Journal of Software Engineering and Applications, 2017, 10, 354-369
http://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Esext3 for Reducing the Effect of Harboring Bug in File System

Raza Muhammad, Basheer Riskhan

School of Computer Science, Huazhong University of Science and Technology, Wuhan, China
Email: razacom_2000@yahoo.com

How to cite this paper: Muhammad, R. and Riskhan, B. (2017) Esext3 for Reducing the Effect of Harboring Bug in File System. *Journal of Software Engineering and Applications*, **10**, 354-369.
https://doi.org/10.4236/jsea.2017.104021

Received: February 25, 2017
Accepted: April 24, 2017
Published: April 27, 2017

Abstract

One of the most critical threats to the reliability and robustness for file system is harboring bug (silent data corruption). In this research we focus on checksum mismatch since it occurs not only in the user data but also in file system. Our proposed solution has the ability to check this bug in file system of Linux. In our proposed solution there is no need to invoke or revoke checker utility, it comes as the integrated part of file system and has the ability to check upcoming updates before harboring bug make unrecoverable changes that leads significant data loses. Demonstration testing shows satisfactory results in file server and web server environments in terms of less memory consumption and avoidable delay in system's updating.

Keywords

Harboring Bug, Ext3, Esext3, File Server, Web Server, Database Server

1. Introduction

File system is the major part of computer system which must be robust, reliable and intelligent enough that can handle faults and bugs. However, issues and problems still occur and machine can be crashed due unavailability of robust solution. Just like hardware failures, software bugs such as harboring bug in the system. The UNIX system has usually effected to split all obtainable resources to the most degree likely. Therefore, a single user in all users can assign all the accessible space in the file system. In many environments this is unacceptable because if admin assigned whole file system to one user, what about other users? As a result, a share mechanism has been added for limiting the amount of file system resources that a user can acquire [1].

Collision-resistant cryptographic hashes for meta-data have the ability to defend the journaling of file system by making it secure and increase the reliability

and integrity of security in journaling file system [2]. Fixed file system journaling defends the ordering of meta-data processes to maintain consistency in the occurrence of crashes. Though, journaling does not defend significant system meta-data and application data from modification and falsification by damaged or wicked storage devices [2].

The long structured memory policy has two types of benefits that can increase the speed and clean the memory from garbage that is responsible to make latency in file read and written in primary memory. It also increases the speed of the bandwidth of memory up to 6×. It has the ability to reduce the cost of memory by its normal operations that has the ability to run concurrently with other operations [3]. Unfortunately, by the time to time increase of disk size (nearly 2 to 3 times) leads the traditional recovery techniques to consume (2 to 3 times) more disk space for backups in computer system and responsible for putting more delay in recovering process. In most cases, the system's availability is critical, so machine required the mechanism that will avoid the need of expensive recovery in terms of memory and time costs.

The fault tolerance and rapid fault discovery for both hardware and software is key structure blocks for building always accessible applications. But many structured software is not enough in their own to achieve the task of removing bug and corruption that leads towards crash in system. Likewise, built-in error revival is significant for handling the rising rate of fleeting hardware and software errors. But for designers of highly accessible systems having challenges those lies in addressing the problem such as the environment, cyber-attacks and silent data corruption [4]. Hashing is an important technique that has the ability to come over scalable retrieval problems [5], and it is mostly involved in hash code generation and hash functions learning. MDS also has a unique feature but can create problem when it is used for long length code so some scientist offered XI-Code in [6] to resolve this issue.

2. Related Work

In memory the phase change is also very big issue. Some scientist compared different types of storage to observe the performance that include the latency, input and output of disk and the performance of cache memory [7] [8]. The workstation file system for the Cedar programming environment was customized to get better and include robustness so performances become high. Formerly the file system used hardware-provided labels to the disk blocks to add robustness in hardware and software to prevent errors and bugs in the system [9]. The distributed database with high through put can only be done by a special type of file system which is completed dedicated for this purpose. ClvinFS is the scalable and robust enough to handle the distributed database system standard. This file system has the ability to work with semantics that can including fully linear by the help of random writes by handling concurrent users for the flow of random byte offsets as theses are in same file and it may be located across wide geographic areas [10] [11].

EXPLODE can comprehensively test storage systems by acclimatizing key designs from model scrutiny in a way that keeps their authority but removes their intrusiveness. Its crossing point lets programmer and designers rapidly program storage managers and also with normally compile them from existing modules. These system checkers can be executed on live and online system that represents not to imitate moreover, the environment also pieces of the system [12]. The Heavily-tested the file systems that was modeled by the scientists checked were and the harshness of the errors establish it emerged that model checking exertion good in the context of file systems. The fundamental ground for its efficiency in this context appears to be since file systems must do so many multifaceted effects correct [13].

Most of the operating system has the ability to fight with any harmful behavior from device and from program. Some scientists investigate the inter-operating system problem, which can increase in terms of amplified cost with the input and output virtual addresses allocator, which regularly encourages linear involvedness [14]. Optimization in all the operating system is present but the behaviors of all of these are very different. Some have the ability to optimize all by the help of one-to-one mapping system for their metadata file system. A new mapping system that is known as many-to-one has the ability to improve the file system optimization up to 27% in total [15]. Time series process- ing need more time to make sample in file system on high sample rate in database system which is very difficult to handle it. The design of database is usually based of different time series that can easily respond in the development of different technologies in the internet of things [16]. QoS is also a significant feature of any network based architecture some scientist [17] suggested very useful and efficient solution to overcome QoS issue.

3. Significance of Esext3

Based on our pervious literature review we noticed that the ext3 files system has poor performance due to its same allocation phenomena for both indirect block and data block by using tree structure in system. So through these strategies fsck faces more difficult to find harboring bug in system. Esext3 file system is based on Ext3 file system. This architecture contains a number of changes to improve not only speed also improves the memory consumption in file system. In proposed architecture the checker is integrated part of the file system so there is no need to invoke or revoke the checker. It works automatically when updates are ready to install in any field of the file system. Our proposed checker has the capable to report the presence of harboring bug in upcoming update in file system. Moreover, it uses XOR operation for checking harboring bug as we know that XOR is one of the most accurate and reliable operation among all operations those are used to check checksum mismatch.

Thanks to our proposed architecture of Esext3 file system which has the ability to works on TDMA (time division multiple access) that has the ability not only prevent harboring bug in file system also capable to do multitasking that is

not available in Ext3 file system of Linux operating system. It also has the ability to stop upcoming updates and report harboring bug to the server and start updating system where it broke sequence when it receive harboring bug free update. This checker also has the ability to rollback and restores updates, if it found that server is not able to send correct and bug free update. We expect that Esext3 to meet the following criteria to reduce the memory consumption in the file system. While Ext3 which is limited roughly 5% of the main disk speed, we expect that the Esext3 can scan system with the greatest possibilities. As we saw in previous model that the performance dropped very quickly as the file sizes grow and the age of file also played an important role in delay scanning. We believe that this new file system checker can check on a constant speed. It will also allow administrator to decide when to execute the checker in the system. To repair file system on responsiveness cannot come together in productive environment. We focus to make sure that our file system can perform better than Ext3 file system.

The checker in file system scans the system in mean of ascending order (from 1st area to last area). First the checker check each group and their respective inode if found any unnecessary cross reference it will immediately discard it. Then it will read corresponding indirect region for self check if it found any cross reference metadata it will again discard it. These discarding of cross reference reduce the possibility of unwanted rescan of indirect area of corresponding field. After these operations the checker will authenticate the number of blocks including their size and check harboring bug by using XOR checksum in each layer of upcoming updates.

4. Design and Implementation

In this section, we describe the design and implementation of Esext3 file system. Esext3 stands for Extended Secure Ext3. It is the new secure version of Ext3 as shown in Figure 1. Most of the fields are same as Ext3 but we introduced a new field that is known as "Checker". It has the ability to prevent Harboring bug not only in Metadata of Data bitmap but also can remove Harboring bug from whole Esext3 and also reduce the memory consumption. It works on TDMA mecha-

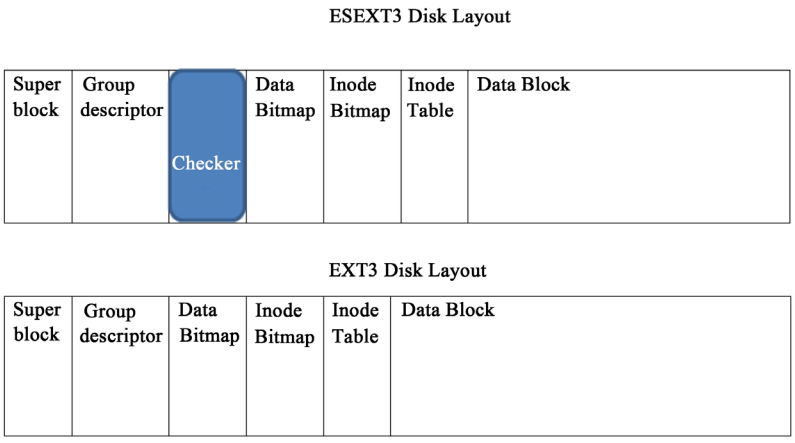

Figure 1. Disk layout Esext3 vs. Ext3.

nism that allows it to divide time in different updates. It uses intelligent TDMA for example if it gives 5 micro seconds to one update and update need 3 micro seconds, it will give only 3 micro seconds to it and go forward. It will not create its own backups and roll back system; it will use the traditional mechanism of restore point of OS. So we can save memory and time.

Suppose, system needs to update Data Bitmap, in our new system of Esext3 the system cannot update the field. System has to send data to checker and first checker scans that files and forward towards data bitmap for updating. If the checker found any harboring bug in system, it will stop and send back only that area of file that contain the bug and start entertaining other update if available. Thanks to its mechanism works on TDMA (Time Division Multiple Access) that can save time. In the meantime if system sends back the correct file it will start updating data bit map from same place where it found bug. It will use the restore system of OS so it can also help to reduce memory and time consumption.

Figure 2 states the finite states automate of Esext3 file system. In Figure 2, "S0" is the system that needs to update "S2" that is the particulate field in Esext3 file system. "S1" is the checker that will check the system for bug. Whereas "Q1" is the mean system is checking Esext3 field for update is needed or not. In "Q2" system is sending update to checker for bug checking if checker not found bug in it checker will send update towards "S2" (field of Esext3) for further action. If checker found harboring bug in update it will stop and send back the particular area of update to "S0" (system) for resend the update.

In Table 1 we have described the mechanism of scan and read-through jobs of our checker. This scan job is divided into 5 segments. In first segment the checker

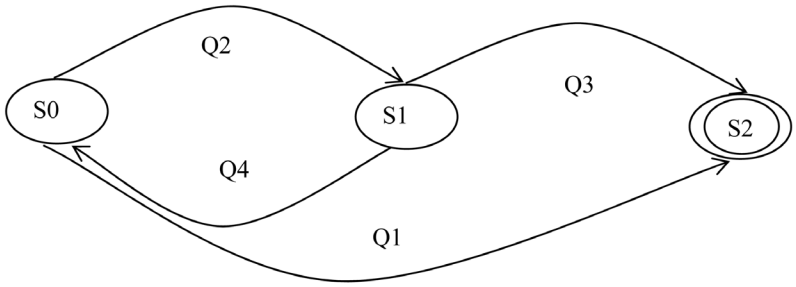

Figure 2. Finite state diagram of checker. Whereas: S0 = system (sender of update); S1 = Checker of Esext3; S3 = Esext3 filed that to be update; Q1, 2, 3, 4 = show the flow of data.

Table 1. Segments of checker.

Segment	Scan and read-through job
1	Scan all fields of the Esext3 files system if found duplicate clime then rescan system and define the ownership of the file need to be updated
2	Separately check each directory for harboring bug before update the filed
3	Check the harboring bug in intercommunication between all directories
4	Check meta-data reference area if found harboring bug remove it and restore original file
5	Update file and field if necessary

scans all fields in the file system if it found any duplicate clam then it will rescan the entire field in system. In the second segment if it found harboring bug in a single directory it scans each directory separately before the updating any field. In the third segment if the directories are interlinking with each other and having intercommunication and these files and directories also need updates so before update checker scans these files and directories. In the fourth segment checker also check the meta-data reference area for harboring bug if found it will remove it and restore original files and directories. In the last fifth segment the system will update all the necessary filed in the system and close all files and directories to reduce the chance of corruption.

In Figure 3, we can observe the internal mechanism of our checker, that how data is flowing from staring state of checking and how the checking is ending and issue resolved. In this flowchart we took "$" as an assumption that need to be updated in system. In other words, we can say that "$" is an upcoming update of the system and need to be verifying that it is free from harboring bug. There

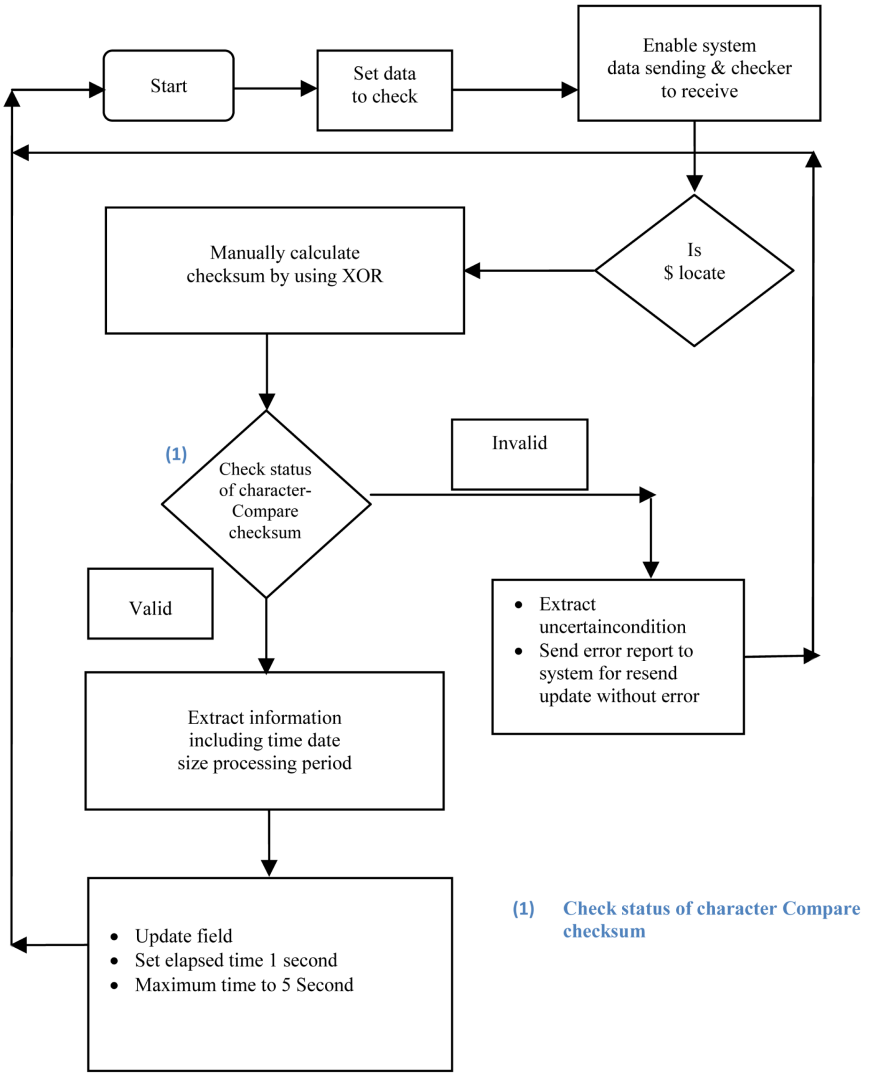

Figure 3. Flow chart of checker.

are eight phases of checker in total. In first phase the checker will collect the data which need to be scan. In second phase checker will enable system data sending and checker to receive the required data that is "$". In third phase the system will check the availability of data and make sure that the same date received that need to be update in system. In forth phase system applies the manual XOR operation of checksum to validate the checksum mismatch and validate the status of update. In fifth phase the checker validates that presence of bug if bug not found the checker will send it to next sixth of to extract information including time, data size and processing period. After that phase data will move to phase seventh that is the phase where system update finally started. In case if bug found in fifth phase it will report it to sender by making data's invalid flag from 0 to 1, this phase extract uncertain condition that leaded checksum mismatch and create the report for sending to server about system harboring bug.

Step-1: Let S [System]: = 1 [Initialize]

Step-2: Let C [Checker]: = 1 [Initialize]

Step-3: Let Field [F]: = 1 [F Initialize]

Step-4: Connect S to F

Step-5: Detect whether update required

Step-6: If update needed go to step 7 else go to Step 11

Step-7: Send data toward C for harboring bug checking

Step-8: Check data if harboring bug found send acknowledgment to S for re-sending data to S and go to step 7 else go to Step 11

Step-9: If data has no error updates F and go to Step 6

Step-10: If any other filed need to be update go to step 4 else go to Step 11

Step-11: Exit

In **Figure 4**, we can observe that Esext3 consuming more time as compared with traditional checker FSCK of Ext3 file system. It is because of file system checking through XOR operation in system of our provided solution. We simulated one thousand transactions to one hundred twenty eight thousand transactions and we found that our system is consuming more time but the time consumption is avoidable just like in case of one hundred twenty eight thousand transactions FSCK consuming approximately 5500 seconds and our proposed solution consuming approximately 6000 seconds, it is just 500 seconds difference but with reliability this difference is negligible.

Figure 5 shows the memory consumption of both FSCK and checker of ESext3 file system. Our proposed architecture utilizes less memory as compared with FSCK checker. The main reason is that FSCK is not the part of Ext3 file system so we need to invoke it every time when we need to scan our system. FSCK does not have the facility to check the system by itself when updates are received, it only works on admin request or when system executes the update after installation and found harboring bug. On the other hand, our proposed checker is the integrated part of Esext3 system, so no need to invoke it. It check all upcoming updates in file system so that harboring bug cannot take place in file system and also no need rush scan when file system needs to execute any particular area of software.

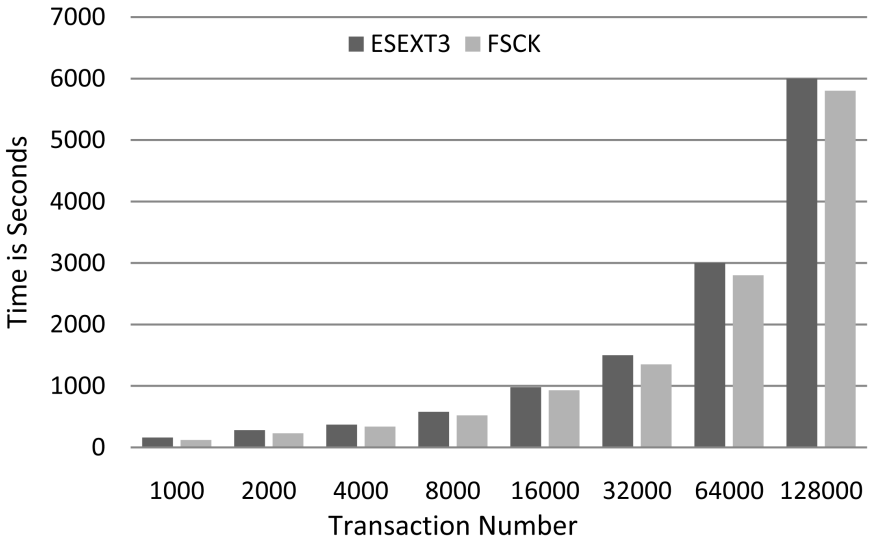

Figure 4. Average number of transaction.

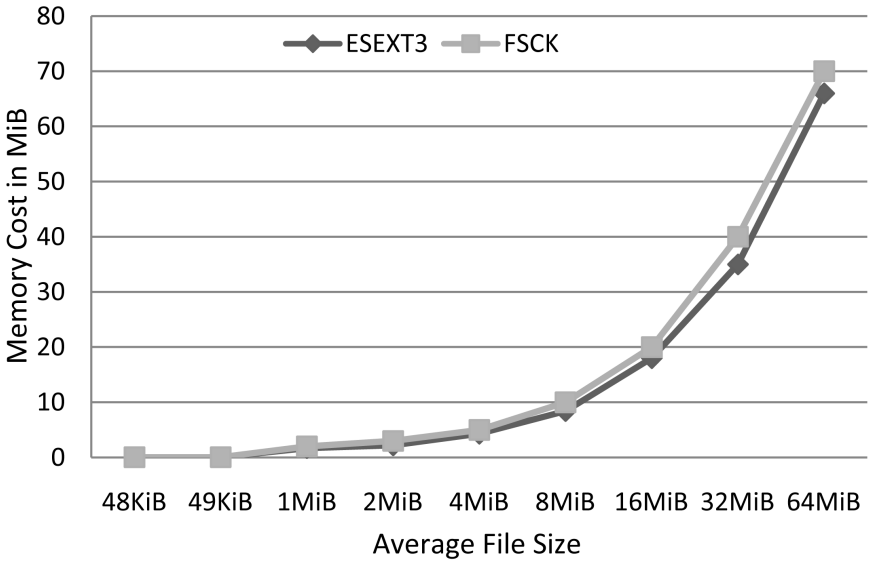

Figure 5. Average memory utilization.

5. Esext3 Performance Analysis

We conducted demonstration testing of our proposed solution on three main platforms 1) file server environment, 2) web server environment and 3) database server environment to insure its working capabilities and analyze the performance of it. More details of figures are cited blow so that we can observe all in deeply and understand the mechanism of our proposed solution. Furthermore, in conclusion we can understand the future work of our proposed Esext3 file system. In all three experiments of file server, web server and database server we used two variables first is I/O operation and other is offset operations. I/O operations are responsible for system updates getting data from server and (known as read) and update the required field if update is free from harboring bug. The offset operations are responsible for making connection between server and the

client that need the update in the system.

We performed all the experiments on 2.1GHz Intel core i5 processor with DDR3 4GiB of RAM with Samsung SATA 500GiB of hard disk drive testing system with Linux kernel simulator 2.6.2 complete parameters are shown in Table 2.

Table 3 shows the key workload between Ext3 and Esext3 file system. In total there seven type of workload in our experiments. The first File and directory count this shows that how both file system work Ext3 work on single file per scan but our proposed solution work on TDMA (Time Division Multiple Access) so many files can be scan at a time. The directory tree depth the workload shows that the Ext3 file system often goes deeply in directories and creates nested directories. On the other hand, our proposed solution has some issue that directories remain shallow because it tries to make scan uniform. Our proposed solution can work on Multi-gigabyte disk image file but Ext3 learn towards many small files that increase memory and time cost. In second workload property in Table 3, Meta-data operations we can observe that low rate of Meta-data operations as low as 40% present but in our proposed solution the Meta-data operations are nearly 72% and it is higher than normal Meta-data operations.

Table 2. Linux kernel simulator configuration parameters.

Parameter	Linux kernel simulator 2.6.2	
No. CPUs Cores	4 (2.1 GHz)	
Memory	4 GiB DDR3	
Disk Drive	500 GiB	
Disk Image Format	Think Flat VMDK	
Files System	Ext3	Esext3
I/O Scheduler	CFQ	

Table 3. Key I/O operation.

No.	Workload property	Ext3	Esext3
1	File and Directory count	Single file per scan	Many files and Directories
	Directory tree deepness	Often deeply nested directories	Low and Homogeneous
	File size	Incline towards many small files	Multi-gigabyte disk image files
2	Meta-data operations	Lowest (40%)	Many (72%)
3	I/O synchronization	Asynchronous and synchronization	Asynchronous and synchronization
4	In-file randomness	Workload-dependent	Workload-dependent
	Cross-file randomness	Workload-dependent	Workload-dependent
5	I/O size	Workload-dependent	Workload-dependent
6	Read-modify-write	Infrequent	Infrequent
7	Processing time	Workload-dependent	Increased because of manual XOR checksum

As we know that our proposed solution is based on Ext3 so in third workload in table 3, I/O synchronization the Ext3 and our proposed solution Esext3 both Synchronization and Asynchronous to manage the I/O operations. In the fourth, fifth and sixth our proposed solution inherits the property of Ext3 file system. In the seventh property of Table 3 the processing time, in Ext3 we can observe that it totally depends on workload but in our proposed solution the processing time increased due to manual checksum of XOR. In Figures 6-12 the percentage of requests explain the data flow between server and client for read and write of hand shaking. For example in Figure 6 we can observer that the percentages of request for different servers are different. Such as for file server it is approximately 45 percentages, this 45 percentage is calculated by the total mean of read

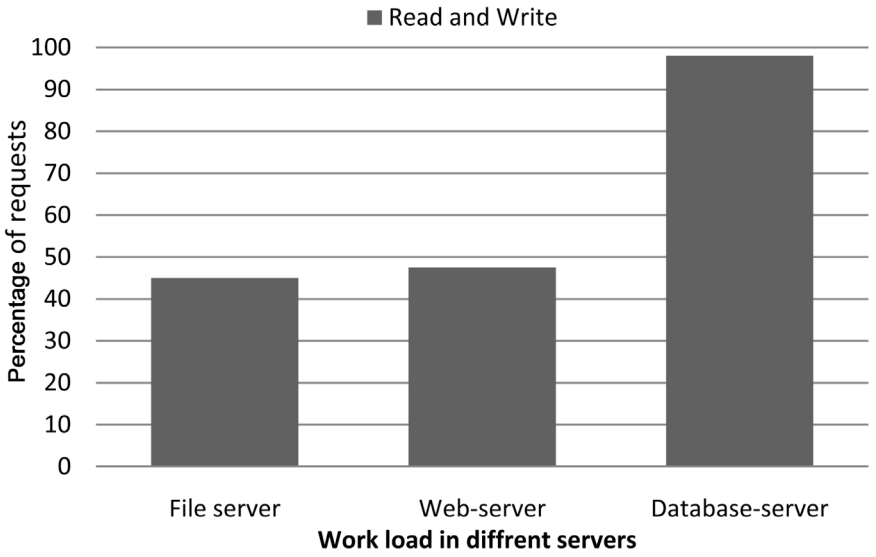

Figure 6. Workload of different environment.

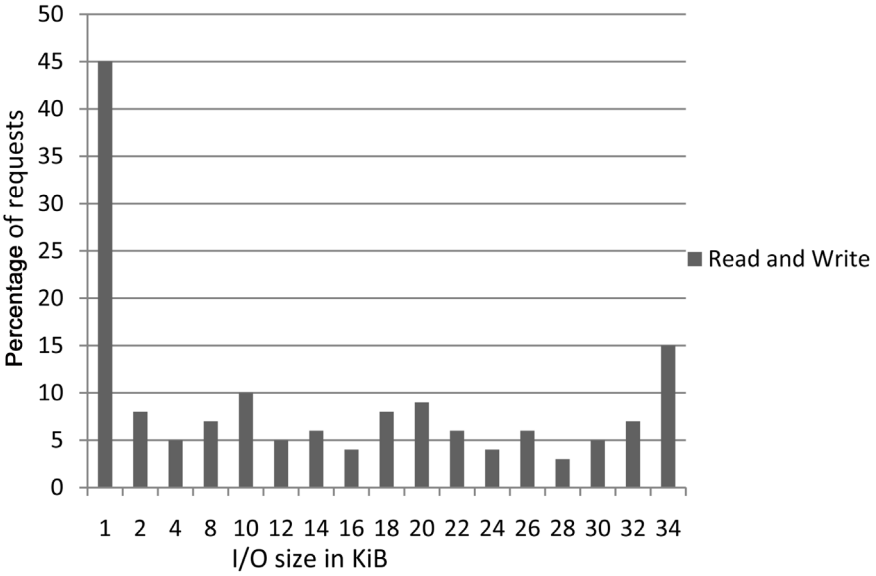

Figure 7. I/O operation in file server.

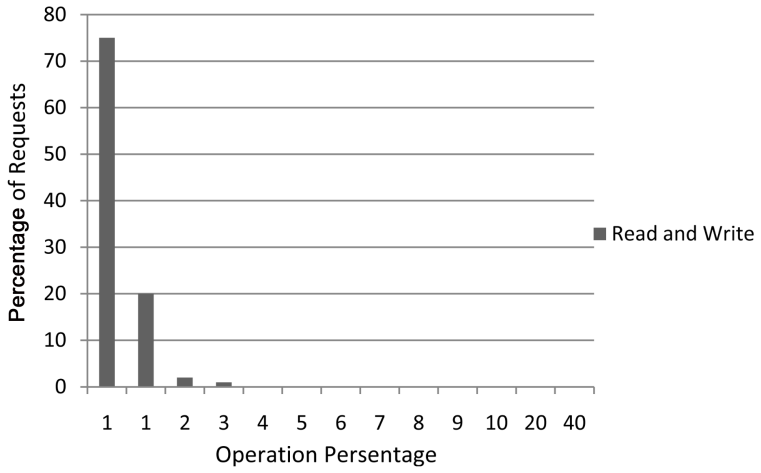

Figure 8. Offset operations in file server.

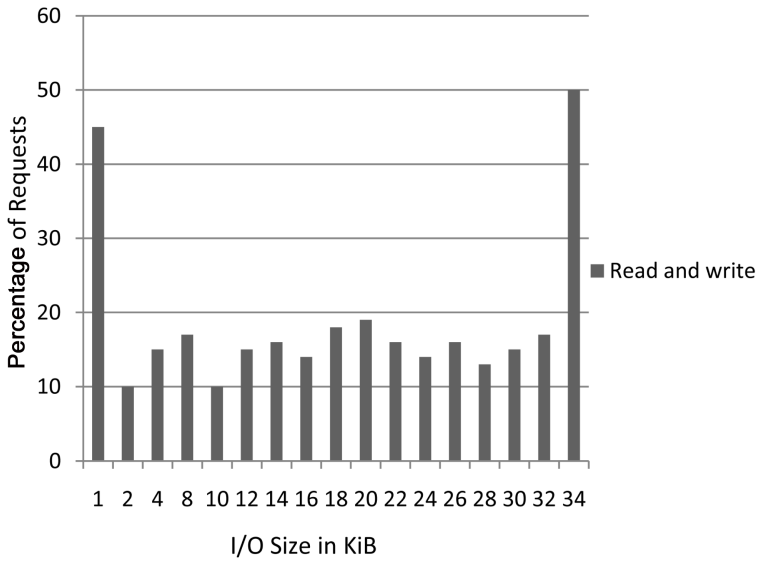

Figure 9. Operation in web server.

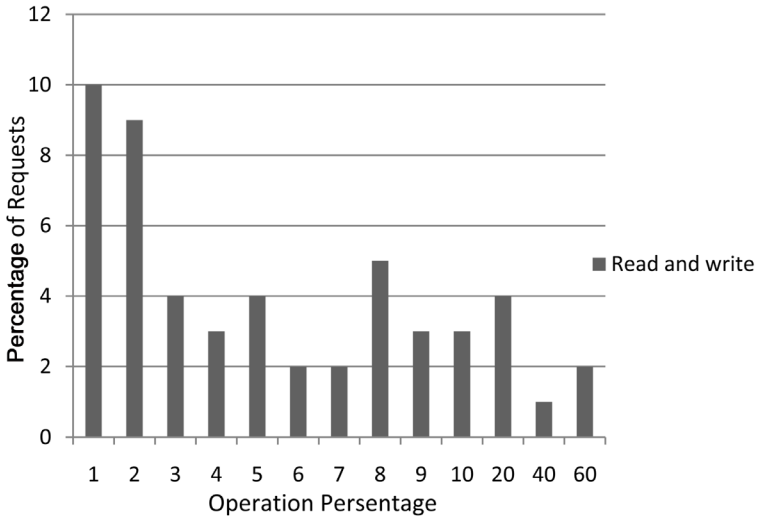

Figure 10. Offset operations in web serve.

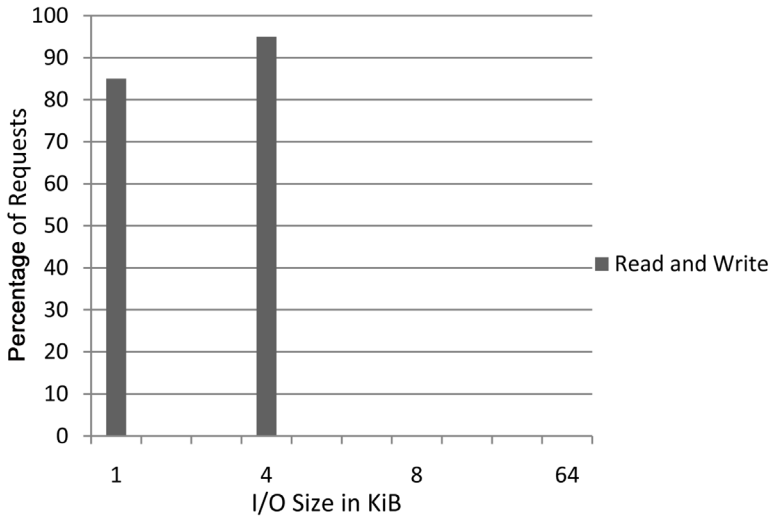

Figure 11. I/O operations in database server.

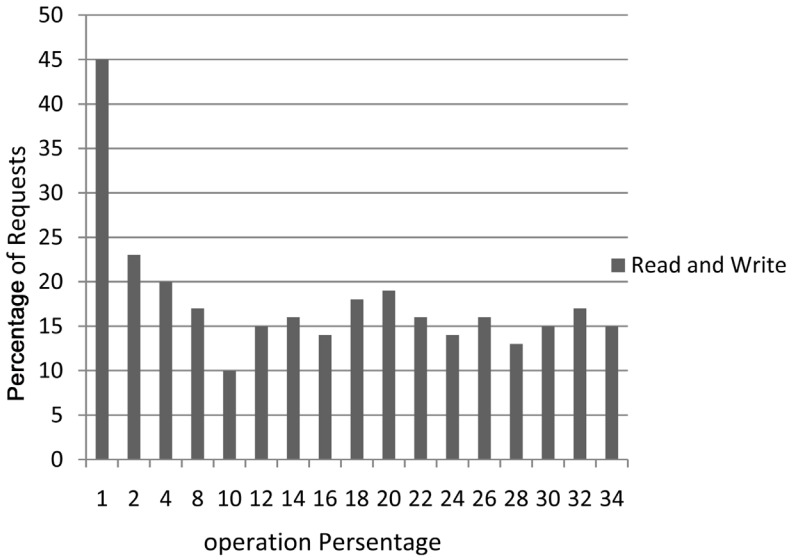

Figure 12. Offset operation in database server.

and write operation done by client for its upcoming updates. Whenever client observer that there is an update coming from server for the update of file system, the server first send hand shake request to client and that request is considered as read and when client perform some sort of operation on that hand shake and prepare the disk to manage update that is known as write. So, for ever update there is read and write operations to update system and checker perform check for harboring bug through our prescribed procedure explained in Figure 3.

In Figure 6, we can observe the workload of our proposed solution. We carried out our experiment in three main areas of possible file updates. The first is file server second is web server and third is database server. In our experiment we observe that our proposed solution can work better in file server, moderate results observe in web server and unavoidable delay and offset requests are encountered when updates received from database server.

In Figure 7, (Figure 7 and Figure 8 explaining the results of file server) we can observe that I/O operations are normal most of the read and write in the system are normal that shows the system is working properly. In Figure 8, we can observe that offset operations are nearly zero except the initial connection built because on first connection there are some hand shaking is the system and these hand shaking working on TCP/IP and FTP protocols.

In Figure 9, (Figure 9 and Figure 10 explaining the results of web server) we can observe that read and write operation numbers are higher as we earlier mentioned that the functionality of our proposed solution in web server is moderate not good nor bad. Furthermore, in Figure 10, we observe that there are too many offset operations for hand shaking with server to get updates, but all operations are successful. In the web server architecture our proposed model showed some delay and hand sharing issues and reconnection because of HTTP protocol.

In Figure 11, (Figure 11 and Figure 12 explaining the results of database server) we can observe that the system is not working well unavoidable delay in I/O operating of read and write. It also poses unavoidable data flow in offset operation that putted system in halt position as shown in Figure 12.

Memory utilization is the key feature of any architecture that can make system more robust and reliable. Herein the memory does not mean the hard disk drive; it refers to the RAM the primary memory. And secondly we ignored the usage of virtual memory in our system because it has less speed and more time consuming memory. If we include it we should talk about paging so that it is purely different prospective of Linux file system. So that we are not involved in paging but most probably we will work on it near future.

In Figure 13, (Figure 13 shows results of memory utilization of I/O operations and Figure 14 shows the offset operation memory utilization in files server,

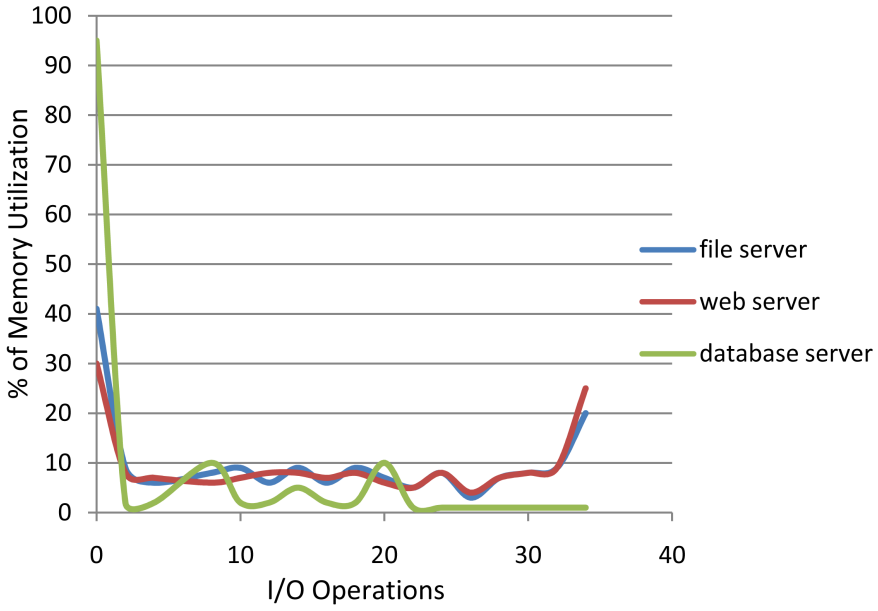

Figure 13. Memory utilization in I/O operations in all environments.

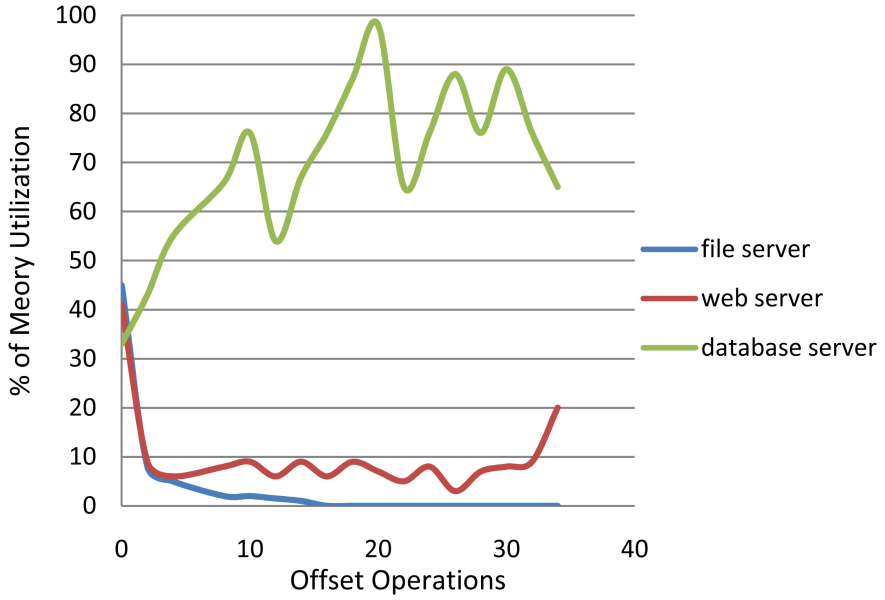

Figure 14. Memory utilization in offset operations in all environments.

web server and database server environments.) we can observer that system of our proposed architecture Esext3 file system uses less memory while updating in the environments of file server and web server, but in the case of database server not utilizing much memory due to repetitively disconnection form updating server. In Figure 14, shows the offset operation memory utilization, we can observe that file server has very low utilization of memory because there is no disconnection, but little high utilization of memory in web server because it faced some disconnection from server so it needed more offset operations to reconnect from updating server. On the other hand, database server used unavoidable memory for offset operations to reconnect from updating server. And it can be the potential drawback of our proposed architecture.

6. Limitations

Our proposed architecture has some limitations as it come with new field of checker. This works on XOR operation to check checksum mismatch in file system so it consume more time as compared with traditional checker FSCK. It is using the traditional backup and rollback feature of ext3 so these features also responsible for making delay in updating the field in file system but this feature reduces the memory consumption as shown in Figure 5. Moreover, system update using network is most important feature of the system. In network there are many protocols used to transfer the file or system update. Some uses FTP over TCP and some system uses TFTP over UDP it totally depending upon the system requirements. As we know that our system using the Acknowledgment for error reporting to the server that needs to update the Esext3 files system. In our system's simulation we observe that system updated normally when using TCP or FTP but when we used TFTP over UDP the system's update showed unavoidable latency because TFTP over UDP has minimum features and doesn't

have authentication. In the condition of UDP protocol we used offline system update. First we downloaded update packages and update the field in Esext3 files system. In this condition of offline update, if those packages contain the harboring bug then we have to report it manually to the update server and ask to resend the updates manually or upload the updates on their server manually. Furthermore, bottleneck and network latency also creates delay in system update.

7. Conclusion and Future Work

This architecture is good and suitable for the system which has zero tolerance for harboring bug. This Esext3 introduced new system that came with new filed known as checker having the responsibility to check system's upcoming updates in any field of Esext3 files system. Our results showed that the system is not suitable for the machine or environment where time is more important as compared with reliability such as database environment. As we all know that "***Security never comes for free***". It is the tradeoff between security and different type of costs just like time, memory, resources etc. This system has better performance where updates are offline as compared with online updates. Delay in updating filed can be potential drawback of our architecture while using network in the database server environment and it is an open issue. We are working on it and soon introduce new phenomena to overcome these delay in system.

References

[1] McKusick, M.K., *et al.* (1984) A Fast File System for UNIX. *ACM Transactions on Computer Systems*, **2**, 181-197. https://doi.org/10.1145/989.990

[2] Stein, C.A., Howard, J.H. and Seltzer, M.I. (2001) Unifying File System Protection. *USENIX Annual Technical Conference*, *General Track*, Boston, 25-30 June 2001, 79-90.

[3] Rumble, S.M., Kejriwal, A. and Ousterhout, J.K. (2014) Log-Structured Memory for DRAM-Based Storage. *FAST*, Vol. 14, Santa, 17-20 February 2014, 1-16.

[4] Bartlett, W. and Spainhower, L. (2004) Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, **1**, 87-96. https://doi.org/10.1109/TDSC.2004.4

[5] Zhou, K., Liu, Y., Song, J., Yan, L., Zou, F. and Shen, F. (2015) Deep Self-Taught Hashing for Image Retrieval. 23*th ACM International Conference on Multimedia*, Brisbane, 26-30 October 2015, 1215-1218. https://doi.org/10.1145/2733373.2806320

[6] Huang, Z., Jiang, H., Zhou, K., Wang, C. and Zhao, Y. (2016) XI-Code: A Family of Practical Lowest Density MDS Array Codes of Distance 4. *IEEE Transactions on Communications*, **64**, 2707-2718. https://doi.org/10.1109/TCOMM.2016.2568205

[7] Kim, H., Seshadri, S., Dickey, C.L. and Chiu, L. (2014) Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches. *ACM Transactions on Storage*, **10**, 15. https://doi.org/10.1145/2668128

[8] Deng, Y. (2011) What Is the Future of Disk Drives, Death or Rebirth? *ACM Computing Surveys*, **43**, Article 23. https://doi.org/10.1145/1922649.1922660

[9] Hagmann, R. (1987) Reimplementing the Cedar File System Using Logging and Group Commit. *ACM SIGOPS Operating Systems Review*, **21**, 155-162. https://doi.org/10.1145/41457.37518

[10] Thomson, A. and Abadi, D.J. (2015) Calvin FS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. *FAST*, Vol. 14, Santa, 17-20 February 2014, 1-14.

[11] Hong, L., Yan, Y., Ouyang, M., Tian, H. and He, X. (2017) Vulnerability Effects of Passengers' Intermodal Transfer Distance Preference and Subway Expansion on Complementary Urban Public Transportation Systems. *Reliability Engineering and System Safety*, **158**, 58-72.

[12] Yang, J., Sar, C. and Engler, D. (2006) Explode: A Lightweight, General System for Finding Serious Storage System Errors. *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, 6-8 November 2006, 10.

[13] Yang, J., *et al.* (2006) Using Model Checking to Find Serious File System Errors. *ACM Transactions on Computer Systems*, **24**, 393-423. https://doi.org/10.1145/1189256.1189259

[14] Malka, M., Amit, N. and Tsafrir, D. (2015) Efficient Intra-Operating System Protection against Harmful DMAs. *FAST*, Vol. 14, Santa, 17-20 February 2014, 29-44.

[15] Zhang, S., Catanese, H. and Wang, A.I.A. (2016) The Composite-File File System: Decoupling the One-to-One Mapping of Files and Metadata for Better Performance. *FAST*, Vol. 14, Santa, 17-20 February 2014, 15-22.

[16] Andersen, M.P. and Culler, D.E. (2016) BTrDB: Optimizing Storage System Design for Time Series Processing. *FAST*, Vol. 14, Santa, 17-20 February 2014, 39-52.

[17] Zhao, Y., Jiang, H., Zhou, K., Huang, Z. and Huang, P. (2016) DREAM-(L) G: A Distributed Grouping-Based Algorithm for Resource Assignment for Bandwidth-Intensive Applications in the Cloud. *IEEE Transactions on Parallel and Distributed Systems*, **27**, 3469-3484. https://doi.org/10.1109/TPDS.2016.2537334