DATA LOSS MINIMIZATION FOR FILE SYSTEM CRASH RECOVERY USING PERSISTENT MAIN MEMORY

Author

Cheng-Yu Hung

Supervisor

Li-Pin Chang

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
AT
NATIONAL CHIAO TUNG UNIVERSITY
HSINCHU, TAIWAN
JULY 2013

Table of Contents

List of Tables

List of Figures

Abstract

System failure always happens due to power outage, there would lost many data if power outage occurs. Even there have some methods like UPS to prevent data loss, but it costs too high. We proposed a technique based on battery-backed memory to protect file systems from losing data across power failures and guaranteed file system integrity. The idea is simple: when power restores after a power outage, our method salvages modified but unwritten data by scanning and parsing the contents in memory. We have implement our method on Ext3 file system and improve recovery ability compared to the ordinary file system recovery. Besides, our approach is also beneficial to integrity-aware applications like databases.

Acknowledgements

Here, I have many thanks. First, sincerely thanks for my advisor Li-Ping Chang, without you, I can't finish my research work. Although I always confused on what I do, but your patient make me learned efficiently. In these two years, I have learned a lot from my advisor, not only how to do the research, but learned how to deal with people. I'm grateful to have you enlighten me, words are really not enough to thank you.

During these two years, I have to thank for all members in our lab. First thanks for Po-Han Sung, Tung-Yang Chou, Wen-Ping Li, Yi-Kang Chang and Chen-Yi Wen, thank you for giving advices to me when I got troubles in research. And I have to thank to Ting-Chieh Haung, Chao-Yuan Mao, and Sheng-Min Huang, thank you for making my master life so brilliant and enrich my life, I have the best wishes for you in the future. Except for members mentioned above, thanks for all the junior, thank you Chun-Yu Lin, Pin-Xue Lai, Guan-Chun Chen, Min-Chi Yan, Po-Hong Chen and Yu-Syun Liou for bringing joyous atmosphere to our lab.

Except for members in lab, I have to thank for my girlfriend, Li-Ling Chen, thank you for supporting me when I'm down during research or my life, and always told me "Anywhere seems nowhere, yet somewhere proves to be everywhere."

Last but not least, I have the most special thanks for my parents. I deeply thanks for my parents to raise me up. It is you make everything possible. Without you, I can't finish this work today. Thank you and I love you.

Chapter 1

Introduction

When the traditional file system encounter the accidental power failure or system crash, there would lost many data. Since metadata is the most important part of the file system, if the lost data is metadata, it may cause file system damage. On the other hand, if the lost data is user data, even user data wouldn't hurt the file system, but it closely related to the data that we use. So we don't want to miss either metadata or user data at all.

System failure always happens due to power outage, operating system crashes, or hardware downs. Even that there have some methods, for example, fsck, the file system consistency check. File system consistency check is performed after crash and before file system is re-mounted. However, fsck has a big problem: they are too slow. With a very large disk volume, scanning the entire disk to find all the allocated blocks and read the entire directory tree may take many minutes or even hours [1], which is unacceptable for servers demanding high availability. Thus, in order to speed up the recovery of the file system, the journaling file system [2] was proposed. Journaling is a technique that guarantees file-operation atomicity for structure consistency of file systems. However, as file systems use page cache for efficient data access, data in main memory are lost after a power cut.

The method mentioned above is about recovery. If we want to protect the data against power failure before recovery, it will be more complicated. First, in nowadays, the main memory used in computer is always volatile, when the power failure, the data in the memory would all vanish. Second, in order to maintain the consistency of the file system, journaling file system would discard a lot of data even they are already on the disk to guarantee file-system consistency. To solve the problems of protecting data, the UPS (Un-interruptible Power Supply) was used, however, it costs too much. The UPS could cost almost nearing NT$30,000 under high security circumstance to keep data against power failure. Besides, UPS has some potential problems, suck like it is infeasible for embedded devices, and it is possible power failure on broken UPS. On the contrary, the NV (non-volatile) Memory or the battery backup memory would be more economical. The battery we used in the memory costs only NT$100∼NT$200, it costs almost one tenth of the UPS. File-system designs using battery-backed memory has been proven feasible in prior work [3]. However, the prior work aims at efficient file system metadata access rather than data protection and recovery.

This work introduces a technique based on battery-backed memory to protect file systems from losing data across power failures. The idea is simple: when power restores after a power outage, our method salvages modified but unwritten data by scanning and parsing the contents in memory. Basically our method guarantees that user data are persistent after a system call to the file system returns. This idea looks simple, but there exist some major technical difficulty need to overcome. First, we need to analysis the data structure of the file system when the power restoration. Second, there must have some incomplete file operation during power failure, if we write back all the data in the memory into the disk without any consideration, there might cause file system inconsistency. Hence, in order to maintain file system consistency, we have to minimize the data that need to discard. Third, we cannot increase the overhead of the data that are not related to what we want to recover. Our proposed method records the related metadata in the same file-system atomic operation, once the power failure occur, we can parse memory contents and retrieve modified data of completed file operations, while discard the data of incomplete file operations.

Our proposed method was implement on ext3 file system successfully. The experimental results show that, without incurring any file-system inconsistency, our method fully recovered all the modified but unwritten data in main memory, while the original ext3 with journaling lost 20% of the data. The reason is that once the file system's system calls returned, it means that all of the related file operations finished, hence it wouldn't lose data when the power failure occurs. Besides, we implemented our method to measure the performance on the databases. Databases frequently perform fsync() to ensure that modified data reach stable storage, since our proposed method using battery backup memory, it doesn't need to perform fsync() once the file operation completed. The experiment result shows that implement our method on MySQL+sysbench would have three times faster than original scheme.

The rest of this paper is organized as follows. In chapter 2, we describe the background of this work. Then, we elaborate the system overview in chapter 3. In chapter 4, we explain how we do the recovery scheme. Performance evaluation results are discussed in chapter5. Finally, we summarize the conclusion in Section 6.

Chapter 2

Related Work

When the file system suddenly crash, there are many methods to fix it. For example, the file system check, which is usually used to check the state of the file system and do recovery before the file system mounted. However, fsck(and similar approaches) have a bigger and perhaps more fundamental problem: they are too slow. With a very large disk volume, scanning the entire disk to find all the allocated blocks and read the entire directory tree may take many minutes or even hours, and cause the problem of scalability [1]. Thus, in order to speed up the recovery of the file system, the journaling file system [2] was proposed. However, journaling file system was proposed to maintain file system integrity, not aim to protect data from suddenly system crash, hence when the system suddenly crash, there would lose a lot of data as usual.

When talk to non-volatile RAM (NVRAM), NVRAM can keep data when power failure occur, just like what the disk does, but has much better performance. Hence, NVRAM can make a system more reliable and get higher performance. [4] proposed using NVRAM as a cache on client site to reduce write traffic to servers in distributed file system. However, it still cannot protect the data in main memory, once the power failure, the data in main memory would all gone. The conquest file system [3] was a file system composed of disk/persistent-RAM, it reduced disk access time and improve performance by storing small files, metadata, and shared libraries in persistent RAM; disks hold only the data content of remaining large files. It looks like make the best use of NVRAM, but from the data safety side, it didn't protect data in main memory at all. Once the power failure, the data in main memory would all vanish as well. Another study about NVRAM was main memory databases. [5] used NVRAM as a logging space and archive database. Once the system crash, it merge data on the archive database and the log to obtain data needed to recover to the most recent consistent state. However, it just cannot protect the data in main memory as well.

Another study about NVRAM, such like [6], they proposed a technique to remove lots of redundant writes from the memory and hence extending the lifetime of the PCM. They also implemented PCM to achieve the best tradeoff between energy and performance. [7] used new byte addressable memory to design a file system that has five times faster than traditional file system. [8] handled crash recovery by using asynchronous mirroring, which keep backup data on-line, and periodically transfers snapshots of the data from source volume to destination volume. However, these usages are far from what we proposed.

Chapter 3

Design Concept

3.1 Overview

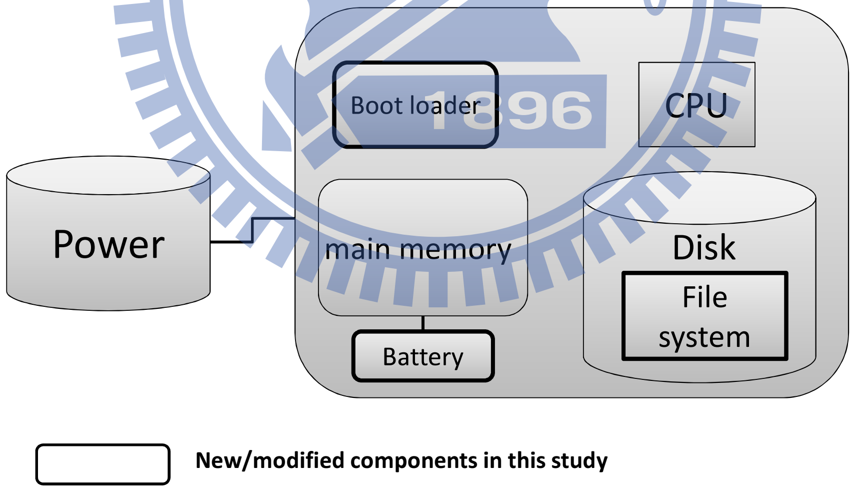


Figure 3.1: System Overview

A system we used included file system with Ext3, reboot, a boot loader of the system, a battery-backed memory, consists of 1GB SDRAM (contain 256MB disabled RAID cache) and two 18650 lithiumion battery, each has 2200mAh and the voltage 3.6V, which can keep the contents of the main memory for up to 72 hours.

A unit that contains data we want to write called a block buffer, and file system adds special signatures to essential data structures related to block buffer of the underlying storage devices.

When power failure occurs, the battery supports the main memory until power restores. Upon power restoration, if the file system is not cleanly un-mounted, then boot loader intervene the bootstrap process for data recovery. To keep the data in the main-memory intact, the data recovery procedure must precede the loading of the operating system. The boot loader scans the memory for block buffers with signatures, and writes them back to the storage device.

Because power failures occur in the middle of some file system operations, writing all the block buffers salvaged from main memory to the storage device will damage the file-system integrity. Our recovery process must judiciously discard a small amount of data to prevent the recovery process from corrupting the file system in the disk.

3.2 Checkpoint-Based Data Recovery

The data copied from main memory to disk during recovery must not contain any modified block buffers belong to incomplete file system operation. We first proposed a checkpoint-based method for data recovery. The file system commits a checkpoint in a fixed frequency. A checkpoint is a snapshot of cached pages; in a snapshot there is no file-system-level inconsistency. To commit a checkpoint, the file system locks all the block buffers and waits until all ongoing file system operations complete. Subsequent writes to the locked buffers are handled using copy-on-write: when a file system operation modified a locked buffer, the file system creates a shadow buffer of the locked buffer and write new data in the shadow buffer. Figure 3.2 shows the timeline of checkpoint-based recovery.

After checkpointing in the memory, these data would flush to disk then. When all the checkpointed data flush to the stable storage, we call these kind of data are durable. Instead, if the data just finish checkpoint in the memory and power failure occurs, we call these data are not durable. This kind of data can be secured when doing recovery. Except for these two kinds of data, all the data of ongoing operations would be discard during

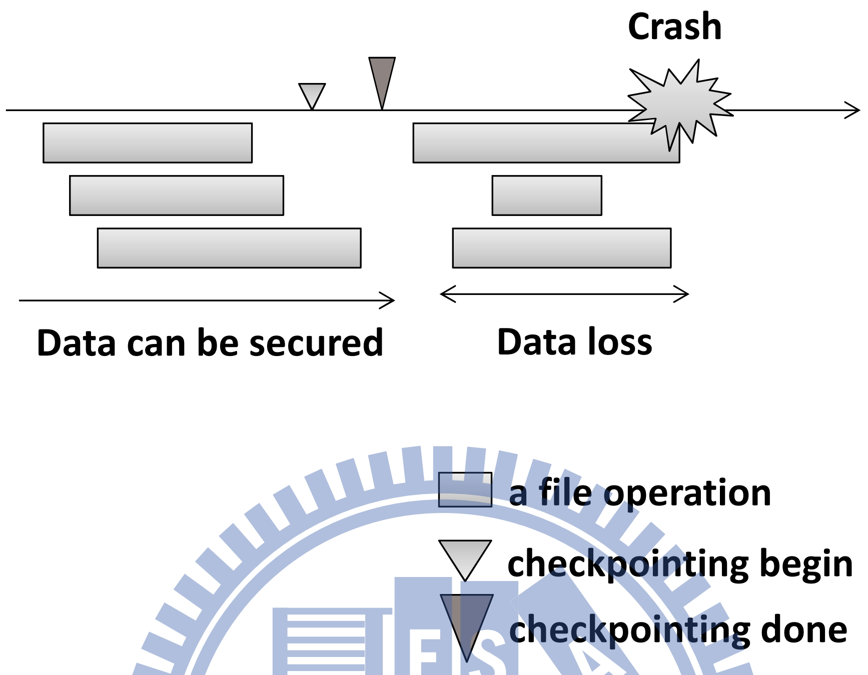


Figure 3.2: Timeline of Checkpoint-Based Data Recovery

recovery process, as figure 3.3 shows.

The recovery procedure writes back only the block buffers of a committed checkpoint. The block buffers that do not belong to a committed checkpoint are discarded during recovery.

The proposed recovery method requires to identify the block buffers of committed checkpoints. We propose adding a signature and a checkpoint ID to a block buffer when the file system allocates the block buffer. The signature is for the boot loader to identify block buffers during crash recovery. Figure 3.4 shows the process of adding signature to a block buffer. The purpose of the checkpoint ID is to identify whether a block buffer belongs to a committed checkpoint. The file systems must record in main memory the IDs of committed checkpoints.

Notice that our method do not put all the block buffers of a committed checkpoint in a linked list and have the boot loader traverse the link list for data recovery. Traversing in the linked list requires to de-reference pointers, whose memory addresses are logical not physical. Without the original page table the boot loader can hardly translating the logical

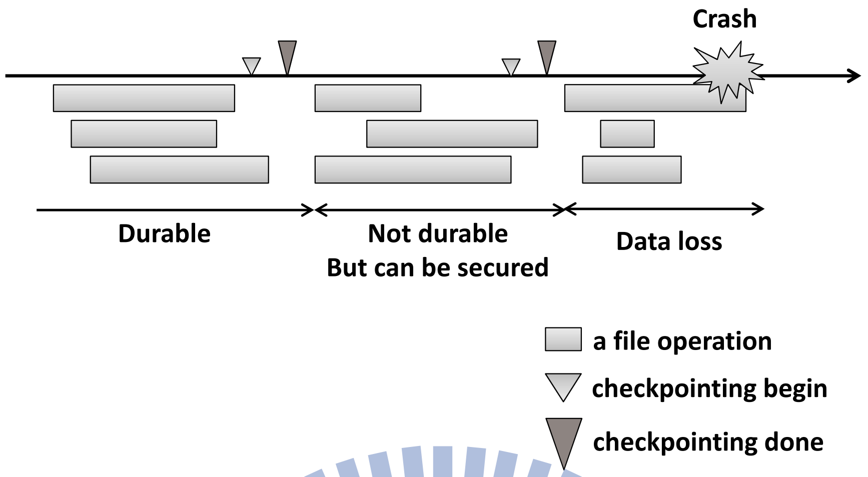


Figure 3.3: Timeline of three different kinds of data

memory addresses into physical memory addresses.

3.3 File-Operation-Based Data Recovery

A major concern of the checkpoint-based method is that the amount of data loss is proportional to the checkpointing frequency. The longer the checkpointing period, the more data will be discarded during the recovery procedure. However, decreasing the checkpointing frequency would increase the pressure of memory as the page cache writes back the data frequently.

Recovering all dirty block buffers from the main-memory could possibly corrupt the file system because some of the block buffers are modified by the file operations that are interrupted by power failures. Thus, to guarantee file-system integrity during data recovery, the recovered block buffer must not be related to any interrupted file system operations. Now let an atomic set of a file operation be a collection of all the block buffers modified by the same file operation. As figure 3.5 shows, an atomic set is marked incomplete if any of its block buffers is modified by an ongoing file operation. It is marked complete otherwise. A simplistic approach is that, the file system creates an atomic set for a file operation and adds the modified block buffers to the atomic set during the operation. The recovery procedure copies back only the atomic set that are not related to an interrupted

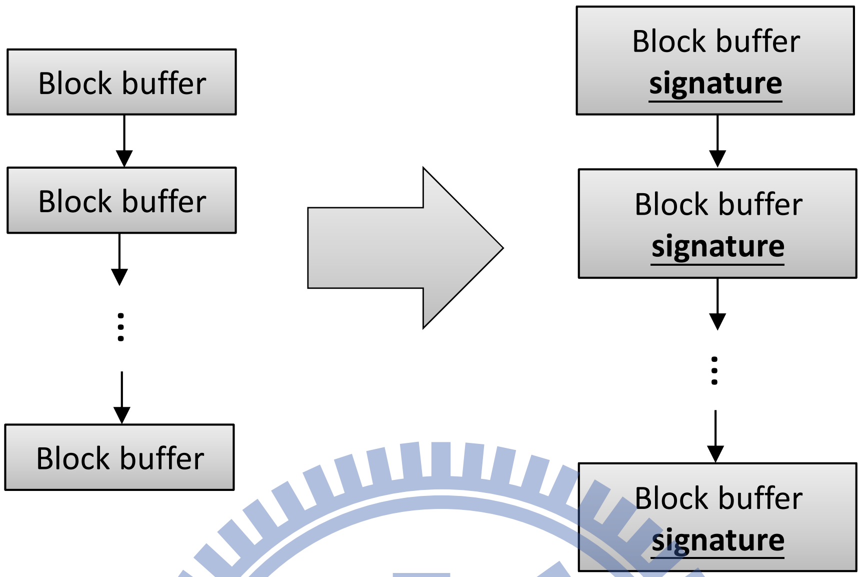


Figure 3.4: Adding signature to block buffer for memory scanning use

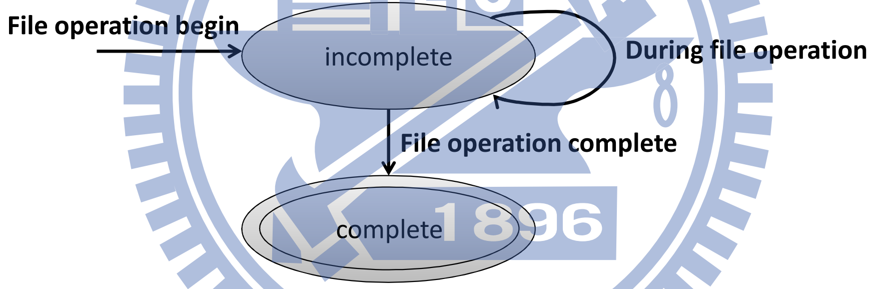


Figure 3.5: The status of an atomic set

file operation. Figure 3.6 shows the concept of File-Operation-Based Data Recovery, the third kind of data can be secured only if it belongs to a complete file operation (atomic set).

However, the simplistic approach has a problem because operations on different files can modify the same block buffer of metadata. In most of the cases for ext3 file system, the overlap is modification on the block-allocation bitmap. For example, figure 3.7 shows that the atomic set 1 is a complete set, and another modification on the block buffer, which marked as dark-grey was completed modified by atomic set 1 already, is performed on atomic set 2. In this case, both atomic set 1 and atomic set 2 must be discard. That is, if the atomic set of an interrupted file operation overlaps the atomic set of a completed file operation, then both the two atomic sets must be discarded during data recovery.

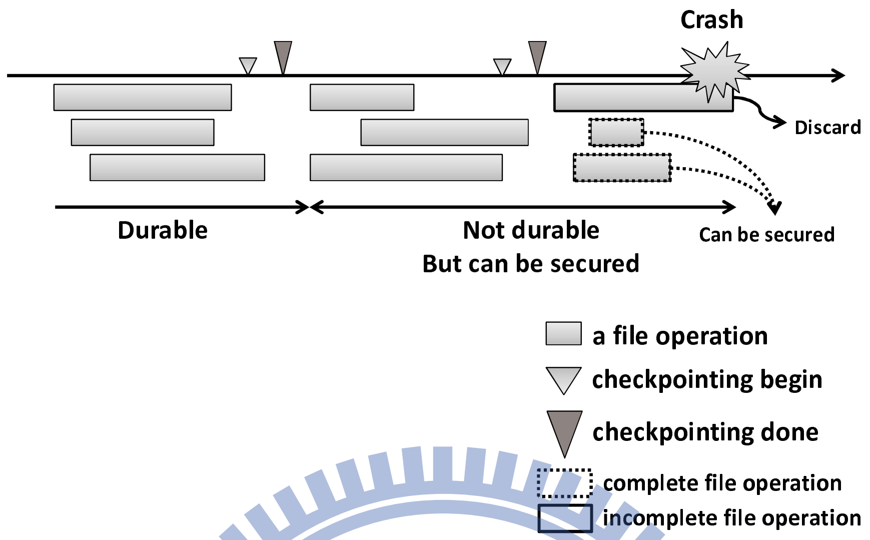


Figure 3.6: Timeline of File-Operation-Based Data Recovery

To deal with the problem, we propose merging two atomic sets if they overlap each other. Figure 3.7 shows the merge operation. Thus, when a complete atomic set is merged with an incomplete atomic set, the new set is incomplete. This design prevents the recovery procedure from copying partially modified structural information of the file system to the storage device.

3.4 Implementation for Ext3 File System

3.4.1 The Recovery Procedure in the Boot Loader

We implemented our proposed method on Ext3 file system. Our implementation involves both the file system and the boot loader. When the power failure occurred during doing some file operations, there would exist some data in completed atomic set and some incompleted atomic set in memory. To keep the data in the main-memory intact, the data recovery procedure must precede the loading of the operating system. The boot loader first scans the memory, deciding whether or not write back the salvaged block buffer according to the recovery policy (i.e., CBR or FBR). Finally, hand over the boot process to the original boot loader. Note that boot loader here would load into the special location of the

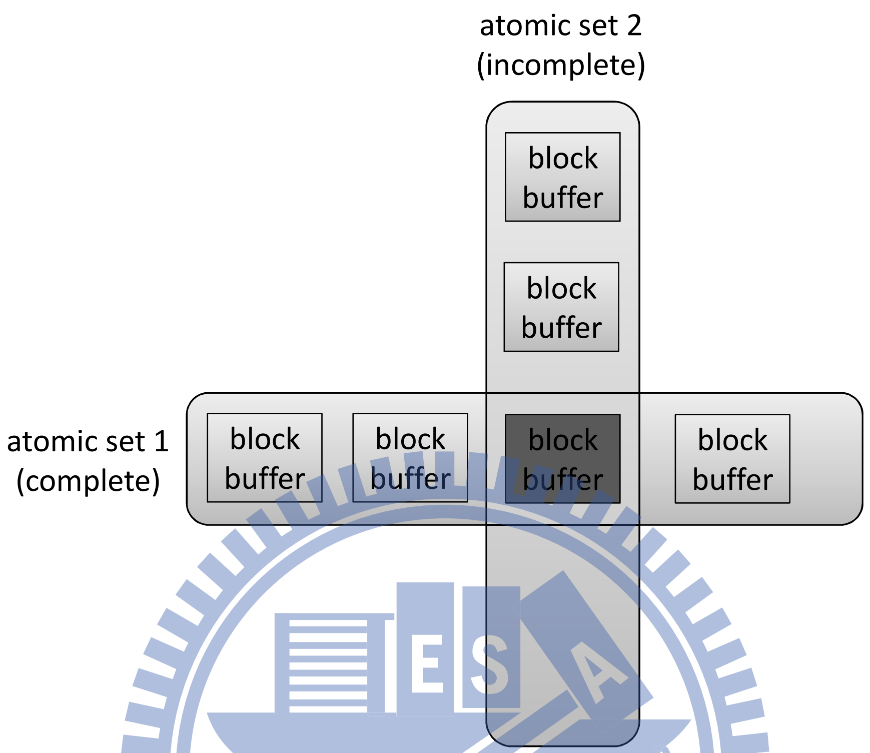


Figure 3.7: The overlaps on the complete atomic set and incomplete atomic set

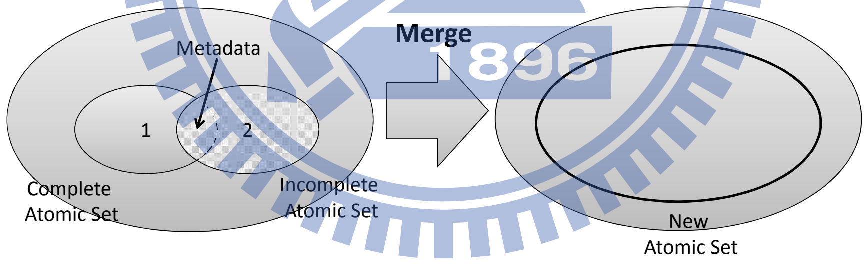


Figure 3.8: Merge of two atomic set

memory at the beginning that do not interfere recovery process.

3.4.2 Checkpointing with Ext3 Transactions

In the journaling file system, the read/write requests is cut into many data sets called transaction in the order of time, the transaction here can be mapped to the checkpoint mentioned before, and write these data sets to journal before writing them to the disk, called commit. The corresponding written data stores in buffer head, which is manage by

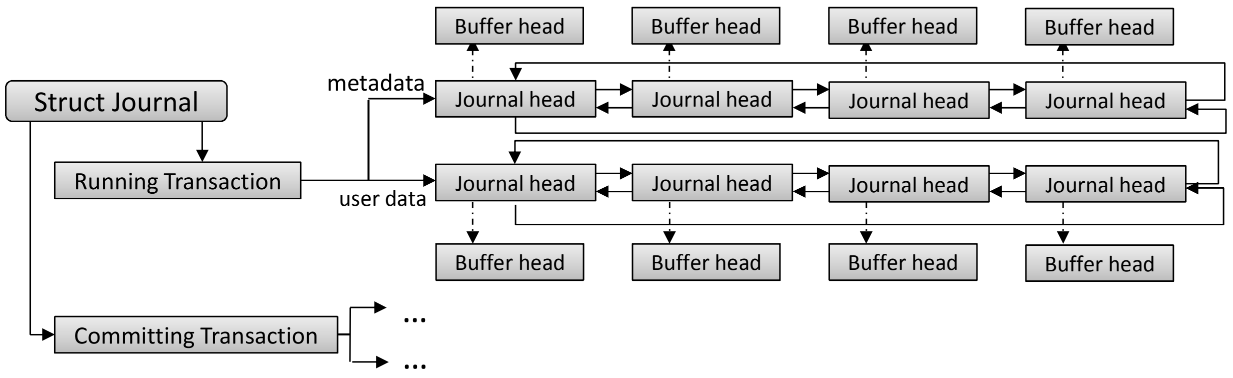


Figure 3.9: The relation between journal, transaction, and journal head

journal head individually, and add into current active transaction, i.e., running transaction. Note that the journal head is the block buffer mentioned before. In each journal head, there records the corresponding transaction ID called tid, meaning that which transaction this journal head belongs to.

A running transaction transits into a committing transaction every 5 seconds, i.e., the default commit interval. After that, journaling file system writes the data in committing transaction to disk journal, this step can be mapped to durable data in Figure 3.3. In the Ext3 ordered mode, user data must write back to disk first, then flush metadata to journal, finally write back metadata to disk. Since it only flush data in committing transaction to disk and disk journal, their might lose data in running transaction, which might lose important data we need. However, there exists a critical part of committing transaction, called commit record. Each transaction has its own commit record, and each commit record is written to journal when data in this transaction are all flush to disk journal to ensure the data can be safely retrieved from disk journal after untimely crash. Once there is no time to write a commit record, the system would think that the data in this transaction is incomplete and do not retrieve them during recovery. However, in our proposed method, we can recover the data in committing transaction, whether it is written commit record or not, since all the data are in memory, we don't need to wait for data flushing to disk journal.

We put a signature on journal head upon journal head allocation during online operations. After a power failure, when power restores, the recovery procedure (in the boot loader) detects that the file system is dirty and begins scanning the memory. The recovery procedure is of two passes. In the first pass, it scans for the signature on transaction to identify the tid of the running transaction and the committing transaction. In the second pass, the procedure scans for journal heads by another signature. The second pass copies all journal head that are with the committing transaction, while discard the journal head that are with the running transaction.

3.4.3 Managing Atomic Sets using Ext3 Handles

Handle is an important structure in Ext3 file system. Each system call modifying the file system gives rise to a single atomic operation handle. To prevent data corruption, it must ensure each system call is handled in an atomic way, i.e., either all or none. Ext3 allocates a handle before a file operation, and de-allocates the handle upon the completion of the file operation.

The manipulation of atomic sets is very like that of handles. An atomic set has a a reference count, an id number, a link list of journal heads, and also has a signature. Reference count here represents how many ongoing file operations are in this atomic set. In our method (i.e., Checkpoint-Based Recovery), the journal head includes an atomic-set id, called jid, to indicate which atomic set it belongs to. When the file system starts a file operation, it allocates an atomic set and increments its reference count. During the operation, the file system adds the modified journal heads to the link list of the atomic set. Upon the completion of the operation, the file system decrements the reference count. Figure 3.10 shows the concept of reference count on atomic sets.

As mentioned previously, overlapping atomic sets are merged together. Before adding a journal head to the current atomic set, the file system checks whether the journal head already belongs to an existing atomic set. If not, then the journal head is added to the current atomic set. Otherwise, the current atomic set is merged into the existing atomic set, and the reference count of the existing atomic set is increased by 1. Figure 3.11 depicts the relationship between journal heads and atomic sets. We record jid on journal head with the number equal to the atomic set it belongs to.

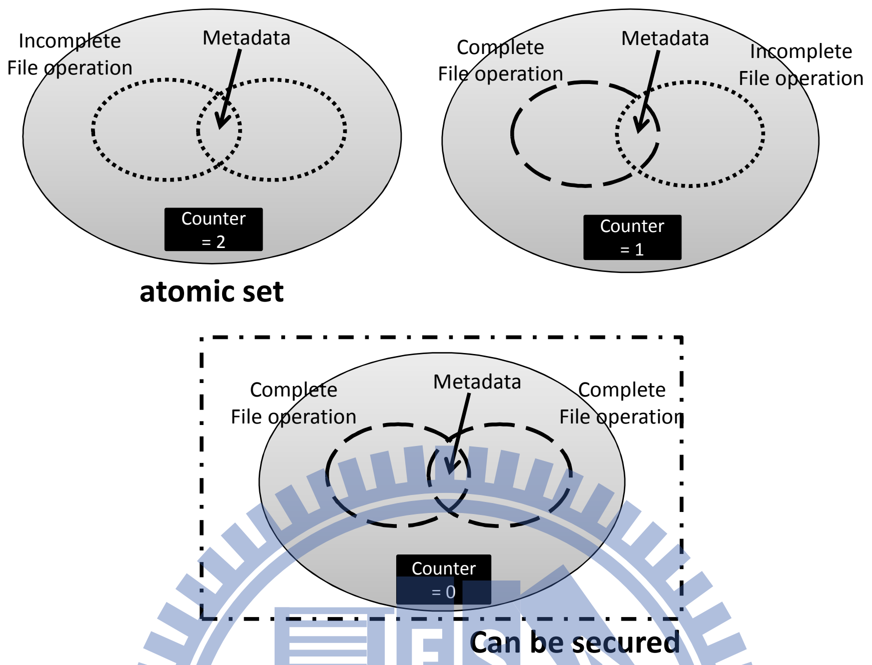


Figure 3.10: The concept of reference count on atomic sets

Like the journal heads, the atomic sets also have a special signature. The recovery procedure is of two passes. In the first pass, the boot loader scans the memory for atomic sets with reference counts of zero, and record it ID in a hash table. This is because if an atomic set whose reference count is not zero then the atomic set is undergoing a file operation that has been interrupted by a power failure. In the second pass, the boot loader scans the journal heads and compare jid with the atomic set ID in hash table, copying only the journal heads whose associated atomic sets have a zero reference count to disk journal. Then mounting file system and file system takes over the remaining recovery procedure, that is, write the data from disk journal to file system.

Note that in Ext3, the physical address of the file data is a pointer pointed by a buffer head, since we don't use de-referencing pointers during recovery, we used a variable to record its physical address. During recovery, we can only use signature scanning for buffer head and use offset to find the variable that store data's physical address to find data and write back them.

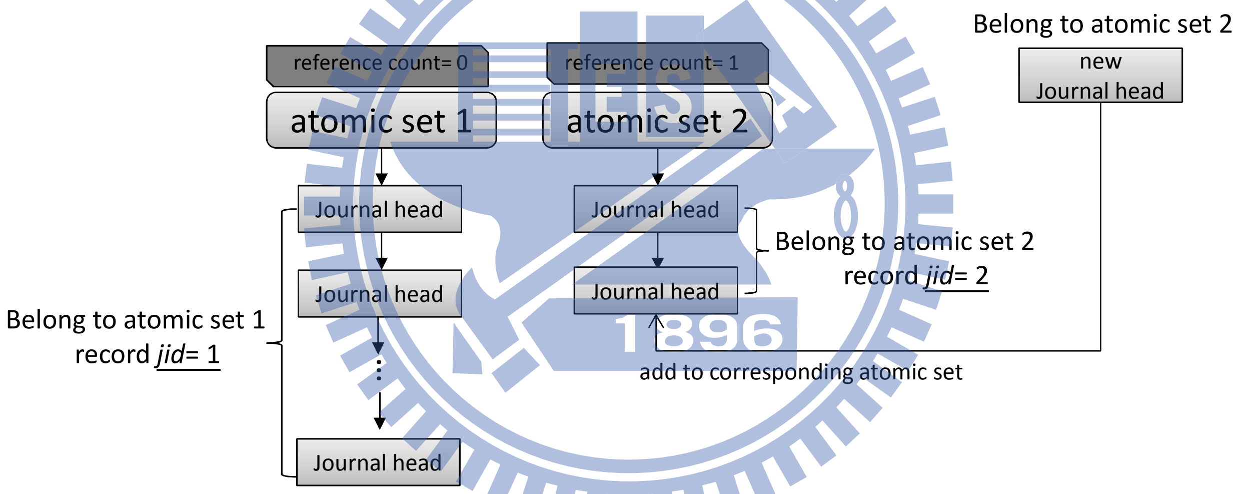


Figure 3.11: The relation between journal heads and atomic sets

Chapter 4

Experimental Results

4.1 Experimental Setup and Performance Metrics

The platform used in our experiments was collected from VessRAID 1841i, a NAS (Network-Attached Storage). Note that the memory used in VessRAID 1841i was provided courtesy of Promise Inc. with a battery-backed memory, consists of 1GB SDRAM (contain 256MB disabled RAID cache) and two 18650 lithiumion battery, each has 2200mAh and the voltage 3.6V. Table 4.1 is a platform we used.

Platform	Operating system	File system
NAS	Linux 2.6.17	Ext3

Table 4.1: Characteristic of the experimental platform

At the beginning, we used instruction *cp* to copy a file on VessRAID 1841i, once the prompt returned, let the system power off and power on immediately. Since our proposed recovery scheme can save more data than original recovery scheme, we compared destination file with source file, using the data lost&correct file ratio as the compared metric. The more correct data exist, means that the more efficient our proposed method has. Second, we used *sysbench* as the benchmark to evaluate the synchronous write performance on database. The total execution time would reduce since we don't need to perform fsync, thus we use the execution time as the performance metric.

Besides, the original ext3 would lose data in committing transaction which doesn't write commit record yet, called unwritten committing transaction, and lose data in running transaction also. In our proposed method, checkpoint-based recovery (CBR) could save data from unwritten committing transactions and file-operation-based recovery (FBR) could save data from running transactions.

Reiterate that the file system we used was Linux 2.6.17 whose file system was Ext3 with ordered mode, setting commit interval to a default value, 5 sec, and the disk volume size was 1 TB. Since it is ordered mode, user data must flush to disk before metadata, so there would not exist the corrupted file, we only discussed the correct and lost file below.

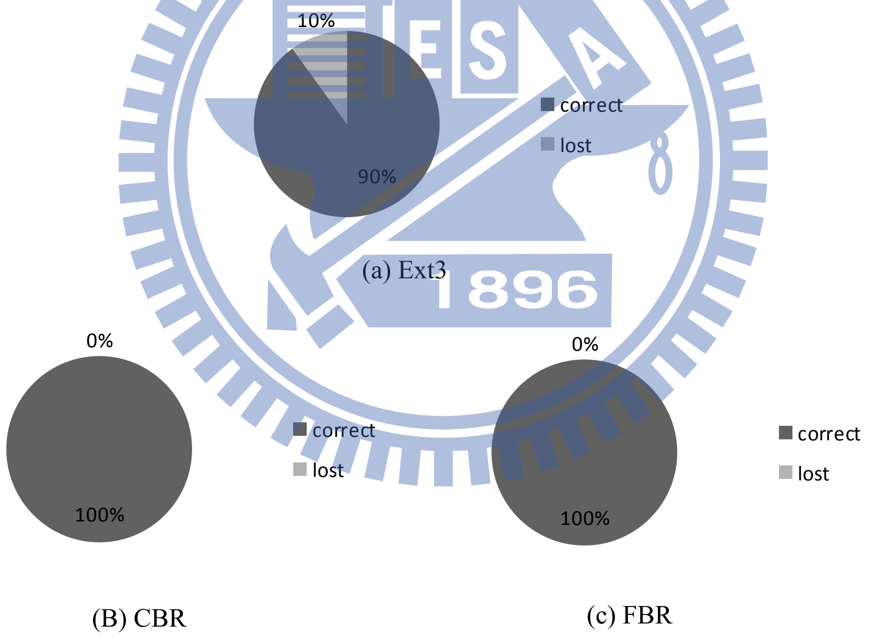


Figure 4.1: commit intercal= 5 sec

4.2 Data Recovery Capability

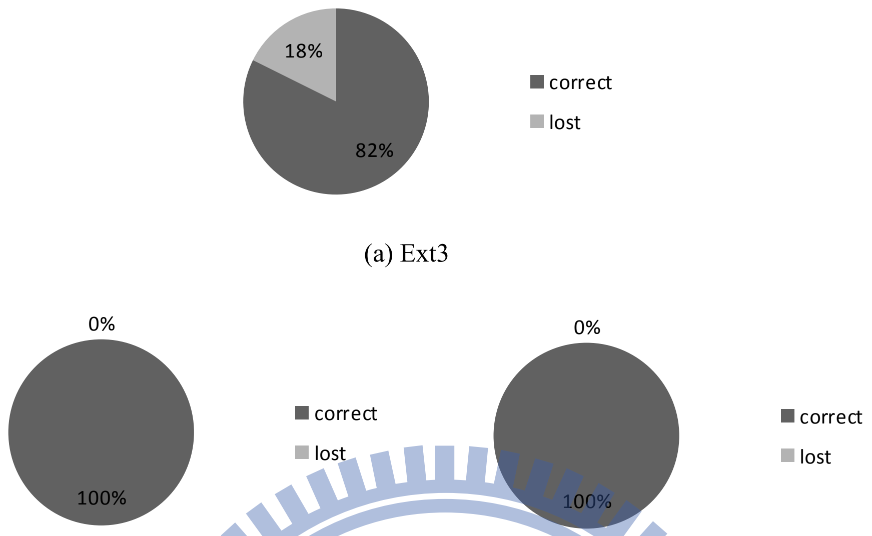
4.2.1 Data loss comparison: Prompt Return

In this experiment, we executed file copy process with a 148MB file consists of 27532 data. Once the prompt returned, let the system power off and power on immediately, and compare three cases: original ext3, CBR, FBR with their recovery capability.

Figure4.1 shows that, in the default commit interval 5 seconds, original ext3 lost 10% of the data. Note that original ext3 with journaling may reduce data loss by changing the commit interval to smaller value, but it may degrade performance. We adjust the commit interval to 10 sec, 20 sec, 40sec, i.e., some larger values, and find that the longer commit interval was, the more data lost in original ext3. On contrast, CBR and FBR could still save 100% of the data since all the file operation are finished. Figure4.2, 4.3, 4.4 show the result of these experiments. Figures 4.4 shows that when prompt return, all the data are in the running transaction, since the commit interval in figure 4.4 is larger than copy process time, thus CBR cannot save any data here. Besides, the reason why sometimes the data recovered by Ext3 equals to CBR and some times FBR equal to CBR, such like figure 4.1, 4.2, 4.3 is that, when prompt return, there might exist committing transaction or running transaction. If there exist running transaction, then CBR is good-for-nothing, thus CBR equals to Ext3. If the last running transaction has transferred to committing transaction when power off, we can only see committing transaction in the memory. Although FBR is good-for-nothing here, FBR still equal to CBR since the data in committing transaction are all transfer from the previous running transaction.

4.2.2 Data loss comparison: Worst-case Scenario

If the system accidentally shut down at the moment that the committing transaction have not write the commit record and the running transaction has not locked, there would have the maximum discrepancy between our proposed method and original ext3.



(a) Ext3

(B) CBR

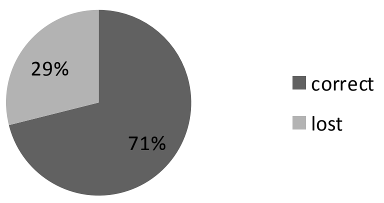
(c) FBR

Figure 4.2: commit intercal= 10 sec

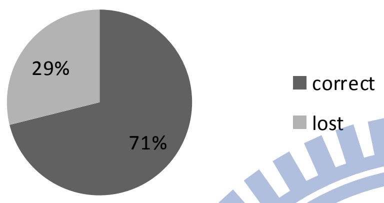
In this part, we show 3 different cases of this experiment. We setup a special circumstance to show the superiority of our recovery scheme. When the running transaction meet the commit interval, it would transfer to a committing transaction, once it transfers to a committing transaction, CBR could save all the data in committing transaction, as the arrow marked 1 shows in figure4.5. The original ext3 could save data only in the transaction which has written commit record, the arrow marked 2 in figure4.5 shows this circumstance. Other narrows, i.e., narrow 3 to narrow 5 mainly shows the superiority of FBR. At this point, i.e., 27 sec, FBR could save nearly 88% of the data, since it contains data in running transaction. CBR could save about 80% since it contains a committing transaction which has not written commit record yet. However, there does not have committing transaction which has written commit record, original ext3 could save only 61% of the data. In figure4.5 we show the maximum discrepancy between our proposed method and original ext3.

4.3 MySQL+Sysbench

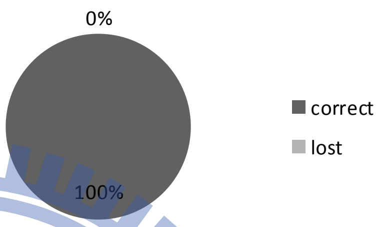
The tools used in this experiment is shown in Table 4.2. Since the general database consists lots of synchronous write during its operation, we used MySQL as our database workload,



(a) Ext3



(B) CBR



(c) FBR

Figure 4.3: commit intercal= 20 sec

and *blktrace* we used to observe how many synchronous/asynchronous write were involved in.

Tools	Description
BlkTrace	a block layer IO tracing tool
SysBench	a benchmark use for evaluate database server performance
MySQL	a database

Table 4.2: The tools used for experiment

SysBench is a database benchmark tool developed by MySQL that supports both common and distributed database. The BlkTrace evaluate both read and write performance, but here only care about write performance of synchronous write, we only discuss write portion below.

MySQL frequently performs fsync to ensure that modified data reach stable storage, since CBR and FBR using battery-backed memory, it doesn't need to perform fsync once the file operation completed. The traditional file system may wait to ensure all the data is flushed to the stable storage during database operations, waste lots of time. Since we use battery-backed memory, a kind of stable storage, no matter the power failure or not, we can always keep the data there, do not lose them, and shortens the waiting time, hence

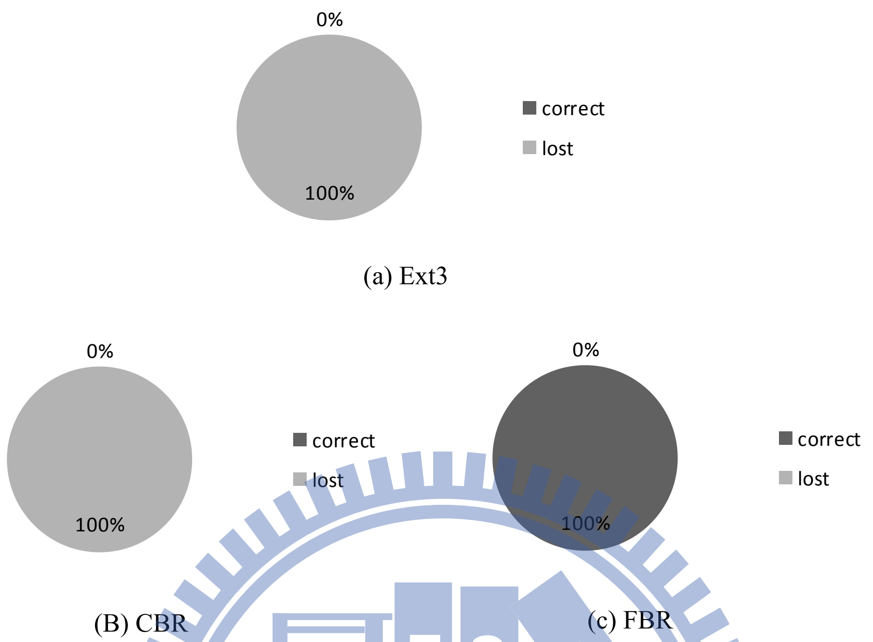


Figure 4.4: commit intercal= 40 sec

improve the performance of synchronous write.

Now we disable fsync since we had a battery-backed memory as a stable storage. We setup parameter for the SysBench with 16 threads, 10,000 table size, and 10,000 requests. The request operations within SysBench include alter-table, large table, connect, create, insert, select and transaction. Figure 4.6 shows the result of the experiment.

As the figure4.6 shows, the performance without synchronous write is better than that with synchronous write, and this is what we expected.

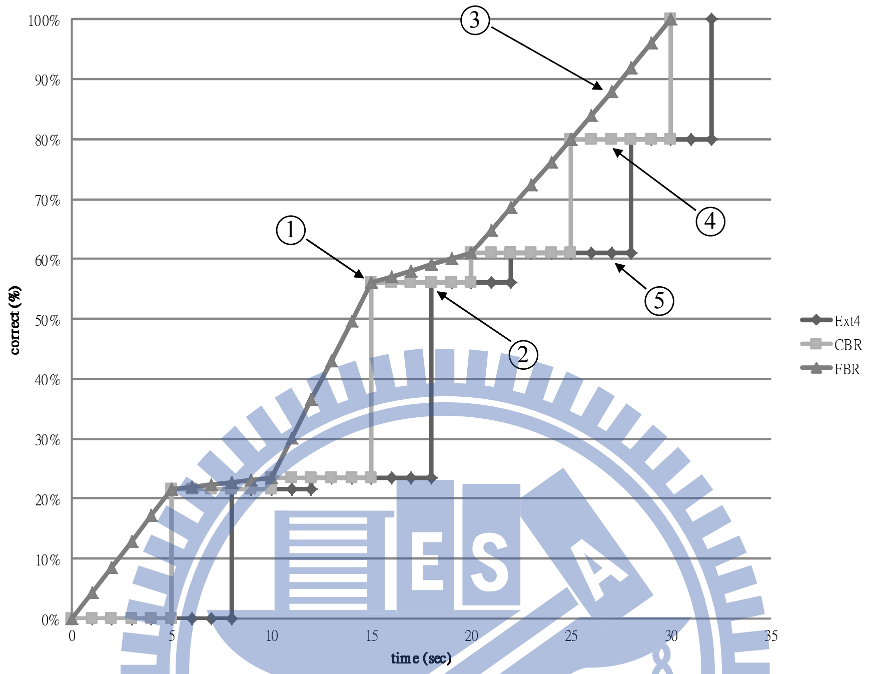


Figure 4.5: Three methods comparison

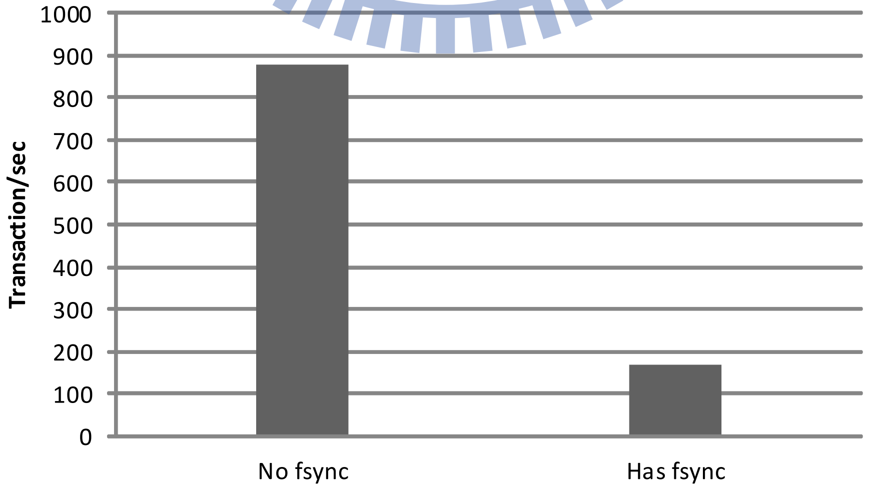


Figure 4.6: Sysbench Evaluation

Chapter 5

Conclusion

This work introduces a technique based on battery-backed memory to protect file systems from losing data across power failures and guaranteed file system integrity. The idea is simple: when power restores after a power outage, our method salvages modified but unwritten data by scanning and parsing the contents in memory. We have implement our method on Ext3 file system and the experimental results shows that our methods definitely improve recovery ability compared to the ordinary file system recovery. Besides, our method is efficient in terms of hardware cost and time/space overheads and also make some integrity-aware applications like databases have much better performance.

Bibliography

[1] A. A.-D. Remzi Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 0th ed. Arpaci-Dusseau Books, 2012.

[2] S. Tweedie, "Ext3, journaling filesystem," in *Ottawa Linux Symposium*, 2000, pp. 24–29.

[3] A.-I. A. Wang, P. Reiher, G. J. Popek, and G. H. Kuenning, "Conquest: Better performance through a disk/persistent-ram hybrid file system," in *Proceedings of the 2002 USENIX Annual technical Conference*, 2002.

[4] M. Baker, S. Asami, E. Deprit, J. Ouseterhout, and M. Seltzer, "Non-volatile memory for fast, reliable file systems," in *ACM SIGPLAN Notices*, vol. 27, no. 9. ACM, 1992, pp. 10–22.

[5] M. H. Eich, "Main memory database recovery," in *Proceedings of 1986 ACM Fall joint computer conference*. IEEE Computer Society Press, 1986, pp. 1226–1232.

[6] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 14–23.

[7] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 133–146.

[8] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara, "Snapmirror®: file system based asynchronous mirroring for disaster recovery," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*. USENIX Association, 2002, pp. 9–9.