ADVANCED I/O TECHNIQUES FOR EFFICIENT AND HIGHLY AVAILABLE PROCESS CRASH RECOVERY PROTOCOLS

Jason Warren Cornwell
Columbus State University

The College of Business and Computer Science

The Graduate Program in Applied Computer Science

Advanced I/O Techniques for Efficient and

Highly Available Process Crash Recovery Protocols

A Thesis in
Applied Computer Science
by

Jason Warren Cornwell

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

March 2011

©2011 by Jason Warren Cornwell
I have submitted this thesis in partial fulfillment of the requirements for the degree of Master of Science.

Date  
Jason Cornwell

We approve the thesis of Jason Warren Cornwell as presented here.

Date  3/28/2011  
Angkul Kongmunvattana. Ph.D.  
Associate Professor.  
Thesis Advisor

Date  March 28th, 2011  
Eugen Ionascu. Ph.D.  
Professor of Mathematics

Date  March 28th, 2011  
Wayne Summers. Ph.D.  
Distinguished Chairperson and Professor of Computer Science
Acknowledgments

First and foremost, I would like to thank Professor Angkul Kongmunvattana. He has been an outstanding advisor and role model. Without his guidance and trust, many of my proudest accomplishments during the past two years would never have come to pass. I have learned many life and career lessons during our collaboration and hope that I have also shown him about life’s many blessings.

I would also like to thank Justin Whaley for setting up and configuring the NFS cluster used in this experiment. It has been enjoyable to have his assistance as well as the opportunity to share some of what I have learned and experienced with him.

I am deeply grateful to my parents Larry and Judy Cornwell. In recent years, they have taught me to stand up for what I believe in, even when it meant standing alone. Their love and support have provided me power to accomplish more than I could have imagined. Words cannot express my love for them both.
Abstract

As the number of CPU cores in high-performance computing platforms continues to grow, the availability and reliability of these systems become a primary concern. As such, some solutions are physical (i.e., power backup) and some are software driven. Lawrence Berkeley National Laboratory has created a system-level fault-tolerant checkpoint/restart implementation for Linux Clusters. This allows processes to restart computations at the last known checkpoint in the event the system crashes. The checkpoint data creation is highly dependent on system input and output operations. This paper proposes: (i) a technique to improve the efficiency of these I/O operations and (ii) an alternative checkpoint creation method to increase availability and reliability of checkpointing data.

Keywords: caching, checkpoint, crash recovery, fault-tolerant system, high performance computing, network file system, optimization, performance evaluation, remote checkpointing
# Table of Contents

Acknowledgments .............................................................................. iii  
Abstract ............................................................................................... iv  
List of Figures ....................................................................................... vii  
List of Tables ......................................................................................... viii  
1 Introduction ....................................................................................... 1  
2 Pertinent Background ...................................................................... 3  
3 Proposed Technique .......................................................................... 7  
  3.1 I/O Optimization ......................................................................... 7  
    3.1.1 Checkpoint Caching ................................................................. 7  
    3.1.2 Optimized BLCR's Write Operation ........................................... 9  
    3.1.3 Implementation ...................................................................... 10  
  3.2 Remote Checkpoint ...................................................................... 12  
    3.2.1 Remote Checkpoint Server ...................................................... 12  
    3.2.2 Modified BLCR's Write Operation .......................................... 14  
    3.2.3 Implementation ...................................................................... 15  
4 Experimental Setup ........................................................................... 18  
5 Experimental Results ......................................................................... 21  
  5.1 NP-Complete Problem ................................................................. 24  
  5.2 Data Encryption ........................................................................... 25  
  5.3 Linear Equation Solver ............................................................... 27  
  5.4 File Compression .......................................................................... 29
5.5: Summary .......................................................... 30

6 Conclusion and Future Research .................................. 32

Bibliography .................................................................. 35
List of Figures

1  Pseudo Code for the Caching Process .............................................. 8
2  Checkpoint Caching Protocol ............................................................. 10
3  Pseudo Code for the Modified BLR’s Write ........................................ 11
4  Pseudo Code for the Server Process ................................................... 13
5  Pseudo Code for the Modified BLR’s Write ........................................ 14
6  Remote Checkpoint Protocol .............................................................. 16
7  Local Write Overhead for Checkpoint Creation .................................... 26
8  Remote Write Overhead for Checkpoint Creation .................................. 28
## List of Tables

1. Overhead Trend of Benchmark Programs ........................................... 21
2. Traveling Sales Problem Results ......................................................... 22
3. Advanced Encryption Standard Results .............................................. 22
4. Gaussian Elimination Results ............................................................. 23
5. Huffman Compression Results ............................................................ 23
6. Remote Checkpoint Results ............................................................... 24
1 Introduction

Multicore systems have become a common platform for high-performance computing. As the number of CPU cores in these systems grows, the issue of permanent or transient failures is a major concern. System failure of any nature continues to threaten reliability as complexity and process duration increase. Current techniques favor redundancy of resources to prevent or marginalize failure but this is not always an adequate solution. Lawrence Berkeley National Laboratory has attempted to solve this problem with a system-level fault-tolerant roll-back module [8, 13] for the Linux environment.

Berkeley Lab Checkpoint/Restart (BLCR) aims at saving the status of a process and all related data to non-volatile storage in the event the task needs to be restarted or migrated to another system. The interval and/or storage location of the checkpoint data can adversely affect the overall performance of BLCR. Since checkpointing is heavily dependent on input and output (I/O) operations, the slowest form of memory manipulation, to create a process checkpoint, efficient I/O management can improve overall performance and reliability [4, 10]. Moreover, BLCR does not currently implement any advanced I/O management techniques to optimize disk operations. This paper proposes an efficient alternative method for checkpoint creation. Additionally, advanced output features are extremely limited within BLCR, especially in the area of task migration and/or non-transient failure. When checkpoint data is stored locally on the device performing the computations, the machine becomes a single point of failure. Thus, this paper also introduces a remote checkpoint technique. This alternative method of checkpoint creation is designed to provide redundancy if the local checkpoint is not accessible after system failure.
Our experimental results demonstrate that the proposed techniques reduces the latency of the write operations compared with the BLCR implementation. Secondly, higher system reliability can be achieved when compared with existing technologies.

This paper is organized into six chapters. Chapter 2 provides basic background pertinent to this paper. Our optimized checkpoint technique and remote checkpoint protocol is introduced in Chapter 3. The experimental environment, which includes the hardware platform for each technique, is described in Chapter 4. Our applications benchmarks and experimental results are discussed in Chapter 5. Finally, we conclude in Chapter 6.
2 Pertinent Background

Checkpoint/restart [6, 7, 11, 15, 21, 22] is a common feature in many high-performance computing (HPC) platforms. It is designed to save the execution state of a process, in real time, to reliable storage. This process recovery point contains a copy of the computational data such as stack, heap, and registers and possibly signals, signal handlers, etc. The recovery information can then be used by the Operating System (OS) to restart/reconstruct the process at a later time if the process terminates before successful completion.

The checkpointing ability provides two key benefits for HPC systems [20]. First, it enables a fault tolerant feature that is not currently provided by the original OS management. This allows manual or automated process checkpoint creation to increase process reliability. Second, it is a mechanism for task migration. In systems where virtual memory is extremely limited, where the operating system does not have sufficient resources to allocate memory to all runnable processes, the checkpointing feature may be used to capture the state of the process and remove the process from the execution queue. After a successful checkpoint is created, the process can be relocated to another machine or restarted when sufficient resources are available.

Many fault-tolerant techniques currently prefer an application-level checkpoint implementation due to the application-specific knowledge that can be gathered about the process. Chen et al. [6] presented a user-level library that provides semi-transparent checkpointing for commercial computers. Laadan et al. [15] described a technique for checkpointing distributed networks without library, kernel or network protocol modification. Dieter and Lumpp [7] discussed a user-level library for Unix platforms capable of handling multi-
threaded applications. Ruscio et al. [21] developed a new mechanism for checkpoint creation, migration and recovery at the user-level. Generally more efficient checkpoints can be created by using knowledge about the application’s design to reduce the I/O footprint. Although this is a key benefit of application-level checkpoint creation, Litzkow and Soloman [16] discuss limitations of user-level checkpointing.

System-level checkpointing has also been explored. Duell et al. [8] described how a kernel-level checkpoint is created. Gioiosa et al. [11] proposed a software architecture for a fault-tolerant procedure in Linux clusters. System-level checkpointing has a unique set of advantages over user-level checkpointing:

- Full resource restoration including process ID, session ID, etc.
- Limited restrictions on the number of processes that can be checkpointed since most data structures are accessible to the kernel
- Flexibility in initiating the checkpoint application due to system level preemption

A checkpoint at the kernel-level requires modification and/or linking to the kernel space. The current implementation uses a kernel module and user-shared libraries that can be easily deployed without recompiling or rebooting the kernel. When an application is marked for checkpoint, it first must be linked to predefined BLCR library codes. These libraries are loaded into the application at start up. If the program is not linked correctly, the checkpoint task will fail the checkpoint procedure gracefully.

Once BLCR is fully initialized, a registered callback thread is spawned. This thread is designed to block key system calls in the kernel until the checkpoint occurs.
the checkpoint is invoked, either manually or automated. the recovery point utility process begins. This unblocks the callback thread, which checks to see if any of the process/threads involved in the recovery creation has died unexpectedly. The callback thread now enters the kernel and signals all remaining threads that the checkpoint process is active. Each thread that interpreted the broadcast invokes the predefined library call routines of BLCR, allowing each thread to enter kernel-space.

Once all threads have completed their callback routines and have entered the kernel, the most challenging section of checkpoint creation begins: coordination and synchronization. Each thread initially reaches the first barrier where one thread is chosen to write shared header information and process state to local storage. After a successful write, the threads continue until the next barrier is reached. At this barrier, one thread will write all shared resources such as virtual memory, file descriptor, and other resources shared by the thread group. After the first thread completes, every thread, including the first thread, writes the process ID, registers, and signal handlers unique to each thread. This memory dump procedure continues until all relevant data about the process is successfully copied to the checkpoint file.

Once each thread completes the memory dump procedure, it continues until the final barrier is reached. If the checkpoint parameters specify that the application is to terminate after checkpoint creation, the task is removed from the process queue. Otherwise, each thread is allowed through the barrier and is returned to user-space where the task will run until completion or the checkpoint is re-initiated.

Currently, two main checkpoint functions are implemented: checkpoint creation and
process recovery. Two key factors limiting the checkpoint creation performance and availability are (i) the implementation of the I/O operations accessing disk storage and (ii) the predefined local disk checkpoint creation. Proposed in Chapter 3.1 is the I/O optimization technique. This procedure implements a caching operation to reduce the write latency of BLCR. Additionally, Chapter 3.2 describes the remote checkpoint protocol. This procedure allows for recovery point creation outside the local disk storage.
3 Proposed Technique

The methodology behind checkpointing is to create a snapshot of a process state during execution and store the process recovery point on reliable media. Computations can then be reconstructed from the checkpoint file descriptor for crash recovery purposes. The proposed techniques are designed as an experimental proof of concept for increasing the write performance of BLCR. Each technique aims to avoid extreme modification to original BLCR design.

3.1 I/O Optimization

A factor limiting the checkpoint creation performance is the implementation of the I/O operations accessing disk storage. Our implementation can be divided into two main categories. The first is a method of transferring checkpoint data to a temporary storage buffer. The second is modification to the write sequencing of BLCR. Described below is a description of the checkpoint caching technique followed by the modified write procedure.

3.1.1 Checkpoint Caching

One method of avoiding BLCR inefficient write operation is to employ a buffer to store checkpoint data until a sufficient quantity can be flushed to local disk in a large buffer block. We accomplish this by creating temporary cache, in active memory, to hold checkpoint data before it is stored on local disk. This is not without a trade-off between resources consumption and improved I/O operation efficiency. This system overhead, in our experiments, displayed minimal resource utilization while a 4-fold performance gain or more
can be achieved with this technique. The caching method is designed to copy checkpoint
data originally passed to the BLCR write procedure. This operation copies checkpoint data
into a temporary storage buffer in active memory. This buffer block will be used for future
write operations to transmit large data chunks to the checkpoint file descriptor on local disk.
A pseudo code for this protocol is shown in Figure 1.

When this process begins, a data buffer and buffer size referencing checkpoint infor-
mation is provided. This information will either be used to establish a new \texttt{chkptBuf} buffer
or will be concatenated at the end of the existing storage buffer. The first function is to
determine if the \texttt{chkptBuf} volume, of any size, exists. If this storage buffer does not ex-
ist, a storage volume is created that is equal to the incoming checkpoint buffer size. Next,
the checkpoint data will be copied to this temporary storage volume to be used in future memory flush operations.

If the \textit{chkptBuf} containing checkpoint data already exists, an additional temporary storage volume is created to assist in the snapshot storage resizing operation. The \textit{tempBuf} storage is equal in size to the \textit{chkptBuf} volume increased by the \textit{chkptData} size. Next, the snapshot storage volume is copied to this local temporary volume followed by the checkpoint data concatenated at the end of the \textit{chkptBuf}. The snapshot storage volume is now expanded to equal the \textit{tempBuf} volume size that accommodates the extra information. All data of the temporary storage is relocated, using the \textit{memmove} system call, to the buffer referenced by \textit{chkptBuf}. The temporary storage buffer used for the \textit{chkptBuf} memory re-sizing can now be released for other OS purposes.

The maximum size the temporary data buffer, in any experiments we tested, never exceeded 300 KB. Thus, the greatest memory overhead cost associated with this implementation is at most 300KB.

3.1.2 Optimized BLCR’s Write Operation

BLCR original write operation transmitted all checkpoint buffers to local disk immediately upon receipt of the data, irrelevant of the data size. If the data buffer is extremely small, only a few bytes for example, calling the write operation is inefficient. Provided in Figure 2, is an example diagram of BLCR and optimized BLCR solution that requires temporarily storing checkpoint data until a large data block is accumulated before invoking the write procedure. This technique results in fewer calls to store data but with much larger buffer
blocks. This does not reduce the total storage footprint but reduces the total overhead cost associated with the write operations.

In this example, the maximum size of the $chkptBuf$ is thirty two data units. When the first write operation occurs, eight units of data are added to temporary storage and not transmitted to local disk. The second disk operation copies sixteen units of data to the $chkptBuf$ since the additional data does not exceed the buffers maximum storage volume. When the $chkptBuf$ reaches the limit of its allotted storage space, the data will be flushed to local disk in a large block. This continues until the completion of the checkpoint operation.

3.1.3 Implementation

All fundamental checkpoint operations of BLCR are divided into smaller function calls. Each of these function invokes the write procedure multiple times. In turn, each write function call is designed to flush checkpoint data to local disk immediately. The internal procedure provided by the Linux kernel to store all data to storage media is the virtual file system write, $vfs_write()$, operation. This function is an abstract system call provided to
ease development and unify the programming interface. The underlying procedure manages each file system media unique disk structures. When vfs_write() is activated, the I/O system calls needed to read/write the appropriate media (ext3, NTFS, CD_ROM) is processed internally by the kernel.

The vfs_write() procedure is invoked hundreds to thousands of times in BLCR, depending on the checkpointed process. This operation is executed for data buffers ranging from tens of bits to hundreds of bytes. Each call incurs a system overhead delay, which affects overall disk performance. When the total overhead is compiled, this time delta can significantly increase the total checkpoint process creation time.

Our optimized design blocks the checkpoint process at the write operation. Once the process is trapped, the checkpoint data will be transmitted to disk or cached in temporary storage for future transmission. If the chkptBuf volume is small, the incoming checkpoint data will be added to this temporary buffer. If the incoming data is larger than available free space in the chkptBuf volume, first the temporary buffer is transmitted to disk followed by

```c
write(chkptData, count)
    if(chkptBuf has space for the incoming chkptData)
        cr_copy(ckptData, count);
    else
        vfs_write(ckptBuf);
        vfs_write(ckptData);
        kfree(ckptBuf);
    end if
```

**Figure 3: Pseudo code for the Modified BLCR's Write.**
the incoming checkpoint data. It is vital to maintain the write order of checkpoint creation so that the checkpoint restart procedure can successfully reconstruct the process data. A pseudo code for this protocol is shown in Figure 3.

3.2 Remote Checkpoint

A factor limiting the checkpoint availability is the predefined local disk creation of the recovery point. This design only creates a checkpoint file descriptor in the directory where the process is first initiated. One method of avoiding BLCR local disk checkpoint creation is to share media directories. This is accomplished using Network File System (NFS) to remotely share a directory of a server with a client. As a result, when the checkpoint file is created in the home directory of the client, for example, the data is transmitted over the network to the shared server physical storage using the NFS protocol [3, 19].

The proposed technique is designed to avoid such complex systems setups by modifying BLCR to support remote checkpoint creation. Our design can be separated into two main categories. The first is a server application to listen for incoming checkpoint data whereas the second is modification to the write function of BLCR. Described below is a description of the server application followed by the new remote checkpoint write operation.

3.2.1 Remote Checkpoint Server

The remote checkpoint server process is a single threaded daemon. This program is added to the startup application list and initiated at system startup. For this experiment, each
while(true)
create socket;
bind to address;
listen for incoming connections;
wait for client to connect;
create file descriptor;
while(data buffered received)
    write checkpoint data;
close file descriptor;
close socket;

Figure 4: Pseudo code for the Server Process.

application was compiled using the GCC compiler on a Linux platform but could easily be built for any Microsoft derivative. Pseudo code for the server process is shown in Figure 4.

When the process begins, it creates a user level socket. This socket waits for communication from the client, acknowledging the start of checkpoint creation. Before accepting data transmissions, the server process creates a file descriptor representing the checkpoint data. The application now checks the data socket for incoming checkpoint packets. When the data packet is fully buffered, the packets payload will be written to local disk. After successful storage of the checkpoint data, this process writes the next data buffered until the end of the checkpoint transmission is reached.

The final packet sent to this daemon is unique to each checkpoint procedure and defines a successful checkpoint transaction. When this packet is received, the program will finalize execution by closing the checkpoint file descriptor and returning to idle state.
3.2.2 Modified BLCR’s Write Operation

Implementing the remote checkpoint support requires modification to BLCR write operation as shown in Figure 5. All fundamental checkpoint operations are divided into smaller function calls, each of which invokes the write procedure multiple times. Each function call is designed to make a copy of key data about the process and write that data to a checkpoint file on a local disk. Our design traps each write operation and transmits the data to the remote checkpoint server (or create a checkpoint file on local disk when the remote checkpoint server is unavailable).

```plaintext
if (remote checkpoint)
    if (socket is NULL)
        create socket;
        establish connection, if handshake fails
            break and perform the original_checkpoint
    end if
    package checkpoint data;
    send data message;
end if

if (original_checkpoint)
    original BLCR write operation;
end if
```

**Figure 5: Pseudo code for the Modified BLCR’s Write.**
3.2.3 Implementation

During the checkpoint operation, relevant data about the process is buffered for delivery to local disk. All data saved to storage media executes the virtual file system write operation, \( \text{vfs\_write()} \), provided by the Linux kernel. Our remote checkpoint operation captures the data to this procedure and remotely transmits the checkpoint data. Figure 6 describes this blocking protocol with some of the key checkpoint functions.

The first call to \( \text{vfs\_write()} \) is invoked by the function \( \text{cr\_save\_header()} \). This operation copies relevant file system data, files, and signal handlers to local disk. The remote checkpoint feature will intercept this process at the write call and redirect the data buffer to our remote checkpoint operation. The second function to invoke the write operation is the \( \text{cr\_save\_linkage()} \) method. This procedure is designed to copy the process ID, file descriptor, and process signals to local disk. This buffered data is also blocked and forwarded to the remote checkpoint write procedure.

The main function in Figure 6 is the \( \text{cr\_do\_vmadump()} \) method. This procedure implements the core memory dump operations essential to checkpoint creation. This memory dump operation is dependent on the pre-existing kernel module called Virtual Memory Area Dumper (VMA_Dump). This module is designed to serialize the memory regions of a process. The procedures function is to copy the memory mapping between an applications address space and store it to local disk. This method may be called multiple times to capture all process relevant data while copying to the checkpoint file. The transaction concludes with \( \text{store\_page\_list()} \) and \( \text{cr\_save\_pathname()} \), which save the memory page chunks and active file pointer, respectively.
When the remote checkpoint write procedure is first initiated, the status of the remote checkpoint data socket must be determined. If this socket has not been initialized, a call to create the data socket is issued using the internet address and port number parameters specified by the user at process initialization. After the socket is created, a connection between the client and server is attempted. If the three-way TCP handshake process fails to establish a connection, the original checkpoint process begins to write to the local storage to avoid loss of checkpoint data.

Once communication between the client and server is established, checkpoint data will be encapsulated within TCP packets for transmission. If the message size does not satisfy the maximum transmission unit (MTU) of a TCP packet, the message will be held for future delivery. When a message is suspended due to its size, subsequent vfs_write() data buffers
are added to the message until MTU is reached. Only when the data packet is full will the packet be transmitted to the server. Holding the packets transmission incurs minimal resource consumption but can greatly decrease the total TCP overhead associated with data transfer as well as a reduction in acknowledge responses between the client and server. By reducing the total transmission size, better performance can be achieved.

The total number of packets needed to send the checkpoint data is unique to each checkpointed process. If MTU is not reached by the last checkpoint write function, the packet will be padded and transmitted. Once all checkpoint data has been successfully transmitted to the server, a reserved packet representing the successful end of the checkpoint procedure is sent. This packet defines the end of communication for the existing checkpoint task.

The checkpoint process concludes by terminating the shared connection and returning the programs lock. This allows the remaining BLCR operations to finish the cleanup processes and gracefully exit.
4 Experimental Setup

Our experimental I/O optimization results were gathered using two implementations of checkpoint creation: BLCR and the optimized BLCR (O-BLCR) technique. The type of machine used to gather statistics was a Dell OptiPlex SX260 workstation equipped with a 3.06 GHz Intel Pentium 4, 1 GB of memory, 5,400 rpm hard disk and installed with Linux version 2.6.

To achieve a baseline reference, all benchmark applications described in Chapter 5 were first executed on an isolated system. This workstation is installed with the BLCR kernel module and disconnected from the network interface. Detaching this system from the campus network helps to ensure that the CPU is minimally interrupted by unrelated operations.

The second setup consists of an identical machine, also disconnected from the internal network. This workstation is installed with the O-BLCR kernel module. This system was used to gather comparative statistics to determine the effectiveness of the proposed optimization technique.

Our experimental remote checkpoint results were gathered with three implementations of checkpoint creation: BLCR, BLCR with NFS (BLCR+NFS), and BLCR with our remote checkpoint technique (BLCR+R). Two types of machines were used to gather statistics: one is a Dell OptiPlex SX260 workstation equipped with a 3.06 GHz Intel Pentium 4 with 1 GB of memory, 5,400 rpm hard disk and running Linux 2.6; the other is a 2.80 GHz dual-processor Intel Pentium 4 Dell PowerEdge 700 server, 3GB of memory, 5,400 rpm hard disk and is running Linux 2.6.
To achieve a baseline reference for the remote checkpoint procedure, we used the same configuration setup as the baseline I/O optimization technique. Hence, this workstation is installed with the BLCR kernel module and disconnected from the network interface.

The second setup consists of a server and one workstation connected through the NFS protocol. The NFS infrastructure is connected through a local network, isolated from the campus network. This client is configured to access the shared home directory of the NFS server as if it were local media. This workstation is also installed with an original version of the BLCR kernel module. The result is a client workstation that shares files over a network in a similar fashion to how local storage is accessed. In this system, all computations are performed by the workstation until the checkpoint data is stored. Since the home directory of the client is mapped to the home directory of the NFS server, all data is transmitted through the local network. The checkpoint data is encapsulated within TCP packets and transmitted to the server, all of which is managed internally by the NFS protocol.

The final setup consists of two workstation computers connected to a local area network or connected directly to the campus network. One computer is used as the client and the other is used as the server. The client is installed with the remote checkpoint kernel module and the server is installed with the corresponding daemon.

It is important to recognize the difference between the two network configurations. The first is a local area network, the optimal solution, since the only network traffic is the checkpoint TCP packets. The second configuration consists of two workstation computers that are directly connected to the internal campus network. In this setup it is possible for the transmissions to be delayed by network hubs, switches, etc. It is also possible that
checkpoint data may incur additionally delays if competing for limited campus bandwidth.

The results, compiled in the previous setup, provide an approximate upper and lower bound between the two network configurations. It is not possible to define the finite upper bound since every network has unique delays, but represented is an acceptable approximation for comparison.
5 Experimental Results

To evaluate the performance gain of our optimized BLCR technique and remote checkpoint technique we used four benchmark programs: an NP-complete problem, data encryption, linear equation solver, and file compression. Table I outlines the consumption of system resources under each application benchmark. Each program displays a unique process duration, active memory usage, and disk consumption in order to accurately simulate a wide range of potential calculations.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CPU</th>
<th>Memory</th>
<th>I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSP</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>AES</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>GE</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>HC</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Table I: Overhead Trend of Benchmark Programs.

Each program’s duration ranges from minutes to tens of minutes but only the write procedure is timed. Only the write function is evaluated as to accurately compare how each technique outperforms the original implementation. Table II, III, IV and V lists the problem size, execution time, and performance gain for each benchmark program tested with the BLCR and O-BLCR technique. Table VI lists each implementation, problem size and write latency for the BLCR, BLCR+NFS, and BLCR+R.

Each application has a predefined execution delta before the call to initiate the checkpoint. Each process initiates the checkpoint procedure with a minimal time discrepancy of the predefined scheduled execution. The time variance and computational effects are negligible since the only delays are CPU task switching and/or memory page eviction that
**Table II: Traveling Sales Problem Results.**

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Checkpoint Technique</th>
<th>Write Latency (μs)</th>
<th>Optimization Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>BLCR</td>
<td>14265</td>
<td>19.02</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>750</td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>BLCR</td>
<td>14265</td>
<td>19.02</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>750</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>BLCR</td>
<td>14327</td>
<td>18.25</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>814</td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>BLCR</td>
<td>14327</td>
<td>16.01</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>895</td>
<td></td>
</tr>
</tbody>
</table>

**Table III: Advanced Encryption Standard Results.**

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Checkpoint Technique</th>
<th>Write Latency (μs)</th>
<th>Optimization Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>BLCR</td>
<td>12568</td>
<td>14.53</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>865</td>
<td></td>
</tr>
<tr>
<td>35,000</td>
<td>BLCR</td>
<td>18435</td>
<td>13.31</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>1385</td>
<td></td>
</tr>
<tr>
<td>65,535</td>
<td>BLCR</td>
<td>20254</td>
<td>8.44</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>2399</td>
<td></td>
</tr>
<tr>
<td>100,000</td>
<td>BLCR</td>
<td>19938</td>
<td>7.45</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>2678</td>
<td></td>
</tr>
<tr>
<td>150,000</td>
<td>BLCR</td>
<td>26772</td>
<td>7.14</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>3751</td>
<td></td>
</tr>
<tr>
<td>200,000</td>
<td>BLCR</td>
<td>21375</td>
<td>4.45</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>4857</td>
<td></td>
</tr>
<tr>
<td>250,000</td>
<td>BLCR</td>
<td>27296</td>
<td>4.76</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>5776</td>
<td></td>
</tr>
</tbody>
</table>
### Table IV: Gaussian Elimination Results.

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Checkpoint Technique</th>
<th>Write Latency (µs)</th>
<th>Optimization Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,500</td>
<td>BLCR</td>
<td>847,678</td>
<td>20.20</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>41,939</td>
<td></td>
</tr>
<tr>
<td>3,500</td>
<td>BLCR</td>
<td>1,293,557</td>
<td>24.39</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>53,033</td>
<td></td>
</tr>
<tr>
<td>4,500</td>
<td>BLCR</td>
<td>1,708,774</td>
<td>25.26</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>67,647</td>
<td></td>
</tr>
<tr>
<td>5,500</td>
<td>BLCR</td>
<td>1,978,991</td>
<td>25.32</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>78,171</td>
<td></td>
</tr>
<tr>
<td>6,500</td>
<td>BLCR</td>
<td>2,364,449</td>
<td>30.25</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>78,171</td>
<td></td>
</tr>
<tr>
<td>9,500</td>
<td>BLCR</td>
<td>3,386,847</td>
<td>39.01</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>86,809</td>
<td></td>
</tr>
<tr>
<td>12,500</td>
<td>BLCR</td>
<td>4,414,118</td>
<td>34.74</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>127,064</td>
<td></td>
</tr>
<tr>
<td>15,500</td>
<td>BLCR</td>
<td>5,523,018</td>
<td>26.22</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>210,067</td>
<td></td>
</tr>
<tr>
<td>18,500</td>
<td>BLCR</td>
<td>6,340,560</td>
<td>24.87</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>254,986</td>
<td></td>
</tr>
</tbody>
</table>

### Table V: Huffman Compression Results.

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Checkpoint Technique</th>
<th>Write Latency (µs)</th>
<th>Optimization Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>BLCR</td>
<td>86,180</td>
<td>6.13</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>14,049</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>BLCR</td>
<td>253,134</td>
<td>8.51</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>29,732</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>BLCR</td>
<td>836,537</td>
<td>16.42</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>50,947</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>BLCR</td>
<td>1,507,152</td>
<td>17.63</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>85,510</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>BLCR</td>
<td>2,256,725</td>
<td>16.41</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>137,528</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>BLCR</td>
<td>2,669,130</td>
<td>14.23</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>187,364</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>BLCR</td>
<td>3,213,296</td>
<td>16.29</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>197,252</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>BLCR</td>
<td>3,798,030</td>
<td>15.75</td>
</tr>
<tr>
<td></td>
<td>O-BLCR</td>
<td>241,070</td>
<td></td>
</tr>
</tbody>
</table>
Table VI: Remote Checkpoint Results.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Benchmark</th>
<th>Problem Size</th>
<th>Write Latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLCR</td>
<td>TSP</td>
<td>100 - 250 Nodes</td>
<td>14.3 - 14.9</td>
</tr>
<tr>
<td></td>
<td>AES</td>
<td>10,000 - 250,000 Characters</td>
<td>12.5 - 37.2</td>
</tr>
<tr>
<td></td>
<td>GE</td>
<td>3,500 x 3,500 - 18,500 x 18,500</td>
<td>1,200 - 6,300</td>
</tr>
<tr>
<td></td>
<td>HC</td>
<td>4.5 - 78 Mbytes</td>
<td>80 - 4,300</td>
</tr>
<tr>
<td>BLCR+R (optimal)</td>
<td>TSP</td>
<td>100 - 250 Nodes</td>
<td>1.5 - 1.7</td>
</tr>
<tr>
<td></td>
<td>AES</td>
<td>10,000 - 250,000 Characters</td>
<td>1.9 - 85.6</td>
</tr>
<tr>
<td></td>
<td>GE</td>
<td>3,500 x 3,500 - 18,500 x 18,500</td>
<td>90 - 300</td>
</tr>
<tr>
<td></td>
<td>HC</td>
<td>4.5 - 78 Mbytes</td>
<td>200 - 6,600</td>
</tr>
<tr>
<td>BLCR+R (load)</td>
<td>TSP</td>
<td>100 - 2500 Nodes</td>
<td>4.1 - 6.1</td>
</tr>
<tr>
<td></td>
<td>AES</td>
<td>10,000 - 250,000 Characters</td>
<td>9.7 - 110.1</td>
</tr>
<tr>
<td></td>
<td>GE</td>
<td>3,500 x 3,500 - 18,500 x 18,500</td>
<td>300 - 1,800</td>
</tr>
<tr>
<td></td>
<td>HC</td>
<td>4.5 - 78 Mbytes</td>
<td>200 - 8,300</td>
</tr>
<tr>
<td>BLCR+NFS</td>
<td>TSP</td>
<td>100 - 250 Nodes</td>
<td>900 - 950</td>
</tr>
<tr>
<td></td>
<td>AES</td>
<td>10,000 - 250,000 Characters</td>
<td>800 - 1,000</td>
</tr>
<tr>
<td></td>
<td>GE</td>
<td>3,500 x 3,500 - 18,500 x 18,500</td>
<td>5200 - 493,500</td>
</tr>
<tr>
<td></td>
<td>HC</td>
<td>4.5 - 78 Mbytes</td>
<td>1,500 - 9,300</td>
</tr>
</tbody>
</table>

has little to no effect on the hard disk operations imposed by BLCR.

Figure 7 presents the overhead cost of BLCR and O-BLCR. Figure 8 presents the overhead of BLCR, BLCR+NFS, BLCR+R. For each benchmark program, time is measured in microseconds and checkpoint file size is in kilobytes. Described below is an examination of results from each benchmark program.

5.1 NP-Complete Problem

This benchmark application solves the traveling sales problem (TSP). The experiment began with an analysis of 100 connected nodes and concluded with 250 networked nodes. It exhibits high CPU consumption with minimal memory and I/O overhead.

From the experiment (I/O optimization), all checkpoint file sizes ranged from 168KB
to 205KB. The results depict a linear gradient in all TSP tests. Checkpoint creation in O-BLCR averaged 0.8ms to complete the write procedure. The BLCR implementation averaged 14.6ms to create a checkpoint file. The O-BLCR procedure provides a significant improvement over BLCR in this experiment.

From the experiment (remote checkpoint), all checkpoint file sizes ranged from 171KB to 211KB. The results depict a linear gradient in all TSP tests. Checkpoint creation in BLCR+NFS averaged 0.9s to complete the write procedure. The BLCR configuration averaged 14.6ms to create a checkpoint file. The optimal BLCR+R averaged 1.6ms and the campus networked BLCR+R setup averaged 5.1ms. In all experimental tests, the BLCR+NFS checkpoint creation remained the least efficient compared with other results. The BLCR+R procedure with or without network traffic provides a significant improvement over BLCR.

5.2 Data Encryption

This benchmark program implements an algorithm based on the Advanced Encryption Standard (AES). The experiment began with a plain-text of 10,000 characters and concluded with a 250,000 character plain-text. It displays high CPU consumption with moderate I/O usage and minimal memory overhead.

From the experiment (I/O optimization), all checkpoint file sizes ranged from 223KB to 2036KB. The results depict a linear gradient in BLCR and O-BLCR. Checkpoint creation in O-BLCR averaged 3.3s to complete the write procedure. The BLCR implementation averaged 23.8s to create a checkpoint file. O-BLCR showed a minimum of a 4.45-fold
Figure 7: Local Write Overhead for Checkpoint Creation.
benefit over BLCR.

From the experiment (remote checkpoint), all checkpoint file sizes ranged from 227KB to 2048KB. The results depict a linear gradient in BLCR and BLCR+NFS implementations but a logarithmic progression in each BLCR+R. Checkpoint creation in BLCR+NFS averaged 0.9s to complete the write procedure. The BLCR configuration averaged 23.8s to create a checkpoint file. The optimal BLCR+R averaged 36.9ms and the full networked BLCR+R setup averaged 53.8ms. In all experimental tests, the BLCR+NFS implementation remained the least efficient compared to other results. In these experiments where CPU and I/O operations approach the upper limit of the hardware, the remote checkpoint procedure has a greater overhead compared with BLCR.

5.3 Linear Equation Solver

This benchmark application solves a system of linear equations based on the Gaussian elimination method. The experiment began with a square matrix of 2,500 elements and concluded with a square matrix of 18,500 elements. It exhibits low CPU consumption with high memory and I/O overhead.

From the experiment (I/O optimization), all checkpoint file sizes ranged from 20MB to 75MB. The results depict a linear gradient in all Gaussian elimination tests. Checkpoint creation in O-BLCR averaged 0.1s to complete the write procedure. The BLCR implementation averaged 3.3s to create a checkpoint file.

From the experiment (remote checkpoint), all checkpoint file sizes ranged from 15MB to 76MB. The results depict a linear gradient in all Gaussian elimination tests. Checkpoint
Figure 8: Remote Write Overhead for Checkpoint Creation.
creation in BLCR+NFS averaged 225s to complete the write procedure. The BLCR configuration averaged 3.3s to create a checkpoint file. The optimal BLCR+R averaged 0.2s and the full networked BLCR+R setup averaged 1.0s. In all experimental tests, the BLCR+NFS checkpoint creation remained the least efficient compared to other results. The remote checkpoint procedure with or without network traffic provides significant improvement over BLCR.

5.4 File Compression

In the final experiment, this benchmark application is based on Huffman compression. The experiment began with an evaluation of the contents of a 12MB file and concluded with an evaluation of a 77MB data file. The CPU, memory, and I/O consumption ranged from medium to high depending on the input parameter.

From the experiment (I/O optimization), all checkpoint file sizes ranged from 5MB to 67MB. The results depict a linear gradient in all compression tests. Checkpoint creation in O-BLCR averaged 0.1s to complete the write procedure. The BLCR implementation averaged 1.9s to create a checkpoint file. In each experimental test, BLCR checkpoint creation remained the least efficient.

From the experiment (remote checkpoint), all checkpoint file sizes ranged from 5MB to 78MB. The results depict a linear gradient in all compression tests. Checkpoint creation in BLCR+NFS averaged 4.8s to complete the write procedure. The BLCR configuration averaged 1.9s to create a checkpoint file. The optimal BLCR+R averaged 2.9s and the full networked BLCR+R setup averaged 3.7s. In all experimental tests, the BLCR+NFS
checkpoint creation remained the least efficient compared to other results. In these programs where CPU and I/O operations approach the upper limit of the system resources, the remote checkpoint procedure has a greater overhead compared to BLCR.

5.5 Summary

From the I/O optimization experiment, we conclude that the O-BLCR technique successfully created checkpoint file descriptors with greater disk efficiency. In each experimental tests, BLCR without optimization displays a costly overhead, in order of magnitude, greater than O-BLCR. The write procedures overhead is directly linked to the number of write calls invoked by the kernel. By reducing the quantity of I/O request, not by reducing the total storage volume, the time to store data on local disk is greatly reduced. The trade between caching, in order to increase the size of each I/O request, and the resources overhead is shown to be minimal and beneficial.

In the remote checkpoint experiment, our results show that the NFS protocol used in checkpoint creation is consistently the least efficient technique. In each experimental tests, BLCR+NFS displayed a costly overhead, in order of magnitude, greater than BLCR and BLCR+R. By managing the packet protocols from within BLCR+R and not with a separate procedure, a greater response can be seen in packet transmission which leads to reduced checkpoint times.

In the TSP and Gaussian elimination experiments, the BLCR+R depicted a decrease in the time delta to complete the write operation. In the experiments AES and Huffman compression, the BLCR+R displayed a increase in the time delta of the write operation.
The key difference in the performance of TSP and Gaussian elimination versus AES and Huffman compression is due to the burden the process exerted on the systems resources. As CPU consumption and I/O operations approach the limits of the hardware platform, the remote checkpoint operation becomes less advantageous. Our proposed remote checkpoint technique is at best more efficient than BLCR and at worst more efficient than the BLCR+NFS for remote checkpoint creation.
6 Conclusion and Future Research

We have proposed an efficient optimization technique for checkpoint creation and a remote checkpoint protocol in this paper. The experimental outcomes demonstrate that (i) using a checkpoint caching technique within BLCR allows for more efficient checkpoint creation times within the write operation and (ii) integrating remote checkpointing support into BLCR allows for more efficient checkpoint creation when the CPU and I/O load is below the threshold of the hardware system. Our techniques only modified the BLCR write procedure to show the overhead delay, future versions of BLCR should be modified to internally handle the proposed caching technique to maximize checkpoint creation and the remote checkpoint procedure to increase system availability.

Future work in this area would focus on (i) authentication and (ii) encryption protocols as well as (iii) automated process load balancing. It would be possible to add an authentication scheme into the source code of BLCR. One such possibility would be the Diffie–Hellman key exchange procedure. For maintenance and flexibility it could be desirable to not modify BLCR but link to pre-defined system library’s such as OpenSSL [17]. Most Linux version come pre-installed with OpenSSL or is easily accessible through the systems repository. Using an additional software package creates dependencies for BLCR but allows for greater flexibility in cryptographic standards.

The second area to research further is how to secure checkpoint data transmitted over a network connection. To theorize about this procedure, a decision must first be made about the desired implementation of this cryptographic technique. Since checkpoint creation is a single threaded process and encryption can consume a significant amount of
system resources (ie. CPU cycles), encryption and related I/O operations should execute on a separate BLCR thread. This second encryption and I/O thread may reduce the check-pointed process suspension time, allowing the process to resume computations earlier. With this design approach, using a system library to perform the encryption could increase the threads overhead. I believe future work should implement a security protocol directly into BLCR so that all data buffered during vfs_write() would be immediately forwarded to a cryptographic engine for cipher text creation.

Additional research in this area must examine the possibility of internal load balance control for clusters implementing BLCR. Nuttall and Solomon [18] discusses conditions when task migration is advantageous. Harchol-Balter and Downey [12] and Kacer and Tvrdik [14] both describe how processes should be executed by remote machines (ie. process migration or remote execution). Wang et al. [23] presents an experimental load balancing system for cluster computing that improves resources utilization.

BLCR primary focus is on checkpoint creation. Checkpointing, such as BLCR, is a key feature in load balancing for cluster computing. It may be possible to expand the flexibility of BLCR using the foundational research of MOSIX [1, 2]. This management system targets high-performance computing on Linux clusters, multi-clusters, GPU clusters and Clouds. MOSIX currently implements automatic resource discovery and dynamic workload distribution. With the integration of BLCR, it would be possible to remove a job on a system that has become a bottleneck and relocate the task, at the processes current state, on a system with available resources. A collaboration between MOSIX migration techniques and BLCR checkpoint operations could result in a powerful new approach for the two tech-
nologies. MOSIX resources sharing could provided for greater efficiency for checkpoint creation as well as an automated load balancing protocol built on the MOSIX foundation.
References


