ABSTRACT

MISHRA, SHOBHIT. Design and Implementation of Process Migration and Cloning in BLCR. (Under the direction of Frank Mueller.)

The reliability of a high performance computing (HPC) environment deteriorates with an increase in the number of nodes. For large applications, a failure results in the loss of several hours of execution time. Today, we use application checkpointing to deal with intermittent failures. Applications are restarted from the last checkpoint if a failure occurs. However, checkpoint/restart (C/R) is a reactive approach of fault tolerance and results in high overheads for a large application.

We suggest a proactive approach of fault tolerance by migrating a process from a failing node to a healthy node. We describe two different mechanisms to move a process from one physical node to another. In first approach, we halt the process at the failing node and transfer it to the destination node via sockets. This approach is known as frozen migration. The second approach, known as live migration, allows the process to run while the migration is being carried out. We can allow the process to run at the source node as well as the destination node thus creating a process clone. We rely on an external agent to detect any imminent failure and trigger the migration process. We use a small kernel patch to keep track of modified pages. The work is carried out within the latest release of the Berkley Labs Checkpoint Restart (BLCR) code. The migration utilities will be integrated into BLCR in future releases. Experimental results demonstrate the significance of these proactive fault tolerance techniques to improve the reliability in HPC environments.

Design and Implementation of Process Migration and Cloning in BLCR

by
Shobhit Mishra

A thesis 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

2011

APPROVED BY:

Xiaohui (Helen) Gu

Steffen Heber

Frank Mueller
Chair of Advisory Committee

DEDICATION

To my parents.

BIOGRAPHY

Shobhit Mishra was born in Ghazipur, a small town in the state of Uttar Pradesh in India. He did his schooling in Ghazipur and went to Indian Institute of Information Technlogy Allahabad for his B.Tech in Information Technology. He joined CSC India Pvt. Ltd. as a System Administrator and worked there for three years. He came to NC State in Fall 2009 as a Master's student in the department of Computer Science. He has been working under Dr. Frank Mueller as a Research Assistant since August 2010.

ACKNOWLEDGEMENTS

This work would not have been possible without the collective effort of a lot of people. First and foremost, I would like to thank my advisor Dr. Frank Mueller for showing confidence in me and giving me the opportunity to work on this project. His guidance and feedback put me on the right track from time to time. I would like to thank Paul H. Hargrove at Lawrence Berkley National Laboratory for acting as my mentor. He was always there to help me when I hit an impasse. I am thankful to Dr. Helen Gu and Dr. Steffen Heber to serve on my committee. I would also like to thank Chao Wang and Manav Vasavada for their foundation work on this project. Lastly, I would like to thank my labmates in the System Research Lab and my family for their support.

TABLE OF CONTENTS

List of Figures . . . . **vii**

Chapter 1 Introduction **1**
1.1 High Performance Computing (HPC) 1
1.2 Fault Tolerance 2
1.3 Berkley Labs Checkpoint Restart (BLCR) 2
1.4 MPI 4
1.5 Extensions to BLCR 4

Chapter 2 Motivation, Hypothesis and Contribution **6**
2.1 Motivation 6
2.2 Hypothesis 7
2.3 Contribution 7

Chapter 3 Design **8**
3.1 Added Utilities 8
3.2 Steps of Migration 9

Chapter 4 Implementation **12**
4.1 Initialization and Restoration: 12
4.2 Memory Precopy: 16
4.3 Stop-and-Copy: 18

Chapter 5 Experiments **23**
5.1 Framework 23
5.2 Experiments 23
5.2.1 Instrumentation Techniques 23
5.2.2 Memory Tests 24
5.2.3 Other Tests 26

Chapter 6 Related Work **28**

Chapter 7 Conclusion and Future Work **30**
7.1 Conclusion 30
7.2 Future Work 30
7.2.1 Integration with OpenMPI 31
7.2.2 Cloning 31
7.2.3 Using WB Approach 31
7.2.4 Using File Descriptors 32

References **33**

LIST OF FIGURES

Figure 1.1 Application running with and without checkpointing 3

Figure 3.1 Overall flow of the process migration 11

Figure 4.1 Overall flow of the cr_restore utility 13
Figure 4.2 Dirty bit tracking scheme 17
Figure 4.3 Migration with precopy 19
Figure 4.4 Migration without precopy 20
Figure 4.5 Sequence of data sent during precopy 22
Figure 4.6 Sequence of data sent during stop-and-copy 22

Figure 5.1 Stop-and-Copy Timings with a Stride of One 25
Figure 5.2 Stop-and-Copy Timings with a Stride of Ten 25
Figure 5.3 Timings for Live and Frozen Migration 26

Chapter 1

Introduction

1.1 High Performance Computing (HPC)

HPC uses several parallel processing techniques to solve advanced computational problems quickly and reliably. HPC is widely used in scientific computing applications like weather forecasting, molecular modeling, complex system simulations, etc. Traditional supercomputers are custom made and very expensive. A cluster, on the other hand, consists of loosely coupled off-the-shelf components.

Special programming techniques are required to exploit HPC capabilities. The most common programming paradigm in such machines is message passing. Each node is allocated a small part of the overall problem and they communicate through coordinated message passing. Message Passing Interface (MPI) implementations provide scalability and portability without compromising performance and is the de-facto standard for HPC applications. There are several MPI implementations available today. OpenMPI [13], MPICH [3], and WINMPICH [19] are some of the most common implementations. Since the frequency scaling in a uniprocessor system has hit a wall, more and more researchers are turning to HPC for scientific computations. As of June 2011, ten supercomputers have crossed the Petaflop barrier in max performance and thirteen have crossed in peak performance [1]. Development of parallel applications, however, is significantly different from sequential programming. More and more applications are being rewritten to efficiently run on HPC machines.

1.2 Fault Tolerance

A fault may be defined as a deviation from normal or expected behavior such that it causes an application failure. A fault marks an underlying inconsistency in the hardware or software. Software faults are caused by application or operating system bugs. Hardware faults may occur because of the hardware component failure or due to hardware limitations. Such faults include but not limited to I/O error, faulty RAM, network failure, disk corruption, and machine overheating. In our work, we mainly focus on fail-stop hardware failures that can be detected by an external monitoring agent.
Since the advent of Supercomputers, the number of cores in HPC machines has burgeoned into hundreds of thousands of cores. Huge scientific applications like climate modeling and molecular modeling with highly parallelized algorithms can take several hours to complete. A number of HPC clusters use off-the-shelf components to achieve this massive computational power. Reliability data of contemporary machines indicates the frequency of failure in large installations. The mean time between failure (MTBF) is in the range of 6.5 to 40 hours [16]. This points to the high probability of a job failure due to hardware/software failures during execution. For large processes, a job failure may lead to restarting the process from the beginning and thus wasting a significant amount of processing time. In addition to processing delays, this would also result in excessive use of power for duplicate computation.
Fault tolerance may be defined as the ability of a system to complete the designated task in the presence of hardware and software failures. It also includes graceful degradation and sensible program termination. It involves detecting the error and taking appropriate measures to handle it. There may be several fault tolerance actions like exiting the program and error reporting or continuing the execution on a different node. An efficient fault tolerance policy can save significant execution time and drastically reduce the power consumption.

1.3 Berkley Labs Checkpoint Restart (BLCR)

There are many approaches to enable fault tolerance in an HPC environment. One of the widely used methods is Checkpoint/Restart (C/R). C/R involves saving the context of a job/application at regular intervals and restarting the application from a saved context if a failure occurs. This approach saves significant time because we do not have to start the job from scratch.

For example, consider an application with 10 nodes and 15 hours of execution time. Further assume a checkpoint scheme to save the application context every hour. If the application fails after 10 hours then we restart the application from the last checkpoint instead of running it again from the beginning. See figure 1.1.

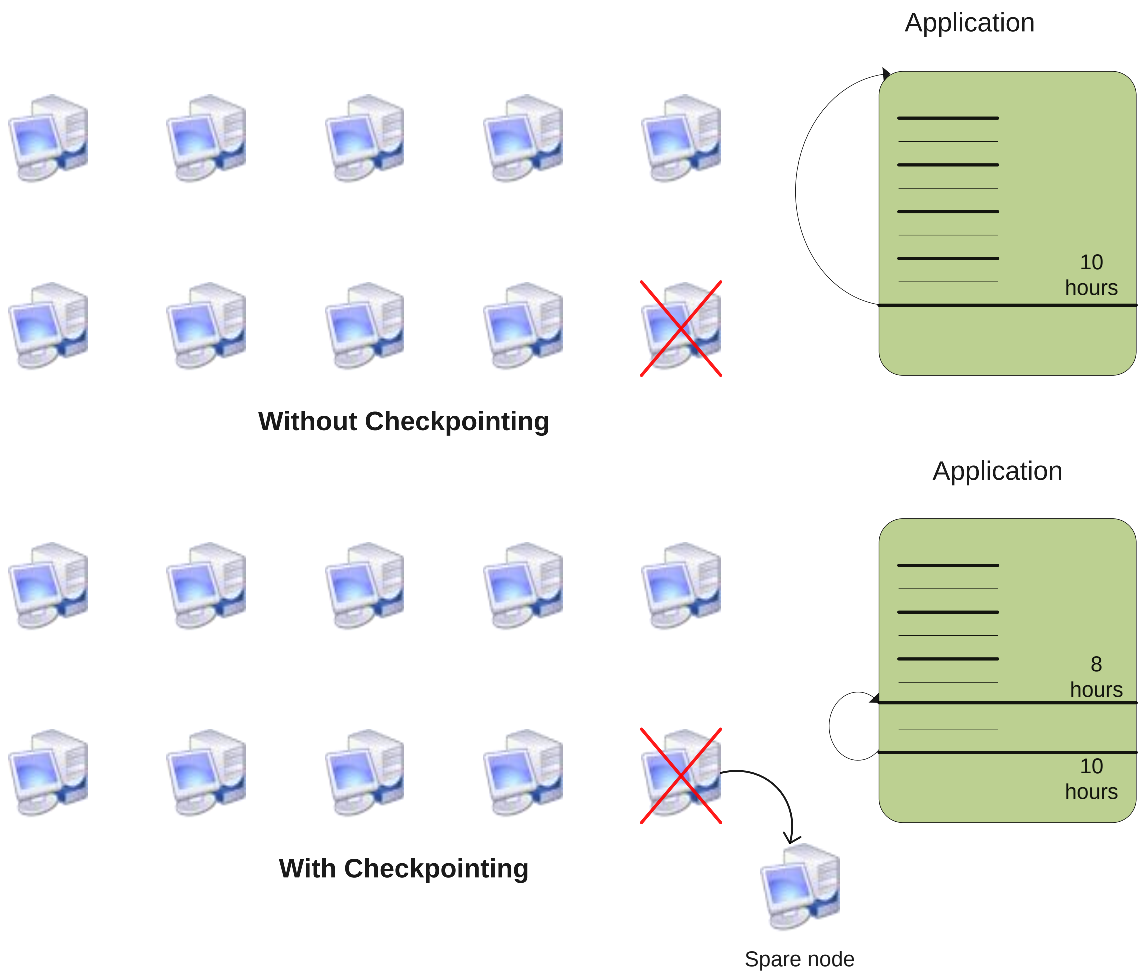


Figure 1.1: Application running with and without checkpointing

There have been several frameworks for the application-level and system-level C/R. BLCR is a hybrid kernel/user implementation for C/R. It is developed and maintained by the Future Technologies Group at Lawrence Berkley National Laboratory (LBNL). It is a robust, production-quality implementation that checkpoints a wide range of applications. It does not require any change in the application source code and works for several architectures and Linux distributions. BLCR has been integrated with a number of MPI implantations namely LAM/MPI, OpenMPI and MPICH to checkpoint and restart parallel applications running on multiple nodes. Researchers at North Carolina State University have been working on various extensions of BLCR [33] [34] [35].

1.4 MPI

The Message Passing Interface (MPI) is an application programming interface (API) that allows processes to communicate with each other by passing messages. It is widely used in the HPC environment and is a de-facto standard for parallel programming in computer clusters and supercomputers. In the MPI programming model, a computation comprises one or more processes that communicate by sending and receiving messages. Processes can use point-to-point or collective communication operations to send a message. MPI also has message probe routines to support asynchronous communication. In most MPI implementations, a fixed number of processes are created at program initialization. All these programs together are referred as a job in a cluster. MPI uses language independent specifications for calls and language bindings.

1.5 Extensions to BLCR

Checkpoint and restart of a parallel application has its own overheads. These overheads are directly proportional to the number of nodes involved. They also depend on disk I/O speed, network bandwidth and the size of the application. The size of the checkpoint file depends on the size of the application and the frequency of checkpoints. Large applications tend to have a higher checkpoint frequency that results in considerable disk and bandwidth overhead.

To reduce the checkpoint/restart overhead, several extensions are designed for BLCR at the NCSU systems research group. The first extension is an in-place rollback feature in BLCR to reduce the overhead by reusing existing resources at the time of restart. The second extension is the incremental checkpointing. It aims to minimize the disk usage by saving only the modified pages. The third extension is live migration of a process from one node to another using network sockets. This extension works via node-to-node transfer i.e. there is no need to save the context file on a disk. Instead of checkpointing the complete job, it moves a single process from a node to another node thus saving the disk usage and bandwidth. We shall look at this extension in later sections.

In this thesis, we propose two different methods of moving a process from one physical node to another node. We propose an approach to stop a process at one node and migrate it to the other node. We use several BLCR library functions and techniques to do so. We also propose an alternative approach to allow the process to run at the source node while it is being migrated to the destination node. We can choose to kill the process at the source node once migration is complete or to let it run on the source as well as the destination node. The second method is also known as cloning. We also propose a modular approach for the implementation of the two approaches that would allow the user to choose between live and frozen migration/cloning.

Chapter 2

Motivation, Hypothesis and Contribution

2.1 Motivation

We have worked on several extensions of BLCR. The first extension was an in-place process rollback. On any failure, BLCR kills all the processes and moves the checkpoint image of the failing process to a spare node and restarts the application from the previous checkpoint. Since we kill even the healthy processes and resubmit the entire MPI application again, this method has a significant overhead in terms of time because other jobs in the queue will be executed before this resubmitted job gets a time slice. The second extension was incremental checkpointing. Incremental checkpointing checkpoints only the modified pages since the last checkpoint. It only stores the difference between two checkpoints instead of two full fledged checkpoint files. Incremental checkpointing helps in saving disk space and optimizes I/O bandwidth. There are two methods to identify the modified pages name by the dirty bit approach and the write bit approach. The dirty bit approach requires a small kernel patch to access the dirty bit information maintained by kernel. The write bit approach uses page protection mechanisms to trigger a signal when a page is modified.
Although these two approaches have several advantages compared to a naïve approach, they still are a reactive approach of fault tolerance. They help in application recovery once the fault has already occurred. The third extension to BLCR is frozen/live migration, which is a proactive approach of fault tolerance. We rely on a health monitoring

fault prediction agent to detect deteriorating health of a node and to then trigger the migration process. We assume that the agent is able to predict the failure of the node and to subsequently take proactive measures such as process migration. The idea is to move the process from the bad node to a healthy node via network sockets without creating any checkpoint file. This significantly reduces disk space requirement and avoids slow disk access for large network files. Instead of saving the process data on a disk and restoring it, we do the restoration on the fly. In fact, the application stops just for a fraction of a second before resuming execution instead of coming to a complete halt before resuming the execution from a checkpoint file. We have implemented two different flavors of migration, namely frozen and live migration. Frozen migration stops the process at the source node and transfers all the process-related information in one pass. Live migration, on the other hand, transfers the memory pages to the destination node before stopping the process and saving the rest of the process control block. In live migration, memory pages are transferred iteratively. In each iteration, only modified pages since the last iteration are transferred to the destination node. We identify dirty pages using the dirty bit approach as mentioned above. We will discuss the design and implementation details in the following sections.

2.2 Hypothesis

We hypothesize that we can migrate a process from one physical node to another node via sockets while the application is running, which has the potential to result in lower overall runtime than stopping an application before migrating it. We can also create a copy of the process on two nodes thus creating a process clone. These mechanisms, once integrated with MPI run time, can aid the traditional C/R approach to provide proactive fault tolerance mechanisms.

2.3 Contribution

We present the design and implementation of a process migration scheme at the kernel level which does not require any application-level modifications. We use the dirty bit scheme to track modified pages. It requires a small kernel patch to access the dirty bit information of a page. The work is carried out within the latest version of the BLCR.

Chapter 3

Design

This section describes the design details of migration in BLCR. One of the basic design principles is to integrate the migration utility with BLCR with minimal changes in the original source code. We have reused a significant amount of BLCR functionality and modified it for better suitability on the migration utility. The BLCR C/R utilities are completely independent of the migration utility. They should work seamlessly when integrated with live/frozen migration in the upcoming release.

3.1 Added Utilities

We have added the following user- level utilities in BLCR:

- **cr_migrate:** This utility is invoked at the source node to start the live migration of a process. It takes the PID of the process and the destination node as input. It is invoked only if we want to trigger a live migration of the process.
- **cr_restore:** This utility is invoked at the destination node. It has to be running on the destination node before any migration utility is invoked at the source node. A switch decides if the restore corresponds to live migration or frozen migration.
- **cr_stop:** This utility is invoked at the source node in two cases. It is invoked when the cr_migrate utility returns or when we choose frozen migration instead of live migration. It takes the pid, a destination node and a live switch as arguments.
- **cr_clone:** This utility works similar to the cr_stop utility but instead of killing the process at the source node, it allows the process to continue thus creating a clone

of the process at the destination node. This utility also takes the pid, a destination node and a live switch as arguments.

These utilities are developed independently of the cr_checkpoint, cr_restart and cr_run utilities of BLCR. The cr_migrate and cr_stop utilities are very similar to cr_checkpoint. Yet, instead of writing to the checkpoint file, they write to a socket connected to the destination node. One of the main design aspects is to keep the code modular and provide sufficient entry points to call MPI-related functions in the overall execution. We have to provide synchronization in both the kernel module and at the MPI level.
Once we determine that a node is failing, we need to find a spare node to which the process can migrate. If there is not a spare node then we can overload the least overloaded machine/node by migrating the process to it.

3.2 Steps of Migration

Initialization at the Destination Node: Once a destination node is identified, an external scheduler invokes the restoration utility. We assume that we have enough spare nodes in the cluster that can be used as a default destination during process migration. This can be ensured by over provisioning of nodes at the time of job submission. If no spare node is available then the least loaded node can be chosen as the destination node. The restoration utility allocates the data structure for process restoration and establishes a socket communication with the source node. This utility should be invoked at the destination node before invoking any migration command at the source node.

Memory Precopy: Once a destination node is identified and the socket communication is established, we can start the transfer of pages to the destination node. The objective of the memory precopy is to transfer a memory snapshot of the process to the spare node while allowing the application to run. The memory transfer occurs at page granularity. Memory pages are sent to the destination node iteratively until we reach a steady state. In the first iteration, all non-zero pages are sent to the destination node. In every subsequent iteration, only the modified pages since the last iteration are sent to the destination node. The loop terminates when the change in the number of modified pages between consecutive iterations drops below a threshold level or there are no more modified pages. The mechanism of dirty page tracking is discussed in the implementation section of memory precopy. This step is carried out by the cr_migrate utility.

Message Draining: Once the precopy step is complete, we have to stop the process at the source node and transfer the last pages and other process-related information to the destination node. But before doing that, all MPI tasks need to reach a consistent global state. We need to drain all in-flight messages, and all MPI tasks should stop at a barrier. Message draining and other MPI-level synchronization are independent of the BLCR and implemented by the MPI library. A corresponding handler is invoked from all nodes once cr_migrate returns. This capability is yet to be integrated.

Stop-and-Copy: Once all MPI tasks hit the barrier, the process on the source node suspends the application execution and copies the remaining pages, the process state, descriptors of open files, linkage information, pipes and other process-related information to the destination node. All MPI tasks are suspended until stop-and-copy is finished. The process keeps running at the source node if cloning is enabled or exits if cloning is disabled. This task is accomplished by the cr_stop utility at the source node. If the live migration switch is on then cr_stop is preceded by a cr_migrate invocation otherwise, it runs alone after the message draining stage has completed.

Process Restart and Job Continuation: The process is reconstructed at the destination node. Linkage structure, open files and pipes are restored. This is the last step of process migration followed by the restoration of communication channels and drained messages. This task is accomplished by the cr_restore utility at the destination node. This utility works similarly to the cr_restart with the exception of the use of sockets instead of a file descriptor to read/receive a process image. There is no deliberate synchronization between cr_migrate, cr_stop and cr_restore. The cr_restore command recreates the process and returns. Restored process execution is separate from the cr_-restore context. When the process is ready on the destination node, it reestablishes the communication channel with all other MPI tasks. Subsequently, the drained in-flight messages are restored and the job continues. We shall have two copies of the process if clowning is enabled. In that case, we need to take care of the redundancy as MPI cannot accommodate two processes with the same signature.
The overall flow and timeline is shown in the Figure 3.1.

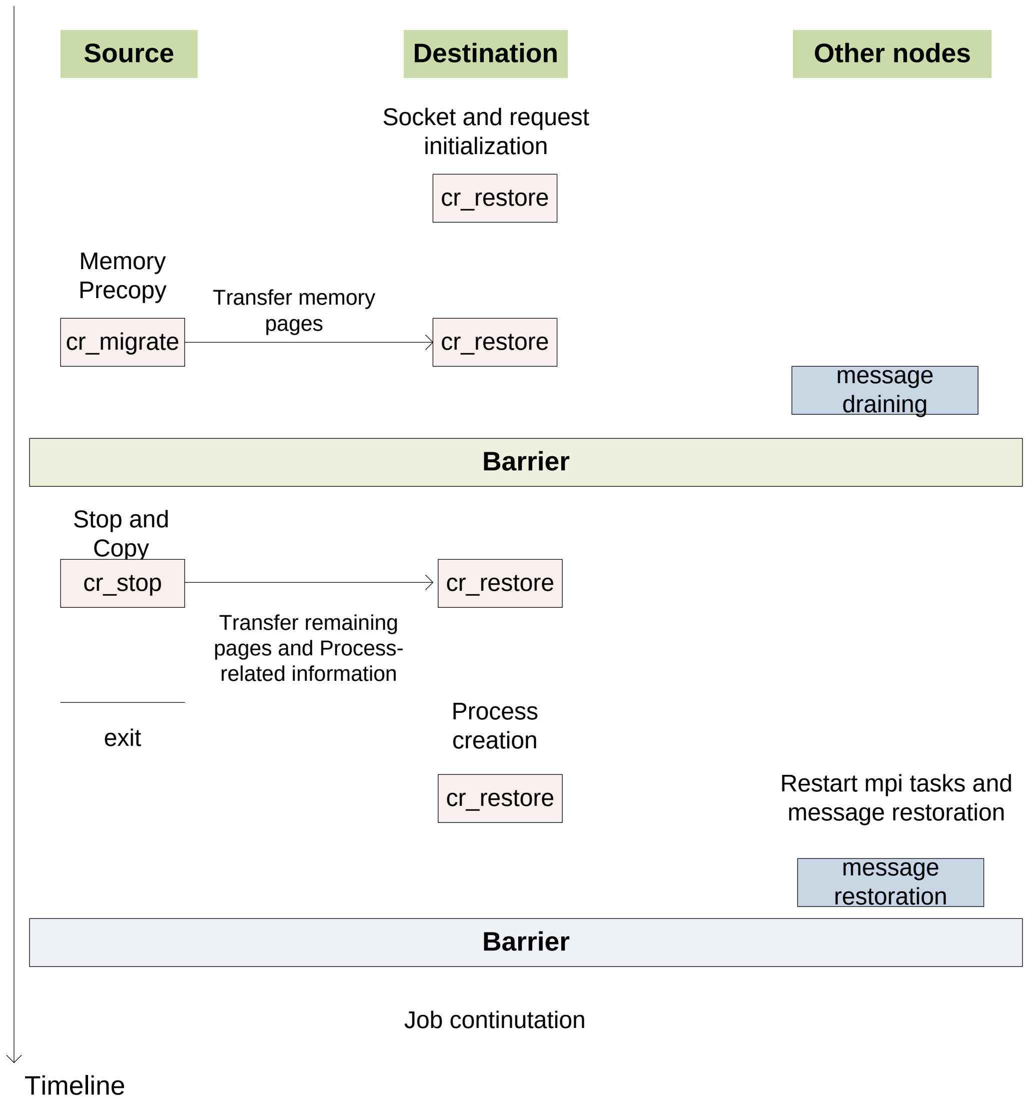


Figure 3.1: Overall flow of the process migration

Chapter 4

Implementation

This section describes the implementation details of different utilities referenced in the previous section. We depict the code flow of these utilities and describe the user-level and kernel-level interaction. We also mention the code changes in the standard BLCR library. We tried to keep changes to the existing code to a minimum. We added several new structures, e.g., ioctl macros and functions, to the BLCR library.

4.1 Initialization and Restoration:

The cr_restore utility performs the initialization at the destination node. A switch decides if the restore corresponds to live migration or frozen migration. The cr_restore is a user level utility that interacts with BLCR via the ioctl interface. We added ioctl macros for restore calls in blcr_ioctl.h. blcr_ioctl.h consists of all the ioctl macros corresponding to the commands of user level utilities. The cr_restore performs the following operations at the user level:

1. It builds the user level restore request by populating the cr_restore_arguments structure. It connects to the kernel and issues the restoration request using cri_syscall function as cri_syscall(CR_OP_RESTORE_REQ, (uintptr_t)&req).

2. The cr_restore utility then performs a fork and the child process calls the restore_-child_main function. The parent process, however, waits for the restoration to complete by calling cri_syscall(CR_OP_RESTORE_DONE, (uintptr_t)NULL) .

3. The parent process then fills in the signal structure and performs a reap on the restoration request. The child process issues the RESTORE_CLONE and RESTORE_CHILD request via the ioctl interface before exiting.

These functions are invoked at the user level. They, in turn, call their kernel level counterparts. The overall flow of cr_restore is depicted in Figure 4.1.

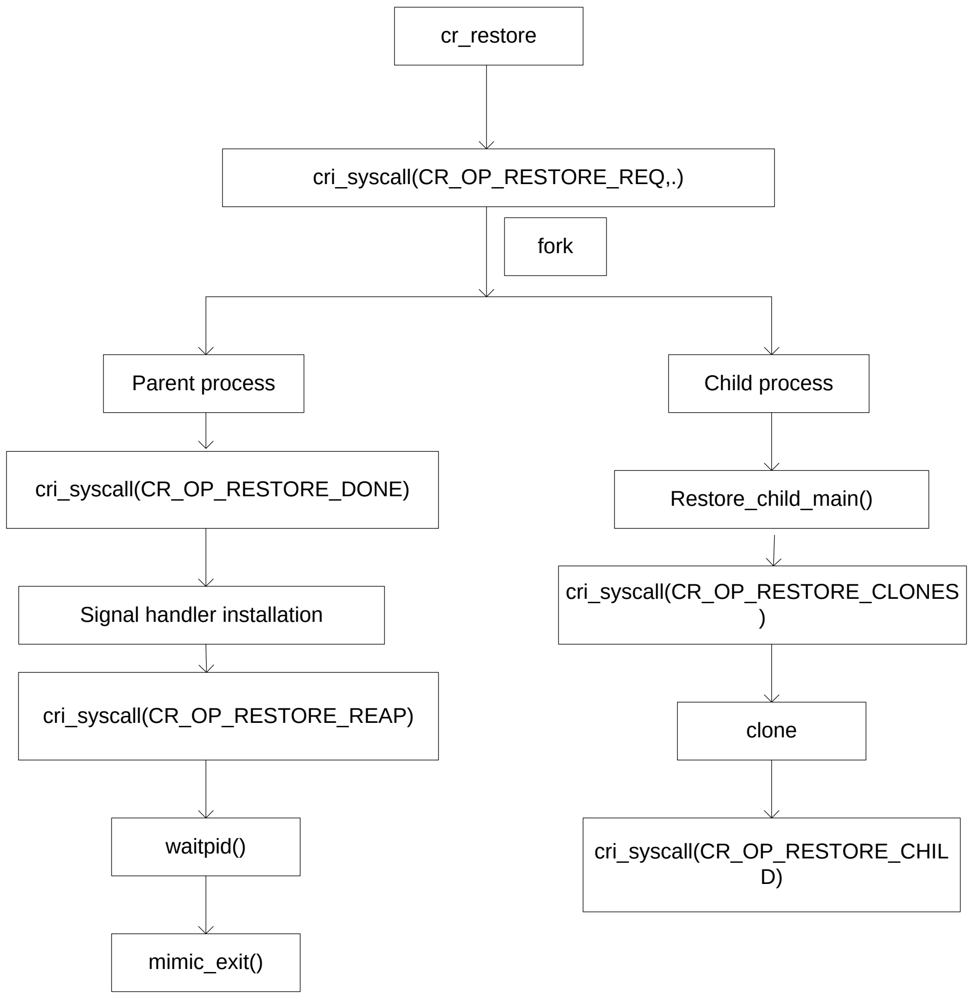

cr_restore
cri_syscall(CR_OP_RESTORE_REQ,.)
fork
Parent process
Child process
cri_syscall(CR_OP_RESTORE_DONE)
Restore_child_main()
cri_syscall(CR_OP_RESTORE_CLONES)
Signal handler installation
cri_syscall(CR_OP_RESTORE_REAP)
clone
waitpid()
cri_syscall(CR_OP_RESTORE_CHILD)
mimic_exit()

Figure 4.1: Overall flow of the cr_restore utility

cri_syscall(CR_OP_RESTORE_REQ, (uintptr_t)&req) sends a restore request to the kernel module. Every ioctl request is intercepted by ctrl_ioctl function defined in cr_fops.c. The ctrl_ioctl function matches the ioctl request with the corresponding function. It calls the cr_restore_request function at CR_OP_RESTORE_REQ ioctl call. The cr_restore_request function initializes the kernel level restore request and creates a socket. It issues a series of blocking read calls to receive the data from the source node. It first reads the number of threads in the migrated process, then copies this value in the restore request and returns. The restore_child_main function performs the task of thread creation by invoking the clone system call. It calls the cri_syscall(CR_OP_RESTORE_CLONES) function in a loop to create the required number of threads. The cr_restore_clones function decrements the clone_needed variable and allocates a one page stack to every cloned process. The restore_child_main function then calls the cr_restore_child function, which populates the remaining fields of restore request and calls cr_start_self function. Subsequent process restoration is performed by the cr_start_self function in the following steps:

1. The cr_start_self function reads the linkage structure sent by the source node and allocates the memory for linkage structure. The linkage structure of a process consists of pid, ppid, session information, group leader information and process group information.

2. It then reserves the pid, tgid and session ids to create the process. BLCR tries to create the process with same pid and tgid if possible. If the requested pid and tgid is not available, then it fails with an error message.

3. The third step is to restore the credentials of the process. Credentials include userid and group id. The userid and groupid are received from the source node and the exact same values are used for calls to the sys_setresuid and the sys_setresgid functions.

4. The fourth step is to restore the register values and map the received pages into the process address space. This is achieved by the virtual memory area dumper (vmadump) interface integrated in the BLCR. Vmadump assumes that it is operating on the current process and does not handle multi-threaded processes. To overcome this limitation, BLCR uses a wrapper function around vmadump functions to separate

the thread-specific requirement from the process wide requirement. The mechanism of register restoration and page mapping involves the following steps:

(a) The 'cpu registers' restoration is triggered by the start_vmadump_thaw_threads function. This function is called by all the threads in the process. All the threads but one wait at the barrier while one thread performs the actual restoration and page mapping.

(b) The thaw_threads function calls the start_vmadump_thaw_proc function, which receives the pages from the source node and maps them to the current process.

(c) The thaw_proc function then calls the start_vmadump_restore_cpu function, which restores the register values. This function is highly dependent on the architecture. We provide two versions of this function corresponding to the i386 and the x86_64 architectures.

5. The cr_start_self function then tries to restore the linkage information of the process. This is one of the most crucial steps for the restoration of a multi-threaded application. This part of the code is susceptible to kernel changes. This task is carried out by the cr_start_restore_linkage function in the following way:

   (a) It stops the process at a barrier so that none of the threads try to execute until the process linkage is restored.

   (b) It iterates through the task list of the restore request and manually restores the parent and real_parent values.

   (c) It then restores the group_leader value for every task.

   (d) It removes the process linkage of the thread group leader by unhashing it. It then attaches the pid and tgid value of the task. Subsequently, the process group and session leader information is restored. We finally restore the self_exec_id and parent_exec_id.

   (e) The pid, tgid and exec_ids are restored for the non-group leader tasks.

6. The sixth step is to restore the fs_struct of the migrated process. The cr_start_restore_fs_struct function restores the umask, reads the current working directory and changes the current working directory.

7. Open files are restored at last by the cr_start_restore_all_files function. This function loads the file structure by reading from the socket. It uses this information to retrieve the file type. It then tries to restore open file, open directory, open link, open socket, open fifo, open chr, open blk and open dup.

4.2 Memory Precopy:

Memory precopy is performed at the source node by the cr_migrate utility. Memory pages constitute the maximum amount of the process related data. The idea is to transfer the memory pages to the destination node while the process is still running at the source node. The cr_migrate utility connects to the kernel and issues a migration request invoking the cr_migrate_req function. The cr_migrate_req function builds the migration request and raises a signal indicating a migration request. This signal is then caught and handled in the user space by the cri_sig_handler function defined in cr_core.c. The cri_sig_handler calls a handler corresponding to the request. For a migration request, it calls the do_-create_manager_thread function. This function creates the precopy thread to transfer the memory pages to the destination node. The precopy thread performs the following tasks:

1. The precopy thread creates a socket and connects to the destination node.

2. It then issues an ioctl call to access the number of threads associated with the process and writes this information to the socket.

3. It counts the number of memory maps by issuing another ioctl call and sends it to the destination node.

4. The precopy thread then starts sending the memory pages in an iterative manner. In the first iteration, it sends all the pages to the destination node and clears the dirty bit.

5. In every subsequent iteration, it sends only the modified pages since the last iteration to the destination node and clears the dirty bit. It keeps iterating until the dirty_page_count and the difference between the dirty_page_count in two subsequent iterations is greater than 256.

6. It then closes the socket and returns.

We use the dirty bit approach to keep track of dirty pages. It uses a kernel patch to copy the dirty bit maintained by the kernel to the user space [27]. The kernel uses the dirty bit to keep track of modified pages. The patch uses the free bits in the page table entry (PTE) to maintain the status of the dirty bit for a given page.

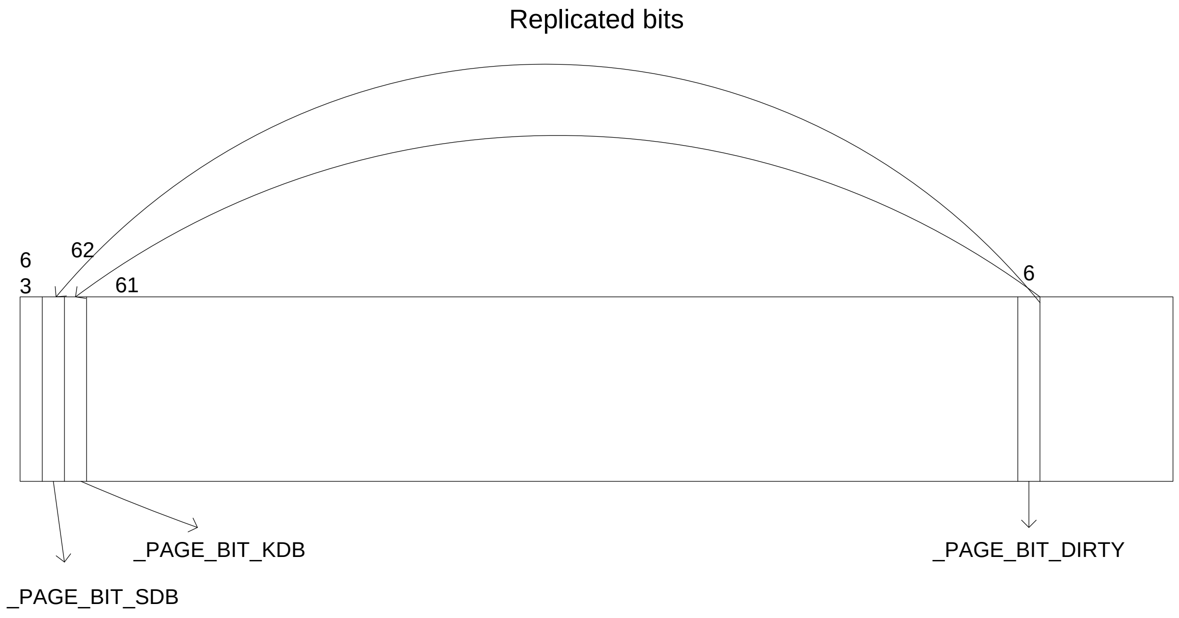


Figure 4.2: Dirty bit tracking scheme

As shown in Figure 4.2, Linux uses a bit to keep track of modified pages. The patch replicates this bit into two unused bits of the page table entry. The _PAGE_BIT_KDB maintains the kernel state of the dirty bit while _PAGE_BIT_SDB maintains the user state of the dirty bit. The kernel bit maintains the consistency of the dirty bit and returns its value for the last access. The user bit, however, maintains the state of the dirty bit between two function calls. This patch provides the DB_test_and_clear macro to test the dirty bit of a page and reset it. We use this macro in the is_dirty function to detect and clear the dirty bit for a page.

4.3 Stop-and-Copy:

Once all the MPI tasks reach a consistent global state, the process on the source node freezes, sends the remaining pages, the process state, descriptors of open files, linkage information, pipes and other process-related information to the destination node. The cr_stop utility signals the process on the source node to freeze the execution. Threads of the process subsequently copy their state information to the destination node. This task is performed inside the kernel. We can invoke the cr_stop utility without the prior invocation of cr_migrate on the source node thus avoiding the memory precopy phase. However, there are tradeoffs between frozen migration and the alternative live migration, which allows precopy with continued application progress. Live migration comes at the expense of background network activity although it may result in an overall shorter application downtime. There is also a high probability of repeated transmission of dirty pages. Figures 4.3 and 4.4 show the steps in live and frozen (without memory precopy) migration.

The cr_stop command signals the process on the source node to stop the execution and send the dirtied pages in the last iteration of the precopy step to the destination node. Threads of the process then take turns to send their state information. These tasks are performed inside the Linux kernel. The lower half of the Figure 4.3 illustrates these steps. The cr_stop issues the stop request via an ioctl call and waits until it is complete. The stop request triggers the following actions:

1. It allocates a task request, builds a task list and calls the cr_trigger_stop1 function. The stop request is handled by the cr_stop_req function defined in cr_stop_req.c.

2. The trigger function populates a siginfo structure to uniquely identify the stop request and raises the signal. We use the si_uid field in the siginfo structure to identify different types of requests. For example, the si_uid value of the stop request is 936316 while the si_uid value of the migrate request is 936315.

3. The signal is caught and handled by the cri_sig_handler function defined in cr_core.c. The handler function calls the do_stop function for a stop request. Since the handler executes in user space, do_stop issues another ioctl call to make the transition to the kernel space. The do_stop function calls the cri_syscall(CR_OP_HAND_STOP, flags) function, which in turn invokes the cr_stop_and_copy_self function defined in the corresponding C file.

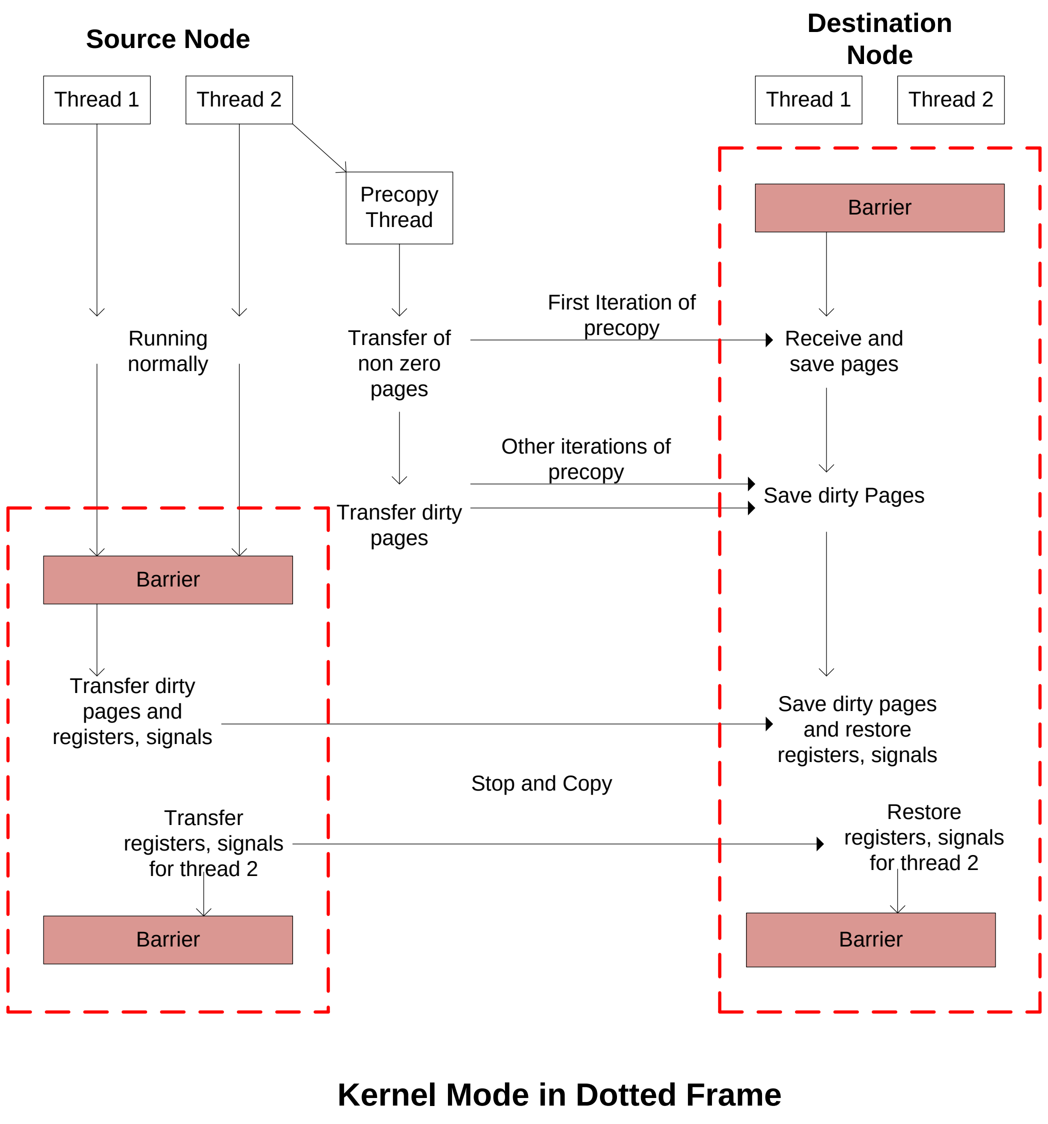
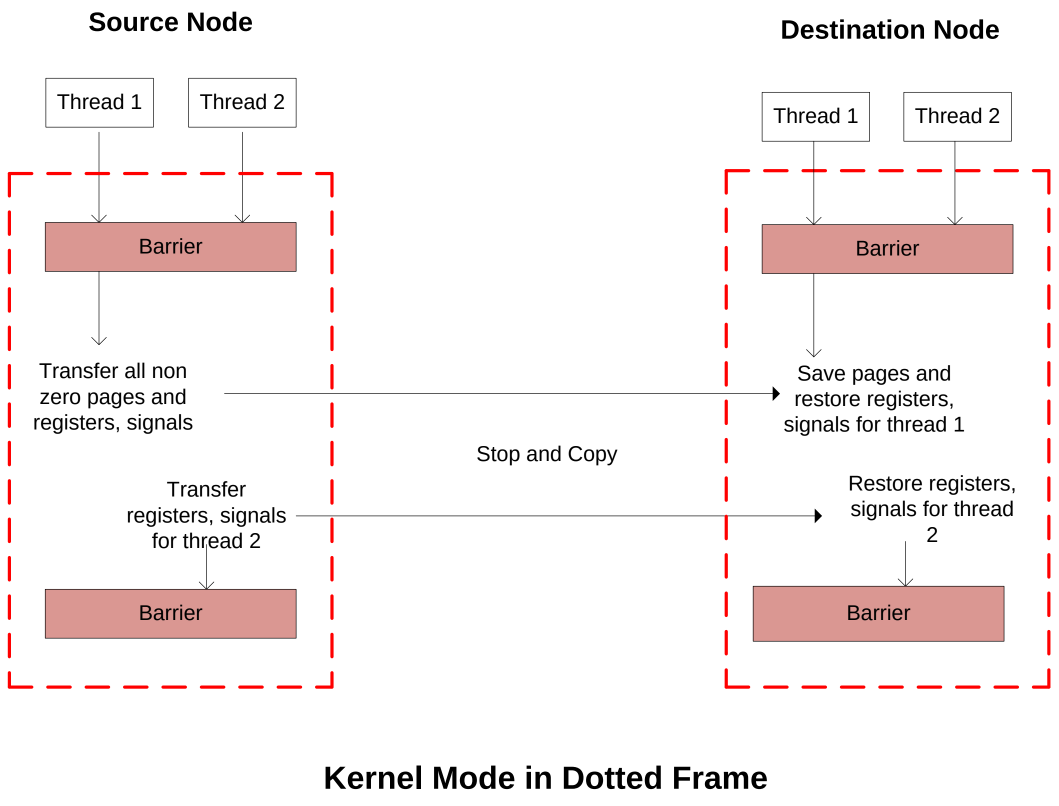

Source Node
Destination Node
Thread 1
Thread 2
Thread 1
Thread 2
Precopy Thread
Barrier
Running normally
Transfer of non zero pages
First Iteration of precopy
Receive and save pages
Other iterations of precopy
Transfer dirty pages
Save dirty Pages
Barrier
Transfer dirty pages and registers, signals
Save dirty pages and restore registers, signals
Stop and Copy
Transfer registers, signals for thread 2
Restore registers, signals for thread 2
Barrier
Barrier
Kernel Mode in Dotted Frame

Figure 4.3: Migration with precopy

4. The cr_stop_and_copy_self function locates the current process to find the matching stop request. The process then hits the predump barrier and calls the cr_do_stop_-vmadump function.

5. The cr_do_stop_vmadump function saves the process control block in a step-by-



Source Node
Destination Node
Thread 1
Thread 2
Thread 1
Thread 2
Barrier
Barrier
Transfer all non zero pages and registers, signals
Save pages and restore registers, signals for thread 1
Stop and Copy
Transfer registers, signals for thread 2
Restore registers, signals for thread 2
Barrier
Barrier
Kernel Mode in Dotted Frame

Figure 4.4: Migration without precopy

step manner. It first calls the cr_stop_save_linkage function, which connects to the destination node and writes the process linkage structure. The process linkage includes the parent, real_parent, group_leader, session, leader, pid, tgid, pgrp, exit_-signal and the file pointer associated with the stop request. If the stop request is not preceded by the memory precopy step then it sends the task count to the destination node before sending other process-related information. The task count is equal to the number of threads in the process.

6. We send the credential information to the destination node once the save_linkage function returns. The credential information includes uid, euid, suid, gid, egid and sgid. We use the cr_context_creds structure defined in BLCR to store these fields. The complete structure is written to the socket in one pass.

7. The stop_vmadump function then calls the stop_vmadump_freeze_threads function, which is similar to the vmadump_freeze_threads function in the standard BLCR code. The stop version of this function works on a socket instead of a file descriptor.

8. The freeze_thread function calls the stop_vmadump_freeze_proc function, which stops the process execution, writes the memory info to the socket and starts copying pages to the destination node. The page copying process is very similar to the memory precopy stage but instead of performing it iteratively, we copy all the pages in one pass.

9. The next step is to copy the register information. This task is carried out by stop_vmadump_store_cpu function. We have two different versions of the store_cpu function. One version is for i386 architecture while the other version handles the dumping request for x86_64 architecture.

10. The freeze_proc function then writes the signal information and other miscellaneous information to the socket and returns.

11. The freeze_threads function return value indicates the completion of register and signal writing to the socket. The next function is stop_cr_save_fs_struct to save the umask and current working directory of the current process.

12. The next step is to save all open files. This step is carried out by the stop_cr_save_all_files function. This function iterates through all file descriptors held by the process, saves the corresponding file header and file info. It then calls different functions based on the file information. In the current version of BLCR, we save open files, directories, fifo and pipes.

Saving the file information completes the dumping process. The stop_and_copy function then sends the saved signals to the current process and returns. The complete format of dumping is depicted in Figures 4.5 and 4.6. This is very close to the BLCR checkpoint file format but differs in the context file header and a few other fields.

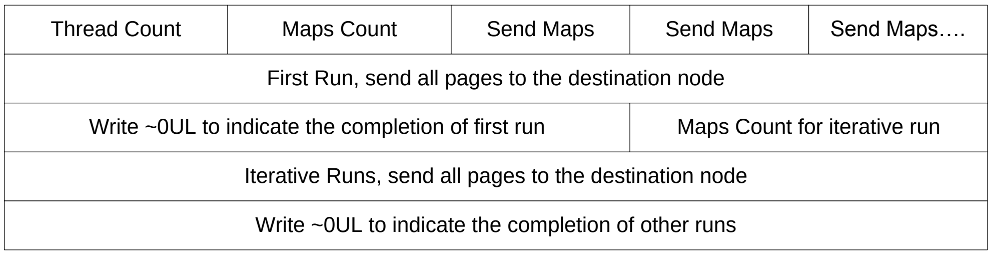

Thread Count
Maps Count
Send Maps
Send Maps
Send Maps….
First Run, send all pages to the destination node
Write ~0UL to indicate the completion of first run
Maps Count for iterative run
Iterative Runs, send all pages to the destination node
Write ~0UL to indicate the completion of other runs

Figure 4.5: Sequence of data sent during precopy

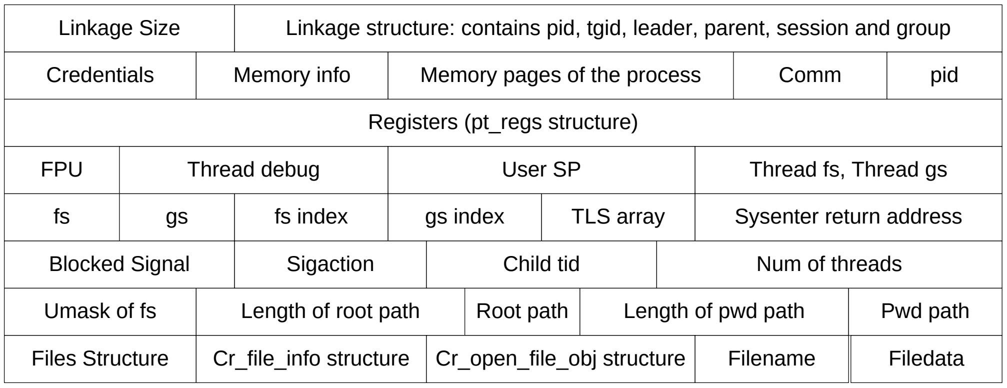


Figure 4.6: Sequence of data sent during stop-and-copy

Chapter 5

Experiments

5.1 Framework

We conducted our experiments on the local Opt cluster. This cluster has 18 nodes running Fedora Core 13 Linux x86-64 connected by two Gigabit switches. Each node is equipped with 2-way SMPs with dual-core AMD Opteron 265 processors. It has a 750 GB RAID5 array to provide storage through NFS and a local disk. We conducted timing measurements in user space for live and frozen migration.

5.2 Experiments

We use three BLCR test codes to test the functionality of migration utilities. These test codes include a single-threaded counting application, a multi-threaded counting application and a file counting application. In addition to these tests, we also developed a micro benchmark to assess the overhead of different approaches of migration. The experiments are designed to test the functionality of the migration utilities and to analyze their performance for applications of different sizes.

5.2.1 Instrumentation Techniques

We augmented the timing measurements with migration utilities at the user level. We used the timing utilities provided by Linux to measure the time at a micro second granularity. We record the time via gettimeofday() right before and after the migration request and determine the difference. We then send the data over a network via sockets. Hence, network fluctuation can result in varied timings for different runs. To mitigate these local effects, we perform the experiments when the cluster is least used and determine the average over four samples per experiment. We observe a variation of 7% in timing measurements for large applications. We observe large variations in timings for small applications, which can be attributed to small execution times. For memory experiments, we used large size applications to get a consistent pattern.

5.2.2 Memory Tests

The memory test is to assess the overhead associated with an application migration. We take separate timings for the memory precopy and the stop-and-copy phases of live migration. We designed a micro benchmark that allocates a large number of pages and dirties those pages in an iterative manner. We also defined a stride for dirtying pages. The test was conducted on a large data set to mitigate the effect of external factors. We observe that the migration time is directly proportional to the number of pages allocated to the application, which in turn defines the application size. The number of pages allocated for the following experiment are 1000, 10000, 100000, and 500000. The results are presented in Figures 5.1 and 5.2 for strides one and ten, respectively.

We observe that migration time increases with an increase in the number of pages of the application. The increase in time is almost linear. We also measured the time for live and frozen migration combined (precopy + stop-and-copy) and compared it with frozen migration (only stop-and-copy) alone. As we discussed in the previous sections, there is a tradeoff between live and frozen migration. Live migration takes a longer time to complete but it lets the application run while the memory pages are being transferred to the destination node. Precopy is immediately followed by a stop-and-copy which takes less time than the stand-alone frozen migration. These results are depicted in Figure 5.3.

We observe that precopy migration takes more time than stop-and-copy and frozen migration. In the precopy phase, we keep iterating until we reach a steady state. In each iteration, we only send the modified pages since the last iteration. For a large application, each iteration can take a significant amount of time. However, the application keeps running in the precopy phase. We also observe that stop-and-copy takes less time than frozen migration although they involve same steps. Stop-and-copy sends only those pages to the destination node that have been dirtied since the last iteration of precopy. Frozen

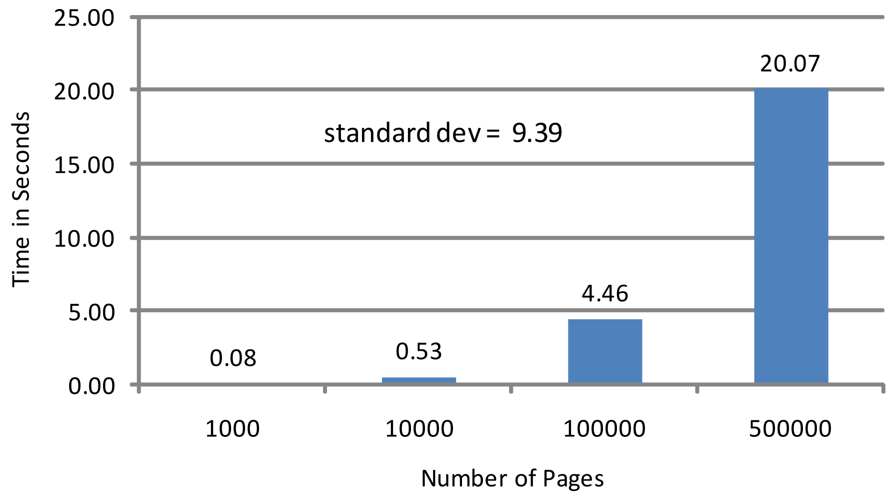


Figure 5.1: Stop-and-Copy Timings with a Stride of One

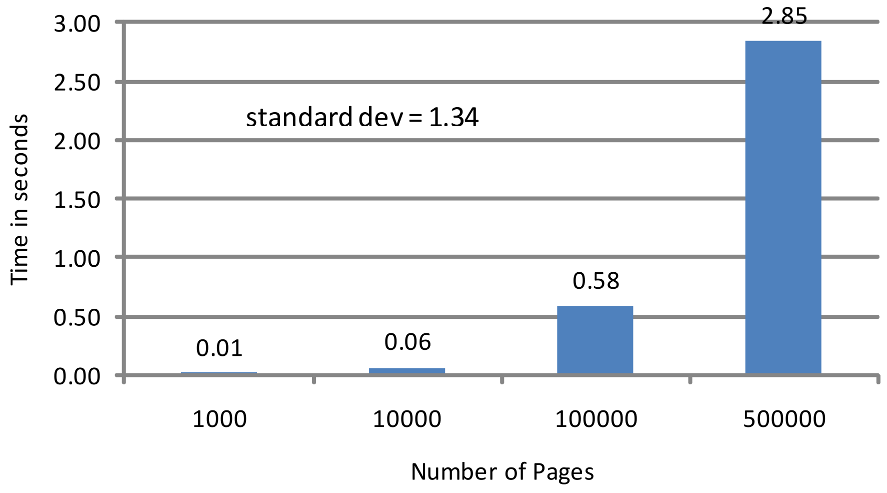


Figure 5.2: Stop-and-Copy Timings with a Stride of Ten

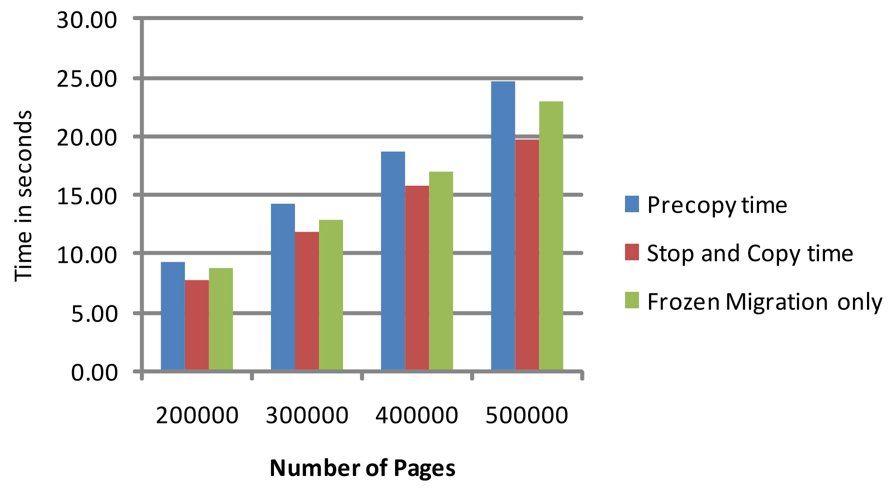


Figure 5.3: Timings for Live and Frozen Migration

migration, on the other hand, essentially sends all non-zero pages to the destination node. Therefore, the application down time is less in live migration than frozen migration at the cost of higher network activity.

5.2.3 Other Tests

We ran several other tests to verify the functional integrity of the migration utilities. These tests involve different types of applications. Some of the important tests are described below:

1. We tried to migrate a multi-threaded application using the standard BLCR code. This program creates three threads and increments a global counter in a loop. Each thread takes turns to increment the counter. Multithreaded applications are tricky to migrate because we have to keep track of thread group leader and synchronize the threads at the destination node. The linkage restoration function ensures that each thread has the correct group leader associated with it. We are able to migrate the multi-threaded application using both live and frozen migration. We also tested

the signal handling of the migrated process at the destination node and it works fine. We ran these tests with and without cloning.

2. We migrated a file counting application successfully to the destination node. This application opens a file and starts writing to it. The migrated process restores the file at the destination node and starts writing at current file pointer. We used a network file for this test. This test, however, works only in absence of cloning. If we enable cloning, then two processes would try to write into the file at the same time and the process would fail after cloning.

Besides these test cases, we tried to migrate several variations of counting and memory mapped applications. There is still room to add other test cases for different scenarios. Some of the more sophisticated test cases can be designed more easily once the migration utility is integrated with OpenMPI.

Chapter 6

Related Work

This thesis aims to present a fault tolerance technique to aid the conventional C/R method for a HPC application. In this chapter, we talk about the research related to fault tolerance in MPI jobs and migration approaches in general.
The feasibility of proactive fault tolerance has been demonstrated at the job scheduling level [23], within OS virtualization [31] and in Adaptive MPI [26, 5, 4] using object virtualization and message logging [12] for Charm++ applications. Our fault tolerance approach, however, is at the process level and independent of the MPI runtime layer. Our approach encapsulates most of the process control block, including open file descriptors, pipes and sockets.
There are two aspects of proactive fault tolerance techniques. The first aspect is failure prediction and there are a number of research efforts put in this direction [25, 14, 15]. These papers report high failure prediction accuracy with a warning window, which is the premise for our proposed process migration mechanism. The second aspect of proactive fault tolerance is the migration mechanism adopted for a particular environment. Various migration techniques have been developed in the past [21, 24, 29, 18, 2, 7, 11]. MPI-Mitten [10], an MPI library between the MPI and application layer, provides proactive fault tolerance to MPI applications. It uses HPCM [8] as a middleware to support user-level heterogeneous process migration. In [30], the authors provide a generic framework based on a modular architecture to implement a proactive fault tolerance mechanism. An agent-oriented framework [17] was developed for grid computing environments. In this framework, agents monitor classes or subclasses of faults and either tolerate them or take corrective actions. Sun et al. provide fault-aware systems, such as FARS [20] and

FENCE [28], to increase the accuracy of fault prediction and improve system resilience to failures with different fault management mechanisms including process migration. They also model the migration cost and introduce a suitable scheduling mechanism [9]. These prior works with their fault models, mechanisms and their evaluation schemes assert that process migration is a suitable approach for proactive fault tolerance. It involves less overhead than operating system (OS) virtualization, which reinforces the significance of our approach.

Our proactive live migration solution provides new BLCR capabilities and supports continued execution of applications during migration. It parallels live migration at the OS virtualization layer [6], which has been studied in the context of proactive fault tolerance of MPI applications [22], an approach that supports health monitoring and live migration over Xen guests. We contribute process-level migration and demonstrate its efficiency. In HPC, process-level solutions are more widely accepted than OS virtualization because of its simplicity and less overhead. Hence, our contribution has significant potential to have practical impact.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

We present two different approaches for migration, namely live and frozen migration. We also present a mechanism to clone the process. They complement reactive fault-tolerant mechanisms, such as checkpointing, resulting in a reduction in the number of required checkpoints. Checkpoint/restart can be deployed together with incremental checkpointing and live migration (1) to handle the faults proactively, (2) to reduce the number of required full checkpoints, (3) to reduce the checkpoint overhead and, (4) to clone the process thus providing redundancy. We compared the performance of two approaches and showed the tradeoff between continued execution and application downtime. We presented the results of both approaches for a variety of process migration scenarios and for a memory benchmark. Overall, this confirms our hypothesis that we can migrate a process from one node to another node via sockets, reduce overheads and create process clones.

7.2 Future Work

There is a lot of scope for future work related to this thesis. Some of them are discussed below:

7.2.1 Integration with OpenMPI

The migration techniques are designed to work with multi-node processes. As of now, any single-node, multi-threaded process can be migrated to a physical node using the migration commands. However, we have to integrate the migration code, e.g., with the OpenMPI run time layer to migrate an MPI process. The implementation would be very similar to the cr_checkpoint integration with OpenMPI. We need to define a suitable interface for the MPI run time layer to execute migration commands and define call-backs to return from BLCR code to OpenMPI process execution. We also have to synchronize the processes at different stages of migration. For example, once the migrating process returns from cr_migrate, all the MPI processes should hit a barrier before the stop-and-copy phase can be started. Figure 3.1 depicts the required steps.

7.2.2 Cloning

Process cloning can be supported as follows: In our cloning mechanism, we do not kill the process at the source node once migration is complete. We do it using the cr_clone utility. This utility works similar to cr_stop utility but it allows the process to run at the source node thus creating a clone. Cloning works fine in single process environment because these two processes are independent of each other. In the MPI environment, however, two processes with the same signature cannot co-exist. We need to modify MPI run time layer to make sure these processes do not conflict each other and only one process is active in MPI environment at a time.

7.2.3 Using WB Approach

As explained in Chapter 4, we use the dirty bit approach to keep track of the modified pages. The dirty bit approach requires a small kernel patch. Although the dirty bit approach is very efficient, it is not always viable to apply the dirty bit patch due to security reasons or insufficient privileges. Sometimes, the patches are dependent on other patches and we may need to apply a lot of other patches before the dirty bit patch can be applied.
There is another approach to track modified pages, namely the write bit approach. This approach uses the page protection mechanism to lock a page for writing. When a page is modified, a page fault occurs and the corresponding handler marks the page as dirty.

The write bit approach does not require any kernel modification. Vasavada et al. used the write bit approach for incremental checkpointing [32].

7.2.4 Using File Descriptors

As of now, we send and receive the process date via network sockets. Most of the BLCR utilities, however, work on a file descriptor. We would like to tweak the code to use a file pointer instead of sockets to make the design more general. Moreover, most of the BLCR functions are specific to checkpoint and restart requests, which limit the code re-usability. We would like to change the functions to make them independent of request types.

As part of the future work, we would like to test the live migration utility on bigger systems in an MPI environment. We would like to test it for a wide variety of applications on different architectures. We would also like to make the code more robust by adding error handling and testing corner cases. We have developed the migration utility within the latest BLCR release and would like to test the migration utility in co-ordination with the checkpoint/restart utilities in this context.

REFERENCES

[1] Top 500 supercomputers list. `http://www.top500.org/list/2011/06/100`.

[2] Amnon Barak and Richard Wheeler. Mobility. chapter MOSIX: an integrated multiprocessor UNIX, pages 41–53. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[3] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, and Anton Selikhov. Mpich-v: toward a scalable fault tolerant mpi for volatile nodes. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–18, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[4] Sayantan Chakravorty and L. V. Kale. A fault tolerance protocol with fast fault recovery. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*. IEEE Press, 2007.

[5] Sayantan Chakravorty, Celso L. Mendes, and Laxmikant V. Kalé. Proactive fault tolerance in mpi applications via task migration. In *HiPC*, volume 4297 of *Lecture Notes in Computer Science*, pages 485–496. Springer, 2006.

[6] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[7] Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software - Practice and Experience*, 21:757–785, 1991.

[8] Cong Du and Xian he Sun. Hpcm: A precompiler aided middleware for the mobility of legacy code. In *in Proc. IEEE Cluster Computing Conference, Hong Kong*, 2003.

[9] Cong Du, Xian-He Sun, and Ming Wu. Dynamic scheduling with process migration. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '07, pages 92–99, Washington, DC, USA, 2007. IEEE Computer Society.

[10] Cong Du and Xian-He Sun Sun. Mpi-mitten: Enabling migration technology in mpi. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 11 – 18, may 2006.

[11] Jason Duell. The design and implementation of berkeley labs linux checkpoint/restart. Technical report, 2003.

[12] E.N. Elnozahy and W. Zwaenepoel. Manetho: transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *Computers, IEEE Transactions on*, 41(5):526 –531, may 1992.

[13] 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.

[14] Xiaohui Gu, Spiros Papadimitriou, Philip S. Yu, and Shu-Ping Chang. Toward predictive failure management for distributed stream processing systems. In *Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, ICDCS '08, pages 825–832, Washington, DC, USA, 2008. IEEE Computer Society.

[15] Prashasta Gujrati, Yawei Li, Zhiling Lan, Rajeev Thakur, and John White. A meta-learning failure predictor for blue gene/l systems. In *Proceedings of the 2007 International Conference on Parallel Processing*, ICPP '07, pages 40–, Washington, DC, USA, 2007. IEEE Computer Society.

[16] Chung-hsing Hsu and Wu-chun Feng. A power-aware run-time system for high-performance computing. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 1–, Washington, DC, USA, 2005. IEEE Computer Society.

[17] Mohammad Tanvir Huda, Heinz W. Schmidt, and Ian D. Peake. An agent oriented proactive fault-tolerant framework for grid computing. In *Proceedings of the first International Conference on e-Science and Grid Computing*, pages 304–311, Washington, DC, USA, 2005. IEEE Computer Society.

[18] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Trans. Comput. Syst.*, 6:109–133, February 1988.

[19] Jayesh Krishna, Pavan Balaji, Ewing Lusk, Rajeev Thakur, and Fabian Tillier. Implementing mpi on windows: comparison with common approaches on unix. In *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, EuroMPI'10, pages 160–169, Berlin, Heidelberg, 2010. Springer-Verlag.

[20] Yawei Li, P. Gujrati, Zhiling Lan, and Xian he Sun. Fault-driven re-scheduling for improving system-level fault resilience. In *Parallel Processing, 2007. ICPP 2007. International Conference on*, page 39, sept. 2007.

[21] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32:241–299, September 2000.

[22] Arun Babu Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive fault tolerance for hpc with xen virtualization. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, pages 23–32, New York, NY, USA, 2007. ACM.

[23] A. J. Oliner, R. K. Sahoo, J. E. Moreira, M. Gupta, and A. Sivasubramaniam. Fault-aware job scheduling for bluegene/l systems. In *In IEEE IPDPS, Intl. Parallel and Distributed Processing Symposium*, pages 64–73, 2004.

[24] Michael L. Powell and Barton P. Miller. Process migration in demos/mp. In *Proceedings of the ninth ACM symposium on Operating systems principles*, SOSP '83, pages 110–119, New York, NY, USA, 1983. ACM.

[25] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '03, pages 426–435, New York, NY, USA, 2003. ACM.

[26] Sayantan Chakravorty, Celso Mendes and L. V. Kale. Proactive fault tolerance in large systems. In *HPCRI Workshop in conjunction with HPCA 2005*, 2005.

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

[28] Xian-He Sun, Zhiling Lan, Yawei Li, Hui Jin, and Ziming Zheng. Towards a fault-aware computing environment. In *HAPCW*, March 2008.

[29] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the v-system. In *Proceedings of the tenth ACM symposium on Operating systems principles*, SOSP '85, pages 2–12, New York, NY, USA, 1985. ACM.

[30] Geoffroy Vallee, Kulathep Charoenpornwattana, Christian Engelmann, Anand Tikotekar, Chokchai Leangsuksun, Thomas Naughton, and Stephen L. Scott. A framework for proactive fault tolerance. In *Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*, pages 659–664, Washington, DC, USA, 2008. IEEE Computer Society.

[31] Jyothish Varma, Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Scalable, fault tolerant membership for mpi tasks on hpc systems. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 219–228, New York, NY, USA, 2006. ACM.

[32] Manav M Vasavada. *Innovative Schemes to Support Incremental Checkpointing.* PhD thesis, North Carolina State University, 2009.

[33] Chao Wang, F. Mueller, C. Engelmann, and S.L. Scott. Hybrid checkpointing for mpi jobs in hpc environments. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 524 –533, dec. 2010.

[34] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. A job pause service under lam/mpi+blcr for transparent fault tolerance. *Parallel and Distributed Processing Symposium, International*, 0:117, 2007.

[35] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration in hpc environments. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 43:1–43:12, Piscataway, NJ, USA, 2008. IEEE Press.