance with just DCA. The comparison of first two bars is important since it helps showcase DCA performance in isolation. Average improvement of DCA over baseline for small I/Os is 6% and for large I/Os is 13%. When LRO and HWP are enabled, small I/Os with DCA looses performance by 6% on average and by 2% for large I/Os. DCA performance from these experiments was contrary to our expectation since in [7], DCA showed from 15% improvement for small I/Os up to 43% improvement for large I/Os with two 1Gb/s ports. Relatively, DCA shows a smaller gain at 10Gb/s and actually hurts performance when combined with HWP. Lower performance implies higher time spent per packet. To understand performance difference with [7], we compare per packet execution profile between 1 and 10Gb/s.

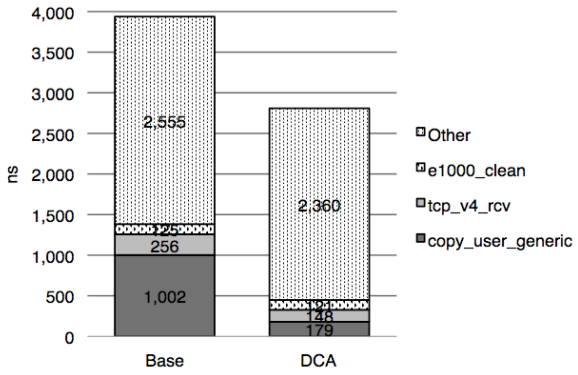

Figure 8. 4KB I/O per packet profile @1GbE

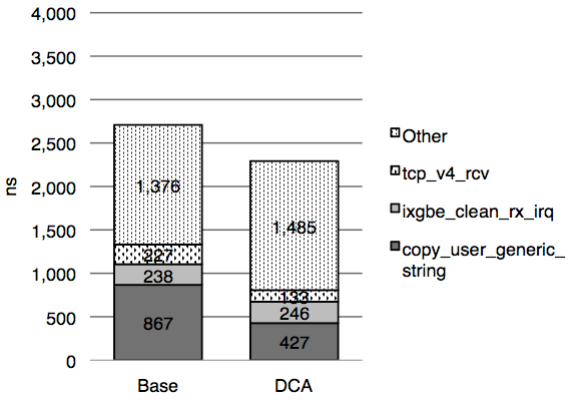

Figure 9. 4KB I/O per packet profile @10GbE

Figure 8 shows that DCA significantly reduced time for packet's payload copy from kernel to user buffer at 1Gb/s. For 10Gb/s however (in Figure 9), the copy time reduces only by half. Note that baseline has also improved a little compared to Figure 8 due to a copy optimization in newer kernel version. In a separate experiment, DCA functionality on SUT was verified using CPU performance counters, the packets prefetched into cache are same as received by NIC. This made us suspect that higher data rate packet injection is leading to packet evictions prior to copies. To verify this theory, we ran a few multi-gigabit experiments where the 10GbE port is connected to several 1Gb/s ports via an Ethernet switch. Throughput is increased in steps of 1Gb/s and results are shown in Figure 10.

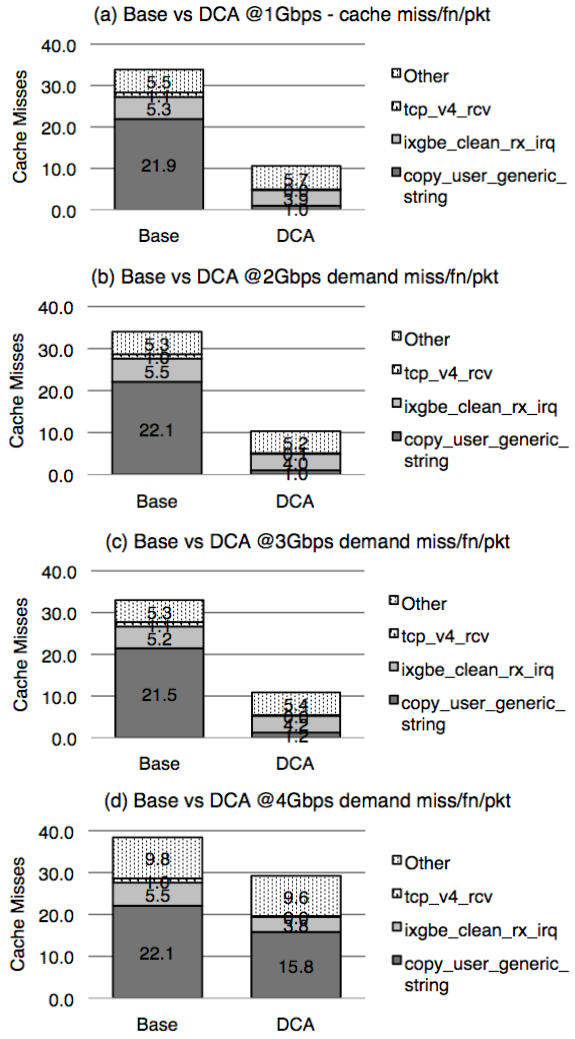

Figure 10. Per packet cache miss profile at 4KB I/O

Figure 10(a) plots average number of cache misses per packet at 1Gb/s collected using Oprofile (with `LLC_MISSES` performance counter) – a popular Linux profiling tool. The average measured packet payload size is 1387 bytes. Thus, an average of 21.7 cache lines need to be copied from kernel to user buffer for each packet. In other words, without prefetch the copy routine will miss cache on 21.7

lines. Oprofile for baseline in Figure 10(a) shows that copy routine misses on 21.9 cache lines. Note that the fractional portion of the number of cache lines does not indicate that partial cache lines are involved.

With DCA, all the cache misses for copy routine are ideally expected to be zero. We also expect one less cache miss for header and one for NIC descriptor. Since most performance benefit with DCA is derived from reduced copy overhead, we focus on cache misses for *copy_user_generic_string()* routine. Figure 10(a) shows that at 1Gb/s, DCA has only one cache line miss during copy. Thus, when DCA on 10Gb/s NIC receives just 1Gb/s data rate, for all practical purposes it does not show any inadvertent cache displacement. Since Figure 10 showed that DCA was not as effective on a back to back 10GbE test, we expect that as network throughput is increased from 1 to 2Gb/s, 2 to 3Gb/s and 3 to 4Gb/s, we would observe a progression of cache misses. However, Figure 10(b) and 10(c) show that the DCA is working effectively up to 3Gb/s but experiences a lot of misses in copy at 4Gb/s (Figure 10d). Note that CPU utilization at 4Gb/s is 100% but in all other cases, it was below 100%. This experiment demonstrates that cache misses with DCA are not due to the data rate but there is some relationship to CPU utilization. This can be better explained better by understanding life of a packet in Linux network stack.

4.2. Life of an incoming packet on Linux stack and the batching anomaly

When the NIC receives a packet, it transfers the packet into a predetermined kernel buffer in memory. NIC then issues an interrupt which is typically moderated by using only one interrupt for a batch of packets. An Interrupt Service Routine (ISR) does minimal packet processing but schedules a new task (softIRQ) to do that. On returning from the ISR, kernel checks for a pending softIRQ and invokes the corresponding callback function. The callback function is called *net_rx_action()* in case of receive packets. *net_rx_action()* in turn invokes IP and TCP processing functions and packet's TCP socket is determined. The packet is then linked to TCP socket's list of pending packets to be copied to application. Since the user buffer is not pinned in memory, copy can only be performed when application is scheduled. However, if there are more incoming packets, softIRQ will process their headers since it is higher priority. Thus softIRQ handler processes several packets before yielding to the application. This reduces context switching overhead after every packet and thus increases total number of packets processed by the CPU. Note that both ISR and softIRQ handler run at a higher priority than applications. Thus an application can execute only if they both complete processing or yield explicitly. The application is waiting on a blocking *recv()* system call which invokes kernel's *tcp_recvmsg()* that performs packet copies when the application is scheduled.

Since DCA effectively pushes incoming packets into a CPU cache prior to raising an interrupt, the stack is able to deliver the packet to application from cache while CPU is able to keep up with incoming traffic. When the CPU utilization is below 100% (Figure 10 (a), (b) and (c)), an ISR, softIRQ and kernel-to-user copy are able to executing one after the other without leaving a backlog of packets before the next interrupt. Thus, packets present in cache are copied before eviction. However, when CPU is at 100%, it is busy processing headers in interrupt and softIRQ context while copies are delayed since the application is a lower priority process. Eventually, as the sender does not receive acknowledgements, the TCP window closes and sender stops transmission. But since the sender was able to transmit data that fits in TCP window for all connections, with DCA all these packets are sent to the receive-side CPU's cache. Due to the dilation in producer to consumer temporal relationship, new incoming packets evict older packets that are yet to be copied. Once a few packets are copied after TCP window closed, acks are sent back and TCP window allows for more packets. This process continues and packets are always backed up for copies and we call it a batching anomaly.

To measure the extent of batch processing in experiments shown in Figure 10, we instrumented the kernel to report average socket queue length every time application is scheduled to copy. The instrumentation was relatively low overhead and the results are averaged over long runs. CPU used in Figure 3 has a 4MB cache for each pair of cores and the experiment was to measure packets cache footprint. A test with 3 Gb/s and 24 TCP connections (8 connections per transmit port) showed that an aggregate throughput of 2.8 Gb/s was achieved at 90% CPU and 2 packets/connection on average were pending prior to copy. Thus, packet cache footprint based on average packet size was 62KB. In the 4 Gb/s experiment with 32 TCP connections (8 per port), CPU was at 100% before achieving line rate. In comparison, aggregate throughput was 3.3 Gb/s with a backlog of 47 packets/connection prior to each copy. Packet cache footprint was 2,023KB - half the size of the L2 cache. This clearly shows that each packet injected into the cache has a very high probability of getting evicted prior to use.

4.3. Solutions to the batching anomaly

One of the first solutions that we tried was to reduce the interrupt rate so that there is a larger interval between interrupts and the application gets more time to copy. This reduces overall overhead on the system because of fewer interrupts although at the cost of higher latency, however, it does not help reduce packet batching for copies. Execution time saved from interrupts is used up by softIRQ which processes headers and takes up CPU until there are no more packets. Since the application scheduling is still pushed out until TCP window stops sender, the packets are backed up for copies.

The second option was to yield more frequently from softIRQ to application. The idea was to allow an application to keep up copies with header processing. However, reducing the maximum number of packets that can be processed back to back by softIRQ down to even one packet did not reduce cache evictions. In this scenario, the application copies packets more often and sends acknowledgements to the sender, which in turn moves the TCP window ahead. Since the copy happens in smaller chunks the window adjusts by smaller amounts, however, the packet footprint on cache is still the entire TCP window. Moreover, due to frequent kernel to user context transitions, the overhead results in a lower performance.

We also attempted to make TCP window protocol cache injection aware by adjusting TCP window so that cache footprint is minimized. Since TCP window is controlled by a bandwidth delay product, it cannot be overridden by a local configuration parameter. Our conclusion from these experiments was that if network cache injection was turned on, then distributing incoming flows to multiple cores seems to be the best option. Ensuring that one or a few cores are not saturated by network processing works best for DCA. Note that this result is true for any cache injection policy because a saturated core(s) due to network traffic will be flow controlled by TCP. If core saturation cannot be avoided for some network workload, then cache injection can be dynamically turned on and off between packets by the driver.

It is also important to observe that packet batching may not occur in packet routing/forwarding type of applications. Since forwarding requires only header inspection which usually happens in IRQ context without significant delays, most benefit can be achieved by header cache injection alone. For a Windows end host, this cache pollution is also unlikely to occur since in Windows application buffers are pinned. The packets are copied by the driver in DPC without application scheduling delays.

5. 10GbE Multi-core performance with DCA

Cache injection with multiple cores is a challenging proposition since both hardware and system software needs to be well aligned to take advantage of it. It is important that packet processing happens on the core that is interrupted and the application is also scheduled on the same core. To scale processing on multiple cores, NIC should distribute incoming flows to multiple cores to load balance and maintain flow to core affinity. Receive Side Scaling (RSS) [15] defines a methodology to dynamically route incoming flows to cores. Modern NICs support RSS using multiple queues and the Linux networking stack as well has recently added support for receive and transmit queues.

In addition to a NIC supporting as many interrupt vectors as CPU cores in the system, it is important to have an I/O aware interrupt balance service. IRQ balancer from [18] is a smart interrupt balance service that is aware of CPU cache hierarchy, tries to maintain interrupt vector to core affinity and dynamically moves the interrupt vectors to less utilized cores to balance the workload. The interrupt balancer and OS scheduling policy have to be aware of each other so that application thread receiving I/O executes on the interrupted core. These factors are imperative for scaling multi-core performance and cache injection to function properly on a multi-core system.

For multi-core DCA tests, we have used the setup as described in Figure 3 with all 8 cores active and Iperf can scheduled on any core by the OS. Interrupts are not pinned to any specific core(s) and irqbalance service is used to manage interrupt affinity. The driver ensures that the NIC injects packet data from multiple connections to the cache of a core that is being interrupted for processing packets for all those connections.

5.1. DCA multi-core measurements

Figure 11 and 12 show DCA performance on a multi core setup. Markers in the figures plot average CPU utilization of 8 cores and the bars plot network throughput. For each I/O experiment, 8 TCP connections are run consecutively. On SUT, one or more TCP connections were mapped to each NIC receive queue and thus a core may process 0 or 1 or more connections. This happens because NIC uses the Toeplitz hash algorithm [15] on all incoming connections to assign them one of the 8 receive queues. The distribution can vary from run to run and multiple steams can be assigned to a single receive

queue depending on hash output. This can cause one core's utilization to be higher on average than another core. Irqbalance service helps in load balancing by frequently monitoring and moving the interrupts every few seconds. Since no explicit process or interrupt affinity was used in the tests, gains with DCA in these experiments are expected to be observed with real applications.

For both small I/Os and large I/Os experiments (Figure 11 and 12) CPU utilization is not pushed to 100%. In case of small I/Os, throughput is limited by the transmit side system. Percent performance gain is thus calculated by normalizing network throughput to 100% CPU utilization. Thus the gains mentioned below (for DCA over baseline) are relative and not the absolute CPU utilization difference. The first two bars of the graphs below plot the system network performance without hardware prefetchers (HWP). DCA helps reduce CPU utilization or increase throughput or both. The relative percent improvement with DCA for small I/Os is 20% on average and 32% for large I/Os. When HWP are enabled (last two bars in the graphs), DCA adds to performance by 4% for small I/Os and 8% for large I/Os.

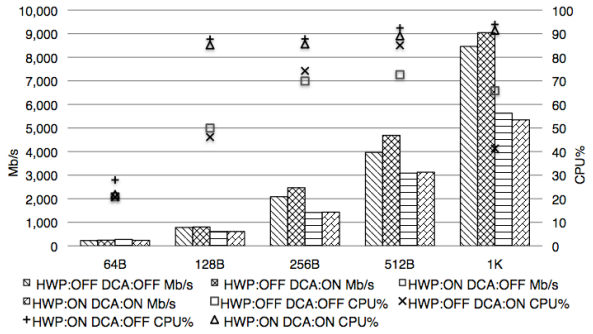

Figure 11. Small I/O 8 core performance w/ DCA (no LRO)

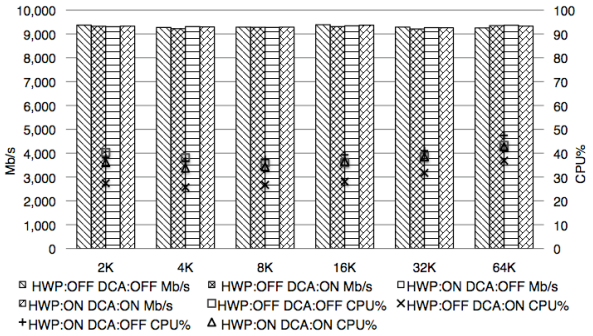

Figure 12. Large I/O 8 core performance w/ DCA (no LRO)

An important observation with HWP in this experiment is that overall network performance is hurt regardless of DCA (third bar from left). Recall that HWP had improved single core network processing performance. This impact is primarily due to aggressive prefetching by all cores in the system that leads to dilated memory access latency and excessive cache contention due to redundant data. DCA helps improve the performance but not by as big a margin as without HWP.

We'd like to call out another important result here that the data delivery to caches has scaled very well with multiple cores and at 10Gb/s. Figure 13 and Figure 8 show time spent per function for each packet for at 10Gb/s and 1Gb/s data rate respectively. The copy time and header processing time for the same I/O size is very comparable in both cases.

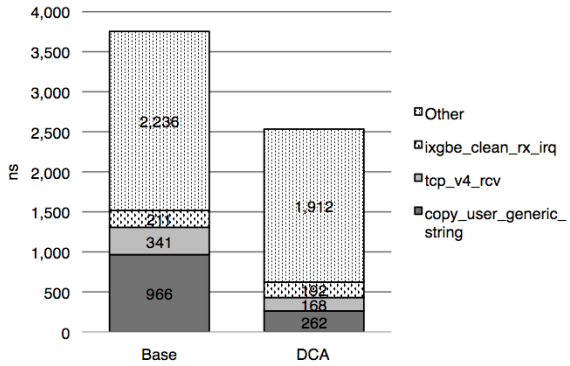

Figure 13. 4KB I/O per packet profile @10Gb/s with 8 Cores

5.2. DCA sensitivity to LRO

With LRO, incoming packets on the same connection are coalesced together to reduce header processing overhead. This significantly increases throughput for small I/O and lowers CPU cycles for large I/O. Without HWP, DCA further improves LRO performance by 7% for small I/O and 48% for large I/O (first two bars in Figure 14 and 15). For a few small I/O tests, DCA hurts performance when it is turned on with LRO. If HWP are enabled (last two bars in Figure 14 and 15), we observe an overall performance drop as seen in section 5.1 also. In this configuration, DCA performance with small I/O is even worse. So we investigated a little more into small I/O performance.

The average CPU utilization and throughput numbers did not show much insight into issues with small I/Os. Observing the system behavior at run time, we noticed that the NIC does not evenly allocate incoming connections to eight receive queues. The distribution was more balanced when the incoming connections were in the order of hundreds. Thus for 8

TCP connections, more than one connection would get assigned to one receive queue and because of lots of small packets, the NIC interrupted more often. The result of small packet processing and interrupt overhead pushes the core to 100% resulting in the batching anomaly explained in Section 4.2. Since the total connections in the test were 8 and several of those were processed by one core, some cores in the system were idle. Thus the irqbalancer moved the interrupts to idle cores every few seconds in order to better load balance the system. However, since the workload from multiple connections would saturate any core that they are processed on, data pushed into caches resulted in the batching anomaly. Since each core is not 100% busy during the entire experiment, the anomaly is not evident from average core utilization.

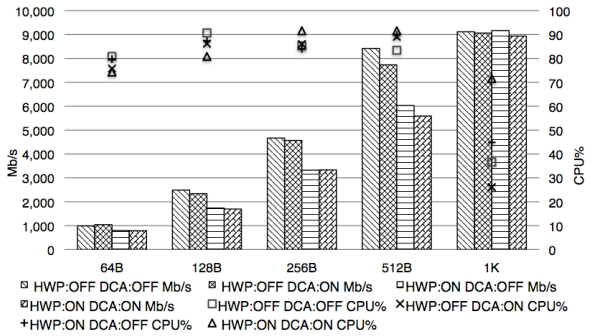

Figure 14. Small I/O DCA sensitivity to LRO w/ HWP on 8 cores

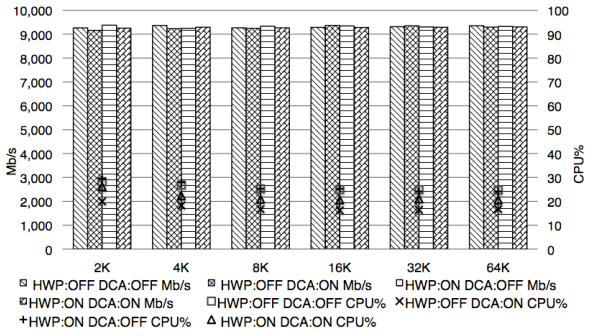

Figure 15. Large I/O DCA sensitivity to LRO w/ HWP on 8 cores

The small I/O issue is not a big concern for DCA since real workloads usually do not constantly blast the wire with small packets. Small packets usually come in spikes during the workload or the sender coalesces data into larger frames before transmitting. In both cases, even if core saturates it will occur for a short period only. A better characterization will be using SpecWeb benchmark which uses a mix of small and large packets.

6. Limitations and Future Challenges

Directing network data to caches despite the challenges has been shown to work well, however the trend towards integrated memory controllers adds another challenge. The system now has to ensure that the application executes on a CPU node that allocated buffers on memory attached to it and that interrupt follows the same affinity. Despite ensuring proper affinity for network workloads, we have seen that they cause tremendous impact to other applications sharing the same cache. Such issues are present regardless of cache injection and in this section we show the impact at just 2Gb/s data rate. The performance loss is going to be even higher at 10Gb/s and unless there are solutions to address this issue, we will observe gross inefficiencies at 40 and 100Gb/s.

6.1. NUMA and Non uniform I/O access

With a trend towards NUMA style architecture and point-to-point high bandwidth coherent links between CPUs, we expect that bandwidth contention issues will be temporarily alleviated. An awkward problem that will surface potentially even at 10Gb/s, is the disconnect between flow affinity and NUMA affinity. We observed earlier that it is important to affinitize interrupt processing to the application context (not the other way around). However, NUMA configurations make it important that the network data

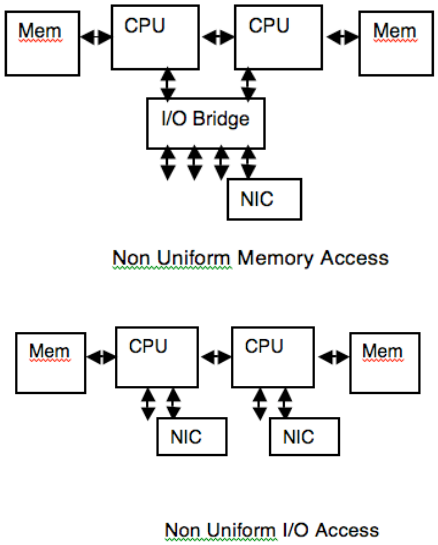

Figure 16. Non-Uniform memory and I/O access pre-dispose networking to scaling problems

structures also be allocated local to the CPU running the application context. Otherwise benefits such as lower memory access latency and higher bandwidth will be compromised.

While NUMA implies physical memory affinity and memory-aware scheduling, a similar issue arises when physical I/O ports are associated directly with CPUs. Physical I/O ports being directly attached to CPUs may be justified by cost reduction and scalability considerations. However, when I/O attached to one CPU is used to feed remote CPUs, longer latency of interaction and the utilization of CPU to CPU coherent links will become concerns. This adds another dimension to affinitization problems. In this case, in performance sensitive scenarios, consuming application threads will have to be scheduled on the same CPU that the producing I/O ports are connected to.

In both these cases, DCA would still add value not only by hiding memory access latency from local nodes but also if the data has to be fetched from memory attached to a remote CPU node. In summary, affinitization and new data movement protocols enable the possibility of multi-core scaling that is only limited by on-die interconnect and cache bandwidth that typically far exceed Ethernet bandwidths. Such enhancements do not exist on typical server platforms at this time but their implementation can clear the path towards near-linear multi-core scaling.

6.2. Shared cache usage by networking

Any touch of network packet payload by cores can pollute both networking and non-networking working sets existing in shared caches of a cache hierarchy. In a standard network stack, host copies have such an impact and any subsequent copies by layers above TCP/IP can further exacerbate the problem. Since both Rx-side and Tx-side processing involve copies, both scenarios will exhibit interference effects. Figure 17 shows a characterization of the interference with a focused test with two cores that share a 4MB, 16-way L2 cache. On the first core, we ran SPEC2000 applications, while the second core was dedicated for 2 Gb/s of network receive traffic and observed performance degradation relative to the case where there was no interference. By using bandwidth measurements, we observed that for a majority of the applications, performance degradation is due to cache interference and not external bus or memory bandwidth contention.

A three-prong approach to minimizing interference effects is currently being researched: (1) minimize the address space used by packet buffers (2) bypass shared caches delivering data directly into a smaller cache or buffer within a core and (3) backup the small cache/buffer by limited ways in a shared cache.

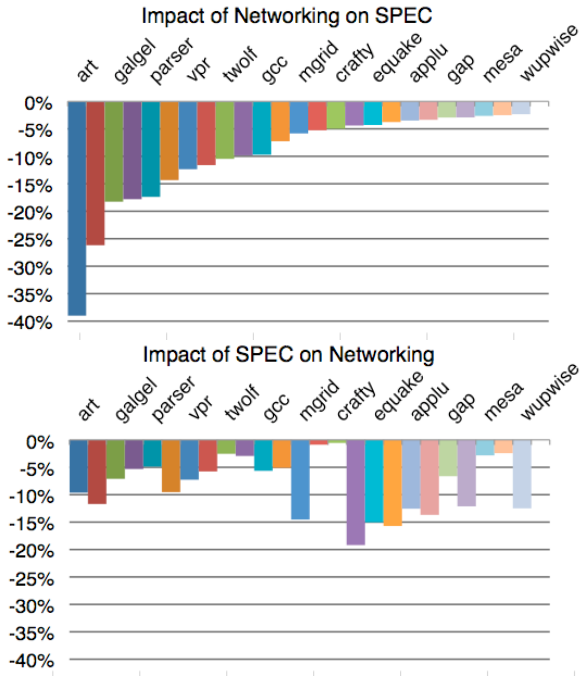

Figure 17. Cache interference effect between applications and network Rx processing

7. Conclusion

With improvements such as TSO, LRO, Intel® Core™ micro-architecture and multi-core CPUs, general purpose architectures are on a trajectory that can accommodate 10Gb/s or higher Ethernet rates with reasonable efficiency. One exception is that of memory dependency of network protocol processing. Memory focused architectural improvements such as Direct Cache Access must be evaluated against various conditions to understand the full extent of benefit. Due to the complexity of hardware-software interactions, such evaluations are often only possible with realistic implementations with commercial operating systems and network stacks. In this paper, conditions that surface uniquely at higher rates such as 10GbE rates relative to 1GbE were analyzed in detail. We demonstrated the sensitivity to distribution of network flows to the multiple cores in the system and the importance of affinitizing the interrupt and application contexts. Results indicate that once a good distribution is achieved, the relative performance (throughput or CPU utilization) improvement with DCA in multi-core measurements is significant – 20% for small I/Os and 32% for large I/Os. In addition to application and interrupt affinity, we reviewed that

memory and physical I/O affinity are important considerations for multi-core scaling. Once efficient multi-core scaling is achieved, numerous, realistic applications that depend on large I/O network transfers may not need specialized hardware acceleration. However, as mentioned earlier, networking technology is inevitably trending to 40GbE and 100GbE standards. When compounded with small I/O sizes, the next wave of changes will be fundamental in nature. Going forward, for high performance computing applications that want to take advantage of Ethernet, user mode network stacks and iWARP have been suggested. We anticipate that in addition to significant changes to the network stack, dedicated cores even if they are from a homogenous pool of cores and mechanisms to bypass or limit the use of shared caches will be needed. Integration of the network interface [16] and a new model of core and network interface interactions may be needed for scalability to greater bandwidth per port, more ports and greater compute density.

8. References

[1] G. Regnier et al, "TCP Onloading For Data center Servers", IEEE Computer. November 2004.

[2] Doug Freimuth et al., "Server network scalability and TCP offload", Proceedings of the USENIX Annual Technical Conference, Anaheim, CA. 2005.

[3] G. Narayanaswamy et al., "An Analysis of 10-Gigabit Ethernet Protocol Stacks in Multicore Environments", IEEE International Symposium on High Performance Interconnects (HotI), August 2007.

[4] A. Foong et al., "TCP Performance Analysis Re-visited", IEEE Int'l Symposium on Performance Analysis of Software and Systems (ISPASS), Mar 2003.

[5] "Inside Intel® Core™ Microarchitecture: Setting New Standards for Energy-Efficient Performance". http://www.intel.com/technology/architecture-silicon/core/.

[6] Ram Huggahalli., "Direct Cache Access for High Bandwidth Network I/O", 32nd Annual International Symposium on Computer Architecture (ISCA'05), 2005.

[7] Amit Kumar et al., "Impact of Cache Coherence Protocols on the Processing of Network Traffic". 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40), 2007.

[8] Edgar Leon et al., "Reducing the Impact of the MemoryWall for I/O Using Cache Injection". Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects, 2007.

[9] PCI Express® Base Specification 2.0. http://www.pcisig.com/specifications/pciexpress.

[10] Write Combining Memory Implementation Guidelines. Application Note. Intel Corporation. Order Number: 244422-001. November 1998.

[11] Ashish Jha and Darren Yee. "Increasing Memory Throughput with Intel® Streaming SIMD Extensions 4 (Intel® SSE4) Streaming Load". May 22, 2008

[12] http://dast.nlanr.net/Projects/Iperf

[13] J. Nagle, "Congestion control in IP/TCP internetworks". RFC 896, Internet Engineering Task Force, ftp://ftp.isi.edu/in-notes/rfc896.txt. January 1984.

[14] Jamal Hadi Salim, "When NAPI comes to town", White paper on NAPI – 2005

[15] "Scalable Networking: Eliminating the Receive Processing Bottleneck – Introducing RSS". Microsoft WinHEC April 2004.

[16] Nathan Binkert et al., "Integrated network interfaces for high-bandwidth TCP/IP". Proceedings of the 2006 ASPLOS Conference. December 2006.

[17] H. Krawczyk, ``New hash functions for message authentication," Proc. Eurocrypt'95, LNCS 921, L.C. Guillou and J.-J. Quisquater, Eds., Springer-Verlag, 1995, pp. 301--310.

[18] http://www.irqbalance.org/