Incremental Checkpointing in BLCR

Name: Manav Vasavada
Unity Id: mmvasava

Abstract

This project is funded by Lawrence Berkeley National Labs(LBNL) and aims to implement and analyze different approaches of incremental checkpointing in Berkeley Labs Checkpoint Restart (BLCR) framework. Incremental checkpointing enables BLCR to checkpoint to save only modified process data and thus saving disk space and optimizing I/O bandwidth. Two approaches are implemented based on the method of page modification detection. One approach implements the incremental approach without any modifications to kernel while the other approach patches the kernel to detect page modifications. The implementation details for both are described in later sections. We also discuss various experiments to compare the performance of both approaches and their setup.

Introduction

With the development in the supercomputer field, the number of core in high performance computing has scaled upto thousands of processor cores. Huge scientific applications with highly parallel execution patterns are exploited to be able to be used on these machines. Even with the amount of processor power available, some applications still can take upto days to execute on such high end machines. Such applications include climate modeling, protein folding algorithm, 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 chances 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 doing 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. The solution for this is checkpoint restart. 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 naïve 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[Citation needed]. Incremental checkpointing involves checkpointing only the data of a process which has been modified and ignoring the rest. This helps save disk space and optimizes I/O bandwidth. The modification detection in this mechanism is currently done 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. The next section gives a brief introduction of BLCR which is followed by description of two different approaches considered for the implementation. Section 4 gives the technical details of both the

implementations and their integration into BLCR. Section 5 describes the experimental setup and results of the experiments. We finally conclude with future work and conclusion of analysis of both the approaches.

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 be made to application code. This work focuses on checkpointing parallel applications that communicate through MPI. BLCR support has been integrated on various MPI implementations like LAM/MPI, MVAPICH, OpenMPI etc. Researchers at North Carolina State University have been working on various extensions for BLCR[Citations of various papers]. The incremental checkpointing is one of the extensions to BLCR by this research group.

Approach

I) Write bit approach [Needs to be renamed]

The first approach was considered taking the fact that no kernel patch should be applied. For this approach a previous work on incremental checkpointing on XtreemOs was considered[Citation needed]. This approach track the page modification based on the write bit protection. The mechanism works as follows. In the beginning all the pages of a process address space are write protected with the write bit of each pte cleared. During the course of execution, when a certain page is written to, it generates a write fault hence alerting the checkpoint mechanism to the fact that the page was modified. This specific approach, however, had several shortcomings. This approach did not support shared memory modifications. Also XtreemOS uses a modified kernel which is not feasible for a generic approach. Hence most of the parts of this approach had to be re-thought for a generic linux kernel.

The final approach used the write bit to detect modification on private anonymous pages. Other corner cases which were to be handled are as follows:

VM area changes

- Case I - Unmap

If the pages are unmapped between checkpoints, the corresponding tracking structure for that page should also be invalidated or removed. To do this we need to have a hook in the when munmap happens. Also during restart, care should be taken not to restore those areas from the full checkpoint file which have been unmapped in the subsequent incremental checkpoints.

Solution: There is already a unmap hook in the mm_struct data structure which is called whenever any region gets unmapped. We can use that to invalidate a page when it gets unmapped.

- Case II - New regions

For new regions which are mapped between checkpoints, we need to checkpoint them at the next checkpoint regardless of whether they are modified or not.

Solution: For such checkpoints, we will not allocate a tracking structure. During checkpoint, if we find a page without a tracking structure we assume it is newly mapped and checkpoint it completely.

- Case IV - Change in size

For a vm area like stack, if it grows we have to track change in the vm area.For example a full checkpoint checkpoints 4 pages of a stack. At the next pages, the 4 pages remain unchanged but three other pages are added. So the three pages not only should be checkpointed as dirty, but there should be something to indicate the change in vma bounds of the stack otherwise stack pointer will get sigsegv on restart.

Solution: The context file format of incremental checkpoint will mostly take care of this.

- Copy-on-write

One of the main concern was that by making the page write protected, they will be copied when written to every time. For large pages writing them at every fault thrown due to the write protection can be very expensive.

Result: When the pages are mapped as MAP_PRIVATE, it does not have its own copy(In case its file backed it is still from the file and in case of Anonymous mapping it is the zero page). The first time the page is written to, a fault will be thrown. Before this, the pte for that page will not have been initialized so __do_fault() will be called (For code see here) which copies the page. After this, the anonymous bit in mapping field of page descriptor will be set and pte will be initialized. If we write protect it again and then write to it, the fault will not go to __do_fault but into do_wp_page which will check to see if the page is anonymous and it will not copy it again. This works for our incremental checkpointing approach. For shared maps no cow is necessary.

- mprotect

Another more serious problem is that of mprotect. When a vm area is mprotected against writes, it relies on the write bit of each page to throw faults which then check VM permissions. Setting the write protect bit after mprotect can circumvent protection. Moreover if someone writes to a page, which sets the write bit and then mprotects it thus clearing the bit, the next checkpoint will think the page is unmodified and skip it which is wrong. To state this more clearly, see the following example:

1) we take a checkpoint and clear the WRITE bit
2) user writes the page and takes a fault, which sets the WRITE bit

3) user calls mprotect() w/ PROT_READ|PROT_EXEC, which clears the WRITE bit
4) we take another checkpoint and see the WRITE bit clear and thus miss the fact that the page has been written

Result: Since we always disable writes on a page we should not be worried about circumventing mprotect. However the next case fails our tracking of the write bit. This can be avoided by maintaining a VM write bit in our own data structure. This way if the VM permission has changed through mprotect we will know it.

II) Dirty bit approach

This approach is based on previous work by HP on IA64 architecture[Citation needed]. Instead of relying on BLCR tracking system, this approach proposes to rely directly on the kernel for page modification detection. The kernel uses dirty bit to track changes to pages internally and maintain consistency of memory. However, kernel modifies the dirty bit during its processing and hence it cannot be used directly. This approach proposes to modify kernel and duplicate the dirty bit. This shadowed dirty bit is not modified everytime kernel modifies the dirty bit but maintains the tracking of the page which can be requested by the application.

Modifications to the kernel

This approach, instead of using any shadow bits in BLCR, uses unused pte bits to store the shadow dirty bit. For each architecture there are a few bits in the pte which are as yet unused. So this patch duplicates the dirty bit from the pte modification macros in the page table header files. On request by the application, it will return the status of the dirty bit relative to the last call which is what the incremental approach needs. This approach prevents the page faults at every write unlike the previous approach.

Design

The design of the incremental checkpointing is designed to be modular. The entire process is controlled by an incremental checkpointer object which implements certain methods. These methods can be replaced based on the approach that is required to be used. Such modular approach can enable users to switch between the two approaches on the fly and analyze performance of both.

Memory Optimization of BLCR page tracking structure

The BLCR tracking structure for pages for the write bit approach was earlier supposed to identically have a page table structure. This meant allocating a page table like structure in BLCR memory to keep track of a shared writes and unmapping of memory regions. For maintaining this information only two bits are required. Initially, the data structure was using unsigned long (8bytes) for storing this information. This can create a lot of space wastage with large enough address spaces. To avoid this, the memory regions were optimized to use only a 4 bits per page

rather than 8 bytes. This compresses the information storage and saves space for BLCR page table allocation.

Interface flags for BLCR

The interface flags for blcr has been implemented to include incremental checkpointing and bypass the entire incremental code without any considerable penalty when not being used. For the entire interface implementation discussion see:

http://www4.ncsu.edu/~mmvasava/doc/interface.htm

Experimentation

The experiments include a 16 node cluster with 4 GB of ram on each. NAS parallel benchmarks were run to compare performances of both the approaches. LAM/MPI is used as the MPI implementation. Modified kernel is installed for the dirty bit approach.

Experiment Results:

The experiments are still being conducted and result will be available shortly.

For the project code and test cases, see the project website.