A Behavior Based File Checkpointing Strategy

Yifan Zhou

Instructor: Yong Wu

Wuxi Big Bridge Academy

Wuxi, China

A Behavior Based File Checkpointing Strategy

Yifan Zhou

Wuxi Big Bridge Academy
Wuxi, China

Abstract

Checkpoint/restart is an important mechanism for system fault tolerance. It saves the state of an executing program periodically and recovers it after a failure. As many applications involve file operations, supporting file rollbacks is essential for checkpoint/restart. Complete backup can restore files to the correct state, but its cost is too high. In this paper we propose a behavior based file checkpointing strategy (BBFC), which provides a correct recovery of file data and ensures consistency between file state and other states of a process when a rollback is done by restarting the program from the last checkpoint. BBFC classifies details of the file operation behaviors and provides a guidance on what to be saved during file checkpointing according to those behaviors. It dramatically minimizes the overhead of file checkpointing due to the reduction of file data which need to be saved. And it is transparent and easy to use.

Keywords

computer software; file checkpointing; behavior based; rollback; recovery; consistency; checkpoint interval

Contents

1. Introduction

With the development of information technology, computers are widely used in scientific computing, data analysis, information processing and other fields. As a result, system reliability has become an important concern. Checkpoint/restart is a significant mechanism for system fault tolerance. It saves the execution state of a running process to disk at a certain point of time, and rolls back to the saved state to continue execution at a later time. It can reduce the amount of lost work after certain types of failures comparing with starting execution from the beginning. And it also provides a practical solution when there's not enough computing resources for long-running applications.

Since most applications cannot work without data processing and file accessing, file operations occur frequently. So it is quite important for checkpoint/restart scheme to support file rollbacks. Implementation of file checkpointing should provide correct recovery of file contents, and ensure the consistency between file state and other states of a process when restarting a program to the last checkpoint. Complete backup of file attributes and file contents can ensure consistent file checkpointing. But as applications involve more and more large files in practice, complete backup is obviously too expensive.

In this paper we propose a behavior based file checkpointing strategy (BBFC), which can dramatically minimize the overhead of file checkpointing due to the reduction of file contents which need to be saved in most cases. And as a result, it makes the checkpoint/restart scheme more practical.

2. Background

There are several existing checkpoint/restart projects like Condor[1], libckpt[2], libckp[3], CRAK[4], and BLCR[5]. Condor, libckpt and libckp use a user-level strategy for implementation, with the operating system unmodified. User-level implemented checkpointing has a good portability, as well as quite a few restrictions. It may require source code modifications or re-linking to applications. And since kernel data structures are not accessible in user-space, it cannot save a complete process state including operating system environment. CRAK and BLCR implement checkpoint/restart scheme at the kernel level. System implementation requires modification to the operating system kernel. It guarantees a complete checkpoint of all the running states and a transparent recovery at a later time. Berkeley Lab's Linux Checkpoint/Restart project (BLCR) is a practical implementation, which consists of two kernel modules, some user-level libraries, and several command-line executables. It provides support for checkpoint/restart of single threaded applications, multithreaded applications, process trees and process groups in Linux[5].

It is essential for checkpoint/restart scheme to handle correct saving and recovery of all the file states related to the program execution, which consist of file attributes and file contents. Some file attributes are independent of program execution, such as file name, file owner and file permissions, while others are dependent of program execution, such as file description, access mode, file size and file offset. File contents belong to the persistent state, which will not be lost after the program's termination. When some modifications are made to an active file since the last checkpoint and the pre-modified data are not saved, an incorrect rollback may occur due to the inconsistency between file state and other states of the process.

Saving file attributes as well as all of the file contents can guarantee a correct file rollback. But it also leads to unacceptable run-time and space overhead in applications using large files. As a result, most existing checkpoint projects, including Condor[1] and BLCR[5], save only file attributes like file name and file offset at the time of checkpoint, rather than entire file states. A few checkpoint implementations provide entire file state's rollback. Libckp is one of the library implementations developed by AT&T Labs. It regards the contents of an active file as part of the process state and makes a shadow copy of opened file before it is changed or deleted[6]. Libfcp, an improved version of libckp, uses an in-place update with undo logs approach to checkpointing files. Libfcp creates an undo log of restoring the pre-modification data when the file is about to be modified. When a rollback occurs, these undo logs are applied in a reversed order to restore the original files[7]. Though optimization applied, these two strategies are still quite expensive for checkpointing files. LIBVFO[8], developed by Tsinghua University, uses a type of delayed-write strategy. LIBVFO temporarily buffers the file operations occurring in the interval between checkpoints in the system memory, and writes them back into the file at the time of next checkpoint. LIBVFO adds a level of specific virtual file system in the kernel and makes multiple changes within the operating system, which is quite difficult to implement.

3. File Operation Behaviors and Checkpointing Strategies

According to the access mode of a file, we can predict the behavior of file operations. For example, files opened with a read-only mode can only be read, while files opened with a write-only mode can only be written. And files opened with a read/write mode may randomly be read or written. Suppose the program terminates at time t which is a point of time after checkpoint n, and then rolls back to the state of previous checkpoint n, to continue execution. What should be saved for checkpoint can be different according to the behavior of file operations.

3.1. Checkpointing Strategy for Files Opened with Read-Only Mode

Since the contents of files opened with read-only mode will not be changed by the program, only file attributes including file name, access mode and file offset need to be saved. It is unnecessary for file contents to be saved as part of the checkpoint. The file states will be correctly recovered when the file is reopened and the file attributes are restored at a restarting time.

3.2. Checkpointing Strategy for Files Opened with Write-Only Mode

Files opened with write-only mode may be modified during the interval between checkpoints. In the first case, some part of the file is modified. Figure 1(a) gives the file state at checkpoint n, and Figure 1(b) gives the state of file modified at time t, which is a point of time after checkpoint n. In such a situation, only file attributes including file name, access mode and file offset need to be saved at the time of checkpoint n. After a rollback, restarting process will continue correct execution because the subsequent re-write operation at the same file position will overwrite the dirty data. So it is unnecessary to save pre-modifications.

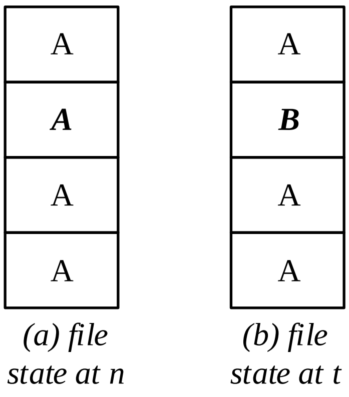

Fig.1 File state before and after being modified

In the second case, file is appended. Figure 2(a) gives the file state at checkpoint n, and figure 2(b) gives the state of file appended at time t, which is a point of time after checkpoint n. Since opening a file with an append mode leads to all write operations to be forced to the end-of-file, the file size at the time of checkpoint n must be recovered as well as other file attributes when a rollback is initiated. Otherwise, if the file is not truncated to the previous size when a rollback occurs, it will lead to an inconsistent recovery due to the twice appending of same data. Same as the first case, file contents need not to be saved. Restarting process will continue correct execution due to the overwriting of the dirty data by the subsequent re-write operation at the same file position.

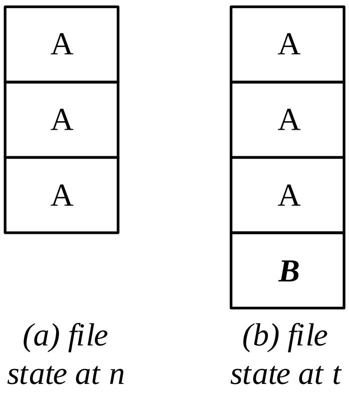

Fig.2 File state before and after being appended

3.3. Checkpointing Strategy for Files Opened with Read/Write Mode

Files opened with read/write mode may have more complex behaviors which are demonstrated as below.

1) There is no write operation occurring in the interval between checkpoint n and time t, with file contents unchanged. As a result, only file attributes need to be saved as part of the checkpoint n, but not the file contents.

2) There are write operations occurring in the interval between checkpoint n and time t. We suppose a write operation occurs at time w, which is a point of time between checkpoint n and time t, and there is no read operation occurring in the interval between checkpoint n and time w. Then the file states can also be demonstrated by Figure 1 or Figure 2. In both circumstances, file attributes including file name, file size, access mode and file offset should be saved at the checkpoint time. When the program is restarted, file attributes are restored and the file is truncated to the recorded size. Since the subsequent re-write operation at the same position will overwrite the dirty data, the file contents do not need to be saved.

3) There are write operations occurring in the interval between checkpoint n and time t. We suppose a write operation occurs at time w which is a point of time between checkpoint n and time t, and there is a read operation occurring at time r which is a point of time between checkpoint n and time w. As Figure 3 shows, with no file contents saved, an error will occur when the program rolls back to the state of checkpoint n and restarts execution because what is read at time r' is different from what is read at time r. So if there is a read-before-write operation related with the same segment of file occurring in the interval between checkpoints, pre-modifications must be recorded in the checkpoint.

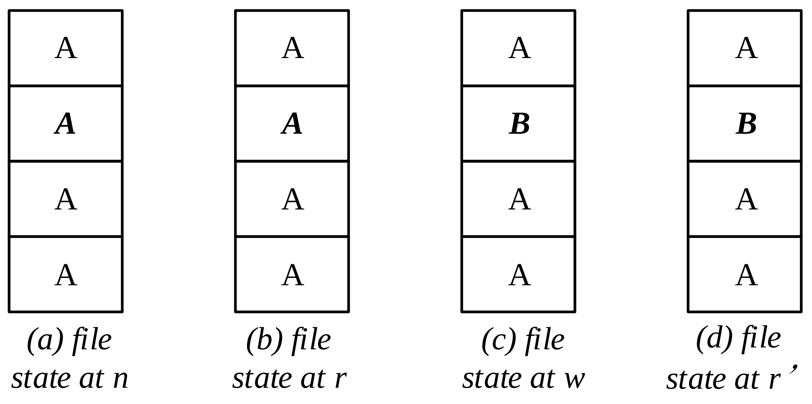

Fig.3 File being read before modified

4. Algorithm and Implementation of BBFC

4.1. Basic Idea of BBFC

A complete backup of attributes and contents of a file is a straightforward way to keep the consistency between file state and other process states in checkpoint/restart scheme. But it also brings unacceptable run-time and space overhead which would seriously affect the system usability. In this paper we propose a behavior based file checkpointing strategy (BBFC), based on the behaviors of file operations. BBFC can substantially reduce the file contents which need to be saved in the checkpoint in most cases, and make checkpoint be taken with a much lower overhead.

The basic idea of BBFC is to classify details of the file operation behaviors and save as little file contents as possible to lower the overhead in accordance with the file operations' behavior characters. File checkpointing strategy is developed according to the mode which the file is opened with. For files opened with read-only or write-only mode, only file attributes need to be included in the checkpoint rather than file contents. To do this, first save file name, file size, access mode and file offset at the checkpoint time, and then reopen the file when the process rolls back, and restore the file attributes as well as truncate the file to the recorded size.

For files opened with read/write mode, when a read-before-write operation related with the same segment of file occurs in the interval between checkpoints, pre-modifications should be saved as well as file attributes to avoid dirty data being read by the restarting program. In other cases only file attributes should be saved. To do this, the read or write operations should be marked in the R/W state maps of the files which are not only opened with a read/write mode but also to be checkpointed.

Files are divided into blocks of same lengths. When the first write operation occurs after a read operation associated with the same file block, offset and pre-modified contents of the file block will be recorded in a log file. After a rollback, restarting process will not only reopen the file, restore the file attributes and truncate the file to the recorded length, but also write the contents of the file block saved in the log file back to the original file. Offset and data of the file block was recorded when the first read-before-write operation associated with the same file block occurs during the checkpoint interval. As a result, one file block is recorded at most once per checkpoint interval even if there are multiple write operations occurring at a later time, thus avoiding repeated recording of the same file block. This scheme can lower the overhead of file checkpointing and also make the file state after rollback independent of the written sequence of the file blocks from the log file.

4.2. Data Structures

4.2.1. R/W State Maps

BBFC marks file operations by file blocks. Files are divided into blocks of same lengths. Even when there are multiple read-before-write operations associated with the same file block in the interval between checkpoints, the file block is recorded only once per checkpoint interval, which obviously lowers the overhead of file checkpointing. Read state map and write state map are created when a checkpoint is initiated or a program is restarted.

As Figure 4 shows, one bit in the read state map or write state map corresponds to one file block. The bit in the read state map is set to 1 when the corresponding file block has been read since last checkpoint, and it is set to 0 when the corresponding file block has not been read since last checkpoint. Similarly, the bit in the write state map is set to 1 when the corresponding file block has been written since last checkpoint, and it is set to 0 when the corresponding file block has not been written since last checkpoint.

We suppose the file size at the time of checkpoint n is L and the size of file block is B, then the size of R/W state maps will be L/(8B) bytes aligned. Since there must be an append operation before any read operations if the range of file access is outside L, which is the file size at the last checkpoint time, these kinds of file operations do not need to be marked in R/W state maps due to the inconformity with the read-before-write sequence.

The data structure of fileop_state_map is shown as below.

```
#define BBFC_BLOCK_SIZE 4096     //Size of file block, tentatively as 4096B；
struct fileop_state_map{
    char BBFC_flag;                  //BBFC flag, 1 - true, 0 - false;
    unsigned long file_length;       //File size at the time of last checkpoint;
    char * r_state_map;              //Pointer to the read state map;
    char * w_state_map;              // Pointer to the write state map;
};
```

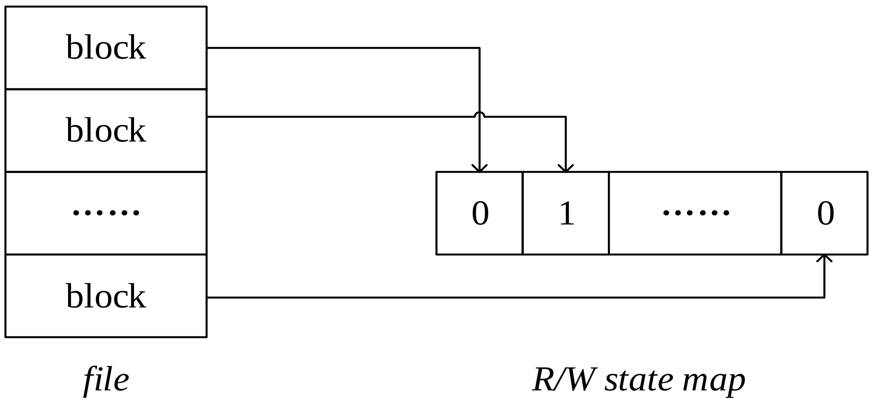

Fig.4 Read/write state map

4.2.2. Log File

In BBFC every file opened with read/write mode corresponds to a log file. Log file is created at the first checkpoint after the file is opened, and cleared after checkpoints done. Only file block information which need to be saved for a correct recovery is recorded in the log file, including offset and contents of the associated file blocks. Every recorded file block corresponds to an offset field and a fixed-length data field in the log file. When the process rolls back and recovers, it will read the file block information from the log file, and write file block data back to the corresponding position of the original file. The format of the log file is shown as Figure 5.

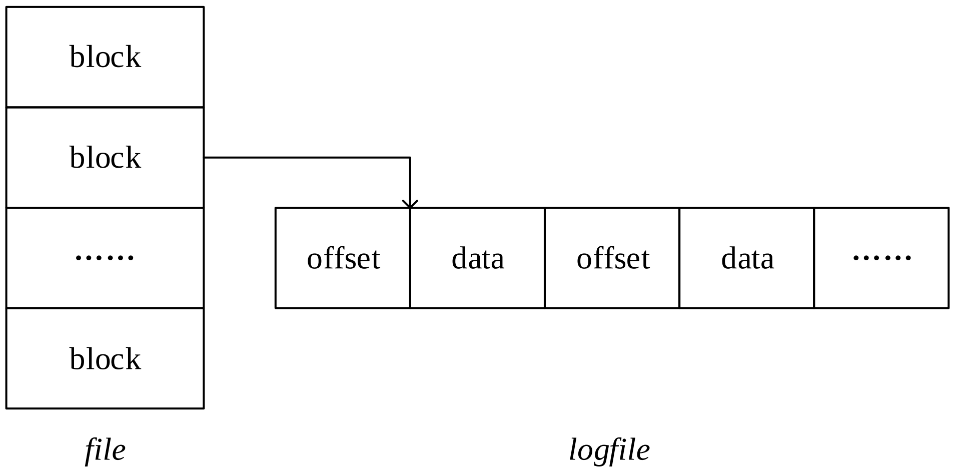

Fig.5 Log file format

4.3. Algorithm

4.3.1. Encapsulation of System Calls of File Operations

To monitor the behavior of file operations, BBFC encapsulates system calls like read, write, truncate and unlink, and gets detailed information about file operations in these functions to conduct R/W state maps' maintenance and file blocks' backup. The system calls' encapsulation is transparent to users, and source code modifications to applications are not required. The detailed algorithms are given below.

```
read(){
    if(BBFC_flag==1){
        for each bit in the read state map corresponding to the read file blocks
        in the range of L {
            corresponding bit in read state map = 1;
        }
    }
    original_read();
}
write(){
    if (BBFC_flag==1){
        for each bit in R/W state maps corresponding to the written file blocks
        in the range of L {
            if(corresponding bit in read state map ==1 &&
            corresponding bit in write state map ==0){
                record offset and data of the corresponding file block
                to file.log;
            }
            corresponding bit in write state map = 1;
        }
    }
    original_write();
}
```

Truncate can be regarded as a write operation related to the truncated range, and unlink can be regarded as a write operation related to the range of entire file. Both implementations are similar to write.

4.3.2. File Checkpoint and Restore

The detailed algorithms are given below.

```
file_checkpoint(){
    for each opened file in checkpointed process{
        save file attributes including file name, file size, access mode, file offset;
        if(opened with read/write mode){
```

```
                    BBFC_flag = 1;
                    file_length = L;                        //current file size;
                    if(r_state_map != NULL)
                        free r_state_map;
                    generate r_state_map initialized with zero;
                    if(w_state_map != NULL)
                        free w_state_map;
                    generate w_state_map initialized with zero;
                    if(file.log not existing){
                        create file.log;
                    }else
                        clear file.log;
                }
            }
        }
        file_restore(){
            for each file recorded in the checkpoint{
                open file and restore file attributes;
                truncate the file to the recorded size;
                if(opened with read/write mode){
                    read offset and data of each file block from the file.log,  and
                    write the data back to the original file according to the
                    corresponding offset;
                    BBFC_flag = 1;
                    file_length = L;                   //current file size;
                    generate r_state_map initialized with zero;
                    generate w_state_map initialized with zero;
                    clear file.log;
                }
            }
        }
```

4.4. Performance Evaluation

Since file operations occur frequently, performance is an important concern.

BBFC is a behavior based file checkpointing optimization strategy. The run-time overhead of BBFC mainly includes extra overhead of both checkpointing and file operations. We choose two extreme cases to demonstrate it. In the first case, files are opened with read-only or write-only mode. In the second case, files are opened with read/write mode and conducted with a read-before-write operation to each file block in the checkpoint interval. The overhead of remaining cases must be between the above two. The implementation of BBFC is based on linux2.6.32 and blcr-0.8.5[9], with the fundamental implementation of checkpoint/restart based on BLCR. While BLCR saves only file attributes like pathname and offset for a checkpoint, BBFC supports a correct rollback of file contents and thus makes checkpoint/restart more practical. The test runs on CentOS 6.5, using 1G memory and 40G disk, with the file block size of 4096 bytes. The test result is given in Figure 6, with a file size (G) as horizontal axis, and a run-time overhead (S) as perpendicular axis.

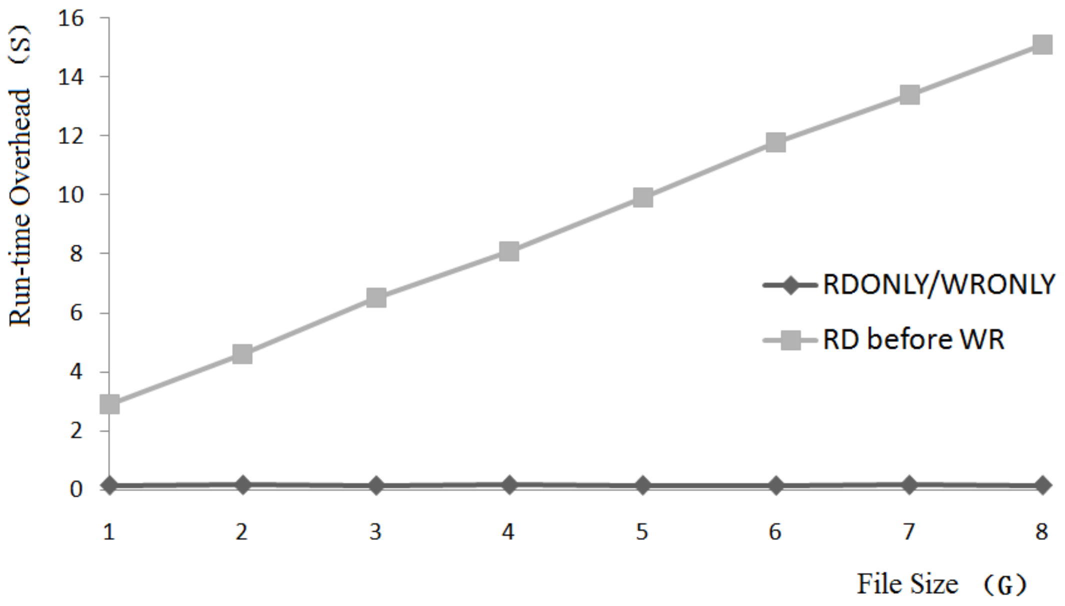

Fig.6 Time overhead of BBFC in two extreme cases

For files opened with read-only or write-only mode, only file attributes are saved as part of the checkpoint. In this case, program execution is almost unaffected by BBFC due to the fairly low overhead. For files opened with read/write mode, maintenance of R/W state maps is required before file operations like read or write. Since the R/W state maps are created in system memory, the overhead of bitmap access is quite low. In case of read-before-write operations associated with the same file block in the interval between checkpoints occurring, backup of relevant file blocks is necessary and this part of overhead is relatively high. As Figure 6 shows, in this case, run-time overhead is almost in direct proportion to the number of file blocks being recorded.

BBFC records file block data only when the file block is associated with a read-before-write operation in the checkpoint interval, while all of the other cases like read-only, write-only and read-after-write are excluded. Furthermore, the corresponding file blocks are recorded at most once per checkpoint interval. By comparing with libckp's shadow copy of the entire file and libfcp's recording of all pre-modifications with an undo log, BBFC can dramatically reduce the file data that need to be saved in most cases, which leads to a much lower run-time and space overhead than the other two file checkpointing systems. Also, BBFC is transparent and easier to implement than LIBVFO due to less modification to the operating system kernel.

5. Conclusion

BBFC strategy implements a complete function of file checkpointing, which provides a correct recovery of file attributes and file data and ensures consistency between file state and other states of a process when a rollback is done by restarting the program from the last checkpoint. BBFC is implemented in Linux kernel. It's transparent to users, and modifications to source code of applications are not required. BBFC substantially reduces the file contents that need to be saved by classifying and analyzing details of the file operation behaviors and effectively lowers the overhead of file checkpointing. As a result, BBFC has a quite good performance by comparing with other file checkpointing systems like libckp and libfcp.

In BBFC, the size of file block is actually very important for the performance of file checkpointing. Too big file block may lead to too much useless data being saved, while too small file block may lead to too frequent saving. In the future work, we are going to design an appropriate dynamic file segmentation strategy by analyzing file operation behaviors of specific applications, in order to further optimize the performance of BBFC.

Referances

[1] Michael Litzkow, Todd Tannenbaum, Jim Basney, et al. Checkpoint and Migration of UNIX Processes in the Condor Distributed System [OL]. [2017-08-29]. http://www.cs.wisc.edu/condor/doc/ckpt97.ps.

[2] J S Plank, M Beck, G Kingsley, et al. Libckpt: Transparent Checkpointing under UNIX [A]. Usenix Winter 1995 Technology Conference[C]. New Orleans, Louisiana: Usenix, 1995.213-223.

[3] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, et al. Checkpointing and Its Applications[A]. International Symposium on Fault-Tolerant Computing[C]. Pasadena, California: IEEE, 1995. 22-31.

[4] Hua Zhong, Jason Nieh. CRAK: Linux Checkpoint / Restart as a Kernel Module [OL]. [2017-08-29]. http://systems.cs.columbia.edu/files/wpid-cucs-014-01.pdf.

[5] Jason Duell. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart[OL].[2017-08-29].http://crd-legacy.lbl.gov/~jcduell/papers/blcr.pdf.

[6] Eric Roman. A Survey of Checkpoint/Restart Implementations [OL]. [2017-08-29].http://crd.lbl.gov/assets/pubs_presos/CDS/FTG/Papers/2002/checkpointSurvey-020724b.pdf.

[7] P E Chung, Y Huang, S Yajnik, et al. Checkpointing in CosMic: a User-level Process Migration Environment [A]. Proceedings of the 1997 Pacific Rim International Symposium on Fault-Tolerant Systems[C]. Washington DC, USA: IEEE, 1997.187-193.

[8] Liu Shaofeng, Wang Dongsheng, ZhuJing. A File Checkpointing Approach Based on Virtual File Operations[J]. Journal of Software, 2002, 13(8):1528-1533 (in Chinese).

[9] Berkeley Lab Checkpoint/Restart for Linux (BLCR) Downloads [OL]. [2017-08-29].http://crd.lbl.gov/assets/Uploads/FTG/Projects/CheckpointRestart/downloads/blcr-0.8.5.tar.gz.

Acknowledgements

In the past months, I read many papers and existing open source projects, and realized the problems and challenges of checkpoint/restart scheme as an important mechanism of system fault tolerance. After analyzing the behavior of file operations, I tried to design a strategy to optimize the performance of file checkpointing and make it more practical. In this paper, I used an open source project BLCR to implement the basic function of program's checkpointing and restoring, and added an extra support for file rollbacks on the basis of linux2.6.32 and blcr-0.8.5. The performance of BBFC is so far quite satisfactory by comparing with other file checkpointing implementations.

Here I would like to thank my instructor who gives me patient guidance and great encouragement during my research process. I also want to thank my classmates, who would always give me a hand when I have difficulties. Finally I'd like to thank my parents, all my teachers and my friends. Thank you!