CprFS: A User-level File System to Support Consistent File States for Checkpoint and Restart

Ruini Xue
High Performance Computing Institution
Tsinghua University
Beijing 100084, China
xrn05@mails.tsinghua.edu.cn

Wenguang Chen
High Performance Computing Institution
Tsinghua University
Beijing 100084, China
cwg@tsinghua.edu.cn

Weimin Zheng
High Performance Computing Institution
Tsinghua University
Beijing 100084, China
zwm-dcs@tsinghua.edu.cn

ABSTRACT

Checkpoint and Restart (CPR) is becoming critical to large scale parallel computers, whose Mean Time Between Failures (MTBF) may be much shorter than the execution times of the applications. The CPR mechanism should be able to store and recover the states of virtual memory, communication and files for the applications in a consistent way.

However, many CPR tools ignore file states, which may cause errors for applications with file operations on recovery. Some CPR tools adopt library-based approaches or kernel-level file systems to deal with file states, but they only support limited types of file operations which are not sufficient for some applications. Moreover, many library-based approaches are not transparent to user applications because they wrap file APIs. Kernel-level file systems are difficult to deploy in production systems due to unnecessary overhead they may introduce to applications that do not need CPR.

In this paper we propose a user-level file system, CprFS, to address these problems. As a file system, CprFS can guarantee transparency to user applications, and is convenient to support arbitrary file operations. It can be deployed on applications' demand to avoid intervention with other applications. Experimental results show that CprFS introduces acceptable overhead and has little impact on checkpointing systems.

Categories and Subject Descriptors

D.4.3 [**OPERATING SYSTEMS**]: File Systems Management; D.4.5 [**OPERATING SYSTEMS**]: Reliability

General Terms

Design, Reliability, Experimentation

Keywords

checkpoint and restart, file checkpointing, fault tolerance, parallel computing

ICS'08, June 7–12, 2008, Island of Kos, Aegean Sea, Greece.

1. INTRODUCTION

The need for fault tolerance in large-scale parallel computers is becoming critical. Some scientific computing applications may run for days, weeks or even longer. However, because of the extraordinarily large component count of such machines — for instance, the latest IBM's BlueGene/L has 106,496 nodes (212,992 processors) — their MTBF may be much shorter than the execution times of the applications running on them. The absence of some mechanism for fault tolerance in such systems is catastrophic for the running application.

Checkpoint and Restart is a straightforward solution for providing fault tolerance: the state of the computation is saved periodically on stable storage. The saved state is called a checkpoint. When a failure is detected, the latest checkpoint is loaded and computation is restarted from the saved state.

Generally, checkpointing can be decomposed into two steps: (1) saving virtual memory (heap, stack, registers etc.) and communication states (sockets, pipes etc.), namely *memory checkpointing*; (2) handling files states, namely *file checkpointing*. To successfully restart a process, these states should be restored consistently[25]. Many CPR tools address the consistency of virtual memory[20, 29] and communication[22, 1, 9, 5], while they ignore file states. This can lead to obvious errors as illustrated in Figure 1. Checkpoint `i` was taken in line 2, and an error occurred after a write to `example` in line 7. Then the application restarted from checkpoint `i`, but `buf` read in line 3 is the new value written in the aborted run in line 7 which is not the expected data.

```
1  fd = open("example", O_RDWR);
2  checkpoint i
3  read(fd, buf, len);
4  /* use buf */
5  /* set buf to new value */
6  lseek(fd, 0, SEEK_SET);
7  write(fd, buf, len);
8  fsync(fd);
9  /* Error occurs! */
10 /* restart from checkpoint i */
```

Figure 1: Inconsistency error if data files are not handled.

Some studies have been conducted on file checkpointing with library-based approach, however they have several limitations:

- The supported file access patterns are simple[13, 19].

```
checkpointing i
fd = open("example", O_WRONLY | O_APPEND);
write(fd, buf, len);
close(fd);
/* Error occurs! */
/* restart from checkpoint i */
...
checkpoint i + 1
```

Figure 2: Inconsistency error if *non-active files* are not handled.

Some CPR tools assume that there is only one kind of non-idempotent operation[1] for each file between two consecutive checkpoints. However, in real-world applications, any file operations can be performed, and file access patterns can be quite complicated.

- Only *active files* (opened and not closed yet when checkpointing) are handled[25, 12, 17, 7]. Some tools assume the files are always opened, and just save some file information when checkpointing, but do not track the files during normal running. Therefore, the files open-and-closed between two checkpoints can not be handled properly. In Figure 2, `fd` is non-active because it is opened after checkpoint `i` and closed before checkpoint `i+1`; therefore, it is ignored by these CPR tools. This can also lead to inconsistency: `buf` will be incorrectly appended again to `example` after restarting from checkpoint `i`.

- User applications require modifications[13, 18, 19, 8]. Many tools use wrappers on standard I/O interfaces, which demand enormously invasive source changes to use the new APIs.

A file system can avoid these problems and is a better solution than library-based approach. However, though kernel-level file systems are expected of good performance, it is hard to implement a kernel module and it could limit compatibility to certain kernel versions or devices. Additionally, after being deployed, a kernel-level file system can affect applications that do not need CPR. These constraints prohibit CPR from being widely used, since file operations are common place in real-world applications.

To address these problems, we propose CprFS, *Checkpoint and Restart oriented File System*, a user-level file system for file checkpointing. It leverages the advantages of file system and user-level design with basic transactions support. The prototype is implemented under Linux currently, and it can be ported to different operating systems with little effort. Since CprFS buffers file changes locally, it needs special treatment for this scenario: in a shared storage system, one process of a parallel application wants to access the file data updated by another process in different node. Direct accessing can result in inconsistency. CprFS detects the inconsistency and can roll back the file states.

[1]Generally, an operation is idempotent if, whenever it is applied twice to any element, it gives the same result as if it were applied once. Considering file operations, idempotency means the retrieval of data and information from files and does not modify them (e.g., `read`, `stat`), while an operation is non-idempotent if it changes either contents or attributes of a file (e.g., `write`, `truncate`, `chmod`)[13].

This paper makes three main contributions.

- To the best of our knowledge, CprFS is the first attempt to address file checkpointing with a user-level file system. As a file system, CprFS is transparent to user applications and allows the separation of file checkpointing from memory checkpointing. It allows us to remount any file system, distribute and deploy it on application's demand without interfering applications that do not need file checkpointing. Meanwhile, user-level solution has good portability and maintainability while keeps the implementation easy. By establishing a file access pattern model, it supports arbitrary file operations between checkpoints. The model is not restricted to CprFS and can be added to other systems.

- As a local file system, CprFS is designed to work with different types of applications and storage systems. For parallel applications with file sharing over shared storage systems, a solution similar to transactional memory is devised to process possible file access conflicts. It bookkeeps necessary tracing information of file requests during normal running, and checks this information among all involved processes to decide whether the file states can be committed.

- As overhead is the major concern for user-level file system, we conducted extensive experiments on CprFS to evaluate its performance and impact on an MPI CPR system. Experiments on micro-benchmark show that CprFS leads to 11.62% speedup for sequential write, 7.30% slowdown for sequential read, and random accesses slow down of at most 15.60%. We also evaluate CprFS on 4 real-world applications and obtain an average **speedup** of 3.58% with short checkpointing intervals. The overhead of CprFS is acceptable and it can even accelerate those applications containing lots of small writes by aggregation.

The paper is organized as follows. Related works are discussed in Section 2, and an overview of CprFS design is then presented in Section 3. Discussion of file state transition follows in Section 4, then the support for parallel applications and file systems is described Section 5. Section 6 provides a detailed evaluation of our system. Section 7 presents limitations and directions for future work, then we conclude in Section 8.

2. RELATED WORK

Many CPR tools are implemented as libraries to address file checkpointing. Libckp[25] and Condor[12] assume all files are opened in `append` mode, and it records the file length upon opening and truncates it on recovery. All ftIO[13] file operations are implemented as wrappers around the standard file operations. The entire file is copied upon the first *write*, and subsequent file operations are performed on the replica. When checkpointing, it replaces the original file with its replica by means of the atomic `rename` operation. Libfcp[2] deploys a "inplace update with undo logs" scheme, and the file is rolled back according to the undo logs on recovery. Libra[18] combines a "copy-on-change" strategy and undo log to record the parts that are really changed in order to reduce the log size. MOB[19] and Metamori[8]

also wrap file operations, and buffer all file changes between checkpoints and commit in the next checkpoint.

These tools are implemented as static libraries, and file operations are implemented as wrappers around the standard file APIs. User applications have to be modified to use these wrappers. DejaVu[21] intercepts file operations via a shared library and rolls back all changes made to the file system since the last checkpoint on restart. User applications have to load the shared library first to do interception. Additionally, libraries share memory spaces with applications, therefore if applications crash, all information maintained by libraries will be lost: it is difficult to separate file checkpointing from memory checkpointing.

ReFS[10] is the first attempt to address file checkpointing using file system. It inserts an *address translator layer* into the Linux kernel and extends ext2 to a kernel-level versioning file system. Some kernel modules save the opened file descriptors and re-open them on restart in a way similar to libckp[4, 30, 17, 7]. Particularly, BLCR supports memory mapped files, and shared mappings are stored only once for any group of processes. They only handle *active files*. This is not sufficient to restart some applications successfully. ReviveI/O[15] is implemented as a "pseudo device driver"[14] and uses hardware to support I/O buffering, which requires changing the directory protocol.

Though kernel-level file systems and kernel modules can avoid the problems in library-based approaches, they are difficult to implement and maintain. A versioning file system is a variant case of shadow copy, and it doesn't fit well onto CPR without heavy adjustment: first, it is inconvenient for user applications to retrieve the old version transparently and second, old versions are usually read-only. The hardware solution is expensive and requires much effort to deploy.

There are two types of files in UNIX like systems: regular files and special files (such as device files). As most of the data files in scientific applications are regular files, we focus on regular files in this paper.

3. DESIGN OVERVIEW

3.1 Transactions and Work Modes

For CPR, an *atomic transaction* is considered to be the execution of a program between two consecutive checkpoints[13]. 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.

Transactions in CPR are coarse-grained in contrast to conventional transactional file systems. For file checkpointing, a fine-grained record of each individual file operation is not necessary. CprFS buffers all file operations, and the files are not modified until the next checkpoint. An `open/close` pair can not be used as the boundaries of transactions, otherwise inconsistency remains as illustrated in Figure 2: if the file changes are committed on `close` in line 4, the file `example` will be appended once again after recovery.

For parallel applications, generally there are two types of file sharing among different processes with respect to dependencies between file accesses. (1) Each process accesses a certain part of the file and there are no overlaps between accessed areas. (2) The data one process wants to access is produced by another one. Accordingly, CprFS can run in two modes: *buffer* mode and *shadow* mode. In buffer mode, CprFS captures all file operations between checkpoints, records them in logs, organizes these logs into transactions, and performs *proper* actions according to program states (commit on checkpoint, replay or abort on restart). In shadow mode, CprFS creates a copy for the file on its first non-idempotent operation, and all operations are performed on the original file. In case of error, the application replaces the changed file with the replica.

Since most scientific applications belong to the first category, this paper concentrates on buffer mode mainly, and shadow mode is explained in Section 5.

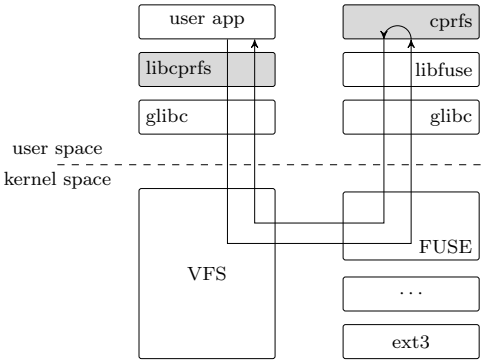


Figure 3: CprFS structure. cprfs is the run-time file system, and libcprfs is the user application library, which is usually incorporated into checkpoint libraries.

3.2 System Architecture

CprFS consists of two main components: a run-time file system and a user application library. Figure 3 shows the relation between CprFS and other components in the operating system. The run-time file system executes entirely in user space with FUSE[23], supervises all file operations and tracks file state transition. This architecture decouples memory checkpointing and file checkpointing. A user space approach, aside from providing greater flexibility and easier implementation, also avoids cumbersome interaction with the Linux VFS and page cache, both of which were designed for a different interface and workload. The library exposes the run-time file system to user applications by three transaction related APIs: `cprfs_begin_tran`, `cprfs_commit_tran` and `cprfs_abort_tran`, which are used to begin, commit and abort transactions respectively.

3.3 Name Translation and Backing Store

CprFS can remount any file system (low-level file system). The low-level file system can be either local file system, network file system or parallel file system, which stores data and provides standard file interface to user applications. From this point of view, CprFS is a plug-in for low-level file systems. It extends the functionality of low-level file systems on demand without creating a whole new file system. To avoid mis-operations, name translation in CprFS works as a map layer between user applications and files: it accepts requests from user applications and forwards them to real files in low-level file system.

The mount-point of CprFS is called the *agent data path* (*ADP*), and the corresponding low-level file system directory is the *real data path* (*RDP*). *RDP* is where data files are

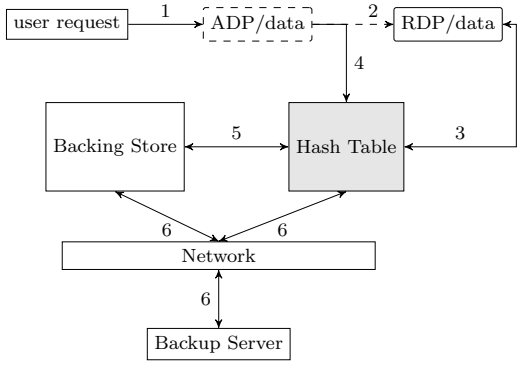


Figure 4: Name translation and the backing store. In CprFS, the hash table resides in memory, while the backing store is stored in local disk.

usually located, while *ADP* is usually an empty directory. All file requests to *ADP* are redirected to CprFS by VFS, then CprFS translates the requests to corresponding real files in *RDP*.

Central to CprFS design is a hash table. The real file name is used as the hash key and the buffered data is recorded in a double-linked list. CprFS does not use the immutable inode number as hash key, because on the one hand, the kernel module functions pass file names but not inode numbers, therefore, if the inode number is used as the key, CprFS needs one additional query to lookup the inode number according to the file name; on the other hand, for RENEWED files in Section 4.3, they have no inode numbers before being committed.

Figure 4 demonstrates how CprFS maps user requests from *ADP* to *RDP* and how CprFS manages buffered data. User applications send requests to *ADP* (1). CprFS intercepts these requests and maps them to *RDP* (2), in the meantime, CprFS loads data from *RDP* if necessary (3) and saves new contents from user applications to the hash table (4). At first, all the buffered data is stored in memory (hash table), consuming more and more memories as the buffered data increases. CprFS sets two thresholds: if the buffered data in the memory exceeds the *upper threshold*, CprFS flushes it to the backing store (5) on local disk until the amount of buffered data in memory is no more than the *lower threshold*. To tolerate permanent failures, CprFS periodically backs up the hash table and backing store to a remote server (6).

4. FILE ACCESS PATTERN MODEL

4.1 Operation Log

CprFS organizes all modifications into *operation log*. Operation log is not undo/redo log, but the delta data. Take `write` for example, CprFS captures all the new data, and writes it into the buffer cache. As a result, a sequence of write operations is combined into a single log record, which can be committed on checkpointing or replayed after recovery.

The operation log contains two types of information: the pending data (buffered data) and pending commands. Pending data refers to the newly written data from `write` operations, while pending commands refers to commands performed where no new data is produced, such as `unlink` and `truncate`. Pending data is stored in the hash table, while pending commands are represented by file states presented in Section 4.3.

4.2 File Operation Semantics

CprFS redefines 5 non-idempotent file operations, `open`, `truncate`, `unlink`, `rename`, `write` and 1 idempotent operation, `read`. `open` can be invoked with either the `O_CREAT` or `O_TRUNC` flag; CprFS treats them as `create` and `truncate` respectively. From the user application's viewpoint, the file operation interfaces remain the same, but their semantics are different from the conventions.

- **read** Get data from the pending data (either in the hash table or the backing store), if not found, read from the real file.
- **write** Always inserts new data into the hash table.
- **create** A new file is created in the hash table instead of on disk.
- **truncate** The new file size is recorded instead of truncating the file immediately.
- **unlink** The file is marked with a special flag to indicate it has been deleted instead of removing it from the disk at once.
- **rename** The `rename` operation affects both the source and destination, and changes their states simultaneously. To describe this clearly, we split `rename` into `renameSrc` and `renameDst`, standing for actions against source and destination respectively. `rename` makes the model much more intricate than had been expected. We discuss file state transition on `rename` in a dedicated section.

4.3 File States

To support arbitrary file operations between checkpoints, we have to build a file access pattern model and it is necessary to list all possible file states at first. We start with a new created blank file (an existing file or a nonexistent file), and perform the above actions on it. If the resultant state from an action can not be merged into existing file states, a new file state is created. Then all actions are performed on the new state. This procedure is repeated until no new state is produced. This greedy method guarantees the completeness of file states. This approach produces 10 file states and CprFS breaks them down into three categories (category names are uppercase, and file states are lowercase):

- **DEAD** A file is DEAD if it has been deleted or renamed to another file. That is, the file will not appear on the disk after committing.
- **ALIVE** A file is ALIVE if it has not been DEAD. By this definition, files that do not exist are ALIVE.
- **RENEWED** A file is RENEWED if it has been DEAD and then recreated or renamed from another file.

All 10 file states and their commit actions are listed in Table 1. The file state not only reflects its current status (e.g., normal and deleted), but also certain history information. For example, the "combined" state (dead, reborn) means the file has been renamed to another file at first (dead) and then recreated (reborn). All these combined states appear in the RENEWED category. As one can imagine, (deleted, reborn)

Table 1: CprFS file states and commit actions.

State	Comments	Commit Actions
ALIVE		
normal	new created or write enabled	flush operation log
truncated	truncated by `O_TRUNC` flag or `truncate`	truncate to new length, then flush operation log
DEAD		
deleted	removed by `unlink`	remove the file
dead	the source of `rename` [`renameSrc`]	Noop
RENEWED		
reborn	recreated after being deleted	truncate to 0 and flush operation log
renamed	the destination of `rename` [`renameDst`]	rename and remove the leading "dead" flag of rename source
(dead, reborn)	recreated after being renamed	Noop
(dead, renamed)	renamed as the source and then renamed as the destination	Noop
(dead, renamed, truncated)	renamed as the source, then renmaed as the destination, then truncated	Noop
(renamed, truncated)	renamed as the destination and then truncated	rename, truncate, flush operation log, and remove the leading "dead" flag of rename source

is a possible file state, which means the file is removed at first and then recreated. reborn also indicates the recreation of a DEAD file. Thus, (deleted, reborn) can be merged into reborn, and is not listed in the table.

On committing, CprFS traverses the hash table and flushes all files bound to the process being checkpointed according to their states and commit actions. Table 1 shows that some file states can be committed immediately in one pass (e.g., normal and deleted), while those that start with dead (e.g., (dead, reborn) and (dead, renamed)), require more than one pass to commit. This is because the dead flag in the file state means that the file has been the source of a rename operation, thus it is the duty of the rename destination to remove its rename source before the rename source commits itself. The "Noop" entries in the "Commit Actions" column correspond to the first pass, they do nothing but waiting for their rename destinations to remove the leading dead flags. By "remove the leading dead flag" we mean if the file state is a combined state starting with dead, just remove the leading dead flag from the state; if the file state is a single dead, change it to deleted. Then the file changes its state to the remaining flag only, and performs the actions corresponding to the new state in the next pass.

4.4 CprFS State Machine

CprFS state machine does not differentiate between active and non-active files, because all of them should be handled on checkpointing. In this section we first introduce a subset of the state machine for common cases. Then, we present `rename` related file state transitions.

4.4.1 Basic State Transition

Figure 5 presents the transition diagram for CprFS state machine with the premise that `rename` can only perform on normal files. To clarify the diagram, operations are represented by different types of arrows. These state transitions are easy to follow, and they account for common cases in real world applications.

CprFS records a file in the hash table on its first request. If the requested file does not exist, CprFS inserts an entry into the hash table and sets its state accordingly. `create` is the only valid operation that can be performed on files of DEAD category, which changes deleted to reborn, and dead

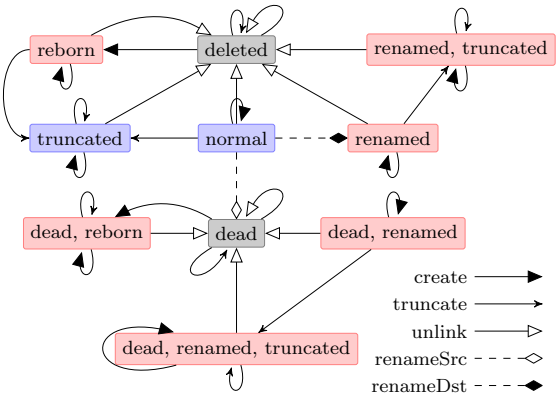


Figure 5: CprFS basic state machine.

to (dead, reborn). `create` on all other states has no effect because it does nothing to existing files.

`truncate` does not result in file state transition if the file state is neither normal, reborn, renamed nor (dead, renamed). If the file has been the source of `rename`, `unlink` changes its state to dead, otherwise it is changed to deleted. In Figure 5, `renameSrc` and `renameDst` only work on normal files, and lead to states of dead and renamed respectively.

4.4.2 `rename` Related State Transition

The `rename` operation affects both the source and destination, and changes their states simultaneously. Furthermore, source and destination files can be in any of the 10 file states, so, there are one hundred possible combinations. Though most of these can be grouped together, it is difficult to draw them directly in a state transition diagram. To improve the readability, we list them as in Table 2.

Table 2 has five columns. The first two columns denote the states of the source and destination files before renaming, column 3 and column 4 are the result states after renaming, and the last column is the line number in each group for reference.

All combinations are split into 4 groups. The 10 states of the source file are distributed over the first column, while the 10 states of the destination are shown in the second

Table 2: `rename` related file states transition. "—" means no state transition.

Before		After		Line #
source	destination	source	destination	
Group 1				
normal, truncated	normal, truncated, deleted, reborn	dead	renamed	1
(dead, renamed), (dead, renamed, truncated)	renamed, (renamed, truncated)		renamed*	2
	dead, (dead, reborn)		(dead, renamed)	3
renamed	(dead, renamed), (dead, renamed, truncated)	deleted	(dead, renamed)*	4
Group 2				
reborn	normal, truncated, deleted, reborn	deleted	renamed	1
	dead, (dead, reborn)		(dead, renamed)	2
(dead, reborn)	renamed, (renamed, truncated), (dead, renamed), (dead, renamed, truncated)	dead	reborn*	3
Group 3				
(renamed, truncated)	normal, truncated, deleted, reborn	deleted	renamed	1
	dead, (dead, reborn)		(dead, renamed)	2
	renamed, (renamed, truncated)		renamed*	3
	(dead, renamed), (dead, renamed, truncated)		(dead, renamed, truncated)*	4
Group 4				
dead, deleted	—	—	—	1

Table 3: An example for a file renamed twice.

Action	a	b	c
1. BEGIN	normal	normal	normal
2. `rename(a, b)`	dead	renamed	normal
3. `rename(c, b)`	deleted	renamed*	dead

column in each group. Therefore, in each group, each entry in column 1 can match any entry in column 2 and their result states are the entries in column 3 and 4 in the same line as the entry in column 2. Entries in column 1 from one group can not match any entry in column 2 of the other three groups. For example, assume that before renaming, the state of the source is normal and the state of the destination is dead. By looking up the table, we find that the source is in Group 1 (column 1, line 1), the corresponding dead state in the same group is in (column 2, line 3), so the result states are the entries in column 3 and 4 in Group 1 line 3: dead and (dead, renamed) for the source and destination respectively.

Some items in the table are marked with an asterisk (*) to denote the destination has to remove the leading dead flag in its old rename source's state. For example, the result destination state is renamed* in line 2 of Group 1. We find that the state of the destination before renaming contains renamed, which indicates it was renamed from a file before this `rename` operation. Now, it is going to be renamed from another file. That means its rename source will be changed, so it has to adjust its old rename source's file state. Otherwise, no one would remove its old source, because its old source must have been marked as dead, and is waiting for the destination to remove it on committing (see Table 1). Table 3 is one example of these scenarios. Files `a`, `b` and `c` are normal at the beginning. In Step 2, `a` is renamed to `b`. `a` changes to dead, `b` changes to renamed, and `b` sets its rename source to `a`. `c` has no state transition. In Step 3, `c` is renamed to `b`; thus `c` changes to dead, and `b` updates its rename source from `a` to `c`. Before doing this, `b` has to remove the leading dead flag for its original rename source, `a`. Otherwise, `a` will not be deleted in commit.

DEAD files can not be rename sources, because they do not exist. Short lines (—) are used to denote impossible states in Group 4, and no state transition happens.

Figure 5 and Table 2 make up of the complete file access pattern model. There is really only one state machine, although there are two represented here to clarify when a transition occurs. Using this model, CprFS is able to support arbitrary file operations between two consecutive checkpoints. Another benefit of this model is that it is independent of the low-level file system and the checkpoint library and can be added to other systems.

5. SUPPORT PARALLEL APPLICATIONS AND PARALLEL FILE SYSTEMS

Since CprFS buffers pending data locally, special treatment is required for parallel applications with file sharing: before committing, the process on one node can not read the new data updated by the processes on other nodes, which would lead to inconsistency. For example, file `foo` is shared in a cluster through a parallel file system; P_0 and P_1 are two processes on $Node_0$ and $Node_1$ respectively. If P_0 changes `foo`, the modification is buffered on $Node_0$ by CprFS, thus P_1 can only read the old data from `foo` but not the new data written by P_0.

A straightforward solution is extending CprFS to a network or parallel file system, thus CprFS can buffer the pending data globally, in which way, all processes across different nodes can share the same copy of the pending data. However, this method has two serious drawbacks. First, implementing CprFS as a network or parallel file system is non-trivial. The complexity of the protocol, distributed lock, and implementation issues can counteract the benefits of a light weight user-level file system. Second, the performance drops dramatically and may become unacceptable. To maintain a single copy of the pending data, client cache is impossible, so all file operations are transferred to network requests. Moreover, the node that buffers pending data is very likely to be a bottleneck.

Due to these difficulties, a local file system is preferred to a global solution. CprFS borrows the idea from transactional memory, and regards this situation as a conflict where: two

or more processes from different nodes access the same part of the same file, and at least one of them is write.

In most scientific applications, different processes access different parts of the data files, and they do not conflict according to the above definition. Our solution is to release the constraints, allow conflicts to occur, and test them before committing. The algorithm is described as following:

1. During normal running, CprFS collects necessary information for conflict test:
   (a) Every file is accompanied with a special file recording information of the processes (hostname and pid) requesting the file. These processes may come from different nodes;
   (b) CprFS maintains an *operation list* for each file, recording which parts of the file are read and written. The list items are coalesced if possible to reduce the item number;
2. Before committing, CprFS tests conflicts:
   (a) All processes dump the operation lists in the same location as their related files;
   (b) For each file recorded in the hash table, CprFS tests the operation lists of different processes from different nodes. The processes accessing this file are recognized from its accompanied special file. If conflicts are found, CprFS writes the result to an agreed global file and returns. Otherwise, goes to Step 2c;
   (c) Check the agreed global file for conflicts from other files. This is because different processes can access different files, and conflicts in any file can abort the transaction.

If no conflict is found, CprFS commits the pending data. Otherwise, it aborts the transaction and the application should be restarted with CprFS running in shadow mode. Though designed for parallel file systems, this algorithm also works well with local file system. For local file system, the special file only records the processes in the same node, and they must have not shared data with processes from other nodes, so the test would pass as expected. Besides, it is easy to understand that this algorithm also works for sequential applications.

6. EXPERIMENTAL RESULTS

In this section, we evaluate CprFS both against micro-benchmarks and real-world applications to demonstrate its performance and impact on an MPI CPR system.

6.1 Methodology

The experiments were conducted on a cluster of 8 nodes. Each node was equipped with dual Intel Itanium2 1.3GHz CPUs, 4GB RAM and a 36GB Ultra320 SCSI disk. The operating system used was Redhat Linux AS3 with kernel 2.4.21-20.EL.ia64. The file system for local disk was ext3. All machines share a storage device via NFS and are connected through a switched 1Gbps Ethernet LAN. We developed a coordinated checkpointing system for MPI applications based on MPICH-1.2.7p1[6] and Thckpt[27]. In our tests, all input and output files are stored in the NFS, and CprFS is mounted over NFS on each node.

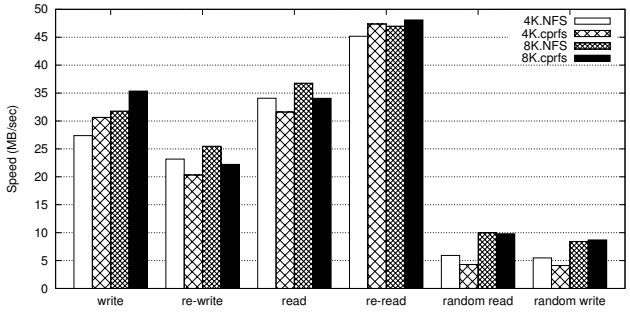


Figure 6: IOzone performance: NFS vs. CprFS.

6.2 Micro-benchmarks

6.2.1 Raw Performance

We used IOzone[16] to evaluate the performance characteristics of CprFS. IOzone calls `sync` after the `write` and `re-write` phases to flush file system buffers, while CprFS does nothing with `sync`. To make a reasonable comparison, we inserted CprFS commit actions after `write` and `re-write` phases into IOzone, otherwise, CprFS would keep the data in memory during the following tests. Wang[24] pointed out that in many scientific applications, small requests account for more than 90% of all requests, so we ran IOzone with record sizes of 4 and 8KB. Figure 6 presents the performance of the NFS and CprFS over NFS. 4K.NFS and 4K.cprfs denote the results of NFS and CprFS for record size of 4KB, while 8K.NFS and 8K.cprfs are for 8KB record size.

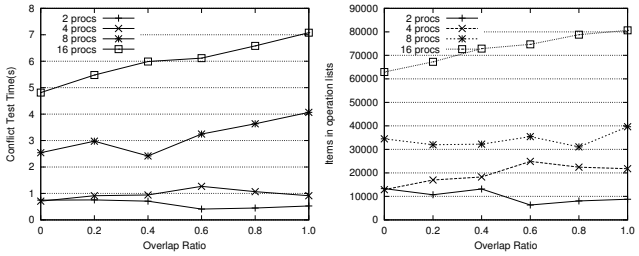
There is about a 11.62% speedup rather than slowdown for sequential `write` in CprFS. This is because CprFS buffers data in memory, and aggregates all record-sized writes into a single large write on committing. `Re-write` incurs about 12.56% overhead, because CprFS has to read in the old data before overwriting it.

In CprFS, sequential `read` is 7.30% slower than NFS. This is because CprFS has to retrieve data from the real file and send it to IOzone, which requires one more context switch. `re-read` is used to test the performance if buffer cache is available with the native file system. CprFS delivers about a 3.60% speedup for `re-read` because all data is in memory after `read`, and CprFS simply feeds them back. This means that CprFS does not diminish the benefit of buffer cache.

`Random read` and `random write` are 15.60% and 11.87% slower respectively in CprFS. Random accesses act in the same way as their sequential counterparts with additional seek operations. `Random write` also needs to read in old data before writing new data.

6.2.2 Conflict Test Overhead

Section 5 explains that conflicts can arise when parallel applications access the same file. This section presents the conflict test overhead in CprFS. Since there are no well-known benchmarks and applications for this scenario, we designed a synthetic MPI application in which conflicts happen. All processes access one 2GB file simultaneously, and the file is separated equally according to the process number, each process performs 10^4 reads and writes in total randomly on its part. By default, there are no conflicts, because no process accesses data outside its boundary. We then set different *overlap ratios*, meaning that the data segment for each pro-



(a) Test overhead (b) Items number

Figure 7: Conflict test performance in CprFS.

Table 4: Geometric mean of overhead in different modes. Negative values indicate speedup instead of slowdown. The last row is the geometric mean overhead for all applications.

Application	cprfs w/o ckpt	cprfs w/ ckpt
BTIO	-11.56%	-3.28%
PAPSM	-15.73%	-14.64%
ClustalW-MPI	1.22%	1.78%
mpiBlast	1.10%	2.86%
Geometric Mean	-6.55%	-3.58%

cess is extended to the left and right, so adjacent processes can both access more of each other's data and the *overlap ratio* increases.

Figure 7a presents the conflict test time in CprFS for different process numbers with different overlap ratios. It is clear that as the number of processes and overlap ratio increase, the overhead increases. This is because increasing processes means one process has to check against more and more other processes, and bigger overlap ratios mean each process produces more items that can not be coalesced in *operation lists*. Figure 7b shows the total number of items in all processes; the curves are similar to those in Figure 7a.

6.3 Real-world Applications

Four real-world applications (including one from NPB 3.2.1) were selected to evaluate CprFS: (1) NPB BTIO[26] is used to test the output capabilities of high performance computing systems, especially parallel systems, (2) ClustalW-MPI[11] is a tool for aligning multiple protein or nucleotide sequences, (3) mpiBLAST[3] is a parallel version of BLAST, and (4) PAPSM[28] is a parallel power system simulator.

All applications are run with different process numbers in three modes: normal running over the native file system without checkpointing, normal running over CprFS without checkpointing committing pending data on program termination, and running over CprFS with periodic checkpointing. These three modes are called "native run", "cprfs w/o ckpt", and "cprfs w/ ckpt" in the following figures. Figure 8 shows the total running times for different applications in different modes, and Table 4 presents the geometric mean of overheads on average.

6.3.1 Running without Checkpointing

We first evaluated the applications in "native run" and "cprfs w/o ckpt" modes to investigate the performance influence of CprFS.

For BTIO, we used subtype `full` — the implementation of the benchmark that takes advantage of the collective I/O operations in the MPI-IO standard. We report results for Class A (for the sake of disk quota) for 4, 9 and 16 processes. BTIO is I/O intensive and outputs a total of 400MB to a single file.

Figure 8a depicts the total running times for BTIO in the different modes. When BTIO runs over CprFS without checkpointing, the speedup is about 11.56%. The benefits come from the better write performance of CprFS.

Each PAPSM process writes several floats after regular time steps, and the output files are about 163MB in all, which leads to about 20 million small write operations. CprFS is inherently suitable to accelerate PAPSM because of its write aggregation. This is illustrated in Figure 8b, which shows that CprFS delivers 12.80%, 18.12% and 16.16% increases in speed when running PAPSM on 2, 4 and 8 processors respectively.

ClustalW-MPI and mpiBlast are computing intensive, and their output files are only 1.01 MB and 67MB respectively. Their total running times are nearly the same across the different modes as illustrated in Figure 8c and Figure 8d. The average overhead for ClustalW-MPI on "cprfs w/o ckpt" is about 1.22%, and for mpiBlast, it is about 1.10%.

Since sequential accesses are most commonly used in scientific applications, CprFS does not hurt their performance because it can accelerate sequential writes due to operation aggregation, and incurs acceptable overhead for sequential read.

6.3.2 Checkpointing Overhead

In our tests, a coordinated checkpoint is taken in four steps: 1) synchronize all processes, 2) dump the memory image to a checkpoint file on the local disk, 3) CprFS commits pending data, including conflict testing, and 4) synchronize all processes again. The checkpoint files are written to the local disk and transferred to shared storage in the background by a dedicated thread. Therefore, we did not include the time for transferring checkpoint files in the checkpoint overhead. Thus, the checkpoint overhead consists of three parts: total synchronization time in Steps 1 and 4, the time to write the checkpoint file to local disk in Step 2 and the duration of CprFS committing pending data in Step 3.[2] Periodic checkpoints were taken every 2 minutes except for BTIO, for which only one checkpoint was taken because its total running time is less than 2 minutes.

As mentioned above, BTIO and PAPSM consist of many small writes and benefit a great deal from CprFS. Figure 8a and Figure 8b show that when running over CprFS with checkpointing BTIO and PAPSM exhibit 3.28% and 14.64% speedup compared to running over the native file system. In contrast, the performance decrement is about 1.78% for ClustalW-MPI, and about 2.86% for mpiBlast as shown in Figure 8c and Figure 8d.

Checkpoint overhead is small with respect to the total running time and leads to acceptable performance deterioration. Figure 9 shows the contribution breakdowns of the different parts during one checkpoint for each application. These figures indicate that "ckpt time" and "sync time" dominate checkpointing overhead. Synchronization time for the

[2]All these applications perform sequential reads and writes, and there are no file access conflicts in them. The conflict test overhead is negligible, therefore, we include it in commit time.

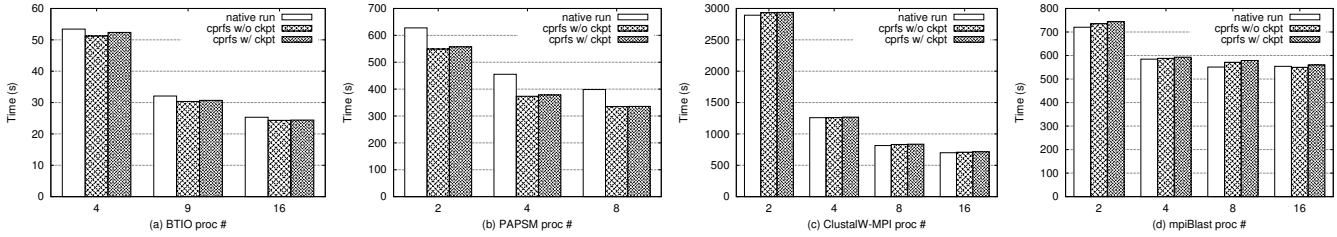


Figure 8: Running times for BTIO, PAPSM, ClustalW-MPI and mpiBlast in different modes.

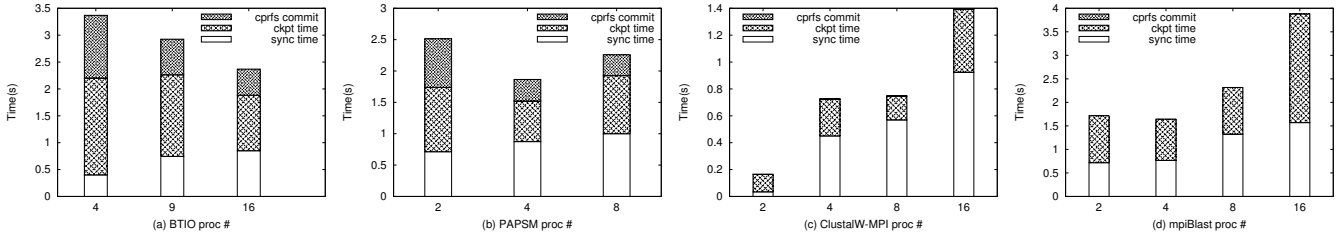


Figure 9: Checkpoint overhead breakdown for BTIO, PAPSM, ClustalW-MPI and mpiBlast. sync time is the synchronization time, ckpt time is the time to write checkpoint file, and cprfs commit is the time for CprFS to commit pending data.

Table 5: Checkpoint file sizes. "–" means the application is not run with the corresponding process number.

Proc #	2	4	8	9	16
BTIO	–	31.97M	–	16.35M	11.44M
PAPSM	27.14M	14.12M	14.62M	–	–
ClustalW-MPI	27.59M	14.05M	14.04M	–	15.13M
mpiBlast	66.15M	65.10M	51.40M	–	70.35M

Table 6: File sizes committed by CprFS. "–" means the application is not run with the corresponding process number.

Proc #	2	4	8	9	16
BTIO	–	57.67M	–	27.40M	14.64M
PAPSM	17.92M	7.65M	3.24M	–	–
ClustalW-MPI	38.82K	38.80K	19.85K	–	58.19K
mpiBlast	582B	580B	582B	–	600B

same number of processors varies little in all cases, while "ckpt time" and "cprfs commit" time are proportional to data sizes (see Table 5 and Table 6).

For ClustalW-MPI and mpiBlast, most of the output data is not written until the application finished, so the pending data sizes are nearly 0 on each checkpoint as shown in Figure 9c and Figure 9d as well as in Table 6. Since CprFS committing is independent of the checkpointing protocol, CprFS does not affect the scalability of coordinated checkpointing.

7. LIMITATIONS AND FUTURE WORK

Our prototype can support many scientific applications. However, there are a still some issues we plan to address in the future.

First of all, CprFS builds a *file* access pattern model, which does not provide perfect support for complex *directory* operations. Simulating a directory in the hash table can complicate the model a great deal. A more elegant scheme is required.

Secondly, Our current implementation of CprFS does not support memory mapped files. This is because all file operations are translated into memory operations for `mmap()`'d files, and it is impossible for CprFS to intercept these operations via standard file IO APIs. This issue will be addressed in the future.

Thirdly, CprFS is independent of low-level file systems, which leads to good portability and deployability. However, some useful functions provided by the low-level file system might be lost in this process (e.g., client cache, file locking). We plan to integrate CprFS more tightly with some well-known parallel file systems, such as PVFS and Lustre.

8. CONCLUSION

We have described the design and implementation of the CprFS system that helps guarantee the consistency between the application and its files during checkpoint and restart. By exposing file system interfaces, CprFS can provide transparency to user applications. User-level implementations get rid of the restrictions of kernel programming and exploit the flexibility of user space programming, and also have good portability, deployability and maintainability. Though performance is the major concern for user-level file systems, our experimental results on both micro-benchmarks and real-world applications show that CprFS introduces acceptable overhead and has little impact on checkpointing systems. Experiences with CprFS demonstrate that: user-level file system is an effective and efficient solution to approach file checkpointing for high performance computing.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for useful feedback. The research was partially supported by Chinese National

973 Basic Research Program under Grant 2007CB310900 and by Tsinghua National Laboratory for Information Science and Technology.

10. REFERENCES

[1] A. Bouteiller, F. Cappello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette. MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging. In *SC'03*, pages 25–41, Washington, DC, USA, 2003. IEEE Computer Society.

[2] P. E. Chung, Y. Huang, S. Yajnik, G. Fowler, K. P. Vo, and Y. M. Wang. Checkpointing in CosMic a User-level Process Migration Environment. In *Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 187–193, Dec. 1997.

[3] A. E. Darling, L. Carey, and W. chun Feng. The Design, Implementation, and Evaluation of mpiBlast, `http://www.mpiblast.org`, June 11, 2003.

[4] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. white paper, Future Technologies Group, 2003.

[5] Q. Gao, W. Yu, W. Huang, and D. K. Panda. Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand. In *ICPP'06*, pages 471–478. IEEE Computer Society, 2006.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[7] G. J. Janakiraman, J. R. Santos, D. Subhraveti, and Y. Turner. Cruz: Application-transparent distributed checkpoint-restart on standard operating systems. In *DSN'05*, pages 260–269, Yokohama, Japan, 28 June – 1 July 2005.

[8] A. R. Jeyakumar. Metamori: A library for Incremental File Checkpointing. Master's thesis, Virgina Tech, Blacksburg, June 21 2004.

[9] H. Jung, D. Shin, H. Han, J. W. Kim, H. Y. Yeom, and J. Lee. Design and implementation of multiple fault-tolerant MPI over myrinet (M^3). In *SC'2005*, Seattle, Washington, USA, Nov. 2005.

[10] H. Kim and H. Yeom. A User-Transparent Recoverable File System for Distributed Computing Environment. In *CLADE 2005*, pages 45–53, July 2005.

[11] K.-B. Li. ClustalW-MPI: ClustalW analysis using distributed and parallel computing. *Bioinformatics*, 19(12):1585–1586, 2003.

[12] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report CS-TR-1997-1346, University of Wisconsin, Madison, Apr. 1997.

[13] I. Lyubashevskiy and V. Strumpen. Fault-tolerant file-I/O for portable checkpointing systems. *The Journal of Supercomputing*, 16(1-2):69–92, 2000.

[14] Y. Masubuchi, S. Hoshina, T. Shimada, H. Hirayama, and N. Kato. Fault Recovery Mechanism for Multiprocessor Servers. In *FTCS'97*, pages 184–193, 1997.

[15] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. ReViveI/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers. In *HPCA'06*, pages 200–211, Austin, Texas, USA, Feb.11–15 2006.

[16] W. D. Norcott and D. Capps. IOzone Filesystem Benchmark, `http://www.iozone.org/`, 2006.

[17] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI'02)*, Dec. 2002.

[18] J. Ouyang and P. Maheshwari. Supporting Cost-Effective Fault Tolerance in Distributed Message-Passing Applications with File Operations. *The Journal of Supercomputing*, 14(3):207–232, 1999.

[19] D. Pei. Modification Operations Buffering: A Lowoverhead Approach to Checkpoint User Files. In *IEEE 29th Symposium on Fault-Tolerant Computing*, pages 36–38, Madison, USA, June 1999.

[20] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under UNIX. In *Proceedings of the USENIX Technical Conference on UNIX and Advanced Computing Systems*, pages 213–224, Berkeley, CA, USA, Jan. 1995.

[21] J. F. Ruscio, M. A. Heffner, and S. Varadarajan. DejaVu: Transparent User-Level Checkpointing, Migration, and Recovery for Distributed Systems. In *IPDPS'07*, pages 1–10. IEEE, 2007.

[22] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *IPPS'96*, pages 526–531, Honolulu, Hawaii, Oct. 02 1996.

[23] M. Szeredi. File System in User Space, `http://fuse.sourceforge.net`, 2006.

[24] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. E. Long, and T. T. McLarty. File system workload analysis for large scale scientific computing applications. In *MSST'04*, College Park, MD, Apr. 2004. IEEE Computer Society Press.

[25] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. M. R. Kintala. Checkpointing and its applications. In *FTCS'95*, pages 22–31, 1995.

[26] P. Wong and R. F. V. der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. Technical Report NAS-03-002, Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division, NASA Ames Research Center, Moffett Field, CA 94035-1000, Jan. 2003.

[27] R. N. Xue, Y. H. Zhang, W. G. Chen, and W. M. Zheng. Thckpt: Transparent Checkpointing of UNIX Processes under IA64. In H. R. Arabnia, editor, *PDPTA'05*, volume 1, pages 325–332, Las Vegas, Nevada, USA, June27–30 2005. CSREA Press.

[28] W. Xue, J. Shu, Y. Wu, and W. Zheng. Parallel Algorithm and Implementation for Realtime Dynamic Simulation of Power System. In *ICPP'2005*, pages 137–144, Oslo, Norway, June 2005. IEEE Computer Society.

[29] V. C. Zandy. ckpt – process checkpoint library, `http://pages.cs.wisc.edu/~zandy/ckpt/`, 2004.

[30] H. Zhong and J. Nieh. CRAK: Linux Checkpoint/Restart As a Kernel Module. Technical Report CUCS-014-01, Department of Computer Science, Columbia University, Nov. 2001.