Multi-threading in Disk Cloner

S.Panichprecha
Epiphany Consulting Co., Ltd.
422 Sukhumvit Rd., Klongtoey
Bangkok, Thailand 10110
sorot.p@gmail.com

N. Pongsawatkul*, C. Mitrpant, U.Ketprom
Digital Forensics Technology Laboratory
National Electronics and Computer Technology Center (NECTEC)
112 Phahon Yothin Rd., Klong 1, Klong Luang,
Pathumthani ,Thailand 12120
*nared.pongsawatkul@nectec.or.th

P. Kovintavewat
Data Storage Technology Research Unit
Nakhon Pathom Rajabhat University
Nakhon Pathom, Thailand 73000
piya@npru.ac.th

***Abstract*—Disk cloner is a device for duplicating or creating a bit-by-bit copy of a hard disk. The current methodology for cloning a large disk is inefficient which results in a very slow speed of such a cloning process. The program called dd is often used for this duplication purpose. The cloning speed depends on both hardware and software. This paper aims to utilize the technique of multi-threading on multi-core processors to enable parallelism between reading and writing data in the disk cloning processes. Comparing to the existing dd program, our multi-threading program can perform the disk cloning function with more than 40% improvement in transfer rate or speed.**

I. INTRODUCTION

Hard disk duplication is a process of making copies of the data stored in a hard disk. Such a process is commonly used for i) Making backups of data for safekeeping of the original data, ii) Duplicating data for setting up multiple computers that have the same configuration (e.g., in a lab or an enterprise environment), and iii) Making copies of data stored in digital evidence for digital forensics.

Digital forensics is the process of acquiring, verifying, analyzing, and storing evidence in digital form that resides in a device, such as a personal computer, a laptop computer, or a mobile phone. This digital data can be used to identify an offender (or a wrongdoer), or it can be considered as evidence in a court of law. The digital evidence is very sensitive because it can be modified, damaged or destroyed if precautions are not carefully taken. Also, unlike “hard” evidence such as fingerprints, the digital evidence can be easily hidden and altered. Therefore, the key procedure in the digital forensic process is to create exact copies of the original evidence. Then, only copies will be used for further analysis instead of using the original evidence. In other words, digital forensic investigators must preserve the integrity of the original evidence. In practice, copies of original evidence can be created using several tools of which the most common one is the dd program (part of the GNU coreutils) [1][2]. The dd program is widely accepted as a tool for creating copies of digital evidence due to the fact that dd preserves data integrity and guarantees that the duplicated data is the exact copy of the original data [3]. The dd program creates duplicated data in a bit-by-bit manner. Once a duplicated disk has been created, a forensic investigator can use additional tools to verify the integrity of the duplicated disk. Such verification can be achieved using a hash function to compute the massage digests of the original data and the duplicated data. There are some variations of the dd program, e.g., dcfldd and dc3dd, that provides an on-the-fly integrity check. These variations produce the message digest of the input data when the program finishes [4][5]. If these two message digests, from the original disk and from the duplicated disk, are equal, we can imply that the duplicated disk is same as the original disk [6]. The dd program duplicates data in a bit-by-bit fashion, where the entire disk even empty spaces are copied. Thus, cloning a large disk will result in an image file which has the same size as the original disk. Due to the sequential bit-by-bit methodology used in dd, it takes a long time to duplicate a large disk

Disk duplication can be a very long process. The time taken to duplicate a disk depends on the speed of the disk and the disk controller bus throughput. Hence, in terms of hardware, there is very little we can do to improve the cloning speed. Even when using the state of the art in hardware, we can only improve the transfer rate up to a certain point. Thus, it is necessary to start pushing the boundary through enhancing the software. The existing disk cloning software, i.e., dd, was implemented as a single-thread program. Therefore, we see that we can improve the speed of the dd’s disk cloning process by utilizing the multi-threading. The main purpose for the multi-threading/processing is to resolve the resource sharing between read thread and write thread when accessing the same resource using the disk cloning software (dd).

II. BACKGROUND

A. *DD Internal working mechanism*

The dd program is commonly found in UNIX operating systems. According to the GNU coretuils manual page, dd is a program for converting and copying a file [3]. However, in practice, dd can be used to perform much more than just copying files. One of the most important capabilities of dd is that it can create a bit-by-bit copy of a hard disk. Due to such a capability, dd has been used widely in the digital forensic community to create copies of digital evidence. In other words, the dd program produces the exact copies of the original evidence. The copies of the original evidence will be used during the forensic analysis so that the integrity of the original evidence is preserved.

There are a few commonly used options when a user invokes dd. These options are as follows:

- The input file (`if`) and output file (`of`) options specify the name of the file to be read (source file) and written (target file) respectively. The arguments to the input and output file options can be a device name (e.g., /dev/sda) or a file name.
- The block size (`bs`) specifies the size of the data to be read and written each time. The block size has a significant impact on the performance of dd. If the size is too small the program performs slowly because it has to read from the disk multiple times. If the size is too large, the program may not gain a performance improvement. The calculation of an optimum block size can be in and of itself another topic; hence, such a calculation is not discussed in this paper.
- The count (`count`) argument specifies the number of blocks of data to be read and written where the size of the block is defined by the bs argument. Note that the numbers of read and written blocks of data are always the same number.
- The skip (`skip`) argument specifies the offset of the input file where the read starts. The value of the skip is the number of block offset from the beginning of the input file where the size of the block is defined by the bs argument.

From the arguments listed above, a dd command may be invoked like this: `dd if=/dev/sda of=image.dd bs=512`. This command creates a bit-by-bit copy of the whole disk (hence the count and skip are not specified) where the source disk is the device `/dev/sda` and the output of the program is the file called `imaged.dd`.

The internal working mechanism of dd is quite simple. We have analyzed the source code of the dd (which is part of the GNU coretuils version 7.6 [8]) and simplified the working mechanisms as follows:

Step1: Validate the command line arguments. This step sets the variables related to the input and output file, the blocks size, number of counts, and the skip values.
Step 2: Open the input file and set the input offset (or skip) if it needs be.
Step 3: Open the output file.
Step 4: Start the timer. This step starts the timer where the time spent running the program and the speed (bytes per second) are reported when the copy routine finishes.
Step 5: Allocate memory portions for the input buffer and the output buffer.
Step 6: Read from the input file and write to the output file. Repeat the read and write subroutines until the read subroutine reaches the end of file (EOF) or number of read blocks reaches the `count` argument.
Step 7: Close the input and output files.
Step 8: Report the statistic of the program, i.e., the time spent and the speed (bytes per second).

From our analysis of the working mechanisms listed above, we have found that the dd spent the majority of the time at Step 6. We have also found that the read and write subroutines are performed sequentially. The speed of the overall program can be improved if we can perform the subroutines in parallel. Therefore, the multi-thread programming theory has been used in our work to improve the speed of dd.

B. *Multi-Thread Programming*

Most modern personal computers nowadays are equipped with multiple logical Central Processing Units (CPUs) with one physical CPU. Such a CPU is called a multi-core CPU. These multi-core processors allow a computer to perform multiple threads (small instructions) simultaneously. Intuitively, performing multiple tasks at the same time will significantly improve the speed of software. However, the utilization of multiple threads have not been used widely due to a few reasons, i.e., software must be modified (or rewritten) to support multi-thread. Also, in many cases, software developer is required to support legacy hardware that is not capable of performing multiple threads simultaneously. In addition, using multiple threads raises an issue with resource sharing where one thread is using a resource and the other thread tries to access the same resource.

From our experiments which will be presented later in this paper, we found that by converting the software into a multi-thread capable, the overall speed of the program has improved significantly. As discussed above regarding the issues of migrating software to multi-thread programming. We had issues with resource sharing. The detail of the issue will be discussed later.

In order to implement a multi-thread program, developers generally use the existing common application programming interface (API) to save their development time. There are several APIs for implementing multi-thread program, to name a few POSIX Threads and OpenMP [8] [9].

III. Multi-threading Disk dump

C. *Multi-threading Disk Duplication*

In our experiments, we use OpenMP (Open Multi-Processing) as an API in our multi-thread disk duplicating program. The program carries out the tasks described in Steps 1 - 8 in the previous section. At Step 6, the program creates two threads: a reading thread and a writing thread. The reading thread reads data from the source specified by a user through '`if`' or input file option, and the writing thread writes data to the target specified by the user through '`of`' or output file option.

Since reading data from a hard disk generally takes less time than writing data to a hard disk, we create a buffer for the program to temporarily store data read in by the reading thread (T0). The data sits in the buffer until it is written to the target by the writing thread (T1) as illustrated in Fig.1.

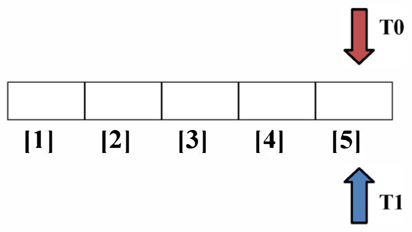

Fig. 1 Buffer in multi-threading processes

The reading thread puts the source data in the buffer data block sequentially from left to right. Once the reading thread reaches the end of the buffer, it goes back to the beginning and continues putting data in to the buffer.

Concurrently, the writing thread takes data sequentially from the right of the buffer to the left and writes the data to the target. Once the writing thread reaches the end of the buffer, it goes back to the beginning and continues taking data from the data blocks.

If the two threads worked independently, we would face a problem of data synchronization because the reading thread might overwrite the part of the buffer that has not been written to the target by the writing thread. On the other hand, the writing thread might take data from a data block with staled data.

We prevent the data synchronization error by imposing conditions on when the two threads can access the buffer. That is the movement of the two threads along the buffer is sequential, and the reading thread is not allowed to access the buffer when the buffer is full while the writing thread is not allowed to access the buffer when the buffer is empty. The buffer is full when all the data blocks in the buffer contain data that has not been written to the target. The buffer is empty when all the data blocks in the buffer have already been written to the target.

IV. Experimental Results

We prepare the experiment by setting up the process with two threads to work in parallel. The first thread is READ; its function is to read data from the source and transfer it to a buffer memory with the pre-determined size, e.g., 64 MB. The buffer memory acts as a parameter for threading process. The second thread is WRITE; its function is to read data from the buffer memory and write the data to the destination following the pre-determined size. In this experiment, we set maximum of five parameters for buffer memory.

The input data for the experiment is set to be 1,5,10 and 20 gigabyte (GB). For each input data size, we set up '`count`' option in dd program accordingly. For 64 MB buffer parameter, an input of 1 GB requires setting up '`count`' option to be 16, and for 20 GB, '`count`' option to be 320. An example of cloning 1 GB of the input data is given by deploying the following dd command:

TABLE I
DD Command

```
dd  if=/dec/sdb  of=/dev/sdc  bs=64MB  count=16
```

The experiment for each input is performed repeatedly several times to calculate an average of the transfer rate. In order to test the transfer rate for different hard disk interfaces, we explore the IDE to IDE interfaces and SATA to SATA interfaces. Most current systems will use SATA but IDE is quite common in old systems. Therefore, it will be useful to study both interfaces.

D. *IDE-IDE*

This IDE-IDE connection is the setup between IDE-interface of source hard disk to the IDE-interface of target hard disk. As shown in Table II, experimental results agree that the transfer rate of multi-threading (MT)-dd is faster than regular dd. The increasing rate is between 43%-48% depending the input data size. Even though the transfer rate for the large-size input data will be a bit slower, it is an insignificant amount of few megabytes per second (MB/s) difference. In term of time spent cloning each input, the cloning time increases linearly as the input size increases as illustrated in Fig.2. Therefore, the difference in time spent cloning 20 GB is much more significant than 1 GB. Using multi-threading-dd, we can save approximately 187 seconds or approximately 3 minutes out of 20 GB cloning.

E. *SATA-SATA*

SATA-SATA connection is commonly used in a high performance system. Comparing with IDE, the transfer rate of SATA-SATA connection is usually double of the IDE rate. The difference in input size has no effect on the average transfer rate as seen in Table III. Each input has the transfer

TABLE II
COMPARISON OF TRANSFER RATE FOR IDE CONNECTION FOR DIFFERENT INPUT

IDE – IDE				
Prog. / Size (GB)	dd		Muti-threading-dd	
	AVG Time (sec)	AVG Rate (MB/s)	AVG Time (sec)	AVG Rate (MB/s)
1	25.8861	39.56	18.1065	56.58
5	132.7290	38.56	90.6395	56.50
10	272.7912	37.56	184.5390	55.46
20	571.5106	35.84	384.6210	53.24

TABLE III
COMPARISON OF TRANSFER RATE FOR SATA CONNECTION FOR DIFFERENT INPUT

SATA – SATA				
Prog. / Size (GB)	dd		Muti-threading-dd	
	AVG Time (sec)	AVG Rate (MB/s)	AVG Time (sec)	AVG Rate (MB/s)
1	14.3621	71.32	10.1556	100.94
5	72.0524	71.08	49.0263	104.60
10	143.9062	71.18	97.4820	105.00
20	287.5482	71.22	194.5674	105.00

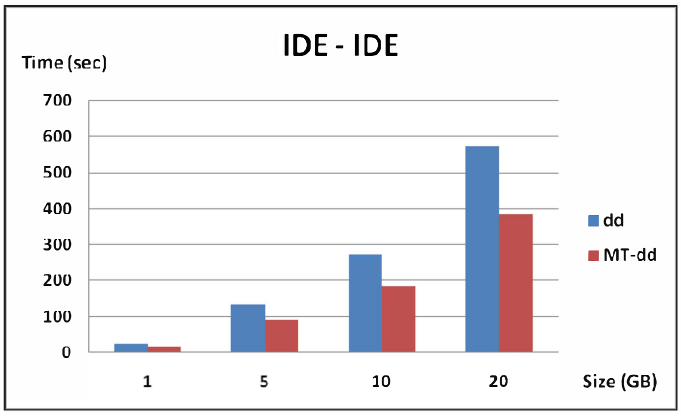

Fig.2 Comparison of IDE transfer rate between dd and multi-threading-dd (shorter bar means faster)

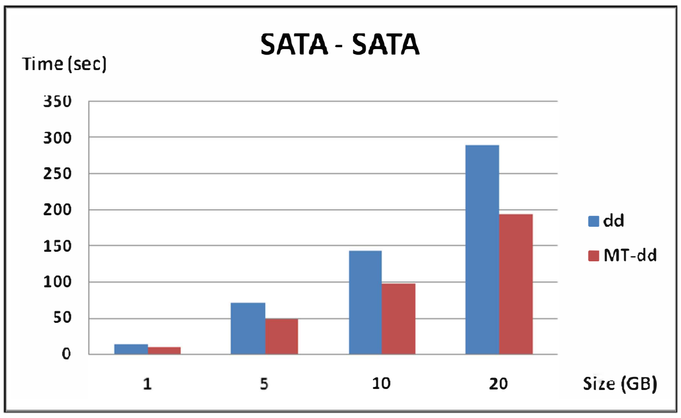

Fig.3 Comparison of SATA transfer rate between dd and multi-threading-dd (shorter bar means faster)

rate approximately at 71 MB/s using the regular dd. Utilizing multi-threading-dd, the average transfer rate increases to be 105 MB/sec or at about 48%. In terms of time spent cloning, the SATA connection is more appealing than IDE connection because it takes less than 2 minutes to clone the 20GB input using multi-threading-dd. Therefore, it is worth spending more money on SATA interface-hardware in order to shorten the cloning time. Similar to previous results with IDE-IDE, SATA-SATA cloning time is also linearly dependent on the size of the input. Therefore, the time spent cloning increases linearly with the input size regardless of the connection type as illustrated in Fig.3.

V. CONCLUSION AND FUTURE WORKS

With the prevalence of multi-core processors, multi-threading programming is brought in to enable parallelism between the data read and write in the disk cloning processes. A buffer for temporary data storage is shared by the reading and writing threads. A buffer access management strategy is implemented to ensure the correct order of data duplication. Comparing to the existing dd program, our multi-threading program can perform the disk cloning function with more than 40% improvement in transfer rate or speed. Further speed improvement could be explored by optimizing the size of the buffer data block, the size of the buffer and the sleep time of the reading threads.

ACKNOWLEDGMENT

This work was supported by a research grant DSTAR-R&D 02-01-52 from I/UCRC in Data Storage Technology and Application Research Center (D*STAR), King Mongkut's Institute of Technology Ladkrabang, and National Electronics and Computer Technology Center (NECTEC). We would like to thank P.Waraporn, S. Makwimanloy, P. Makhapun, S.Laokok, S. Jantarapatin, N. Phookesorn for their great helps on disk cloner development. We also appreciate comments from our colleague and students from Nakhon Pathom Rajabhat University and Thammasat university.

REFERENCES

[1] Gabriela Limon Garcia, "Forensic physical memory analysis: an overview of tools and techniques TKK T-110.5290", Seminar on Network Security, October 2007.

[2] Paul Rubin, David MacKenzie, and Stuart Kemp. dd: GNU coreutils. dd manpage, September 2010.

[3] National Institute of Justice, "Test Results for Disk Imaging Tools: DD GNU fileutils 4.0.36, Provided with Red Hat Linux 7.1"

[4] Dcfldd: http://dcfldd.sourceforge.net/

[5] Dc3dd: http://dc3dd.sourceforge.net/

[6] DD and Computer Forensics – Deuce by Thomas Rude, CISSP : http://www.crazytrain.com/dd2.html

[7] Blaise Barney. POSIX Threads Programming. Tutorials UCRL-MI-133316, Lawrence Livermore National Laboratory, October 2010.

[8] Paul Rubin, David MacKenzie, and Stuart Kemp. dd.c. dd program as part of GNU coreutils version 7.6, 2009.

[9] OpenMP Architecture Review Board. OpenMP: API Specification for Parallel Programming. http://openmp.org, 2010.