Efficient Asynchronous Memory Copy Operations on Multi-Core Systems and I/OAT

K. Vaidyanathan, L. Chai, W. Huang, D. K. Panda

Network-Based Computing Laboratory, The Ohio State University
2015 Neil Ave., Columbus, OH 43210 USA
{ vaidyana, chail, haunwei, panda }@cse.ohio-state.edu

Abstract

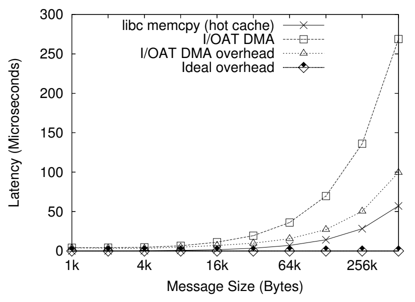
Bulk memory copies incur large overheads such as CPU stalling (i.e., no overlap of computation with memory copy operation), small register-size data movement, cache pollution, etc. Asynchronous copy engines introduced by Intel's I/O Acceleration Technology help in alleviating these overheads by offloading the memory copy operations using several DMA channels. However, the startup overheads associated with these copy engines such as pinning the application buffers, posting the descriptors and checking for completion notifications, limit their overlap capability. In this paper, we propose two schemes to provide complete overlap of memory copy operation with computation by dedicating the critical tasks to a single core in a multi-core system. In the first scheme, MCI (Multi-Core with I/OAT), we offload *the memory copy operation to the copy engine and* onload *the startup overheads to the dedicated core. For systems without any hardware copy engine support, we propose a second scheme, MCNI (Multi-Core with No I/OAT) that* onloads *the memory copy operation to the dedicated core. We further propose a mechanism for an application-transparent asynchronous memory copy operation using memory protection. We analyze our schemes based on overlap efficiency, performance and associated overheads using several micro-benchmarks and applications. Our microbenchmark results show that memory copy operations can be significantly overlapped (up to 100%) with computation using the MCI and MCNI schemes. Evaluation with MPI-based applications such as IS-B and PSTSWM-small using the MCNI scheme show up to 4% and 5% improvement, respectively, as compared to traditional implementations. Evaluations with data-centers using the MCI scheme show up to 37% improvement compared to the traditional implementation. Our evaluations with* gzip *SPEC benchmark using application-transparent asynchronous memory copy show a lot of potential to use such mechanisms in several application domains.*

I. Introduction

In recent years, there has been a rapid growth of compute-intensive as well as memory-intensive applications in the domains of medical informatics, genomics, satellite weather processing, etc. These applications not only demand large compute cycles but also higher memory performance. Emerging trends in processor technology has led to Multi-Core Processors (also known as Chip-level Multiprocessing or CMP) which provide large number of cores on a single node, thus increasing the processing capability of current-generation systems. On the other hand, over the years, improvements in memory performance have not matched the improvements in processor speed. 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. Further, if the application data is not in the cache, the host CPU ends up waiting for the data to be fetched to the cache or the registers before performing the memory block operation thus leading to CPU stalling issues. The problem is further magnified in multi-core systems since several cores can concurrently access the memory leading to memory contention issues. Due to several of the issues mentioned above, the ability to overlap computation and memory copy operation as a memory copy latency-hiding mechanism becomes critical for masking the gap between processor and memory performance.

Recently, Intel's I/O Acceleration Technology (I/OAT) [10] introduced an asynchronous Direct Memory Access (DMA) copy engine within the chip which has direct access to main memory and relatively lesser DMA startup and completion overheads. Figure 1(a) shows the latency of memory copy operation using the I/OAT's DMA copy engine and the associated overheads for different message sizes as mentioned in [16]. Due to the fact that the copy engine is known to give better performance for large memory copies, we focus only on small and medium message sizes. Also, we report the performance of traditional *libc memcpy* when the application buffers are resident in the cache (referred as *libc memcpy (hot-cache)* in the figure). As shown in Figure 1(a), we observe that the traditional *libc memcpy* outperforms the I/OAT DMA engine's performance if the application buffers are in the cache. Further, we observe that the DMA startup overhead associated with the copy engine is much higher than the memory copy time (*libc memcpy hot-cache*), thus removing the benefits of asynchronous memory copy provided by these copy engines. In this paper, we propose to mask this overhead completely (referred as the ideal overhead in the figure) for applications to get true benefits of the copy engine even for small and medium message sizes.

Researchers in the past have looked at different ways of providing memory copy operations as shown in Figure 1(b). The shaded boxes show the components that already exist



(a) Memory Copy Latency

	No I/OAT	I/OAT
Single Core	SCNI Single−Core with No I/OAT	SCI Single−Core with I/OAT
Multiple Cores	MCNI Multi−Core with No I/OAT	MCI Multi−Core with I/OAT

(b) Different Mechanisms for Memory Copies

Fig. 1. Motivation for Using Asynchronous Memory Copy Operations

today. For single-core systems with no hardware copy engine support, traditional *libc memcpy* is used for memory copies. We refer to this scheme as SCNI (Single-Core with No I/OAT). However, if the system has an I/OAT support, applications can *offload* the memory copy to the copy engine. We refer to this scheme as SCI (Single-Core with I/OAT). As multi-core systems are emerging, it opens up new ways to design and implement memory copy operations. Currently, there is no study that has explored the impact of multi-core systems in designing efficient memory copy operations. We take on this challenge and introduce two new schemes as shown in white boxes. In the first scheme, MCI (Multi-Core with I/OAT), we *offload* the memory copy operation to the copy engine and *onload* the startup overheads to a dedicated core. For systems without any hardware copy engine support, we propose a second scheme, MCNI (Multi-Core with No I/OAT) that *onloads* the memory copy operation to a dedicated core. We further propose a mechanism for an application-transparent asynchronous memory copy operations using memory protection.

We evaluate our proposed schemes on multi-core and I/OAT based systems and attempt to bring out the benefits and issues associated with these schemes in terms of performance, overlap capability and overheads using several microbenchmarks and applications. Our microbenchmark-level evaluations using MCI and MCNI schemes show that the memory copy operations can be significantly overlapped (up to 100%) with computation. Evaluation with MPI-based applications such as IS-B and PSTSWM-small using the MCNI scheme show up to 4% and 5% improvement, respectively, as compared to the traditional implementation. Evaluations with data-centers using the MCI scheme show up to 37% improvement as compared to the traditional implementation. In addition, our design and evaluation for an application-transparent asynchronous memory copy using the SPEC benchmarks shows a lot of promise (up to 10% improvement) for many applications that uses memory copies.

II. Motivation and Background

In this section, we first provide a brief motivation for using copy engines in memory copy operations and describe the architecture of I/OAT copy engine. Next, we briefly discuss our previously proposed scheme for offloading memory copies using a copy engine (SCI scheme).

A. Motivations for Copy Offload Engine

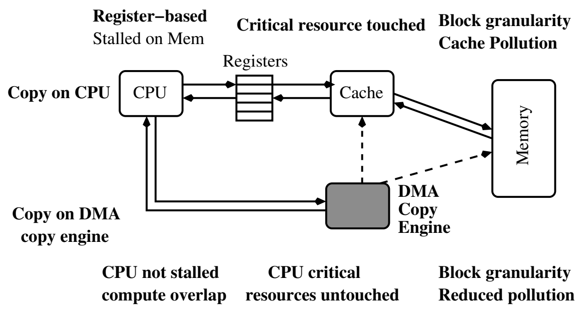
The basic architecture of copy execution using a CPU or a DMA copy engine is shown in Figure 2(a). As mentioned in [18], utilizing a copy engine for memory copies helps in *reducing the CPU resources* occupied and offers *better performance*. CPU-based memory copies are limited by the register-size data movement since the copy operation is implemented as a series of load and store instructions through registers. On the other hand, copy engines can perform memory copies at a faster rate (L2 block size). Further, the load and store instructions used in CPU-based memory copies may end up occupying the CPU resources, thus 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. Since the memory-to-memory copy operation can be performed without host CPU intervention, applications can also *achieve better overlap* with memory copies. Using a copy engine for bulk memory copies also results in *avoiding cache pollution effects* as it can directly perform the copy without getting the data onto the cache.

B. I/OAT Copy Engine Architecture

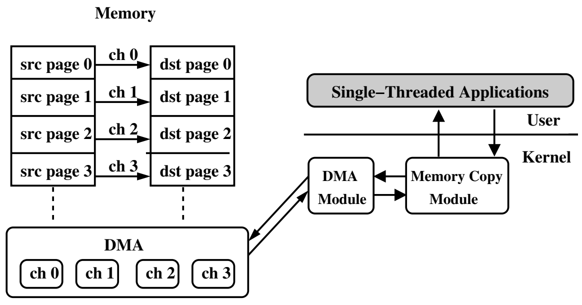
I/O Acceleration Technology [10] offloads the memory copy operation using an asynchronous DMA copy engine. The copy engine is implemented as a PCI-enumerated device in the chipset and has multiple independent DMA channels with direct access to main memory. When the processor requests a block memory copy 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. I/OAT supports several interfaces in kernel space for copying data from a source page/buffer to a destination page/buffer.

C. Single-Core with I/OAT (SCI) Scheme

In our previous work [16], we proposed a SCI scheme that *offloads* the memory copy operation to the I/OAT's hardware copy engine and used a kernel module to expose the



(a) Copy Execution on CPU vs Copy Engines (Courtesy [18])



(b) SCI Scheme

Fig. 2. Copy Engine Architecture and SCI Scheme

features of the hardware copy engine to user applications. User applications communicate with the kernel module (referred as memory copy module in Figure 2(b)) for offloading the copy operation. The kernel module initiates the memory copy operation across each of the DMA channels. On a completion notification request from the user, the kernel module checks the progress of memory copy operation and informs the application accordingly. In addition, tasks such as pinning the application buffers, posting the descriptors, releasing the buffers are also handled by the kernel module. The SCI scheme also supports page caching mechanism to avoid pinning of application buffers in the critical path. In this mechanism, the kernel module caches the virtual to physical page mappings after locking the application buffers. Once the memory copy operation completes, the kernel module does not unlock the application buffers to avoid the pinning cost if the same application buffer is reused later.

III. Proposed Design

In this section, we first describe the overheads associated with the SCI scheme. Next, we present two new schemes that address the limitations of SCI scheme.

A. Overheads of SCI Scheme

Though the SCI scheme offers several benefits such as performance improvement, cache pollution avoidance, overlap capability, it has the following overheads.

Copy Engine Overheads: As mentioned in our previous work [16], in order to perform a memory copy operation using the copy engine, we need to post a descriptor to a channel specifying the source and destination buffer and the size of the data transfer. Due to the presence of multiple channels in the copy engine, we incur the cost for posting the descriptors across each DMA channel. After the copy operation is initiated, we also need to check for the completion of memory copy operation across all the channels. Though the hardware copy engine provides a mechanism to avoid this cost by sending an interrupt after the completion, this may not be suitable for latency-sensitive applications.

Page Locking Overheads: Further, due to the fact that the hardware copy engine can understand only physical addresses, it is mandatory that the application buffers are locked/pinned while the copy operation is in progress. However, page locking cost can be significant since the kernel needs to pin each and every page involved in the memory copy operation.

Context Switch Overheads: Due to the fact that the copy engine is accessible only in kernel space, a context switch occurs for every copy engine related operation performed by the user application. This cost is especially large if multiple applications try to access the copy engine at the same time while the copy operation is still in progress resulting in several context switch penalties.

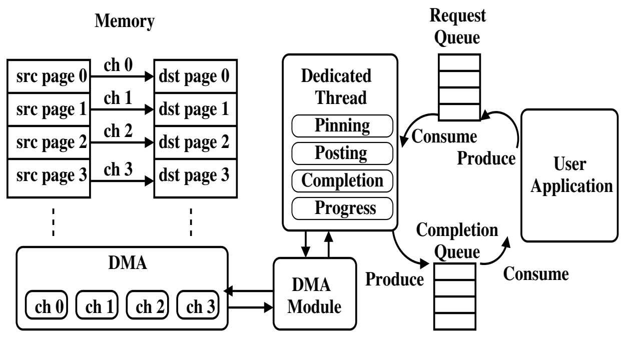
Synchronization Overheads: Several applications can access the hardware copy engine simultaneously and hence the copy engine resources need to be locked for protection.

B. Multi-Core with I/OAT (MCI) Scheme

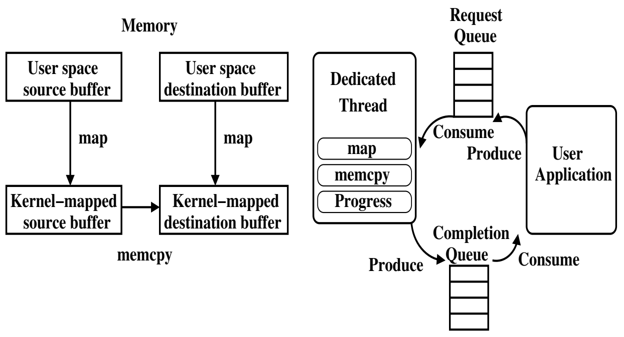
While the SCI scheme helps user applications to *offload* memory copy operations, several critical operations still remain in the critical path. In this section, we propose a novel scheme to alleviate these overheads.

1) Basic Design: In our design, we propose a scheme that takes advantage of the copy engine and multi-core systems to avoid the overheads in the critical path. Specifically, we *offload* the copy operation to the hardware copy engine and *onload* the tasks that fall in the critical path to another core or a processor so that applications can exploit complete overlap of memory operation with computation.

Figure 3(a) shows the various components of the proposed scheme. Since the copy engine is accessible only in the kernel space, we dedicate a kernel thread to handle all copy engine related tasks and allow user applications to communicate with the dedicated thread to perform the copy operation. The dedicated thread also maintains a list of incomplete memory copy requests and attempts to make progress for these initiated requests. Apart from servicing multiple user applications, the dedicated thread also handles tasks such as locking the application buffers, posting the descriptors for each user request on appropriate channels, checking for device completions, releasing the locked application buffers after completion events. Since the critical tasks are onloaded to this dedicated thread, the user application is free to execute other



(a) MCI Scheme



(b) MCNI Scheme

Fig. 3. Asynchronous Memory Copy Operations

computation or even execute other memory operations while the copy operation is still in progress thus allowing complete overlap of memory copy operation and computation.

2) Avoiding Context Switch Overhead: In order to avoid the context switch overhead between the application process and the dedicated thread, we use a similar mechanism proposed by [1], [13], [4]. This mechanism memory maps a region from a user space to kernel space so that both the application and the kernel thread can access this common memory region at the same time. The memory region is divided into a set of request and completion queues. The request queue is used to submit memory copy requests by the application. The dedicated thread constantly looks at the request queue to process new copy requests. Similarly, the completion queue is used to notify the completion of copy operations by the kernel thread. The applications constantly look at the completion queue for completion notifications.

3) Handling Locking and Synchronization: In the SCI scheme, since the kernel module exposes a set of interfaces for applications, several kernel instances can be spawned if multiple applications need to access the copy engine. This increases the requirement of locking the shared resources and careful management for supporting concurrency. However, in the MCI scheme, since the dedicated thread handles all tasks for multiple applications, it avoids the need for locking the resources and becomes easier for managing the shared resources.

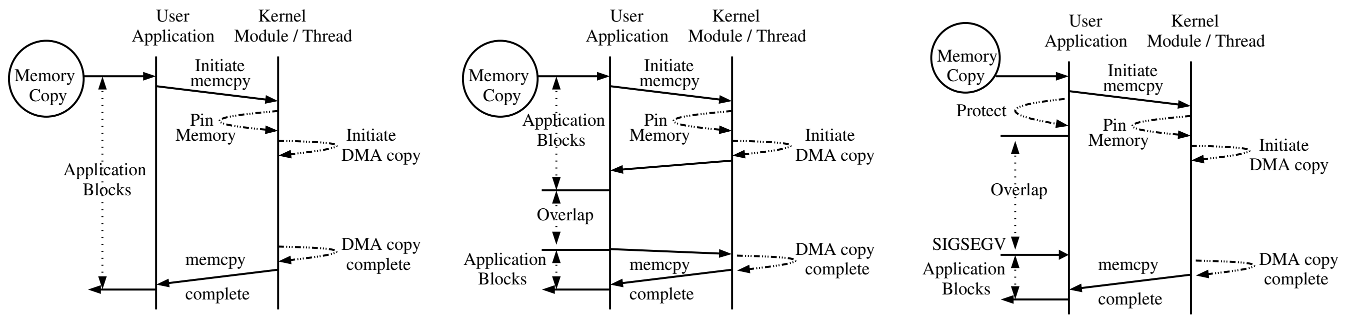
C. Multi-Core with No I/OAT (MCNI) Scheme

The MCI scheme mentioned in the previous section is applicable only for multi-core systems with the copy engine support. However, there are several multi-core systems without copy engine support. In order to provide asynchronous memory copy operations for such systems, we propose a MCNI scheme (Multi-Core systems with No I/OAT) that *onloads* the memory copy operation to another processor or a core in the system. This scheme is similar to the MCI scheme. In this scheme, as shown in Figure 3(b), we dedicate a kernel thread to handle all memory copy operations, thus relieving the main application thread to perform computation.

The MCNI scheme takes help from the operating system kernel to perform memory copy operations. The dedicated thread should have access to the physical pages pointed by the application's virtual address to perform copy operations from one process to another process. This is done through memory mapping mechanism [12], [1] that maps a part of other processes address space into its own address space. After the memory mapping process, the dedicated thread can access the mapped area as its own memory region and perform the copy operation. Since the memory mapping is a costly function, our design also supports for caching the memory mapped pages so that future copy requests with the same source or the destination buffers can avoid the mapping cost and perform the copy operation faster.

D. Application-Transparent Asynchronism

The main idea of application-transparent asynchronism is to avoid blocking the application while the memory copy operation is still in progress. With the asynchronous memory copy interface, the application can explicitly initiate the copy operation and wait for its completion using another function. However, several applications are written with the blocking routine (*memcpy*, *bcopy*), which assumes that the data is copied once the function finishes. Further, the semantics of the *memcpy* operation assumes that the buffer is free to be used after the completion of *memcpy* operation. To transparently provide asynchronous capabilities for such operations, two goals need to be met: (i) the interface should not change; the application can still use the blocking *memcpy* routine and (ii) the application can assume the blocking semantics, i.e., once the control returns to the application, it can read or write the buffer. Here, we use a similar approach proposed by [5]. In our approach, we memory-protect the user buffer (thus disallow the application from accessing it) and perform the copy operations. After the copy operation completes, we release the memory protection so that the applications can access both the source and destination buffers. Since in a *memcpy* operation the source does not get modified, we allow read accesses to the source buffer. However, the destination address gets modified during a copy operation and hence we



(a) Blocking Memory Copy (b) Application–aware Asynchronous Memory Copy (c) Application Transparent Asynchronous Memory Copy

Fig. 4. Different Mechanisms for Asynchronous Memory Copy Operations

do not allow accesses to this memory region during the copy operation. We further optimize the performance for successive memory copy operations by checking if the multiple pages overlap. If they do overlap and if the first memory copy is still in progress, we do not release the protection for the overlapped pages of memory copy, since they will be protected by the second memory copy operation. Further, during a memory copy operation, we check the progress of previous memory copy operations and accordingly release the protection. If the application does not modify the destination buffer for sufficiently long time, then the application will realize only the page protection time, which is considerably lesser compared to the copy operation for large message transfers. Currently, we do not support application-transparent memory copies for overlapping source and destination buffers.

Figure 4 illustrates the designs of memory copy operation using the three different approaches. As shown in Figure 4(a), in all three schemes (SCI, MCI and MCNI), though the memory copy operation is performed in the background, the application blocks for every memory copy operation to finish before performing any other computation. Figure 4(b) shows the impact of an application-aware memory copy operation which needs modification in order to benefit from asynchronism. Figure 4(c) shows the design of an application-transparent asynchronous memory copy operation. As mentioned before, we memory protect buffers before initiating the memory copy operation and return the control to the application. If the application attempts to access the destination buffer or modify the source buffer, a page fault is generated due to page protection. This results in a SIGSEGV signal for the application which is handled by our helper module. In this case, we block for all pending memory copy operations to complete and release the protection appropriately.

E. Extensions to MPI Middleware

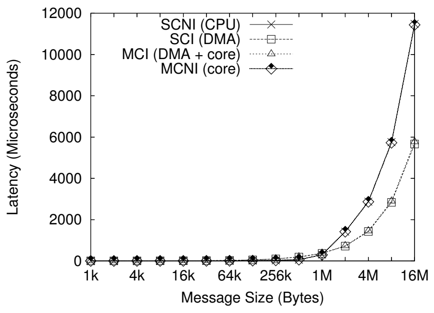
Message Passing Interface (MPI) is the *de facto* standard in high performance computing. In this section, we describe our extension to MPI intra-node communication implementation to take advantage of the asynchronous memory copy operations. Our design is based on MVAPICH, which is a high performance MPI library over InfiniBand [14]. The intra-node communication in MVAPICH is achieved by attaching all the processes to a user space shared memory region. The sender copies messages into the shared memory region and the receiver copies messages out of it. Therefore, at least two copies are involved in this process. Small messages are transferred using an *eager protocol* while large messages are transferred using *rendezvous* protocol. The detailed design of MVAPICH intra-node communication is described in [8]. In our design, small messages are still transferred through the user space shared memory region. For large messages, we use the shared memory region for handshake messages, and use asynchronous memory copy operations for transferring the data. The protocol is described as below:

- Step 1: The sender sends a *request_to_send* message.
- Step 2: The receiver replies with an *ok_to_send* message when it sees the *request_to_send* message and a matching MPI_recv operation is posted.
- Step 3: The receiver then posts its receive request by initiating a non-blocking IPC read request to the kernel for performing asynchronous memory copy operations, and places this request into a *pending_recv_queue*.
- Step 4: When the sender receives the *ok_to_send* message, it posts its send request by initiating a non-blocking IPC write request to the kernel for performing asynchronous memory copy operations, and places this request into a *pending_send_queue*.
- Step 5: When the MPI application tries to make progress, the sender and the receiver check the completion of the pending operations by initiating a non-blocking IPC check request to the kernel to check for completion and inform the upper layer about the completion of the operations.

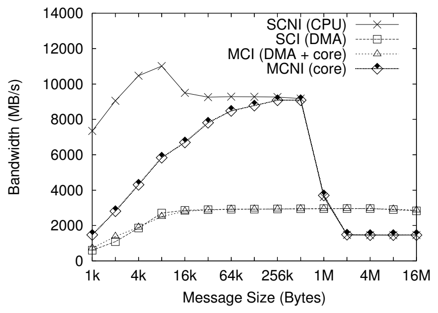
IV. Experimental Results

We ran our experiments on a dual dual-core Intel 3.46 GHz processors with 2 MB L2 cache system using 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 the kernel version 2.6.20 for all our experiments.

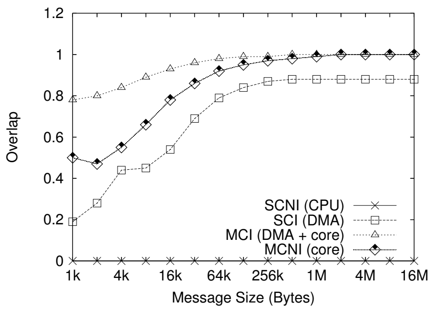
Our experiments are organized as follows. First, we analyze our schemes in terms of performance, overlap capability and the associated overheads. Next, we evaluate the impact of these schemes with MPI-based applications and SPEC benchmarks such as *gzip* and in data-center environments.



(a) Latency



(b) Bandwidth



(c) Overlap

Fig. 5. Basic Micro-Benchmark Performance with Page Caching

A. Micro-benchmarks

In this section, we evaluate the schemes in terms of latency and bandwidth performance, overlap efficiency and its associated overheads.

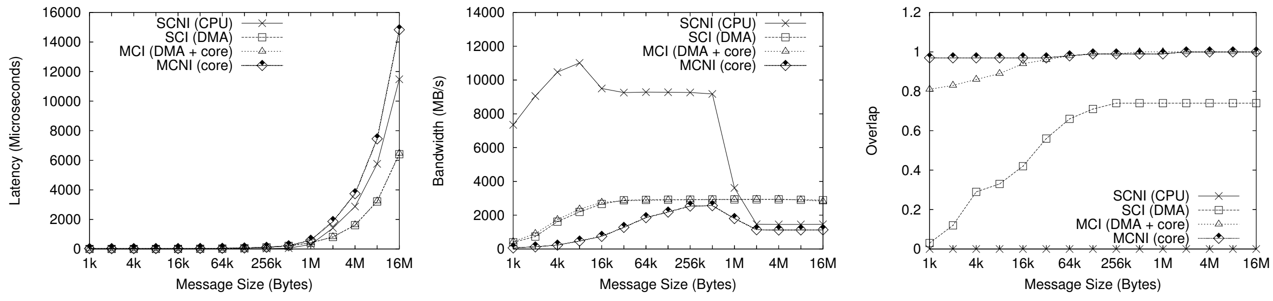
1) Basic Performance with Page Caching: Figure 5 shows the basic performance of memory copy operation using the page caching mechanism, as mentioned in Section II-C. Figure 5(a) shows the latency of all four schemes. For the SCNI scheme, we perform several *memcpy* operation using the *libc* library and average it over several iterations. For the SCI, MCI and MCNI schemes, we initiate the memory copy operation and wait for the completion notification before initiating the next copy operation. As shown in Figure 5(a), we see that the latency of both SCNI and MCNI schemes for message sizes greater than 2 MB is significantly worse compared to the performance of the SCI and MCI schemes. Since the cache size is only 2 MB, both the SCNI and MCNI schemes perform the copy operation in memory using the CPU which is limited by small register-size copy operations. However, for the SCI and MCI schemes, since the copy operation is performed by the DMA channels directly in memory, it is not limited by the register size. Hence, we see a performance improvement of up to a factor of two for the SCI and MCI schemes in comparison with the SCNI and MCNI schemes. For message sizes less than 1 MB, since the buffers can fit in cache, the performance of the SCNI and MCNI schemes is significantly better than the SCI and MCI schemes.

The bandwidth performance of memory copy operation is shown in Figure 5(b). In this experiment, we initiate a window of copy operations and wait for these copy operations to finish. We repeat this experiment for several iterations and report the bandwidth. As shown in Figure 5(b), we see that the bandwidth performance of the SCNI and MCNI schemes for message sizes less than 1 MB is significantly better than the bandwidth performance of the SCI and MCI schemes due to caching effects. The peak bandwidth for the SCNI and MCNI schemes achieved are 11014 MB/s and 9087 MB/s, respectively. However, for message sizes greater than 2 MB, we see that the bandwidth of the SCNI and MCNI schemes drops to 1461 MB/s and 1463 MB/s since the buffers are accessed in memory. On the other hand, the SCI and MCI schemes report a peak bandwidth of up to 2958 MB/s and 2954 MB/s, respectively.

To measure the overlap efficiency, we perform the overlap benchmark as mentioned in [16]. First, the benchmark estimates the copy latency (T_{copy}) by performing a blocking version of memory copy operations. Next, the benchmark initiates the asynchronous memory copy followed by a certain amount of computation ($T_{compute} > T_{copy}$) which takes at least the blocking copy latency and finally waits for the copy completion. 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 (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 5(c) shows the overlap efficiency of all four schemes in performing memory copy operations and computations. For the SCNI scheme, since we only have a blocking version of memory copy (*libc memcpy*), we see that the overlap efficiency is zero. In the SCI scheme, since the copy operation is offloaded, we observe that it can achieve up to 0.88 (88%) overlap efficiency. However, for small message sizes, we see that the overlap efficiency is quite low. On the other hand, we observe that the MCI scheme can achieve up to 1.00 (100%) overlap efficiency for large messages and up to 0.78 (78%) overlap efficiency even for small messages. We also observe that MCNI scheme achieves up to 1.00 (100%) overlap efficiency for large messages and up to 0.5 (50%) overlap efficiency for small messages.

2) Performance without Page Caching: In this section, we measure the performance of our schemes without the page caching mechanism.

Figure 6(a) shows the latency of all four schemes without page caching. For the MCNI scheme, we observe that the latency is significantly worse reaching up to 14829 μs for 16 MB message size. Further, we see that the SCI and MCI schemes report a latency of 6414 μs and 6468 μs, respectively. As mentioned earlier, since the application buffers are not



(a) Latency (b) Bandwidth (c) Overlap

Fig. 6. Basic Micro-Benchmark Performance without Page Caching

cached, every memory copy operation using the SCI, MCI and MCNI schemes incur a page locking cost, thus increasing the latency. However, the SCNI scheme does not show any degradation since the scheme does not depend on the underlying page caching mechanism. The bandwidth performance without page caching mechanism is shown in the Figure 6(b). Since the locking costs can be pipelined with several memory copy operations, we do not observe any degradation in bandwidth for the SCI and MCI schemes. However, for the MCNI scheme, due to huge mapping costs, we see a drop in bandwidth. Figure 6(c) shows the overlap efficiency of all four schemes without page caching mechanism. For the SCI scheme, since the startup overheads fall in the critical path, we observe that it can achieve only 0.74 (74%) overlap efficiency for large message transfers. However, we see that both MCNI and MCI schemes show up to 1.00 (100%) overlap efficiency.

3) Split-up Overhead of SCI, MCI and MCNI Schemes: To understand the low overlap efficiency observed in the previous section, we measure the split-up overhead of the three schemes, namely the SCI, MCI and MCNI schemes. Figure 7 shows the split-up overhead of using memory copy operations with the SCI, MCI and MCNI schemes. For small message sizes, we see that the pinning costs, startup overheads and completion notifications consume considerable amount of time reducing the overlap efficiency for the SCI scheme. Even for large message sizes, we observe that the pinning costs and DMA startup overheads occupy close to 18% and 7%, respectively. However, for the MCI and MCNI schemes, we observe that the only overhead is posting the request and checking for completions through memory transactions in the response queue, thus resulting in almost 100% overlap efficiency.

B. Evaluations with MPI

In this section, we present the MPI-level micro-benchmarks and application performance.

Figure 8 shows the MPI level intra-node latency and bandwidth. In our testbed, we found that the optimal threshold to switch from eager to rendezvous protocol as 32 KB, thus messages smaller than 32 KB are still transferred through shared memory in all the schemes. Therefore, we only show

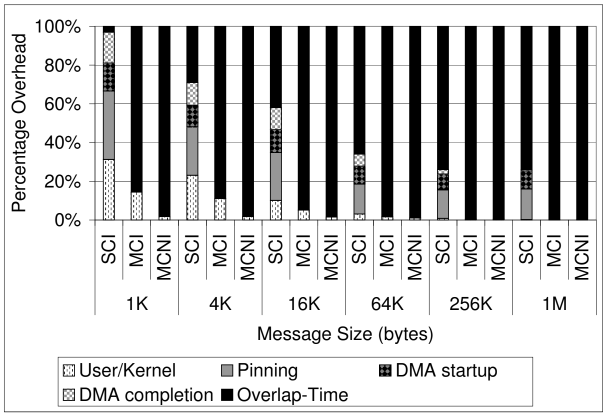
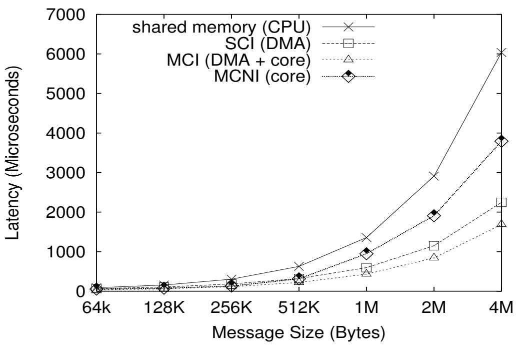


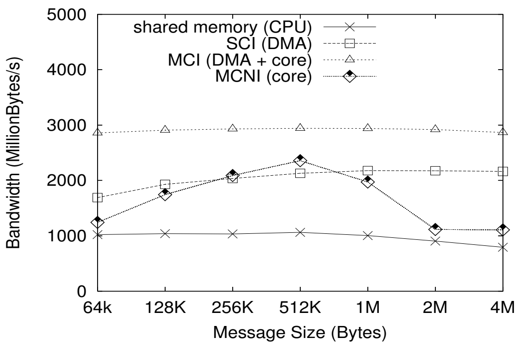
Fig. 7. Overheads of SCI, MCI and MCNI Schemes

the results larger than 32 KB. As shown in Figure 8(a), we see that all the asynchronous memory copy schemes are able to achieve better performance than the shared memory scheme. For example, the MCNI scheme improves the latency by up to 37% compared to the shared memory scheme (SCNI) and the MCI scheme improves the latency by up to 24% compared to SCI scheme. Among the three asynchronous memory copy schemes, the MCI scheme performs the best. The reasons being, compared to the SCI scheme, the MCI scheme onloads the operations in the critical path to another thread; and compared to the MCNI scheme, the MCI scheme uses the DMA engine which performs the copy more efficiently than the CPU-based approach. The bandwidth performance shown in Figure 8(b) reveals the same trend. The MCI scheme shows up to 24% improvement in bandwidth as compared to the SCI scheme. It is to be noted that the bandwidth of both the shared memory scheme and the MCNI scheme drops at 2 MB since the cache size is only 2 MB.

We use IS in NAS parallel benchmarks [11] and PSTSWM [2] for our application level performance evaluations. The normalized execution time is shown in Figure 9. The results were taken on a single node. For MCI and MCNI schemes, we use one of the cores in the system as the dedicated core. Hence, we only show the performance of shared memory (CPU) and SCI scheme for four processes, since our experimental testbed has only four cores. As shown in Figure 9, we see that the MCI scheme improves the



(a) Latency



(b) Bandwidth

Fig. 8. MPI-level Latency and Bandwidth

TABLE I. Application Message Size Distribution

Message Size	0 - 32KB	32KB - 1MB	1MB - 64MB
IS.A.2	68.1%	0	31.9%
IS.A.4	70.6%	0	29.4%
IS.B.2	68.1%	0	31.9%
IS.B.4	70.6%	0	29.4%
IS.C.2	68.1%	0	31.9%
IS.C.4	70.6%	0	29.4%
PSTSWM.sm.2	4.0%	0.4%	95.6%
PSTSWM.sm.4	3.6%	96.4%	0
PSTSWM.med.2	4.0%	0	96.0%
PSTSWM.med.4	3.0%	0.5%	96.5%

performance of IS by 12% and the performance of PSTSWM by 7% as compared to the SCNI scheme. Further, we observe that the MCNI scheme improves the performance of IS-B and PSTSWM-small by 4% and 5%, respectively, as compared to the SCNI scheme, respectively. The improvement seen is expected because both IS and PSTSWM use a lot of large messages, as shown in Table I. However, we observe that the improvement seen in PSTSWM is not significant despite using very large messages to communicate. This is due to the computation intensive nature of the PSTSWM application. For example, when running the medium problem size on four processes, only 6.6% of the total time is spent in MPI.

C. Evaluations with SPEC and Data-Centers

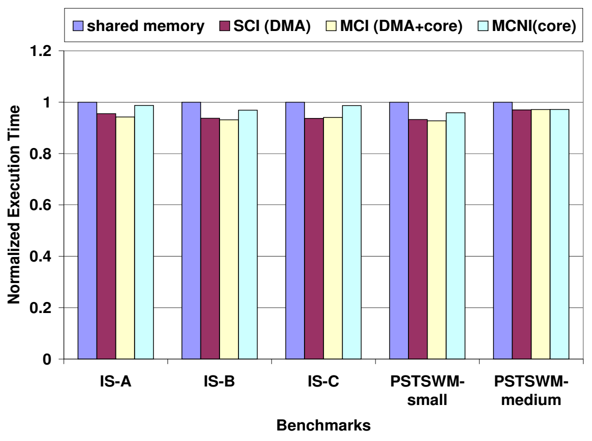
In this section, we evaluate the performance of the proposed schemes with *gzip* SPEC CPU2000 benchmark and simulated data-center services. SPEC CPU2000 benchmark [3] is a set of benchmarks designed to characterize and evaluate the performance of overall system performance such as CPU, memory, etc. In this paper, we focus on one such benchmark, *gzip*, which measures the CPU and memory performance. In order to force SPEC CPU2000 benchmarks to use our schemes, we preloaded a library that intercepts all *memcpy* operations. In all our experiments, we forced the benchmarks to use the different schemes if the message size is greater than 64 KB.

SPEC benchmarks focus on CPU and memory-intensive operations (i.e., memory reads, computations, memory writes) and hence we did not observe any significant improvement in the overall execution time. However, we report the time spent in memory copy operations (greater than 64 KB message size) using the four different schemes during application execution. With protection scheme, we include the time spent in initiating the memory copy and protecting the source and destination buffers and also the time spent in waiting for the memory copy operation to finish when either the source or the destination buffers are touched (i.e. the time spent after receiving a SIGSEGV). Table II shows the total cost of protecting the source and destination buffers before and after the copy operation and we observe that the total protection cost (this includes four *mprotect* calls) is quite less compared to the total time for the memory copy operation. It is to be noted that this cost is the worst case estimate and it can be further optimized for consecutive memory copy operations involving the same source or the destination buffers and if the buffers are not touched in between these memory copy operations.

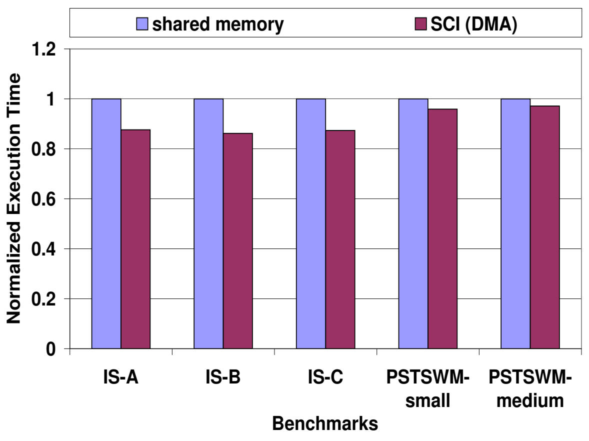
TABLE II. Memory Protection Overhead

Msg. Size	64 KB	256 KB	1 MB	4 MB	16 MB
Cost (usecs)	4.3	7.1	18.6	63.6	227.9

Figure 10(a) shows the performance of *gzip* benchmark with all four schemes. As shown in the figure, we observe that both SCI and MCI schemes improve the performance of memory copy time (i.e., memory copies greater than 64 KB) by up to 35% as compared to SCNI scheme. We profiled the message distribution of *gzip* benchmark for all message sizes greater than 64 KB. We found that more than 50% of the memory copies greater than 64 KB fall between 1 MB and 2 MB. Due to this reason, we observe that the SCI and MCI schemes improve the performance of memory copies. For the MCNI scheme, we observe that the performance does not improve due to large mapping cost overheads. As mentioned in Section III, application-transparent memory copies can further improve the performance if the source and destination buffers



(a) 2 Processes



(b) 4 Processes

Fig. 9. MPI Application Performance

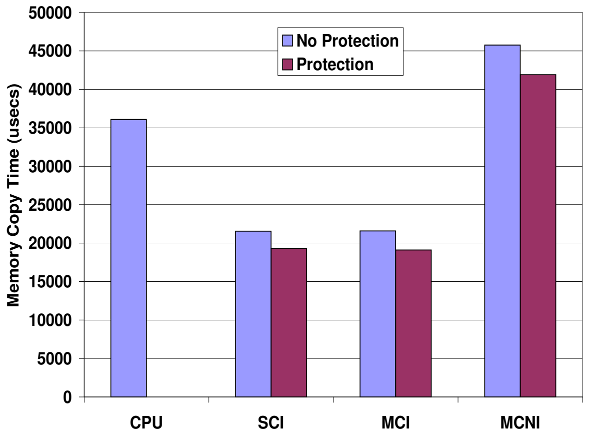
are not accessed immediately after a memory copy operation. As shown in Figure 10(a), we observe that the performance of *gzip* consistently improves by 10% as compared to the performance of blocking memory copy operations for SCI, MCI and MCNI schemes. This result is quite promising for several applications that use memory copies similar to the *gzip* benchmark.

Next, we evaluate the performance of memory copy operations in a data-center environment [15]. For efficiently transferring the data from a remote site to the local node in a data-center environment, a distributed shared state has been proposed in the literature [17]. However, there is no efficient support for transferring the data between a data-center service and the application threads within a node. We use the asynchronous memory copy operations for supporting data sharing within a node. To emulate the multiple threads copying the shared data scenario, we create a single server and three application threads in a single node. The server initiates a data copy operation to all three application servers and waits for the completion operation. The application threads also wait for the completion of the copy operation. For data copy operation, we use a Zipf distribution with varying α value which is common in several data-center environments. According to Zipf law, the relative probability of a request for the ith most popular document is proportional to $1/i^{\alpha}$, where α determines the randomness of file accesses. Higher the α value, higher will be the temporal locality of the document accessed. We use file sizes ranging from 500 bytes to 8 MB in the Zipf trace. We emulate the data-center environment by firing copy requests according the Zipf pattern and measure the average latency after the completion of all copy operations. We report the performance of the SCI, MCI and traditional shared memory (SCNI) schemes for copying the data as shown in Figure 10(b). We observe that the performance of the MCI scheme is significantly better for all Zipf traces. The MCI scheme shows up to 37% performance improvement as compared to the SCI scheme for an α value of 0.5. This is mainly due to avoiding context switch overheads and scheduling the memory copy operations without any delay. The shared memory (SCNI) scheme is better for larger α values compared to the SCI scheme, since majority of the data is transferred through the cache. However, as α value decreases, we see that the performance of shared memory (SCNI) scheme gets worse. Due to a lot of overheads associated with the SCI scheme, the benefits of the SCI scheme show up only when the application uses large memory copies (smaller α values). The performance of the MCNI scheme with other application threads (all four cores are completely utilized) degraded significantly and hence we did not include this result.

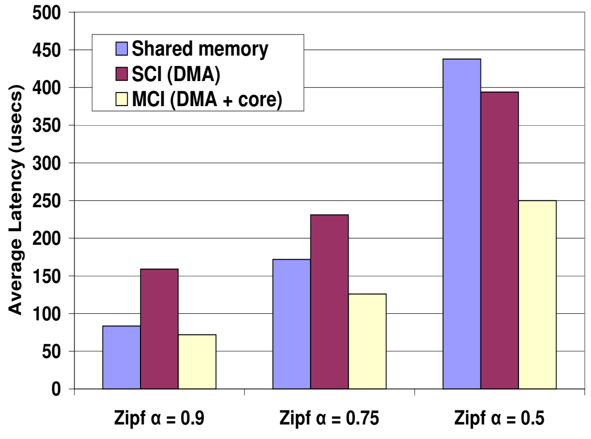
V. Discussion and Related Work

Researchers have proposed several solutions for asynchronous memory operations in the past. Zhao et al [18] talk about hardware support for handling bulk data movement. Calhoun's thesis [7] 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 [9] 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 [10] offers an asynchronous copy engine which improves the copy performance with very little startup costs. In this paper, we use this hardware for supporting asynchronous memory operations. Further, as mentioned in Section III-D, we proposed an application-transparent asynchronous memory copy mechanism which showed up to 10% benefit. This result, though evaluated only for *gzip* application, is quite promising for several applications which can take advantage of complete asynchronism without any modifications to applications.

Regarding MPI intra-node communication, Buntinas et. al. [6] and Chai et. al. [8] have discussed shared memory based approaches and optimizations. Jin et. al. have proposed a kernel assisted memory map based design in [12], which is



(a) Memory Copy Time (> 64 KB) in *gzip* SPEC Benchmark



(b) Emulated Data Sharing in Data-Centers

Fig. 10. Application Performance

similar to the MCNI scheme discussed in this paper. However, the scheme proposed in this paper is more general in that it can be applied not only to MPI but also to other applications such as data-centers. Besides, our scheme dedicates the copy operation to another core thus providing complete overlap of copy operation with computation in a single-threaded application.

VI. Conclusions and Future Work

In this paper, we proposed two schemes to provide complete overlap of memory copy operation with computation by dedicating the critical tasks to a core in a multi-core system. In the first scheme, MCI (Multi-Core with I/OAT), we *offloaded* the memory copy operation to the copy engine and *onloaded* the startup overheads associated with the copy engine to a dedicated core. For systems without any hardware copy engine support, we proposed a second scheme, MCNI (Multi-Core with No I/OAT) that *onloaded* the memory copy operation to a dedicate core. We further proposed a mechanism for an application-transparent asynchronous memory copy operation using memory protection. We analyzed our schemes based on overlap efficiency, performance and associated overheads using several micro-benchmarks and applications. Our microbenchmark results showed that memory copy operations using the MCI and MCNI schemes can be significantly overlapped (up to 100%) with computation. Evaluations with MPI-based applications such as IS-B and PSTSWM-small using the MCNI scheme show up to 4% and 5% performance improvement, respectively, as compared to traditional implementations. Evaluations with data-centers using MCI show up to 37% improvement compared to the traditional implementation. Our evaluations with *gzip* SPEC benchmark using application-transparent asynchronous memory copy show a lot of potential to use such mechanisms in several application domains. We plan to extend the current designs to analyze the impact of application-transparent memory copy mechanism for several other end-applications.

References

[1] HP Research Labs. http://www.hpl.hp.com/.

[2] Parallel Spectral Transform Shallow Water Model. http://www.csm.ornl.gov/chammp/pstswm/.

[3] SPEC CPU2000 benchmarks. http://www.spec.org/cpu2000/.

[4] Boon S. Ang, Derek Chiou, Larry Rudolph, and Arvind. The StarT-Voyager Parallel System. In *IEEE PACT*, 1998.

[5] P. Balaji, S. Bhagvat, H. W. Jin, and D. K. Panda. Asynchronous Zero-copy Communication for Synchronous Sockets in the Sockets Direct Protocol over InfiniBand . In *CAC*, 2006.

[6] Darius Buntinas, Guillaume Mercier, and William Gropp. The design and evaluation of Nemesis, a scalable low-latency message-passing communication subsystem. In *CCGrid*, 2006.

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

[8] L. Chai, A. Hartono, and D. K. Panda. Designing Efficient MPI Intra-node Communication Support for Modern Computer Architectures. In *Cluster Computing*, 2006.

[9] Giuseppe Ciaccio. Using a Self-connected Gigabit Ethernet Adapter as a memcpy() Low-Overhead Engine for MPI. In *Euro PVM/MPI*, 2003.

[10] Intel Corporation. Accelerating High-Speed Networking with Intel I/O Acceleration Technology. http://www.intel.com/technology/ioacceleration/306484.pdf.

[11] D. H. Bailey et al. The NAS parallel benchmarks. volume 5, pages 63–73, Fall 1991.

[12] H. W. Jin, S. Sur, L. Chai, and D. K. Panda. LiMIC: Support for High-Performance MPI Intra-Node Communication on Linux Cluster. In *ICPP*, 2005.

[13] Shubhendu S. Mukherjee. *Design and Evaluation of Network Interfaces for System Area Networks*. PhD thesis, University of Wisconsin-Madison, 1998.

[14] Network-Based Computing Laboratory. MVAPICH: MPI over InfiniBand and iWARP. http://mvapich.cse.ohio-state.edu/.

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

[16] K. Vaidyanathan, W. Huang, L. Chai, and D. K. Panda. Designing Efficient Asynchronous Memory Operations Using Hardware Copy Engine: A Case Study with I/OAT. In *CAC*, 2007.

[17] K. Vaidyanathan, S. Narravula, and D. K. Panda. DDSS: A Low-Overhead Distributed Data Sharing Substrate for Cluster-Based Data-Centers over Modern Interconnects. In *HiPC*, 2006.

[18] Li Zhao, Ravi Iyer, Srihari Makineni, Laxmi Bhuyan, and Don Newell. Hardware Support for Bulk Data Movement in Server Platforms. In *ICCD*, 2005.