Pin-Tool Based Execution Backtracking

Shuangjian Wei[1], Weixing Ji[1(✉)], Qiurui Chen[2], and Yizhuo Wang[1]

[1] Beijing Institute of Technology, Beijing 100081, China
{sjw,jwx,frankwy}@bit.edu.cn

[2] Science and Technology on Special System Simulation Laboratory, Beijing Simulation Center, Beijing 100854, China
qiuruich@126.com

Abstract. Checkpoint/restart is a common fault tolerant technique which periodically dump state to reliable storage and restart applications after failure. Most of existing checkpoint/restart implementations only handle volatile state and lack of support for persistence state of applications. Even the algorithm specifically designed for file checkpointing may not support complex operations and some need to modify source code. This paper presents a new checkpoint technique, which use dynamic instrumentation to temporarily cache disk operations in memory, and use existing memory checkpoint tool to dump or restore process state at runtime. We show that not only can this method create regular checkpoints for both volatile and persistence state, but also has important applications in execution backtracking.

Keywords: Checkpointing · Dynamic instrumentation · Execution backtracking · Volatile state · Persistence state

1 Introduction

Checkpoint/Restart (C/R) is a mainstream fault-tolerant technique. It generates checkpoints periodically to save execution state and recovers from checkpoints after process fails. The behavior of a process has three parts: volatile data, persistent data, and OS environment [1]. Among them, volatile data refers to data in memory and registers, and they are lost after power-off. Persistent data refers to the data stored in stable storage, such as files and databases. OS environment refers to the resources that user processes must access at runtime, such as swap space and monitors. In this paper, we focus on the C/R of both volatile data and persistent data.

The consistency of volatile and persistent data is a prerequisite for process restart. Unlike incorrect recovery of volatile state (which usually leads to obvious process failures), incorrect rollback of persistent state usually leads to more serious losses due to difficulty in tracing, so it has become a major concern for many users. Unfortunately, most existing mainstream checkpoint tools do not support or do not fully support file checkpoints. Fault-tolerant systems that use

© Springer Nature Singapore Pte Ltd. 2020
D. Dong et al. (Eds.): ACA 2020, CCIS 1256, pp. 192–206, 2020.
https://doi.org/10.1007/978-981-15-8135-9_14

such tools roll back the volatile state to the previous checkpoint after a process fails, while keeping the file state unchanged. If the process has modified these files, such as writing, deleting, renaming, etc., it will cause erroneous results. There are many types of these errors, and the following figures show two common cases.

```
checkpoint i;
fd = open("doc",
          O_WRONLY|O_APPEND);
write(fd, buf, buf_size);
/* failure occurs */
checkpoint i+1;
```

Fig. 1. A process writes same data multiple times.

```
fd = open("doc", O_RDWR);
checkpoint i;
read(fd, buf, buf_size);
lseek(fd, 0, SEEK_SET);
write(fd, buf, buf_size);
/* failure occurs */
checkpoint i+1;
```

Fig. 2. A process reads dirty data and causes error.

In Fig. 1, the program opens the file "doc" after checkpoint i, and writes data at the end of the file in appending mode, then an error occurs right before checkpoint $i+1$. During the rollback, since the information of "doc" is not recorded in checkpoint i, the rollback algorithm will not truncate "doc", so that the content will be added again after the execution is resumed. In Fig. 2, the program performs the read-before-write operation at the same position on the "doc" after checkpoint i, and then an error occurs before checkpoint $i+1$. During the rollback, because there is no "doc" information in checkpoint i, the rollback algorithm will not restore the file state, which causes the program to read dirty data after recovery. The above two errors are also known as RARE (Rollback After Real time Event) and RARW (Rollback After Reading and Writing the same area) [2].

Over the past 20 years, many checkpoint algorithms and tools have been proposed. These algorithms and tools play an irreplaceable role in tasks such as scheduling management, process migration, and load balance. Nonetheless, checkpoint related work is far from over, especially in the field of file checkpoints. There are three main shortcomings in existing file checkpoint algorithms:

- Only idempotent operations and very few non-idempotent operations are supported. Idempotent operations include all operations that do not change the consistency state of the file, such as read. User applications that use this type of checkpoint tool can only access files in read-only and appending modes.
- Only active files are supported. Active files are those files that were open at the time the checkpoint was created. Such tools traverse all open file handles and record the current file length when a checkpoint is created.
- The user application source code must be modified to fit the file checkpoint feature. Currently, most of the file checkpoint tools are provided as libraries, and they are implemented by encapsulating file interfaces. User programs that

have been built must modify the source code to accommodate these libraries. This method will not only increase the workload of developers, but also leave security risks for the system.

Although kernel-level checkpointing tools can address all of these issues, they can also introduce significant overhead for applications that do not require checkpointing. This article mainly has the following three contributions. First, this article introduces a new C/R technique that neither modifies program source code nor restricts process file access operations. Secondly, a method to dump the complete state of the process using only the memory checkpoint tool is proposed. Finally, the checkpoint method proposed in this paper not only can set up regular checkpoints, but also assist in execution-backtracking, that is, roll back the process state to any point in the execution history.

In addition to this section, application scenarios are discussed in Sect. 2 and related work is given in Sect. 3. The architecture overview is introduced in Sect. 4, and Sect. 5 presents system implementation details. The evaluation is in Sect. 6 and Sect. 7 is the summary.

2 Application Scenario

To describe the usage scenarios of the ideas presented in this article, it is necessary to first explain how the existing checkpoint tools work. Existing checkpoint tools, such as MOB [2,16] and CprFS [20] mentioned in the following section, will automatically create process checkpoints at regular intervals. When the process execution fails, the system will automatically select the most recent checkpoint, such as checkpoint i in Fig. 1 and Fig. 2, and restart the program from this checkpoint. Restarting the process from checkpoint i discards all changes made to the file during erroneous execution and minimizes work loss, which is also the ultimate purpose of checkpoint tools. Although multiple checkpoints are set during process execution, only one checkpoint (checkpoint i) is involved in the entire C/R process. The process cannot roll back the state to checkpoint $i-1$ or $i-2$, because all file modification data before checkpoint i has been discarded when setting checkpoint i. Therefore, we can conclude that the existing checkpoint tool is to ensure that the target process can be safely and error-freely executed to the end in one execution. Only one checkpoint (the most recent checkpoint) is required to ensure the execution of the process.

Unlike the existing checkpoint tools for program fault tolerance, the checkpoint method proposed in this paper can also be used for execution backtracking. Execution backtracking refers to the operation of rolling back the process state to any moment in its execution history. Taking the simulation programs as an example, in order to obtain the simulation results under different parameters, users need to execute the same simulation program multiple times and enter different parameters for it. To reduce the time it takes to re-execute, we can set a checkpoint before setting the parameters and restart the process from the checkpoint in the next execution. It should be emphasized that the program started from the checkpoint can also set the checkpoint again. These checkpoints will

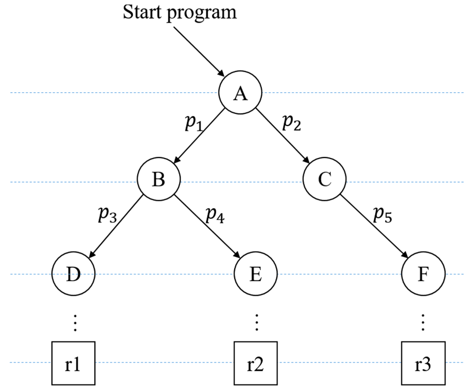

Fig. 3. Tree structure formed by checkpoints

eventually form a tree structure (see Fig. 3), and the process can be restarted and executed from any node in the tree. The non-leaf nodes in the tree in Fig. 3 refer to checkpoints, and the leaf nodes represent program execution results. The strategy proposed in this paper can not only ensure the correct execution of the process, but also meet the requirements of process traceback, that is, restart the program from any checkpoint in the checkpoint tree.

3 Related Work

3.1 Checkpointing

The BLCR (Berkeley Lab's Linux Checkpoint/Restart project) presented in [3–7] is a robust kernel-level checkpoint/restart implementation that can support a variety of parallel scientific codes. In terms of file processing, BLCR only records the file size when creating a checkpoint and simply truncates the file to its original length during recovery. Although this strategy is very lightweight and effective in log-only scientific calculations, it is not applicable in practical applications.

The ftIO system [8] is implemented by encapsulating the standard file interface. In order to avoid file access errors between two adjacent checkpoints, ftIO has designed a new file access protocol. This protocol is based on the copy-on-write [9] concept, where the entire file is copied upon the first write operation. Subsequent file operations are performed on the replica. During checkpointing, the modifications are committed by simply replacing the original file with its replica. The ftIO algorithm is concise and effective, but it introduces a huge time and space overhead when processing large files, which can seriously drag down user processes.

The core idea of Libckp mentioned in [1,10,11] is to use lazy-coordination and shadow copy to solve the problem of inactive files. The basic concept of lazy coordination is that file data is not processed immediately when a checkpoint is

created, but is deferred until the file content actually changes. Just record the file size when the file becomes active, and make a shadow copy of the file when the file content is about to be modified. Libckp's strategy to reduce runtime and space overhead is to perform shadowing page by page.

Libfcp [12] uses in-place updates [13] with undo logs to checkpoint files. It intercepts all file operations except read-only files through the encapsulated file interface. When a file is opened for modification, the size of the file is recorded and a truncated undo log of the file is generated. When the contents of a file are modified, it generates an undo log that restores the contents of the file. Libfcp_RM [14] enhances Libfcp by adding transaction management. It uses transactions to atomize a sequence of file updates in the application. Libra [15] combines a "copy-on-change" strategy with an undo log to keep track of what really changed to reduce the log size.

The MOB (Modification Operation Buffer) mentioned in [2,16] buffers all modification operations after one checkpoint until the next checkpoint, so that all operations between two checkpoints become atomic. MOB's basic buffering strategy is to append new content directly after the existing buffer. If the same area of the file is modified more than twice, the buffer will not append new content, but update the original data in the buffer. MOB transfers the execution of file operations to memory, which can significantly reduce file access time overhead. In addition, MOB uses a disk buffer to limit the amount of memory occupied by the algorithm.

The VFO (Virtual File Operation) proposed in [17] buffers all the write operations after a checkpoint until the next one, making all the operations between two checkpoints atomic. The read and write operations of the user process do not directly interact with the disk file, but access to the virtual file operation management table entries, just like inserting a virtual file layer between the user and the file. Unlike MOB, VFO manages file data in blocks, reducing space overhead. Metamori [18] is an another MOB-like file checkpointing algorithm. It adds support for file streams on top of MOB, which only supports file descriptors. In addition, it also optimizes the related data structure and uses a B-tree to manage the buffer mapping table to improve retrieval efficiency.

CprFS [20] uses the FUSE [21] module in the Linux system to create a file system that executes in user space. For checkpoint, an atomic transaction is considered to be the execution of a program between two consecutive checkpoints. The program either commits its state during checkpointing or aborts at some point during execution, in which case it can be recovered from the last checkpoint. CprFS has high execution efficiency, and does not need to modify the program source code.

3.2 Execution Backtracking

Execution backtracking is the process of restoring the state of a program to any earlier point in its execution history. It is used to facilitate program debugging. The Spyder mentioned by [26] is a system for selective checking of computational sequences. It allows users to step back from the checkpoint without having to

re-execute the program to reach the most recent previous state. [27] describes a debugging method that uses a combination of re-execution and backtracking to find the first difference in the calculation, which may eventually lead to incorrect values indicated by the user. [28] provides a debugging model based on dynamic program slicing and execution backtracking technology that easily lends itself to automation.

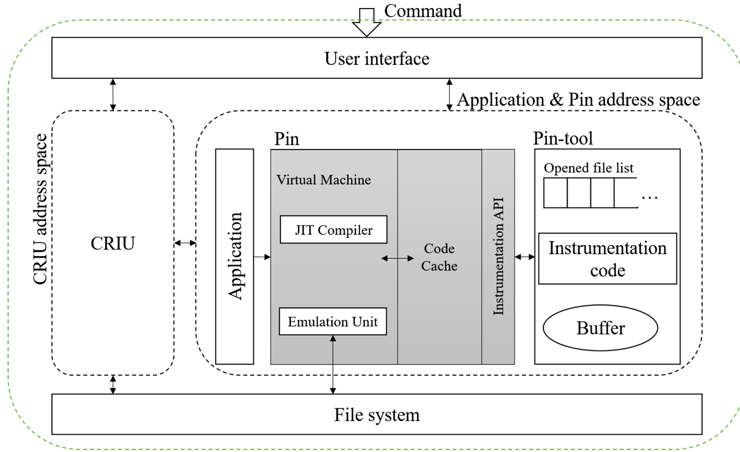

Fig. 4. System architecture diagram

4 Architecture Overview

The traceback system architecture is shown in Fig. 4, where the user interface refers to the client of the system, which displays system status information to users and receives instructions from the users. In addition to this, the user interface is also responsible for managing the tree formed by checkpoints, as well as issuing commands to the checkpoint tool and inputting parameters to the user process. Users can issue checkpoint setting commands through the interface, or select a checkpoint from the tree to restart a process. The checkpoint tool used in the system is CRIU (Checkpoint/Restore In Userspace) [22], which is an open source software on GitHub. CRIU works on the Linux operating system. It can freeze the target process after receiving user commands, and then dump the process data to disk. CRIU completes the checkpoint restarting process by transforming itself into a task to be restored.

Pin [25] allows a tool to insert arbitrary code (written in C or C++) in arbitrary places in the executable. The best way to think about Pin is as a "just in time" (JIT) compiler. The input to this compiler is not bytecode, however, but a regular executable. Here, we use Pin to build a tool (Pin-tool in Fig. 4) for intercepting and caching file operations. Pin-tool is mainly composed of three parts,

a list for managing open files, a series of instrumentation codes for intercepting file operations, and a buffer for buffering file contents. The three components of Pin-tool form a virtual file layer, which can load the contents of the disk file into the buffer, and can also transparently cache the data written by the user process. After CRIU receives the user's request to set a checkpoint, it directly dumps the volatile data of Pin-tool and user application to disk. Because the file modification information is stored in the virtual file layer, each checkpoint created by CRIU contains all the information of the process.

5 Implementation

The disk file always remains the same to ensure that the process can be correctly executed back to any point in the execution history. We use the virtual file layer mentioned in the previous section to buffer all file changes made by the process. Unlike existing methods, we use dynamic instrumentation to intercept and replace the corresponding functions in the process, thereby avoiding the work and risks caused by modifying the source code. Instrumentation is to insert some probes into the program to collect the tested program information on the basis of ensuring the original logical integrity of the tested program. These probes are essentially code segments for information collection, which can be function calls that distribute information or collect information. The code is added dynamically while the executable is running. The functions we intercept and replace include: **open**, **close**, **read**, **write**, **create**, **dup**, **dup2**, **dup3**, **fcntl**, **lseek**, **remove**, etc.

Table 1. Data structure for storing file information.

Name	Description	Name	Description
_fd	File descriptor	_flags	File access mode
_path	File path	_closed	Whether the file is closed
_flags_real	Real file access mode	_pos_wd	File write pointer
_pos_rd	File read pointer	_len_acc	File accessible length
_len_cur	Current file length	_pages	File content buffer

5.1 Data Structure

For each opened file, a global data structure is created to store its information. We call this data structure **FileEntry**. As shown in Table 1, _fd and _flags represent the file descriptor and file open mode, respectively. _path refers to the file path, _closed is used to describe whether the file has been closed. To ensure that the disk file remains unchanged, the virtual file layer will change the file open

mode and record the mode in _flags_real. The following _pos_rd and _pos_wd are file read and write pointers, while _len_acc and _len_cur are the file's accessible length and current length. The open mode of the file will affect the value of _len_acc, and the process of writing will change the value of _len_cur. The virtual file layer manages the file contents in pages and loads the corresponding pages into _pages when needed.

5.2 Virtual File Layer

This section discusses how to transparently perform file operations in the buffer. The virtual file layer is composed of a set of file access functions and a buffer. The main purpose of its existence is to unify memory and disk file so that memory checkpoint tool dumps all process data. The way it works is to intercept all file operations performed by the process and transparently execute the operations in the buffer, so the dynamically inserted code has a completely different role from the native code.

Algorithm 1. New file open function

```
1: function NewOpen(filename, flags, mode)
2:     if flags = O_RDONLY then return open(filename, flags, mode)
3:     end if
4:     if Find(filename, fety) then return ChangeMode(fety)
5:     end if
6:     if access(filename, F_OK)= -1 And (flags&O_CREAT) then
7:         open(filename, O_CREAT, mode)
8:     end if
9:     fety._fd ← open(filename, O_RDONLY)
10:    fety._closed ← false
11:    fety._path ← filename
12:    fety._flags ← flags
13:    fety._pos_rd, fety._pos_wd ← 0
14:    if flags&O_TRUNC then
15:        fety._len_acc ← 0
16:        fety._len_cur ← 0
17:    else
18:        fety._len_acc ← lseek(fety._fd, 0L, SEEK_END)
19:        fety._len_cur ← fety._len_acc
20:    end if
21:    if flags&O_APPEND then fety._pos_wd ← fety._len_cur
22:    end if
23:    g_files_vec.push_back(fety)
24:    return fety._fd
25: end function
```

Each file opened by the user process has an independent buffer containing multiple fixed-size pages. The file just opened by the process does not load any data

into the buffer, and data loading is delayed until the file is actually written or read. To avoid memory overflow caused by excessive file size, the file data always loaded in pages. The file read/write pointer and read/write size jointly determine which page needs to be loaded immediately. The virtual file layer does not handle read-only files, because even if the instrumentation code does nothing, the disk file will not change.

NewOpen is a function for replacing **open** in the virtual file layer. It is used to transparently open a file and return the file descriptor after recording the file information. The virtual file layer does not record any information about files opened in read-only mode, and directly calls **open** and returns the result. For a newly opened file, create a FileEntry instance (fety) and initialize its contents according to the open mode and file status, and finally insert it into the global list. Its detailed description is shown in Algorithm 1, where g_files_vec is the global list used to store information about all open files. The **Find** function is responsible for finding the current file information in g_files_vec to determine whether the file was previously opened. For the file that has been opened, the virtual file layer no longer creates a new FileEntry instance, but uses the **ChangeMode** function to change the file information based on the existing **fety**.

NewClose is a function for replacing **close** in the virtual file layer, and is responsible for closing the opened file descriptor. For files present in g_files_vec, first set the **_closed** flag to true, then close the file descriptor. The reason why the data of the closed file is not deleted is that the user process may reopen the file in the subsequent execution.

Algorithm 2. New file read function

```
1:  function NewRead(fd, buf, count)
2:      if Find(fd, fety) then return fety.ReadFromPages(buf, count)
3:      else
4:          return read(fd, buf, count)
5:      end if
6:  end function
7:  function ReadFromPages(buf, count)
8:      page_str ← _pos_rd/PAGE_SIZE
9:      page_end ← (_pos_rd + count − 1)/PAGE_SIZE
10:     read_num ← 0
11:     for i = page_str → page_end do
12:         LoadOnePage(i)
13:         read_num += ReadFromOnePage(buf + read_num, count − read_num)
14:     end for
15:     return read_num
16: end function
```

NewRead is a function for replacing **read** in the virtual file layer to read a certain number of bytes and return the number of bytes read. When **NewRead**

is called, it first obtains the handle **fety** used to manipulate the file. If **fety** does not exist, it directly calls native **read** and return. As shown in lines 9 and 10 of Algorithm 2, using the _pos_rd of the current file and the parameter count can calculate the pages that may need to be loaded. Then use **LoadOnePage** and **ReadFromOnePage** to load and read out the data in the file (as shown in lines 12 to 17 of Algorithm 2), and finally return the number of bytes read. The judgment of the file boundary (_len_cur) and the change of the read pointer (_pos_rd) are made in **ReadFromOnePage**. The return value may be less than count when touching the file boundary. Because the preset page size is often much larger than the read size, the read operation after a page loads will be much faster than reading directly from the file. The page loaded in the read operation will also speed up the program write operation.

Algorithm 3. New file write function

1: **function** NEWWRITE($fd, buf, count$)
2: **if** FIND($fd, fety$) **then return** $fety$.WRITETOPAGES($buf, count$)
3: **else**
4: **return** $write(fd, buf, count)$
5: **end if**
6: **end function**
7: **function** WRITETOPAGES($buf, count$)
8: $page_str \leftarrow _pos_wd / PAGE_SIZE$
9: $page_end \leftarrow (_pos_wd + count - 1)/PAGE_SIZE$
10: $write_num \leftarrow 0$
11: **for** $i = page_str \rightarrow page_end$ **do**
12: LOADONEPAGE(i)
13: $write_num$ += WRITETOONEPAGE($buf + write_num, count - write_num$)
14: **end for**
15: **return** $write_num$
16: **end function**

The execution flow of the **NewWrite** function used to replace **write** is similar to **NewRead**, and the corresponding page needs to be loaded before writing. The difference is that if the page to be loaded does not exist, **LoadOnePage** will create a blank page instead of doing nothing for writing new data. Unless the memory overflows or other errors occur, the return value is always the same as count.

The core idea of this paper is to use memory buffer file operations to unify volatile and persistent data so that the memory checkpoint tool can dump all the data of the process. The advantage of this strategy is that it is simple and effective for processes that have sufficient memory space or access to files that are not too large, and will not negatively affect the execution speed. But for large files, it may cause a shortage of memory space. It is unrealistic to completely buffer the contents of larger files into memory. If the file accessed by a process is too large, the virtual file layer will write back buffered data to the disk, and at the same time create a file backup on the disk for process backtracking.

In addition to the above four basic file operations, the virtual file layer also supports operations such as **remove**, **rename**, and redirect. When the user program calls the **remove** function, the virtual file layer will first close the corresponding file descriptor, then release the file buffer, and finally set the file accessible length to 0. For the rename operation of the user process, the virtual file layer will change the value of _path in the file entry. User process redirection operations, such as **freopen**, **dup**, **dup2**, **dup3**, etc., will cause the original file descriptor to be closed and create a new file descriptor (or use the specified file descriptor) instead.

6 Evaluation

In this section, we use micro-benchmarks and real-world applications to evaluate our method to prove that dynamic instrumentation and checkpoint overhead are tolerable. The experiment was conducted on a computer with Intel(R) Core(TM) i7-8550U CPU @ 1.80 GHz, 4 GB RAM and a 20 GB disk space. The operating system used was CentOS-7 with kernel 3.10.0-693.el7. The file system for local disk was xfs.

6.1 IOzone Test

How to dump files is the core problem to be solved in process backtracking. The method used in this paper is to insert a virtual file layer between the disk and process through dynamic instrumentation technology. The access speed of the process to the file, especially the speed of writing the file is closely related to the execution speed of the program. We use IOzone [23] to evaluate the execution efficiency of instrumentation code. The experiment uses the original IOzone and the IOzone after dynamic instrumentation to write 1GB data in different block sizes, and then records the writing speed. The experimental results are shown in the figure (see Fig. 5).

The experiment tested the write speed of the xfs file system in different states, where xfs-a and xfs-b respectively represent the write speed of the file system with and without calculating the flush time. The black bar shows the writing speed of the file system after dynamic instrumentation. From the data shown in the figure, we can find that the file access speed after dynamic instrumentation is similar to the native file system, and in most cases is slightly higher than the native file system. Therefore, we believe that the impact of the new virtual file layer on the simulation program is positive, as can be seen from the total time of the simulation program execution in the previous section.

6.2 Pin-Tool Overhead

An important part of the implementation of the backtracking strategy based on checkpoints is dynamic instrumentation. Dynamic instrumentation allows developers to intercept or replace existing methods in the original program without

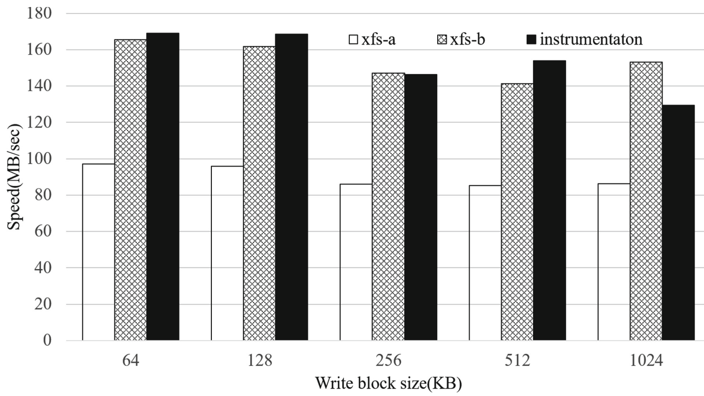

Fig. 5. Write speed of virtual file layer

changing the source code. Obviously, this requires additional memory overhead. In order to detect memory overhead, we instrument the existing program and then sample during the program execution. The test program we use is BWA [24], which is a software package for mapping DNA sequences against a large reference genome (such as the human genome). We can find its source code on GitHub. Figure 6 shows the memory overhead information of the process using 9 samples.

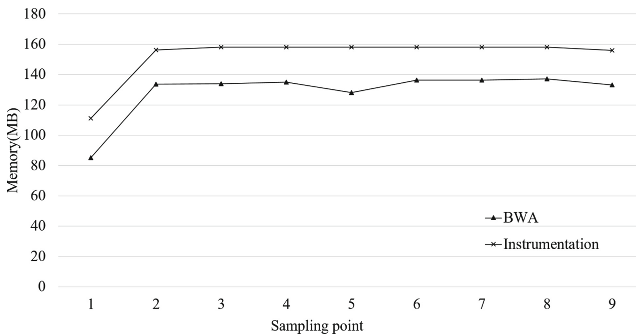

Fig. 6. Memory overhead caused by dynamic instrumentation.

The upper and lower two lines in the figure present the memory occupation trend of the BWA program after dynamic detection and the original BWA program during execution. Since the memory usage of the BWA program in the steady state does not change much during execution, the lower line is almost horizontal. Correspondingly, the memory overhead of the BWA program in the stable state after dynamic instrumentation is also displayed as a horizontal state, which indicates that the memory overhead caused by dynamic instrumentation is an approximately fixed value and does not change with the process size and

execution status. The gap between the upper and lower lines (the difference is about 30M) is the overhead caused by dynamic insertion. This fixed overhead is acceptable for the program.

6.3 Checkpointing Performance

A typical application scenario of process backtracking is simulation backtracking. Setting checkpoints for the simulation program is the core content of the simulation backtracking, and its efficiency is closely related to the backtracking efficiency. We chose a CISE-based [19] simulation program with a run time of approximately 150 s to find the impact of checkpoints on the simulation program execution. The memory checkpoint tool we use is CRIU [22]. We execute the simulation program after instrumentation, then set multiple checkpoints uniformly during its execution, and finally record the execution time of the entire simulation program. The experimental results we recorded are shown in Fig. 7.

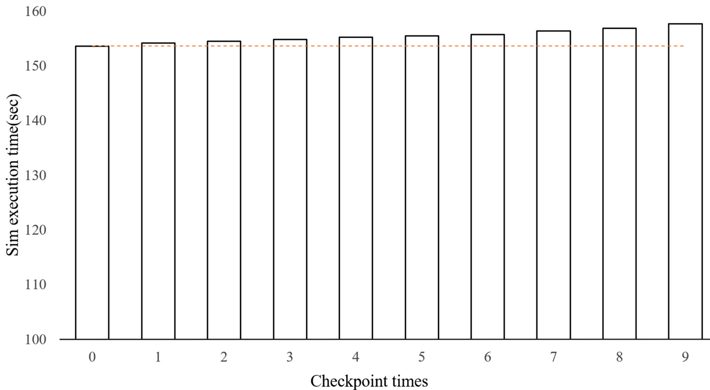

Fig. 7. The performance of the checkpoint setting on the simulation program.

The abscissa in the figure represents the number of checkpoints set on the simulation program. The abscissa is 0 means the time required to execute the simulation program itself. A single checkpoint has a limited impact on the execution time of the simulation program. With the increase in the number of checkpoints, the execution time of the simulation program increases linearly and slowly. For simulation programs that require frequent backtracking, the time overhead of setting checkpoints multiple times is tolerable.

7 Conclusion

We have described a new checkpoint idea, which can not only create process checkpoints, but also help process traceback (this is very useful for simulation programs). The system uses dynamic instrumentation tools to intercept

the native file access interface and insert a virtual file layer to unify the volatile and persistent data of the process without modifying the imitation source code. Although performance is the most important issue of the process, our experimental results on micro-benchmarks and practical applications show that the cost of introducing dynamic instrumentation and virtual file layer is acceptable, and the impact on the process itself is very limited. Our experience shows that the use of dynamic instrumentation tools to insert virtual file layer can satisfy the checkpoint setting requirements of conventional processes, and also provides a solution for process backtracking.

References

1. Wang, Y.M., Huang, Y., Vo, K.-P., Chung, P.-Y., Kintala, C.: Checkpointing and its applications. In: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing, p. 22. Institute of Electrical and Electronics Engineers, Inc., Washington, DC (1995)
2. Pei, D.: Modification operations buffering: a low overhead approach to checkpoint user files. In: Proceedings of IEEE 29th Symposium on Fault-Tolerant Computing, Madison USA, pp. 36–38 (1999)
3. Duell, J.: The design and implementation of Berkeley Lab's Linux checkpoint/restart. Berkeley Lab Technical report, LBNL-54941 (2002)
4. Duell, J., Hargrove, P., Roman, E.: Requirements for Linux checkpoint/restart. Berkeley Lab Technical report, LBNL-49659 (2002)
5. Roman, E.: A survey of checkpoint/restart implementations. Berkeley Lab Technical report, LBNL-54942 (2002)
6. Sankaran, S., et al.: The LAM/MPI checkpoint/restart framework: system-initiated checkpointing. In: LACSI Symposium, LBNL-53808 (2003)
7. Paul H., Duell, J.: Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters. In: Proceedings of SciDAC 2006, LBNL-60520 (2006)
8. Lyubashevskiy, I., Strumpen, V.: Fault-tolerant file-I/O for portable checkpointing systems. J. Supercomput. **16**, 69–92 (2000)
9. Rashid, R., et al.: Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. IEEE Trans. Comput. **37**(8), 896–908 (1998)
10. Zhong, H., Nieh, J.: CRAK: Linux checkpoint/restart as a Kernel module. Technical report CUCS-014-01, Department of Computer Science, Columbia University (2001)
11. Osman, S., Subhraveti, D., Su, G., Nieh, J.: The design and implementation of Zap: a system for migrating computing environments. In: Proceedings of the Fourth Symposium on Operating Systems Design and Implementation. ACM SIGOPS Operating Systems Review (2002). https://doi.org/10.1145/844128.844162
12. Chung, P.E., Huang, Y., Yajnik, S.: Checkpointing in CosMiC: a user-level process migration environment. In: Proceedings of the 1997 Pacific Rim International Symposium on Fault-Tolerant Systems. IEEE Computer Society (1997)
13. Weihl, W.E.: Transaction-processing techniques. In: Distributed Systems, pp. 329–352. ACM Press/Addison-Wesley Publishing, New York (1993)
14. Wang, Y.M., Chung, P.E., Huang, Y.: Integrating checkpointing with transaction processing. In: Proceedings of 27rd Fault-Tolerant Symposium, Seattle, Washington, pp. 24–27. IEEE Computer Society (1997)

15. Ouyang, J., Maheshwari, P.: Supporting cost-effective fault tolerance in distributed message-passing applications with file operations. J. Supercomput. **14**, 207–232 (1999)
16. Pei, D., Wang, D., Shen, M., Zheng, M.: Design and implementation of a low-overhead file checkpointing approach. In: Proceedings of the Fourth International Conference/Exhibition on High Performance Computing, Asia-Pacific Region, pp. 439–441 (2000)
17. Liu, S., Wang, D., Zhu, J.: A files checkpointing approach based on virtual file operations. J. Softw. **13**(8), 1528–1533 (2002)
18. Jeyakumar, A.R.: Metamori: a library for incremental file checkpointing. Master's thesis, Virgina Tech, Blacksburg (2004)
19. Qing, D., et al.: Research of component-based integrated modeling and simulation environment. J. Syst. Environ. **04**, 900–904 (2008)
20. Xue, R., Chen, W., Zheng, W.: CprFS: a user-level file system to support consistent file states for checkpoint and restart. In: Proceedings of the International Conference on Supercomputing, pp. 114–123 (2008)
21. FUSE Doc. https://www.kernel.org/doc/html/latest/filesystems/fuse.html. Accessed 28 Apr 2020
22. CRIU Homepage. https://criu.org/Main_Page. Accessed 28 Apr 2020
23. IOzone Homepage. http://www.iozone.org/. Accessed 29 Apr 2020
24. BWA Homepage. https://github.com/lh3/bwa. Accessed 29 Apr 2020
25. Pin Doc. https://software.intel.com/sites/landingpage/pintool/docs. Accessed 29 Apr 2020
26. Agrawal, H., Demillo, A.R., Spafford, H.E.: An execution-backtracking approach to debugging. IEEE Softw. **8**(3), 21–26 (1991)
27. Matthews, G., Hood, R., Johnson, S., Leggett, P.: Backtracking and re-execution in the automatic debugging of parallelized programs. In: Proceedings 11th IEEE International Symposium on High Performance Distributed Computing, Edinburgh, UK, pp. 150–160 (2002)
28. Agrawal, H., DeMillo, R.A., Spafford, E.H.: Debugging with dynamic slicing and backtracking. Softw. Pract. Exper. **23**(6), 589–616 (1993)