Lightweight Incremental File Checkpointing

Shuangjian Wei\textsuperscript{1,\textcopyright}, Qiurui Chen\textsuperscript{2,\textcopyright,\textsuperscript{b}}
\textsuperscript{1}Beijing Institute of Technology, Beijing 100081, China
\textsuperscript{2}Science and Technology on Special System Simulation Laboratory, Beijing Simulation Center, Beijing 1000854, China
\textsuperscript{\textcopyright}shuangjian_wei@163.com, \textsuperscript{\textcopyright}qiuruich@126.com

Abstract. Checkpoint technology can improve the efficiency of simulation backtracking. Existing checkpointing tools are dedicated to solving system fault tolerance and load balancing problems, and have limited support for persistent data. The overhead of setting and restoring checkpoints for persistent data is critical for simulation backtracking. We designed LibIFC to deal with the problem of checkpoint overhead for persistent data in two aspects. Firstly, the incremental method is used to reduce the space cost of the checkpoint; secondly, a two-way recovery method is used to reduce the time cost of checkpoint recovery. The experimental results show that the above two strategies have achieved significant results in terms of space overhead and checkpoint recovery time overhead.

1. Introduction

Software and hardware failure has been an urgent problem faced by long-running applications [1]. Checkpoint/restart (C/R) is a mainstream fault-tolerant technology that stores the state of a process at a particular time on disk and uses these files to restart the program to avoid data loss when an error occurs [2]. C/R is a useful technology for process migration, load balancing, crash recovery, rollback transactions, job controlling, and many other purposes [3]. Besides, the features of C/R itself can be used for a fast restart of processes. When a process needs to be re-executed, you can use the checkpoint file to start the process directly instead of restarting the process. This feature makes C/R technology a unique advantage in accelerating simulation backtracking. Simulation backtracking is the operation of repeating the simulation program to obtain different simulation results, which is very important for parameter adjustment and result analysis.

A process is an entity that runs in an operating system and usually has an identifier (PID), a register set, an address space, and other specific resources, such as open files. For ease of description, existing theories generally divide process resources into three parts: volatile data, persistent data, and system environments [4]. Volatile data mainly refers to information in memory, including process stack, static, and dynamic data fragments. Persistent data mainly includes disk data, usually referring to user files. The system environment consists of resources that processes need to access through the operating system.

Most existing checkpointing tools provide excellent support for volatile data, but persistent data has not received widespread attention [5]. Persistent data is an integral component of user processes [6]. There are several reasons for file inconsistency, the most important of which is FARW (Failure after Reading and writing the same area). As shown in Figure 1, the process has an error after reading and...
writing the file. If the process resumes execution from ckp0, the next read operation will read the dirty data because the file has changed.

```c
fd = fopen("test", "rb");
Checkpoint ckp0;
 fread (fd, buff_read, buff_size);
 fseek (fd, 0, SEEK_SET);
 fwrite (fd, buff_write, buff_size);
 /* Error occurred*/
Checkpoint ckp1;
```

**Figure 1.** File inconsistency rollback

Since we assume that the execution and recovery of processes performing on the same machine or different devices of the same configuration, the system environment is beyond the scope of this paper. Because of the orthogonality between establishing persistent data checkpoints and volatile data checkpoints, we can design the file checkpoint tool separately. For applications such as simulations that run for a long time and require frequent checkpoints, the efficiency of file checkpoint creation and recovery directly impacts the user experience. LibIFC uses two methods to reduce the time and space overhead of file checkpoints. 1) Incremental file check pointing. 2) A two-way recovery strategy automatically selects a lower cost direction to restore file status.

This paper also includes the remaining four parts. Section 2 provides relevant background information and related works. Section 3 describes the three methods used in LibIFC. Section 4 describes the implementation of LibIFC. Chapter 5 shows the experimental results. Section 6 summarizes this article.

2. Related work

As one of the most commonly used fault-tolerant technologies, C/R technology has created many excellent tools over the years, such as CosMic [7], Condor [8], CoCheck [9], and DMTCP [10] and so on. Here are some checkpoint tools related to this paper.

CRAK mentioned in [11] is a kernel-level C/R tool for Linux. The goal of this project is to develop a lightweight, versatile C/R package. CRAK has five major contributions. 1) Transparent legacy application migration without kernel modification. 2) No run time overhead besides actual checkpoint/restart. 3) True migration without "stub" processes or "home nodes." 4) Virtual networking at a little cost. 5) Network connection migration support. CRAK is the first checkpoint tool to provide network support. Berkeley proposed a kernel-level checkpoint technology called BLCR(Berkeley Lab Checkpoint/Restart) in [12]. In 2002, Berkeley Labs began developing a robust, versatile kernel-level C/R tool. They adopted the CRAK design concept and designed the C/R function as a separate kernel module. BLCR is powerful, not only for checkpoint single-process applications but also for MPI parallel applications. FT-MPI13 and MPICH-V14 also support multithreading. However, due to the limitations of kernel-level checkpoint tools, BLCR has not been widely used. 15. CRIU 16 (checkpoint/restore in userspace) is a user-level C/R tool developed from OpenVZ 17. It uses the Linux/proc file system to get the information needed for an application dump. The CRIU can dump and recover volatile process data while keeping the process ID constant. CRIU has not yet provided support for persistent data. Currently, user-level checkpoint tools are more popular with users, and libckpt 18 is also a user-level checkpoint tool.

Libckp [4] is the first checkpoint library to consider the complete state of an application. It divides process data into three parts: volatile data, persistent data, and system environments. In a fault-tolerant scenario, Libckp uses a shadow copy to resolve file inconsistencies. When an application makes a mistake in a file operation, Libckp replaces the inconsistent file with the previously copied file. Libfcp [6] is part of the Libckp Persistence Checkpoint feature. Libfcp uses undo-log to roll back the file state. The SCR algorithm [19] and Winckp [20] also use this strategy. Libfcp generates a corresponding undo operation for each file operation. Once an error occurs in the file operation, Libfcp will perform an undo
action based on the undo-log to restore the file to its previous state. MOB [5] (modify operation buffer). The core idea is to convert a series of operations between two checkpoints into atomic operations. Use the buffer to store all file operations. Files stored on the hard disk do not change until the checkpoint is set up. If a file operation error occurs, discard the activity in the buffer. Otherwise, write the contents of the buffer to the file.

Metamora [21] is improved based on the MOB. It enhances the management of buffers and operations and adds support for file streams. Metamora paging buffers and manage pages using the LRU algorithm [22]. Also, it uses a binary tree instead of a chain structure to manage operations. Metamora has better performance than the MOB.

3. METHODOLOGY
Time and space overhead are two issues that checkpoint tools must consider. LibIFC solves these two problems using the following two methods.

3.1. Incremental checkpointing
To make the space occupied by the checkpoint as small as possible, we used incremental checkpoint technology. The checkpoint records only part of the operational information between two adjacent checkpoints. These checkpoints are named log checkpoints (log_ckp). It is impossible to restore the file state only using log_ckp. We need another checkpoint that stores full file information, called a full checkpoint (Fullick). Fullick is the starting point for file status rollback. Based on Fullick, the file operation can be rolled back in turn according to the log_ckp record. As shown in FIGURE 2, $S_i$ ($i = 0, 1, 2, 3$) is the state of a file. Fullick and log_ckp are the full checkpoints and log checkpoints for the file, respectively. For different processes, the interval between full checkpoints can significantly affect the speed of checkpoint recovery.

![Figure 2. Fullick and log_ckp](image)

3.2. Two-way recovery
Although skipping some operations can reduce the time overhead of restoring files, the file recovery time overhead will still increase linearly as the number of checkpoints increases. As shown in figure 2, if we roll back from $S_{N+2}$ to $S_1$, when there are multiple checkpoints between $S_i$ and $S_{N+1}$, the time overhead will be high. If the file changes from $S_0$ to $S_1$, the time overhead is minimal. Similarly, the rollback from $S_{N+2}$ to $S_{N+1}$ has the least time overhead. Therefore, LibIFC can not only restore files in reverse but also restore data in the forward direction. LibIFC automatically selects the lowest cost direction to restore the file state.

4. Implementation

4.1. Data structures
LibIFC manages all opened files using a custom file control block (FCB). When the file is opened using the fopen function, LibIFC creates the data structure, then inserts fcb into the file array and returns the fcb subscript. User processes use this integer to manage files, just like file descriptors. Figure 3 shows the structure of the file control block, where $f_{fd}$ is used to store the file descriptor, $pos$ is used to record
the file access offset, isopen is the flag to distinguish whether the file is open, mod identifies the file access mode, entry link points to the file operation list.

| fd  | path | pos | isopen | mod |
|-----|------|-----|--------|-----|
| *entry_link |

**Figure 3.** File control block

Each operation that changes the contents of the file generates a record, and LibIFC uses the op_entry shown in FIGURE 4 to store an action. For activities that cannot change the contents of the file (such as fread, fseek, etc.), LibIFC does not create a new op_entry, only change the pos value in fcb. After generating the poetry, LibIFC will insert it into the op_link and deliver it to other threads. The type in Figure 4 identifies the type of operation, such as fputs, fwrite. Pos_start records the starting position of the file operation. Len_new and len_old indicate the length of buff_new and buff_old, respectively. The data to be written and overwritten are stored in buff_new and buff_old, respectively.

| type | pos_start | len_new | *buff_new | len_old |
|------|-----------|---------|-----------|---------|
| *buff_old | *next |

**Figure 4.** File operation data structure

### 4.2. Forward recovery algorithm
When using the forward recovery algorithm, LibIFC restores full_ckp first. Then read the data in each log_ckp in turn. According to the data recorded in op_entry, LibIFC restores write pointer position first, and then the data in buff_new is written to the file. After all operations are resumed, the file has reached its previous state. The specific algorithm is as follows (see Figure 5).

```
Restore_forward (entries_arr):
    arr_len ← len (entries_arr);
    For i ← 0 to arr_len do:
        /*Traverse all op nodes*/
        Call fseek (fd, entries_arr[i].pos_start, SEEK_SET);
        /*pos where the change occurred*/
        Call fwrite (entries_arr[i].buff_new, 1, entries_arr[i].len_new, fd);
        /*write new data*/
    End for
```

**Figure 5.** Forward recovery algorithm.

### 4.3. Backward recovery algorithm
The backward recovery algorithm is an algorithm that rolls back the file state. Similar to the forward recovery algorithm, LibIFC first restores fullckp and then restores log_ckp in turn. When restoring a file operation, LibIFC first determines where the file has changed, and if it occurs in the middle of the file, writes the old data directly to the appropriate location. If the change occurs at the end of the file, the file is truncated to the length before the operation occurs, and then the old data is written. The specific algorithm is as follows.
5. Experiments
We designed a program with 10 checkpoints to verify our conjecture. There are $O(T \times L)$ file writes between checkpoints, with 10KB data written each time. Compare the time and space cost by creating and recovering the 10 checkpoint data in turn. Here are two objectives of the experiment.

- Compare space overhead with file copying.
- Compare the time cost of restoring file state with libfcp.

```plaintext
Restore_backward(entries_arr):
    arr_len ← len(entry_arr)
    for i ← 0 to arr_len do:
        call fseek(fd,0L,SEEK_END);
        offset_end ← ftell(fd)
        if offset_end = entries_arr[i].pos_end:
            Call ftruncate(fd,entries_arr[i].pos_start);
        Else if offset_end>entries_arr[i].pos_end:
            Call fseek(fd,entries_arr[i].pos_start,SEEK_SET);
        End if
        Call fwrite (entries[i].buff_old, 1, entries_arr[i].lenold, fd);
    End for
```

**Figure 6.** Backward recovery algorithm

**Figure 7.** Log-checkpoint and full checkpoint space overhead.

Figure 7 compares the space overhead of LibIFC and shadow copy. When using a shadow copy strategy, the checkpoint file size increases as the write operation increases. Even if the file between the two states does not change much, it will take up twice the space of the entire file. In the incremental checkpoint, only the first and last checkpoints are complete, and the remaining checkpoints are log checkpoints. As can be seen from the figure, the incremental checkpoint strategy can significantly reduce space overhead. Since checkpoints are established frequently in the simulation backtracking scenario, using LibIFC can save a lot of space overhead.
Figure 8 shows the time overhead of LibIFC and libfcp when recovering checkpoints. Libfcp only supports file state reverse recovery. When using a two-way recovery strategy, LibIFC uses two strategies with the least cost recovery method (this experiment uses the distance-first method). Therefore, compared to libfcp, LibIFC saves nearly half of the time. The definition of minimum overhead is different for different practical applications. Since the operation of any two file states is identical in this experiment, the distance priority is most appropriate. Besides, it is also possible to use $O_{\text{times}}$ to select the lowest cost direction.

6. Conclusion

This paper introduces LibIFC, a library for file checkpoints. LibIFC intercepts file operation requests before they reach the system call. LibIFC takes over the file I/O work and transparently provides checkpoint functionality for applications. When using a wrapper interface, it's like calling a native file manipulation function directly. Compared to existing checkpoint tools, LibIFC can significantly reduce the space overhead of checkpoints. When restoring file status, LibIFC uses a two-way recovery strategy that automatically calculates the cheapest recovery direction and substantially reduces recovery time. Combined with the volatile checkpoint tool, LibIFC can check all application states, including volatile data and persistent data. Experimental results show that space and time overhead of LibIFC can meet the needs of simulation applications.

References

[1] Ifeanyi P. Egwutuoha, David Levy, Bran Selic, Shiping Chen, A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems, The Journal of Supercomputing, September 2013, Vol 65, pp 1302 – 1326.

[2] Miltiadis Siavvas, Erol Gelenbe, Optimum checkpoints for programs with loops [J]. Simulation Modelling Practice and Theory, 2019, 97.

[3] Nuria Losada, Maria J. Martin, Gabriel Rodriguez, Patricia González. Portable Application-level Checkpointing for Hybrid MPI-OpenMP Applications [J]. Procedia Computer Science, 2016, 80.

[4] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung, and Chandra Kintala, “Checkpointing and its applications,” Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers, 27 - 30 June 1995.

[5] Dan Pei, Dongsheng Wang, Meiming Shen, and Weimin Zheng, ” Design and implementation of a low-overhead file checkpointing approach.” Proceedings Fourth International
Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, 14 - 17 May 2000, pp 439 - 441.

[6] Y. M. Wang, P.Y. Chung, Y. Huang, and E.N. Elnozahy, “Integrating Checkpointing with Transaction Processing,” Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing, 24 - 27 June 1997.

[7] Yi-Min Wang, Glenn Fowler, Kiem-Phong, and Yennun Huang,” Checkpointing in CosMiC: A User-Level Process Migration Environment,” Institute of Electrical and Electronics Engineers, Inc.

[8] Litzkow, Mike & Tannenbaum, T & Basney, J & Livny, Miron. (1995). The Condor distributed processing system–checkpoint and migration of UNIX processes. Dr. Dobb's Journal: software tools for the professional programmer. 20.

[9] G. Stellner, "CoCheck: checkpointing and process migration for MPI," Proceedings of International Conference on Parallel Processing, Honolulu, HI, USA, 1996, pp. 526 - 531.

[10] Ansel, Jason, K. Arya, and G. Cooperman . "DMTCP: Transparent checkpointing for cluster computations and the desktop." IEEE International Symposium on Parallel & Distributed Processing IEEE Computer Society, 2009.

[11] Hua Zhong, and Jason Nieh, “CRAK: Linux Checkpoint / Restart As a Kernel Module,” Department of Computer Science, Columbia University Technical Report, CUCS-014-01, November 2001.

[12] Paul H Hargrove1, and Jason C Duell, “Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters,” Journal of Physics: Conference Series, 2006, Vol.46.

[13] Fagg, Graham E., and J. J. Dongarra. "FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world." European Pvm/mpi Users Group Meeting on Recent Advances in Parallel Virtual Machine & Message Passing Interface Springer-Verlag, 2000.

[14] Bosilca, George & Bouteiller, Aurelien & Cappello, Franck & Djilali, S & Fedak, Gilles & Germain, Cecile & Herault, Thomas & Lemarinier, Pierre & Lodygensky, O & Magniette, Frederic & Neri, Vincent & Selikhov, Anton. (2002). MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. Supercomputing. 29-29. 10.1109/SC.2002.10048.

[15] Duell J, and Hargrove Paul, “The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart,” Future Technologies Group white paper, (2003).

[16] CRIU Homepage, https://criu.org/Main_Page, last accessed 2019/7/20.

[17] Romero, Fabian & Hacker, Thomas. (2011), “Live Migration of Parallel Applications with OpenVZ,” 526-531. 10.1109/WAINA.2011.156.

[18] James S. Plank, Micah Beck, Gerry Kingsley, “Libckpt: Transparent checkpointing under Unix,” In Usenix Winter Technical Conference, 213–223 (1995).

[19] WEI Xiaohui , and JU Jiubin, “SCR Algorithm: Saving/Restoring States of File Systems,” Journal of Computer Science and Technology, 2000 (04): 393 - 400.

[20] Emerald Chung, P & Lee, Woei-Jyh & Huang, Yennun & Liang, Deron & Wang, Chung-Yih. (1999). Winckpt: a transparent checkpointing and rollback recovery tool for Windows NT applications. 220 - 223. 10.1109/FTCS.1999.781053.

[21] A. R. Jeyakumar, “Metamori: A library for Incremental File Checkpointing.” Faculty of the Virginia Polytechnic Institute and State University (2004).

[22] Elizabeth J. O’Neil,"The LRU-K page replacement algorithm for database disk buffering,” SIGMOD ‘93 Proceedings of the 1993 ACM SIGMOD international conference on Management of data, 297 - 306 (1993).