Albert Tannous
Christina Patrick
Jesse Scott

Sequential I/O Streams versus Parallel I/O Streams: A Case Study

Source and Destination Disk Aware Improved Data Copying

Table of Contents

Abstract

The manufacturers of CPUs have been steadily increasing the performance of their devices with increasing clock rates and multi-core processors; the performance increase requires increasing data transfer rates to reap the full benefit of their change. The read and write speeds of hard drives have not been able to keep pace with this increased performance, giving rise to the input/output (I/O) bottleneck. Parallelism in I/O is used to increase the throughput of I/O in applications [13, 14]. However, the use of indiscriminate parallel I/O can actually hurt the performance of I/O in applications. Here, we present a case study of a parallel version of the copy command used to perform I/O versus a sequential version of the same command. We present our observations, solutions, and results in this case study.

The copy (cp) command [1] is a frequently used operating system command. In this paper, we propose a novel and simple technique to increase the cp command's performance when copying multiple large files simultaneously in single user environments such as laptops or personal desktops while maintaining varying levels of fairness. We implemented and tested our solution in a Linux environment. The results of the experiments showed that our approach is highly effective. We saw an increase in the performance of the cp command in all the cases we tested, with a peak of a 61% improvement using a relatively small data set. Moreover, we believe that our idea is easily portable to other platforms such as Windows, Solaris, AIX, BSD, and other UNIX-like Operating Systems.

I. Introduction

The copy command is one of the most widely and frequently used in any operating system (OS). The OS usually does not use optimal scheduling as shown by regular usage of the copy/move operations simultaneously on multiple files. Operating systems interleave the copy of files by time slicing the operations in order to utilize the CPU effectively. While this scheduling is optimal for CPU intensive tasks, it degrades the performance drastically for I/O intensive jobs. The time slicing technique causes repetitive switching at the OS level resulting in significant disk head mechanical switching; this increases the time required to complete the copy operation. Our idea improves the performance of the copy operation by reducing the frequency of disk head switching.

We distinguish two cases: parallel copy and sequential copy.

- Parallel copy occurs when the user launches multiple copy operations on several files. The OS creates one process for each operation, and executes them simultaneously. A similar behavior would be launching several commands separated by "&"[1] in a terminal, or selecting one file in the GUI, copy/paste it, and then select another file.

[1] "&" sends the process in the background, and allows for the next command to be executed right away.

```
# cp file1 /destination/path & cp file2 /destination/path
```

- Sequential copy is when the user launches one copy operation of multiple files. The OS creates one new process at a time, completes the I/O, waits for the process to terminate and then launches another process, which results in copying one file after another. A similar behavior exist when launching several commands separated by ";"[2] in a terminal, or selecting multiple files at once in the GUI and copy/paste them.

```
# cp file1 /destination/path ; cp file2 /destination/path
```

It is expected that the performance of the parallel copy operation supersedes that of the sequential copy operation because the files are copied simultaneously. However, we observe the opposite - due to the excessive mechanical switching of the read/write head at hardware level. The parallel copy takes more time compared to the sequential copy. The difference in execution time between parallel and sequential copy grows exponentially with the number of files copied. The table below shows some preliminary measurements when copying 8 files of 350MB each, totaling 2.8GB. We executed the test on both Linux Ubuntu 6.10 and Windows XP SP2, by performing both intra-disk copy as well as inter-disk copy. Table 1 enumerates the 4 possible variants tested as the baseline for the current OS implementation. The table shows a significant increase for all parallel implementations. We propose a novel and simple technique to increase the performance by reducing the copy time of parallel operations, and the results from the experiments are very promising.

	Windows XP	**Linux Ubuntu (6.10)**
Sequential Intra-disk	4'30"	4'15"
Parallel Intra-disk	8'15"	6'45"
Sequential Inter-disk	3'30"	3'15"
Parallel Inter-disk	12'00	6'30"

Table 1: Execution times for current operating system implementations

In Section 2, we provide several ideas to achieve our goal. Section 3 describes the approach we selected and developed. Section 4 provides the experimental results of the primary implementation. In Section 5, we discuss several extensions to our technique and present the analysis of these results. Finally, we discuss future and related work in Section 6 and conclude the paper in Section 7.

[2] ";" start the execution of the next command only after the current process terminates.

II. Proposed Solutions

To solve the scheduling problem, the obvious solution is forcing parallel copies to execute sequentially. We examined several methods to achieve that goal. Every copy command is composed of several system calls including open(), seek(), read(), write(), and close(). System calls are low-level methods that applications use to interact with the kernel. These calls are typically made by wrapper functions implemented in the standard C library for passing parameters to the kernel. Figure 1 depicts the behavior of an individual copy process. When a process issues an I/O request, the CPU scheduler swaps out the process and puts it in the wait queue in favor of another process that is already in the ready queue. Thus processes that are I/O intensive spend most of their time in the wait queue.

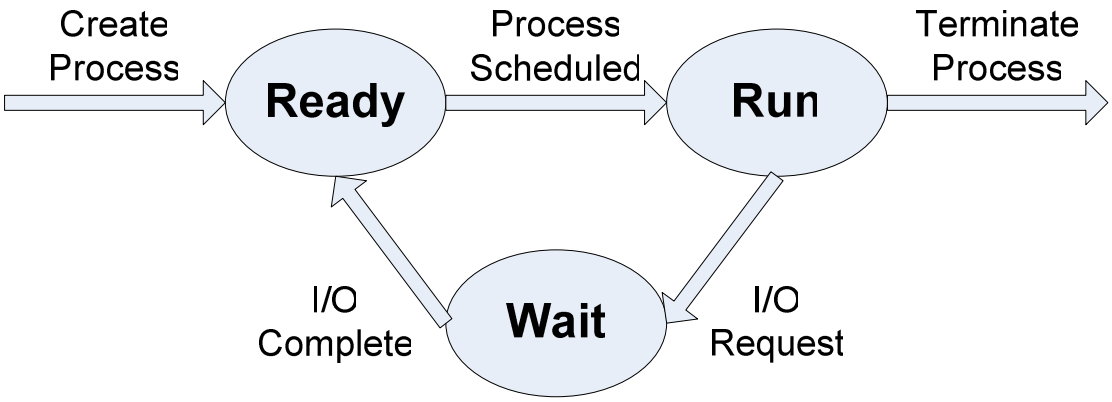

Figure 1: Process Execution State Diagram

➢ Current Behaviors

The current behavior of the copy operation is divided into serial and parallel categories. These methods both use the same underlying cp module for execution. What makes them different is how the OS handles the collection of processes as a group.

i. Serial Method

The serial method, illustrated in Figure 2 can be visualized as a collection of commands that are standing in a single file waiting to be initiated by the OS. Each command is then selected for execution, the process is created, a sequence of I/O requests is completed, and then the process is terminated. The next command in line is then executed in this fashion. Serial behavior makes the OS and the hardware only responsible for one process at a time eliminating disk head switching and decrease total processing time. The downside of this method is the poor response time for commands at the end of the line. Figure 2 illustrates the transitional flow that occurs during a sequential copy command. Take note that the creation of a process does not occur until the previous process has terminated.

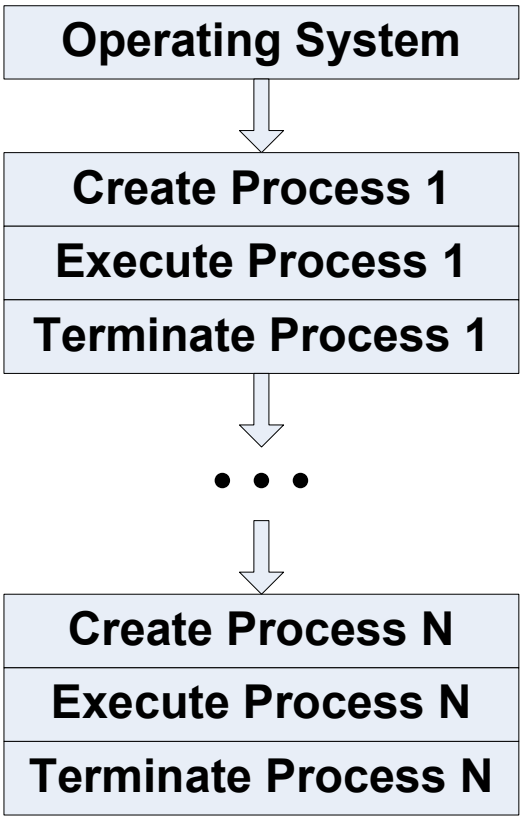

Figure 2: Execution scenario of processes in Sequential Copy

ii. Parallel Method

Parallel behavior uses the same individual processes but allows them all to execute simultaneously. Figure 3 shows, all commands are selected for execution, a process is created for each, and a series of I/O requests are issued from all processes. As mentioned earlier, the CPU scheduler swaps processes continuously in between the run queue and the wait queue so that each process' I/O requests get interleaved by the OS causing the disk head to jump between different areas on the physical disk. Parallel behavior simulates simultaneous execution and provides better response time than the serial method for small files. The effect of the parallel method is an increase in overall time required to complete the I/O for each individual file. Figure 3 shows the process flow of a copy command using the parallel method. The execution stages of each process are still sharing a single I/O path independent of the number of processes attempting to request I/O.

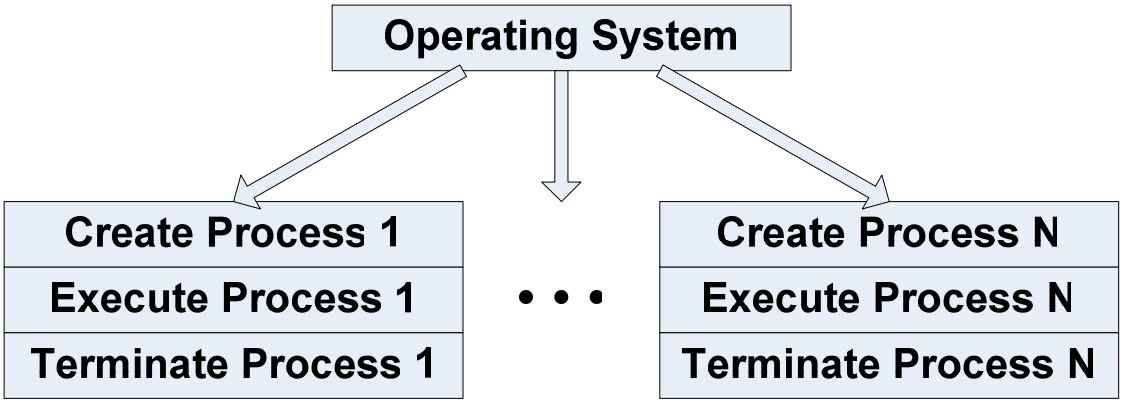

Figure 3: Execution scenario of processes in Parallel Copy

➢ Optional Methods

There are many methods to optimize the I/O in this scenario. The most popular method is using I/O aware schedulers that give preference to I/O intensive jobs or schedulers that that boost I/O performance by idling for some time in the hope that the process may issue more contiguous I/O requests. An example of such a non-work conserving scheduler is the anticipatory disk scheduler [5]. We approach the problem of scheduling I/O intensive processes from an application aware perspective. Rather than changing the scheduler in a system, which may or may not yield better results, we design self scheduling applications that are I/O aware and will always yield better performance. Our goal is to serialize the parallel processes created for multiple copy processes and achieve the same performance as that of the serialized copy. There are many possible ways for accomplishing this serialization. Independent of the method selected, the goal is to schedule the disk access to minimize disk head mechanical switching. Some of the ways in which we can serialize these processes and their pros and cons are considered and presented below.

i. System Call Trap

One method is to trap these calls when the process is created [7, 8], keep track of all information at the process level, and then schedule the I/O calls. This method is similar to the anticipatory disk scheduling algorithm [5]. A straightforward way to trap system calls is to provide a separate library for users to re-link their code. In order to perform the trap method, one has to run a daemon program that traps all the I/O calls made by each cp program and then schedule the processes to run. All the cp processes use the service of this daemon. This method involves starting a daemon at system boot time, requiring modification of the rc scripts in the system. Also, the daemon takes care of interrupts. Since the daemon program is still just a process, the OS can choose to not schedule it or schedule it for very little time and therefore reduce the effectiveness of this method. Since the daemon is still a basic user process that can not control how often it runs, any downtime implies that the cp command will not work.

ii. VFS Layer Trap

Another method is to trap I/O calls in the VFS plug-in layer [4, 6] of the file system. All I/O calls go through the VFS layer. They can be easily trapped here and any additional functionality can be provided in this layer. The core drawback of the system call trapping method is its optimistic reliance on waiting for short intervals, hoping to get additional calls, resulting in sequential I/O. It is expected that the performance benefits of issuing sequential I/O requests will outweigh the disk idling time.

While both the methods discussed would achieve our desired results, they involve a significant amount of overhead. The overhead prevents them from yielding results comparable to our approach. In our approach, the application schedules itself using the semaphore primitive [2, 3] resulting in an elegant and efficient implementation.

III. Our Approach

When a user launches multiple cp commands in the parallel method, the OS starts a process for every one of the cp commands and schedules them. Figure 1 shows the simplified state diagram for each process. Once the OS has scheduled a new process, it enters the READY state waiting to be scheduled by the CPU scheduler. When it is scheduled, it enters the RUN state issuing its I/O request. It then enters the WAIT state until the I/O is complete. This cycle is continued until all the data has been transferred by the process.

Each of the multiple cp commands has its own instantiation of the process state model. The disk subsystem works continuously to meet the needs of each process and its disk head must switch constantly to service the requests from individual processes. The interleaving of each process's data stream causes the disk head mechanical switching overhead to increase proportionally with the number of processes. Our idea proposes to minimize the amount of disk head switching, decreasing the total time required to copy all the files.

➢ New Copy

In our approach, referred to here as "New Copy", we enhance the performance of the cp command by ensuring that when the user launches multiple cp commands on large files simultaneously (parallel copy), only one process may issue the I/O request. Until the copy operation completes, the other copy command does not issue any I/O request, minimizing the need for disk head switching. Figure 4 visualizes the operation of New Copy. The processes are allowed to be created in parallel but the process execution is forced to be serial.

New Copy serializes the copy command using the Interprocess Communication Primitive ***semaphore*** (IPC System V). Semaphores are best described as counters used to control access to shared resources by multiple processes. We used them as a locking mechanism

to prevent cp processes from issuing an I/O request while another cp process is performing I/O. The copy command was modified to acquire this semaphore and is the base for the New Copy command. When the New Copy command runs on the system it ascertains whether the semaphore has been created. If it has not been created, the copy program creates the semaphore using a global key. Since every instance of the New Copy command knows this key, they are able to acquire the semaphore using the same key. The semaphore is acquired, the file is copied and then the semaphore is released.

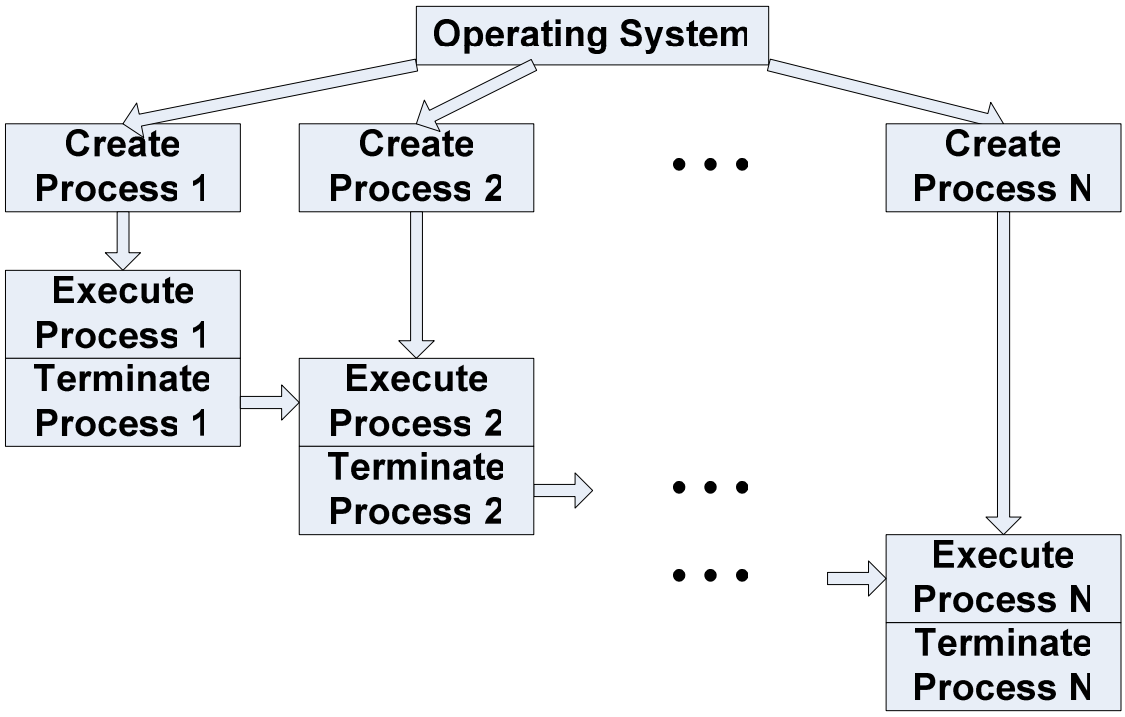

Figure 4: Execution scenario of processes in New Copy

Operating Systems address deadlocks caused when the process that acquired the semaphore crashes or is killed before releasing the semaphore. Linux protects against this problem by maintaining lists of adjustments to the semaphore arrays. The idea is that when these adjustments are applied, the semaphore is returned to the state that it was in before the process' set of semaphore operations were applied. This protection method is accomplished using the SEM_UNDO flag. Figure 5 shows the pseudo code for the original copy command in black and shows the additional pseudo code of our modified copy command in green. The code shows the process acquires the semaphore before performing the actual copy. After completing the entire I/O, it releases the semaphore.

```
#define KEY 12345
int main(int arc, char **argv) {
      int semid;
      …..
      Perform validity checks
      …..
      if semaphore_not_created
      then
            semid = create_semaphore(KEY ….);
            init_semphore(semid);
      else
            semid = open_semaphore(KEY ….);
      end if
      lock_semaphore(semid);
      do_actual_copy();
      unlock_semaphore(semid);
      return 0;
}
```

Figure 5: Pseudo code for copy command. Black is original copy command code. Green indicates changes made to original command for our approach.

IV. Experimental Results

The experiments were conducted on a laptop with the following specifications:
Processor: Intel Pentium M-740 (1.73GHz)
Memory: 2x1GB OCZ DDR2 667 (PC2 5400)
Hard Disc: Western Digital Scorpio WD1200VE 120GB 5400 RPM ATA-6
Operating System: Ubuntu 6.10 (Edgy Elf), Linux Kernel version 2.6.17-11

When dealing with hard drives, there are a lot of factors that add noise to the measurements, like the position of the head, the physical location of the file to copy, etc. The measurements presented in this report are consistent, but may vary over multiple runs.

We copied files of 350MB each from an ext3 partition to a FAT32 partition and measured the time it took for each of the three copy methods to complete. We ran the experiment for 4, 8, 12, 16, 20 and 32 files. Table 2 below summarizes the results.

Number of Files	Set Size	Sequential Copy	Parallel Copy	New Copy
4	**1400 MB**	02:14	02:48	02:13
8	**2800 MB**	04:30	06:10	04:46
12	**4200 MB**	07:24	10:54	07:17
16	**5600 MB**	09:36	15:07	09:34
20	**7000 MB**	11:54	19:50	11:45
32	**11200 MB**	21:10	37:45	19:31

Table 2: Execution times for three copy methods with varying file set sizes.

As we predicted, New Copy significantly improves the performance of the parallel copy, and the improvement increases with the number of files. We also noticed that New Copy command performs better than the sequential copy of files. This difference is due to the fact that sequential copy launches one process at a time, executes it, and terminates it before launching the next one. New Copy launches all the processes at the beginning and then executes them sequentially. The performance difference is explained by the process creation and termination overhead which is parallelized in the New Copy implementation. Figure 6 shows the execution times for each of the three methods presented, clarifying the effectiveness of our approach.

Figure 7 shows a chart with normalized execution times in percentages, with 100 corresponding to the parallel copy completion time. The effectiveness of New Copy increases with the number of files to copy. This performance is the result of a dramatic reduction in performance of the original copy command.

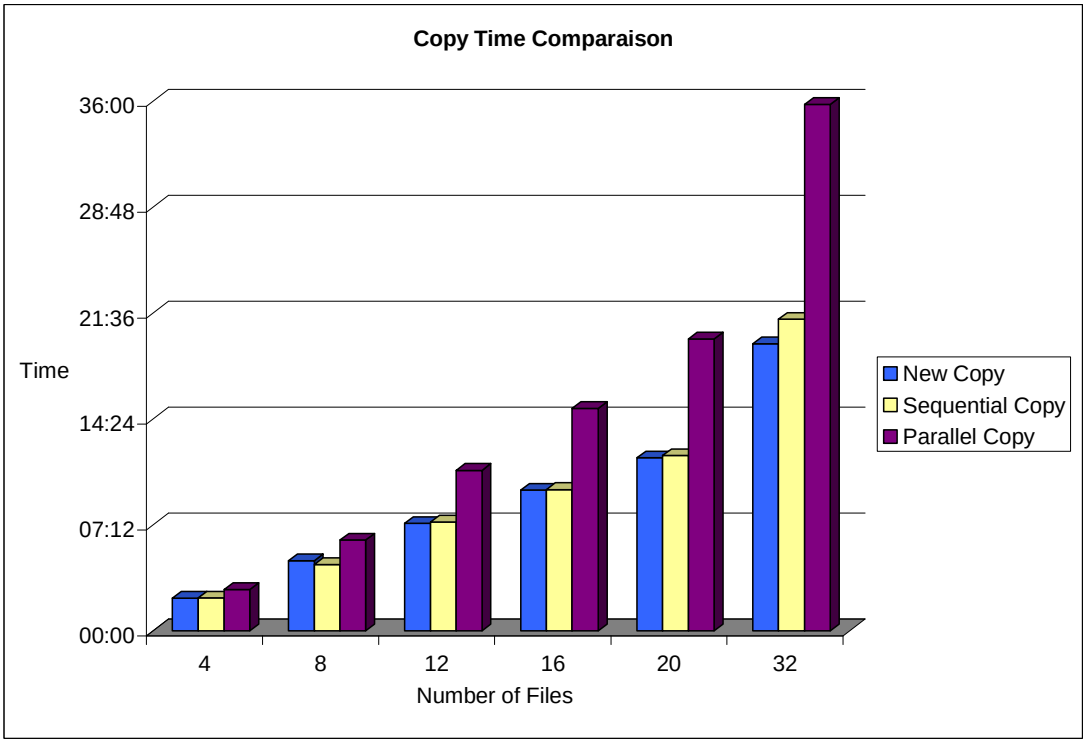

Figure 6: Execution times of the cp command in 3 scenarios: parallel, serial, and New Copy.

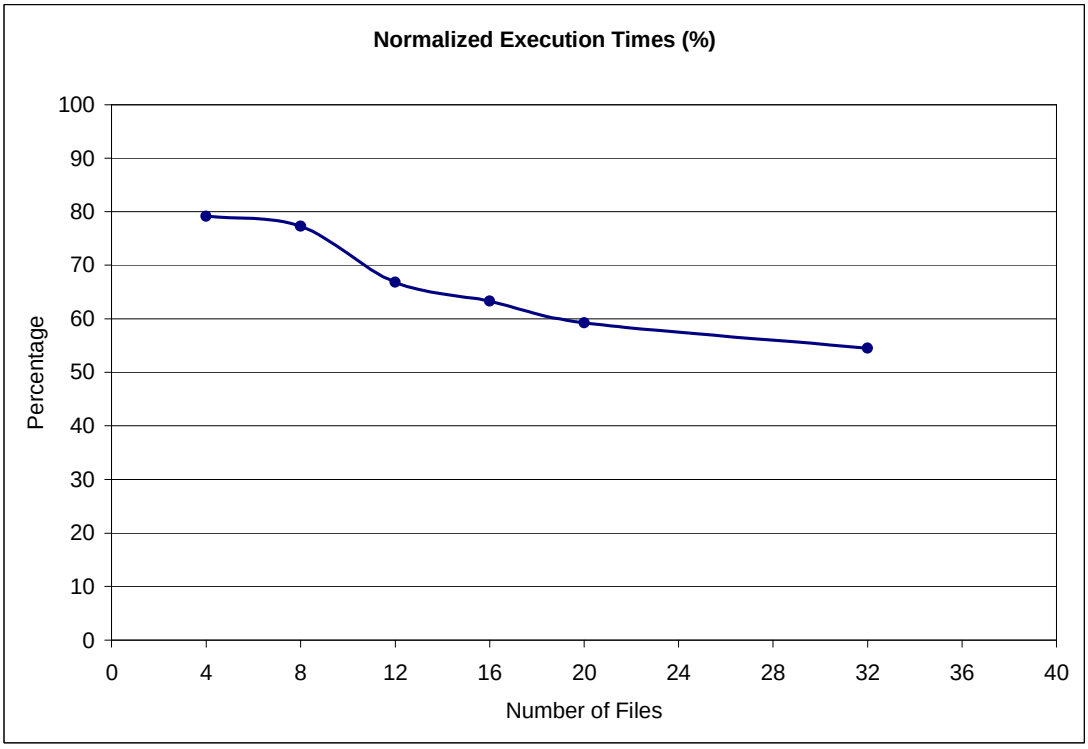

Figure 7: Execution time of the New Copy command normalized to 100% of parallel copy for 4, 8, 16, 20, and 32 file sets.

V. Extensions

While improving the overall performance, New Copy has some fairness issues. In a multiple file copy, a small file could be forced to wait until larger files complete, even though the copy time of the smaller file is a small fraction of the larger files. This issue is sometimes referred to as head-of-line blocking and is a common issue when designing network switches and schedulers. In addition, if we expand the usage of New Copy to multi-user environments where disk storage is shared, the first user to issue a cp command would block other user's copy operations due to the hard key in the semaphore. Also, New Copy will only allow one file to be copied at a time, even though multiple physical hard disks might be accessed by the different copy commands. The extensions we present below aim at solving these issues.

➢ Buffer Size

We first thought that changing the buffer size would enhance the performance, since increasing the buffer size would decrease the disk head switching by requesting larger portions of sequential data at a time. The original size of the buffer used in the read() and write() system calls is 32KB for any copy operation. The buffer size was varied between 16KB and 1MB. As shown in Figure 8, there was no appreciable change in the performance. We hypothesize this is due to the fact that requests issued by the kernel to the device driver are only of fixed data size. The I/O calls end up broken down into

multiple disk driver I/O calls, and interleaving still occurs. Again, the small variations are the result of measurement noise that was mentioned previously.

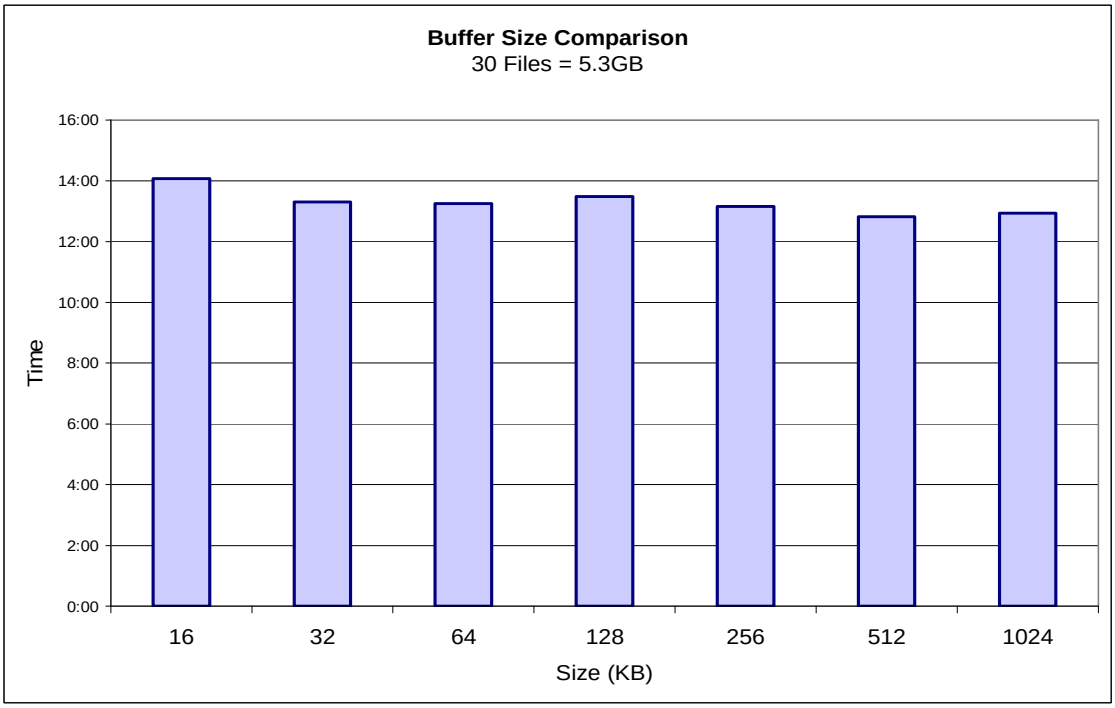

Figure 8: Comparing the performance of different buffer sizes

➢ Threshold

The head-of-line blocking problem, resulting from the New Copy implementation, can significantly increase the response time for users who need to copy small files for normal activities while concurrently copying large files. The copy completion time for small files has negligible impact on the performance gained by New Copy. To accommodate better small file fairness, files below a size threshold are allowed to copy in parallel. Above the size threshold, files become serialized by the semaphore. A threshold of 10MB was chosen because it is large enough to include a large array of high access rate files and is small enough to complete in a nominal amount of time. Figure 9 shows the performance of this implementation variation.

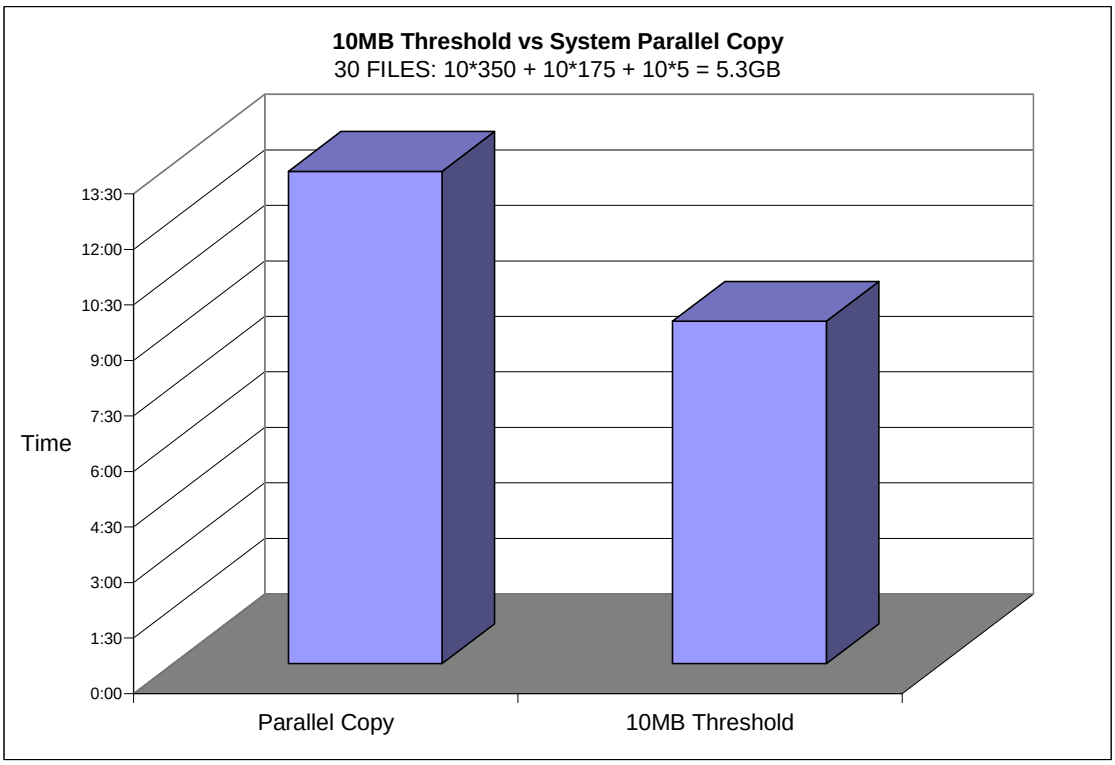

Figure 9: Comparing parallel copy with 10MB threshold variation

➢ Multi-Data Stream

Based on the previous idea, we considered having multiple data streams, allowing files to be grouped according to their size range. Each stream could have its own unique semaphore. The number of files copied in parallel would be equal to the number of streams. The multi-data stream method preserves the small file threshold, allowing these files to bypass the streams and copy in parallel. The idea behind this implementation was to increase fairness. Considering the original parallel copy being the most fair and the serial copy is the least fair, fairness will increase with the number of streams. Figure 10 shows the performance of three and four streams, compared to the 10MB threshold implementation (two streams) and the parallel copy. The thresholds we chose are 10MB (two streams), 10 and 50 MB (three streams) and 10, 50, and 150 MB (four streams).

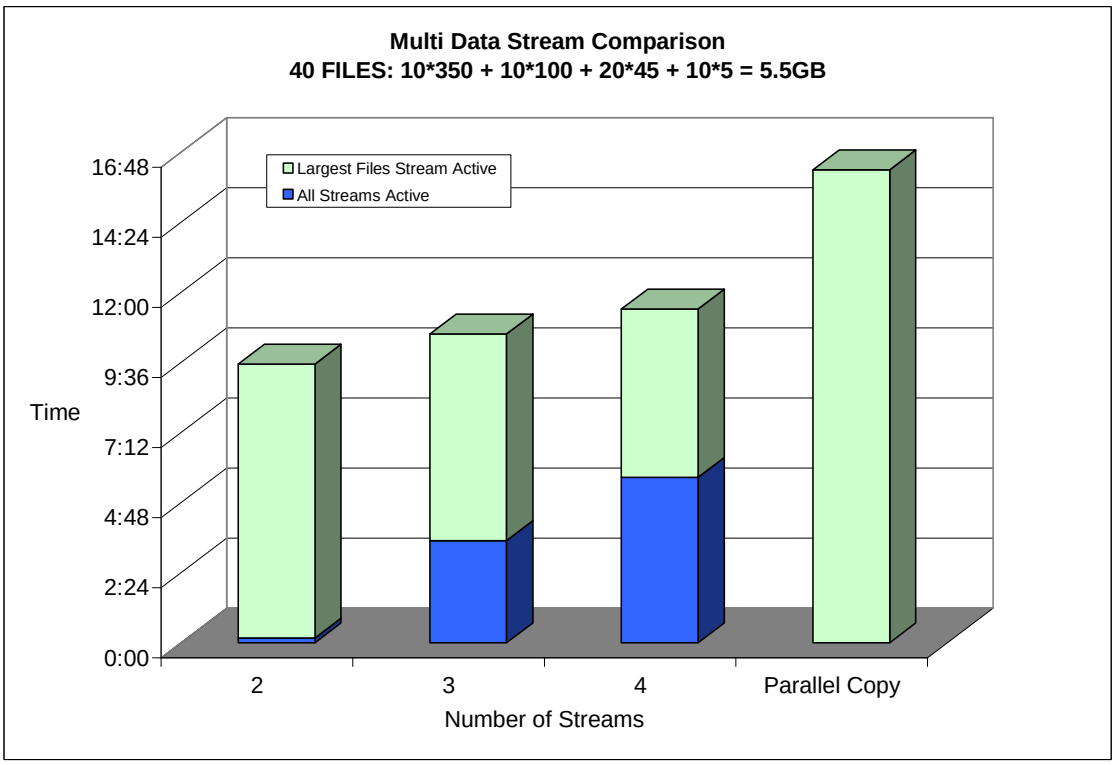

Figure 9: Comparison of multi data stream and parallel copy

It is apparent that the performance degrades as the number of streams increases. We also see that the smaller files complete long before the largest files, thus reducing the response time and time to complete of smaller files. The trade off point between fairness (response time) and speed (completion time) appears to occur at the two stream format.

➢ Multi-User Semaphore

To improve fairness in multi-user environments, we provide a unique semaphore for each user. At any given time, for 'n' users with 'm' streams, there might be 'n' semaphores and 'n x m' data streams copying in parallel. However, 'n x m' data streams would degrade the performance resulting in a large number of parallel semaphores on the system. This situation is similar to the original parallel disk head mechanical switching problem. If the number of users is especially large, to reduce the number of semaphores in the system, the system administrator could use one semaphore for every group of users. However, with several groups, they will still encounter intra-group fairness and head of line blocking issues.

➢ Source-Destination

If commands to copy files are issued to two different physical drives, they can possibly be executed in parallel without impacting the performance. New Copy serializes all the files, regardless of the physical drive used, and intrinsically may leave disks in a multi-

disk system idle unnecessarily. A normal parallel copy would execute the commands simultaneously, and if the data set if small enough, it can even finish faster than New Copy.

To alleviate this drawback, we implemented a multiple physical drive aware copy mechanism. If the source and destination disk are not already in use by another copy process, we run the copy process simultaneously with other processes. This method requires one data stream per disk and is accomplished by using a unique semaphore per disk. If the source and destination of a copy command are on different disks, the process has to acquire both semaphores in order to run. The algorithm is detailed in Figure 10.

1. Get File descriptor number from FILE struct
2. Use fstat() to get the stats about the file (src and dest)
3. Stat struct contains st_dev member which is the device number on which the file is stored
4. Reverse map the st_dev to the major and minor number of the physical device
5. From major and minor number, check to see if it is IDE or SATA drive
6. Accordingly find the physical drive on which the file is stored
7. Check to see if the physical disk used to store src and dest are the same
8. If src == dest then lock_sem(src)
9. If src != dest then lock_sem(src) & lock_sem(dest)
10. Copy the file
11. If src == dest then unlock_sem(src)
12. If src != dest then unlock_sem(src) & unlock_sem(dest)
13. Take care of deadlock while acquiring multiple semaphores by imposing an order in which the semaphores are acquired (eg: semaphores are acquired in the ascending order of their keys)

Figure 10: Algorithm to check, lock, and unlock source and destination disks

Drives are referenced by a major and a minor number [11]. The major number determines the device driver and the minor number indicates the drive and partition. The major number for the first IDE driver is hard-coded at 3. A full list of major numbers can be found in major.h [12]. The minor numbers have the format (<unit>*64) + <part> where <unit> is the IDE drive number on the first controller, either 0 or 1, which is then multiplied by 64. That means all hard disk drives on the first IDE controller have a minor number less than 64. <part> is the partition number, which can be anything from 1 to 20. The major number for all SCSI drives is 8. The minor number is based on the drive number, which is multiplied by 16 instead of 64, like the IDE drives. The partition number is then added to this number to give the minor.

We get the ID (st_dev) of the device from the stat structure using fstat():

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ...
};
```

Linux encodes the major and minor numbers using the formulation below to get st_dev:

```
    st_dev = (MAJOR(dev) << 8) | MINOR(dev);
```

To decode it, we extract the lower 8 bits from st_dev to get the minor number, and then right shift st_dev to compute the major number. From the major number, we determine if the device is IDE or SCSI, and then we can determine the actual physical device corresponding to the st_dev ID. Figure 11 highlights the performance gain of using per disk stream.

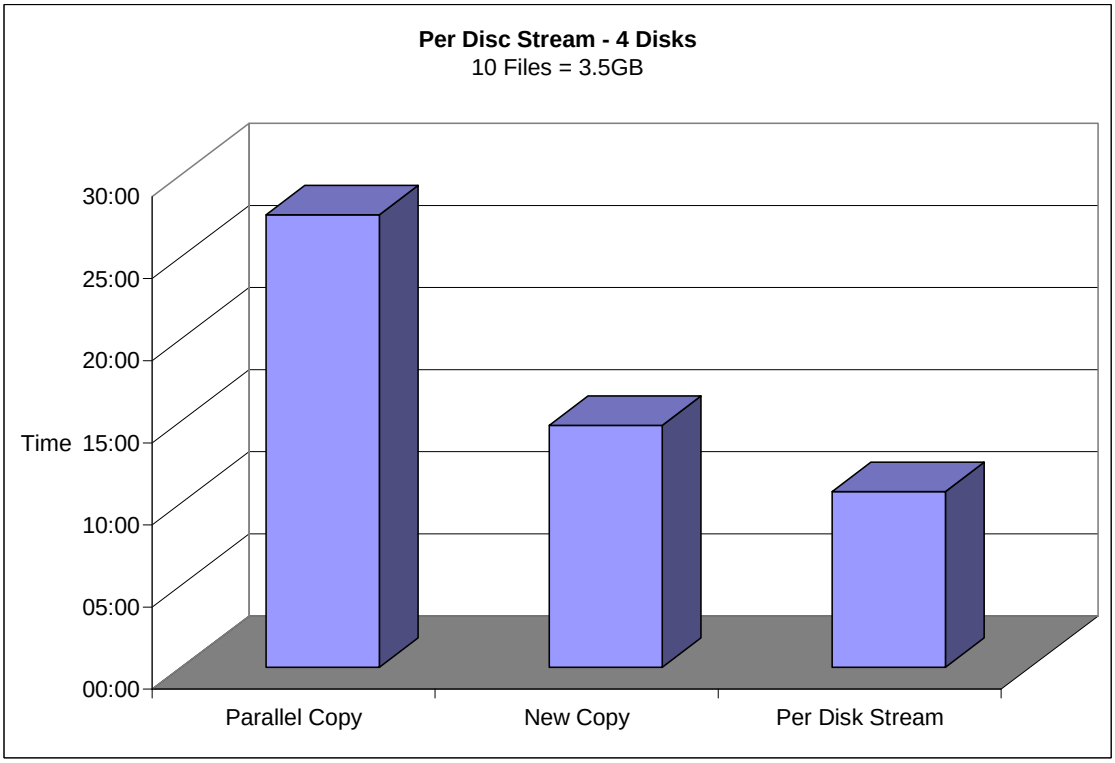

Figure 11: Comparing the 3 versions

➢ Summary of Results

The different attempted implementations demonstrate that the best performance is achieved by a per-disk stream implementation. Combining it with a 10MB threshold, we achieve an acceptable degree of fairness, with a negligible impact on the performance. We believe the copy command implemented combining both these techniques is very effective in an environment where users copy multiple large files simultaneously, even when performing backups or using video streaming applications.

VI. Future Work

We found that a copy/paste command from the GUI doesn't use the regular cp command in Linux. We can integrate our approach in the Linux GUI to improve user experience. Potentially, we can introduce a zero overhead copy. There are four boundary crossings from kernel to user space and redundant buffer copying occurs when using the read() and write() system calls to copy a file. If we fuse the two systems calls into one that performs the functionalities of the two, we are likely to achieve better performance. It involves the introduction of a new Linux system call in the ext3 file system as well as the modification of the VFS and VOPS structure of the kernel. Also, the redundant copying from user to kernel space and vice versa will be eliminated. These changes might prove very useful in the case of slow devices like USB based mass storage devices.

VII. Conclusion

In this case study, we discussed why it is useful to systematically serialize the I/O instead of using multiple data streams accessing the same disk. We presented a study of the copy command and focused on the differences between the sequential and the parallel copy. We explained how the excessive disk head switching degrades the performance of the copy command when copying multiple large files in parallel. We used semaphores to serialize the execution of the copy process in the simultaneous copying of multiple large files. The experimental results presented demonstrate that our idea is very effective. The performance of the copy command increased significantly. To achieve improved performance, we combined two techniques. We implemented a 10MB file size threshold, allowing files below this size to be copied in parallel. The implementation also utilizes streams on different disks to execute in parallel. We believe a copy command using these two approaches is highly effective in an environment where users copy multiple large files simultaneously, especially when performing backups or using video streaming applications. Additionally, it's an ideal replacement of cp in Linux, and we believe the average user will appreciate the performance gain and improved response time of his system while executing such operations.

References

1. http://www.gnu.org/software/coreutils/
2. http://db.ilug-bom.org.in/Documentation/lpg/node1.html
3. Advanced Programming in the UNIX Environment, Addison-Wesley by Richard Stevens
4. Solaris Internals: Core Kernel Architecture by Jim Mauro, Richard McDougall, Sun Microsystems Press
5. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O, Sitaram Iyer and Peter Druschel
6. Vnodes: An Architecture for Multiple File System Types in Sun UNIX by S.R. Kleiman, Sun Microsystems
7. An Efficient, Nonintrusive, Log-Based I/O Mechanism for Scientific Simulations on Clusters by Soumyadeb Mitra, Rishi Rakesh Sinha, Marianne Winslett and Xiangmin Jiao http://drl.cs.uiuc.edu/pubs/winslett/cluster05.pdf
8. Debugging and Performance Tuning with Library Interposers by Greg Nakhimovsky http://developers.sun.com/solaris/articles/lib_interposers.html
9. LXR / The Linux Cross Reference - http://lxr.linux.no/
10. Reverse map of the file descriptor to major and minor device number of physical disks: http://lxr.linux.no/linux/include/linux/kdev_t.h#L26
11. Major and Minor Numbers: http://www.linux-tutorial.info/modules.php?name=MContent&pageid=94
12. Full list of major numbers: http://lxr.linux.no/linux/include/linux/major.h
13. Ligon, III, W.B., and Ross, R. B., "PVFS: Parallel Virtual File System," Beowulf Cluster Computing with Linux, Thomas Sterling, editor, pages 391-430, MIT Press, November, 2001.
14. Peter M. Chen , Edward K. Lee , Garth A. Gibson , Randy H. Katz , David A. Patterson, RAID: high-performance, reliable secondary storage, ACM Computing Surveys (CSUR), v.26 n.2, p.145-185, June 1994.