Benefits of I/O Acceleration Technology (I/OAT) in Clusters

K. VAIDYANATHAN AND D. K. PANDA

Technical Report
Ohio State University (OSU-CISRC-2/07-TR13)

The 2007 IEEE International Symposium on Performance Analysis of Systems and Software

Benefits of I/O Acceleration Technology (I/OAT) in Clusters

K. Vaidyanathan
Computer Science and Engineering,
The Ohio State University
vaidyana@cse.ohio-state.edu

D. K. Panda
Computer Science and Engineering,
The Ohio State University
panda@cse.ohio-state.edu

Abstract

Packet processing in the TCP/IP stack at multi-Gigabit data rates occupies a significant portion of the system overhead. Though there are several techniques to reduce the packet processing overhead on the sender-side, the receiver-side continues to remain as a bottleneck. I/O Acceleration Technology (I/OAT) developed by Intel is a set of features particularly designed to reduce the receiver-side packet overhead. This paper studies the benefits of the I/OAT technology by extensive evaluations through micro-benchmarks as well as evaluations on two different application domains: (1) A multi-tier data-center environment and (2) A Parallel Virtual File System (PVFS). Our micro-benchmark results show that I/OAT results in 38% lower overall CPU utilization in comparison with traditional communication. Due to this reduced CPU utilization, I/OAT delivers better performance and increased network bandwidth. Our experimental results with data-centers and file systems reveal that I/OAT can improve the total number of transactions processed by 14% and throughput by 12%, respectively. In addition, I/OAT can sustain larger number of concurrent threads (up to a factor of 4 as compared to non-I/OAT) in a data-center environment, thus increasing the scalability of the servers.

Keywords: *I/O Acceleration, Sockets, Data-Center, PVFS*

1. Introduction

There has been an incredible growth of highly data-intensive applications in the fields of medical informatics, genomics and satellite weather image analysis in the recent years. With technology trends, the ability to store and share the datasets generated by these applications is also increasing, allowing scientists and institutions to create large dataset repositories and making them available for use by others. On the other hand, clusters consisting of commodity off-the-shelf hardware components have become increasingly attractive as platforms for high-performance computation and scalable servers. Based on these two trends, researchers have proposed the feasibility and potential of cluster-based servers [13, 9, 17, 18].

Several clients request these servers for either the raw or some kind of processed data simultaneously. However, current servers are becoming increasingly incapable of meeting such sky-rocketing processing demands with high-performance and in a scalable manner. Current servers rely on TCP/IP for data communication and typically use Gigabit Ethernet networks for cost-effective designs. The host-based TCP/IP protocols on such networks have high CPU utilization and low bandwidth limiting the maximum capacity (in terms of requests they can handle per unit time). Alternatively, many servers use multiple Gigabit Ethernet cards to cope with the network traffic. However, at multi-Gigabit data rates, packet processing in the TCP/IP stack occupies a significant portion of the system overhead.

Packet processing [11, 12] usually involves manipulating the headers and moving the data through the TCP/IP stack. Though this does not require significant computation, processor time gets wasted due to delays caused by latency of memory access and data movement operations. To overcome these processing overheads, researchers have come up with several technologies [8] such as transport segmentation offload (TSO), jumbo frames, zero-copy data transfer (by using sendfile()), interrupt coalescing, etc. Unfortunately, many of the proposed optimizations are applicable only on the sender side while the receiver side continues to remain as a bottleneck in several cases, thus resulting in a huge performance gap between the CPU overheads of sending and receiving packets.

Intel's I/O Acceleration Technology (I/OAT) [1, 3, 2, 14] is a set of features which attempts to alleviate the receiver packet processing overheads. It has three additional features, namely (i) split headers, (ii) DMA copy offload engine and (iii) multiple receive queues.

At this point, the following open questions arise:

- What kind of benefits can be expected from the current I/OAT architecture?
- How does this benefit translate to end-user applications?
- What are the trade-offs associated with such an architecture?

In this paper, we focus on the above questions. We first analyze the performance of I/OAT based on a detailed suite of micro-benchmarks. Next, we evaluate it on two different application domains:

- A multi-tier Data-Center environment
- A Parallel Virtual File System (PVFS)

Our micro-benchmark results show that I/OAT results in reducing the overall CPU utilization significantly (up to 38%) as compared to traditional communication (non-I/OAT). Due to this reduced CPU utilization, I/OAT delivers better performance and increased network bandwidth. In particular, the DMA engine feature of I/OAT achieves close to 16% lower CPU utilization as compared to non-I/OAT. The split-header feature results in improving the network performance by 32%. Our experimental results with data-centers and file systems reveal that I/OAT can improve the total number of transactions processed by 14% and the throughput by 12%, respectively. Also, our results show that I/OAT can sustain larger number of concurrent threads (up to a factor of 4 as compared to non-I/OAT) in a data-center environment, thus increasing the scalability of the servers.

The remaining part of the paper is organized as follows: Section 2 provides a brief background of I/OAT architecture. In Section 3, we discuss the software infrastructure we used; in particular about Multi-Tier Data-Center environments and Parallel Virtual File System (PVFS). Section 4 deals with the evaluation of a number of micro-benchmarks to study the ideal case benefits of I/OAT over the native kernel implementation. Section 5 and Section 6 present the benefits of I/OAT in a data-center and PVFS environment, respectively. Section 7 presents the advantages and the disadvantages of I/OAT. We conclude the paper in Section 8.

2. Background

In this section, we present some of the socket optimization technologies in detail. Later, we provide a brief background of I/OAT architecture and its features.

2.1 Socket Optimizations

As mentioned in Section 1, due to technological developments, significant performance gap exists between the CPU overhead of sending and receiving packets. In particular, there are two key technologies that make the CPU usage on the sender-side far less than the CPU usage on the receiver-side.

The first technique, TCP segmentation offload (TSO), allows the kernel to pass a large buffer which is more than the maximum transmission unit (MTU) to the network controller. The network controller segments the buffer into individual Ethernet frames and sends it on the wire. In the absence of this technique, the host CPU is expected to break a large buffer into small frames and send the small frames to the network controller. This operation is more CPU intensive. The second technique, *sendfile()* optimization, allows the kernel to perform zero-copy operation. Using this technique, the kernel does not copy the user buffer to network buffer, rather points the pinned pages as the source of the data for data transmission.

However, none of these optimization are applicable in the receiver side. For receiver side optimization, interrupt coalescing technique helps in reducing the number of interrupts generated in the host system. This technique helps in generating one interrupt for multiple packets rather than generating an interrupt for every single packet. However, this optimization improves the performance only when the network is heavily loaded.

2.2 I/O Acceleration Technology (I/OAT) Overview

As mentioned in Section 1, there still exists several bottlenecks in the receiver-side which impacts performance. I/OAT [14] attempts to alleviate these bottlenecks by providing a set of features, namely (i) Split headers, (ii) Asynchronous copy using DMA engine and (iii) Multiple Receive Queues.

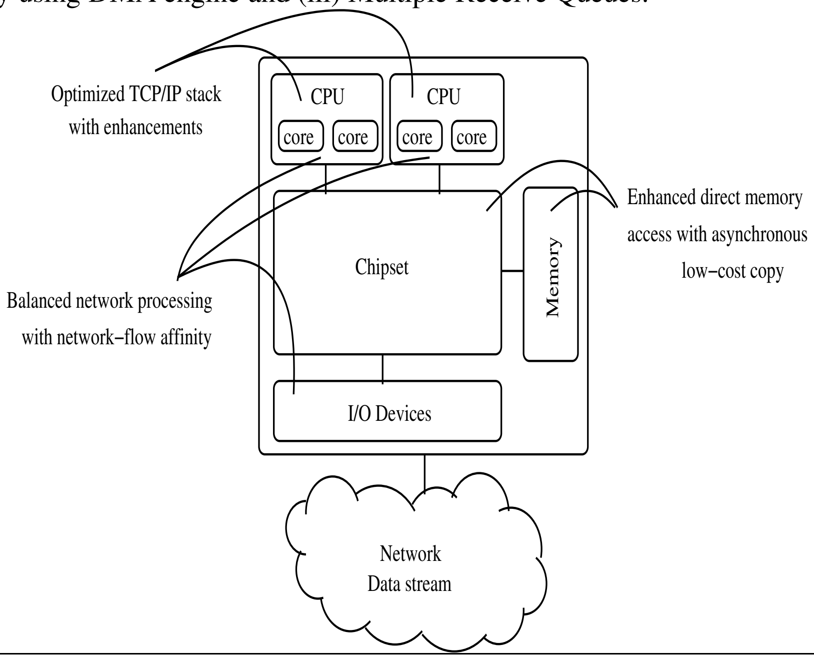


Figure 1. Intel's I/O Acceleration Technology (Courtesy Intel)

2.2.1 Split-headers

As shown in Figure 1, the optimized TCP/IP stack of I/OAT has the split header feature implemented. In TCP/IP-based communication, for transmission of application data, several headers such as TCP header, IP header, Ethernet header are attached along with the application data. Typically, the network controller DMAs the headers and the application data into a single buffer. However, with the split-header feature, the controller partitions the network data into headers and application data and copies them into two separate buffers. This allows the header and data to be optimally aligned. Since the headers are frequently accessed during network processing, this feature also results in better cache utilization by not polluting the CPU's cache with any application data while accessing these headers, thus increasing the locality of the incoming headers. For more information regarding this feature and its benefits, please refer to [14, 15].

2.2.2 Asynchronous copy using DMA engine

As mentioned in [14], most of the time during receive processing is spent in copying the data from kernel buffer to user buffer. In particular, the ratio of useful work done by the CPU to useless overheads such as CPU waiting for memory access or data movement decreases as we go from 1 Gbps links to 10 Gbps links [16]. Especially, for data movement operations, rather than waiting for a memory copy to finish, the host CPU can process other pending packets while the copy is still in progress. I/OAT offloads this copy operation with an additional DMA engine. This is a dedicated device which can perform memory copies. As a result, while the DMA is performing the data copy, the CPU becomes free to process other pending packets.

Though the DMA engine offers several benefits, the following issues need to be taken care of. The memory controller uses physical addresses, so a single transfer cannot span discontinuous physical pages and also the pages cannot be swapped during a copy operation. Hence, the physical pages need to be pinned before initiating the copy operation and memory operations should be broken up into individual page transfers. Also, the copy engine must maintain cache coherence immediately after data transfer. Data movement performed by the memory controller should not ignore the data stored in the processor cache, potentially requiring a cache coherence transaction on the bus. For more details regarding this feature and its implementation, please refer to [14, 16, 15].

2.2.3 Multiple Receive Queues

Processing large packets is generally not CPU-intensive, whereas processing small packets can fully occupy the CPU. Even on multi-CPU systems, processing occurs on a single CPU, the CPU which handles the controller's interrupt. The next generation Intel network adapter has multiple receive queues. It helps in distributing the packets among different receive queues and allows multiple CPUs to act on different packets at the same time. For more details regarding this feature, please refer to [14, 15]. Unfortunately, this feature is currently disabled in Linux platform. Hence, we could not measure the performance impact of this feature in our evaluation.

3. Software Infrastructure

We have carried out the evaluation of I/OAT on two different software infrastructures: Multi-Tier Data Center environment and the Parallel Virtual File System (PVFS). In this section, we discuss each of these in more detail.

3.1 Multi-Tier Data Center environment

More and more people are using web interfaces for a wide range of services. Scalability of these systems is a very important factor. Upgrading a server to single more powerful server is not a feasible method anymore, clusters are the more preferred ones here. There are several benefits of using clusters. Firstly, the nodes are made

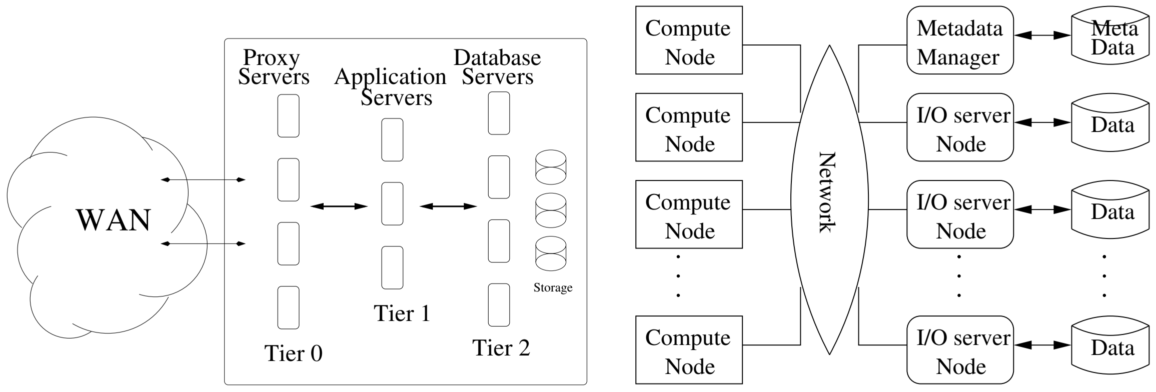


Figure 2. Application Domains (a) A Multi-Tier Data-Center Environment (b) Parallel Virtual File System (PVFS) Environment

of low cost off-the-shelf components reducing the cost of the setup. Secondly, this setup is easy to modularize, so the maintenance cost and the effort is considerably lower. Also, we have a lot of flexibility in setting up the various tiers of a data-center. Nodes can be reconfigured or new ones can be added, on demand, to help the loaded tiers.

A typical Multi-tier Data-center [19, 7, 20] has as its first tier, a cluster of nodes called the edge nodes. These nodes can be thought of as switches (up to the 7th layer) providing load balancing, security, caching etc. The main purpose of this tier is to help increase the performance of the inner tiers. The next tiers are usually the web-servers. These nodes apart from serving static content, can fetch dynamic data from other sources and serve that data in presentable form. The last tier of the data-center is the database tier. It is used to store persistent data. This tier is usually I/O intensive. Figure 2a shows a typical data-center setup.

A request from a client is received by the edge or proxy servers. This request is serviced from cache if possible, otherwise it is forwarded to the web/application servers. Static requests are serviced by the web servers by just returning the requested file to the client via the edge server. This content may be cached at the edge server so that subsequent requests to the same static content may be served from the cache. The application tier handles dynamic content. Any request that needs some value to be computed, searched, analyzed or stored has to use this tier at some stage. The application servers may also need to spend some time on converting data to presentable formats. The back-end database servers are responsible for storing data persistently and respond to queries. These nodes are connected to a persistent storage system. Queries to the database systems can be anything in between, simple seek of required data to performing joins/aggregation/select operations on the data.

3.2 Parallel Virtual File System (PVFS)

Parallel Virtual File System (PVFS) [6] is one of the leading parallel file systems for Linux cluster systems today. It was designed to meet the increasing I/O demands of parallel applications in cluster systems. Figure 2b demonstrates a typical PVFS environment. As demonstrated in the figure, a number of nodes in the cluster system can be configured as I/O servers and one of them (either an I/O server or a different node) as a meta-data manager. It is possible for a node to host computation while serving as an I/O node.

PVFS achieves high performance by striping files across a set of I/O server nodes allowing parallel accesses to the data. It uses the native file system on the I/O servers to store individual file stripes. An I/O daemon runs on each I/O node and services requests from the compute nodes, in particular the read and write requests. Thus, data is transferred directly between the I/O servers and the compute nodes.

A manager daemon runs on a meta-data manager node. It handles meta-data operations involving file permissions, truncation, file stripe characteristics, and so on. Meta-data is also stored on the local file system. The meta-data manager provides a cluster-wide consistent name space to applications. In PVFS, the meta-data manager does not participate in read/write operations.

PVFS supports a set of feature-rich interfaces, including support for both contiguous and non-contiguous accesses to both memory and files [10]. PVFS can be used with multiple APIs: a native API, the UNIX/POSIX API, MPI-IO [21], and an array I/O interface called the Multi-Dimensional Block Interface (MDBI). The presence of multiple popular interfaces contributes to the wide success of PVFS in the industry.

4. I/OAT Micro-Benchmark Results

In this section, we compare the ideal case performance achievable by I/OAT with the native sockets implementation (non-I/OAT) using a set of micro-benchmarks. In Sections 5 and 6, we study the performance achieved by I/OAT and non-I/OAT in the Data-Center and PVFS environments, respectively.

For all our experiments we used two testbeds whose descriptions are as follows:

Testbed 1: A system consisting of two nodes built around SuperMicro X7DB8+ motherboards which include 64-bit 133 MHz PCI-X interfaces. Each node has a dual-core dual Intel 3.46 GHz processors with a 2 MB L2 cache. The machines are connected with three Intel PRO 1000Mbit adapters with two ports each through a 24-port Netgear Gigabit Ethernet switch. We used the Linux RedHat AS 4 operating system and kernel version 2.6.9-30.

Testbed 2: A cluster system consisting of 44 nodes. Each node has a dual Intel Xeon 2.66 GHz processors with 512KB L2 cache and 2GB of main memory.

We used the nodes in Testbed 2 as clients to send requests to the I/OAT servers. For all other experiments, we used Testbed 1. Also, for experiments within Testbed 1, we created a separate VLAN for each network adapter in one node and a corresponding IP address within the same VLAN on the other node to ensure an even distribution of network traffic. In all our experiments, we define the term relative CPU benefit of I/OAT as follows: if a is the % CPU utilization of I/OAT and b is the % CPU utilization of non-I/OAT, the relative CPU benefit of I/OAT is defined as $(b - a)/b$. For example, if I/OAT occupies 30% CPU and non-I/OAT occupies 60% CPU, the relative CPU benefit of I/OAT is 50%, though the absolute difference in CPU usage is only 30%.

4.1 Bandwidth and Bi-directional Bandwidth

Figure 3a shows the bandwidth achieved by I/OAT and non-I/OAT with increasing number of network ports. We use the standard *ttcp* benchmark for measuring the bandwidth. As the number of ports increase, we expect the bandwidth to increase. From the figure, we see that both I/OAT and non-I/OAT achieves similar bandwidth with increasing number of ports. The maximum bandwidth achieved is close to 5635 Mbps with six network ports. However, we see a difference in performance with respect to the CPU utilization on the receiver side. We can observe that the CPU utilization is lower for I/OAT as compared to non-I/OAT using three network ports and this difference increases as we see increase the number of ports from three to six. For a six port configuration, non-I/OAT occupies close to 37% of the CPU while I/OAT occupies only 29% of the CPU. The relative benefit achieved by I/OAT in this case is close to 21%.

In the bi-directional bandwidth test, we used two machines and 2*N threads on each machine with N threads acting as servers and the other N threads as clients. Each thread on one machine has a connection to exactly one thread on the other machine. The client threads connect to the server threads on the other machine. Thus, 2 * N connections are established between these two machines. On each connection, the basic bandwidth test is performed using the ttcp benchmark. The aggregate bandwidth achieved by all the threads is calculated as the bi-directional bandwidth, as shown in Figure 3b. In our experiments, N is equal to the number of network ports. We see that the maximum bi-directional bandwidth is close to 9600 Mbps. However, we also observe that I/OAT shows improvement in CPU utilization using just two ports and this improvement increases with increasing number of ports. With six network ports, non-I/OAT occupies close to 90% of CPU whereas I/OAT occupies only 70% of the CPU. The relative CPU benefit achieved by I/OAT is close to 22%. This trend also suggests that, with an addition of one or two network ports to this configuration, non-I/OAT may not give the best network throughput in comparison with I/OAT since non-I/OAT may end up occupying 100% CPU.

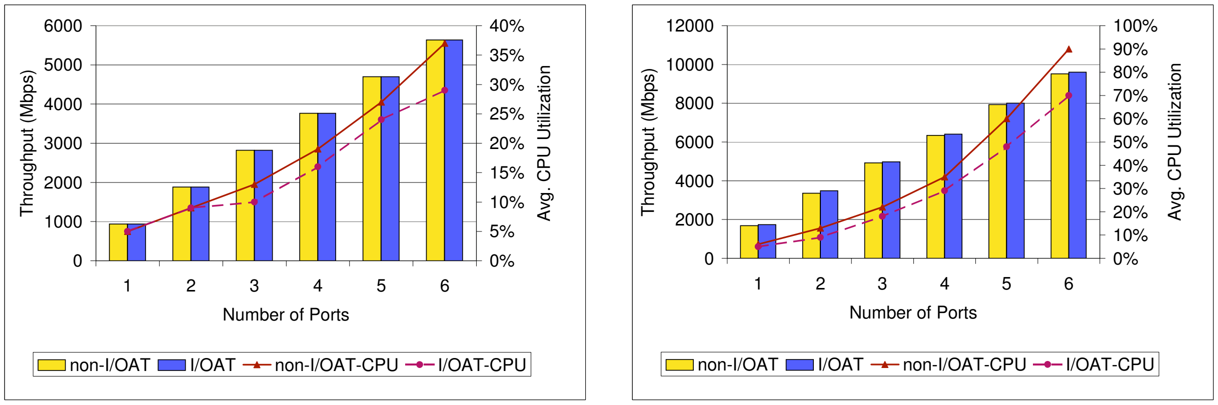


Figure 3. Micro-Benchmarks: (a) Bandwidth, (b) Bi-directional Bandwidth

4.2 Multi-Stream Bandwidth

Multi-stream bandwidth test is very similar to the bi-directional bandwidth test as mentioned above. However, in this experiment, only one machine acts as a server and the other machine as the client. We use two machines and N threads on each machine. Each thread on one machine has a connection to exactly one thread on the other machine. On each connection, the basic bandwidth test is performed. The aggregate bandwidth achieved by all the threads is calculated as the multi-stream bandwidth. As shown in Figure 4, we observe that the bandwidth achieved by non-I/OAT and I/OAT are similar for increasing number of threads. However, when the number of threads increases to 120, we see a degradation in performance of non-I/OAT whereas I/OAT consistently shows no degradation in network bandwidth for increasing number of threads. Further, the CPU utilization for non-I/OAT also increases with increasing number of threads. With 120 threads in the system, we see that non-I/OAT occupies close to 76% CPU whereas I/OAT only occupies 52% resulting in 24% absolute benefit in CPU utilization. The relative CPU benefit of I/OAT in this case, is close to 32%.

4.3 Bandwidth and Bi-directional Bandwidth with Socket Optimizations

As mentioned in Section 2, there exists several optimizations on the sender side to reduce the packet overhead and also to improve the network performance. In this experiment, we considered three such existing optimizations: (i) Large Socket buffer sizes (100M), (ii) Segmentation Offload (TSO) and (iii) Jumbo Frames. Our main aim in this experiment is to understand the impact of each of these optimizations and observe the improvement both in terms of throughput and CPU utilization on the receiver-side. Case 1 uses the default socket options without any optimization. In Case 2, we increased the socket buffer size to 100 MB. For Case 3, we further

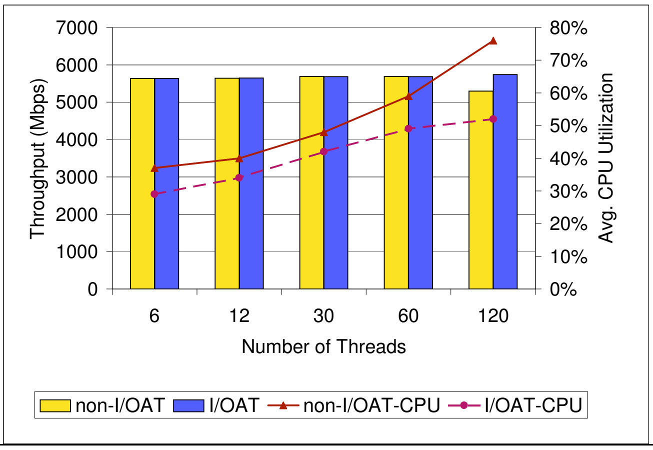


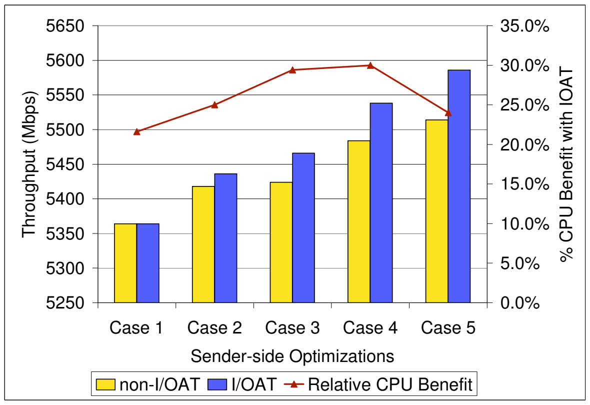
Figure 4. Multi-Stream Bandwidth

improved the optimization by enabling segmentation offload (TSO) so that the host CPU is relieved from fragmenting large packets. In Case 4, including the previous optimizations, we increased the MTU-size to 2048 bytes so that large packets are sent over the network. In addition to the sender-side optimizations, in Case 5, we added the interrupt coalescing feature. Performance numbers with various socket optimizations are shown in Figure 5.

In the bandwidth test, as shown in Figure 5a, we observe two interesting trends. Firstly, as we increase the socket optimizations, we see an increase in the aggregate bandwidth. Further, we also observe that the performance of I/OAT is consistently better than non-I/OAT. Especially for Case 5, which includes all the socket optimizations, the bandwidth achieved by I/OAT is close to 5586 Mbps whereas non-I/OAT achieves only 5514 Mbps. More importantly, we also observe that there is a significant improvement in terms of CPU utilization. As shown in the figure, the relative percentage CPU benefit with I/OAT increases as we increase the socket optimizations. Especially, for Case 4, we observe that I/OAT achieves 30% relative CPU benefit in comparison with non-I/OAT.

We see similar trends with bi-directional bandwidth test as shown in Figure 5b. Further, the relative benefits achieved by I/OAT is much higher as compared to the bandwidth experiment. Especially for Case 4, I/OAT achieves close to 38% relative CPU benefit as compared to non-I/OAT.

In summary, we see that the microbenchmark results show a significant improvement in terms of CPU utilization and network performance for I/OAT. Since I/OAT in Linux has two features, namely the split-header and DMA copy engine, it is also important to understand the benefits attributed by each of these features individually. In the following section, we design several experiments to show the individual benefits.



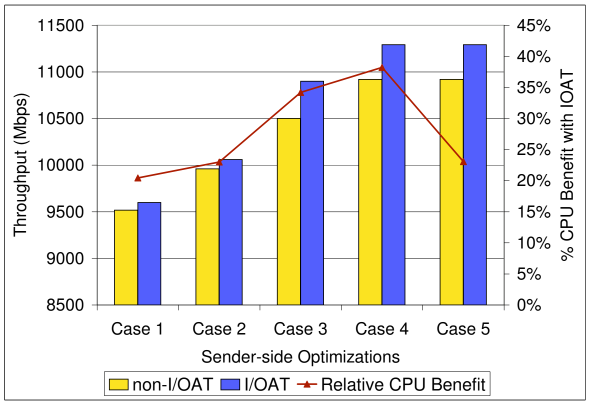


Figure 5. Optimizations: (a) Bandwidth (b) Bi-directional Bandwidth

4.4 Benefits of Asynchronous DMA Copy Engine

In this section, we isolate the DMA engine feature from I/OAT and show the benefits of an asynchronous DMA copy engine. We compare the performance of copy engine with the traditional CPU-based copy and show its benefits in terms of performance and overlap efficiency. For CPU-based copy, we use the standard *memcpy* utility.

Figure 6 compares the cost of performing a copy using the CPU and I/OAT's DMA engine. The *copy-cache* bars denote the performance of CPU-based copy where the source and destination buffers are in cache and the *copy-nocache* bars denote the performance of CPU-based copy with source and destination not in cache. The *DMA-copy* bars include the total cost of performing the copy and the *DMA-overhead* bars include the startup overhead in initiating the copy using the DMA engine. The *Overlap* line denotes the percentage of DMA copy time that can be overlapped with other computation. As shown in Figure 6, we see that *DMA-copy* performs better than *copy-nocache* for message sizes greater than 8K. Further, we also observe that the percentage of overlap increases with increasing message sizes reaching up to 93% for 64K message size. However, if the source and destination buffers are in cache, we observe that the performance of the CPU-based copy is much better than performance of the DMA-based copy approach. The interesting point to note here is that, since the DMA-based copy can be overlapped with processing other packets, we incur only the DMA startup overheads. As observed in the figure, we see that the DMA startup overhead time is much less than the time taken by CPU-based copy approach. Thus, the DMA copy engine can also be useful even if the source and destination buffers are in cache.

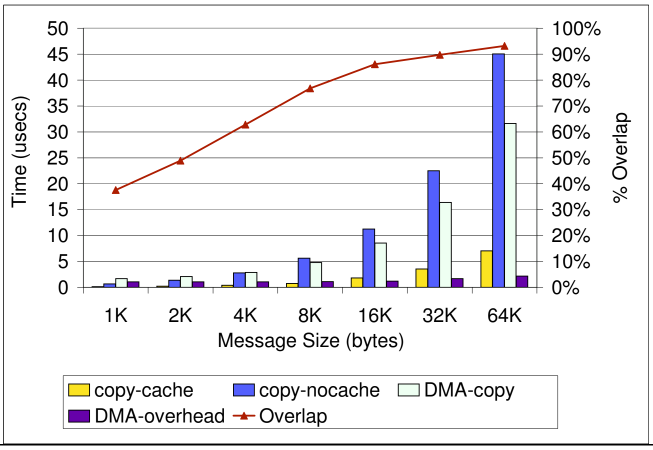


Figure 6. CPU-based Copy vs DMA-based Copy

4.5 I/OAT Split-up

In this section, we breakdown the two features of I/OAT and show its benefits in terms of throughput and CPU utilization, as shown in Figure 7. We first measure the performance of non-I/OAT without the DMA engine and the split header features. Next, we include the DMA engine feature (denoted as I/OAT-DMA in the figure) and measure the performance. We report the differences in performance as DMA engine benefits. Finally, we include the split-header feature along with DMA engine feature (denoted as I/OAT-SPLIT in the figure) and compare the performance of I/OAT with DMA engine and I/OAT with both features. We report the difference in performance as split-header benefits.

We design the experiment in the following way. We use the two nodes from Testbed 1, one acting as a server and the other as a client. We use two Intel network adapters with two ports each on both the server and the client node. Since we have four network ports on each node, we use four clients and perform a bandwidth test with four server threads. We first measure the bandwidth performance for small message sizes for all three cases, i.e., non-I/OAT, I/OAT-DMA and I/OAT-SPLIT and report the relative CPU benefits in Figure 7a. As shown in the figure, we observe that the DMA engine feature achieves close to 16% relative CPU benefit compared to the non-I/OAT case. This improvement is seen for all message sizes from 16K to 128K. However, we do not observe any throughput improvement for all message sizes. Also, the split-header feature of I/OAT does not seem to improve the CPU or the throughput for all message sizes in this range.

As mentioned in Section 2, the split-header feature of I/OAT increases the locality of incoming headers and also helps in better cache utilization by not polluting the CPU's cache with application buffers during network processing. In order to show cache pollution effects, we repeat the previous experiment for large message sizes which cannot fit in the system cache. Figure 7b shows the throughput benefits achieved by I/OAT features for large message sizes. We observe that the split-header feature can achieve up to 26% benefit in throughput for transferring 1 Mbytes of application data. In this case, the I/OAT server gets a total of 4 Mbytes of application data from the four clients and since the cache size is only 2 Mbytes, clearly the application data does not fit in the system cache. For split-header feature, since the CPU's cache is not polluted with the application data, we see a huge improvement. In addition, we see that the benefits achieved by split-header feature decreases for increasing message sizes.

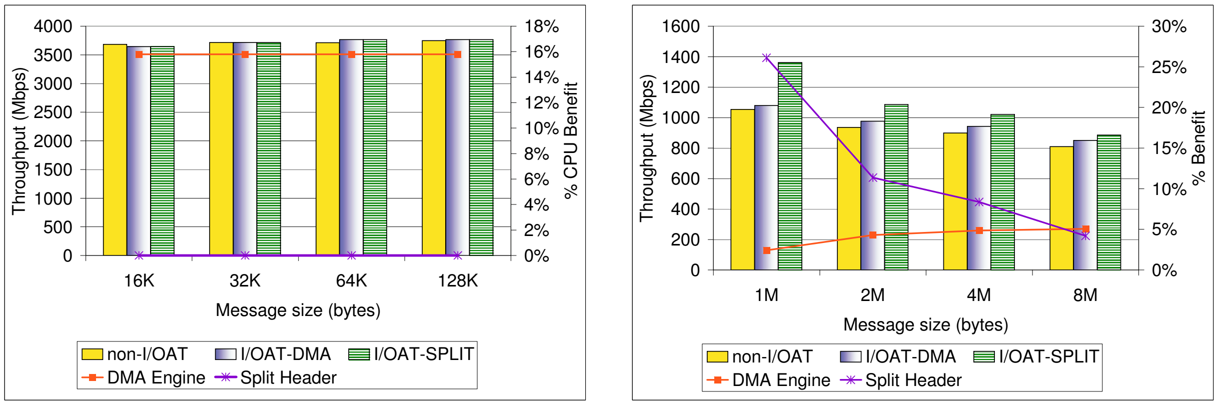


Figure 7. I/OAT split-up benefits: (i) CPU Utilization (ii) Throughput

5. Data-Center Performance Evaluation

In this section, we analyze the performance of a 2-tier data-center environment over I/OAT and compare its performance with non-I/OAT. For all experiments in this section, we used nodes in Testbed 1 (described in Section 4) for the data-center tiers. For the client nodes, we used the nodes in Testbed 2 for most of the experiments. We will notify the readers at appropriate points in this paper when other nodes are used as clients.

5.1 Evaluation Methodology

As mentioned in Section 2, I/OAT is a server architecture geared to improve the receiver-side performance. In other words, I/OAT can be deployed in a data-center environment and client requests coming over the Wide Area Network (WAN) can seamlessly take advantage of I/OAT and get serviced much faster. Further, the communication between the tiers inside the data-center such as proxy tier and application tier can be greatly enhanced using I/OAT thus improving the overall data-center performance and scalability.

We set up a two-tier data-center testbed to determine the performance characteristics of using I/OAT and non-I/OAT. The first tier consists of the front-end proxies. For this, we used the proxy module of Apache 2.2.0. The second tier consists of the web server module of Apache, in order to service static requests. The two tiers in the data-center reside on 1 Gigabit Ethernet network; the clients are connected to the data-center using a 1 Gigabit Ethernet connection.

Typical data-center workloads have a wide range of characteristics. Some workloads may vary from high to low temporal locality, following a Zipf-like distribution [5]. Similarly workloads vary from small documents (e.g., on-line book stores, browsing sites, etc.) to large documents (e.g., download sites, etc.). Further, workloads might contain requests for simple cacheable static or time invariant content or more complex dynamic or time variant content via CGI, PHP, and Java servlets with a back-end database. Due to these varying characteristics of workloads, in this paper, we classify the workloads in three broad categories: (i) Single-file Micro workloads, (ii) Zipf-like workloads and (iii) Dynamic content workloads. However, in this paper, we focus our analysis on the first two categories.

Single-File Micro workloads: This workload contains only a single file. Several clients request the same file multiple times. This workload is used to study the basic performance achievable by the data-center environment without being diluted by other interactions in more complex workloads. We use workloads ranging from 2K to 10K file sizes since this is the average file size for most of the documents in the Internet.

Zipf-like Workloads: It has been well acknowledged in the community that most workloads for data-centers hosting static content, follow a Zipf-like distribution [5]. According to Zipf law, the relative probability of a request for the i'th most popular document is proportional to $1/i^{\alpha}$, where α determines the randomness of file accesses. In our experiments, we vary this α from 0.95 to 0.5, ranging from high temporal locality for documents to low temporal locality.

We also evaluate the performance obtained inside the data-center with proxy servers acting as clients and web-servers with I/OAT capability. As an example, within a data-center, the proxy server sends the requests to the application server which in turn sends the request to the database server. The application server is known to be cpu-intensive due to processing of scripts such as PHP, CGI, servlets, etc. If the application servers have I/OAT capability, clearly, due to reduced CPU utilization, the server not only can accept more number of requests but can also process the pending requests at a faster rate.

For both the scenarios, we use a testbed with one proxy at the first tier and one web-server at the second tier. Each client fires one request at a time and sends another request after getting a reply from the server.

5.2 Experimental Results

In this section, we separate our analysis into two categories. First, we analyze the performance benefits of I/OAT with the two workloads. As mentioned above, we use Testbed 2 to fire requests to the proxy server. Due to the I/OAT capability on the server nodes, we expect the servers to increase the performance.

5.2.1 Analysis with Single File Traces

In this experiment, we used five different traces with varying file size requests. Trace 1 uses an average file size of 2K while Trace 2, 3, 4 and 5 use 4K, 6K, 8K and 10K, respectively. Each client has a 10,000 request subset of different files and reports the TPS (Transactions Per Second) achieved after getting the response from the servers for all the requests. TPS is defined as the total number of transactions serviced per second as seen by the client. Higher TPS values attributes to better server performance.

Figure 8a shows the performance of data-center with varying trace files ranging from 2K to 10K. As shown in the figure, we see that the throughput reported by I/OAT is consistently better than non-I/OAT. It is also to be noted that, for Trace 2 with average file size being 4K, we see a significant throughput improvement for I/OAT. I/OAT reports a TPS which is close to 9754 whereas non-I/OAT reports only 8569 TPS resulting in a 14% overall improvement. For other traces, we see around 5-8% TPS improvement with I/OAT.

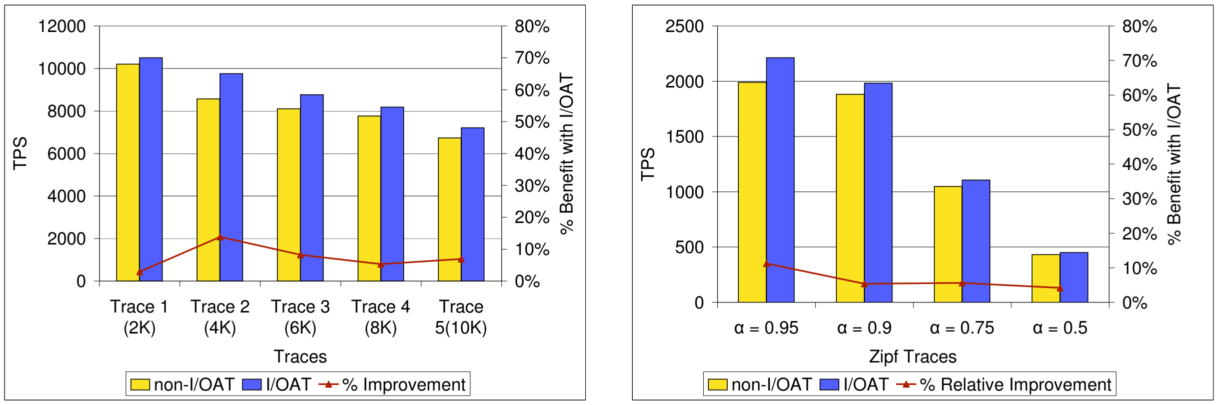


Figure 8. Performance of I/OAT and non-I/OAT (a) Single File Traces (b) Zipf Trace

5.2.2 Analysis with ZipF File Traces

Next, we evaluate the performance of data-center with the zipf-trace. As mentioned earlier, we vary the α values from 0.95 to 0.5 ranging from high temporal locality to low temporal locality. As shown in Figure 8b, I/OAT consistently performs equal or better than non-I/OAT. Further, we note that non-I/OAT achieves close to 1989 TPS whereas I/OAT achieves close to 2212 TPS, i.e., I/OAT achieves up to 11% throughput benefit as compared to non-I/OAT. Also, we measured the CPU utilization for these experiments but the improvement noticed was negligible. However, the throughput improves due to accepting more requests (i.e., reduction in CPU overheads result in greater CPU efficiency).

5.2.3 Analysis with Emulated Clients

Next, we evaluate the performance of I/OAT within a data-center when both the proxy and the web servers have I/OAT capability. In this experiment, the proxy server acts as client in sending the requests to the web servers. Here, we use only the nodes in Testbed 1 for this experiment. Due to reduced CPU usage in proxy nodes using I/OAT, we expect the clients to fire request at a much faster rate and result in higher throughput. Figure 9 shows the performance with I/OAT capability for increasing number of client threads. In this experiment, we fixed the file size to 16K and varied the number of client threads firing the requests from 1 to 256. As shown in the figure, we note that the performance of I/OAT is similar to non-I/OAT from one to sixteen threads. As the number of threads firing the requests increase from 32 to 256, we observe two interesting trends. First, I/OAT throughput performance increases with increasing number of client threads. With 256 client threads, I/OAT achieves close to 14996 TPS whereas non-I/OAT achieves only around 12928, i.e., I/OAT achieves close to 16% throughput improvement compared to non-I/OAT. It is also to be noted that the CPU utilization of I/OAT is consistently lesser than non-I/OAT. In this experiment, we only report the CPU utilization on the client node, since our focus was to capture the client-side benefits when clients have I/OAT capability. We observe that the CPU utilization saturates with non-I/OAT with 64 threads and thus the throughput does not improve any further with increase in the client threads after 64. With I/OAT, we see that the CPU utilization saturates only with 256 client threads, resulting in superior performance. I/OAT not only improves the performance by 16%, but can also support large number of threads (up to a factor of four as compared to non-I/OAT) and deliver superior performance.

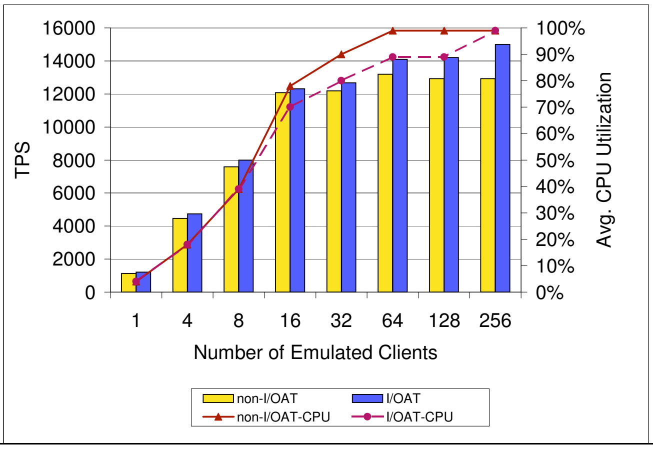


Figure 9. Clients with I/OAT capability using a 16K file trace

6. PVFS Performance Evaluation

In this section, we compare PVFS performance over non-I/OAT and I/OAT with the original PVFS implementation [6]. All experiments in this section were performed on Testbed 1 mentioned in Section 4.

6.1 Evaluation Methodology

It is to be noted that, due to the wide difference between the network and the disk speeds, the effective throughput achieved by PVFS could be limited by that of the disk. However, data used by applications is frequently in server memory, e.g., due to file caching and read-ahead mechanisms. Thus, applications can still benefit from fast networks in such cases. Hence, we designed our experiments based on a memory-resident file system, *ramfs*. These tests are designed to stress the network data transfer independent of any disk activity. In all our experiments, we use Testbed 1 as both the client and server for PVFS.

6.2 Experimental Results

We split our evaluation into two categories. First, we design an experiment that measures the PVFS concurrent read and write bandwidth. Next, we evaluate the file system performance in the presence of multiple streams performing the bandwidth test.

6.2.1 PVFS Concurrent Read and Write on *ramfs*

The test program used for concurrent read and write performance is *pvfs-test*, which is included in the PVFS release package. We followed the same test method as described in [6]. In all tests, each compute node simultaneously reads or writes a single contiguous region of size $2N$ Mbytes, where N is the number of I/O nodes in use. For example, if the number of I/O nodes is four, the request size is then 8 Mbytes. Each compute node will access 2 Mbytes of data from each I/O node. It is to be noted that since I/OAT is a receiver-side optimization, we report the average CPU utilization at the client-side while performing a read operation and report the CPU utilization at the server-side while performing the write operation.

Figure 10a shows the read performance I/OAT and non-I/OAT with different number of I/O servers. With non-I/OAT, the bandwidth increases from 361 MB/s to 649 MB/s with increase in compute nodes. For I/OAT, the bandwidth increases from 360 MB/s to 731 MB/s with increasing compute nodes, i.e., I/OAT achieves 12% increase in throughput as compared to non-I/OAT with six compute nodes. Further, the results are consistent with the micro-benchmark levels evaluations seen in Section 4. In terms of CPU utilization, we see that I/OAT achieves close 15% improvement as compared to non-I/OAT with six compute node experiment. We see similar trends when we decrease the number of I/O servers to five as shown in Figure 10b. However, the improvement seen both in terms of throughput and CPU utilization are much lesser as compared to six I/O server configuration.

Figure 11a shows the write performance with I/OAT and non-I/OAT. Again, we see similar trends as seen with the PVFS read performance. With non-I/OAT, the write performance increases from 464 MB/s to 697 MB/s. For I/OAT, the performance increases from 460 MB/s to 750 MB/s with increasing number of compute nodes, i.e., I/OAT achieves a 8% throughput benefit as compared to non-I/OAT. Further, I/OAT achieves close to 7% CPU benefit as compared to non-I/OAT. We see similar trends when we decrease the number of I/O servers as shown in Figure 11b.

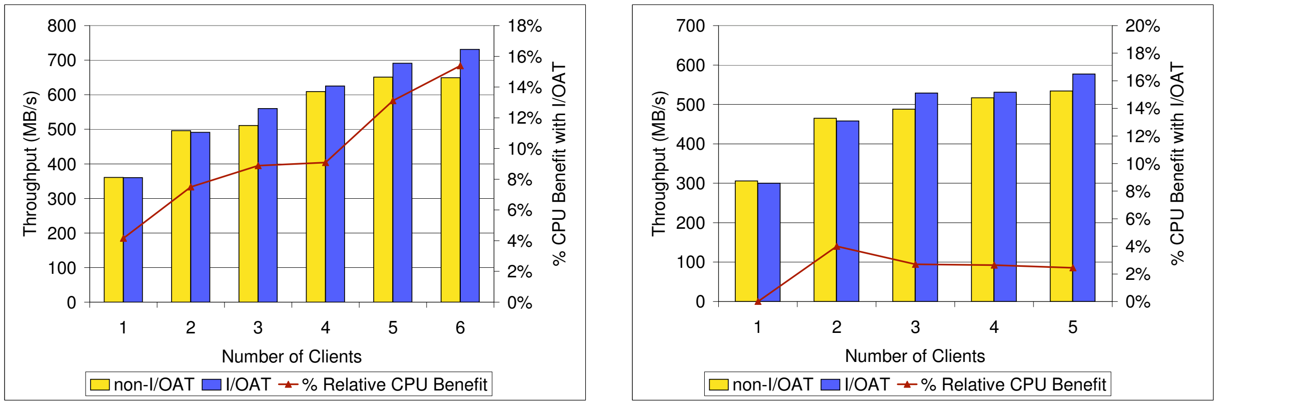


Figure 10. PVFS Concurrent Read Performance: (a) 6 I/O Servers (b) 5 I/O Servers

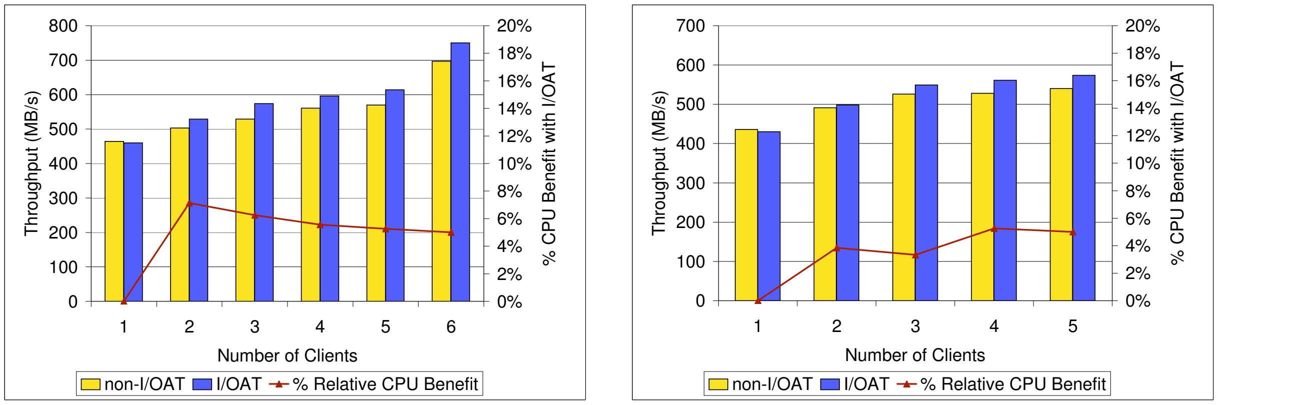


Figure 11. PVFS Concurrent Write Performance: (a) 6 I/O Servers (b) 5 I/O Servers

6.2.2 PVFS read performance with multiple streams

In this experiment, we emulate a scenario where multiple clients accesses the file system. We increase the number of clients from 1 to 64 and study the behavior of concurrent read performance in PVFS. As shown in Figure 12, we observe that the performance of I/OAT is either equal or more than the performance achievable by non-I/OAT. It is to be noted that we report the CPU utilization numbers measured on the client-side which also has the I/OAT capability. Due to the fact that the client with I/OAT can receive the data at a much faster rate, the clients can also fire PVFS read requests at a faster rate as compared to non-I/OAT. As a result, we observe that the CPU on the client-side is around 10-12% higher than the non-I/OAT case.

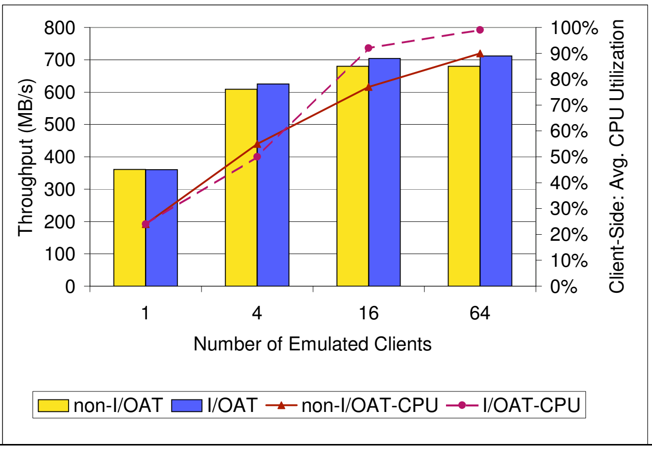


Figure 12. Multi-Stream PVFS Read Performance

In summary, PVFS with I/OAT has a 12% improvement for concurrent reads and 8% improvement for concurrent writes as compared to PVFS with non-I/OAT. In terms of CPU utilization, PVFS with I/OAT has a 15% benefit for concurrent reads and 7% for concurrent writes as compared to PVFS with non-I/OAT.

7. Discussion

As mentioned earlier, I/OAT improves the network performance and lowers CPU utilization through its features such as the asynchronous copy engine, split-headers, etc. The asynchronous copy engine can also be used for other user applications to improve the performance of memory operations and also to improve the communication performance between two processes within the same node. Since the copy engine can operate in parallel with the host CPU, applications can achieve significant benefits by overlapping useful computation with memory operations. Especially, with the introduction of chip-level multiprocessing (CMP), per-core memory bandwidth reduces drastically and thus, increasing the need for such asynchronous memory operations as a memory-latency hiding mechanism. Mohit, et. al., have proposed soft-timer techniques to reduce the receiver-side processing [4]. I/OAT can co-exist with this technology to further reduce the receiver-side overheads and improve the performance.

On the other hand, I/OAT requires special network adapters to support the split-header and the multiple receive queue feature. In addition, the host system should have an in-built copy engine to support the asynchronous copy feature. Also, the memory controller uses physical addresses and hence, the pages cannot be swapped during a copy operation. As a result, the physical pages need to be pinned before initiating the copy operation. Due to the page-pinning requirement, the usefulness of the copy engine becomes questionable if the pinning cost exceeds the copy cost.

8. Concluding Remarks and Future Work

I/O Acceleration technology (I/OAT) developed by Intel is a set of features particularly designed to reduce the system overheads on the receiver-side. This paper studies the benefits of the I/OAT technology by extensive evaluations through micro-benchmarks as well as evaluations on two different application domains: (1) A multi-tier Data-Center environment and (2) A Parallel Virtual File System (PVFS). Our micro-benchmark results show that I/OAT results in 38% lower overall CPU utilization in comparison with the traditional communication. Due to this reduced CPU utilization, I/OAT also delivers better performance and increased network bandwidth. Our experimental results with data-centers and file systems reveal that I/OAT can improve the total number of transactions processed by 14% and throughput by 12%, respectively. In addition, I/OAT can sustain larger number of concurrent threads (up to a factor of 4 as compared to non-I/OAT) in a data-center environment, thus increasing the scalability of the servers.

We are currently working on two broad aspects with respect to I/OAT. Firstly, we are looking for modifications in applications to get the absolute benefit of I/OAT more closer to the end-applications. Secondly, we are trying to provide an asynchronous memory copy operation to user applications. Though this involves some amount of overhead (such as context switches, user page locking, etc), asynchronous memory copy can help applications in using the CPU cycles intelligently.

9. Acknowledgments

We would like to thank Intel Corporation for providing us access to I/OAT-based systems and providing us the kernel patch which includes all their features.

References

[1] Accelerating High-Speed Networking with Intel I/O Acceleration Technology. http://www.intel.com/technology/ioacceleration/306517.pdf.

[2] Increasing network speeds: Technology@intel. http://www.intel.com/technology/magazine/communications/intel-ioat-0305.htm.

[3] Intel I/O Acceleration Technology. http://www.intel.com/technology/ioacceleration/306484.pdf.

[4] ARON, M., AND DRUSCHEL, P. Soft timers: efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems 18*, 3 (2000), 197–228.

[5] Breslau, L., Cao, P., Fan, L., Phillips, G., and Shenker, S. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM (1)* (1999), pp. 126–134.

[6] Carns, P. H., Ligon III, W. B., Ross, R. B., and Thakur, R. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference* (Atlanta, GA, 2000), USENIX Association, pp. 317–327.

[7] Chandra, A., Gong, W., and Shenoy, P. Dynamic Resource Allocation for Shared Data Centers Using Online Measurements. In *Proceedings of ACM Sigmetrics 2003, San Diego, CA* (June 2003).

[8] Chase, J., Gallatin, A., and Yocum, K. End system optimizations for high-speed TCP. *IEEE Communications Magazine 39*, 4 (2001), 68–74.

[9] Chase, J. S., Anderson, D. C., Thakar, P. N., Vahdat, A., and Doyle, R. P. Managing energy and server resources in hosting centres. In *Symposium on Operating Systems Principles* (2001).

[10] Ching, A., Choudhary, A., keng Liao, W., Ross, R., and Gropp, W. Noncontiguous I/O through PVFS. In *Proceedings of the IEEE International Conference on Cluster Computing* (2002).

[11] Clark, J. R. D. D., Jacobson, V., and Salwen, H. An analysis of TCP processing overhead, 1989.

[12] et al, A. F. TCP performance analysis revisited. In *IEEE International Symposium on Performance Analysis of Software and Systems* (March 2003).

[13] Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. Cluster-based scalable network services. In *Symposium on Operating Systems Principles* (1997).

[14] Gover, A., and Leech, C. Accelerating network receiver processing. http://linux.inet.hr/files/ols2005/grover-reprint.pdf.

[15] Makineni, S., and Iyer, R. Architectural characterization of TCP/IP packet processing on the Pentium M microprocessor. In *High Performance Computer Architecture, HPCA-10* (2004).

[16] Regnier, G., Makineni, S., Illikkal, R., Iyer, R., Minturn, D., Huggahalli, R., Newell, D., Cline, L., and Foong, A. TCP Onloading for Data Center Servers. In *IEEE Computer* (Nov 2004).

[17] Reumann, J., Mehra, A., Shin, K. G., and Kandlur, D. Virtual services: A new abstraction for server consolidation. In *In Proceedings of the USENIX 2000 Technical Conferece* (June 2000).

[18] Saito, Y., Bershad, B. N., and Levy, H. M. Manageability, availability and performance in porcupine: A highly scalable, cluster-based mail service. In *Symposium on Operating Systems Principles* (1999).

[19] Shah, H. V., Minturn, D. B., Foong, A., McAlpine, G. L., Madukkarumukumana, R. S., and Regnier, G. J. CSP: A Novel System Architecture for Scalable Internet and Communication Services. In *the Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems* (San Francisco, CA, March 2001), pp. pages 61–72.

[20] Shen, K., Tang, H., Yang, T., and Chu, L. Integrated Resource Management for Cluster-based Internet Services. In *Proceedings of Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)* (Boston, MA, December 2002), pp. 225–238.

[21] Thakur, R., Gropp, W., and Lusk, E. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems* (May 1999), ACM Press, pp. 23–32.