ABSTRACT

VASAVADA, MANAV M. Innovative Schemes to Support Incremental Checkpointing. (Under the direction of Dr. Frank Mueller.)

The rapid increase in the number of cores and nodes in high performance computing (HPC) has made petascale computing a reality with exascale on the horizon. Harnessing such computational power presents a challenge as system reliability deteriorates with the increase of building components of a given single-unit reliability. Todays high-end HPC installations require applications to perform checkpointing if they want to run at scale so that failures during runs over hours or days can be dealt with by restarting from the last checkpoint. Yet, such checkpointing results in high overheads due to often simultaneous writes of all nodes to the parallel file system (PFS), which reduces the productivity of such systems in terms of throughput computing. Recent work on checkpoint/restart (C/R) has shown that incremental C/R techniques can reduce the amount of data written at checkpoints and thus the overall C/R overhead and impact on the PFS.

The contributions of this work are twofold. First, it presents the design and implementation of two memory management schemes that enable incremental checkpointing. We describe unique approaches to incremental checkpointing that do not require kernel patching in one case and only require minimal kernel extensions in the other case. The work is carried out within the latest Berkeley Labs Checkpoint Restart (BLCR) as part of an upcoming release. Second, we evaluate the two schemes in terms of their system overhead for single-node microbenchmarks and multi-node cluster workloads. In short, this work is the final showdown between page write protection and dirty page tracking as a hardware means to support incremental checkpointing.

Innovative Schemes to Support Incremental Checkpointing

by
Manav M. Vasavada

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2010

APPROVED BY:

Dr. Xuxian Jiang	Dr. Xiaosong Ma

Dr. Frank Mueller
Chair of Advisory Committee

DEDICATION

To my parents, Bhavin and Kaka.

BIOGRAPHY

Manav Vasavada was born in Mumbai in the state of Maharashtra in India. After jumping through a couple of schools in different towns and not being quite ecstatic about them, he finished most of his schooling(grudgingly) in the town of Ahmedabad where he grew up. He joined the first batch of Nirma University of Science and Technology for an undergraduate degree in computer engineering and after four more years of kicking and screaming he was awarded a first class in Computer Engineering. He spent one year after his undergrad working for Cognizant Technology Pvt. Ltd, where he worked mostly on the ABN-AMRO account as a developer and also writing test suites for the team. In 2008, in order to pursue higher education, he joined the Master's program at NC State University in computer science. He has since then worked under Dr. Frank Mueller as a Research Assistant.

ACKNOWLEDGEMENTS

This work would not have been possible without the collective effort and support from a lot of people. First of all, I would like to thank my advisor Dr. Frank Mueller for having faith in me and giving me the opportunity to work with him. His patient guidance and expertise has played a major role in the formation of this work. I would also like to thank Paul H. Hargrove and Eric Roman at Lawrence Berkeley National Laboratory for acting as mentors during my summer internship and my thesis work in general. I have learned a lot not only about my current work, but about my field in general from them. I would also like to thank Dr. Xuxian Jiang, Dr. Xiaosong Ma, Dr. Freeh, Dr. Gu and other faculty members of the Systems Research Group. Lastly I would like to thank all my labmates in the Systems Research Lab Karthik Vijayakumar, Divya Dinakar, Amey Deshpande, Balasubramaniam bhat, Sandeep Budanur, Chris Zimmer, Abhik Sarkar, Yongpeng, Xing Wu, Feng Ji, Fei Ming, Abhishek Sreenivas.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

1.1 Parallel Computers

Computer systems have evolved over time to become faster and more efficient. Parallel computers or super computers date back a long time back. Supercomputers gained visibility during 1970s and 1980s when large vector based machines were quite popular. Parallel computers lost some ground during 1990s due to rapid development of fast uniprocessor machines and the challenge in parallel programming. The parallel computing trend has picked up recently since the development of uniprocessor machines has hit the frequency scaling wall. *Parallel computing* today is available in form of multicore machines, where a single node contains a number of processor cores sharing some level of cache and other resources, and in form of clusters where different physical machines are connected through a medium of communication, to coordinate and synchronize tasks. The later type of parallel computers are also divided into homogenous and heterogenous clusters, depending on the type and configuration of the nodes present in the clusters. Parallel computing is also referred to as High Performance Computing (HPC). HPC or Parallel computing involves exploiting various kinds of parallelism in a given application and executing some of the tasks in parallel so as to reduce execution time. Many scientific applications as well as normal applications contain parallelism at various levels, which can be exploited by HPC systems. The various levels of parallelism are instruction level, data, and task parallelism.

Since traditional supercomputers are custom made and therefore quite expensive, clusters are used prominently in academic research. Clusters are made up of components available off the shelf. These are quite cheap and readily available. The most common programming paradigm found in such machines is the message passing paradigm. Each machine is allocated to execute some part of the application. It communicates with other nodes over the network using a standard set of functions defined in the Message Passing Interface (MPI). There are various

implementations of MPI out today. Some of the popular ones are OpenMPI [9], MPICH-V [5], MVAPICH.

With the restrictions in frequency scaling facing the computing field today, more and more institutions and companies are turning to parallel machines or HPC. Recently, the IBM Roadrunner supercomputer at Los Alamos National Lab and the Cray XT-5 Jaguar at Oak Ridge National Laboratory were amongst the machines that broke the Petaflop barrier in Linpack performance [1]. To take maximum advantage of HPC machines, applications need to be rewritten for parallel execution, which is more difficult than writing a sequential program. Parallel programming is the biggest challenge facing the industry today.

1.2 Fault Tolerance

A *Fault* is defined in computing as an event in which the computer deviates from the normal behaviour of the application. A fault is an underlying inconsistency in the hardware or the software which can cause an error and thus cause the program to fail. The faults can be classified into software faults and hardware faults. Software faults constitute erronous program code unable to handle a specific set of input data. Such kind of faults can be solved by programmers on rewriting the application. Hardware faults, on the other hand, are faults generated due to defects or limitations of the underlying hardware. Such faults could include failure of input/output (IO) operations by the IO bus, ECC failure on a faulty RAM due to solar burst or other reasons, block corruption on hard disks while reading data etc. In our work, we mainly concentrate on the hardware faults of the cluster. Faults can be classified into [2]

1. Crash/fail-stop faults – the component either occasionally stops operating or never returns to a valid state;
2. Omission faults – the component completely fails to perform its service;
3. Timing faults – the component does not complete its service on time;
4. Byzantine faults – these are faults of an arbitrary nature.

In our work, we mainly concentrate on the crash faults.

With the development in the supercomputer field, the number of core in high-performance computing has scaled upto thousands of processor cores. Large scientific applications with highly parallel execution patterns are exploited fully on these machines. Even with the amount of processor power available, applications may still take upto days to execute on such high-end machines. Such applications include climate modeling, protein folding, 3D modeling etc. With the increase in use of off-the-shelf components to create parallel machines as well as the increase

in the sheer number of processing cores on a single machine, the Mean Time Between Failure (MTBF) has also decreased significantly[Citation needed]. This indicates the increasing chance of hardware failure while a process is executing. For large processes like the ones mentioned above, this would mean restarting the process from scratch and performing all the work again thus wasting precious processing time. Along with delay in results, this would also mean excessive use of power for doing duplicate computation.

Fault Tolerance can be defined as the ability of the machine to perform its designated function correctly in presence of internal faults. Fault tolerance involves detecting the error, stopping the erronous processes and correcting the error. An efficient fault tolerance policy can save a significant amount of computing power as well as execution time.

1.3 Checkpoint Restart

There have been many approaches to support fault tolerance in HPC. One of the approaches is checkpoint/restart (C/R). This approach involves saving the state of the application at regular intervals. The act of saving the state of the application is known as checkpointing. Hence this approach involves checkpointing the application on each node at regular intervals of time to non-local storage. Upon failure, the computation is simply shifted to a spare node and the application is restarted from the last checkpoint instead of starting the application from the scratch.

For example, consider a parallel application running on 10 nodes with one spare node. This application is being checkpointed every hour. Now consider that one of the nodes fails after 5.5 hours. Without checkpointing, this would require an entire application restart, thus wasting the time already spent executing the process. With checkpointing, we just transfer the last checkpoint to the spare node and start the application from the last checkpoint, i.e., the checkpoint taken at the 5th hour mark. See Figure 1.1 :

Checkpointing involves saving the state of the process at a point in time and then using the same data at the time of restart. There have been various frameworks for application as well as system-level C/R. There has been a study on various type of checkpointing mechanisms [14]

1.4 Berkeley Labs Checkpoint Restart(BLCR)

Berkeley Labs Checkpoint Restart (BLCR) is a hybrid kernel/user implementation of checkpoint/restart developed by Future Technologies Group researchers at Lawrence Berkeley National Laboratory. It is a robust, production quality implementation that checkpoints a wide range of applications without requiring changes to application code. This work focuses into checkpointing parallel applications that communicate through MPI. BLCR support has been

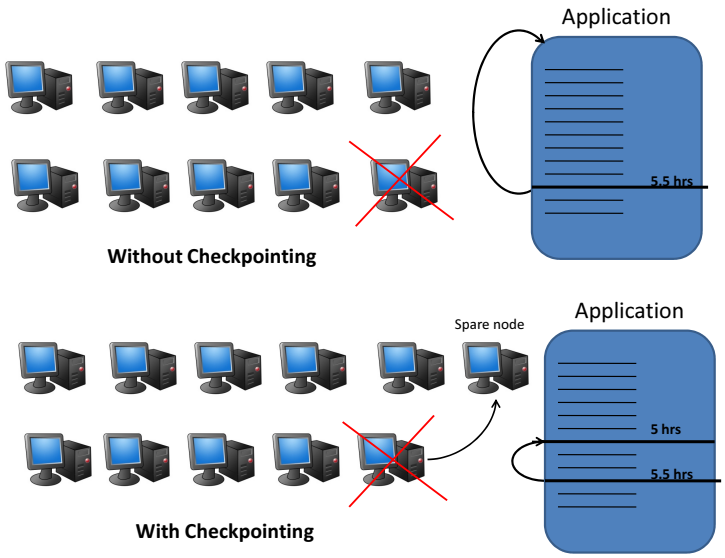

Figure 1.1: Application running with and without checkpointing

integrated on various MPI implementations namely, LAM/MPI, MVAPICH, OpenMPI etc. Researchers at North Carolina State University have been working on various extensions for BLCR [18][19][20].

1.5 Extensions to BLCR

The benefits of checkpointing are also accompanied by a certain amount of processing overhead and a significant amount of disk usage. Taking a checkpoint and restarting an application from a checkpoint incurs a certain amount of overhead. This overhead depends on various factors. Some of the important factors are size of application, disk I/O speed, bandwidth of the connecting medium in case the checkpoint file is being written to a remote destination. Similarly, disk usage is also an important consideration for checkpointing. It depends mainly on the size of the application and the frequency of checkpointing. Applications with higher checkpointing frequency tend to use up large amount of disk space.

To counter the above effects of checkpointing, several extensions to BLCR for optimization were designed by the systems research group at NCSU. The first extension is a rollback feature in BLCR which saves overhead by reusing existing resources at restart time. The second extension is incremental checkpointing. This extension aims to minimize checkpointing overhead as well as disk usage by checkpointing only the modified areas of application. We shall look at both the extensions in later sections.

In this thesis, we propose two different approaches for implementing incremental checkpoint-

ing. We propose an approach which makes use of internal Linux data structures to detect page modifications. We apply an additional patch [17] to enable access to underlying kernel data structure. We also propose a novel alternative approach that uses various in built mechanisms in the Linux kernel to implement incremental checkpointing obliviating the need to patch the kernel. We also propose a modular approach for the implementation of the two approaches that would allow the user to switch easily between the two different approaches. Last, we claim that it is possible to have save time during checkpointing by comparing the two different approaches in terms of their performances on various benchmarks.

Chapter 2

Motivation, Hypothesis and Contribution

In this chapter we present the motivation, Hypothesis, Contribution and Summary of our work.

2.1 Motivation

We have worked on several extension on the Berkeley Labs Checkpoint Restart (BLCR).

The first extension that we have worked on is process rollback. On any failure, BLCR by default kills all processes including the faulty process, then shifts the faulty nodes checkpoint image to a spare node and restarts the entire process. Doing this has several drawbacks. Firstly, we are killing of all the healthy processes in addition to the faulty process. Second, restarting the entire application causes it to be resubmitted to the job queue and get rescheduled again, which is a waste of time since other jobs in front of out resubmitted job in the scheduler queue will be executed first in case of a batch scheduler. Hence we implemented a rollback functionality in BLCR based on previous work by Chao Wang et al.[18]. This approach uses the current existing process containers and resources to just roll the applications on the healthy node to their previous checkpointed states and without ever killing them.

The checkpoint/restart process involves saving state of a process at a point in time. This includes registers, virtual address space, open files, debug registers etc. In case of a failure at any point in future, the process is recreated from the checkpointed data and the execution resumes from the last checkpoint rather than starting from the beginning. The na¨ve approach to checkpointing, however, checkpoints the entire process state at every checkpoint. However most applications spend their time between two checkpoints in a tight loop or some subsection of the process. Incremental checkpointing involves checkpointing only the data of a process which has been modified since the last checkpoint and ignoring the rest. This helps save disk space and

optimizes I/O bandwidth. The modification detection in this mechanism is currently performed at page level. This is done since the linux kernel itself maintains the modification information at page level granularity. The checkpointing process involves taking a full checkpoint at various intervals followed by a set of incremental checkpoints. We will discuss the implementation in the next section. We implement two separate approaches namely dirty-bit (DB) approach and write-bit (WB) approach. We finally conclude with future work and conclusion of analysis of both the approaches.

The third and last extension is the live migration feature. This extension detects the fault in the node and starts to checkpoint data to a socket to a spare node. A spare node can thus restore a checkpoint on-the-fly.

2.2 Hypothesis

We hypothesize that performance savings can be realized by utilizing the approach of "dirty bit" tracking for incremental checkpointing, which we suspect to be lower cost than alternative approaches, such as "write bit" tracking.

2.3 Contribution

The contributions of this work are twofold. First, it presents the design and implementation of two memory management schemes that enable incremental checkpointing. We describe unique approaches to incremental checkpointing that do not require kernel patching in one case and only require minimal kernel extensions in the other case. The work is carried out within the latest Berkeley Labs Checkpoint Restart (BLCR) as part of an upcoming release. Second, we evaluate the two schemes in terms of their system overhead for single-node microbenchmarks and multi-node cluster workloads. In short, this work is the final showdown between page write protection and dirty page tracking as a hardware means to support incremental checkpointing.

Chapter 3

Design & Implementation

This section describes the design of incremental checkpointing in BLCR. The main aim of the incremental checkpointing facility is to integrate it seamlessly with BLCR with minimal modifications to the original source code. The enhanced code should also have a minimal overhead while taking incremental checkpoints. When incremental checkpointing is disabled, it should allow BLCR to checkpoint without any additional complexity. For this purpose, we have divided the checkpoints into three categories:

1. Default Checkpointing - BLCR checkpointing sans incremental code;
2. Full Checkpointing - Fully checkpointing the entire process despite of any modifications;
3. Incremental Checkpointing - Checkpointing only the modified data pages of a process.

In the above list, Default and Full checkpointing would be identical in their output but different in their initialization of various data structures which is essential and shall be discussed later.

The main criteria of the design of incremental checkpointing code is to provide a modular approach. The most critical task in incremental checkpointing is to detect the modification of data pages in order to determine whether it should be checkpointed (saved) or not. Currently, we support two approaches. Based on previous work done at NCSU, the first approach is called the Dirty bit tracking approach or DB approach. The details of this approach are discussed below. This approach would require users to patch their kernels and recompile it. Another approach we designed avoids the patching of the kernel. It instead uses the currently existing mechanisms in the kernel to detect modifications to pages.

In addition to the above approaches, other solutions may be designed in the future depending on the features provided by the Linux kernel and the underlying hardware. To efficiently support different algorithms with minimal code modifications, we designed an interface object

for incremental checkpointing that unifies several of the essential incremental methods. Algorithms simply plug in their methods, which are subsequently called at appropriate places. Hence, BLCR remains agnostic to the underlying incremental checkpointing implementation. This interface needs to encompass all methods required for incremental checkpointing.

3.1 Incremental Interface

The incremental interface uses BLCR to call the incremental checkpointing mechanism in a manner agnostic to the underlying implementation. This enables various incremental checkpointing algorithms to be implemented without major code changes in the main BLCR module. The interface object is defined as :

```
int (*init)(cr_task_t *, void *);
int (*destroy)(cr_task_t *, void *);
int (*register_handlers)(cr_task_t *cr_task, struct vm_area_struct *map);
int (*page_modified)(struct mm_struct *mm, unsigned long addr, struct vm_area_struct *map);
int (*shvma_modified)(struct vm_area_struct *vma);
int (*clear_vma)(struct vm_area_struct *vma);
int (*clear_bits) (struct mm_struct *mm, unsigned long addr);
```

Figure 3.1: BLCR incremental object interface

With this object, existing BLCR code is converted to function calls. If they are not defined, BLCR will behave as it would without any incremental checkpointing. At the first checkpoint, this object would be created per process and associated with a process request. The high-level design is depicted in Figure 3.2.

The initialization function allows a specific incremental approach can use to set up the data structures (if any), initialize pointers etc. Similarly, the `destroy` function lets the specific module free up used memory and/or unregister certain handlers. The detection of modified data pages might utilize existing kernel handlers or hooks which need to be registered. The `register_handlers` function is used for registering specific callbacks. This function is utilized here to register callbacks for memory mapping and shared writes. The mmap callbacks keep track of mapping and unmapping of the memory pages to ensure that the newly mapped pages are not skipped as described in one of the cases.The `page_modified` function is the heart of this interface object. It returns a boolean value indicating whether the page has been modified or not. Similarly, `shvma_modified` returns a boolean for whether a shared page has been modified or not. After each incremental checkpoint `clear_vma` and clear bits can be used to reset the

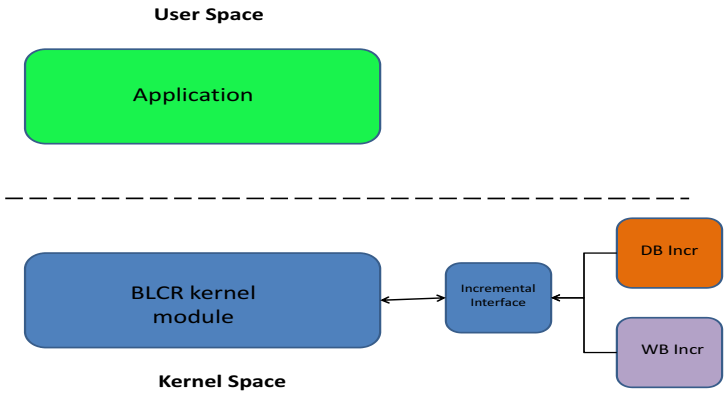

Figure 3.2: BLCR incremental design

bits for the next checkpoint.

3.2 Write bit design

The write bit approach is inspired by work by John Mehnert-Spahn et al. [13] . and tracks the modified data pages. However, they implemented their mechanism on Kerrighad Linux/SSI through source code modifications. One of the main criteria behind the design of this approach was to ensure that no modification of the kernel code was required. Therefore, in addition to the write bit, a number of other mechanisms were used for incremental checkpointing.

In this approach, the write bit is cleared at each checkpoint, and at the next checkpoint we check whether it is set or not. If the page whose write bit is cleared is written to, the Linux kernel generates a page fault. Since the segment permission for writing would be granted, the kernel will simply set the write bit of the associated pte and return. The write bit serves as an indicator that, if set, implies that the page was modified between checkpoints. If it is not set, the page was not modified between the checkpoints. However, this approach does not work for a number of corner cases.We shall look at those cases and discuss how this approach handles them in the following sections.

3.2.1 VM area changes

One of the major issues with the above write bit approach is its requirement to track changes in the virtual memory area. Many memory regions might be mapped or unmapped between two successive checkpoints. Some memory regions may get resized. We need to cover all such cases in order to ensure correctness.We have assigned a data structure for each page that tracks the status of the page. The structure and design of this tracking approach will be discussed shortly. Map tracking includes:

- A page getting unmapped:

  If a page is unmapped between two successive checkpoints, then the corresponding tracking structure for that page needs to be invalidated or removed. To this end, we need to be alerted when a page was unmapped while the process runs. We used the `close` entry provided in the vm_area structure, which is a callback initiated when a virtual memory area is being 'closed' or unmapped. With that hook, we associate required steps when a memory area is unmapped.

- New regions getting mapped:

  This case describes the instance in which new memory regions are added between two checkpoints. For example, consider an incremental checkpoint 1 written to disk. Before incremental checkpoint 2 is taken, page A is mapped into the process address space. At the next checkpoint, if page A was not modified it will not be checkpointed since the write bit would not be set. However, this would be incorrect. To handle this case, we do not allocate the tracking structure for newly mapped regions. Hence, at the next checkpoint on detecting the missing tracking structure, page A will be checkpointed regardless of the status of the write bit in its page table entry.

3.2.2 Corner Cases

One of the more serious cases is posed by the system call `mprotect`. For a vm area protected against writes, the kernel relies on the cleared write bit to raise page faults and then checks the VM permissions. This case can also give erroneous output. For example, assume page A was modified by the user thus setting the write bit. Before the next incremental checkpoint, the user protects the page allowing only reads, effectively clearing the write bit. When the next checkpoint occurs, the checkpoint mechanism fails to identify the modification on the data page and, hence, discounts it as an unmodified page. We have handled this case by using the dirty bit in this case. The `mprotect` function, while setting permission bits, masks the dirty bit. Hence, if the page is modified then we can detect it through the dirty bit.

The other corner case is that of shared memory. In BLCR, only one of the processes will capture shared memory. However, we may miss the modification if the process capturing the shared memory has not modified the data page. To handle this, we reverse map the processes through the cr_task structures and check for modifications in each process tracking structure for the page. If at least one of them is modified, then the shared page is dirty and should be checkpointed.

3.2.3 Tracking Structure

In the WB approach that we have described, we have implemented an approach that does not require patching and recompiling the kernel. Instead it uses the existing resources. The corner cases clearly show that in cases like un-mapping and re-mapping pages or detecting modifications on a shared page, our mechanism, by itself, fails.There is a need for BLCR to maintain the information regarding the data pages of the process. This means that while traversing through the vma, BLCR needs to check certain parameters of the specific page, which will help BLCR to properly handle corner cases with those pages.

To design the page tracking structure, we decided to look at the paging system inside Linux. We have seen before that the memory address space of a process is divided into pages, each of which is of a fixed size. The pages are then accessed on an on-demand basis. Let us take the 32-bit x86 architecture as an example. Normally, the entire process address space is fragmented with very few processes using up the entire address space. If the entire metadata or page information were stored as a single table, it would take up 4 MB of main memory space. To avoid allocation of entire page table at once, the page table structure is divided into multiple levels. In a 32 bit x86 system, the paging mechanism works as shown in Figure 3.3

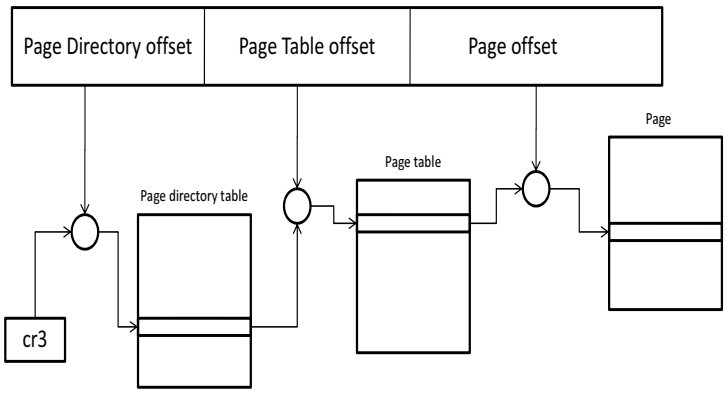

Figure 3.3: Page table structure in Linux

In x86 systems, the cr3 register holds the value of the start of the page directory table (PGDT) for a particular process. When an address is accessed, the bits from bit 22-31 in the address value represent the offset into the PGDT. The value obtained at the specified offset is the start address of the corresponding page table. The bit values from bit 12-21 in the accessed address represent the offset into the particular page table. The value obtained at this level is the Page Table Entry(PTE), which contains the frame number for the main memory and some meta information about the page(access rights, modification information etc).

After the introduction of 64 bit systems, two more levels of paging were introduced in the linux paging system to accommodate the larger memory address space. For 32 bit x86 systems the middle two levels are just ignored.

The page tracking structure in BLCR is modeled after the page table. Each requested object by a process inside BLCR will contain a `cr_cr3` variable, which, like its counterpart, will hold the beginning address of the pseudo page tables for the process in BLCR. The page table levels are designed similar to the Linux paging system. Initially, the design was to replicate the entire page table structure in BLCR to maintain the state of each page. However, the difference here is in the last level of the page hierarchy. In the Linux page table structure, the last level holds the PTE for each page. The PTE is a variable of type `long` and it holds the information about the frame number in the main memory and other information about the page such as access rights. For the tracking structure in BLCR, we only need to maintain information regarding (a) whether the page was un-mapped since last checkpoint; and (b) whether the page was written by another process (in case of shared memory). Since a page tracking structure will have to be created for each process,using a `long` type variable would waste a significant amount of memory. We have optimized the "PTE" in the tracking structure to use only 4 bits per page, i.e., the final level in page table hierarchy will use only 4 bits to maintain the information about the page. One bit will be used to keep track of the mapping of the page (whether it is mapped on unmapped). The second bit is set whenever any process in a process group writes to a specific page. Two more bits are reserved for future use. So the total "PTE" size for each page is compressed to 4 bits.This results in an almost 8 times reduction in memory usage compared to maintaining a properly mirrored page table.

The tracking structure for incremental checkpointing is a virtual page table maintained by the BLCR module. This is done for two purposes: 1) to maintain track of vm area changes like un-mapping, re-mapping, new mapping etc; and 2) to detect writes to shared memory. Let us have a look at how the page tracking structure solves our corner case problems.

Case (a): The page is modified, un-mapped and re-mapped between two checkpoints. Since the page is un-mapped and re-mapped, there is a possibility that we can lose the modification bit since the re-mapped page will have its write bit cleared. This is where our tracking structure helps us. In the first checkpoint, knowing that the page is mapped, the MAPPED bit of the tracking structure will be set. Then when the page is un-mapped, the `close` callback of the `vm_area_struct` is called in which we clear the MAPPED bit for that page in the BLCR tracking structure. Before the next checkpoint, a page may be mapped in to the same address. At the next checkpoint, while detecting the modification on the page, we will also check the MAPPED bit. Since the MAPPED bit was cleared when the previous page was un-mapped, BLCR takes this page as a new page without any incremental considerations and checkpoints it.

Case (b): A shared page is modified by one of the processes in the group. In BLCR, only the

first process in the process group will dump the shared memory in order to remove redundancy of the data. If process A modified the page and process B tries to dump it, it can miss the modification from process A and reject the page as unmodified. With our tracking structure, when process A modifies a shared page, the callback `shared_mkwrite` is called and BLCR sets the SHARED bit in the tracking structure for that page. When process B is dumping the shared pages, it will check the SHARED bit for that page in the tracking structure of each process in the process group. This is done by using the reverse mapping technique. In this case, process B will come across the set bit in process A's tracking structure for the page and store it because of the modification.

3.3 Dirty Bit Approach

The second approach taken by previous work uses the dirty bit for detecting page modifications. It uses an existing Linux kernel patch [17] to copy the PTE dirty bit into user level. The problem with using the dirty bit is that the kernel uses the dirty bit for its own purposes. This introduced an inconsistency if BLCR and the kernel were both using it simultaneously. The patch introduces redundant bits by using free bits in the PTE and maintaining a correct status of the dirty bit for a given page. This approach requires the kernel to be patched. More significantly, this approach prevents page faults from being raised at every write as in the write bit approach.

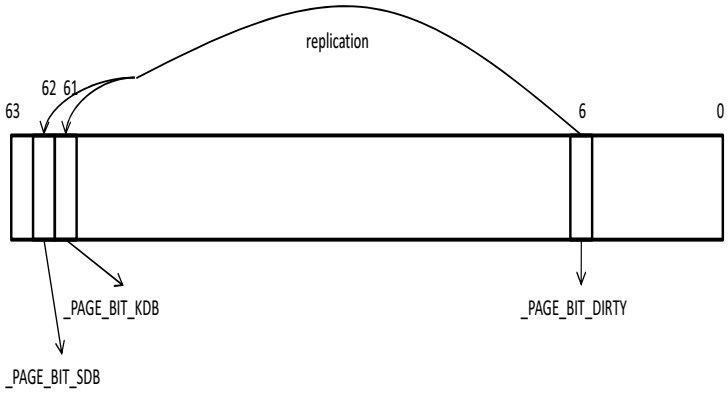

Figure 3.4: Dirty bit approach

As shown in Figure 3.4, Linux uses the dirty bit inside the PTE to maintain the modification status of the page. We observe that a two of bits in the PTE remain unused. The patch uses this bits to replicate and maintain the dirty bit consistently. The two bits are name _PAGE_-

BIT_KDB (to maintain the kernel state of dirty bit) and _PAGE_BIT_SDB (to maintain user state of the dirty bit). KDB maintains the consistency of the dirty bit to the kernel and returns the value of dirty bit with respect to the last access by the kernel. SDB bit maintains the dirty bit between two different function calls. The function call `cr_db_page_status` returns the value of the dirty bit relative to the last call to the function. The dirty bit is reset at each function call.

3.4 Restoring Incremental Checkpoints

Restarting is the latter half process of the C/R process. It involves restarting the process from the context file written during the checkpoint process. BLCR stores all the system information about the process and "recreates" the entire process at restart time. When a checkpoint is taken, the entire information is stored in a specific file format that can be retrieved at the time of restart. Examples of such information include the pages from the process address space, registers, stack pointers, pending signals, register dumps of various threads etc. Let us see the checkpoint file format in detail. Figure 3.5 shows the file format of the checkpoint context file:

When restarting, the `cr_restart` process forks off a child. It then reads the information from the context file in parts and tweaks the characteristics of the child process to match the checkpointed process and will eventually restart the checkpointed process.

While this mechanism works for normal restarts, restarting a process for incremental checkpoints is not so simple. The problem is that not all context files from various incremental checkpoints will have all the information about the process. Starting from a specific incremental checkpoint is difficult since there will be "holes" in the incremental context files where the data was not modified in the interval between the previous checkpoint and the current one. Restarting a process from a single, incomplete context file would yield erronous results. The incremental checkpoints are characterized by their completeness and interval. The incremental checkpointing mechanism will involve a full checkpoint taken at several intervals and incremental checkpoints taken in between. This is called the incremental checkpoint ratio. For example, if the ratio is 1:5, then one full checkpoint will be taken after every 5 incremental checkpoints. It is also clear that to restart an application from any of the incremental checkpoints would be difficult since they would have memory "holes" in them. The correct way would be "stack" the checkpoints from the last full checkpoint up to the current incremental checkpoint and start restoring everything from the full checkpoint, overwriting the parts from subsequent incremental context files as they are processed.

The process of restarting from an incremental checkpoint is more complex than starting from a normal checkpoint. The above described process would involve a change in the context file format so that multiple context files can be overlaid without any inconsistencies. This

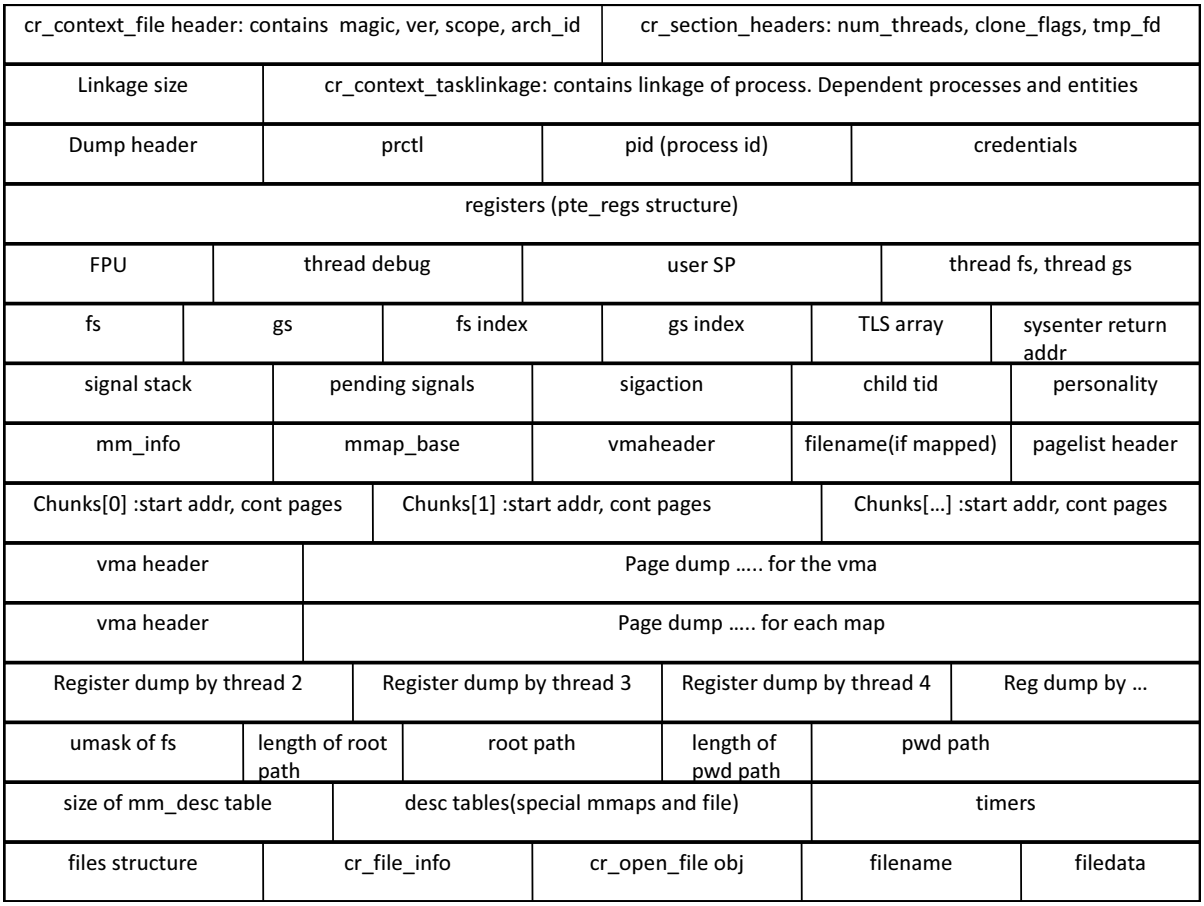

Figure 3.5: Context file format for BLCR

approach is being implemented and is expected to be integrated soon with BLCR.

3.5 Interface to Use Incremental Checkpointing

Along with the implementation of the incremental checkpoint, we needed to design a command line interface to control the incremental checkpointing by the user. The normal BLCR interface provides no such option for checkpointing. Running a `cr_checkpoint` call on a process just checkpoints the entire process. Similarly, for restart, there are no options to specify what kind of context file (full or incremental) is being restarted. We designed the interface with following options:

a) cr_checkpoint -i -pid [pid]

Any checkpoints with the flag -i are considered as incremental and will be checkpointed incrementally. Without the -i flag, a full checkpoint will be taken.

b) cr_checkpoint -p period -pid pid

Here the -p flag indicates the period or the interval of incremental checkpointing. The 'period' argument will be passed to indicate the number of incremental checkpoints to take between two full checkpoints. This is probably not a preferred approach. BLCR doesnt care what is the optimal frequency of incremental checkpointing is. BLCR will only be told whether the current checkpointing is full or incremental.

c) cr_checkpoint –full-incr [method] -pid pid

This is probably the best way to implement the flags. Here we pass the method type [write bit/ dirty bit] along with full incr flag at the start of a set of checkpoints(This also means we can switch methods between two successive sets of checkpoints, even in the same process run.)

3.6 Rollback Functionality

The rollback functionality [18] is designed to cut the overhead in the BLCR checkpoint/restart process. To understand this, let us consider an example. We have a parallel processes running on 10 nodes. Assume that one of the node fails. With our fault tolerance policy, the latest checkpoint taken by BLCR will be transferred to the spare node. BLCR will then kill all the processes, including the healthy ones, and then restart the entire process on all nodes from the last checkpoint. The rollback functionality cuts the overhead of killing the process and resubmitting the job to the job queue. The implementation was done by Wang et al. in the older version of BLCR(ver 0.4.2). We implemented the rollback functionality in the new BLCR(version 0.8.0). The rollback functionality is now a part of the BLCR cvs head and can be accessed by passing the '-r' switch with the cr_restart command.

Chapter 4

Experiments

4.1 Framework

We conducted our performance evaluations on a local cluster that we control. This cluster has 18 compute nodes running Fedora Core 12 Linux x86_64 (Linux kernel- 2.6.31.9-174.fc12.x86_-64) connected by two Gigabit Ethernet switches. Each node in the cluster is equipped with four 1.76GHz processing cores (2-way SMP with dual-core AMD Opteron 265 processors) and 2 GB memory. A large RAID5 array provides shared file service through NFS over one Gigabit switch. Apart from the modifications for incremental checkpointing in BLCR, we also instrumented the code for the BLCR library to measure the time across checkpoints. OpenMPI was used as the MPI platform since BLCR is tightly integrated in its fault tolerance module.

4.2 Experiments

We designed a set of experiments designed to assess the overhead and analyze the behaviour of two different approaches of incremental checkpointing, namely (i) the WB approach and (ii) the DB approach. The experiments are aimed at analyzing the performance of various test benchmarks for these two approaches in isolation and measuring their impact on NAS parallel benchmarks in terms of performance of application benchmarks.

Various NPB benchmarks [7] as well as microbenchmarks have been used to evaluate the performance of the above approaches. From NPB benchmark we chose SP, CG, LU as their runtimes are long enough for checkpoints. We have performed those experiments on Class C of each benchmark as the memory footprint for Class C inputs is suitable for our experiments. In addition, we have designed a microbenchmark which varies memory consumption in order to evaluate the performance of the incremental approaches with scaling memory. We also have a designated set micro benchmarks that evaluate the effect of special cases like mprotect and

memory swapping with dirty bit usage among others. The benchmarks are tested for various parameters like varying memory, modified memory, ratio of incremental checkpoints etc.

4.2.1 Instrumentation Techniques

For getting precise measurement of time, the method of instrumentation is quite important. The BLCR framework has been modified to record timings at two levels. The figure shown below (XXX fig to be done) shows the block diagram of an MPI program since we are going to use the NPB benchmarks. As we see, we can issue an ompi-checkpoint command so that the OpenMPI framework performs a coordinated checkpoint. One option is to measure the time across the ompi-checkpointing call. However, this required modification to the Openmpi code and it will also include the timing for coordination of various MPI processes, which will effect our results. We decided, in this case, to modify the BLCR library instead. We measure the timing across the do_checkpoint call in each of the processes. The processes then output their timing to a common file on the NFS where you can see each node's time performance regarding checkpoints is recorded. This design is shown in the figure below (XXX fig to be done)

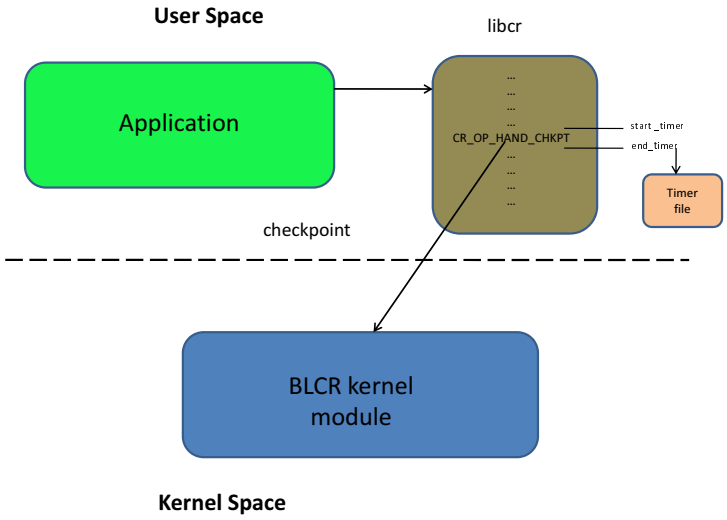

Figure 4.1: BLCR library timer design

There is one small caveat with the above approach. Our initial tests showed very low variation between the two incremental approaches we were comparing. After studying timings for various phases, it was found that most of the checkpoint time was dominated by writes to the context file on the file system. This would cover up any other time like the time taken to detect page modifications. So our approach was to find a way to eradicate the write time from

the total time and measure only the time to detect modifications on the pages. For this, we had to change the BLCR kernel module to measure only the modification commands. The design is shown in Figure 2. We accrue the timing measurements for modification detection across each page chunk. As a post processing step, we calculate the maximum, minimum and average of all checkpoint timings.

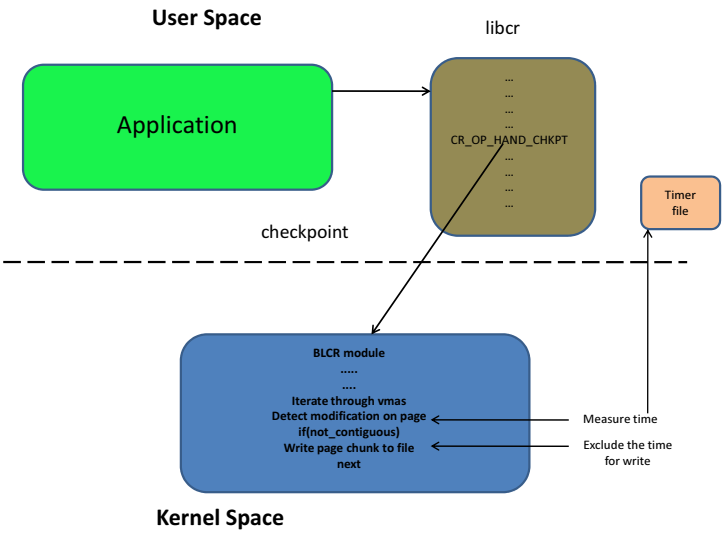

Figure 4.2: BLCR kernel module timer design

Automated checkpoint scripts and use of crut frameworks enables the regular checkpointing of various MPI and non-MPI processes.

4.2.2 Memory Test

We have split the test suite into two parts. The first part, the memory test, is to actually measure the difference between two checkpointing approaches on a single machine. The second part of experiments measures the impact of performance on multi-node MPI benchmarks as the number of nodes and the memory consumption scales.

We will be discussing the first set of experiments in this section. We have written a microbenchmark for measuring the performance difference between the two approaches namely WB and DB. This benchmark allocates a specified number of memory pages and, depending on the configuration specified, it will alter a certain number of pages per checkpoint. This will help us in comparing the performance with scaling memory.

The test is conducted on a large data set. In this test, we map out 200k memory pages in our process. We will keep the number of checkpoints at 20 and the ratio of incremental to full

at 4:1. We will vary the modified pages but here we shall vary them by large increments. The data points for this graph are at 500, 5k, 25k, 50k, and 100k pages modified. The results are presented in Figure 4.3.

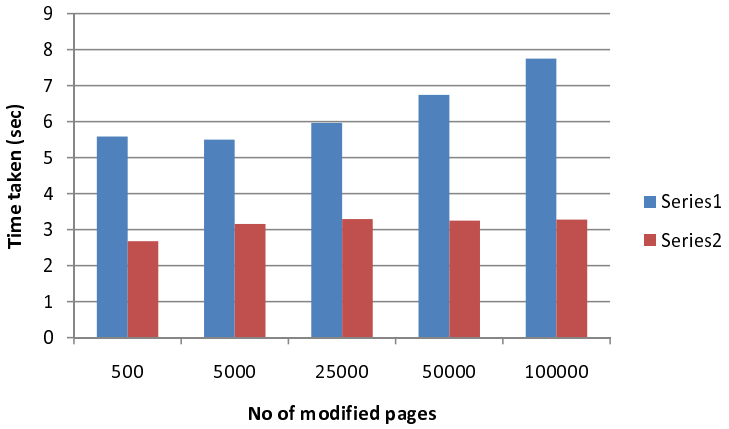

Figure 4.3: Micro benchmark Test

We make an interesting observation. We see that in initial cases when the set of modified pages is low, the difference between the performance of dirty bit and write bit is low. As the number of pages modified keeps increasing, the difference also keeps increasing. When the modified data set is 100k pages, the difference is almost twice that of WB. We can conclude from this experiment that using the dirty bit approach will improve performance.

To understand this result, let us understand the mechanism first. BLCR iterates through every page and tries to check for modified pages. For each page, the WB and DB approach will use their own mechanisms for checking modified pages. In the WB approach, BLCR has to check its own data structure for mappings of the page (mapped or not). It then fetches the PTE from the address passed to it. After detecting whether page has been modified or not, the clear_bit function clears the write bit in the PTE for the next round. For this, the WB approach has to map the PTE again to access it. In DB, on the other hand, the testing for modification and clearing the bit on the PTE happens in a single step within the test-and-clear function. In addition to it, the DB approach does not have to manipulate any internal data structures to keep track of mappings. These factors make DB a much faster approach then WB in the above experiment.

The third case gives us some insight into the comparison of incremental with a default full checkpoint. In this case, we decided to modify alternate pages from the process address space and observe the performance. We have a data set of 100k pages here. With every alternate

page, 50k pages will be modified between each checkpoint. On collecting the checkpoint timings we observe another surprise. The incremental checkpoint takes longer than a full default checkpoint. While it seems counter intuitive that saving a set of smaller incremental data takes more time than saving the full checkpoint data, the answer to it lies in the way BLCR saves the memory space. The BLCR iterates through each vma trying to find contiguous chunks of pages before it issues the write to the file and hence to the disk. When the full checkpoint happens, the entire mapped space is committed as one chunk and hence in one write to disk. When we modify alternate pages, we will encounter an unmodified page after each modified page, which will be discarded by BLCR since its unmodified. Since the chunk breaks after each page, BLCR will issue a write to disk for each single modified page. Therefore we observe a significantly more writes in incremental checkpointing than in full checkpointing. As far as the comparison between the approaches is considered, we see the familiar savings in the DB approach compared to the WB approach.

In this section, we will discuss the effect of multi-node MPI benchmarks on the performance of the write bit and the dirty bit approaches. We have selected the MPI version of the NPB benchmark. Our selection of the benchmarks from the NPB set depends on the running time of the benchmark since we want the benchmark to run long enough to take a suitable number of checkpoints.

We found that for smaller input sizes (lower classes) of benchmarks, on increasing the number of processors, we are also decreasing its runtime, which renders them unusable for our experiments. On the other hand, using very large input sizes (like Class D) for NAS benchmarks on smaller number of processors (1 or 4) is not very practical for taking checkpoints because of its long running time. So we decide to vary our input class according to the number of processes that we run it on.

After running all benchmarks and observing their runtimes, we found three suitable benchmarks that can be used for our tests. They are SP, CG and LU. We present the following experiments and results for the same.

We ran the benchmark on 4, 9, 16, and 36 (since SP requires number of processors to be a perfect square) processors for SP. The benchmark tests were performed on Class C inputs for the SP benchmark with a checkpoint interval of 60 seconds on varying number of nodes. We observe that the DB approach incurs less overhead in the kernel than the WB approach in all of the cases. We see a downwards slope and a decrease in the difference between DB and WB from 4 processors to 9 processors to 16 processors. The reason for the decrease in time spent in the kernel is that as we increase resources the application is more distributed among nodes. This translates into less amount of data for a single node to checkpoint and, hence, less amount of time spent for checkpointing in the kernel. In the case of 36 processors, we see a sudden spike in the time spent. This is an abberation since according to the pattern we should have a

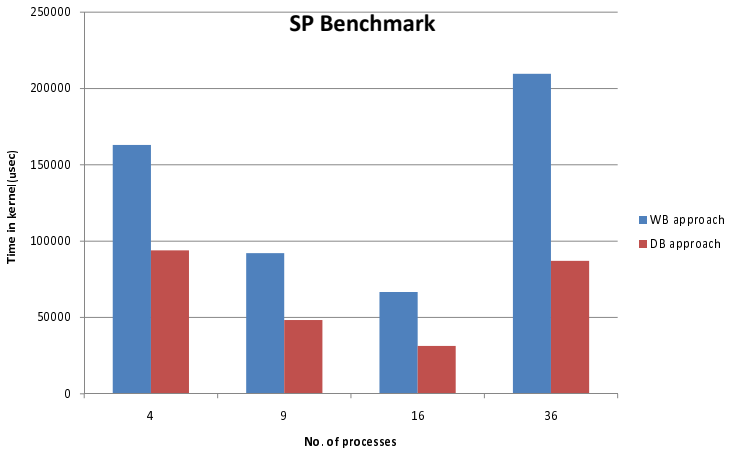

Figure 4.4: SP benchmark

decrease in time spent. This can be attributed to the fact that although we are running the SP benchmark for 36 nodes, we only have 16 physical nodes. Thus, multiple processes are vying for resources on the same node. The processes are contending for cache, networking or other resources.

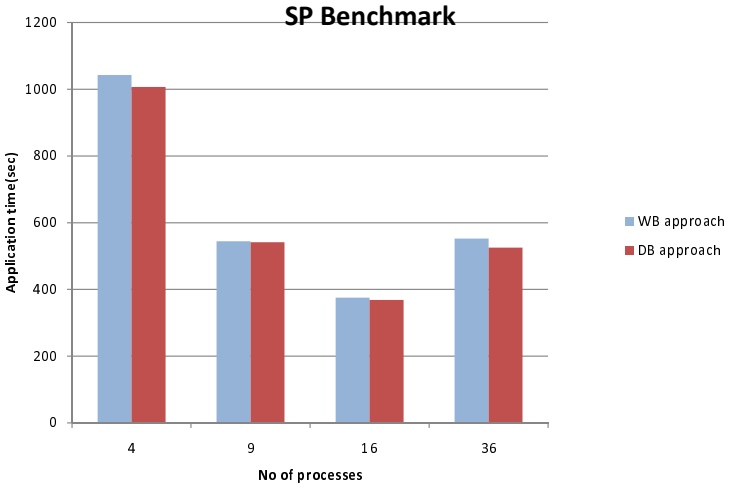

Figure 4.5: SP benchmark (Application time)

Figure 4.5 shows the overall application time for the SP benchmark for different numbers of nodes. We see that the dirty bit approach outperforms the WB approach in all cases for the SP benchmark and incurs less overhead. As the number of processes (and simultaneously processors) increases from 4 to 9 to 16, we see a decrease in total application time. As the number of

processors increases, the work gets distributed between various nodes and the application time reduces. This happens in both the cases (WB & DB). For 32 processes, the application time goes up instead of down. Again, we are running the 32 processes on 16 physical nodes. This causes contention for memory, networking and other resources.

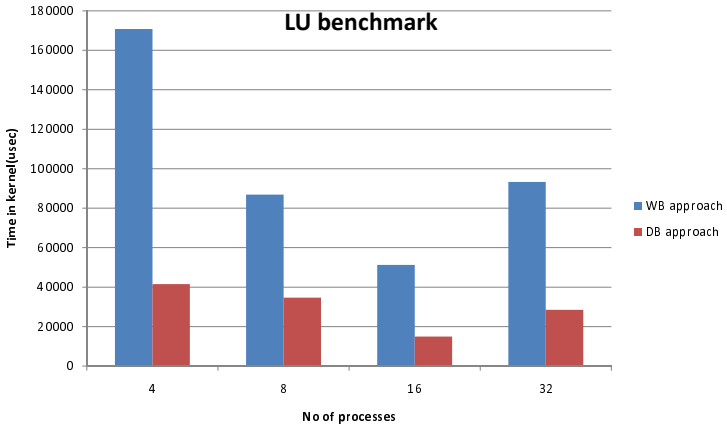

Figure 4.6: LU benchmark

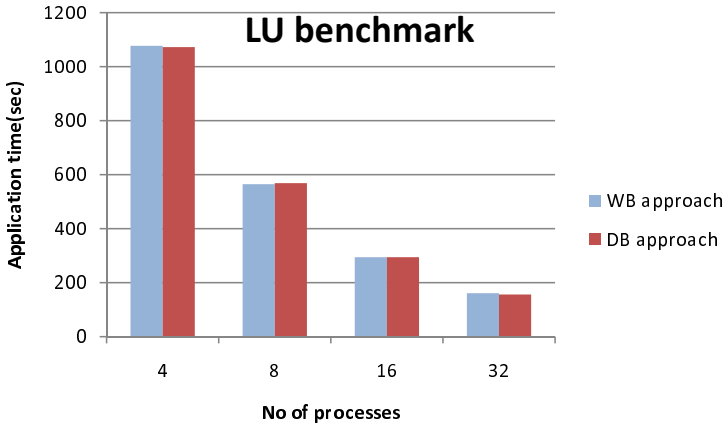

Figure 4.7: LU benchmark (Application time)

The Figure 4.6 and Figure 4.7 shows kernel timings and total application time for the LU benchmark. The benchmark tests were conducted for Class C inputs of the LU benchmark with a checkpoint interval of 45 seconds on varying number of nodes. As on the previous experiments, we see significant savings in time spent in kernel. The total time goes on decreasing as we

increase the nodes up to 16 nodes. With 32 nodes, we have the same behavior as for SP benchmark. The total time taken for 32 nodes goes up. This is due contention for memory, networking and other resources cause due to running 32 processes on 16 physical nodes.

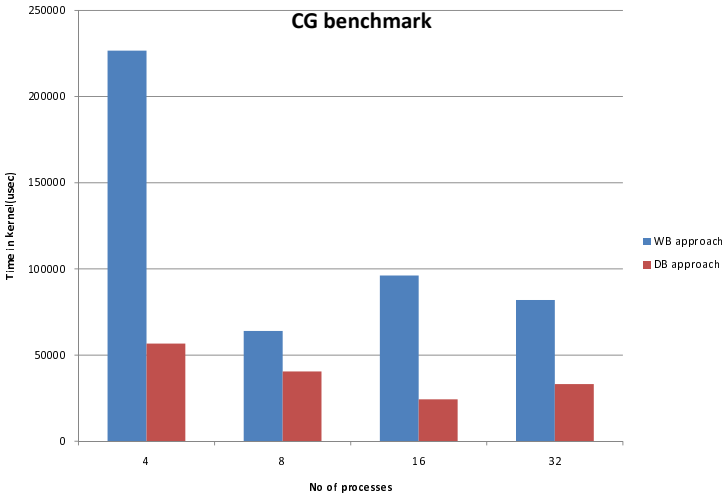

Figure 4.8: CG benchmark

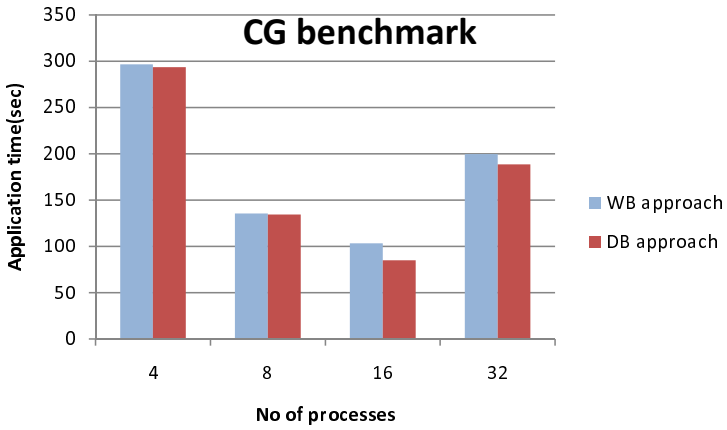

Figure 4.9: CG benchmark (Application time)

Figure 4.8 shows the time spent in kernel for the CG benchmark. The benchmark tests were conducted for Class C inputs of the CG benchmark with a checkpoint interval of 10 seconds on varying number of nodes. For 4 processors, we see significant savings of the DB approach over

the WB approach. We also observe that this graph has an anomaly for 16 processes (on 16 nodes). For the 16 processes run, we see that the difference between the WB and DB approach is quite high. This leads us to an interesting discovery. The total running time of CG is low. Hence, the checkpoints were taken at an interval of 10 seconds. Since the savings in the dirty bit approach were more than ten seconds, the benchmark run with the DB approach had to take fewer checkpoints, which further increases the amount of time saved. When the time saved on the total application time increases beyond the checkpoint interval, then we gain more savings from the fact that fewer checkpoints will be taken for the given process. This fact is also highlighted in Figure 4.9, which shows the total application time for CG, where we see considerably more savings in the case of the 16 node/16 process run.

Chapter 5

Related Work

5.1 Related work

C/R techniques for MPI jobs frequently deployed in HPC environments can be divided into two categories: coordinated (LAM/MPI+BLCR [15, 8], CoCheck [16], etc.) and uncoordinated (MPICH-V [5, 6]). Coordinated techniques commonly rely on a combination of OS support to checkpoint a process image (e.g., via the BLCR Linux module [8]) or user-level runtime library support. Collective communication among MPI tasks is used for the coordinated checkpoint negotiation [15]. Uncoordinated C/R techniques generally rely on logging messages and possibly their temporal ordering for asynchronous non-coordinated checkpointing, e.g., MPICH-V [5, 6] that uses pessimistic message logging. The framework of OpenMPI [4, 12] is designed to allow both coordinated and uncoordinated types of protocols. However, conventional C/R techniques checkpoint the entire process image leading to high checkpoint overhead, heavy I/O bandwidth requirements and considerable hard drive pressure, even though only a subset of the process image of all MPI tasks changes between checkpoints. With our incremental C/R mechanism, we mitigate the situation by checkpointing only the modified pages.

Incremental Checkpointing: Recent studies focus on incremental checkpointing [10, 11]. TICK (Transparent Incremental Checkpointer at Kernel Level) [10] is a system-level checkpointer implemented as a kernel thread. It supports incremental and full checkpoints. However, it checkpoints only sequential applications running on a single process that do not use inter-process communication or dynamically loaded shared libraries. In contrast, our solution transparently supports incremental checkpoints for an entire MPI job with all its processes. Pickpt [11] is a page-level incremental checkpointing facility. It provides space-efficient techniques for automatically removing useless checkpoints aiming at minimizing the use of disk space. This differs from our thread based garbage collection technique. Yi et al. [21] develop an adaptive page-level incremental checkpointing facility based on the dirty page count as a

threshold heuristic to determine whether to checkpoint now or later, a feature complementary to our work that we could adopt within our scheduler component. However, Pickpt and Yis adaptive scheme are constrained to C/R of a single process, just as TICK was, while we cover an entire MPI job with all its processes and threads within processes. Agarwal et al. [3] provide a different adaptive incremental checkpointing mechanism to reduce the checkpoint file size by using a secure hash function to uniquely identify changed blocks in memory. Their solution not only appears to be specific to IBMs compute node kernel on BG/L, it also requires hashes for each memory page to be computed, which tends to be more costly than OS-level dirty-bit support as caches are thrashed when each memory location of a page has to be read in their approach. A prerequisite of incremental checkpointing is the availability of a mechanism to track modified pages during each checkpoint. Two fundamentally different approaches may be employed, namely a page protection mechanism or a page table dirty bit approach. Different implementation variants build on these schemes. One is the bookkeeping and saving scheme that, based on the dirty bit scheme, copies pages into a buffer. Another solution is to exploit page write protection, such as in Pickpt, to save only modified pages as a new checkpoint. The page protection scheme has certain draw-backs. Some address ranges, such as the stack, can only be write protected if an alternate signal stack is employed, which adds calling overhead and increases cache pressure.

We have presented two different approaches in this work. The first approach we present exploits the write bit to detect modifications on a page level. This approach does not require the kernel to be patched. This is different than the prior works since it uses innovative approaches to handle corner cases for detecting modifications on pages. The second approach uses the dirty bit for tracking writes on page. This approach shadows the dirty bit from the kernel within the user level and captures the modification status of the page. Both our approaches work for entire MPI jobs.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This work presents two different approaches for implementing incremental checkpointing. The DB approach makes use of an existing patch to shadow the dirty bit within the user level. We also present a novel, alternate WB approach that uses the existing mechanisms in the kernel to implement incremental checkpointing without the need to patch the kernel. We also compared the performance between the two approaches and showed the tradeoff between time vs. convenience in choosing one of the two approaches. We have also shown the results of running both the approaches on our microbenchmarks and NPB benchmarks and comparatively analyzed the performance of both WB and DB approaches.

6.2 Future Work

There is, however, a lot of scope for future work related to this thesis. Some of the important areas for the future work are discussed below:

6.2.1 Restart Algorithm for Incremental Checkpointing

We have described two approaches for incremental checkpointing. The restart algorithm for restarting incremental checkpoints will be very different from the general restart algorithm in BLCR. Since the data is dispersed in incremental checkpoint context files, the restarting algorithm for incremental checkpoints will also involve making changes in the context file format. The algorithm for restarting incremental checkpointing will need to be fast and efficient since it will have more than one file to process. This algorithm is as of yet un-implemented and is part of the future work.

6.2.2 Integration with OpenMPI

The incremental checkpointing implementation is designed to work with multi-node and multi-core processes. OpenMPI processes can be incrementally checkpointed using our current implementations in BLCR. However, as of now, incremental checkpointing has not been integrated into OpenMPI. The user is required to run a background script to trigger checkpoints of MPI processes and would be unable to pass parameters for incremental approaches implemented in BLCR. A tight integration of the BLCR incremental interface with OpenMPI to enable parameter passing through the OpenMPI checkpoint utilities as well as the OpenMPI checkpointing APIs would aid both OpenMPI and BLCR.

6.2.3 Live Migration

Another key feature in future work is the live migration feature. Live migration is a pro-active fault tolerance technique. We utilize data inputs about a node's health from the Intelligent Platform Monitoring Interface (IPMI) to detect degrading health of a node. Once hardware degradation is detected on any of the nodes, a degrading node starts to checkpoint its application. Instead of writing out to a disk and then copying its data to a spare node, live migration writes the checkpoint to an outgoing port, connected to a spare node, on-the-fly. The spare node reads the incoming checkpoint data from the port and recreates the process on-the-fly. The live migration technique intends to improve performance by saving the overhead of writing to disk. The implementation of live migration is future work.

As a part of the future work, we would also like to test our current incremental checkpointing approaches on bigger systems with a large number of nodes to study scalability.

REFERENCES

[1] Top 500 list. http://www.top500.org/, June 2002.

[2] Fault tolerance techniques for distributed systems. http://www.ibm.com/, July 2004.

[3] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 277–286, New York, NY, USA, 2004. ACM.

[4] B. Barrett, J. M. Squyres, A. Lumsdaine, R. L. Graham, and G. Bosilca. Analysis of the component architecture overhead in Open MPI. In *European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.

[5] G. Bosilca, A. Boutellier, and F. Cappello. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing*, November 2002.

[6] Bouteiller Bouteiller, Franck Cappello, Thomas Herault, Krawezik Krawezik, Pierre Lemarinier, and Magniette Magniette. MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *Supercomputing*, 2003.

[7] Weeratunga D., Barscz E., Barton J., Browning D., Carter R., Dagum L., Fatoohi R., and Fineberg S. The nas parallel benchmarks. Technical report, NASA Ames Research center, 1994.

[8] J. Duell. The design and implementation of berkeley lab's linux checkpoint/restart. Tr, Lawrence Berkeley National Laboratory, 2000.

[9] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[10] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Supercomputing*, 2005.

[11] Junyoung Heo, Sangho Yi, Yookun Cho, Jiman Hong, and Sung Y. Shin. Space-efficient page-level incremental checkpointing. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1558–1562, New York, NY, USA, 2005. ACM.

[12] Joshua Hursey, Jeffrey M. Squyres, and Andrew Lumsdaine. A checkpoint and restart service specification for Open MPI. Technical report, Indiana University, Computer Science Department, 2006.

[13] Michael Schoettner John Mehnert-Spahn, Eugen Feller. Incremental checkpointing for grids. In *Linux Symposium 2009*, July 2009.

[14] E. Roman. A survey of checkpoint/restart implementations. Technical report, Lawrence Berkeley National Laboratory, 2002.

[15] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium*, October 2003.

[16] G. Stellner. CoCheck: checkpointing and process migration for MPI. In IEEE, editor, *Proceedings of IPPS '96. The 10th International Parallel Processing Symposium: Honolulu, HI, USA, 15–19 April 1996*, pages 526–531, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[17] Luciano A. Stertz. Readable dirty-bits for ia64 linux. Internal requirement specification, Hewlett-Packard, 2003.

[18] C. Wang, F. Mueller, C. Engelmann, and S. Scott. A job pause service under LAM/MPI+BLCR for transparent fault tolerance. In *International Parallel and Distributed Processing Symposium*, April 2007.

[19] C. Wang, F. Mueller, C. Engelmann, and S. Scott. Hybrid full/incremental checkpoint/restart for mpi jobs in hpc environments. Technical Report TR 2009-14, Dept. of Computer Science, North Carolina State University, 2009.

[20] C. Wang, F. Mueller, C. Engelmann, and S. Scott. Proactive process-level live migration and back migration in hpc environments. Technical Report TR 2009-15, Dept. of Computer Science, North Carolina State University, 2009.

[21] Sangho Yi, Junyoung Heo, Yookun Cho, and Jiman Hong. Adaptive page-level incremental checkpointing based on expected recovery time. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1472–1476, New York, NY, USA, 2006. ACM.