NUMA Implications for Storage I/O Throughput in Modern Servers

Shoaib Akram, Manolis Marazkis, and Angelos Bilas[†]
Foundation for Research and Technology - Hellas (FORTH)
Institute of Computer Science (ICS)
100 N. Plastira Av., Vassilika Vouton, Heraklion, GR-70013, Greece
Email: {shbakram,maraz,bilas}@ics.forth.gr

I. Abstract

Current server architectures have started to move away from traditional memory buses that do not scale and towards point-to-point interconnects for communication among processors, memories, and I/O devices. As a result, memory modules are not equidistant from all cores leading to significant differences in memory access performance from different cores. Similar to memory modules, I/O devices are connected today to processor sockets in a NUMA manner. This results in NUMA effects for transfers between I/O devices and memory banks, as well as processor I/O (PIO) accesses to I/O devices. This trend towards NUMA architectures increases complexity for buffer placement, device data transfers, and code execution, creating a complex affinity space. In this paper, we discuss problems that arise when performing I/O and present a preliminary evaluation of the impact of different types of affinity. We use a server-type system with two Intel Xeon processors, four storage controllers, and 24 solid-state-disks (SSDs). Our experiments with various machine configurations show that compared to local transfers between devices and memory, remote transfers have the potential to reduce maximum achievable throughput from 8% up to 40%. Further, for I/O-intensive applications, remote transfers can potentially increase I/O-completion time up to 130%.

II. Introduction

A predominant number of servers deployed in data-centres today use multiple processors on a single motherboard. The processors, memory modules, and the I/O devices are connected together by a cache-coherent, point-to-point interconnect [26], [4]. Such architectures result in non-uniform communication overheads between different devices and memory modules. A known problem in this direction has been the non-uniform latency of memory accesses by a processor to a local or remote memory module. Each processor has faster access to memory modules connected locally to it and slower access to the rest of the (remote) memory modules. In addition, today, accesses from one processor to a remote memory module need to traverse other processors' sockets (also called NUMA domains), interfering with local traffic. Given the current trend towards increasing number of cores in each processor and also the number of sockets, we expect that this non-uniformity will become more diverse with multiple crossings from other processors' sockets for memory accesses. Solutions have been proposed to deal with this problem at the Operating System (OS) layer [13], [8] mainly using various memory management techniques as well as hardware caching approaches. However, these approaches alone are inadequate to deal with affinity issues that arise during transfers between I/O devices and memory. The affinity that a transfer of data exhibits, e.g. from a local memory module to a local I/O device can impact performance.

Figure 1 shows a typical modern server architecture based on a point-to-point interconnect. Note that the number of processors in NUMA architectures has been increasing [12] and the trend is projected to continue. In this paper, we quantify the impact of affinity in non-uniform architectures (NUMA) on storage I/O throughput. Our initial evaluation of a server-class machine with an architecture similar to the one shown in Figure 1 shows that the maximum achievable storage throughput degrades significantly if communication is done without considering proper affinity. In particular, we observe that the maximum achievable throughput can reduce significantly if processor (A) reads data from storage devices connected to chipset (b) compared to reading from devices connected to chipset (a).

The main objective of this paper is to give an initial evaluation of the impact of affinity on storage throughput. In particular, we present the impact of remote buffer placement (improper affinity) on application throughput, device throughput, time taken for completion of OS tasks (system time) and time taken for completion of I/O requests (iowait time). We quantify this impact by placing buffers and scheduling threads manually. We use a simple classification scheme to build four configurations with different approaches to buffer placement and scheduling threads. We evaluate the performance of various applications using these configurations.

Typically in real applications, buffers are allocated in memory modules closest to the processor. However, systems try to balance the use of memory across modules to allow for

[†]Also, with the Department of Computer Science, University of Crete, P.O. Box 2208, Heraklion, GR-71409, Greece.

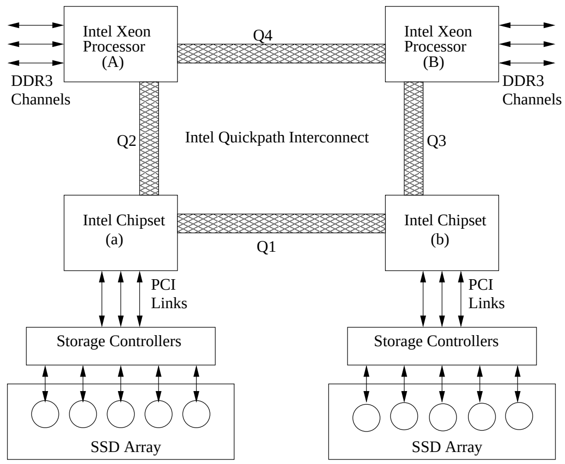


Fig. 1. The top-level architecture of a server machine with non-uniformity.

higher throughput. In addition, the system scheduler may move threads around resulting in the initiation of transfers between devices and memory modules with improper affinity: Data requested by a processor could be located on a device that is either closer to the processor or remote, as shown in Figure 1, whereas the buffers used for the transfer can have different affinity to the processor, resulting in significant variations in the observed performance.

Our work shows that compared to the configuration where transfers between devices and memory are local :

- I/O-intensive workloads suffer from 69% up to 130% increase in I/O-completion time due to remote transfers.
- Filesystem-intensive workloads suffer from 40% up to 57% increase in system time (time for performing OS-related activities) due to remote transfers.
- Throughput-oriented workloads such as state checkpointing or data-streaming suffer up to 20% drop in read/write throughput due to remote transfers.

The rest of the paper is organized as follows. Section III describes a taxonomy of NUMA affinity in modern servers involving memory and devices. In the same section, we describe four configurations with different policies for buffer placement and thread scheduling Section IV describes our methodology for evaluation and in Section V we discuss the results of our evaluation. We provide a discussion of the shortcomings of our work in Section V. We conclude this work in Section VIII.

III. I/O Affinity Taxonomy

In real applications, when a processor accesses a block device for a file, it first allocates a buffer in memory for reading a block from the block device. For instance, consider a worst-case scenario (Figure 1 where a process running on processor (A) allocates a buffer in memory module closer to processor (B) and requests a block of file to be read from the devices connected to the chipset (b). The three high-level operations are 1) issuing the I/O operation, 2) serving the I/O request, and 3) using the data that is returned. We ignore the first operation because unlike the other two operations, issuing an I/O request does not depend on the size of data. The second operation is the type of transfer (local or remote) and the third operation is the usage of data (local or remote). We further differentiate based on the type of transfer (read or write) and the type of usage (load or store).

Table I presents our taxonomy. The best case is when a transfer occurs with proper affinity between a memory module and an I/O controller that are located close to the same CPU socket. Conversely, the worst case is when the transfer buffer and the I/O controller are located in different sockets (also called NUMA domains). An even worse case is when not only the transfers are remote but the subsequent use of the data is by a processor that is located remotely to where the memory module is located. Some typical scenarios for real applications include:

- TLORP0I0 : I/O transfers are local, the transfer operation is read, and data is not used by the processor.
- TRORP0I0 : I/O transfers are remote, the transfer operation is read, and data not used by processor.
- TRORPRIR : I/O transfers are remote, transfer operation is read, and the data that is returned is accessed by remote processor.
- TLORPRIR : I/O transfers are local, transfer operation is read, and the data is used by a remote processor.
- TRORPRIR : I/O transfers are remote, transfer operation is read, and data usage is by remote processor (load).
- TLORPLIR : I/O transfers are local, transfer operation is read, and data is used by the same (local) processor, where data is returned.

The last three cases are depicted in Figure 2: circles denote CPUs or devices involved in the I/O operation. Arrows denote the transfer path taken by an I/O request. The first transfer is from chipset to memory DIMM. Next, we discuss buffer management and thread scheduling taking NUMA effects into account. Proper buffer management involves placing data in the same memory module that is connected to the socket as the storage controller responsible for the I/O operation. Thread scheduling involves running threads on the CPU that is connected to the memory module containing data needed by the CPU. In this paper, we do not propose new algorithms for scheduling and buffer placement. Instead, we place threads and buffers manually and build five configurations for evaluating the possible range in performance degradation. In order to understand the configurations, we first describe the copies that take place when data is transfered from from a device to application memory.

The I/O stack of a typical OS today is shown in Figure 3. For each I/O request made, there are two buffers involved in the transfer from the device to the application: One buffer in the application address space and one in the kernel. The placement of the application buffer is controlled in our experiments via `numactl` that is able to pin threads and buffers to specific sockets. Kernel-buffer placement cannot be controlled; I/O buffers in the kernel are part of the buffer cache and are shared by all contexts performing I/O in the kernel. Thus, a context

TABLE I
TRANSFER AFFINITY TAXONOMY.

Transfer (T)	Transfer Operation (O)	Core access (P)	Access type (I)
Local (L)	Read (R)	Local (L)	Load (R)
Remote (R)	Write (W)	Remote (R)	Store (W)
		None (0)	

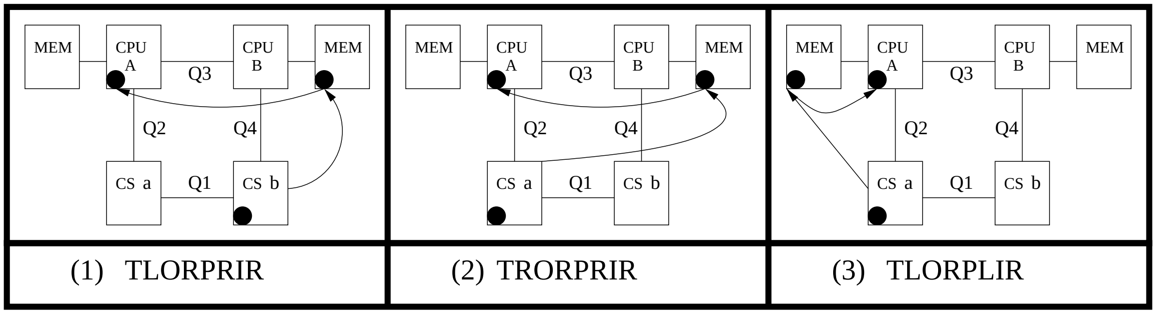


Fig. 2. Pictorial representation of three cases derived from the taxonomy described in Table I.

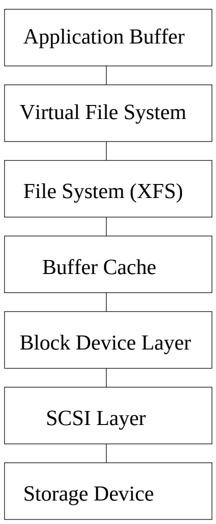


Fig. 3. I/O Stack in Modern Operating Systems.

might use a buffer that is located in any memory module. Creating different buffer pools for each socket could allow proper kernel buffer placement and use, however, requires extensive kernel modifications. In our experiments, buffer allocations are initiated by user contexts entering the kernel (we always start experiments with a clean buffer cache). This results in (properly) placing buffers initially in the socket where the user context is running. Although during each experiment buffers can be reused by other contexts performing I/O resulting in degraded affinity, this is not very pronounced due to the large memory size in our setup.

Based upon buffer placement and thread scheduling, we use five configurations shown in Table II. The axis for classification are: (1) local versus remote transfers between I/O device and memory and (2) local versus remote copy operation. This copy operation is between the application buffers and

TABLE II
CONFIGURATIONS.

Transfer (TR)	Copy Operation (CP)	Configuration
Local(L)	Local(L)	TRLCPL
Remote(R)	Remote(R)	TRRCPR
Remote(R)	Local(L)	TRRCPR
Local(L)	Remote(R)	TRRCPR

the buffers of the OS-managed cache. We manually control the source and destination of each copy operation by placing threads and their buffers appropriately via `numactl`.

IV. EVALUATION METHODOLOGY

In this section, we describe our experimental platform, applications for evaluation, and our methodology for evaluation.

A. Testbed for Evaluation

The top-level diagram of our evaluation platform is similar to the one shown in Figure 1. The server uses Intel Xeon Quadcore processors with four cores and eight hardware threads (two-way hyperthreaded). The server is equipped with two chipsets also from Intel (Tylersburg 5520). We populate the three memory slots with three DDR3 DIMMs. Each DIMM occupy a separate physical channel. We use four storage controllers (two per chipset). The storage controllers are form LSI (Megasas 9260). We use a total of 24 SSDs (Intel X-25 SLC). Each storage controller is connected to six SSDs. We create a software RAID device on top of six SSDs connected to each storage controller. Therefore, each processor has two software RAID devices that are local to it with better affinity and two that are remote with worst affinity. We use CentOS release 5.5 OS distribution with 2.6.18-194.32.1.el5 kernel (64-bit). For placing buffers and contexts, we use the `numactl` library for Linux (version 2.0.7).

B. Bandwidth Characterization of System Components

In this section, we describe the bandwidth of individual system components in order to understand the peak limitations in our system. The bandwidth of the QPI links (labeled Q1, Q2, Q3, Q4) is 24 GBytes/s. Each storage controller from LSI is able to achieve 1.6 GBytes/s. The SSDs can sustain a throughput of about 200 MBytes/s for sequential writes and 270 MBytes/s for sequential (or random) reads. To measure the memory bandwidth in our system, we use a in-house benchmark modeled after STREAM [14] called mstress. We run multiple instances of mstress and measure the memory throughput with local and remote affinity. Figure 4(a) shows our results. The peak bandwidth of storage controllers is much less than the memory subsystem and the QPI interconnect, neither of these is a potential bottleneck when performing I/O.

C. Methodology

To evaluate the impact of wrong buffer placement on application performance, we use the following benchmarks and applications:

1) zmIO: is an in-house benchmark that fully stresses the storage sub-system of our high-end server machines (4 storage controllers each capable of doing 1.6 GB/s). zmIO uses the asynchronous API of Linux for performing I/O operations [1]. zmIO issues multiple (user-defined parameter) I/O operations and keep track of the status of each of the operation in a queue called status queue. When the status queue is full, zmIO performs a blocking operation and waits for an I/O operation to complete. A new operation is issued after completing a pending operation. The completion of I/O operations by CPU and the completion of outstanding I/O operations by the storage devices happens in parallel. We run zmIO in direct mode. Note that in direct mode, zmIO performs I/O access to storage devices that does not go through the page cache in the kernel.

2) fsmark: is a filesystem stress benchmark that stresses various features of the filesystem. fs_mark runs a sequence of operations on filesystem layer. In particular, we use it to perform the operation sequence create,open,write,read, and close. We run fs_mark using 128 threads with each thread creating a single directory and 128 files within each directory. Each thread chooses a random directory and performs the specified sequence of operations on any of the files within the directory.

3) IOR: simulates checkpointing support in compute- intensive applications [18]. We use the MPI API for performing I/O operations. We run IOR on top of the XFS filesystem. We use 32 processes that checkpoint a 2 GB state to a shared file (aggregate file size is 64 GB). Each process works with a single file using sequential offsets within the single file.

4) Stream: is a synthetic application that simulates the end-to-end datapath of data streaming systems [11]. The application consists of a consumer thread that reads 64 KB records in a buffer. The consumer thread enqueues the pointer to buffers in a list of descriptors. The list has 128K entries. The producer thread reads the buffer from the list of descriptors, performs some conditioning on the buffer, updates the list of descriptors and stores the record to storage device.

5) Psearchy: is a file indexing benchmark in the MOS-BENCH [10] suite. File indexing is mainly done as a backend job in data centres and web hosting facilities. We run Psearchy using multiple processes. Each processes picks a file from a shared queue of file names. Each process has a hash table for storing in-memory BDB indices. The hash tables are written to storage devices once they reach a particular size. We use 32 processes, 128 MB hash tables per process, and 2 KB reads and character oriented writes. We use 100 GB corpus, 10 MB file size, 100 files in each directory and 100 directories.

For evaluating NUMA effects, we run a workload consisting of four instances of the same application or benchmark. We assign one RAID 0 device consisting of six SSDs to each instance. Next, we define various metrics for our evaluation.

To project results to future systems with more components, it is important to use appropriate metrics for evaluation and observe how various components of the system are stressed instead of merely observing the application throughput. For this reason we use:

- Application Throughput (GB/s): The application throughput refers to the aggregate bytes accessed by the application divided by the execution time. Usually, read and write throughput is reported separately based upon the total bytes read or written during the execution time.
- Cycles per I/O (CPIO): In this work, we define and use CPIO as a new metric for characterizing behavior of applications that mainly process I/Os. We define CPIO as the total cycles spent by the application divided by the total sectors read and written by the device. We believe that CPIO is particularly important for data-centric applications that perform a one-pass over the dataset as it gives an estimate of the work performed per I/O sector. Ideally, as the number of cores increase, CPIO should remain the same. Thus, it is a measure of how well the applications scale on new generations of systems.
- Throughput per socket: For one application, we report the results in terms of throughput per socket. Because of non-uniformity in the server systems, it is important to maintain similar throughput across the entire system. We show that for one of the applications, the throughput is different for each socket depending upon the scheduling scheme.

Since CPIO is a new metric we use in this paper, we discuss it in detail below. We calculate cpio for each application by running each application in a *meaningful* configuration; applications when run, should generate I/O traffic. For instance, cases where the workload fits in the available memory and exhibit low I/O are probably not typical of future configurations since the demand for data grows faster than DRAM capacity. For this purpose, we select datasets that are big enough to not fit in memory and generate I/O throughout execution.

To calculate CPIO, we measure the average execution time breakdown as reported by the OS and consisting of user, system, idle, and wait time. We also note the number of I/Os that occurred during the same interval. There are two issues related to the cpio calculation. First, what each of the components means and second which ones should be taken into account to come up with a meaningful metric. We next briefly explain what each component of the breakdown means.

user time refers to the time an application spends executing code in the user space. When the user application request services by the OS, the time spent is classified as *system time*. The time an application spends waiting for I/Os to complete is classified as *wait time*. *idle time* refers to the time that the application either has no more work to perform within the current quantum or because it is waiting for resources that are not available, for instance, locks. We use the modified terms called $CPIO_{iow}$and $CPIO_{sys}$respectively to describe the two components in terms of $CPIO$. In our evaluation, we use sector-size I/Os, with each sector being 512 bytes. Note that since CPU cycles proportionate to power [15], and given the increasing emphasis on energy efficiency in data centres, CPIO is an important metric.

V. RESULTS

In this section, we describe the results of our evaluation.

A. zmIO

We run zmIO in *direct* mode, and therefore, I/O accesses do not go through the page cache in the kernel. Hence, there is no distinction between local and remote copies. For DMA transfers between devices and memory, the buffer provided by the application is used instead. Note that this buffer is aligned across the page boundary. In order to evaluate the impact of affinity on throughput of zmIO, we use the affinity taxonomy listed in Table I for describing our results. We mainly focus on three issues:

- The impact of affinity between source and destination of a transfer operation on storage throughput. Effectively, this shows how much better or worse I/O transfers can become by employing the wrong affinity.
- The impact of processor memory accesses on data transfers, in combination with affinity. Typically, programs that perform I/O also use CPU cycles to process data. We examine the impact of accessing memory from the processor to I/O transfer throughput.
- The impact of contention between processor and I/O memory accesses on maximum achievable memory throughput. Although this issues is similar to above, in this case we are interested in whether simultaneous accesses from processors and I/O devices to memory result in a degradation of the maximum throughput, rather than the impact on I/O throughput.

To evaluate the impact of affinity between source and destination on storage bandwidth, we run multiple instances of zmIO and measure the throughput. Figure 4(b) shows the throughput of zmIO with up to eight instances. The reduction in throughput with more than two instances and remote affinity is up to 40%.

At this point, it should be mentioned that we measured throughput of zmIO using different machine configurations. We observed that NUMA effects on throughput of zmIO depend on a number of factors including OS distribution, the version of Linux kernel, version of `numactl` library, and even the type of motherboard. We observed that while Figure 4(b) shows a 40% drop in throughput, one of the machine configuration with a newer OS distribution and kernel, we observed 8% drop in throughput due to remote transfers. We believe that the range of degradation that an application can potentially suffer due to remote transfers is important to quantify and improve.

Next, we optionally perform a summation operation over all the bytes returned by the I/O read to observe the impact of TLORPRIL and TRORPRIL. The variable that stores the sum of the bytes is pinned in memory. The size of each transfer is 1 MByte. Figure 4(b) shows results with zmIO touching the data. Note that the absolute throughput for local transfers and local use (TLORPLIR) is lower to that of TLORP0I0 because both the outstanding I/Os and the summation operation accesses memory simultaneously. The reduction in throughput for TLORPRIR when the data is used by a remote processor is 5% with four instances of the benchmark. Beyond four instances, TLORPRIL and TLORPLIL behave similarly. We do not show results for TRORPLIR as it is also bounded by the bandwidth of remote transfer operation and behaves similar to the second case (TRORPRIR).

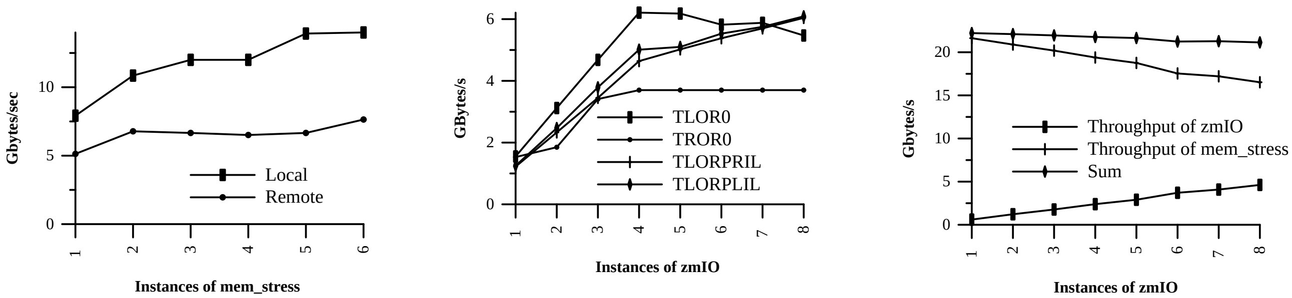
Next, we show how memory contention can hurt the performance of storage I/O throughput in case of TLORPLIR in Figure 4(c). We run instances of zmIO and mstress together. We run up to eight instances of zmIO. Neither mstress nor zmIO is bottlenecked by the CPU in this experiment. We run zmIO in TLORPLIR mode. The absolute throughput of zmIO drops by 23% for eight instances when there is contention for memory throughput i.e., mstress is running. The sum of memory bandwidth used by zmIO and mstress together is never greater than 22 GBytes/s which is the maximum memory bandwidth in the system.

B. fsmark

We discuss the results for fsmark in terms of cycles per I/O. Since fsmark mostly perform operations related to the filesystem, the system time is high. Also, due to contention from multiple threads for I/O devices, iowait time is high. Figure 5(a) shows the breakdown of CPIO in terms of $CPIO_{sys}$ and $CPIO_{iow}$. Remote transfers (TRRCPR) result in a 40% increase in $CPIO_{sys}$ compared to local transfers (TRLCPL). Also if transfers are local, remote memory copy operation (TRLCPR) result in a 15% increase in $CPIO_{sys}$ compared to TRLCPL. There is a 130% increase in $CPIO_{iow}$ due to remote transfers. The difference in $CPIO_{iow}$ due to remote copies is not noticeable.

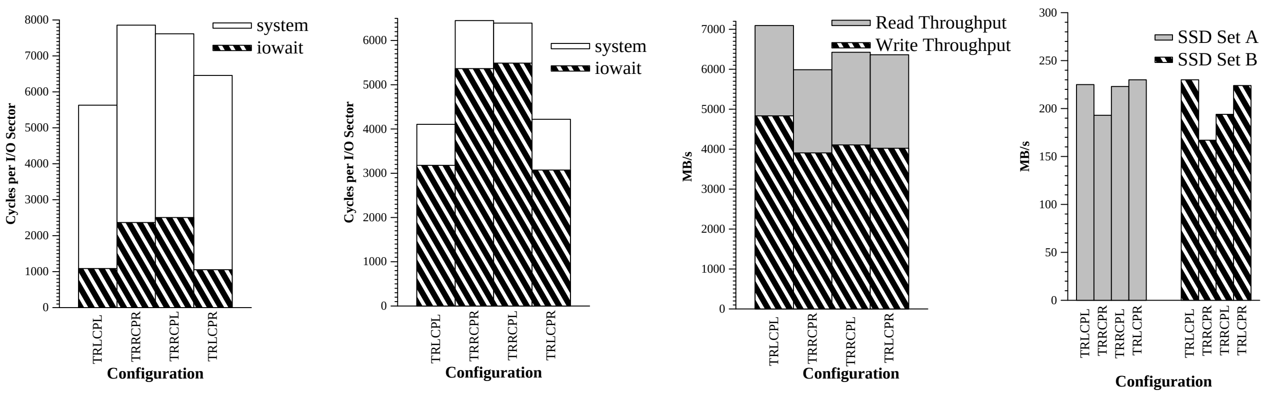
C. Psearchy

The results for Psearchy are shown in Figure 5(b). Again, we discuss the results in terms of the cycles per I/O metric.



(a) Memory throughput (b) NUMA affinity results for zmIO (c) Contention results

Fig. 4. Results for mstress and zmIO.



(a) fsmark (b) Psearchy (c) IOR (d) Stream

Fig. 5. NUMA affinity results for benchmarks and real applications.

First, we observe that remote transfers result in an increase in $CPIO_{sys}$ and $CPIO_{iow}$. However, remote copies does not show a noticeable difference. In particular, TRRCPR results in a 57% and 69% increase in $CPIO_{sys}$ and $CPIO_{iow}$ respectively relative to TRRCPL.

D. IOR

We report read and write throughput of IOR for different configurations in Figure 5(c). Note that for IOR, read operations can potentially complete in memory and thus the aggregate throughput in Figure 5(c) goes up to 7 GB/s. We observe that the read throughput decreases by 16% for the worst case (TRRCPR) compared to the best case (TRLCPL). Similarly, write throughput decreases by 19% due to remote transfers.

E. Stream

Figure 5(d) shows the results for the streaming workload. We show the throughput observed on each of the two sets of SSDs. Note that one set of 12 SSDs is connected to two storage controllers. Compared to TRLCPL, we observe a 14% and 27% drop in throughput respectively for the two set of SSDs in case of TRRCPR.

VI. Summary and Discussion

In this section, we first summarize the results of our evaluation. We then provide implications of our results for other important data-centric applications. We also discuss the shortcomings of our methodology for evaluation.

A. Summary of Results

We summarize our results as follows:

- Applications that are I/O-intensive suffer from 70% up to 130% increase in iowait time and from 40% up to 57% increase in system time due to remote transfers.
- For streaming workloads, remote transfers can potentially result in asymmetric throughput across the system i.e., some (NUMA) domains can provide more throughput compared to other domains.
- Checkpointing applications can potentially suffer a 20% degradation in write throughput due to remote transfers.
- Finally, raw device throughput, as measured by microbenchmarks such as zmIO, can drop from 8% up to 40% depending upon the machine configuration.

B. Discussion

Our main purpose is to discuss the I/O behavior of many emerging data-centric applications. In particular, we are interested in NUMA affinity effects on the performance of these applications. The applications we collected for evaluation comes from various domains. In particular, these applications are part of various benchmark suites including PARSEC [6], MOSBENCH [10], two OLTP workloads from the TPC foundation, and emerging data stores. A brief description of the

TABLE III
APPLICATIONS AND DATA SETS FOR EVALUATION.

Application	Description
zmIO	I/O subsystem stress test: direct mode (D) or through VFS.
fs_mark	File system stress test.
IOR	Application checkpointing.
Psearchy	File indexing: Directories (D) can be small (L) or large (L); files (F) can be small (L) or large (L).
Dedup	File compression: Files can be small (S) or Large (L).
Ferret	Content similarity search: Files could be Small (S) or Large(L).
Metis	Mapredce library: Word Count (C) or Linear Regression (LR).
Borealis	Data streaming: Record size could be 64 KB (Bor64), 128 bytes (Bor128), or 1 KB (Bor1024)
HBase	Non-relational database.
BDB	Key-value data store.
TPC-C	OLTP workload (Warehouse).
TPC-E	OLTP workload (Stock broker).
Tarrif	Profiling of Call Detail Records.

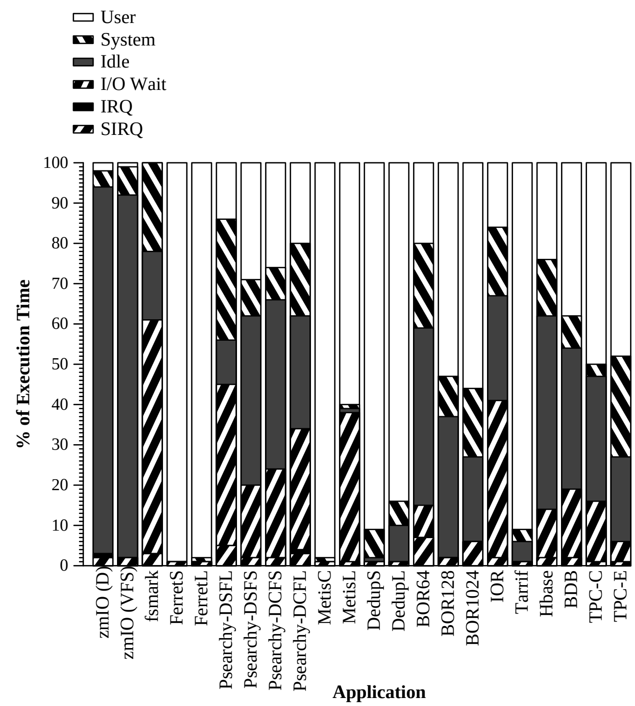


Fig. 6. Breakdown of time spent by various applications in terms of user, system, idle, iowait, serving hardware interrupts (IRQ) and serving software interrupts (SIRQ).

applications along with the type of data sets is given in Table III.

In terms of I/O behavior, most applications in Table III does not have high system or iowait times. Further, most applications does not stress the storage subsystem in a manner similar to applications we evaluate in Section V. For this reason, using different configurations do not show a noticeable difference in application throughput, CPIO, or physical device throughput. We suspect two potential reasons for this behavior as follows:

Figure 6 shows the breakdown of execution time of the applications in Table III in terms of user, system, idle, and iowait time. The breakdown is collected by running one instance of each application on top of a software RAID device consisting of 24 SSD devices. We note from the figure that most applications exhibit neither a significant component of system time nor iowait time. This lead us to the conclusion that in current NUMA systems, transfers from remotely located devices are detrimental to performance only if the application exhibit significant system or iowait time.

Finally, our experimental results, performed under controlled circumstances, strongly suggest that the kernel allocates buffers for paging purposes locally. Nevertheless, we can not manually control the placement of kernel buffers. Most applications in Table III have complex runtime layers and a large user-level application code base. Therefore, proper placement of kernel buffers can not be guaranteed.

VII. RELATED WORK

Much work has been done for NUMA-aware process scheduling and memory management in the context of shared memory multiple processors [24], [16], [23]. Here, we discuss recent work for modern server machines with multiple sockets on a single motherboard.

With the trend towards multiple cores on a single processor chip in commodity desktop and server machines, there is no longer a one-to-one mapping between I/O devices (network interface cards (NIC) or storage controllers) and processing resources (cores, virtual threads or even processors in motherboards with multiple sockets). For instance, a network interface card (NIC) can route the incoming traffic pertaining to a particular socket to a specific core and the rest of traffic to some other core. Recent 10 GBit/s Ethernet NICs from Intel (IX10GBE) provide multiple hardware queues and mechanisms to associate each queue in hardware to a particular software queue (which in turn is bind to a single core) [3], [2].

NUMA memory management is the problem of assigning memory in a NUMA processor such that threads use memory located next to the processor that they mostly run. These issues are discussed in the realm of traditional multiprocessor systems in [9], [17]. Recently, with multiple cores becoming commonplace, commodity OS developers have started to invest efforts to provide a NUMA API for programmers [5].

The authors in [20], [19] quantify NUMA effects in the memory subsystem of Xeon 5520 processor from Intel. The authors report that current memory controllers favor remote memory accesses to local memory accesses which implies that scheduling for data locality is not always a good idea. Also, they show that throughput of remote memory accesses are limited by QPI bandwidth. In this work, we show that along with remote memory accesses, accessing remote I/O devices can also hurt performance of realistic workloads.

Recently, there is a surge in literature dealing with thread scheduling for modern servers. The authors in [25], [7] discuss scheduling policies that address shared resource contention. Their scheduling policies are built on a classification scheme for threads and addresses contention in the memory subsystem. In this paper, we use the transfer path from I/O devices to physical memory and the processor that subsequently uses the data to classify our scheduling policies.

Finally, energy efficiency in data centres is becoming more and more important. In [21], [22], the authors discuss the issues of co-scheduling processes considering both memory bandwidth and potential of frequency scaling.

VIII. Conclusions

In this paper, we described a problem in modern server machines that use point-to-point interconnects to connect CPUs, memory and devices. We discuss the performance degradation on applications if processes access data from memory modules or devices that are located remotely to the processor. As systems are built with more CPUs and sockets, with each CPU having many cores and various memory modules, the performance degradation due to the presence of NUMA affinity in the system will increase. We propose a taxonomy based upon the transfers from storage devices to memory modules and the use of data by the process running on the local or the remote socket. We describe four configurations with different buffer placement and process scheduling policies. We classify the configurations based upon how the transfers occur between the storage devices and the kernel memory, and from the kernel memory to the buffer reserved by the application. Our results show that NUMA effects are particularly degrading for I/O-intensive applications. As systems become more and more heterogeneous, a more general solution to the placement and scheduling problem will become essential for NUMA servers.

IX. Acknowledgments

We thankfully acknowledge the support of the European Commission under the 7th Framework Programs through the IOLANES (FP7-ICT-248615), HiPEAC2 (FP7-ICT-217068), and SCALUS (FP7-PEOPLE-ITN-2008-238808) projects. We are thankful to Yannis Klonatos for modifications to the original fsmark benchmark, Michail Flouris for modifications to Psearchy, and Zoe Sebepou and Markos Fountoulakis for providing the zmIO benchmark.

References

[1] Kernel Asynchronous I/O (AIO) Support for Linux . http://lse.sourceforge.net/io/aio.html .
[2] Receive flow steering . http://lwn.net/Articles/382428/ .
[3] Receive packet steering . http://lwn.net/Articles/362339/ .
[4] Advanced Micro Devices, Inc. AMD HyperTransport™ Technology. http://www.amd.com.
[5] A.Kleen. A numa api for linux. In *http://www.firstfloor.org/ andi/numa.html.*, 2004.
[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT'08, pages 72–81, New York, NY, USA, 2008. ACM.
[7] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT'10, pages 557–558, New York, NY, USA, 2010. ACM.
[8] M. J. Bligh. Linux on numa systems. In *Proceedings of the Linux Symposium*, 2004.
[9] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for numa memory management. In *Proceedings of the twelfth ACM symposium on Operating systems principles*, SOSP'89, pages 19–31, New York, NY, USA, 1989. ACM.
[10] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.
[11] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. etintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *In CIDR*, 2003.
[12] M. Feldman. Dell unveils eight-socket hpc box. In *http://www.hpcwire.com/features/Dell-Unveils-Eight-Socket-HPC-Server-116201574.html*. HPCwire, 2011.
[13] http://lse.sourceforge.net/numa/. Linux Support for NUMA Hardware.
[14] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/ .
[15] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid. Server engineering insights for large-scale online services. *IEEE Micro*, 30:8–19, July 2010.
[16] R. P. LaRowe, Jr., C. S. Ellis, and L. S. Kaplan. The robustness of numa memory management. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP'91, pages 137–151, New York, NY, USA, 1991. ACM.
[17] R. P. Larowe, Jr. and C. Schlatter Ellis. Experimental comparison of memory management policies for numa multiprocessors. *ACM Trans. Comput. Syst.*, 9:319–363, November 1991.
[18] llnl.gov. ASC Sequoia Benchmark Codes . https://asc.llnl.gov/sequoia/benchmarks/.
[19] Z. Majo and T. R. Gross. Memory management in numa multicore systems: trapped between cache contention and interconnect overhead. In *Proceedings of the international symposium on Memory management*, ISMM'11, pages 11–20, New York, NY, USA, 2011. ACM.
[20] Z. Majo and T. R. Gross. Memory system performance in a numa multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR'11, pages 12:1–12:10, New York, NY, USA, 2011. ACM.
[21] A. Merkel and F. Bellosa. Memory-aware scheduling for energy efficiency on multicore processors. In *Proceedings of the 2008 conference on Power aware computing and systems*, HotPower'08, pages 1–1, Berkeley, CA, USA, 2008. USENIX Association.
[22] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems*, EuroSys'10, pages 153–166, New York, NY, USA, 2010. ACM.
[23] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th international symposium on Memory management*, ISMM'06, pages 84–94, New York, NY, USA, 2006. ACM.
[24] M. Steckermeier and F. Bellosa. Using locality information in userlevel scheduling. Technical report, University Erlangen-Ng, IMMD IV, 1995.
[25] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS'10, pages 129–142, New York, NY, USA, 2010. ACM.
[26] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek. Intel® quick-path interconnect architectural features supporting scalable system architectures. *High-Performance Interconnects, Symposium on*, 0:1–6, 2010.