Comparing different approaches for Incremental Checkpointing: The Showdown

Manav Vasavada, Frank Mueller
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7534
e-mail: mueller@cs.ncsu.edu

Paul H. Hargrove, Eric Roman
Future Technologies Group
Lawrence Berkeley National Laboratory
Berkeley, CA 94720

Abstract

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. Today's 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 bit (WB) protection and dirty bit (DB) page tracking as a hardware means to support incremental checkpointing. Our results show savings of the DB approach over WB approach in almost all the tests. Further, DB has the potential of a significant reduction in kernel activity, which is of utmost relevance for proactive fault tolerance where an immanent fault can be circumvented if DB-based live migrations moves a process away from hardware about to fail.

1 Introduction

With the number of cores increasing manifold at a rapid rate, high performance computing systems have scaled up to thousands of nodes or processor cores. Also, with the increase in the availability of off-the-shelf components, parallel machines are no more a niche market. Huge scientific applications and even non-scientific applications with highly parallel patterns exploit such machines and, hence, provide faster time-to-solution. Even with the high amount of processing power available, such high-end applications experience execution times in the order of hours or even days in some cases. Examples of such applications are general scientific applications, climate modeling, protein folding and 3D modeling. With the use of off-the-shelf components, the Mean Time Between Failure (MTBF) has also been reduced substantially [12], which indicates an increasing probability of hardware failure on such machines. After a failure, the current process would need to be restarted from the scratch. This approach would not only waste CPU cycles and power in duplicated work but also delay the results by a substantial amount of time. To address these problems, fault tolerance is needed.

There have been many approaches to support fault tolerance in HPC. One of the approaches is checkpoint/restart (C/R). This approach involves checkpointing the application on each node at regular intervals of time to non-local storage. Upon failure, the checkpoint is simply shifted to a spare node and the checkpoint

is restarted from the last checkpoint instead of restarting the application from the scratch. 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.

The checkpoint restart framework which this paper revolves around is Berkeley Labs Checkpoint Restart (BLCR) [6]. BLCR is a hybrid kernel/user implementation of C/R for Linux developed by the Future Technologies Group at Lawrence Berkeley National Laboratory. It is a robust, production quality implementation that checkpoints a wide range of applications without requiring any changes made to the code. The checkpoint process involves saving the process state including registers, virtual address space, open files, debug registers etc., and using the data to restart the process. BLCR support has been tightly integrated into various MPI implementations like LAM/MPI, MVAPICH, OpenMPI and others to enable checkpointing of parallel applications that communicate through MPI.

Researchers at North Carolina State University (NCSU) have been working on various extensions for BLCR. One of the extensions for the BLCR was incremental checkpointing. BLCR's current naive approach checkpoints the entire state of the process at every checkpoint period. In most cases, in accordance with the 90/10 law, the process might sit in a tight loop for the entire period between two checkpoints and only modify a subset of application state. In such cases, checkpointing the entire process not only wastes memory but also time. With large applications, write throughput to disk can rapidly become the bottleneck for large checkpoints. Hence, reducing write pressure on the time-critical path of execution through incremental checkpointing can become quite important.

With the incremental checkpointing approach, the key virtue is the detection of modified data. The most convenient approach would be to detect modifications at page granularity. However, there can be various methods to detect modifications on a page. The previous approach taken by researchers at NCSU was to propagate the dirty bit in the page table entry to user level by using a kernel patch [15].

Contributions:

This paper presents the design, implementation and evaluation of two different approaches to incremental checkpointing. Our contributions are as follows:

- We present an approach for the detection of modified data pages that does not require patching the kernel as in previous work and can instead be used on vanilla kernels.
- We compare and contrast the two approaches for performance and establish the pros and cons of each. This helps the users decide which approach to select based on their constraints.
- We compare the performance of the two approaches against base checkpointing to assess the benefits and limitations of each.
- We show that our lower overhead dirty-bit tracking has the potential of a significant reduction in kernel activity. When utilized for proactive fault tolerance, an immanent fault could more likely be circumvented by dirty bit-based live migration than by a write protection-based scheme due to these overhead. As a result, a process could be migrated from a node about to fail to healthy node with a higher probability under dirty-but tracking than under write protection.

2 Related Work

C/R techniques for MPI jobs frequently deployed in HPC environments can be divided into two categories: coordinated checkpointing, such as LAM/MPI+BLCR [13, 6] and CoCheck [14], and uncoordinated checkpointing, such as MPICH-V [4, 5]. Coordinated techniques commonly rely on a combination of operating system support to checkpoint a process image (e.g., via the BLCR Linux module [6]) or user-level runtime library support. Collective communication among MPI tasks is used for the coordinated checkpoint negotiation [13]. Uncoordinated C/R techniques generally rely on logging messages and their temporal ordering for asynchronous non-coordinated checkpointing, e.g., by pessimistic message logging as in MPICH-V [4, 5]. The framework of OpenMPI [3, 10] 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 cost by checkpointing only the modified pages.

Incremental Checkpointing: Recent studies focus on incremental checkpointing [7, 9]. TICK (Transparent Incremental Checkpointer at Kernel Level) [7] 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 [9] 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. Yi et al. [17] 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. [1] 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 for the write bit (WB) or a page table dirty bit (DB) approach. Different implementation variants build on these schemes. One is the bookkeeping and saving scheme that, based on the DB scheme, copies pages into a buffer. Another solution is to exploit page write protection, such as in Pickpt and checkpointing for Grids under XtreemOS [11], 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 present two different approaches to incremental checkpointing in this work. The first approach exploits the write bit (WB) to detect modifications on a page level. This approach does not require the kernel to be patched (unlike Grid checkpointing under XtreemOS, which required a patch [11]). This is different than the prior work since it uses innovative approaches to handle corner cases for detecting modifications on pages. The second approach uses the dirty bit (DB) for tracking writes on page. This approach shadows the DB from the kernel within the user level and captures the modification status of the page. Both our approaches work for entire MPI jobs.

3 Design

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.

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

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

The main criteria of the design of incremental checkpointing 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 (DB)

approach. The details of this approach are discussed below. This approach requires 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 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 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 algorithms to be implemented without major code changes in the main BLCR module. The interface object is depicted in Figure 1.

```
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 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 2.

The initialization function allows a specific incremental approach 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 that need to be registered. The register_handler function is used for registering specific hooks. This function is utilized here to register hooks for memory mapping and shared

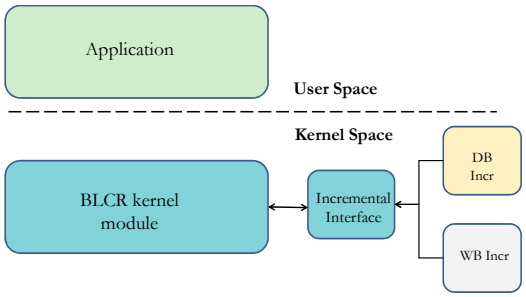

Figure 2: BLCR incremental design

writes. The mmap hooks 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 bits for the next checkpoint

3.2 Write Bit Approach

The WB approach is inspired by work by Mehnert-Spahn et al. [11] 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 modifications of kernel code were required. Therefore, in addition to the WB, additional mechanisms were utilized for incremental checkpointing.

In this approach, the WB is cleared at each checkpoint. At the next checkpoint, we check whether the WB is set or not. If the page whose WB 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 page table entry and return. The WB 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 they can be handled in the following.

3.2.1 VM Area Changes

One of the major issues with the above WB 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 be 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 in the next section. Map tracking includes:

- A page is 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 hook called 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 are 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 WB 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 WB 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 WB. Before the next incremental checkpoint, the user protects the page allowing only reads, effectively clearing the WB. 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 DB. The mprotect function, while setting permission bits, masks the DB. Hence, if the page is modified then we can detect it through the DB.

The other corner case is that of shared memory. In BLCR only one of the processes will capture the 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 even one of them is modified, then the shared page is dirty and should be checkpointed.

3.2.3 Tracking Structure

The tracking structure for incremental checkpointing is a virtual page table maintained by the BLCR module. This is done for two purposes: (1) to track VM area changes like unmapping, remapping, new mapping etc; (2) to detect writes to shared memory. Only two bits suffice to maintain the tracking state of the page. Initially, the design was to replicate a page table structure in BLCR to maintain the state of each page. Since this will have to be performed for the entire process, using a long type variable would waste a significant amount of memory. We have optimized this tracking structure to use only 4 bits per page. This results in an almost eight-fold reduction in memory usage as compared to maintaining a properly mirrored page table.

3.3 Dirty Bit Approach

The second approach taken by previous work uses the DB for detecting page modifications. It uses an existing Linux kernel patch to copy the PTE DB into user level [15]. The problem with using the DB is that the kernel uses the DB for its own purpose, which might introduce an inconsistency if BLCR and the Linux 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 WB approach but still allows dirty page tracking.

4 Framework

We conducted our performance evaluations on a local cluster. 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 a 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.

5 Experiments

We designed a set of experiments to assess the overheads and analyze the behavior 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 the performance of application benchmarks.

Various NAS Parallel Benchmarks [2] (NPB) as well as a microbenchmark have been used to evaluate the performance of above two approaches. From the NPB suite, we chose SP, CG, and LU as their runtimes are long enough for checkpoints. In addition, we devised a microbenchmark that scales from low to high memory consumption in order to evaluate the performance of the incremental approaches under varying memory utilization.

5.1 Instrumentation Techniques

For getting precise measurement of time, the method of instrumentation is quite important. The BLCR framework has been modified to record timings of two levels. Figure 3 depicts the block diagram of an application. In the context of the NPB suite, this would be an MPI program with cross-node communication via message passing [8]. We can issue an ompi-checkpoint command so that the OpenMPI framework will engage in a coordinated checkpoint [10]. To assess the performance, we could simply measure the time across the ompi-checkpointing call. However, this would required modifications to OpenMPI. It would also include the timing for the coordination of MPI processes due to an implicit barrier, which would skew our results. Instead, we modified the BLCR library. We measure the timing across the do_checkpoint call in each of the processes. The processes then output their time to a common file (see Figure 3).

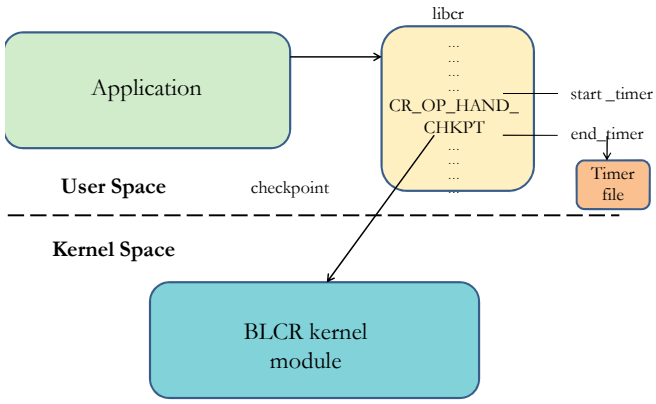

Figure 3: BLCR library timer design

There is one small caveat with the above approach. Our initial tests showed very low variations between the two incremental approaches. 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 overhead was dominating any other time like, including the time to detect page modifications. Our approach thus was aimed at excluding the write time from the total time. We wanted to only measure the time to detect page modifications. To this end, we enhanced the BLCR kernel module to measure only the modification commands. The design is as depicted in Figure 4. 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.

Automated checkpoint scripts enable regular checkpointing of various MPI and non-MPI processes.

5.2 Memory Test

We have split the test suite into two parts. The first part, the memory test, measures the difference between two checkpointing approaches on a single machine. The sec-

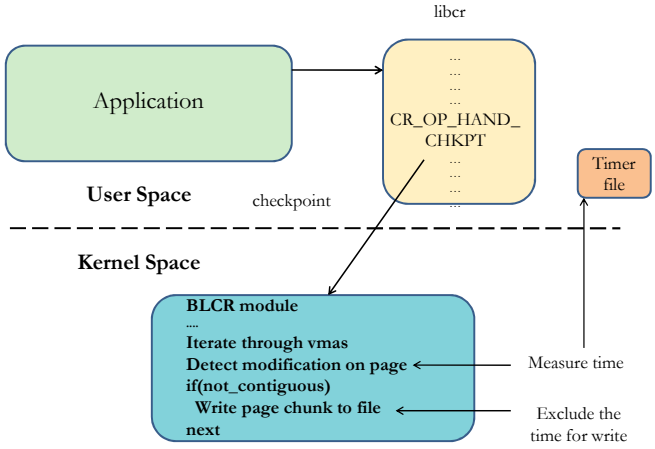

Figure 4: BLCR kernel module timer design

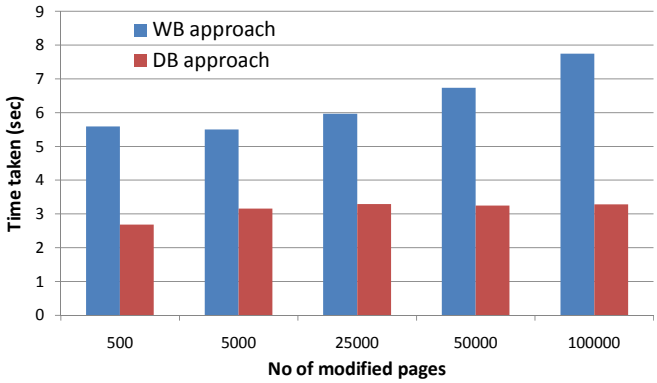

Figure 5: Micro benchmark Test

ond part of experiments measures the impact of performance on multi-node MPI benchmarks as the number of nodes and the memory consumption scales. We discuss the first set of experiments in this section. We have devised a microbenchmark for measuring the performance difference between the two approaches of WB and DB. This benchmark allocates a specified number of memory pages and, depending on the configuration specified, alters a certain number of pages per checkpoint. This allows a comparison of the performance under varying memory utilization.

The experiment is conducted on a large data set. We create a map of 200,000 memory pages within a single process. We constrain the number of checkpoints at 20 with a ratio of incremental to full checkpoints at 4:1. This means a full checkpoint is always followed by four incremental checkpoints as such a hybrid scheme was shown to be superior to only incremental checkpointing [16]. We vary the number of modified pages by large increments. The data points for this graph are at 500, 5k, 25k, 50k, and 100k modified pages. The results are depicted in Figure 5.

Figure 5 indicates that the difference between the performance of DB and WB is low when the set of modified pages is low. As the number of modified pages increases, the difference also increases. When the modified data set reaches 100,000 pages, the difference is almost twice that of WB. We can conclude from this experiment that using the DB approach has significant potential to improve performance.

To understand this result, let us explain the mechanism first. BLCR iterates through every page and checks 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 a page has been modified or not, the clear bit function clears the WB 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 sweep 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.

We devised a second case using alternate pages to provide insight into the performance difference of incremental vs. default full checkpointing. In this case, alternate pages from the process address space are modified and the performance is assessed. We provided a fixed-size data set of 100k pages here. By writing to ever other page, 50k pages will be modified between checkpoints. We observed that incremental checkpointing takes significantly longer than full default checkpointing. It seems counter intuitive that saving a smaller data set for incremental checkpointing takes more time than saving the full checkpoint data, yet the explanation to this anomaly lies in the way BLCR saves a process' memory space. BLCR iterates through each virtual memory area (VMA) structure to gather contiguous chunks of pages before committing them to stable storage. Upon a full checkpoint, the entire mapped space becomes one chunk written to disk through a single system call. When we modify alternate pages, we encounter an unmodified page after each modified page, where the former is discarded by BLCR as it is unmod-

ified. Since the chunk breaks there, BLCR will have to issue a write to disk for each single modified page. Therefore, we issue significantly more write calls in incremental checkpointing than in full checkpointing. Notice that this deficiency is being addressed by aggregating non-contiguous chunks before committing them to stable storage, but such an approach comes at the cost of additional meta-data to describe the internal check structure.

5.3 NAS Benchmarks

In this section, we analyze the performance of multi-node MPI benchmarks for the WB and DB approaches. We selected the MPI version of the NPB benchmark site [2]. We constrained the selection of benchmarks from the NPB set to those with sufficiently long runtimes to take a suitable number of checkpoints.

We devised a set of experiment using strong scaling by increasing the number of processors. With such increase in computing resources, we decrease the per-node and overall runtime. However, this renders some benchmarks unusable for our experiments as the runtime was not sufficient to issue a checkpoint, particularly for smaller input sizes (classes A and B) of the NPB suite. Yet, using large input sizes (class D) under NPB on fewer processors (1 to 4) is not practical either due to excessively long checkpointing times. Hence, we settled for input sizes of class C for the experiments.

Considering all benchmarks and assessing their runtimes, we selected three suitable benchmarks for our tests: SP, CG and LU. We present the following experiments and results for the same.

We assessed the performance for the SP benchmark on 4, 9, 16 and 36 processor. Notice that SP requires the number of processors to be a perfect square. The experiments were performed on class C inputs with a checkpoint interval of 60 seconds over a varying number of nodes. Figure 6 depicts the runtime spent inside the Linux kernel portion of BLCR. 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 implies less data per node to checkpoint and, hence, less time spent on checkpointing in the kernel.

In the case of 36 processors, we see a sudden spike in kernel runtime. This anomaly is attributed to the fact we only have 16 physical nodes but deploy the application across 36 processes. Thus, multiple processes are vying for resources on the some nodes. The processes are contending for cache, networking and disk. Hence, the 36-processor case (and any oversubscription case studied in the following) should only be considered by itself and not in comparison to lower number of processes.

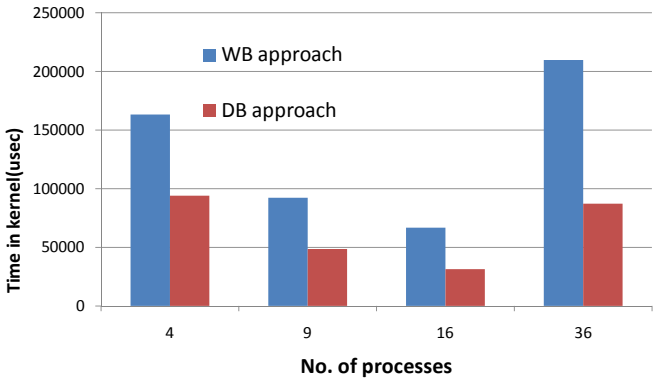

Figure 6: SP benchmark

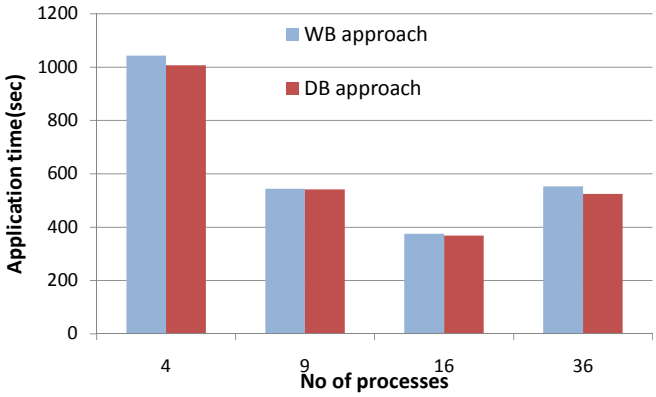

Figure 7: SP benchmark (Application time)

Figure 7 depicts the overall application time for the SP benchmark for different numbers of nodes. We see that the DB approach slightly outperforms the WB approach in all cases. As the number of processes (and processors) increases from 4 over 9 to 16, we see a decrease in total application time. Recall that the work gets distributed between various nodes as the number of processors increases while the application time decreases. This happens under both the configurations, WB and DB. For 36 processes, the application time goes. Again, we are oversubscribing with 36 processes on 16 physi-

cal nodes. This causes contention for memory, networking and disk. The DB approach shows slightly higher performance gains for this oversubscription case likely due to mutual exclusion inside the kernel (kernel locks), which impacts WB more due to more time spent in the kernel.

The next NPB program tested was the CG benchmark for class C inputs. We varied the number of nodes from 4 over 8 and 16 to 32 processors. The incremental to full checkpoint ratio is kept at 4:1. Checkpoints are taken every 10 seconds.

Figure 8 depicts the kernel runtime for CG. These results indicate considerable savings for DB over WB for 4 processors and smaller savings for 8 processors. At 16 processors, more savings materialize for DB. In contrast, the overhead of WB increases drastically. This anomaly can be explained as follows. The total running time of CG is low. Checkpoints were taken at an interval of 10 seconds. Since the savings due to the DB approach exceed 10 seconds, the benchmark run under DB resulted in fewer checkpoints, which further decreased the application runtime. In contrast, WB ran past the next checkpoint interval, which incurred one additional checkpoint. Thus, we may gain by issuing fewer checkpoints under DB due to the lower kernel overhead of the latter.

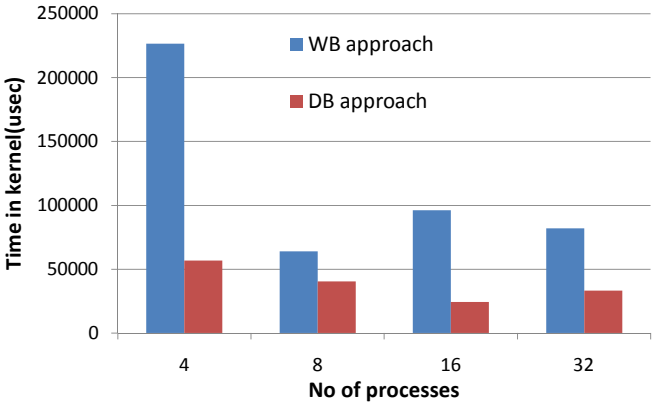

Figure 8: CG benchmark

Figure 9 depicts the total application time for CG. We see considerably more savings of DB over WB in the case of 16 nodes than for 4 or 8 nodes due to the lower number of checkpoints for DB. In all other cases, DB slightly outperforms WB. The higher overall runtime for 32 processes is caused by node oversubscription again.

Next, we assessed the performance under the LU benchmark for 4,8,16 and 32 processors under class C inputs.

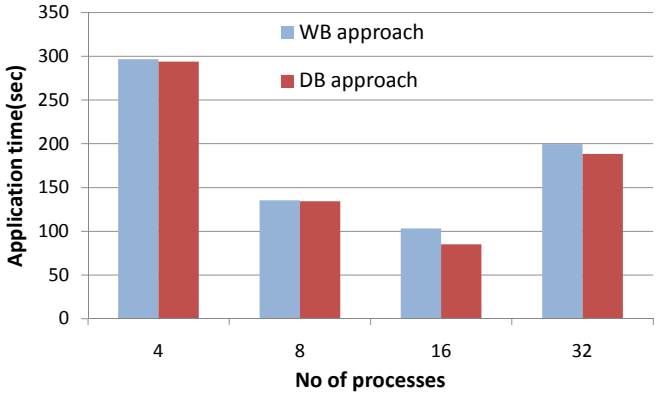

Figure 9: CG benchmark (Application time)

Checkpoints were taken every 45 seconds, and the incremental to full checkpoint ratio was 4:1.

Figure 10 depicts the kernel time. As in the previous experiment, there are significant savings in time spent in the kernel. The total time decreases as the number of nodes increases from 4 over 8 to 16. Under 32 processes, we see an increase of total time relative to the prior process counts due to node oversubscription. The savings of the DB compared to WB are significant in all cases. As in the previous savings, these savings in the order of microseconds only materialize in minor overall application runtime reductions as application runtime is in the order of seconds. The results for total application time for LU are depicted in Figure 11.

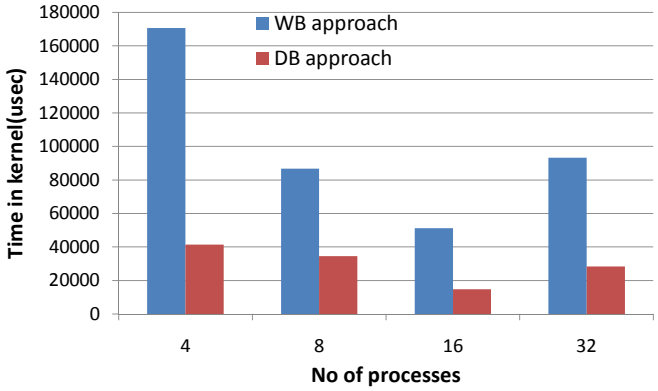

Figure 10: LU benchmark

In terms of overall runtime for LU, we see that DB is at par or slightly outperforms WB. We also see that the percent are quite low compared to the percent savings for kernel runtime in the previous graph. As explained before, this is due to the fact that the time spent in kernel is measured in microseconds while application time is measured in seconds.

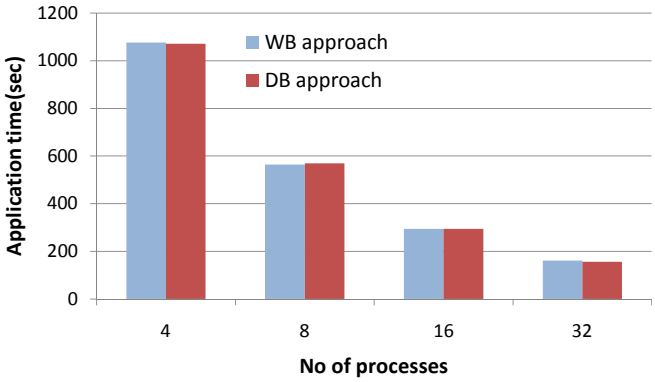

Figure 11: LU benchmark (Application time)

In summary, we observe that the DB approach incurs significantly less kernel overhead but only slightly less overhead than WB approach for most test cases. Larger savings are observed for longer-running applications as a slight reduction in DB overhead may aggregate so that fewer overall checkpoints are taken at the same checkpoint interval.

6 Future Work

We have developed and evaluated two different approaches to incremental checkpointing. One (WB) required patching the Linux kernel to detect modifications at page level while the other (DP) did not require any patches. We have further quantitatively compared the two approaches. We are currently investigating if patching of the kernel for DB can be omitted when swap is turned off. This would alleviate the user from the tedious kernel patching and recompilation of the kernel. This approach is particularly promising for high-performance computing under MPI as swap tends to be disabled. We are currently DB usage within the kernel beyond swap functionality to determine if utilization of DB by the BLCR would create any side effects for the kernel. We are also considering dynamic activation and deactivation of swap while a process is running. In that case, the DB functionality bit should be gracefully handed over to the kernel without affecting ongoing checkpoints. These issues are currently being investigated and we aim to implement them in the future. Furthermore, we are considering to integrate both incremental checkpointing mechanisms, DB and WB, with the latest BLCR release. The mechanism are already in the BLCR repository and the integration work is under way.

7 Conclusion

In this paper, we outlined two different approaches to incremental checkpointing and quantitatively compared them. We conducted several experiments with the NPB suite to determine the performance of the DB and WB approaches in head-to-head comparison them. We make the following observations from the experimental results. (i) The DB approach is faster significantly than the WB approach than DB with respect to kernel activity. (ii) DB also slightly outperforms WB for overall application in nearly all cases, and particularly for long-running application where DB may result in fewer checkpoints than WB. (iii) The WB approach does not required kernel patching or kernel recompilation. (iv) The difference in performance between the WB and the DB approach increases with the amount of memory utilization within a process. (v) The advantage of DB for kernel activity could be significant for proactive fault tolerance where an immanent fault can be circumvented if DB-based live migrations moves a process away from hardware about to fail.

8 Acknowledgement

This work was supported in part by NSF grants 0937908 and 0958311 as well as by subcontract LBL-6871849 from Lawrence Berkeley National Laboratory.

References

[1] 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.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[3] 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.

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

[5] 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.

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

[7] 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.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[9] 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.

[10] 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.

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

[12] Ian Philp. Software failures and the road to a petaflop machine. In *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*. IEEE Computer Society, 2005.

[13] 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.

[14] 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.

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

[16] C. Wang, F. Mueller, C. Engelmann, and S. Scott. Hybrid full/incremental checkpoint/restart for mpi jobs in hpc environments. In *International Conference on Parallel and Distributed Systems*, December 2011.

[17] 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.