Metamori: A library for Incremental File Checkpointing

Ashwin Raju Jeyakumar
Department of Computer Science
Virginia Tech, Blacksburg, VA 24061

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
Computer Science and Applications

Examining Committee:

Srinidhi Varadarajan, Chair
Naren Ramakrishnan,
Calvin J. Ribbens

June 3, 2004
Blacksburg, Virginia

Keywords: Checkpointing, Rollback-recovery, File Checkpointing, Fault-tolerance

Metamori: A library for Incremental File Checkpointing

Ashwin Raju Jeyakumar

Abstract

The advent of cluster computing has resulted in a thrust towards providing software mechanisms for reliability on clusters. The prevalent model for such mechanisms is to take a snapshot of the state of an application, called a *checkpoint* and commit it to stable storage. This checkpoint has sufficient meta-data, so that if the application fails, it can be restarted from the checkpoint. This operation is called a *restore*. In order to record a process' complete state, both its volatile and persistent state must be checkpointed. Several libraries exist for checkpointing volatile state. Some of these libraries feature incremental checkpointing, where only the changes since the last checkpoint are recorded in the next checkpoint. Such incremental checkpointing is advantageous since otherwise, the time taken for each successive checkpoint becomes larger and larger. Also, when checkpointing is done in increments, we can restore state to any of the previous checkpoints; a vital feature for adaptive applications. This thesis presents a user-level incremental checkpointing library for files: Metamori. This brings the advantages of incremental memory checkpointing to files as well, thereby providing a low-overhead approach to checkpoint persistent state. Thus, the complete state of an application can now be incrementally checkpointed, as compared to earlier approaches where volatile state was checkpointed incrementally but persistent state had no such facilities.

Acknowledgements

I would like to thank my committee: Dr. Srinidhi Varadarajan, Dr. Calvin J. Ribbens and Dr. Naren Ramakrishnan. Dr. Varadarajan: Thank you for introducing me to this field and providing me with the resources for me to complete my thesis. It has been my pleasure and privilege to work with you, and I appreciate the insights that I have gained along the way on High Performance Computing. I am also indebted to you for reviewing this document and providing valuable feedback. Dr. Ribbens: Thank you for proof-reading this document and for being a great teacher. Dr. Ramakrishnan: Your support has always been a source of encouragement for me.

I would like to thank Bharath Ramesh for being my 'sysadmin' throughout my time here at Virginia Tech. Your love for Linux is infectious, and I owe it to you for learning a thing or two about computing systems. Thanks are also due to Muthukumar Thirunavukkarasu; I have often relied on your objective analysis to solve many problems. Your constant support and encouragement, coupled with our many conversations about life has kept me going all this while. Gautam Swaminathan, Sumithra Bhakthavatsalam, Sandeep Prabhakhar, Padmapriya Kandhadai, Prachi Bora, Beatriz-Diaz Acosta and Malarvizhi Chinnusamy: For being great friends and companions. "Coffee, anyone?" and "GSPAM" would have been very boring without you guys. Gautam and Malar: Thank you for providing feedback on my defense slides.

This thesis would not have been possible without the love and support of my parents: R. Jeyakumar and Rajeswari Raju. Dad: Your optimism is what saw me through this thesis. Mom: Thanks for being a great listener.

And finally, Divya Rangarajan: Thank you for teaching me to go with the flow. "You will graduate!" was all I needed every day to do my best.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The chief motivation for this library is in the context of *cluster computing*. As explained in [4], cluster computing is ”the practice of networking together commodity off-the-shelf hardware components to create a system that works together to solve large problems or a large number of small problems that would take an unreasonable amount of time on a single computing resource”. Each node in a cluster is an independent computing resource, with its own memory, processing power and storage. These nodes are then tied together using a communication fabric and software to function as a single machine.

The most compelling reason to use clusters instead of traditional supercomputing equipment is the price/performance that these systems offer. It has been widely noted that clusters running an operating system such as Linux on Intel or AMD processors record anywhere from 2x to 10x better price/performance than RISC based SMP machines running UNIX. While it is obvious that the price of off-the-shelf components must be less than custom-built niche machines, the comparable performance that these machines achieve when put together must be explained. The answer is that these clusters take advantage of the increased speed of communication that is afforded by today's communication fabrics.

Clusters, however, are not without their flaws. Since the basic unit of processing of a cluster is an everyday machine geared towards desktop use, cluster nodes have higher failure rates than the custom-built machines that were used earlier. As a result, a combinatorial increase in failure is realized with the growth of a cluster, forcing the people who work on such machines to come up with ways and means to ensure that jobs complete succesfully in the face of intermittent failures of cluster components. Thus, it was envisaged that software could be put on a cluster that would prevent the complete loss of a job. The goal was a system where a failed job could be restarted from some previous point in its execution, called a *checkpoint*. Once the checkpoint is taken, all the work done by the program is assumed to be committed to stable storage and the job can then be restarted from that point either on the same machine or on a different machine. Many such systems have since been conceived.

1.1 Problem Statement:

The checkpointing space can be broadly divided into three sub-problems, based on the three components of program behavior: *volatile state*, *persistent state* and *OS environment*. This thesis focuses on persistent state. While many optimizations have been proposed for volatile state checkpointing, developments in persistent state checkpointing have not kept pace. In this thesis, we present a library for checkpointing files. We differ from previous approaches because we use the idea of *incremental checkpointing*, storing only the changes made to a file since its last checkpoint at the next checkpoint. We also provide the ability for the application to restore its file state to any one of the previously taken checkpoints. We do not checkpoint just active files (i.e., files that have been opened and not yet closed by a process), but all files that have been touched by the process.

1.2 Readers Guide:

Chapter 2 presents some background to the problem of checkpointing. In Chapter 3 we present a survey of related work in the field of checkpointing, highlighting contributions that are relevant to our problem. Chapter 4 discusses the design and implementation of Metamori, the library that is the software artifact of this thesis. Chapter 5 presents the tests that were used to evaluate this library and benchmarks to measure performance of the library. Chapter 6 discusses the work done in this thesis and also outlines directions for future research in this field.

Chapter 2

Background

Checkpointing itself is merely one way of solving a larger problem known as *Rollback and Recovery*. Earlier, high performance computing (HPC) used niche machines which were custom-built for the task they were to execute. This restriction of the problem domain caused these machines to be very reliable. However, recent decrease in the price of hardware and improvements in the field of computer networks have led to the increased use of *COTS (Commercial Off The Shelf)* components to build high performance computing machines. This new breed of machines has been named *clusters*. Clusters suffer from one major drawback though: hardware reliability.

Consider that the average COTS computer has a reliability of x. Often x is what appears to be a very high percentage, perhaps 99.9%. This is more than acceptable to the normal PC consumer, who has no issue with having to reboot his machine, say once a year. The problem arises however, when one examines precisely how x behaves with regards to the probability of any node in a cluster failing. From basic probability, the probability *P* of any node in a cluster failing given a group of *p* nodes is given by:

$P = p(1 - x) = p(1 - 0.999) = p(0.001)$

It is easy to see that as p increases, the probability of node failure on a given day increases. Indeed, once p reaches 1000, a not uncommon number of nodes for large clusters, it is almost guaranteed that a component will fail on a daily basis. Any parallel job that was utilizing the failed component would then have to be restarted. As many of the HPC jobs that supercomputers are utilized for can take days if not weeks to complete, it would most likely require several failed attempts before a job manages to complete, if at all. This would appear to render cluster supercomputers infeasible for truly large scale super-computing.

The stability of large clusters is obviously an important issue in the ability of long-running parallel codes to execute through completion. As there is no way to limit the failure of COTS components, focus must instead lie on tolerating these itinerant failures. The most widely accepted group of methods are variations of Rollback and Recovery; [11] is an excellent survey of these methods. Abstractly, Rollback and Recovery refers to the act of tolerating a failure by reverting the system to some valid state that existed prior to the failure and restarting the execution from that point. The manner in which that state is saved and recovered is what distinguishes the different methods.

There are two dimensions to classifying Rollback and Recovery systems. The first is the level

of abstraction at which the state of the processes is saved. In *system-level* Rollback and Recovery, checkpoints save the state of the entire system on stable storage. This is a large amount of data and is therefore unsuitable for larger systems, such as those in high performance computing. The other end of the spectrum is *application-level* Rollback and Recovery. Here the application programmer decides what needs to be saved to stable storage and when. The obvious disadvantage to this is that it complicates application code, though it can potentially result in much less data being moved to stable storage.

The second dimension to classifying Rollback and Recovery techniques is as *transparent* and *non-transparent*. Transparent techniques do not require any intervention on the part of the application or programmer. The system automatically takes checkpoints according to some specified policy, and recovers automatically from failures if they occur. This is the preferred approach for most applications in high performance computing since it allows programmers to write applications without regard to fault tolerance.

Transparent Checkpointing can be divided into two broad categories: *Checkpoint-based* rollback-recovery and *Log-based* rollback-recovery. The chief difference between the two is that the former relies only on checkpoints to provide fault-tolerance while the latter combines checkpointing with logging of non-deterministic events. The relevance of log-based rollback-recovery in high performance computing is low. This is because most applications in this area do not have sufficient frequency of non-deterministic events to necessitate log-based rollback-recovery protocols.

Our approach is an application-level semi-transparent checkpoint-based rollback recovery technique. It is semi-transparent because, while the application is not aware of the checkpointing work that is going on during normal operations, the application nevertheless needs to call the checkpoint and restore functions.

Chapter 3

Literature Survey

Several packages have attempted file checkpointing, some in user space and some in kernel space. A few of these are presented below:

3.1 BLCR:

The Berkeley Lab's Linux Checkpoint/Restart project does not provide any kind of file checkpointing. However, it intends to and [10] discusses a number of possible approaches to the problem. At checkpoint time, just the filename and its current offset can be saved for a reload. A checksum can also be added to this which can be verified at restart time. If the checksum fails the restart fails. A third alternative is to truncate the file to a size recorded at checkpoint time. This provides a primitive rollback method for files that are written to a continuous stream. More complete alternatives exist in the form of making a full copy of the file at checkpoint time. A common scenario in UNIX processes is that a file is opened by an application and then deleted from the file system, remaining visible only to the application and existing only while the application keeps its handle open. This behavior can be mimicked in a restore scenario.

3.2 Condor:

Condor [13] was the first framework to recognize the advantages of checkpointing with a view to migrating processes. The Condor package is a hunter of free workstations, and uses these workstations to execute tasks which are waiting to be scheduled. Checkpointing is therefore used to halt a job on one machine and migrate it across the network to some other machine. The package provides the distinguishing feature of preserving a large portion of the originating machine's execution environment, even if the originating and execution machines do not share a common file system. To achieve this end, Condor's checkpointing mechanisms are implemented completely in user space. This ensures that so long as the basic C libraries common across all flavors of UNIX are available, the checkpointing mechanisms can occur. Of course, the price paid for this flexibility is seen in the speed and completeness of process migration.

The Condor process migration mechanism is not for all UNIX processes. A typical flaw is when inter-process communication is involved. If a process is communication with other processes at the time of checkpoint, and if those other processes are no longer existent at the time of restore then that part of the process' state cannot be restored. However, the fraction of general UNIX processes that do inter-process communication is small, and this disability is therefore ignored.

Condor provides checkpoint capabilities for memory, processor registers and active files. Memory consists of the text, data and stack segments. Of these, restoring the text segments is simple since it exists unmodified from the original executable. In the interest of portability, the approaches to restoring data and stack segments have changed with time. The current implementation has removed dependencies on specific implementations and has resorted to basic UNIX mechanisms that have been ported by the provider of the kernel and the C library.

When a checkpoint is taken, a core dump of the process is created. This follows from the realization that the data needed to debug a process is almost the same as the data needed to restart it. Hence the data and stack segments are generated from the core file. The developers avoided the usage of machine-dependent routines to restore volatile state such as the program counter and register contents. Instead, they used signal-handling routines and the C library's setjmp/longjmp facility.

Condor circumvents the user job's main() call, and replaces it with its own MAIN routine. This routine establishes a handle for the SIGTSTP signal, and uses this signal to inform a job to checkpoint itself. Condor also circumvents the open system call and uses its version of the open call to keep track of all open files that the process currently has. The only information it records for the call is the seek pointer for the file. It makes the assumption that all file-related system calls are idempotent, something which is not true in practice.

3.3 Libckpt:

Libckpt [15] is a checkpointing library implemented in user-space. It provides semi-transparent checkpointing. This means that linking an application with the Libckpt library alone is not sufficient to start checkpointing; changes need to be made to the application code as well. Libckpt requires that the initial C procedure in his code be changes from main() to ckpt_target(). This effectively gives control of the program to Libckpt to perform required initialization operations. Libckpt implemented incremental checkpointing for volatile state. It uses page-protection hardware to identify the unchanged portion of the checkpoint. It then saves only the changed portions, thus greatly reducing the overhead of checkpointing. It also implements another optimization called main-memory checkpointing. The idea is to make a copy of the program's data space and use an asynchronously executing thread of control to write the checkpoint file. Since checkpointing is now interleaved with normal process execution, this cuts down checkpoint overhead.

Libckpt also makes forays into user-directed checkpointing. Two techniques are presented: *Memory Exclusion* and *Synchronous Checkpointing*. Memory exclusion is in-part implemented automatically by the way Libckpt does incremental checkpointing. It uses the *mprotect* system call and traps segmentation faults to detect pages being modified. This approach has the following disadvantages: it has page-granularity, system calls can fail rather than generate a *SEGV* signal

when asked to write to a protected page, and on some systems mprotect() is not reliable. Hence the programmer is also given a set of calls to let him exclude memory he deems unnecessary. Two kinds of memory locations are of interest: dead locations and clean locations. A dead location is one whose value in memory will never be read or written, and thus does not need to be saved. A clean location is one whose value in memory exists in a previous checkpoint and has not been changed. Synchronous checkpointing capability is provided to the users in Libckpt. This enables users with knowledge of their applications to decide where to checkpoint. Libckpt is distinguished as the first library to provide synchronous checkpointing capabilities.

While Libckpt's memory checkpointing capabilities are tremendous, it provides very primitive support for file checkpointing. It merely saves the state of the open file table at each checkpoint. It has been shown in [23] that this approach is insufficient for correctly recording the state of files affected by a process.

3.4 CRAK:

CRAK [24] implements the checkpoint/restart facility as a Linux kernel module. It is transparent to the application. It supports parallel processes and migration of networked applications on UNIX/Linux environments. Unlike Condor, there is no concept of stub processes or home nodes. Processes only depend on the nodes they are currently running on. It assumes that the operating system it runs on supports loadable kernel modules and that such kernel modules can access the private data of a process. Other assumptions made are that the environment is homogeneous (all machines have the same architecture, OS and CRAK installed) and that they access the same files across all machines (either by a global file system such as NFS or Coda or installed locally). Checkpointing and restarting in CRAK is viewed more as a tool for process migration than for rollback recovery.

With regard to files, CRAK again deals with only active files. It provides mechanisms to deal with named files and socket descriptors. The checkpoint process is simplistic. This is because it is not geared toward rollback recovery but toward process migration. Therefore, failure conditions and backing up of data are not dealt with very seriously. While it can detect whether a file that was checkpointed has been deleted or modified since the checkpoint, it cannot restore it to its original state. In the presence of a global file system like NFS, just the path name is saved; otherwise the entire content of the file is saved instead of just the path name. On restart, it is necessary to ensure that the files are opened with exactly the same file descriptors as before. This is done using the dup2 system call and is demonstrated in the code fragment shown in Figure 3.1.

Thus if reopening the file does not yield the desired descriptor, the dup2 system call is used to ensure consistency. For the problem of migrating sockets, the designers have discounted the use of a stub process on the home node to relay information, since if the home node fails, this method will not work. Instead, CRAK leverages its position as a kernel module to directly alter the socket information of both the migrating process and the counterpart it is communicating with. It cannot stop with just modifying the transport layer's data structures however, but must also alter the routing table cache. With regard to checkpointing pipes, the system proposes a working solution if all the pipes belong to a set of processes that is being restarted together. It accomplishes this

```
int open_force (int fd, char * filename, int flags, int mode) {
int ret = open(filename, flags, mode); // open the file
if (ret < 0 || ret == fd) return ret; // if error or already the desired descriptor
just return
if (dup2(ret, fd) < 0) return -1; // dup the descriptor, note this should not fail
close(ret); // close the original descriptor
return fd;
}
```

Figure 3.1: Descriptor re-opening in CRAK.

by maintaining a cache of pipes that restarting processes can scan and connect to. This approach requires explicit knowledge of when a group of parallel processes has begun or finished. Also, while it can restore the pipe itself, it cannot refill the data in the pipe that existed at checkpoint time.

3.5 Libckp:

Libckp [23] was the first comprehensive file checkpointing library. While designing Libckp, the authors broke down the checkpointing problem into three sub-problems based on the three elements of program behavior: Volatile state consists of the program stack and the static and dynamic data segments, Persistent state includes all the user files that are related to the current program execution, OS Environment refers to the resources that the user processes must access through the operating system. It further defines two terms: Process state refers to everything that is included in a checkpoint and process environment refers to everything that is not included in a checkpoint but can affect program behavior. The chief difference between the two is that process environment is not recoverable. The designers of Libckp asserted that persistent state can be included either in process state or in the process environment, depending on the application.

Libckp makes the best case for persistent state checkpointing. It argues that while incorrect rollbacks in memory checkpointing will lead to obvious process failures which will be detected very quickly, file corruption as a result of rollback seldom yields errors during execution. As a result, the job will continue to run with corrupt data files, and will continue to store results to these corrupt files. The result would be that the run would complete with results that cannot be deciphered, as most high performance computing jobs write their results to files. However, the active file checkpointing technique envisaged by Libckpt is inadequate.

The approach taken by Libckp is to model the volatile state and persistent state as a multi-process system, and file operations as inter-process communications. This makes the consistency problem a checkpoint coordination problem. This problem is then solved by using dependency tracking for file operations and lazy checkpoint coordination. Lazy coordination, as the name suggests, does not force all processes to take checkpoints for file operations when an initiating process takes a checkpoint. It instead waits till the state inconsistency between the set of processes cannot be masked due to a message dependency. For the purpose of persistent state checkpointing,

each user file is considered as a separate process and the main process as the checkpoint initiator. Therefore, for user files that are not active at checkpoint time, it suffices to record the file size when the file becomes active again and make a shadow copy of the file when a portion that existed prior to checkpoint time is modified. If a file is unlinked, a shadow copy of the file is generated. Then, if there is a subsequent failure, the shadow copy and the recorded size can be used to restore the file to have both correct size and contents.

Another important concept introduced by the designers of Libckp was that of *environment diversity*. They argue that a restarted program, though working with the same volatile and persistent state, and the same data can still behave differently if the OS environment is different. They advocate the use of the program construct in Figure 3.2 to protect applications against environmental problems.

```
retry_count = 0;
while ((ptr = malloc(size)) == NULL) {
        retry_count = retry_count + 1;
        if (retry_count == MAX_RETRY_COUNT) {
           if (chkpnt() <= 0) {
              print malloc error message;
              exit;
           } else retry_count = 0;   /*  recovery */
        }
        sleep(RETRY_WAIT_PERIOD);
}
Use ptr;
```

Figure 3.2: Program Construct to hide environmental problems.

Essentially, the malloc call is tried several times until it succeeds. However, if it fails, a checkpoint is taken before the program actually fails. If the failure is not transient, process migration can be used to restart the program on another node.

3.6 MOB:

Modified Operations Buffering [14] is a novel technique to provide low overhead file checkpointing. This technique was used in Libcsm, the underlying checkpointing library for the ChaRM system [22]. MOB was specifically designed to address inconsistent rollbacks produced by the simplistic checkpointing of file offsets for active files. The designers of MOB present several scenarios where this approach fails.

The first scenario discussed is *Rollback After Real time Event (RARE)*. Figure 3.3 shows a code block that illustrates this scenario. In the code block, *example* is not an active file until after

checkpoint i. Therefore the size of the file is not recorded in checkpoint *i*. This will result in the data being incorrectly appended twice.

```
checkpoint i
fd = open ( "example",O_WRONLY|O_APPEND);
write (  fd, write_buf, BLOCKSIZE);
/*failure occurs, rollback */
checkpoint i+1 /*should be here */
```

Figure 3.3: RARE event.

The second scenario discussed is *Rollback After Reading and Writing the same area (RARW)*. This is shown by the code block in Figure 3.4.Here we see that a portion of the file is modified before failure. Since active file approaches just keep track of file size and pay no attention to file content, the data that is read after a rollback will not be the same.As a modification of this scenario, if the file was unlinked prior to failure, the read will return erroneously. This is called *Rollback After Deletion (RAD)*.

```
fd = open( "example",  O_RDWR);
checkpoint  i
read  (  fd,  read_buf,  BLOCKSIZE);
lseek (  fd,  0,  SEEK_SET);
write (  fd,  write_buf,  BLOCKSIZE);
/* failure occurs, rollback */
checkpoint i+1 /* should be here */
```

Figure 3.4: RARW event.

MOB's underlying principle is to make all operations between two checkpoints atomic. To this end, the effects of these operations are buffered until the next checkpoint occurs. Then all the changes are committed to disk and execution continues. This makes failure recovery quite simple, since all the effects of the buffered operations are simply deleted and execution continues normally from the checkpoint.

Every file opened by the process is allocated a an entry in a structure known as the *File Information Table (FIT)*. Each FIT entry has the contents shown in Figure 3.5.

As can be seen, the FIT entry is responsible for buffering all operations that take place between checkpoints. The actual data that is written to the file is buffered in a linked list. The structure of the linked list's nodes is shown in Figure 3.6.

FileName	Fd	AccessMode	FilePointer	FileSize
Duplist	WasDeleted	WasClosed	AMLHead	
MemBufferStart	MemBufferPointer	MemBufferSize	MemBufferFull	
DiskBufferFd	DiskBufferPointer	... ...		

Figure 3.5: File Information Table.

Start	End	MemOrDisk	BufferStart	BufferEnd	Next

Figure 3.6: Address Mapping List.

This list is called the *Address Mapping List (AML)*. An entry in the AML corresponds to a file area that is continuous in the buffer. The purpose of the AML is to record the offsets in the original file where the data must be written when flushed. Two buffers are available to every opened file: a memory buffer and a disk buffer. Once the memory buffer is filled, the disk buffer is used. No caching is employed.

However, no form of descriptor reloading is mentioned. Thus, MOB makes the assumption that on restore, files will always use the same descriptors they were working with prior to failure.

3.7 Discussion:

Of the surveyed approaches, the only implementations of file checkpointing that provide the advantages that Modified Operations Buffering proposes are Libckp and Libfcp.

Libckp uses a shadow copy approach: it makes a copy of any portion of the file that is going to be modified. It also records the file size when the file becomes active. During RARE rollback, this recorded file size is used to truncate the file to its original length. During RARW and RAD, the shadow copies are used to reconstruct the original contents.

Libfcp, which is a later avatar of Libckpt, uses undo logs to the same effect. When a file is opened for modifications, its size is recorded and an undo log of file truncation is generated. When a portion of the file that existed before checkpoint time is modified, an undo log is created to restore pre-checkpoint data. For a rollback, these undo logs need to be applied in a reversed order. Therefore, a normal write operation in Libfcp leads to two additional disk accesses: reading out content from the original file and writing it into an undo log. Also, for smaller modifications, the undo logs cannot be created in memory since data in physical memory is usually not accessible when failures occur.

Libfcp and MOB are better approaches than Libckp in terms of space overhead. It is often the case that the portion modified between two checkpoints is much smaller than the entire file. Thus making a shadow copy of the entire file at checkpoint time is usually unnecessary. In HPC, files are usually very large (of the order of tens of gigabytes), mostly read-only with data appended to

their end as a result of calculations. Shadow copies are simply infeasible for this kind of data. Moreover, MOB is much more scalable than Libfcp as well, since if the same portion is modified more than once, Libfcp will require more than one undo log, while MOB will just overwrite the data in its already existing buffer.

MOB also presents the lowest recovery overhead. Two things need to be done for recovery: free the memory buffer and clear the disk buffer. Libckp does provide less overhead, since all that needs to be done is rename the shadow copy, but considering the overhead of creating the shadow copy, it is not suitable for HPC. Applying undo logs in reversed order in the manner of Libfcp takes much more time.

Chapter 4

Design

This chapter presents the data structures and functions that comprise the Metamori library. Several drawbacks exist in the way MOB [14] handles file checkpointing. Our library addresses many of these deficiencies aside from providing incremental checkpointing. We will first show where our library resides with reference to the operating system and the application. We will then elaborate on the basic data structures that we work with, followed by the functions that operate on them, and will provide justification as and when needed. The main features of this library are the following:

1. Support for both file descriptors and streams
2. Descriptor safety across checkpoints
3. An LRU cache for the buffered data
4. Incremental checkpointing capability

At this point, incremental checkpointing must be explained. Single checkpointing and incremental checkpointing are commonly used terms for checkpointing volatile state; [17] gives a good definition of both these terms. In the context of persistent state checkpointing, a single checkpoint saves all the changes to a file since the beginning of a process up to the time of the checkpoint. Such checkpoints therefore work in a cumulative fashion, summing up all the changes made. A system that takes such checkpoints is capable of restoring only to the last such checkpoint taken. An incremental checkpoint, in contrast, is one where only the changes since the last checkpoint are saved. This is obviously much less overhead than a cumulative checkpoint. Also, we can restore the state to any of the earlier checkpoints taken, not just the last one. The chief application of this added functionality is adaptive applications: applications that fail, then restore to an earlier point and resume execution in such a manner that the failure does not occur again. If these applications were to work with a single checkpoint, they may or may not be able to adapt depending on whether the checkpoint was taken before or after the point where they want to alter their execution. Hence incremental checkpointing is a valuable feature for such applications.

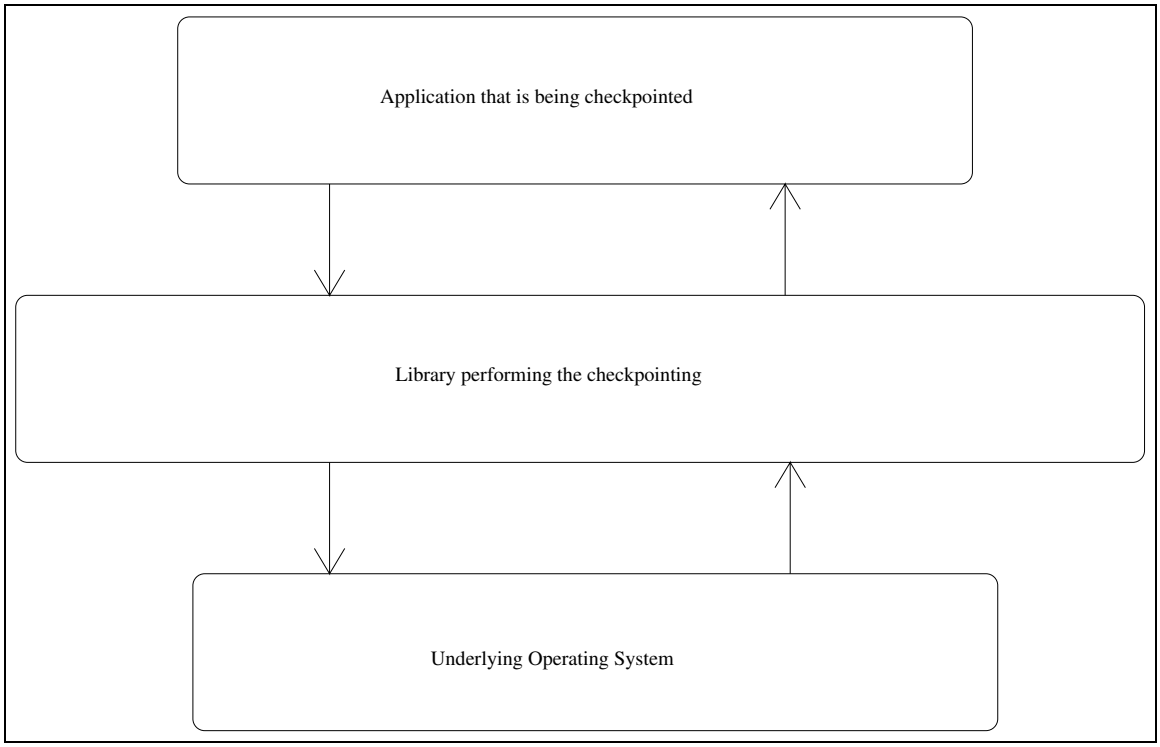

Figure 4.1: The library intercepts calls to the underlying operating system.

4.1 Architecture:

Metamori intercepts the input/output calls made by an application before they reach the underlying operating system. It then performs all the bookkeeping work that is needed for checkpointing and restoring state before it calls the underlying system call or library function. In some cases, the underlying call is not made at all, since the library itself can handle the request. Figure 4.1 shows the architecture that we have just described.

Naturally, such functionality cannot be achieved unless we overload the input/output library of the underlying operating system, which in this case is Linux. We have already mentioned that we support both handle-based and stream-based calls. The following are the handle-based calls that we overload:

1. close
2. creat
3. dup
4. dup2
5. lseek

6. open
7. read
8. write

The list of overloaded stream-based calls is given below:

1. clearerr
2. fclose
3. fdopen
4. feof
5. ferror
6. fgetc
7. fgetpos
8. fgets
9. fopen
10. fprintf
11. fputc
12. fputs
13. fread
14. freopen
15. fseek
16. fsetpos
17. ftell
18. fwrite
19. getc
20. putc
21. rewind
22. ungetc

Finally, the *unlink* call is also overloaded, but it uses a pathname as an argument and therefore does not fall in either of the above categories.

4.2 Data Structures:

The FIT used in [14] serves as the library's substitute for actual file table entries. Support was provided for file descriptors only; it was assumed that streams would use the underlying file descriptor. This is not usually the case though, since the writers of the streams can change the name they use to refer to the system calls. In that case, the system calls will never be called. The only alternative is to overload the stream based calls directly and this is what we do.

Also, this design of the FIT is very primitive, in that both the properties of the file descriptor and the physical file are encompassed in the same data structure. This poses problems when multiple descriptors are opened on the same file. For example, the default operation performed when a close is effected is to set the FIT's wasClosed flag. That would disallow further reads and writes via that descriptor. At checkpoint time, all such closed FITs have their buffers committed, then unlinked. However, if multiple descriptors reference the same file, a race condition will result, since they all have the same disk buffer. This means that one descriptor unlinking the disk buffer can cause trouble for other descriptors referencing the same physical file. This problem does not occur in traditional UNIX file systems since the descriptor's information is maintained separately from the actual file's information [5].

The separation that the UNIX filesystem achieves is a cleaner approach, hence we have redesigned our FIT to hold only information pertinent to a file descriptor or stream. The information regarding the physical file that the descriptor or stream references is abstracted into another construct, the *Physical File (PIF)*.

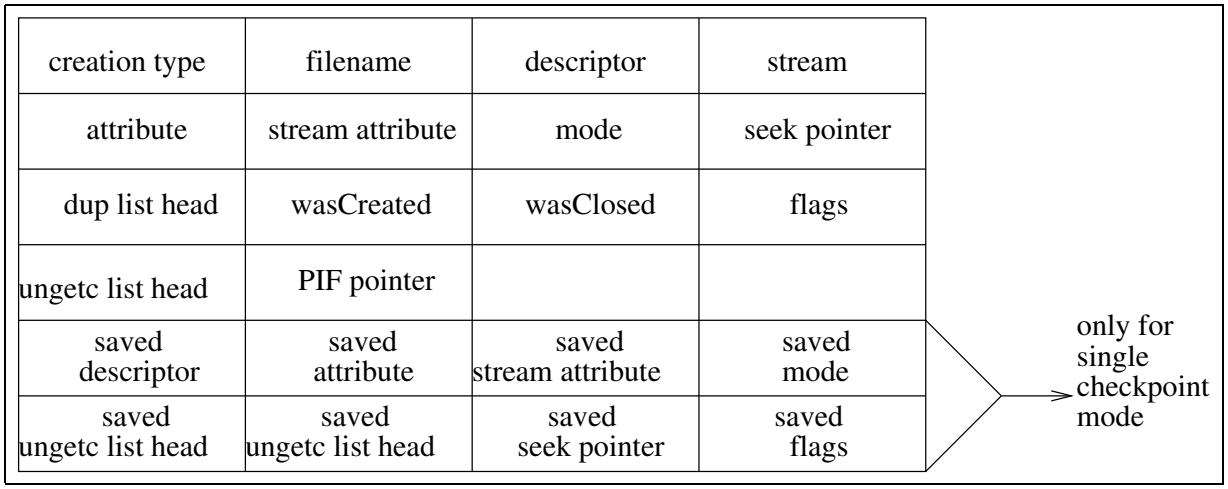

Figure 4.2: Modified File Information Table.

The new FIT structure is presented in Figure 4.2. The Disk Buffer is no longer part of the FIT, instead we have a pointer to a PIF entry. The only other major changes to the FIT are that the list of duped descriptors is included in the FIT itself and support is given for file pointers as well as descriptors. The *creation type* flag is used to distinguish the FIT as one holding a descriptor or a stream pointer, depending on which we use either the *attribute* or *stream attribute* entry. *mode* is also used only for descriptors, as is *dup list head* since only descriptors are subject to the dup system call. The *ungetc list head* entry is used only for streams. Also for single checkpoints, most of the data to be stored is held in memory itself since this library is meant to be used in conjunction with a memory checkpointing library; writing to files would be wasteful for such a bounded case. The FITs are placed in a linked list with a global head. So any file access goes through this list of FITs. These FITs are accessed by their descriptors or stream pointer values.

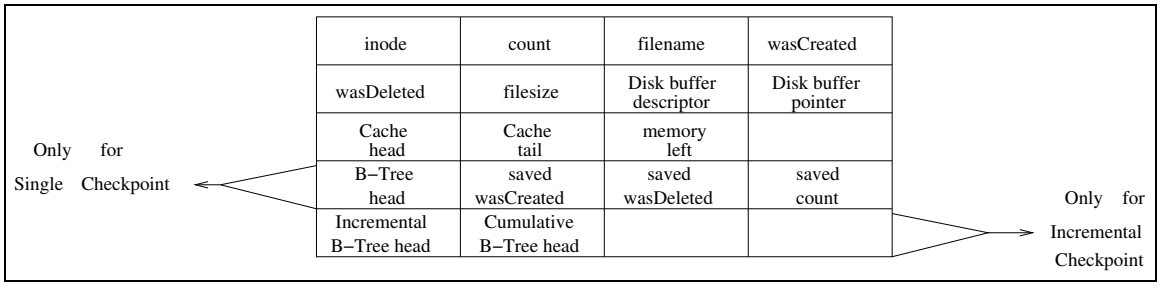

Figure 4.3: Physical File Structure.

The PIF structure stores data pertinent to the physical file. Its contents are shown in Figure 4.3. This structure holds the buffers for the file, both in memory and on disk. Since creation and deletion affects the physical file itself, the PIF also has entries for these events. These values are boolean in the single checkpoint case, but in the incremental checkpoint scenario they refer to the checkpoint interval when the file was created or deleted. Like the FITs, the PIFs are also chained together in a linked list. The PIFs are indexed by the inode number of the underlying file.

The AML structure (defined in section 3.6) is largely unchanged, save for the addition of a *dirty* bit for caching purposes. In [14] a linked list was used to save the AML structures. For scalability reasons, we have replaced this with a B Tree with mechanisms to merge contiguous nodes. What has changed is that we now have an LRU cache, as opposed to the old scheme of filling up the available memory space, then writing all subsequent buffers to disk. This will help applications with more localized access, as it will be in-memory.

Descriptor safety is provided by an extra layer of indirection between the file descriptors or stream pointers and their FITs. We maintain two arrays, one of integers (for descriptors) and another of stream pointers. When a file is opened, we place the value of the actual descriptor in one of the unused entries in this array and return the index of that entry to the application. Hence the application does not work with the actual descriptors. What this means is that we can close and transparently reopen the file in the underlying library as many times as we want so long as we update the entry in the array. This helps us during a restore, since we can close all the open files and reopen only the necessary ones, without having to worry that the application's descriptors at

the restore point will be invalid. The only alternative to this approach is the use of dup2 as in [24]. Using dup2 in this way is not a very elegant approach: consider the scenario where we have two files with their descriptors interchanged from what we want them to be, i.e., the first file has the descriptor the second wants and the second has the descriptor the first wants. Since dup2 results in both the descriptors pointing to the same entry, we will have to close and reopen one of the files. Also no equivalent of dup2 exists for streams, so indirection is our only course of action.

4.3 Functions:

4.3.1 File management:

The following functions deal with the management of FITs and PIFs:

1. file_insert
2. createPhysicalFile
3. getFile
4. findPhysicalFile
5. get_fit
6. get_libcall_fit
7. get_actual_fit
8. get_actual_libcall_fit

file_insert is called whenever a descriptor or stream pointer is opened. This function creates a new FIT and adds it to the list of FITs maintained by the library. If a descriptor is being duplicated via the *dup* system call, the dup'ed descriptor is entered into the FIT for the original descriptor; no new FIT is created. It also checks to see if a PIF has been created for that file. If a PIF already exists, it makes its PIF entry point to it and increments the PIF's reference count. If not, it calls *createPhysicalFile* to create a new PIF. The *createPhysicalFile* function initializes the B-Trees for the file, that will hold the single and incremental checkpoint information. It also initializes the cache. The rest of the functions are accessors; *getFile* and *findphyscialFile* retrieve a PIF entry indexed by inode number and file name respectively. *get_fit* and *get_libcall_fit* retrieve the FIT entry corresponding to a descriptor or stream pointer. It must be noted that these routines take as arguments the descriptor or stream pointer that was handed back to the application; if we want to bypass the indirection we use *get_actual_fit* and *get_actual_libcall_fit*.

4.3.2 Caching:

The following functions form the caching subsystem:

1. convertMemAMLToDisk
2. convertDiskAMLToMemory
3. swapAMLsToDisk
4. findAMLBoundary
5. deleteAMLBoundary
6. freeAMLBoundary
7. insertNewYoungest
8. makeNewYoungest

The cache is maintained as a circular doubly linked list, with the youngest element at the tail of the list and the oldest at its head. Each element of the list is called an AMLBoundary element. *convertMemAMLToDisk* converts an in-memory AML to an on-disk AML. The in-memory buffer is written to disk and the disk offsets are initialized in the AML. It also updates the amount of memory available to the file buffer in light of this conversion, and removes this AML from the cache list. *convertDiskAMLToMemory* moves an on-disk AML into memory, and essentially performs the reverse of the operations done by *convertDiskAMLToMemory*. *swampAMLsToDisk* takes as its argument the number of bytes to be freed up. It then traverses the AMLBoundary list, swapping the oldest AMLs to disk until there is at least as much space in the memory buffers as has been requested. *findAMLBoundary* finds the AMLBoundary element corresponding to a particular AML. deleteAMLBoundary removes an AMLBoundary element from the cache list. *freeAMLBoundary* traverses and frees the cache list for a particular PIF entry. Finally, *makeNewYoungest* moves an older AMLBoundary node to the tail of the cache, if it exists, otherwise it creates a new youngest node in the cache using *insertNewYoungest*.

4.3.3 B-Tree management:

The following are the library functions that belong in this category:

1. binTreeInsert
2. binTreeSearch
3. BTreeSearch
4. commitBinTree

5. deleteBinTree
6. writeBinTreeToFile
7. BTreeInsert
8. BTreeInsertSameCkpt

binTreeInsert is a wrapper function to create a new AML. The created AML is then inserted into only one or both the B-Trees depending on whether the checkpointing mode is single or incremental. This function does not decide whether the created AML should be in-memory or on-disk. This decision is made by its calling routine. Also, this function has the precondition that the AML it creates does not overlap with others. This management is taken care of in the calling routine, as well. *binTreeSearch* returns the first AML with key value greater than or equal to the value to search for. *commitBinTree* is utilized only in the single checkpoint case. This routine traverses the B-Tree's records in sequential fashion and commits them to the original file. This function is therefore called at checkpoint time to commit buffered data. The reason it is never used in incremental checkpointing is because we never commit data to the original file, until a restore is done. *deleteBinTree* traverses the B-Tree's records, and free's memory corresponding to in-memory AMLs. It then closes the B-Tree. In the incremental checkpoint case it does this for the newest B-Tree in the linked list of such B-Trees held by a PIF entry. *writeBinTreeToFile* is used to create what is known as the *offset file* in the incremental checkpointing case. This file stores a set of tuples, the structure of each is given by figure 4.4. This file is then used to commit data for a restore, since it gives us the starting and ending positions of data in the disk buffer and where this data belongs in the original file.

Filename format: <inode number>.offset.<Checkpoint index>.<Process id>

Start offset in original file	End offset in original file	Start offset in disk buffer	End offset in disk buffer

Figure 4.4: Offset file format.

BTreeInsert is the routine that inserts data into the incremental checkpoint tree for a PIF entry. The manner of insertion is such that data written in later checkpoints overrides data written in earlier checkpoints. This facilitates the incremental read since all that needs to be done when reading a fragment of data that has been written to across checkpoints is to traverse this B-Tree, retrieve the data from the buffers, and access the original file for data whenever it is not in the B-Tree. Take the following simple example: the same file had offsets 1 to 20 written in the first checkpoint interval, 3 to 11 in the second checkpoint interval and 16 to 25 in the third checkpoint interval. Now, if we read offsets 1 to 30 after the third checkpoint, we would want to read 1 to

2 from the first checkpoint interval, 3 to 11 from the second checkpoint, 12 to 15 from the first checkpoint interval, 16 to 25 from the third checkpoint interval and 26 to 30 from the original file. This ordering is maintained by the *BTreeInsert* routine. When we are inserting an AML from the current checkpoint interval into this B-Tree, five main scenarios arise, dependent on the start and end offsets of the AML being inserted and the start and end offsets of the AMLs already in the B-Tree:

1. The AML being inserted does not overlap with any existing AML
2. The AML being inserted has a start offset that lies before an existing AML, but its end offset lies within the existing AML
3. The AML being inserted has a start offset that lies before an existing AML, but its end offset lies beyond the existing AML
4. The AML being inserted has start and end offsets that lie within an existing AML
5. The AML being inserted has a start offset that lies within an existing AML and an end offset that lies beyond the existing AML

We use the end offset as a key for our B-Tree and insert a pointer to the AML as the value. When we insert values into the B-Tree, we must remember that the start and end offsets need not be the same as those for the actual AMLs. They can rather be subranges of the range that the original AML has. For example, an AML that extends from 1 to 50 might have two entries in the B-Tree pointing to it, one from 1 to 25 and another from 30 to 50, depending on the way other AMLs override it in later checkpoints. Here is the pseudo code for the *BTreeInsert* routine.

```
BTreeInsert(BTree TreeToInsertIn, AML AMLToInsert)
  FoundAML=The first AML node in sequence s.t.
    FoundAML.End >= AMLToInsert.Start
  /* Scenario 1 */
  if AMLToInsert lies before FoundAML completely i.e. no
    overlap
    Create a new key-value pair for AMLToInsert and insert it
    return
  Since there is an overlap, delete FoundAML from the B-Tree.
  /* We now tackle Scenarios 2 and 3.*/
  if AMLToInsert.End lies before or is equal to FoundAML.End
    /* Scenario 3 */
    if AMLToInsert.Start lies beyond FoundAML.Start
      Insert FoundAML back into the B-Tree, but extending from
        FoundAML.Start
      to AMLToInsert.Start-1
    /* Common to scenarios 2 and 3 */
    Insert AMLToInsert into the B-Tree, extending from
```

```
    AMLToInsert.Start to AMLToInsert.End
  if FoundAML extends beyond AMLToInsert
    Insert FoundAML into the B-Tree extending from
      AMLToInsert.End + 1 to FoundAML.End
  return
/* Now we tackle scenarios 4 and 5 */
if AMLToInsert.Start lies before FoundAML.Start
  /* Scenario 5 */
  Insert FoundAML into the B-Tree extending from
    FoundAML.Start to AMLToInsert.Start - 1
/*Common to Scenarios 4 and 5 */
Delete all B-Tree nodes which are encompassed by
  AMLToInsert
Make FoundAML point to the first B-Tree node to
  not be deleted
if FoundAML is not NULL
  This has two sub-cases:
    if AMLToInsert.End lies before FoundAML.Start
      Insert AMLToInsert into the B-Tree, extending
        from AMLToInsert.Start to AMLToInsert.End
      return
    /* AMLToInsert overlaps with FoundAML */
    Delete FoundAML from the B-Tree
    if AMLToInsert.Start lies beyond FoundAML.Start
      Insert FoundAML into the B-Tree, but extending from
        FoundAML.Start to AMLToInsert.Start -1
    Insert AMLToInsert into the B-Tree, extending from
      AMLToInsert.Start to AMLToInsert.End
    if FoundAML extends beyond AMLToInsert
      Insert FoundAML into the B-Tree, but extending from
        AMLToInsert.End + 1 to FoundAML.End
  return
/* FoundAML is NULL */
Insert AMLToInsert into the B-Tree, extending from
  AMLToInsert.Start to AMLToInsert.End
return
```

The *BTreeInsertSameCkpt* routine inserts AMLs into the B-Tree for that checkpoint. The algorithm is similar to *BTreeInsert*, the only difference being that deleted AMLs have their buffer resources freed up since they will not be needed anymore. We cannot do this in *BtreeInsert* since, in an incremental checkpoint scenario, if an AML in a later checkpoint completely overrides an AML in an earlier checkpoint, we will still need the data corresponding to the older AML in the event of a restore.

4.3.4 Reading and Writing

The reading and writing of actual file content is handled by the following functions:

1. LRU_write_to_amls
2. LRU_read_from_amls
3. LRU_libcall_write_to_amls
4. LRU_libcall_read_from_amls
5. createNewAML
6. LRU_underlying_read_from_amls
7. LRU_underlying_write_to_amls
8. insert_into_ungetc_list
9. undo_ungetc_list

LRU_write_to_amls and *LRU_libcall_write_to_amls* are wrapper functions that check erroneous conditions, such as writing to a read-only file. Additionally, *LRU_libcall_write_to_amls* returns the number of *items* written, rather than the number of bytes. This is necessary for the *fwrite* system call, which returns the number of items written rather than the number of bytes. The read functions work similarly. *createNewAML*, as the name indicates, creates a new AML. It checks up if the available memory buffer is sufficient to create an in-memory AML, otherwise, it swaps out as many AMLs as required to free up memory space. If the AML is too large to fit in the memory allocated for a PIF, it creates the AML on disk. This situation usually arises when in-memory AML caching is disabled. *LRU_underlying_write_to_amls* is the function that actually creates new AMLs or writes to existing AMLs. Moreover, if a non-contiguous region becomes contiguous on a new write, this routine merges the AMLs to minimize overhead. This merging facility is the reason why the B-Tree was selected as a data structure for the AMLs in the first place. A linked list would provide the same convenience, but at the cost of access time. Access time would be less had we used an AVL tree, but the merge operation becomes too complex. Since a B-Tree is essentially a doubly linked list with an indexing mechanism built on top of it, it proves the most suitable solution to our problem. The pseudo code for *LRU_underlying_write_to_amls* is given below. The overall design is given, details like swapping to and from the cache and updating the seek pointer are omitted. The emphasis is on outlining the merge mechanisms. The following scenarios result when writing to an AML:

1. The offsets being written to will create a new AML disjoint from any previous AML
2. The offsets being written to lie completely within an existing AML
3. Start lies before an existing AML, End lies within

4. Start lies within an existing AML, End lies beyond

5. Start lies before an existing AML, End lies beyond

Each of these cases has several sub-cases depending on the existence of merge-able nodes. The pseudo code shows how these merges are done.

```
LRU_underlying_write_to_amls(FIT, writeBuffer, number_of_bytes)
{
  Set Start to the offset at which writing will begin
  Set End to the offset at which writing will end
  while(Start<=End)
  {
    Initialize found_node to the first node such that
      found_node.End>=Start
    Initialize prev_node to an AML such that
      prev_node.End=Start-1
    Initialize next_node to an AML such that
      next_node.Start=End+1
    /* Scenario 1: */
    if found_node does not exist or End<found_node->Start
    {
      if prev_node, the new data and next_node fit in memory
      {
        Merge all three into prev_node
        Update the B-Tree(s)
        return
      }
      /* If not we find the larger range given by the next
      two cases and merge that if it fits. We do this so
      that we maximize the amount of data in a single AML
      */
      if prev_node and the new data fit in memory
      {
        Merge both into prev_node
        Update the B-Tree(s)
        return
      }
      if the new data and next_node fit in memory
      {
        Create a new AML merging both
        Update the B-Tree(s)
        return
```

```
  }
  if prev_node exists and memory allocated to the PIF
    is zero
  {
    /*This is an optimization which allows merging of AMLs
    on disk when memory caching is turned off*/
    Create a new AML merging both by simply writing the new
      data at the end of the DiskBuffer
    return
  }
  /*If none of these cases work create a new AML from Start
  to End. This AML need not be an in-memory AML. createNewAML
  makes the decision of whether it will be in-memory or on-disk
  depending on whether it fits into memory or not
  */
  Create a new AML
}
/*Scenario 2: */
if both Start and End lie within found_node
{
  Update the data in found_node
  if found_node is on disk bring it into memory as the youngest
    cache member
  if already in memory, make it the youngest cache member
  return
}
/*Scenario 3: */
if Start lies before found_node and End lies within it
{
  /*In this case, we dont have to worry about next_node, since
  that merge is impossible
  */
  if prev_node, new data and found_node can fit  in memory
  {
    Merge all three into prev_node
    Update the B-Tree(s)
    return
  }
  if data and found_node can fit in memory
  {
    Create a new AML merging both
    Update the B-Tree(s)
```

```
    return
  }
  /*If all the previous cases are not possible create a new AML
  from Start to found_node.Start-1 i.e. no merging
  */
  create a new AML
  return
}
/*Scenario 4: */
if Start lies within found_node and End lies beyond
{
  if modified found_node and next_node can fit in memory
  {
    Merge both into found_node
    Update the B-Tree(s)
  }
  if modified found_node alone can fit in memory: End lies
    within an AML
  {
    Merge found_node and the node within which End lies
    Update the B-Tree(s)
  }
  if modified found_node alone can fit in memory: End lies
    outside an AML
  {
    Modify and extend found_node
    Update the B-tree(s)
  }
  if modified found_node will not fit in memory
  {
    Update found_node
    /*The rest of the data will be handled in the next
      iteration*/
  }
}
/*Scenario 5: */
if Start lies before found_node and End lies beyond
{
  if prev_node and next_node exist and prev_node.Start to
    next_node.End fits in memory
  {
    Expand prev_node to next_node.End
```

```
  Update B-Tree(s)
  return
}
if prev_node doesnt exist, and Start to next_node.End
  fits in memory
{
  Create a new AML from Start to next_node.End
  Update B-Tree(s)
  return
}
if prev_node exists, next_node doesnt; End lies within
  an AML end_node and prev_node.Start to end_node.End
    fits in memory
{
  Expand prev_node to go up to end_node.End
  Update B-Tree(s)
  return
}
if prev_node exists, next_node doesnt; End does not lie
  within an AML and prev_node.Start to End
    fits in memory
{
  Expand prev_node to go upto End
  Update B-Tree(s)
  return
}
Both prev_node and next_node do not exist, but End lies
  within an AML end_node and Start to end_node.End
    fits in memory
{
  Create a new AML from Start to end_node.End
  Update B-Tree(s)
  return
}
Both prev_node and next_node do not exist, End does not
  lie within and Start to End fits in memory
{
  /*This case is needed since default operation in this
  scenario is to create a new AML running from Start to
  found_node->Start-1
  */
  Create a new AML from Start to End
```

```
        Update B-tree(s)
        return
      }
      /*default: Create a new AML from Start to
      found_node->Start-1
      */
    }
  }
}
```

LRU_underlying_read_from_amls is a much simpler function, as no new AMLs are created and no merges are done. It merely reads data from the AMLs and from the original file if no AMLs exist. Finally *insert_into_ungetc_list* and *undo_ungetc_list* are used to push back characters into the stream for the ungetc library call.

4.3.5 Checkpoint and Restore:

The checkpointing and restore capabilities are provided by the following functions:

1. checkpoint
2. restore
3. commitFitList
4. commitFileList
5. rollbackFitList
6. rollbackFileList
7. createCheckpointFile
8. createFitMetaFile
9. createFileMetaFile
10. reloadDescriptors
11. reloadFitMetaFile
12. reloadFileMetaFile

checkpoint is the checkpoint call that the library exposes to application programs. In the single checkpoint case, it would result in all buffered changes being committed to the actual files, i.e., data that was written in the last checkpoint will now be written to the actual file, close and unlink operations will take effect. In the incremental checkpoint case, the operations are not committed

to the actual file since we can at any point restore to an earlier checkpoint. Instead, *meta* files are written, with all the information needed to replay those operations. Also, while in the single checkpoint mode *checkpoint* returns with either a success or failure, in the incremental checkpoint mode it imparts a *checkpoint index*, a handle which the application can then use to indicate which checkpoint it wants to restore to. The *restore* function, also available to applications, restores to the last checkpoint in single checkpoint mode. In incremental checkpoint mode, it restores to any previous checkpoint, as indicated by the checkpoint index.

commitFitList traverses the FIT list. in the single checkpoint case, it saves values to in-memory variables which can be used for the restore. In the incremental checkpoint case, it creates the FIT meta file and adds the descriptor/stream pointer to a list of descriptors/stream pointers that existed at checkpoint time. This list, which eventually becomes the checkpoint meta file, identifies which meta files need to be used to reload a checkpoint. If the descriptor/stream pointer was closed, it is not added to the list and the reference count for its underlying PIF entry is decremented.

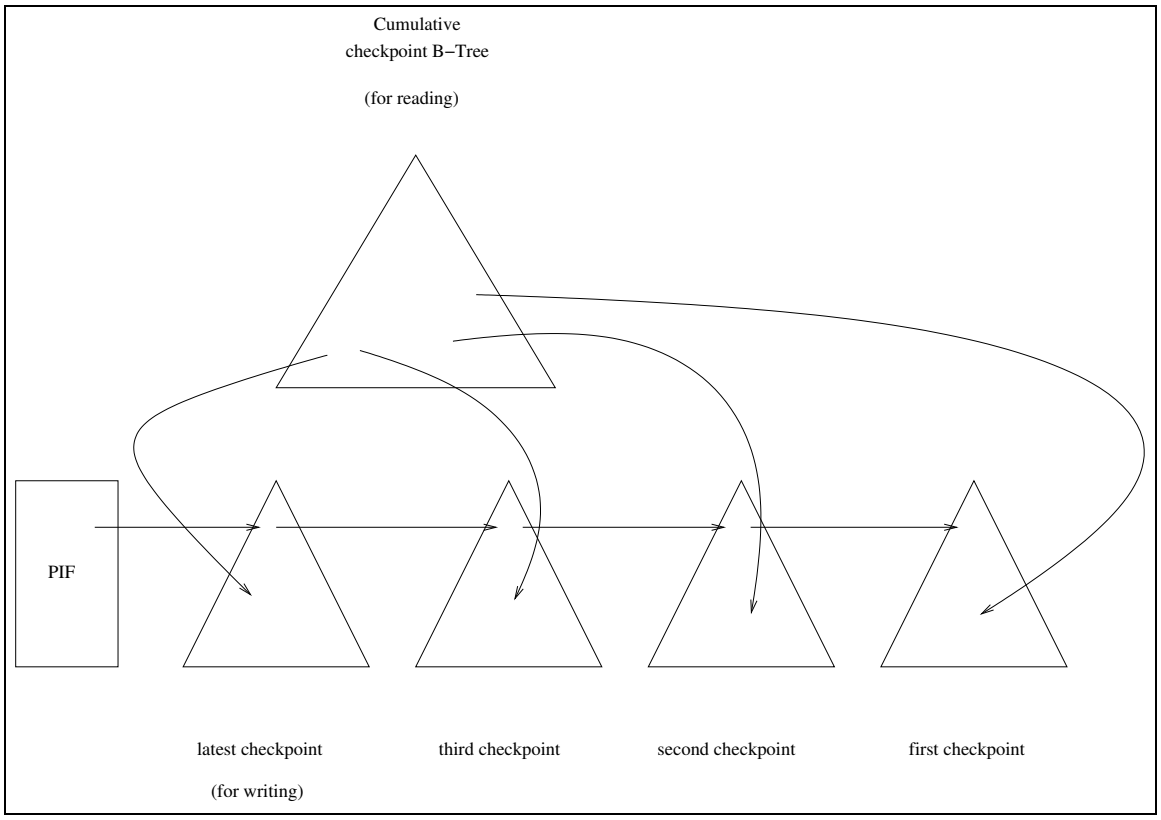

Figure 4.5: Reading and writing in the incremental checkpoint scenario.

commitFileList, in the single checkpoint case, commits the data to the original file. It then deletes all the data stored in the B-Tree and creates a new one. It also clears the cache. A few values, such as the count of the number of FITs that reference the PIF, are also saved in memory variables, for a restore. In the incremental checkpoint case, it creates the PIF meta file. All the in-memory AMLs are swapped to disk, then their offsets are recorded in the offset file. In incremental

checkpoint mode, a linked list of B-Trees is maintained, apart from the cumulative B-Tree which is used when data is read. Each of these B-trees holds the written data at each checkpoint after the last restore. Hence, a new B-Tree is created and inserted at the head of the list. This structure can be better understood from Figure 4.5. All writes now insert data into this latest B-Tree, as well as the cumulative B-Tree, which maintains the property that data written in later checkpoints overrides data written in previous checkpoints. Hence, at checkpoint time, the last checkpoint's B-Tree is the one that has all its AMLs swapped to disk and the offset file created for it.

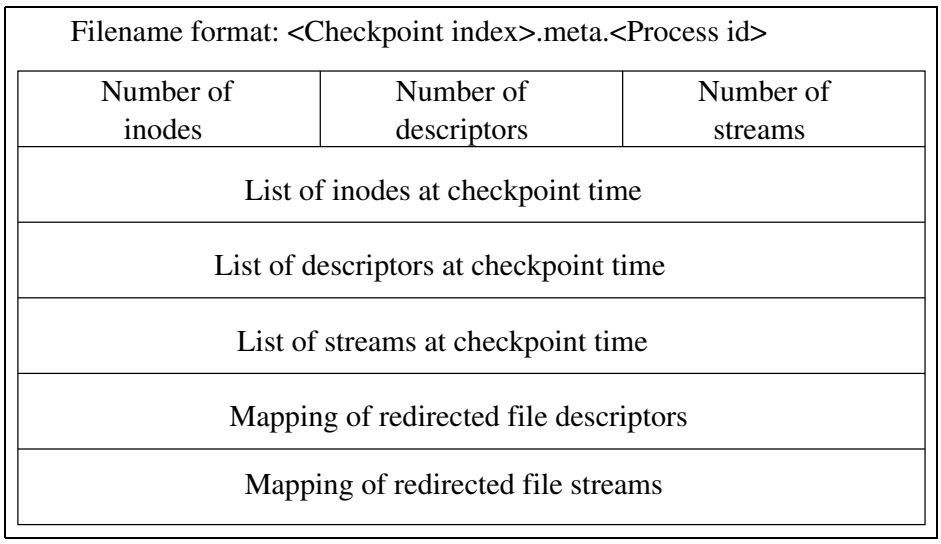

Figure 4.6: Checkpoint Meta File Format.

rollbackFitList serves to make the FIT list as it were at the checkpoint we restore to. In the single checkpoint case, this means removing all FITs that were created at the last checkpoint, and reloading the saved values for the existing FITs. In the incremental checkpoint case, this function finds the appropriate meta file for each FIT and reconstructs the FIT using meta file data. Since the FIT list is cleared at the start of a rollback, this function also reconstructs the list by inserting every successfully reconstructed FIT into it. *rollbackFileList* performs the same operations for PIFs. Again for single checkpoints, a simple clearing of the existing buffers and cache and a reload of saved memory values proves sufficient. In the incremental checkpoint scenario, the PIFs are reloaded from their corresponding meta files and the PIF list is reconstructed.

createCheckpointFile saves the data needed for a particular checkpoint in a serial format in a meta file. The file format is given in Figure 4.6. Similarly *createFitMetaFile* saves all the pertinent information for each FIT in a file: this file format is given in Figure 4.7. The format for the PIF meta file that *createFileMetaFile* creates is given in Figure 4.8.

4.3.6 Overloaded system and library calls

While most of the overloaded system and library calls make use of the underlying functionality in obvious ways, a few deserve special mention. The *close* system call does not actually set the

Filename format: <descriptor/stream pointer>.<Checkpoint index>.<Process id>

creation type	descriptor	stream pointer
attributes	stream attributes	mode
filename	seek pointer	wasCreated
wasClosed	flags	PIF's inode
list of duped descriptors		
list of pushed back characters (ungetc)		

Figure 4.7: FIT Meta File Format.

wasClosed flag for a FIT until its original descriptor is closed and its list of duped descriptors is empty. Until then, the descriptors are simply removed from the FIT one by one so that successive calls cannot reference them. The *dup* system call simply adds the new descriptor to the linked list of duped descriptors maintained by the FIT. This enables all these descriptors to share the seek pointer which is common to a FIT. Finally the *unlink* call must be examined, since unlinking a file is counter-intuitive when we talk of buffered operations. The problem is that if we unlink a file, then create it again without calling a checkpoint, the newly created file will conflict with its unlinked predecessor, since the unlink does not take effect till checkpoint time. The solution is to rename the unlinked file so that this conflict will not occur, achieved by tacking on a ".unlinked" extension along with the checkpoint interval in which the file was unlinked. Any descriptors that reference the renamed file will still work, hence this is not a problem to open descriptors and streams. In incremental checkpointing, the renaming operation need be done only for the first unlink following a checkpoint; all successive unlinks need not rename since the restore will only need this first copy.

4.3.7 Summary

We have presented the subsystems of our software framework and their functions. What remains to be explained is how these components interact with one another to provide the functionality that we expect. There are two cases to be covered: the single checkpointing case and the incremental checkpointing case. Let us consider how this library works for the simple case of opening files, reading and writing them, and closing them.

Filename format: <inode number>.<Checkpoint index>.<Process id>		
inode	count	filename
wasCreated	wasDeleted	filesize

Figure 4.8: PIF Meta File Format.

Single checkpoint case:

Let us assume that we have a per-file memory buffer of 3 MB. The following is our scenario:

1. Create two files *a.txt* and *b.txt*, and open them for reading and writing.
2. Write 5 MB of data to a.txt in 1 KB blocks
3. Call a checkpoint
4. Write 3 MB of data to b.txt in 1 KB blocks
5. Call a checkpoint
6. Close the descriptor or stream to b.txt
7. Unlink b.txt
8. Call a checkpoint

In Step 1, two PIFs will be created; one for *a.txt* and one for *b.txt*. Two FITs will also be created, one for each file. When 5 MB of data is written to a.txt, an in-memory AML will be created that grows in size from 1 KB to 3 MB, since the data is written contiguously. Once the 3 MB memory limit is reached, this AML will be swapped to disk by the caching subsystem, to make way for new data. A new AML will be created, which will subsequently grow from 1KB to 2 MB to accomodate data from offsets 3 MB to 5 MB. At checkpoint time, all this buffered data is committed to *a.txt*, so that after the first checkpoint, a.txt will be a 5 MB file while b.txt will be empty. The memory buffer is cleared and the disk buffer is truncated. Any failure after this and before the next checkpoint will leave the files in that state. The operation of the library in the next checkpoint interval is similar, resulting in 3 MB of data being committed to *b.txt*. In the third checkpoint interval, we close the open descriptor and unlink the underlying file. This results in *b.txt* being marked for deletion, and immediately being renamed to *b.txt.unlinked*. This renaming is necessary because otherwise, if a new b.txt file is created, it might use the existing,

yet to be unlinked file. This is a consequence of buffered operations which are committed only at checkpoint time. The unlink operation needs to take effect immediately, but this is not possible due to the buffered semantics that we are trying to maintain. Hence the rename is used to make it seem to the program that the file actually does not exist anymore.

Incremental checkpoint case:

Let us consider what happens for the following set of operations in the incremental checkpoint scenario. Let us assume again that we are working with a 3 MB per-file memory buffer.

1. Create *a.txt*, and open it for reading and writing.
2. Write 5 MB of data to a.txt in 1 KB blocks
3. Call a checkpoint
4. Create *b.txt*, and open it for reading and writing.
5. Write 3 MB of data to b.txt in 1 KB blocks
6. Call a checkpoint
7. Close the descriptor or stream to b.txt
8. Unlink b.txt
9. Call a checkpoint

In Step 1, a PIF is created for a.txt and the data is written into its buffers in the same manner as the single checkpoint case. At the first checkpoint, a checkpoint meta file is created recording that the inode corresponding to *a.txt* existed at checkpoint time, and the value for the corresponding open descriptor or stream. The FIT entry and the PIF also have corresponding meta files created for them. These two also incorporate the checkpoint index in their filenames. Similarly, at the second checkpoint, meta files are created for the PIF and FIT corresponding to *b.txt*. The meta file for the second checkpoint now records two inodes, one for *a.txt* and another for *b.txt*. In the third checkpoint interval, the only difference from the single checkpoint scenario is that the renamed filename also incorporates the checkpoint index, so that we will know which copy of the file to use for a restore opearation. This is because the same file can be unlinked in different checkpoint intervals and the rename will have to be performed for the first such operation after every checkpoint. Now if we were to restore to the first checkpoint, *b.txt* should not exist. This is accomplished by the restore method looking at the *wasCreated* field of each PIF, which gives the checkpoint interval in which the file was created. Then, if we are restoring to a point that precedes the creation of the file, the file is unlinked during the restore.

In the single checkpoint scenario, the changes to a file are not committed until we call a checkpoint. Similarly, in the incremental case, the changes are not committed upto a checkpoint unless we restore to that checkpoint. Since it is possible that a checkpoint is not called by the user when a program exits, we override the POSIX *atexit* call to effect this final commit.

Chapter 5

Results

The main goal of Metamori is to provide both single and incremental checkpointing factilities to programs. This library can then be deployed with an incremental memory checkpointer to provide system-level checkpoint and restore capabilities. There are two parameters on which the library needs to be evaluated: The one is correctness and the other is performance.

This chapter presents Metamori's results on both these fronts. The first part of the chapter is a discussion of the various verification tests that Metamori was put through. These tests are heavily based on the Linux man pages [3]. The second part evaluates the performance overhead by deploying the library to checkpoint an I/O benchmark: Bonnie++ [1].

5.1 Experimental Setup:

Since we are just running normal Linux programs with a shared library linked in for checkpointing, a single PC is sufficient for experimental observation. Even so, the configuration of the machine is presented here with a view to proving that Metamori works on an off-the shelf PC running Linux. The configuration of the test machine is as follows: 2 Ghz CPU, 1 Gigabyte RAM, 200 GB Hard Disk Drive and a Debian distribution of Linux running kernel 2.6.1.

5.2 Verification Testing:

Metamori overloads a number of I/O system and library calls. This section presents an overview of the tests performed to ensure that Metamori upholds the behaviour of these system and library calls. The justification for the tests is also presented. The four main areas of interest are reading, writing, opening and closing files, since these are the basic file operations. Of these, the opening of descriptors and streams are trivial, in that the underlying open/fopen call must be made anyway. This operation cannot be buffered. Hence most of the scenarios for open are covered when the underlying open or fopen routine is called. Hence no major tests are required for opening files.

The read tests were performed for both sequential reads and reads of a file starting at intermediate points. They were also performed for files with holes. In all cases the overloaded read calls

returned the same results that the test programs gave when the library was not used. Further, to test the correctness of incremental checkpoint and restore operations, a checkpoint was taken after every test and a restore to one of the previous checkpoints was made after the last test. When this was done all the test files created after that restore point disappeared, as they were expected to. The write tests were more complicated. Since the read tests did not create new AMLs, they are lower in complexity. However, for the writes, every subcase of the *LRU_underlying_write_to_AMLs* algorithm was tested by a separate program. Also, a comprehensive test was also drawn up for both the single and incrmental cases. These were the operations tested:

1. Writing sequentially to a file
2. Non-overlapping interleaved writes to a file
3. Overlapping interleaved writes where the later block overlaps the existing one at the front
4. Overlapping interleaved writes where the later block overlaps the existing one at the back
5. Overlapping interleaved writes where the later block completely encompasses its predecessor

In addition a checkpoint was taken at the end of each test and a restore could be done at the end of all the tests to any of the previous checkpoints. All the tests worked correctly, and the restore succeeded in reproducing the file state at any of the earlier checkpoints. The dup system call presents an interesting case for incremental checkpoints in that the set of duped descriptors might have changed between checkpoints. Hence, this particular aspect of descriptor reloading was tested: a descriptor was duped before the first checkpoint, the duped descriptors were closed in subsequent checkpoints and then a restore was done to the first checkpoint. Following this restore, the original set of duped descriptors were written to, and the writes succeeded, thereby proving that the list of duped descriptors was succesfully reconstructed. For the fdopen call, we check that closing the stream closes the underlying descriptor as well. The feof, ferror and clearerr tests check that the correct values of the flags are returned for the correct scenarios. Test cases were also written to verify the correct operation of seeks for descriptors and streams. Finally, all the stream variations of reads and writes such as getc, fgetc, fgets, putc, fputc and fputs were verified.

5.3 Performance testing:

As mentioned before, Bonnie++ was used to evaluate the performance overhead of using Metamori. The tests were carried out in the absence of the library and then repeated with the library loaded in. One minor change had to be made to make the library work with Bonnie++. Most notably, since the library overloads all output calls including fprintf, we had to make Bonnie++ use the underlying fprintf call directly, else the output would not be displayed. This was the only change made.

We briefly decribe the tests that Bonnie++ runs. It ascertains the underlying file system's block size. The first test, the block write test, writes 2 Gigabytes of data, one block at a time. The second test, called the block re-write test, reads each block, dirties it and writes it back. The third and final test, the block read test, reads the 2 Gigabytes block by block. The reason 2 Gigabytes is the operating size is that is twice the memory size on our test system. Also, 32-bit Operating Systems without large block support cannot create files of sizes greater than 2 Gigabytes.

5.3.1 Performance overhead:

File size: 2 Gigabytes	No Checkpointing	100 MB Cache	50 MB Cache	25 MB Cache	No Cache
Block Write Tests	49115	41361	46560	47725	51276
	49863	42375	43898	46507	51604
	48437	40212	45143	50606	51073
	50473	36460	43549	47620	51154
	49557	38520	44340	45270	52279
Block Re−write Tests	17446	18286	18800	19575	17128
	17980	19017	18948	18657	18242
	18207	20158	19364	18706	17792
	17757	20351	18643	18809	18239
	17552	18834	18068	18022	18058
Block Read Tests	30815	38097	38083	38416	36749
	32448	37149	36306	40148	38907
	35439	37340	38534	37229	37511
	31126	36979	37110	36596	39003
	34594	38408	39055	37235	37956

All results in Kbps

Figure 5.1: Bonnie++ results for a 2 GB test file size.

Figure 5.1 shows the numbers attained in the absence and presence of Metamori. We have also modified the size of the per-file memory cache to see the effect it has on performance. As can be seen, the performance with the library linked in is almost commensurate with the performance in the absence of the library. It must be noted that the block write and block read tests effectively test the worst case performance of the cache since they do not employ locality at all. The re-write tests have greater locality, and in these we see that the results match to a great degree.

Tables 5.1 and 5.2 show the standard deviation and variance for the values obtained above.

Figures 5.2, 5.3 and 5.4 plot the averages of the runs shown in Figure 5.1. It can be seen that in the case where locality is exploited, namely the block re-write tests, the presence of the cache greatly aids performance.

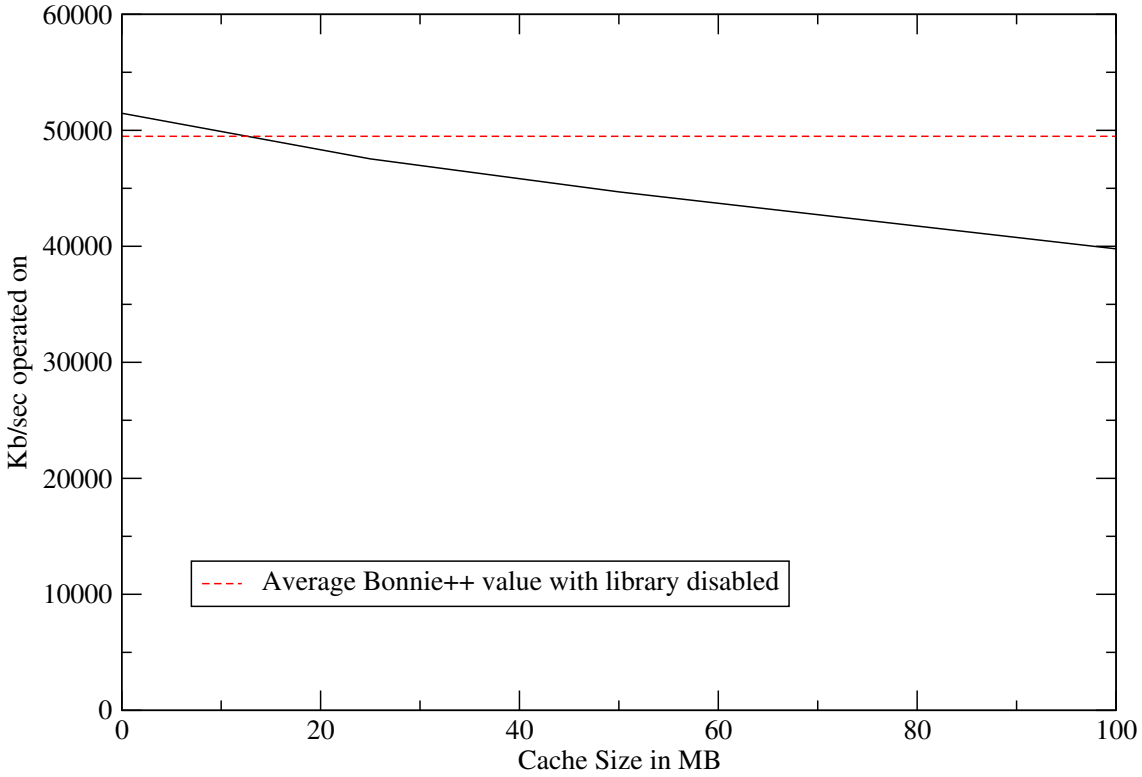

Figure 5.2: Plot of averages for Block Write tests.

5.3.2 Timing breakdown:

This section presents the time taken in each section of the library's code in an attempt to explain the performance overhead. The overloaded write calls can be broken down into three subsections:

1. Search the BTrees for a predecessor, a candidate node and a successor
2. Update/Create the AMLs
3. Insert into the BTrees

Searching the B-Trees, as it turns out in this case, is not an expensive affair. This is because of the merging of contiguous data that our write algorithm affords. The only time a new AML is

File Size: 2 Gigabytes	No checkpointing	100 MB Cache	50 MB Cache	25 MB Cache	No cache
Block Write Tests	686.93	2099.05	1072.70	1769.79	439.84
Block Re-write Tests	277.87	795.23	422.90	494.89	415.73
Block Read Tests	1844.64	557.92	990	1257.74	852.24

Table 5.1: Standard Deviation for Bonnie++ results.

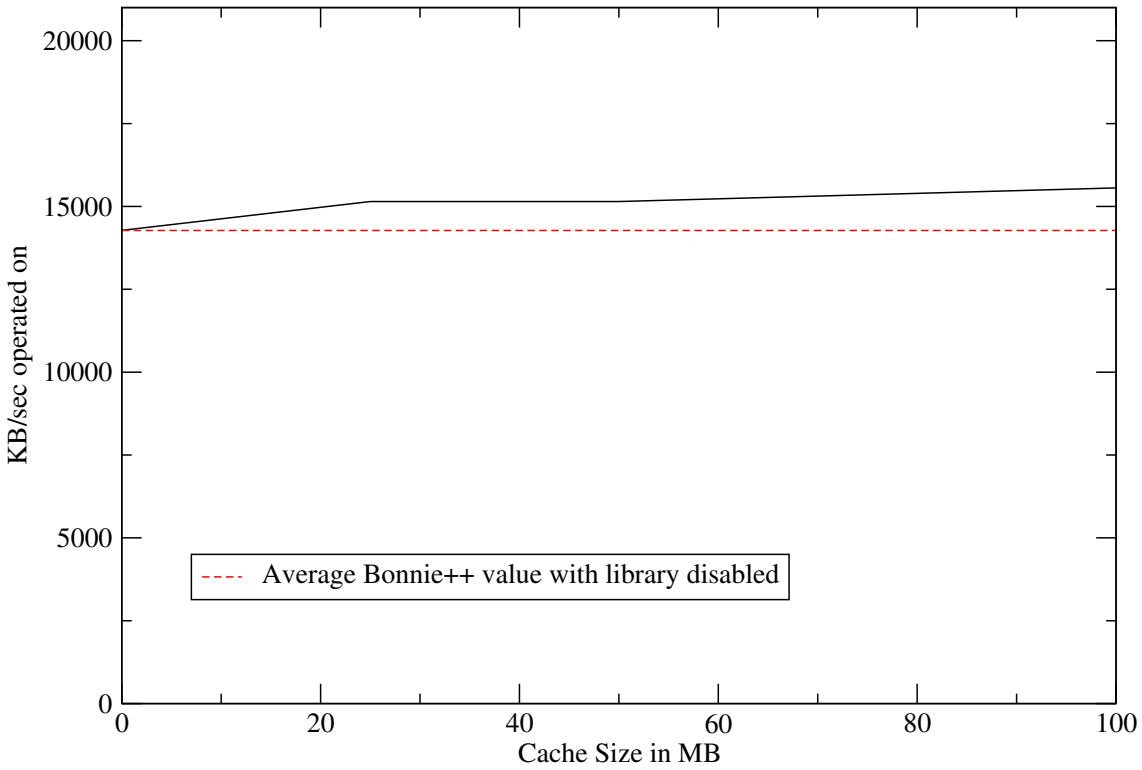

Figure 5.3: Plot of averages for Block Rewrite tests.

created is when the size of the existing AML approaches the cache size, which is 25, 50 or 100 MB. Therefore the B-Trees have a very small number of nodes which explains the very low access times in finding the predecessor, successor and candidate. Table 5.3 shows the time taken.

Updating the AMLs involves extending an existing AML to include the newly written data. In the in-memory case, this involves calling a *realloc* on the existing memory buffer and appending the data at the end. Again, since these operations take place completely in memory (except the case when the memory cache is disabled), the average time taken is not too much. Table 5.4 shows the average times taken. The 0 MB case takes much more time because in this case, we are writing to disk rather than memory.

Creating a new AML, in contrast, takes a lot more time. This is because of the pattern of writes in Bonnie++. Since the writes are sequential, a new AML is created only when the memory

File Size: 2 Gigabytes	No checkpointing	100 MB Cache	50 MB Cache	25 MB Cache	No cache
Block Write Tests	471867.2	4406010.64	1150686.8	3132162.64	193455.76
Block Re-write Tests	77209.04	632388.56	178841.44	244920.56	172832.16
Block Read Tests	3402709.04	311273.84	980091.44	1581913.36	726308.16

Table 5.2: Variance for Bonnie++ results.

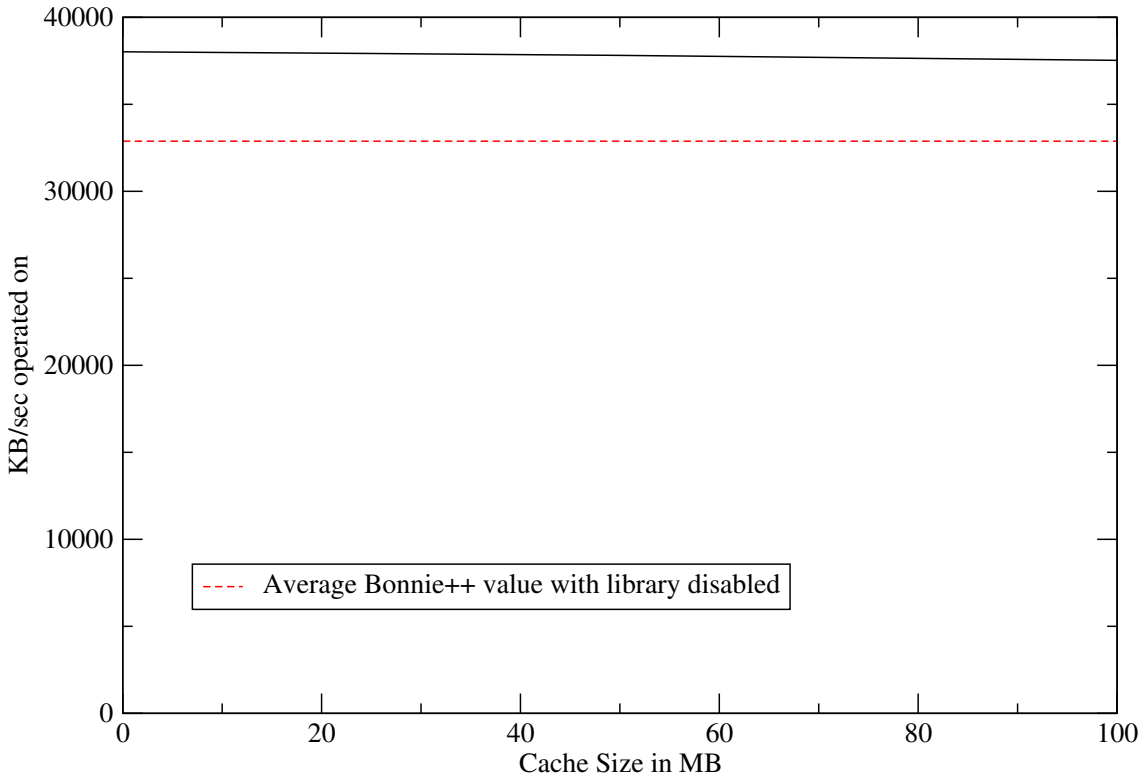

Figure 5.4: Plot of averages for Block Read tests.

buffer is filled. This consequently means that the old data (which completely occupies the memory buffer) must be swapped to disk before the new AML can be created. Therefore, the entire cache is written to disk while creating a new AML, a time-consuming operation. This is reflected in our timing breakdown, where we see that except for the case where there is no memory buffer, the time taken is much more than anything we have seen yet in our timing breakdown. The absence of the memory buffer, however, means that creating a new AML in that case takes very little time. Table 5.5 shows the time take. Further to prove that it is the swapping to disk that is taking all the time when creating a new AML, we have separatley timed the *swapAMLsToDisk* function. The results, presented in Table 5.6 shows the time taken. Also, to prove that the non-linearity in the writes

Cache Size	Average Time for B-Tree Search
0 MB	$11\mu s$
25 MB	$12\mu s$
50 MB	$12\mu s$
100 MB	$9\mu s$

Table 5.3: Timing breakdown for initial search phase.

Cache Size	Average Time for extending an AML
0 MB	$122\mu s$
25 MB	$15\mu s$
50 MB	$16\mu s$
100 MB	$14\mu s$

Table 5.4: Timing breakdown for extending AMLs.

Cache Size	Average Time for Creating a new AML
0 MB	$41\mu s$
25 MB	$402985\mu s$
50 MB	$898879\mu s$
100 MB	$1861875\mu s$

Table 5.5: Timing breakdown for creating a new AML.

was due to the underlying system and not due to our library, we have tabulated the total time taken executing write system calls. While one would expect that the time taken to write the same amount of data in 100 MB chunks would be less than in 50 MB chunks, we see from Table 5.7 that this is not the case.

Finally, the updated AMLs must be inserted back into the B-Trees, and the timing breakdown for this operation is shown in Table Finally, the updated AMLs must be inserted back into the B-Trees, and the timing breakdown for this operation is shown in Table 5.8. These are the average times recorded only for the case where the AML is being extended since we have encompassed the B-Tree insertion time for new AMLs in the numbers for creating a new AML. There is nothing new to be learned by timing this insertion separately for new AML insertion since we have already ascertained that the cause of the overhead is the swap to disk of the memory buffer.

Cache Size	Average Time for swapping an AML to disk
25 MB	$356305\mu s$
50 MB	$756347\mu s$
100 MB	$1860719\mu s$

Table 5.6: Timing breakdown for swapping an AML to disk.

Cache Size	Total Time spent in write system calls
25 MB	$30861530\mu s$
50 MB	$26016693\mu s$
100 MB	$28974338\mu s$

Table 5.7: Total time spent in write system calls.

Cache Size	Average Time for updating the B-Trees
0 MB	$19\mu s$
25 MB	$16\mu s$
50 MB	$16\mu s$
100 MB	$15\mu s$

Table 5.8: Timing breakdown for updating the B-Trees.

Chapter 6

Conclusions and Future Work

6.1 Summary:

This thesis has presented Metamori, a library for file checkpointing. The main difference from previous work in this field is that this library is capable of checkpointing files incrementally, and restoring to any of the checkpoints taken. The justification for this work is that in the presence of an incremental memory checkpointer, this library furthers the goals of completely checkpointing the state of the application.

The library integrates with existing code by providing a set of function calls identical to Linux's family of system and library calls for input/output operations. Thus, while the application thinks that it is calling the underlying C library, control is handed to Metamori with a view to providing checkpointing. All the operations between checkpoints are modeled as transactions which are not committed till the next checkpoint is taken in the single checkpoint case. In the incremental checkpoint case, the underlying files are not mutated until a restore operation is done, which essentially lets us backtrack to any previous checkpoint. The extra level of indirection afforded by our library also makes it possible to mask from the user the underlying changes to file descriptors and stream pointers that happens on a restore.

Verification testing shows that the library works as the underlying C library would. Performance testing has shown that the overhead is acceptable, particularly when the I/O pattern is such that the LRU caching scheme is used.

In summary, this thesis provides the following contributions:

1. Support for both file descriptors and streams
2. Descriptor safety across checkpoints
3. An LRU cache for the buffered data
4. Incremental checkpointing capability

6.2 Future Work:

Now that we have designed and implemented an incremental file checkpointing library, the next step would be to checkpoint the complete state of an application, both volatile and persistent. Hence, Metamori should be integrated with an existing checkpointing application capable of checkpointing volatile state. This would then result in a fully functioning incremental checkpointer for applications.

Another possible application of this work is to use it in adaptive frameworks: frameworks that can run an application, watch it fail, figure out what went wrong, then rollback to a previous state and continue execution in such a way as to correct the previous error. Incremental checkpointers are a vital component of such execution control.

Bibliography

[1] Bonnie++: An i/o benchmark library. `www.coker.com.au/bonnie++`.

[2] Cap research project: Checkpoint/restart for solaris and linux. `http://cap.anu.edu.au/cap/projects/esky/index.html`.

[3] Linux man pages. `http://linux.ctyme.com`.

[4] Paracel: Applied high performance computing. `http://www.paracel.com/pc/why.htm`.

[5] M. J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall of India, New Delhi, India, 2004.

[6] Y. Chen, J. S. Plank, and K. Li. *CLIP: A Checkpointing Tool for Message-Passing Parallel Programs*, pages 182–200. The MIT Press, Cambridge, MA, 2004.

[7] P.E. Chung, Y. Huang, S. Yajnik, G. Fowler, K. Vo, and Y. Wang. Checkpointing in cosmic: a user-level process migration environment. In *Pacific Rim International Symposium on Fault-Tolerant Systems.*, 1997.

[8] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[9] Dan Pei Dongsheng. Design and implementation of a low-overhead file checkpointing approach. In *The Fourth International Conference on High Performance Computing in the Asia-Pacific Region(HPC-Asia 2000)*, pages 439–441, 2000.

[10] J. Duell, P. Hargrove, and E. Roman. The design and implementation of berkeley lab's linux checkpoint/restart. Future Technologies Group white paper, 2003.

[11] E. Elnozahy, D. Johnson, and Y. Wang. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-96-181, Carnegie Mellon University, October 1996.

[12] K. A. Iskra, F. van der Linden, Z. W. Hendrikse, B. J. Overeinder, G. D. van Albada, and P. M. A. Sloot. The implementation of dynamite: an environment for migrating pvm tasks. *SIGOPS Oper. Syst. Rev.*, 34(3):40–55, 2000.

[13] M. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the unix kernel. In *Proc. of the Winter 1992 USENIX Conference*, pages 283–290, San Francisco, California, 1991.

[14] D. Pei. Modification operations buffering: A low overhead approach to checkpoint user files. In *Proceedings of IEEE 29th Symposium on Fault-Tolerant Computing*, pages 36–38, 1999.

[15] J. S. Plank, M. Beck, G. Kingsley, and K. Li. **Libckpt**: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

[16] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. Technical Report UT-CS-96-335, University of Tennessee, August 1996.

[17] J. S. Plank, J. Xu, and R. H. B. Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, August 1995.

[18] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.

[19] Asim Shankar. Process checkpointing and restarting (using dumped core). `http://www.geocities.com/asimshankar/checkpointing/`.

[20] L. M. Silva and J. G. Silva. System-level versus user-defined checkpointing. In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, page 68. IEEE Computer Society, 1998.

[21] W. Richard Stevens. *Advanced programming in the UNIX environment*. Addison Wesley Longman Publishing Co., Inc., 1992.

[22] Z. Weimin W. Dongsheng, S. Meiming and P. Dan. A checkpoint-based rollback recovery and process migration system. pages 68–73, January 1999.

[23] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung, and Chandra M. R. Kintala. Checkpointing and its applications. In *Symposium on Fault-Tolerant Computing*, pages 22–31, 1995.

[24] H. Zhong and J. Nieh. Crak: Linux checkpoint/restart as a kernel module. Technical Report CUCS-014-01, Columbia University, November 2001.

Vita

Ashwin Raju Jeyakumar was born in Chennai (formerly Madras), India. He received his Baccalaureate degree in Computer Science and Engineering from the College of Engineering, Anna University. As part of his undergraduate research, he designed and implemented an Active Congestion Control system using Explicit Congestion Notification.

Ashwin enrolled in the Master's program in the Department of Computer Science at Virginia Tech in Fall 2002. In his first year in graduate school, he worked part-time as a programmer for the Instructional Technology Master's Program (ITMA). After his first year in graduate school, he interned with Microsoft Corp.'s Exchange Server team as a Software Design Engineer in Test. In his second year in graduate school, he worked as a Research Assistant in the Computing Systems Research Laboratory (CSRL), developing a library for incremental file checkpointing which eventually became his thesis. His research interests during his graduate studies have included computer networks, computer architecture, grid computing, operating systems, advanced parallel computing, algorithms and rollback-recovery. He plans to return to Microsoft Corp.'s Exchange Server division as a Software Design Engineer in Test upon graduation in July 2004.