Designing Efficient Asynchronous Memory Operations Using Hardware Copy Engine: A Case Study with I/OAT*

K. Vaidyanathan W. Huang L. Chai D. K. Panda
Computer Science and Engineering,
The Ohio State University,
{vaidyana, huanwei, chail, panda}@cse.ohio-state.edu

Abstract

Memory copies for bulk data transport incur large overheads due to CPU stalling, small register-size data movement, etc. Intel's I/O Acceleration Technology offers an asynchronous memory copy engine in kernel space which alleviates such overheads. In this paper, we propose a set of designs for asynchronous memory operations in user space for both single process (as an offloaded memcpy()*) and IPC using the copy engine. We analyze our design based on overlap efficiency, performance and cache utilization. Our microbenchmark results show that using the copy engine for performing memory copies can achieve close to 87% overlap with computation. Further, the copy engine improves the copy latency of bulk memory data transfers by 50% and avoids cache pollution effects. With the emergence of multi-core architectures, the support for asynchronous memory operations holds a lot of promise in reducing the gap between the memory and processor performance.*

1 Introduction

Several applications in the fields of biomedical informatics, satellite weather image analysis, engineering and sciences not only demand for large number of compute cycles but also for higher memory and network performance. To address the compute cycle requirement, large number of processors are getting added to current-generation systems. Emerging new technologies such as Multi-Core Processors (also known as Chip-level Multiprocessing or CMP) provide several cores on a single node. Currently dual-core architectures (two cores per die) are widely available from various industry leaders including Intel, AMD, Sun (with up to 8 cores) and IBM. Similarly, network performance has also been increasing at a tremendous rate with the introduction of high-performance networks such as InfiniBand [1], 10 Gigabit Ethernet (10 GigE) [7], etc. Today, several industries are taking the next step in high-speed networking with multi ten-gigabit networks such as the Mellanox 20-Gigabit IBA DDR adapters, IBM 30-Gigabit IBA adapters, etc.

On the other hand, memory performance has not been improving at a significant pace resulting in a huge gap between the processor and memory performance. The limited memory bandwidth is often addressed as the major performance degradation factor for many scientific applications. Several memory block operations such as copy, compare, move, etc., are performed by the host CPU leading to an inefficient use of the host compute cycles. In addition, such operations also affect the caching hierarchy since the host CPU fetches the data onto cache, thereby, evicting some other valuable resources in cache. The problem gets even worse with the introduction of multi-core systems since several cores can concurrently access the memory leading to memory contention issues, CPU stalling issues, etc. Due to several of the issues mentioned above, the ability to overlap computation and memory operation as a memory latency-hiding mechanism becomes critical for masking the gap between processor and memory performance.

Direct Memory Access (DMA) has been traditionally used to transfer the data directly from the host memory to any input/output device without the host CPU intervention. Several networks such as InfiniBand and Myrinet provide a zero-copy data transfer support. However, such solutions are mainly used for transferring data from one node to another. Researchers in the past have attempted to use DMA engines to accelerate bulk data movement within a node. Many of these approaches have not entirely succeeded due to huge DMA startup costs, completion notification costs

*This research is supported in part by DOE grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; NSF grants #CNS-0403342 and #CNS-0509452; grants from Intel, Mellanox, Cisco systems, Linux Networx and Sun Microsystems; and equipment donations from Intel, Mellanox, AMD, Apple, Appro, Dell, Microway, PathScale, IBM, SilverStorm and Sun Microsystems.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

and other performance-related issues. Recently, Intel's I/O Acceleration Technology (I/OAT) [6, 9, 11] introduced an Asynchronous DMA Copy Engine (ADCE) in kernel space that has direct access to main memory to improve performance and reduce the overheads mentioned above.

In this paper, we propose a set of designs for asynchronous memory operations in user space for both single process (as an offloaded *memcpy()*) and Inter-Process Communication (IPC) using the copy engine. In order to achieve this, we design and implement a kernel module that provides a set of interfaces for userspace applications. In our design, we efficiently handle IPC synchronization, memory alignment, scheduling across DMA channels and avoiding user buffer pinning costs. We analyze our design based on overlap efficiency, performance and cache utilization.

Our experimental results show that using ADCE for memory copies can achieve close to 87% overlap with computation. Further, it improves the latency of large memory data transfers by 50% and increases the IPC bandwidth for large message sizes by a factor of two. Also, the copy engine assists in avoiding 30% performance degradation due to cache pollution effects.

2 Background

In this section we provide a brief motivation for using copy engines in bulk data transfers and describe the architecture of I/OAT Copy Engine.

2.1 Motivations for Copy Offload Engine

Figure 1 illustrates the basic architecture of a copy execution using a CPU vs using a DMA copy engine. As mentioned in [12], utilizing a copy engine for bulk data transfer offers several benefits:

1. *Reduction in CPU Resources and Better Performance:* Memory copies are usually implemented as a series of load and store instructions through registers. Data is fetched onto the cache and then onto the registers. Typically, the CPU performs the copy by register size which is 32 or 64 bit long. On the other hand, using a copy engine, memory copies can be done at a faster rate (close to block sizes) since it directly operates with main memory. Further, the load and store instructions used in CPU-based copies may end up occupying the CPU resources, limiting the CPU to not look far ahead in the instruction window. Copy engines can help in freeing up CPU resources so that other useful instructions can be executed.

2. *Computation-Memory Copy Overlap:* Since the memory-to-memory copy operation can be performed without host CPU intervention using an asynchronous copy engine, we can achieve better overlap with memory copies. This is similar to DMA operation where data is transfered directly between the memory and device which is commonly used by networks such as InfiniBand, Quadrics, etc.

3. *Avoiding Cache Pollution Effects:* Large memory copies can pollute the cache significantly. Unless the source or destination buffers are needed by the application, allocating this buffer in the cache may result in polluting the cache as it can evict other valuable resources in the cache. As mentioned in [12], cluster applications such as web servers do not touch the data immediately even after completing the memory copy. Using a copy engine in this case, results in avoiding any cache pollution as it can directly perform the copy without getting the data onto the cache.

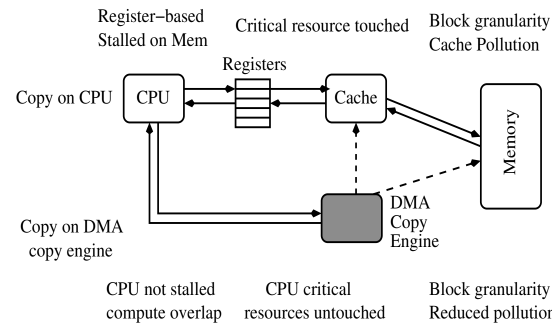

Figure 1. Copy execution on CPU vs Copy Engines [12]

2.2 I/OAT Copy Engine Architecture

I/OAT [6] offloads the data copy operation from the CPU with the addition of an asynchronous DMA copy engine (ADCE). ADCE is implemented as a PCI-enumerated device in the chipset and has multiple independent DMA channels with direct access to main memory. When the processor requests a block memory operation from the engine, it can then asynchronously perform the data transfer with no host processor intervention. When the engine completes a copy, it can optionally generate an interrupt.

Though ADCE 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. Hence, memory operations should be broken up into individual page transfers. Secondly, memory copies whose source and destination overlap should be carefully handled. Applications need to schedule such operations in an appropriate order so as to preserve the semantics of the operation. Lastly, 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.

3 Proposed Design

In this section, we describe our proposed design in supporting asynchronous memory operation for userspace applications. We first describe the design for single process and IPC. Later, we discuss how efficiently we handle issues such as synchronization, memory alignment, user buffer pinning, etc.

3.1 Basic Design for User-Space Applications

Currently, Intel supports several interfaces in kernel space for copying data from a source page/buffer to a destination page/buffer. These interfaces are asynchronous and the copy is not guaranteed to be completed when the function returns. These interfaces return a non-negative cookie value on success, which is used to check for completion of a particular memory operation. It is necessary to wait on another function to wait for the copies to complete.

A memory copy operation typically involves three operands: (i) a source address, (ii) a destination address and (iii) number of bytes to be copied. For user-space applications, the source and destination addresses are virtual addresses. However, as mentioned in Section 2, the DMA copy engine can only understand physical addresses. The first step in performing the copy is to translate the virtual address to physical addresses. For various reasons related to security and protection, this is done at the kernel space. Once we get the physical address, we also need to make sure that the physical pages that are mapped to the user application does not get swapped onto the disk while the copy engine performs the data transfer. Hence, we need to lock the pages in memory before initiating the DMA and unlock the pages after the completion of the copy operation, if required. We use the *get_user_pages()* function in the kernel space to lock the user pages.

Table 1. Basic ADCE Interface

Operation	Description
adma_copy(src, dst, len)	Blocking copy routine
adma_icopy(src, dst, len)	Non-blocking copy routine
adma_check_copy(cookie)	(Non-blocking) check for completion
adma_wait_copy(cookie)	(Blocking) wait for completion

In order for the userspace applications to use the copy engine, we propose the addition of the following interfaces, as shown in Table 1. The *adma_icopy* operation helps in initiating the copy and returns a cookie which can be used later to check for completion while the *adma_copy_check* operation helps in checking if the corresponding memory operation has completed. The *adma_copy_wait* operation waits for the corresponding memory operation to complete and the *adma_copy* operation is a blocking version which uses the copy engine and does not return until the copy finishes.

3.2 Design for Inter-Process Communication (IPC)

In addition to offloaded memory operations within a single process, applications also require support for exchanging messages across different processes in a single node. Typically, parallel applications which run on different processors use such mechanisms for inter process communication. As shown in Figure 2, there are many ways of performing inter process communication. The most common way followed is the user space shared memory based approach. In this approach, processes A and B create a shared memory region. Process A copies the source data onto the shared memory and process B copies this shared memory segment to its destination. Clearly, this approach involves an extra copy. Several MPI implementations use this approach [4]. As mentioned in Section 2, this approach also occupies some CPU resources. Another approach is the NIC-based loop back approach wherein network device can DMA the data from the source to the destination. The third approach [8] is to map the user buffer in kernel space and use the standard copy operation in kernel to avoid an extra copy incurred by user space shared memory approach. In this paper, we propose a fourth approach, which is utilizing the DMA copy engine to perform the copy. Such an approach does not incur any extra copies, not touch many CPU critical resources and also avoids any cache pollution effects.

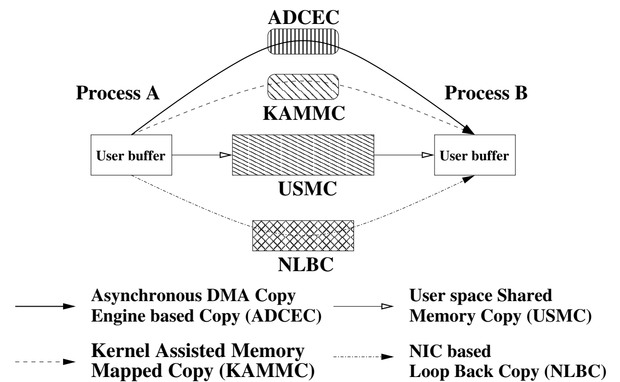

Figure 2. Different IPC Mechanisms

We support the following user interface, as shown in Table 2, for applications to exchange messages across different processes. The *adma_read* and *adma_write* operations

read and write data onto another process and *adma_iread* and *adma_iwrite* operations initiate the data transfer. However, due to the presence of two different processes, synchronization becomes a bottleneck for performance. Data transfer cannot be initiated unless both the processes have posted their buffers for data transfer. We describe the challenges in handling these scenarios in Section 3.3. The *adma_icheck* operation checks whether the memory operation has completed and the *adma_wait* operation waits till the memory operation completes.

Table 2. ADCE Interface for IPC

Operation	Description
adma_iread(fd, addr, len)	Non-blocking read routine
adma_iwrite(fd, addr, len)	Non-blocking write routine
adma_read(fd, addr, len)	Blocking read routine
adma_write(fd, addr, len)	Blocking write routine
adma_check(cookie)	(Non-blocking) check for read/write completion
adma_wait(cookie)	(Blocking) Wait for read/write completion

Figure 3 shows the mechanism by which we support IPCs. Our design can be easily integrated with the pipe or socket semantics. Currently, we support the socket semantics for establishing the connection between different processes. Once the connection is made, processes can use the set of interfaces mentioned above for utilizing the copy engine. Let us consider two connected processes (A and B). If process A needs to send data to process B, process A makes a request to the kernel (Step 1). In step 2, the kernel locks the user page and adds the entry to a list of cached virtual to physical mappings. The kernel then makes an entry to a list of pending read and write requests. At this time, if process B posts its read buffer (Step 4), the kernel locks the user page and caches the page mapping (Step 5). The kernel searches the list to find the matching write request (Step 6). Since the write buffer is already posted, it initiates the DMA copy (Step 7). Process A waits for the completion of operation (Step 8) by issuing a request to the kernel. The kernel first makes sure that the corresponding buffers are posted by waiting on a semaphore (Step 9a). This semaphore is initially in a locked state and released when both the read and write buffers match. Steps 10-11 are similar to Steps 8-9.

3.3 Handling IPC Synchronization

Since we have two different processes performing communication using the copy engine, synchronization becomes a critical issue before initiating the DMA transaction. For example, consider processes A and B wanting to

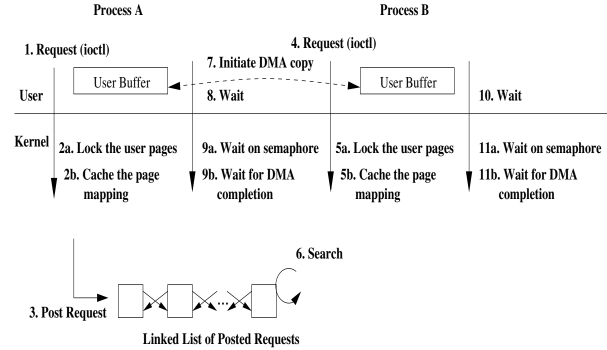

Figure 3. IPC using DMA copy engine

communicate a buffer of size 1 KB. We need to handle the following cases making sure that latency, progress and CPU utilization do not get affected significantly. Case 1: Process A posts the write buffer and waits for the operation to finish. Then process B posts an *adma_iread* operation. Case 2: Process B posts the read buffer and waits for the operation to finish. Then process A posts an *adma_iwrite* operation. Case 3: Both processes A and B post their respective buffers before performing the wait operation. To address these cases, we use a binary semaphore in our implementation. For Case 1, we queue the request posted by Process A during the write request and we allow the process to wait on the semaphore during the wait operation. When Process B posts a read buffer, the DMA is initiated and immediately process A is woken up by releasing the semaphore. Process A then waits on the DMA copy to finish and the control is given back to the user process. For Case 2, a similar approach is followed except that Process A wakes up process B after process A posts the corresponding write buffer. In Case 3, both processes A and B see a matching request posted and thus do not wait on any semaphore and directly check for DMA completion. All three cases avoid unnecessary polling and the control is released immediately after the buffers are posted so that DMA completion is checked immediately leading to better notification.

3.4 Handling Memory Alignment

Another issue is the memory alignment problem associated with source and destination buffers. Since the copy engine operates with main memory, the performance of the copy operation can be enhanced if the memory is page-aligned. For example, lets say that the source address starts at offset 0 and the destination address at 2K. If we assume the page size to be 4 KB, then we can only schedule a maximum of 2 KB copy since the copy length is required to be within the page-boundary, leading 2000 such operations if we assume a 4 MB data transfer. On the other hand, if the addresses were page-aligned, we only need 1000 such op-

erations. In the worst case, we may end up issuing copies for very small messages (<100 bytes) for several iterations. Clearly, by making the addresses page-aligned, we can save on the number of copy operations and more importantly avoid issuing very small data transfers using the copy engine.

3.5 Handling User Buffer Locking

As mentioned in Section 2, the copy engine deals with physical addresses as it directly operates on main memory. To avoid swapping of user pages to the disk during a copy operation, it is mandatory that the kernel locks the user buffers before initiating the DMA copy. Usually this locking cost is quite large, in the order of μs contributing significantly to the total time required for data transfer. To reduce this cost, we lock the buffers initially and do not release the locked buffers even after the completion of data transfer. For subsequent data transfers, if the same user buffer is reused, we can avoid the locking costs and directly use the physical address that maps to the virtual address. However, if the application uses *malloc()* and *free()* calls, the kernel module needs to be aware of such changes and appropriately release these buffers.

3.6 Handling Scheduling across DMA channels

Several applications can use the DMA engine simultaneously. Hence, it is possible that a small memory operation is queued behind several large memory operations. Due to the fact that we have several DMA channels, scheduling these memory operations on appropriate channels becomes a challenging task. Currently, we use a simple approach of using the channels in a round-robin manner and schedule the memory operations. We plan to extend this work on using dedicated channels and adaptive schemes in future.

4 Experimental Results

We ran our experiments on a dual dual-core Intel 3.46 GHz processors and 2 MB L2 cache system with SuperMicro X7DB8+ motherboards which include 64-bit 133 MHz PCI-X interfaces. The machine is connected with an Intel PRO 1000Mbit adapter. We used the Linux RedHat AS 4 operating system and kernel version 2.6.9-30.

4.1 Latency and Bandwidth Performance

Figure 4a shows the performance of copy latency using CPU and ADCE for small message sizes. In this experiment, both source and destination buffers fit in the cache. For CPU-based copy operation, we measure the *memcpy* operation of *libc* library and average it over several iterations. This is indicated as *libc memcpy (CPU-based)* line in the figure. For ADCE, we use the *adma_icopy* operation followed by the *adma_wait* operation and measure the time to finish both the operations.

As shown in Figure 4a, we see that CPU-based approach performs well for all message sizes. This is mainly due to the cache size which is 2 MB. Since both source and destination buffers can fit in the cache, CPU-based approach performs better. Figure 4b show the performance of copy latency for small messages when source and destination buffers are not in the cache. In this experiment, we use two 64 MB buffers as source and destination. After every copy operation, we move the source and destination pointers by message size, so that memory copy always uses different buffers. We repeat this for a large number of iterations and ensure that the buffers are not in the cache. As observed in the figure, we see that ADCE using four channels performs better from 16 KB. As mentioned earlier, since we are using buffers that are not in the cache, for ADCE, we also incur penalties with huge pinning costs for every copy operation. As a result, we see that the performance is little worse compared to the previous experiment where the buffers are in the cache. Also, the performance of ADCE with one channel gets better after 256 KB message size. However, as shown in Figure 4c, we see that the performance of ADCE for large message sizes is significantly better than the CPU-based approach. For 4 MB message size, we observe that ADCE with four channels results in 50% improvement in latency as compared to the CPU-based approach. Also, we observe that ADCE using four channels achieves better latency compared to ADCE with one channel.

The bandwidth performance of copy operation is shown in Figure 5. In this experiment, we post a window of *adma_icopy* operations (128 in our case) and wait for these memory operations to finish. We repeat this experiment for several iterations and report the bandwidth. For CPU-based approach, we use the *libc memcpy* instead of the *adma_icopy* operation. As shown in Figure 5, for message sizes till 1 MB, the CPU-based approach yields a maximum bandwidth of 9189 MB/s. This is mainly due to the caching effect since the copy happens inside the cache. For message sizes greater than 1 MB, we observe a huge drop in bandwidth for CPU-based approach achieving close to 1443 MB/s. However, ADCE with four channels achieves a peak bandwidth of 2912 MB/s, almost double the bandwidth achieved by CPU-based approach. ADCE with one channel achieves close 2048 MB/s.

4.2 Overlap of Computation and Memory Operation

In this section, we evaluate the ability of ADCE to effectively overlap memory copy process and computation. To

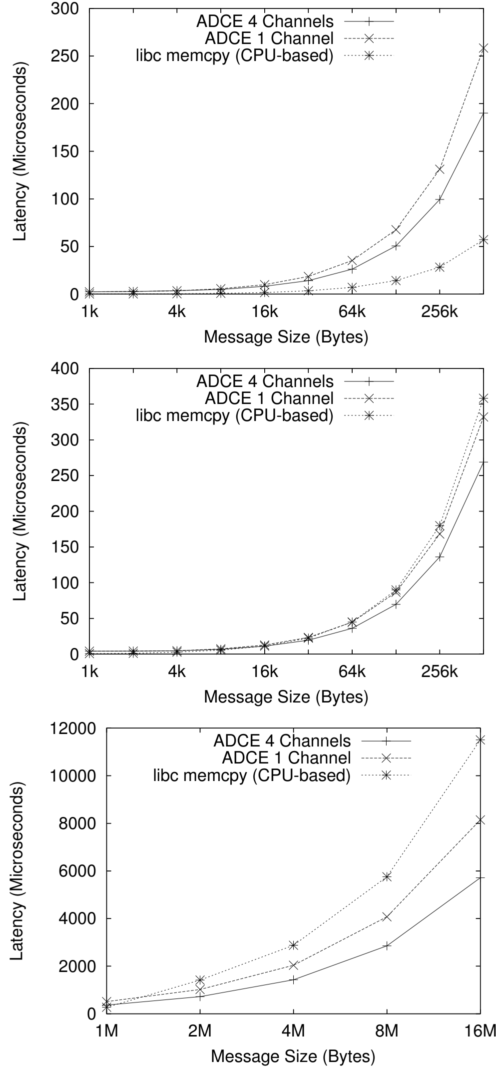

Figure 4. Latency: (a) small message hot-cache, (b) small message cold-cache and (c) large message cold-cache

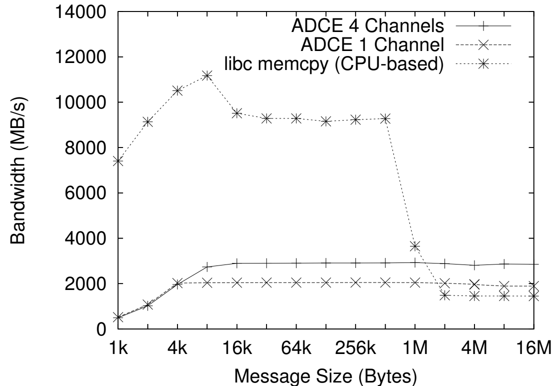

Figure 5. Bandwidth

carry this evaluation we design an overlap benchmark. For a certain message size, the benchmark first estimates the latency of blocking memory copy T_{copy} (*adma_icopy* operation immediately followed by *adma_wait* operation). To test the overlap efficiency, the benchmark initiates an asynchronous memory copy (*adma_icopy*) followed by a certain amount of computation which at least takes time $T_{compute}$ $> T_{copy}$, and finally waits for the completion (*adma_wait*). The total time is recorded as T_{total}. If the memory copy is totally overlapped by computation, we should have T_{total} = $T_{compute}$. If the memory copy is not overlapped, we should have T_{total} = T_{copy} + $T_{compute}$. The actual measured value will be in between, and we define overlap as:

Overlap = (T_{copy} + $T_{compute}$ - T_{total}) / T_{copy}

Based on the above definition, the value of *overlap* will be between 0 (non-overlap) and 1 (totally overlapped). A value close to 1 indicates a higher overlap efficiency. Figure 6a illustrates the overlap efficiency we measured. As we can see, CPU-based copy using *memcpy* is blocking, thus we always get an overlap efficiency of 0. By using ADCE for large size memory copies, we are able to achieve up to 0.92 (92%) and 0.87 (87%) overlap using one and four channels, respectively. For ADCE with four channels, we check the completion across four channels and thus it results in lesser overlap compared to ADCE with one channel case. For smaller sizes, the overlap efficiency is small due to DMA startup overheads. We see similar trend in overlap efficiency when the source and destination buffers are not in the cache as shown in Figure 6b. However, the actual percentages seen are much lower. We explain the reason for such lower percentages in the section below.

4.3 Asynchronous Memory Copy Overheads

In order to understand the low overlap efficiency observed in the previous section, we measure the split-up/overhead of ADCE. Figure 7 shows the split-up overhead of ADCE using four channels. In this experiment, we ran the copy latency test with source and destination buffers not in the cache and measure the overhead of user/kernel transition, pinning of user buffer, DMA startup and completion. We observe that the pinning cost occupies a significant fraction of the total overhead. For small message sizes, we see that all four overheads contribute equally towards the latency and there is very little room for overlap. For larger message sizes, we see that the pinning cost and DMA startup cost occupies 30% and 7%, respectively. The remaining time is overlapped with the computation (62%).

4.4 Cache Pollution Effects

In this section, we measure the effect of cache pollution with applications. We design the experiment in the

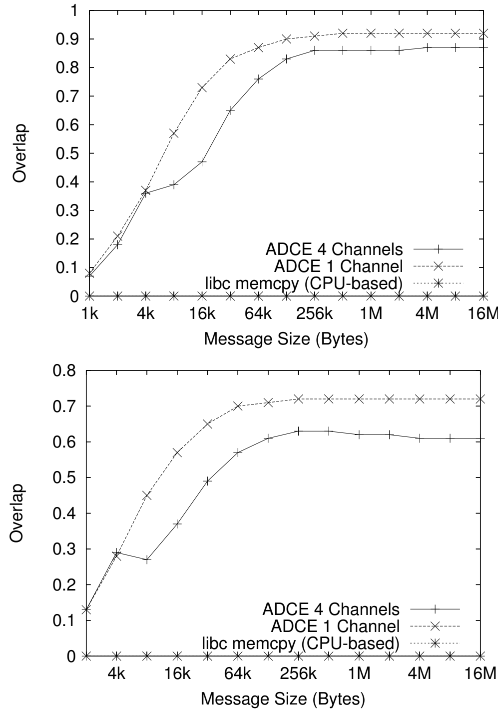

Figure 6. Computation-Memory Copy Overlap: (a) hot-cache and (b) cold-cache

following way. We perform a large memory copy operation and perform a column-wise access of a small memory buffer which can fit in the cache. Figure 8 shows the access time for various memory sizes. We measure the access time without the memory copy and report it as *access w/o copy* and for remaining cases, we perform the memory copy using CPU and ADCE. As shown in figure, the access time after performing the copy using ADCE does not change with the normal access time. However, CPU-based approach increases the access time by 30% due to cache eviction. Since ADCE operates directly on main memory, ADCE avoids cache pollution effects. As a result, the access latency after using ADCE does not change. However, CPU-based approach evicts some of the entries in the cache resulting in an increase in access time latency.

4.5 IPC Latency and Bandwidth

Figure 9a shows the IPC latency for ADCE based copy (ADCEC), NIC loopback based copy (NLBC) and Kernel-assisted memory mapped based copy (KAMMC). For 4 MB message size, we see that ADCEC achieves close to 2954 μs whereas KAMMC and NLBC achieve close to 5803 and 14333 μs, respectively. Further, for increasing message sizes, the performance of ADCEC is much better than KAMMC and NLBC, respectively.

Figure 9b shows the IPC bandwidth with ADCEC, KAMMC and NLBC. Since the buffers can fit in the cache, we observe that the performance of KAMMC is better than ADCEC and NLBC till 256 KB message size achieving close to 8191 MB/s. However, for message sizes greater than 1 MB, we see that ADCEC achieves 2932 MB/s whereas KAMMC and NLBC achieve only 1438 MB/s and 720 MB/s, respectively.

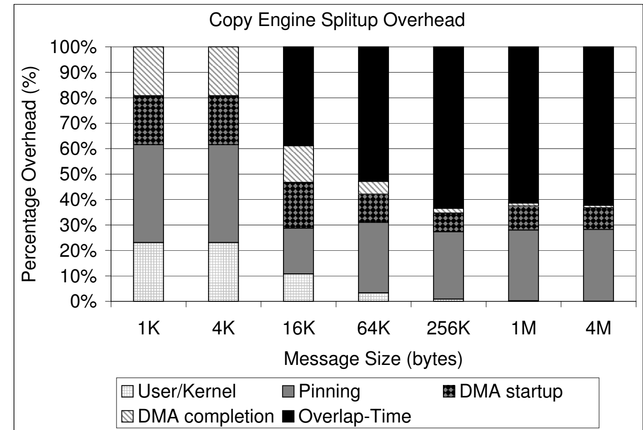

Figure 7. Asynchronous Copy Overhead

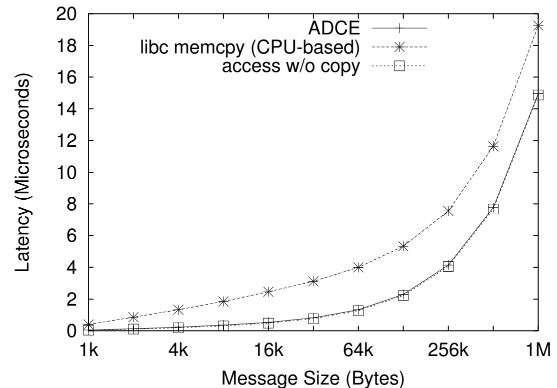

Figure 8. Cache Pollution Effects

5 Discussion and Related Work

Emerging technologies such as multi-core processors (also known as Chip-level Multiprocess) provide several cores on a single node. Since several of these cores access memory at the same time, memory contention issues become very common in such environments. Also, due to the gap between memory and processor performance, contention issues will only get worse with more and more cores. The inability to perform useful computation by stalling on

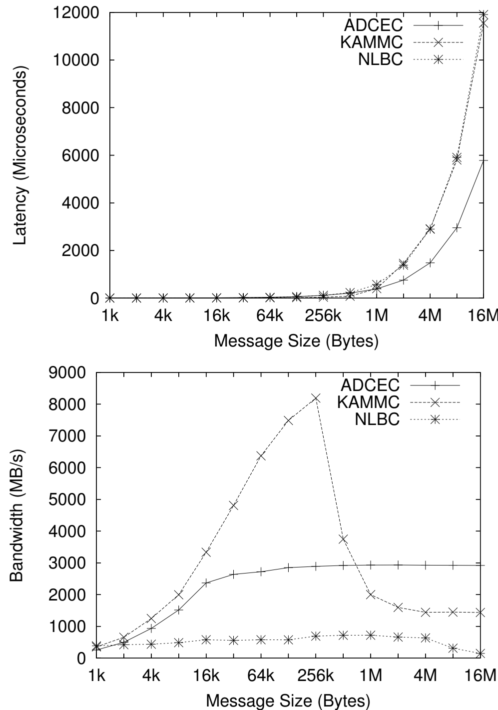

Figure 9. Inter Process Communication: (a) Latency and (b) Bandwidth

memory operations is often considered the bottleneck in many of these environments. ADCE helps in alleviating this bottleneck by its asynchronous feature, thus allowing the cores to perform useful computation during a memory copy operation.

Researchers have proposed several solutions for asynchronous memory operations in the past. User-level DMA [10, 2] deal with providing asynchronous DMA explicitly at the user space. Zhao et al [12] talk about hardware support for handling bulk data movement. Calhoun's thesis [3] proposes the need for dedicated memory controller copy engine and centralized handling of memory operations to improve performance. However, many of these solutions are simulation-based. Ciaccio [5] proposed the use of self-connected network devices for offloading memory copies. Though this approach can provide an asynchronous memory copy feature, it has a lot of performance-related issues. I/OAT [6] offers an asynchronous DMA copy engine (ADCE) which improves the copy performance with very little startup costs. In this paper, we use this hardware for supporting asynchronous memory operations.

6 Conclusions and Future Work

Intel's I/O Acceleration Technology offers an asynchronous memory copy engine in kernel space that alleviates copy overheads such as CPU stalling, small register-size data movements, etc. In this paper, we proposed a set of designs for asynchronous memory operations in user space for both single process (as an offloaded *memcpy()*) and IPC using the copy engine. We analyzed our design based on overlap efficiency, performance and cache utilization. Our microbenchmark results showed that using the copy engine for performing memory copies can achieve close to 87% overlap with computation. Further, the copy latency of bulk memory data transfers is improved by 50%.

We plan to analyze the impact of the copy engine with several MPI-based applications and also other distributed applications such as web servers as a part of future work. We also propose to improve our design so that applications can achieve close to 100% overlap with computation.

References

[1] InfiniBand Trade Association. http://www.infinibandta.com.

[2] M. A. Blumrich, C. Dubnicki, E. W. Felten, and K. Li. Protected, user-level DMA for the SHRIMP network interface. In *Proc. of the 2nd IEEE Symp. on High-Performance Computer Architecture (HPCA-2)*, 1996.

[3] M. Calhoun. Characterization of block memory operations. In *Masters Thesis, Rice University*, 2006.

[4] L. Chai, A. Hartono, and D. K. Panda. Designing high performance and scalable mpi intra-node communication support for clusters. In *IEEE International Conference on Cluster Computing*, 2006.

[5] G. Ciaccio. Using a self-connected gigabit ethernet adapter as a memcpy() low-overhead engine for mpi. In *Euro PVM/MPI*, 2003.

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

[7] J. Hurwitz and W. Feng. End-to-End Performance of 10-Gigabit Ethernet on Commodity Systems. *IEEE Micro*, January 2004.

[8] H. W. Jin, S. Sur, L. Chai, and D. K. Panda. Limic: Support for high-performance mpi intra-node communication on linux cluster. In *International Conference on Parallel Processing (ICPP)*, 2005.

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

[10] E. P. Markatos and M. G. H. Katevenis. User-level DMA without operating system kernel modification. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture, (HPCA)*, 1997.

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

[12] L. Zhao, R. Iyer, S. Makineni, L. Bhuyan, and D. Newell. Hardware support for bulk data movement in server platforms. In *Proceedings of International Conference on Computer Design*, 2005.