Efficient Storage Middleware Design in InfiniBand Clusters for High End Computing

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Xiangyong Ouyang, M. Sc.

Graduate Program in Computer Science and Engineering

The Ohio State University

2012

Dissertation Committee:

Prof. D. K. Panda, Advisor
Prof. P. Sadayappan
Prof. R. Teodorescu
Abstract

High End Computing (HEC) has been growing dramatically over the past decade. Both High Performance Computing (HPC) and Cloud Computing clusters are becoming increasingly homogeneous in their hardware platforms. Meanwhile both domains are facing a common challenge caused by the IO subsystem. In HPC domain, the most problematic scenarios include Checkpoint/Restart (C/R) and Process Migration, which involves dumping of huge quantities of data to stable storage. In Cloud Computing the IO bottleneck is usually felt by the data caching agency and database storage stratum.

This dissertation aims at designing an efficient IO middleware that can largely mitigate the aforementioned IO bottleneck. We design a write-aggregation scheme to reduce IO overhead during C/R activities. Based on that a hierarchical data staging framework is designed to substantially reduce the time cost of C/R. To tackle the performance issue in Process Migration, we propose a new protocol, Pipelined Process Migration with RDMA (PPMR), that fully pipelines data writing, data transfer, and data read operations during different phases of a migration cycle. Additionally we explore how to adopt Solid State Disk (SSD) into the IO stack to leverage SSD’s superior performance. We extend the SSD Flash Translation Layer to provide a new IO primitive called Atomic Write, which significantly improves the database performance by removing the overhead of the atomic completion guarantee for a group of discrete IO requests. We also propose an SSD-Assisted Hybrid Memory that expands RAM with SSD to make available a large amount of memory that can be used as an efficient data caching layer for datacenters.
This dissertation concludes that the IO bottleneck in HEC can be mitigated by optimizing the IO software stack, together with leveraging new storage technologies. Part of the software developed in the dissertation has been released with MVAPICH2, which is a popular open-source implementation of MPI over high performance networks.
Dedicated to my parents, Fanren and Chunying.
Acknowledgments

It’s an invaluable experience to have been able to spent my PhD years with so many great people in Nowlab. While the few words below cannot fully express my gratitude, I write these paragraphs to thank them for their guidance, support and friendliness.

I would like to thank my adviser, Prof. D. K. Panda for his support and guidance throughout the duration of my PhD study. I’m greatly indebted to him for the time and efforts he spent to lead me into the computer science research community. I have been learning a lot from his hard work and dedication as an academic researcher.

I would like to thank my committee members Prof. Sadayappan and Prof. Radu for their valuable guidance and suggestions.

I’m lucky enough to have spent my PhD years working closely with so many talented students at Nowlab. I would like to thank the senior colleagues in Nowlab including Dr. Ranjit Noronha, Dr. Lei Chai, Dr. Matthew Koop, Dr. Wei Huang, Dr. Sayantan Sur, Dr. Hao Wang, Dr. Jerome Vienne and Dr. Xavier Besseron. I’m also grateful to have worked with many colleagues in Nowlab including Miao, Hari, Krishna, Sreeram, Raghu, Jithin, Siddhesh, Ashish, Jian, Nusrat, Devendar, Joanthan, Mark. I thank them all for their innumerable discussions on technical and non-technical issues, as well as their friendship.

I would like to thank my family including my parents, my sisters and my brother. They always encourage me to march forward when I’m going through challenges and tough times.
Vita

October 26, 1978 . . . . . . . . . . . . . . . . . . . . . . . . . . . . Born - Yongfeng, China

September 1997 - June 2001 . . . . . . . . . . . . . . . Bachelor in Electronic Engineering,
    Tsinghua University, China

September 2001 - June 2004 . . . . . . . . . . . . . . . Master in Electronic Engineering,
    Tsinghua University, China

September 2006 - June 2007 . . . . . . . . . . . . . . . Graduate Teaching Associate,
    The Ohio State University.

June 2009 - September 2009 . . . . . . . . . . . . . . . Research Intern,
    Fusion-io, Utah.

June 2010 - September 2010 . . . . . . . . . . . . . . . Research Intern,
    Fusion-io, Utah.

July 2007 - Present . . . . . . . . . . . . . . . . . . . . . Graduate Research Associate,
    The Ohio State University.

Publications

and D. K. Panda, “High-Performance Design of HBase with RDMA over InfiniBand”, Int’l Parallel
and Distributed Processing Symposium (IPDPS ’12), May 2012

Lightweight User-Level Filesystem for Generic Checkpoint/Restart”, Int’l Conference on Parallel
Processing (ICPP ’11), Sep. 2011.

R. Rajachandrsekar, X. Ouyang, X. Besseron, V. Meshram and D. K. Panda, “Can Check-
point/Restart Mechanisms Benefit from Hierarchical Data Staging?”, Resiliency in High Perform-
ance Computing in Clusters, Clouds and Grids (Resilience 2011), in conjunction with Euro-Par,


vii


**Fields of Study**

Major Field: Computer Science and Engineering

Studies in:

- Computer Architecture  Prof. D. K. Panda
- Computer Networks    Prof. D. Xuan
- Software Systems     Prof. G. Agrawal
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiv</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Background</td>
<td>6</td>
</tr>
<tr>
<td>2.1 InfiniBand Architecture</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Checkpoint/Restart (C/R)</td>
<td>9</td>
</tr>
<tr>
<td>2.2.1 Types of Checkpoint</td>
<td>9</td>
</tr>
<tr>
<td>2.2.2 Berkeley Lab Checkpoint/Restart (BLCR)</td>
<td>10</td>
</tr>
<tr>
<td>2.2.3 MPI Checkpoint/Restart</td>
<td>10</td>
</tr>
<tr>
<td>2.2.4 Checkpoint/Restart in MVAPICH2</td>
<td>11</td>
</tr>
<tr>
<td>2.3 Process Migration</td>
<td>11</td>
</tr>
<tr>
<td>2.4 Filesystem in Userspace (FUSE)</td>
<td>12</td>
</tr>
<tr>
<td>2.5 NAND-Flash Based Solid State Drive (SSD)</td>
<td>13</td>
</tr>
<tr>
<td>2.5.1 NAND Flash Memory</td>
<td>14</td>
</tr>
<tr>
<td>2.5.2 Flash Translation Layer</td>
<td>14</td>
</tr>
<tr>
<td>2.6 Memcached: an Object Caching System</td>
<td>15</td>
</tr>
</tbody>
</table>
3. Problem Statement ................................................. 16

4. Accelerate Checkpointing with Write-Aggregation ........................... 22
   4.1 Characterizing Checkpoint Writing .................................... 22
   4.2 Reducing Checkpoint Writing Overhead by Write-Aggregation with Buffer Interleaving (WAI) ........................................... 27
      4.2.1 Design Strategy .................................................. 28
      4.2.2 Overlapping Between Application Processes and IO Threads ........... 29
      4.2.3 Design Choices .................................................. 30
      4.2.4 Reconstruct Checkpoint Files .................................... 30
   4.3 Performance Evaluation of WAI ....................................... 31
      4.3.1 Experimental Setup .............................................. 31
      4.3.2 WAI: Checkpoint Writing ....................................... 31
      4.3.3 WAI: Overlapping .............................................. 33
      4.3.4 WAI: Restart .................................................. 34
   4.4 CRFS: A Lightweight User-Level Filesystem for Generic Checkpoint/Restart ........................................... 35
      4.4.1 CRFS Design and Implementation ................................ 35
      4.4.2 File Open .................................................. 36
      4.4.3 File Write .................................................. 37
      4.4.4 File Close .................................................. 38
      4.4.5 Other Filesystem Operations ................................... 38
   4.5 Performance Evaluation of CRFS ....................................... 39
      4.5.1 Experimental Setup .............................................. 39
      4.5.2 CRFS: Raw Performance ....................................... 39
      4.5.3 CRFS: Checkpointing Performance ............................... 41
      4.5.4 CRFS: Reasons of Improvements ................................ 44
      4.5.5 Restart .................................................. 45
   4.6 Related Work .................................................. 46

5. A Hierarchical Data Staging Framework to Accelerate Checkpoint/Restart ...... 48
   5.1 Problems with Sequential Checkpoint Writing ............................. 48
   5.2 Hierarchical Data Staging Framework ................................... 50
      5.2.1 Write Aggregation on Client Side ................................ 51
      5.2.2 Staging Area on Server Side ................................... 52
   5.3 Performance Evaluation ............................................. 53
      5.3.1 Aggregated Write Bandwidth ................................... 54
      5.3.2 Application Checkpoint Time ................................... 55
   5.4 Related Work .................................................. 57
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.3.1</td>
<td>Design Challenges</td>
<td>105</td>
</tr>
<tr>
<td>8.3.2</td>
<td>SSD-Assisted Hybrid Memory</td>
<td>105</td>
</tr>
<tr>
<td>8.3.3</td>
<td>Integrate Hybrid Memory into Memcached</td>
<td>109</td>
</tr>
<tr>
<td>8.4</td>
<td>Analytical Model of Memcached</td>
<td>109</td>
</tr>
<tr>
<td>8.5</td>
<td>Performance Evaluation</td>
<td>111</td>
</tr>
<tr>
<td>8.5.1</td>
<td>Experimental Setup</td>
<td>112</td>
</tr>
<tr>
<td>8.5.2</td>
<td>Hybrid Memory: Basic Performance</td>
<td>112</td>
</tr>
<tr>
<td>8.5.2.1</td>
<td>Random Read Latency</td>
<td>113</td>
</tr>
<tr>
<td>8.5.2.2</td>
<td>Random Write Latency</td>
<td>113</td>
</tr>
<tr>
<td>8.5.2.3</td>
<td>Create New Object Latency</td>
<td>114</td>
</tr>
<tr>
<td>8.5.2.4</td>
<td>Random Read Latency at Different in-RAM Hit Ratio</td>
<td>114</td>
</tr>
<tr>
<td>8.5.3</td>
<td>Hybrid Memory: Operation Throughput</td>
<td>115</td>
</tr>
<tr>
<td>8.5.4</td>
<td>Hybrid Memory: Read/Write Volume to SSD</td>
<td>116</td>
</tr>
<tr>
<td>8.5.5</td>
<td>Hybrid Memory: Garbage Collector with FSC</td>
<td>118</td>
</tr>
<tr>
<td>8.5.6</td>
<td>Memcached Performance</td>
<td>118</td>
</tr>
<tr>
<td>8.5.6.1</td>
<td>Memcached Latency</td>
<td>120</td>
</tr>
<tr>
<td>8.5.6.2</td>
<td>Memcached Throughput</td>
<td>120</td>
</tr>
<tr>
<td>8.5.7</td>
<td>Performance in Datacenter Environment</td>
<td>121</td>
</tr>
<tr>
<td>8.5.8</td>
<td>Validation of the Analytical Model and Projections</td>
<td>122</td>
</tr>
<tr>
<td>8.6</td>
<td>Related Work</td>
<td>123</td>
</tr>
<tr>
<td>9.</td>
<td>Software Distribution</td>
<td>125</td>
</tr>
<tr>
<td>10.</td>
<td>Conclusions and Future Research Directions</td>
<td>127</td>
</tr>
<tr>
<td>10.1</td>
<td>Summary of Research Contributions</td>
<td>127</td>
</tr>
<tr>
<td>10.1.1</td>
<td>Write-Aggregation to Reduce I/O Overhead in Checkpoint</td>
<td>128</td>
</tr>
<tr>
<td>10.1.2</td>
<td>A Hierarchical Data Staging Framework</td>
<td>128</td>
</tr>
<tr>
<td>10.1.3</td>
<td>High Performance Pipelined Process Migration with RDMA</td>
<td>129</td>
</tr>
<tr>
<td>10.1.4</td>
<td>Improving Non-Contiguous Access with Atomic-Write</td>
<td>129</td>
</tr>
<tr>
<td>10.1.5</td>
<td>SSD-Assisted Hybrid Memory</td>
<td>130</td>
</tr>
<tr>
<td>10.2</td>
<td>Future Research Directions</td>
<td>130</td>
</tr>
<tr>
<td>10.2.1</td>
<td>Improvements on Checkpoint/Restart</td>
<td>131</td>
</tr>
<tr>
<td>10.2.2</td>
<td>Improvements in PPMR protocol</td>
<td>131</td>
</tr>
<tr>
<td>10.2.3</td>
<td>Improvements on SSD-Assisted Hybrid Memory</td>
<td>131</td>
</tr>
</tbody>
</table>

Bibliography | 133  |
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Basic Checkpoint Information (Class C, 64 processes on 8 Compute Nodes)</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>Checkpoint File Write Profile of LU.C.64</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>Overhead at Restart (seconds)</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>Checkpoint Sizes of Different Applications with Varied MPI Stacks. MVAPICH2 and OpenMPI use InfiniBand Transport. MPICH2 uses TCP Transport.</td>
<td>41</td>
</tr>
<tr>
<td>5.1</td>
<td>Device Raw Write Bandwidth (MB/second)</td>
<td>54</td>
</tr>
<tr>
<td>6.1</td>
<td>Migration Time Cost with Different Memory Footprint (BT.C/D, 64 processes on 8 compute nodes)</td>
<td>74</td>
</tr>
<tr>
<td>7.1</td>
<td>Experimental Machine Configuration</td>
<td>89</td>
</tr>
<tr>
<td>7.2</td>
<td>Write Latency in Microseconds</td>
<td>90</td>
</tr>
<tr>
<td>7.3</td>
<td>Write Bandwidth in MB/s</td>
<td>92</td>
</tr>
<tr>
<td>8.1</td>
<td>Latency of Memcached and MySQL with Different Network (1KB Record Size). SSD Latency is Measured with Fusion-Io ioDrive [7] 80GB at 1 KB Access Size.</td>
<td>103</td>
</tr>
<tr>
<td>8.2</td>
<td>Resemble a Datacenter Environment: Key-Value Query Latency (µs) with Zipf Distribution of 80/20.</td>
<td>122</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>InfiniBand Architecture (Courtesy IBTA)</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>IBA Communication Stack (Courtesy IBTA)</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>FUSE Architecture (Courtesy of [2])</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Conceptual Framework of the Proposed Work</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>Improve Checkpointing Performance in proposed research framework</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>Basic Profiling of Checkpointing for NAS Parallel Benchmark</td>
<td>24</td>
</tr>
<tr>
<td>4.3</td>
<td>Cumulative Write Time for Each Process (LU.C.64, ext3)</td>
<td>26</td>
</tr>
<tr>
<td>4.4</td>
<td>Write Aggregation with Interleaving (WAI)</td>
<td>28</td>
</tr>
<tr>
<td>4.5</td>
<td>How the Application Process (AP) and IO Threads Overlap</td>
<td>29</td>
</tr>
<tr>
<td>4.6</td>
<td>Evaluation of WAI</td>
<td>32</td>
</tr>
<tr>
<td>4.7</td>
<td>IO Threads Overlapping with Application</td>
<td>33</td>
</tr>
<tr>
<td>4.8</td>
<td>CRFS Design Diagram</td>
<td>36</td>
</tr>
<tr>
<td>4.9</td>
<td>CRFS Raw Write Bandwidth (8 processes on a single node)</td>
<td>40</td>
</tr>
<tr>
<td>4.10</td>
<td>Checkpoint Writing Time with MVAPICH2 (Lower is Better)</td>
<td>42</td>
</tr>
<tr>
<td>4.11</td>
<td>Checkpoint Writing Time with MPICH2 (Lower is Better)</td>
<td>43</td>
</tr>
</tbody>
</table>

xiv
4.12 Checkpoint Writing Time with OpenMPI (Lower is Better). We could not get the result of native Lustre for LU.C.128, despite many attempts.

4.13 Block IO Layer Trace on One Node. Checkpoint Writing of LU.C.64 on 8 Nodes, ext3

4.14 Cumulative Write Time for Each Process (LU.C.64, ext3 vs. ext3+CRFS)

5.1 Hierarchical Data Staging in Proposed Research Framework

5.2 Comparison between the direct checkpoint and the checkpoint staging approaches

5.3 Checkpoint Write Staging

5.4 Aggregated Write Bandwidth with 4 Data Servers

5.5 Checkpoint Write Time: LU.C.64

5.6 Checkpoint Write Time: BT.C.64

5.7 Checkpoint Write Time: LU.C.64

5.8 Checkpoint Write Time: BT.C.64

6.1 New Migration Approach in the Proposed Research Framework

6.2 Three Non-optimal Process Migration Schemes

6.3 Step Overlaps in the Different Migration Approaches

6.4 Decomposed Time Cost to Complete One Migration (64 Processes on 8 Compute Nodes, 8 Processes are Migrated)

6.5 Overlap in the Pipelined Process Migration Cycle

6.6 Pipelined Process Migration with RDMA

6.7 InfiniBand RDMA Read Bandwidth

6.8 Aggregation Bandwidth (Higher is Better)

6.9 Network Transfer Bandwidth and Pipeline Bandwidth (Higher is Better)

6.10 Time to Complete One Migration (Lower is Better)
8.9 Actual Read/Write Volume to SSD, Normalized to VMS. 50% Read, 50% Write. Have Preloaded 30 GB Data, Object Size=1KB.

8.10 Hybrid Memory Read/Write Throughput with GC in Background. 1 Reader, 1 Writer.

8.11 Memcached Performance with Hybrid Memory. Have Preloaded 30 GB data, Hybrid Memory Buffer Size=256 MB.

8.12 Predicted Memcached Query Latency by using the Analytical Model from Section 8.4. Latency Normalized to InfiniBand-Verbs. Hit Ratio=0.95.
Chapter 1: INTRODUCTION

High End Computing (HEC) has been growing dramatically over the past decade. It is fueled by the rapid advances in newer technologies including multicore architecture and high performance networks. Multi- and many-core processors are becoming increasingly widespread in commodity clusters systems, primarily due to their high performance-to-cost ratio. Meanwhile System Area Network (SAN) such as InfiniBand [10] and 10 GigE [1] are also drastically gaining momentum for designing high end clusters. With the wide availability of high performance components at a low cost, we are witnessing a trend that both HPC and Cloud Computing clusters are being built out of commodity multicore architecture with high performance commodity interconnections in mass product.

As the hardware platforms are becoming more and more homogeneous, both communities are striving to extract higher performance from a large number of computing cores with advanced interconnections, while also struggling to obtain objectives such as low-power consumption and better manageability. One common challenge faced by both domains is the IO subsystem that is painfully lagging behind. Compared to the rapid improvement in other parts of a computer system, no significant innovations have been introduced in the storage subsystem for decades [118], and it has become the weakest component in the whole stack. This bottleneck becomes most prominent for IO intensive scenarios in large scale clusters. Some of the most commonly encountered scenarios in HPC domain include Checkpoint/Restart (C/R) [96] and Process Migration [119],
which involves dumping of huge quantities of data to stable storage. In Cloud Computing the IO bottleneck is usually felt by the data caching agency [15] and database storage stratum.

Checkpoint-Restart (C/R) [51, 92] is the most widely deployed fault-tolerance mechanism for large scale applications. A typical C/R mechanism saves a snapshot of the current state for all processes in an application to a global shared file system, which is later used to recover the application from a failure by rolling back the entire application to the previous checkpoint. Upon a failure, the entire application has to be aborted and resubmitted. Although effective to achieve fault-tolerance, C/R mechanisms are notorious for their bulky persistent storage space demands, intense I/O operations and heavy overheads they impose on application performance [37, 93].

Job/Process migration [3, 44, 114, 119], a pro-active fault-tolerance mechanism, has been proposed as a complement to C/R. Upon the detection of an imminent failure event, the processes running on a deteriorating node are checkpointed and the checkpoint data is transferred to a healthy spare node where the processes are restarted. Job Migration can work in synergy with C/R by significantly reducing the frequency of full checkpoint [114]. However, process migration still suffers from high cost arising from three factors involving vast amount of file IO: (1) overhead at the source node to write process images into files; (2) overhead to copy the image files from the source to the target node; and (3) overhead at the target node to read process images files to restart.

Another challenge rises from the access pattern of many scientific applications that conduct large number of small non-contiguous I/O. It’s very difficult for a conventional storage system to deliver high performance while also guarantee the consistency and atomicity for such a non-contiguous access. This difficulty is interwoven with the data being partitioned into overlapping sub-domains. Under concurrent accesses these sub-domains may interleave in an inconsistent manner, leading to data corruption. Traditionally a coordinated locking mechanism is needed to
ensure the data consistency in such concurrent non-contiguous I/O, which further adds to the total overhead in parallel I/O.

In recent years, Flash memory based storage technology starts to mature [48, 55, 80] Flash memory has been poised as a revolution in the storage systems [43]. It owns many technical advantages such as fast random access, low cost, low power consumption, non-volatility and compact form factors [71]. Today NAND-flash based Solid State Disk (SSD) is being widely deployed in both personal computing and HEC domains [75, 76]. In addition to being regarded as an alternative storage device [76], There have been proposals [6] to expand effective memory size by mapping SSD into the virtual memory. These approaches effectively treat SSD as a swap device, and allows applications to transparently access a huge virtual memory space that is managed by the operating system. However, this type of approach suffers from heavy overhead at virtual memory management [72, 84], which manipulates SSD at memory-page granularity.

Given these new technology trends including multicore architecture, advanced interconnections, and SSDs, which are becoming more and more pervasive in both HPC and Cloud domains, new strategies are needed to leverage these features to address the broader IO challenge caused by the inefficient storage system that is experienced across both areas. In this dissertation we aim to leverage the advanced features of aforementioned emerging technologies to design an efficient IO middleware that can largely mitigate the IO bottleneck that is plaguing both HPC and Cloud Computing. We begin our research in HPC area to tackle the IO overhead issue with Checkpoint/Restart and Process Migration, then we extend our work to Cloud Computing area to address the performance problem related to inefficient IO. More specifically we will address the following questions:
• What are the underlying causes of IO bottleneck seen in a checkpoint operation? Based on the understanding of such causes, can we design a strategy to accelerate checkpoint writing on multicore platforms?

• Can we design a new I/O framework to improve parallel checkpoint performance? In such a new framework, can we integrate new technologies to boost the performance, such as advanced interconnections such as InfiniBand and new storage technology such as SSD?

• What are the main factors that dominate the heavy overhead in process migration? Can we design an efficient mechanism to handle Process Migration? Can we optimize the different steps involved in the data flow to achieve optimal performance?

• How to exploit SSD potentials to solve the performance issue related to non-contiguous I/O? Can this solution be applied to database storage engine to obtain a speedup in database performance?

• In addition to be used as hard disk replacement, can SSD be leveraged to improve OS virtual memory system? Specifically how to utilize SSD as an augment to RAM so as to increase the available memory size to applications?

The rest of this dissertation is organized as follows. Section 2 introduces a brief background of Checkpoint/Restart and Process Migration, which demand high IO capacity from the storage subsystem. It also covers some basics about Flash memory and SSD. In Chapter 3, we present the problem statement of this proposal. In Chapter 4, we propose our approach to reduce the I/O overhead related to checkpoint writing, and how to integrate this optimization into the virtual filesystem. In Section 5, a hierarchical staging framework is proposed to mitigate the checkpoint overhead experienced by applications. Chapter 6 introduces our new design to accelerate process
migration. Chapter 7 describes a new IO primitive designed for SSD to improve non-contiguous write performance. Chapter 8 shows our work that leverages SSD to boost a key-value object caching system in datacenter environment. Software distributions as part of this dissertation are described in Chapter 9. Chapter 10 provides the conclusion and possible future research directions.
Chapter 2: BACKGROUND

In this chapter, we discuss several important technologies that are related to our research, including InfiniBand architecture, two commonly used fault-tolerance mechanisms: Checkpoint/Restart and Process Migration, FUSE as a user-level filesystem library, and NAND-flash based Solid State Disks. This will provide appropriate background information for this dissertation.

2.1 InfiniBand Architecture

The InfiniBand Architecture [10] (IBA) defines a switched network fabric for interconnecting compute and I/O nodes. In an InfiniBand network, compute and I/O nodes are connected to the fabric using Channel Adapters (CAs). IBA describes the service interface between a host channel adapter and the operating system by a set of semantics called verbs. Verbs describe operations that take place between a CA and the host operating system for submitting work requests to the channel adapter and returning completion status. Figure 2.1 depicts the architecture of an InfiniBand network.

InfiniBand uses a queue based model. A consumer can queue up a set of instructions that the hardware executes. This facility is referred to as a Work Queue (WQ). Work queues are always created in pairs, called a Queue Pair (QP), one for send operations and one for receive operations. In general, the send work queue holds instructions that cause data to be transferred between the
consumer’s memory and another consumer’s memory, and the receive work queue holds instructions about where to place data that is received from another consumer. The completion of Work Queue Entries (WQEs) is reported through Completion Queues (CQ). Figure 2.2 shows a Queue Pair connecting two consumers and communication through the send and the receive queues.

InfiniBand supports two types of communication semantics: channel and memory semantics. In channel semantics, the sender and the receiver both must post work request entries (WQEs) to their respective QPs. After the sender places the send work request, the hardware transfers the data in the corresponding memory region to the receiver end. It is to be noted that the receive work request needs to be present before the sender initiates the data transfer. This restriction is prevalent in most high-performance networks like Myrinet [19], and others. The sender will not complete the work request until a receive request has been posted on the receiver. This allows for no buffering and zero-copy transfers.
When using channel semantics, the receive buffer size must be the same or greater than that of the sending side. Receive WQEs are consumed in the same order that they are posted. In the case of reliable transports, if a send operation is sent on a QP where the next receive WQE buffer size is smaller than needed the QPs on both ends of communication are transited into the error state.

In memory semantics, Remote Direct Memory Access (RDMA) operations are used instead of send/receive operations. These RDMA operations are one-sided and do not require software involvement at the target. The remote host does not have to issue any work request for the data transfer. Both RDMA Write (write to remote memory location) and RDMA Read (read from remote memory location) are supported in InfiniBand, although not all transports support it. RDMA operations allow a node to directly access a remote node’s memory contents without using the CPU at the remote side. These operations are transparent at the remote end since they do not involve the remote CPU in the communication.
2.2 Checkpoint/Restart (C/R)

As High End Computing (HEC) clusters are continuously growing in terms of scale and complexity, more frequent failures are expected during an application execution. Therefore fault-tolerance has become a necessity. Checkpoint/Restart is the most commonly used strategy to achieve fault tolerance. A checkpoint saves the state of the process at a given point of time during its execution. It includes enough information to restart a process from that point. An application maybe checkpointed periodically so that in an event of a failure, it can be restored from the most recent checkpoint to continue its execution, rather than start again from the beginning.

2.2.1 Types of Checkpoint

There are two potential approaches to initiating a checkpoint; application initiated and system initiated.

In application initiated checkpointing, the application decides when to start a checkpoint and requests the system to take the checkpoint on the application’s behalf. Transparent application-level checkpointing may be achieved through compiler techniques [109]. Additionally, a hybrid approach is possible where the application participates in the creation of a checkpoint but is assisted by a user-level library [95].

In system initiated checkpointing, the system directly initiates the application checkpoint, without interaction with the application. The application maybe unaware of the fact that it is being checkpointed. This is usually achieved through a kernel component, as with Berkeley Lab Checkpoint Restart (BLCR) [49]. BLCR has been combined with several Message Passing Libraries (MPI) such as LAM/MPI [113], OpenMPI [23] and MVAPICH2 [96] to checkpoint parallel jobs running on multiple nodes.
2.2.2 Berkeley Lab Checkpoint/Restart (BLCR)

Berkeley Lab Checkpoint/Restart (BLCR) [57] allows programs running on Linux systems to be checkpointed by writing the process image to a file and then later be restarted from the saved process image file. BLCR by itself can only checkpoint/restart processes on a single node. But it provides callbacks to be extended by applications, so that a parallel application can also be checkpointed.

2.2.3 MPI Checkpoint/Restart

MPI is the de facto standard for parallel programming. Many scientific applications written in MPI take days to complete their computation. Given that the MTBF of modern clusters is smaller than the average running time of the application, failures are expected during the lifetime of a large scale application. Many MPI libraries have built-in checkpointing capabilities that allow applications to be checkpointed at regular intervals. Checkpointing saves the complete state of the MPI process to disk so that in the event of a failure, the process can be restarted from the saved image. Berkeley Lab Checkpoint/Restart software package (BLCR) [49] is a popular Checkpoint/Restart solution that is used by many MPI implementations, including MVAPICH2 [17,96], OpenMPI [23] and LAM/MPI [99].

Although BLCR has the capability to save and restore the MPI process’s execution context, most modern interconnects store a substantial amount of information pertaining to the communication endpoint on the interconnect hardware itself. BLCR cannot access this information and so cannot save/restore the communication endpoint. Additionally, all the processes that are part of the MPI job must be in a consistent state before they are checkpointed. As a result, the process of checkpointing an MPI application usually involves the following phases.
Phase 1: Suspend communication between all processes in the parallel application and tear down the communication end points.

Phase 2: Use a checkpoint library to dump the individual process’s memory image to a checkpoint file.

Phase 3: Re-establish connections among the MPI processes and continue execution.

2.2.4 Checkpoint/Restart in MVAPICH2

MVAPICH2 is a MPI library with native support for InfiniBand and 10GigE/iWARP [17]. It supports application initiated and system initiated checkpointing [54, 96] using the BLCR Library for Checkpoint/Restart [57]. Checkpointing in MVAPICH2 involves the following three steps.

- Draining the communication channels of all pending messages and tearing down the communication endpoints on each process.

- Using the BLCR Library to independently request the checkpoint of every process that is part of the MPI job. The checkpoint is taken by BLCR in a blocking manner with the data being written to one file per process.

- Re-establishing the communication endpoints on every process.

The application continues its execution after the checkpoint is taken.

2.3 Process Migration

Job/process migration [3, 44, 114, 119], a pro-active fault-tolerance mechanism, has been proposed as a complement to C/R. During a migration, the processes running on a source node are checkpointed and the checkpoint data is transferred to a healthy spare node where the processes are restarted. All other processes of the application are paused when a migration is initiated and
resume execution when the migrated processes are restored. Migration overcomes the two key
drawbacks of C/R, namely the unnecessary dumping of all processes’ snapshots and the queuing
latency during restart, because of re-submitting the parallel job. Job Migration can work in synergy
with C/R by significantly reducing the frequency of full checkpoint [114], providing two prereq-
uisites: the capability to predict a subset of imminent failures with health monitoring mechanisms
such as IPMI [13] and varied failure prediction models [100, 107], and the availability of healthy
spare nodes.

Additionally, process migration is also a desirable feature to meet many other demanding re-
quirements such as cluster-wide load balancing, server consolidation, performance isolation and
ease of management. Hence any progress to improve process migration performance will likely be
perceived by a wide spectrum of demanding cluster applications.

Process Migration has been supported by several popular MPI stacks including MVAPICH2 [18]
and OpenMPI [23]. Experiments show that these implementations cannot achieve an optimal per-
formance. Generally the high cost arises from three factors involving vast amount of file IO: (1)
overhead at the source node to write process images into files; (2) overhead to copy the image files
from the source to the target node; and (3) overhead at the target node to read process images files
to restart.

2.4 Filesystem in Userspace (FUSE)

Filesystem in Userspace (FUSE) [4] is a software that allows to create a virtual filesystem in the
user level. As illustrated in Fig. 2.3, it relies on a kernel module to perform privileged operations
at the kernel level, and provides a userspace library that eases communication with this kernel
module. FUSE is widely used to create filesystems that do not really store the data itself but relies
on other resources to effectively store the data. Then, a FUSE virtual filesystem is like a way to present and organize data to users through the classic filesystem interface.

![FUSE Architecture Diagram](image)

Figure 2.3: FUSE Architecture (Courtesy of [2])

## 2.5 NAND-Flash Based Solid State Drive (SSD)

Conventional hard drives are constrained by the mechanical rotating disk, which results in poor random access performance and excessively high power consumption [48, 55, 80]. These limitations are related to its physical characteristics and hence very hard to solve within the same domain. In order to overcome these performance constraints many storage technologies have been studied. Nowadays NAND-flash memory is deemed the most popular storage alternatives to hard disks.
2.5.1 NAND Flash Memory

Flash memory is a kind of electronic-erasable non-volatile memory. It has traditionally been widely used in consumer electronics [66] due to their low power consumption and shock resistance. Because NAND flash allows a denser layout at a cheaper cost, it follows the Moore’s law aggressively. NAND-flash exhibits many technical merits such as fast random access, low power consumption and high density. However it has a distinguishing feature of asymmetric read/write performance [75, 76]. NAND-flash has a fast random read latency (tens of microseconds), but a flash page cannot be overwritten (in-place update) unless it is erased before the writing. Erase operation is very slow (several milliseconds) and can only be performed at a much bigger granularity called Erase Block (EB) that contains 64-256 flash pages. After a certain number of erase-program cycles (1K-100K), flash page wear out and cannot be reliably written to. The typical access latencies for read, write, and erase operations are 25 microseconds, 200 microseconds, and 1500 microseconds, respectively [76]. Flash memory has been poised as a revolution in the storage systems [43]. Due to its many advantages such as fast random access, low cost, low power consumption, non-volatility and compact form factors [66], it stands in the middle between RAM and mechanical hard disk in terms of both cost and performance [43], making it an ideal gap filler.

2.5.2 Flash Translation Layer

In order to cope with the inefficiency of flash memory, significant amount of studies have been carried out [66, 77, 78]. The outcome is a Flash Translation Layer (FTL), an intermediate software layer that maps data’s logical address to physical address in flash memory. The FTL receives logical read and write commands from the applications and converts them to the internal flash memory commands. To emulate disk like in-place update operation for a logical page $L_p$, the FTL
writes data into a new physical page $P_p$, maintains a mapping between logical pages and physical pages, and marks the previous physical location of $L_p$ as invalid for future garbage collection.

Although FTL allows current disk based application to use SSD without any modifications, it needs to internally deal with flash physical constraint of erasing a block before overwriting a page in that block. Besides the in-place update problem, FTL uses various wear leveling techniques to solve the wear out problem by evenly distributing the erasure operations of different blocks in the flash memory to increase its overall longevity [66, 78]. Recently NAND-flash based Solid State Disk (SSD) with their built-in FTL [7, 12] starts to receive a widespread deployment in both personal computer and high performance computing domains.

### 2.6 Memcached: an Object Caching System

Recently dynamic Web and social network applications are generating gigantic amount of dynamic data, which needs to be stored in a reliable storage media for future retrieval and analysis. It is hard to scale databases to accommodate such huge data volumes with a satisfactory performance. A new memory caching layer, memcached was proposed by Fitzpatrick [52] to cache database request results such that future requests could be serviced directly from the caching layer to avoid an expensive database query round trip.

Due to its generic nature and open-source distribution [15], it has been quickly adopted in a wide variety of environments. Using Memcached, spare memory in data-center servers can be aggregated to speed up lookups of frequently accessed information, be it database queries, results of API calls or web page rendering elements. A typical deployment of Memcached is shown in Figure 8.3. The performance of Memcached is directly related to that of the underlying networking technology. In addition to that, it is also tightly coupled with the aggregated amount of RAM which determines the hit ratio and how often the database shall be queried.
Chapter 3: PROBLEM STATEMENT

The main objective of our research is to reduce the I/O overhead widely seen in High End Computing (HEC). We propose to build a IO middleware layer in HEC as illustrated in Figure 3.1. In this conceptual framework, the shaded boxes indicate the components we have been working on. As can be seen, we work on multiple aspects of the I/O subsystems in HEC.

We plan to investigate causes of IO bottleneck with inefficient parallel data access, which is typically seen by parallel checkpoint/restart and process migration. With the acquired knowledge we will propose new strategies to optimize the parallel data access for more efficient data transfer so as to reduce the IO overhead experienced by applications. Additionally we will explore how data staging technique can benefit application checkpoint, and propose a hierarchical and modular data staging framework to reduce the burden of checkpointing on client nodes without penalizing them in terms of performance. We will also study how SSD can potentially impact the storage substratum design in Cloud Computing, including database server storage engine and data caching layer.

More specifically, we aim to address the following questions with this dissertation:

• **What are the data access patterns during a parallel application checkpoint writing on multicore architecture that lead to IO bottleneck in HPC environment?** Based on the parallel write patterns, can we design a strategy to accelerate checkpoint writing on multicore platforms?
With the rapid growth of multicore processors, a parallel job spawns more and more processes running in a node. Thus, more processes perform concurrent write from a same node to the storage system during a checkpoint. We want to understand the characteristics of those simultaneous write streams and gain more insights into the causes that leads to IO performance degradation.

Based on the knowledge acquired we plan to design a write-aggregation scheme that can mitigate the concurrent IO contentions during checkpoint writing by coalescing multiple data streams into a small shared buffer pool, and overlap the file writing with checkpoint data copying. Furthermore, we propose to decouple the application checkpointing progress from slow file IO. While application processes are copying checkpoint data to the shared buffer
pool, a set of IO threads simultaneously write the buffered data to a file system. The application is allowed to proceed once the process context is saved to the buffer pool, meanwhile the IO threads flush the buffered data to checkpoint files.

- **How to design a hierarchical data staging architecture that can relieve compute nodes from the relatively slow checkpoint writing? How to leverage high speed network and new storage media such as SSD to accelerate staging I/O performance?**

With the rapid advances in technology, many clusters are being built with high performance commercial components such as high-speed, low-latency networks and advanced storage devices such as Solid State Drives (SSDs). These advanced technologies provide an opportunity to redesign existing solutions to tackle the I/O challenges imposed by Checkpoint/Restart. In this paper, we propose a hierarchical data staging architecture to address the I/O bottleneck caused by Checkpoint/Restart. In this I/O staging framework, checkpoint data is streamed from compute node to a set of dedicated server nodes via high speed networks to achieve fast data movement. The server nodes have advanced SSDs installed to further increase the I/O bandwidth. The application on compute nodes are allowed to resume once the checkpoint data has been pumped to the server nodes through the high speed interconnect. Thus the application execution is decoupled from the slow data copy from the server node to a backend parallel filesystem at the background in an asynchronous manner.

- **Can we design an efficient mechanism to handle Process Migration? Can we optimize the different steps involved in the data flow to achieve optimal performance?**

Although Checkpoint/Restart(C/R) is widely adopted, it’s not capable enough to meet the demands of upcoming exascale systems, due to its heavy I/O overhead. Process migration has already been proposed. Migration overcomes the two key drawbacks of C/R, namely the
unnecessary dumping of all processes’ snapshots and the queuing latency during restart of the job. Job Migration can work in synergy with C/R by significantly reducing the frequency of full checkpoint [114]. Additionally, process migration is also a desirable feature to meet many other demanding requirements such as cluster-wide load balancing, server consolidation, performance isolation and ease of management.

We will conduct profiling on several process migration mechanisms already deployed, and reveal the principal factors that determines the overall cost of a migration. We want to propose a new approach, Pipelined Process Migration with RDMA (PPMR), to overcome these overheads. Our new protocol fully pipelines data writing, data transfer, and data read operations during different phases of a migration cycle.

- **How to exploit SSD characteristics to improve the non-contiguous data access efficiency? Can this potential be applied to database server so as to obtain a speedup in database performance?**

Many applications perform heavy parallel I/O during its execution, which imposes a colossal challenge to the underlying storage system. In addition to the sheer amount of I/O size, a lot of scientific applications conduct large number of small non-contiguous I/O accesses, which is especially unfavorable for a conventional storage system to deliver high performance. Another challenge in this context is to meet the demands of applications that partition data into overlapping subdomains. Since these subdomains overlap, under concurrent accesses they may interleave in a non-consistent manner, resulting in data corruption. Hence write atomicity becomes a requirement in parallel IO context.

As the flash-based storage technique matures, we find that it represents unique opportunities to optimize parallel I/O from the fundamentally bottom layer. Modern NAND-flash based
SSD exhibits high throughput and low random access latency, which matches with the extensive non-contiguous access pattern typically seen in many scientific applications. Based on these distinct characteristics we propose to design a new I/O primitive that batches multiple discrete I/O requests into a logical group which is completed in just one I/O call. We want to extend the SSD’s Flash Translation Layer to guarantee the atomic completion of such a group request. We call this new primitive Atomic-Write. We will demonstrate its effectiveness with MySQL database workload. This new primitive can also be applied to generic parallel I/O.

- In addition to be used as hard disk replacement, can SSD be leveraged to improve OS virtual memory system? Specifically can SSD be utilized as an augment to RAM so as to increase the available memory size to applications?

Flash memory has been poised as a revolution in the storage systems [43]. Recently NAND-flash based SSD has been widely deployed in both personal computing and high performance computing (HPC) domains. NAND-flash has many advantages such as fast random access, low cost, low power consumption, non-volatility and compact form factors [71]. It stands in the middle between RAM and mechanical hard disk in terms of both cost and performance [43], making it an ideal gap filler.

A straightforward strategy is to use SSD as a virtual memory swap device. This approach leads to severe under-utilization of SSD capabilities because, operating system operates virtual memory (VM) at page granularity, and every object read/write will cause an entire flash page to be read/written no matter how small the object is [65, 84]. This not only wastes the SSD IO bandwidth, but also results in unnecessary flash wearing and reduces the lifespan of SSD [118].
We propose to augment RAM with SSD to form an object-cache for key-value pairs. Unlike VM swap system that manipulates memory and SSD at page level, we manage memory allocation at object level. We will evaluate the efficiency of this new approach in contrast to other alternatives.
Chapter 4: ACCELERATE CHECKPOINTING WITH WRITE-AGGREGATION

In this chapter, we focus on how to reduce checkpoint writing overhead from an intra-node perspective. We propose a strategy, Write-Aggregation with Interleaving (WAI), to mitigate the checkpoint writing cost within a node. We also explore how to incorporate this new strategy into a virtual filesystem so as to carry on the benefit to a wide range of MPI stacks. More specifically, we work on the highlighted part in Figure 4.1 of our proposed research framework.

In Section 4.1 we analyze the parallel I/O patterns of typical MPI checkpointing to understand the root cause of poor checkpoint writing performance. In Section 4.2, we propose a strategy called Write-Aggregation with Interleaving (WAI), to reduce checkpoint writing overhead from within a node. In Section 4.4 we incorporate WAI into a virtual filesystem that helps a wide range of MPI stacks to reduce the IO bottleneck in Checkpoint/Restart. We carry out performance evaluation in Section 4.3 and 4.5. Related work is discussed in Section 4.6.

4.1 Characterizing Checkpoint Writing

Basic checkpoint is widely recognized as a very costly operation because of its heavy I/O load on the storage system and the severe overhead it causes on the application execution time. To better understand the characteristics of checkpointing, we profiled the checkpoint writing to both local ext3 and a parallel filesystem (PVFS2 [25]) for several applications from the NAS Parallel
Figure 4.1: Improve Checkpointing Performance in proposed research framework

Benchmark suite version 3.2.1 [116] using MVAPICH2 [18] with BLCR 0.8.2 in a multicore Linux cluster. Each compute node has 8 processor cores on 2 Intel Xeon 2.33 GHz Quad-core CPUs. We choose Class C with 64 processes. Each process runs on a separate processor core, so 8 nodes are used in the test, and each process saves its checkpoint image to a separate file. The BLCR is modified to provide profiling information pertaining to checkpoint file writing. Table 4.1 shows some basic information including checkpoint time cost, file size and number of file write accesses.

Table 4.1: Basic Checkpoint Information (Class C, 64 processes on 8 Compute Nodes)

<table>
<thead>
<tr>
<th></th>
<th>LU</th>
<th>BT</th>
<th>SP</th>
<th>CG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time for one checkpoint(seconds)</td>
<td>7.6</td>
<td>11.3</td>
<td>10.3</td>
<td>7.1</td>
</tr>
<tr>
<td>Checkpoint Data size(MB) per node</td>
<td>184.0</td>
<td>320.0</td>
<td>316.0</td>
<td>163.2</td>
</tr>
<tr>
<td>Number of VFS write per process</td>
<td>975</td>
<td>1057</td>
<td>1367</td>
<td>820</td>
</tr>
<tr>
<td>Total number of VFS writes per node</td>
<td>7800</td>
<td>8456</td>
<td>10936</td>
<td>6560</td>
</tr>
</tbody>
</table>

First, we measure the execution time of the application without checkpoints and with three checkpoints evenly distributed during the application execution. The results are displayed in Figure 23.
4.2(a). For example, BT.C.64 takes 169.9 seconds to complete without any checkpoint. When 3 checkpoints are taken and written to local ext3, it takes 205.19 seconds to complete, an overhead of 20.77%. If checkpoints are stored to PVFS2, then the application finishes in 274.9 seconds, which translates to an overhead of 61.85%. For larger scale application with thousands of processes, we expect the checkpoint overhead to be more adverse. Our observation is consistent with the results reported in [63].

![Figure 4.2: Basic Profiling of Checkpointing for NAS Parallel Benchmark](image)

Checkpointing in MVAPICH2 involves the following three steps. **Phase 1**: Draining the communication channels of all pending messages and tearing down the communication endpoints on each process. **Phase 2**: Using the BLCR Library to checkpoint each process of the MPI job. The checkpoint is taken by BLCR in a blocking manner with the process image data being written to one file per process. **Phase 3**: Re-establishing the communication endpoints on every process. We have measured the time spent in each phase of a checkpoint cycle, and the results are illustrated in Figure 4.2(b). It’s clear that Phase 2 is the dominant factor of the overall time cost in a checkpoint.
We have further profiled the checkpoint creation (*Phase 2*) for these applications to understand the checkpoint file write patterns to ext3 filesystem. Table 4.2 represents the checkpoint profiling for application LU.C.64 running on 8 compute nodes with 8 processes per node. It decomposes the checkpoint writing into different categories according to the size of writes. The first column is the size of write belonging to that category. The second column is percentage of writes within that range. The third column is percentage of data amount written by that type of write. The fourth column is percentage of time spent by VFS writes belonging to that category. From Table 4.2 it can be observed that around 8 seconds are taken for a checkpoint to complete. During a checkpoint, each process generates a 23 MB snapshot to be saved, and a total of 7800 `write()` system calls are performed by all the 8 processes on a same node.

![Table 4.2: Checkpoint File Write Profile of LU.C.64](image)

<table>
<thead>
<tr>
<th>Size</th>
<th>% of Writes</th>
<th>% of Data</th>
<th>% of Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-64</td>
<td>50.86</td>
<td>0.04</td>
<td>0.17</td>
</tr>
<tr>
<td>64-256</td>
<td>0.61</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>256-1K</td>
<td>0.25</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>1K-4K</td>
<td>9.46</td>
<td>1.53</td>
<td>0.01</td>
</tr>
<tr>
<td>4K-16K</td>
<td>36.49</td>
<td>11.36</td>
<td>44.66</td>
</tr>
<tr>
<td>16K-64K</td>
<td>0.74</td>
<td>0.77</td>
<td>6.55</td>
</tr>
<tr>
<td>64K-256K</td>
<td>0.49</td>
<td>3.79</td>
<td>11.80</td>
</tr>
<tr>
<td>256K-512K</td>
<td>0.25</td>
<td>3.58</td>
<td>1.75</td>
</tr>
<tr>
<td>512K-1M</td>
<td>0.61</td>
<td>17.72</td>
<td>14.72</td>
</tr>
<tr>
<td>&gt;1M</td>
<td>0.25</td>
<td>61.21</td>
<td>20.35</td>
</tr>
</tbody>
</table>

We observed that a large number of small writes (64 bytes to 1 KB) cost a tiny fraction of time because they are quickly absorbed by the VFS page cache. These small writes are primarily storing the process context including CPU registers, signal handler table, timers, open-file tables, process/group/session ids, and various BLCR metadata structures necessary to restore a process.
On the other hand, 37% of write accesses are in the medium range (4 K to 64 K) and are responsible for 50% of all checkpoint time, even though they only deliver 13% of the total data. There are a few (less than 1%) very large writes (≥256 KB) that contribute to the majority of the data (80%). However large sequential writes are relatively efficient and they only cost 35% of checkpoint time. During a checkpoint, BLCR also stores the virtual memory area (VMA) of a process. BLCR scans all VMAs of a process, and saves non-zero contiguous data pages to the checkpoint file. An application process usually has many VMAs. Many VMAs contain a handful of contiguous pages that need to be dumped to file, which become a medium write. A few VMAs contain large block of contiguous pages to dump. They are the source of large writes.

During checkpoint writing, multiple processes simultaneously issue their write requests and each medium request needs new pages to be allocated in page cache. These concurrent write streams cause severe contentions in the VFS layer, leading to degraded performance. This observation implies that optimizations are needed to reduce medium write requests.

![Figure 4.3: Cumulative Write Time for Each Process (LU.C.64, ext3)](image)

The contentions induced by the concurrent write requests not only degrade write throughput, but also create a large variation for individual processes to complete their writing. This is shown
in Figure 4.3. Each line represents the time spent by a process to perform write operations, shown in a cumulative manner with respect to the write size. We observe a large variation in the processes completion time, ranging from 4 seconds to 8 seconds. Some processes are able to finish their writing very quick, however they have to coordinate with slower counterparts to re-establish communication channels before resuming execution (as is discussed in Section 2.2.4). Consequently all processes are delayed by the progress of the slowest processes, resulting in a longer checkpoint time overhead.

4.2 Reducing Checkpoint Writing Overhead by Write-Aggregation with Buffer Interleaving (WAI)

In this section we introduce our improved design strategies to accelerate checkpointing parallel applications on multicore systems.

Through the profiling data collected in section 4.1, we find that medium VFS writes consume a significant portion of checkpoint write time. Therefore we propose to coalesce all VFS writes into a shared buffer pool, and overlap the file writing with checkpoint data copying. Furthermore, we propose to decouple the application checkpointing progress from slow file IO. While BLCR library is copying process image data to the shared buffer pool, a set of IO threads simultaneously write the buffered data to a file system. Once the process context is saved to the buffer pool, the application is allowed to proceed. Simultaneously the IO threads are flushing the buffered data to checkpoint files. This design strategy achieves better checkpoint writing performance at the cost of additional memory usage. On the other hand, a recent study [28] suggests that even large scale parallel jobs seldom use all available local memory. Therefore we feel it is reasonable to allocate part of the available local memory to the buffer pool to accommodate checkpoint data writing.
4.2.1 Design Strategy

Figure 4.4 illustrates our Write Aggregation with Interleaving (WAI) design. When a checkpoint is initiated, a user-level daemon (Buffer Manager) prepares a shared buffer pool. This buffer pool is mapped to kernel space, so that the BLCR module running in kernel space is able to access this buffer pool. During a checkpoint, each process issues a sequence of VFS writes via BLCR to store its image data. Instead of following the VFS write data path, a process first grabs a free chunk of buffer from the buffer pool. All subsequent VFS writes from the same process are then coalesced into this buffer chunk. When the chunk is filled up, the process returns it to the buffer pool, and requests for a new free chunk from the pool. This procedure is repeated by all processes till they finish writing all their checkpoint data.

A set of IO threads constantly monitor the state of the buffer pool. Once a chunk is filled up with data and returned to the buffer pool, an IO thread is activated to write this chunk to a separate file. The process rank information and the offset of the data is encoded within the file name. This
information is used to rebuild the original checkpoint file from many smaller chunk files at restart phase. When writing this data file is finished, the chunk is returned to buffer pool to be reused.

![Diagram of Application Process (AP) and IO Threads Overlap](image)

Figure 4.5: How the Application Process (AP) and IO Threads Overlap

### 4.2.2 Overlapping Between Application Processes and IO Threads

The benefit of WAI comes from two aspects. First, by coalescing multiple VFS writes of varied sizes into fewer writes of larger chunks, it’s able to improve file write throughput. Second, it’s able to hide the checkpoint write delay from the application. This is illustrated in Figure 4.5. At time $t_1$, a checkpoint is requested. All the application processes (APs) enter phase 1 to suspend communications. They then enter phase 2 to store their process images using WAI. In this phase an application process repeatedly grabs free chunks from the buffer pool, copies data to the buffer chunk, and returns full chunks to the buffer pool. As full chunks are returned to pool, IO threads are woken up to write these chunks to disk. When an application process finishes copying its checkpoint data, it enters phase 3 to reestablish the communication channels with other processes. At some moment $t_2$, all APs return from phase 3. At this point of time, the parallel job resumes.
its computation. From an application’s point of view, the “perceived checkpoint time” is $t_2 - t_1$. However, the checkpoint data is not completely written to disk till time $t_3$. Hence, the “actual checkpoint time” is $t_3 - t_1$. The checkpoint delay perceived by an application is only $t_2 - t_1$.

4.2.3 Design Choices

Several parameters play important roles in WAI and WAI-Staging design: (1) **Buffer pool size**. This parameter determines the degree of overlapping between application processes and IO threads. Large buffer pools provide higher opportunities to overlap, since more data can be held in the buffer pools. We measure the impact of this parameter in checkpoint creation in next section. Our experiments indicate that a moderate size of 64MB is capable of improving checkpoint writing significantly. (2) **Chunk size**. For a given size of the buffer pool, the chunk size determines the number of chunks in the buffer pool, and therefore impacts the waiting time of an application process to grab a free chunk. In our experiments we stick to 4MB chunk size.

4.2.4 Reconstruct Checkpoint Files

Our design alters the structure of data stored in the persistent storage. At restart, it’s necessary to reconstruct the data into the original checkpoint file format required by BLCR. When a data server writes the data chunks to persistent storage, the metadata for each data chunk (process ID, data size and offset of data) are accumulated and saved to an index file. At restart, a client node collects the index files from all data servers. The metadata of data chunks belonging to this client is extracted out of these index files. Using this condensed metadata, the client contacts the data servers to retrieve corresponding data. These data chunks are then concatenated in order of offset to rebuild the original checkpoint files.
4.3 Performance Evaluation of WAI

We have implemented WAI and WAI-Staging designs into BLCR-0.8.0. We have also integrated the modified BLCR into MVAPICH2 1.6 [18] checkpoint/restart framework. In this section, we conduct various experiments to evaluate the performance of our design.

4.3.1 Experimental Setup

A 64 node RedHat Enterprise Linux 5 cluster is used in the evaluation. The nodes are connected with Mellanox MT25208 DDR InfiniBand HCAs. Each node has 8 processor cores on 2 Intel Xeon 2.33 GHz Quad-core CPUs. All our experiments are based on MVAPICH2 1.6 as the MPI library with modified BLCR 0.8.2. Our experiments measure the checkpointing performance of the WAI and WAI-Staging schemes using the NAS Parallel Benchmark suite 3.2.1 [116]. Applications LU and BT are chosen because they produce relatively large checkpoint images. Each application is evaluated with class C and 64 processes on 8 compute nodes.

4.3.2 WAI: Checkpoint Writing

Figure 4.6(a) shows the breakdown of the time to do one checkpoint. The checkpoint time is decomposed into 3 phases as described in section 1. Phase 2 is further divided into 3 parts. “Buffer” denotes the time spent by a process to acquire/return buffers. “Memcpy” denotes the time for a process to copy its memory image to the acquired buffers. “Other” denotes the time spent on the rest of the operations in phase 2, such as freezing threads, synchronization, etc.

The values below each bar indicate the buffer pool sizes. A chunk size of 4 MB is used in each case. From this figure, it can be seen that WAI significantly reduces the time cost to make a checkpoint. WAI does a very good job in reducing the time spent in phase 2 to write checkpoint data. Although time spent in phase 1 and phase 3 remains relatively constant at a given problem
size, the total checkpoint time drops significantly with WAI, because it considerably improves the time spent in phase 2. In the test with BT.C.64, original BLCR incurs an overhead of 11.2 seconds to make a checkpoint (as indicated in Figure 4.2(b)), while WAI reduces this overhead to only 4.27 seconds when a 64 MB buffer pool is used. This leads to a speedup of 2.62 times. Faster checkpoint creation can be achieved when the buffer pool is enlarged, as can be seen in Figure 4.6(a).

![Diagram](a) Decomposition of Checkpoint Time  
(b) Application Execution Time w/o 3 Checkpoints in Succession

Figure 4.6: Evaluation of WAI

Figure 4.6(b) reports the overall application execution time at different buffer pool sizes. Take BT.C.64 for example. Without any checkpoint the application completes in 169.9 seconds. When 3 checkpoints are taken at equal intervals using original BLCR, the execution time is prolonged to 204.9 seconds, which implies the checkpoint overhead to be 20.77%. When WAI with 64 MB buffer pool is used to take 3 checkpoints, the application completes in 181.5 seconds. The overhead is driven down to only 6.86%.
Using large buffer pool can further reduce checkpoint time, but the improvement flattens beyond certain amount of buffer. This is because the IO threads and application processes are totally overlapped at certain buffer pool size. Increasing buffer pool beyond this level isn’t able to yield additional benefits. This “critical level” depends on an application’s virtual memory usage pattern. We plan to investigate along this direction in future work.

![Figure 4.7: IO Threads Overlapping with Application](image)

### 4.3.3 WAI: Overlapping

WAI effectively overlaps IO and computation. In this experiment we measure WAI’s overlapping time between IO threads and application processes at different buffer pool sizes. Figure 4.5 illustrates how IO threads are overlapped with application by WAI. After copying checkpoint data to the buffer pool, a checkpoint is regarded as completed at time \( t_2 \) when communication endpoints are reestablished between all processes. But IO threads haven’t finished writing checkpoint data to disk files until time \( t_3 \). The period between \( t_2 \) and \( t_3 \) is overlapping between IO and computation. Figure 4.7 reports this time at varied buffer pool sizes. The legend “N MB” represents the buffer pool size of N MB. Table 4.1 indicates that LU.C.64 generates 184 MB of data per node in
one checkpoint, while BT.C.64 generates 320 MB data per node in one checkpoint. Therefore IO threads need longer time to write BT.C.64’s checkpoint data to disk files, which leads to a longer overlapping time than LU.C.64. We also find this overlapping time tends to be shorter for a smaller buffer pool size. This is because an application process spends longer time in phase 2 to acquire free buffers at a smaller buffer pool (as can be seen in Figure 4.6(a)). IO threads start writing to files at phase 2. A longer phase 2 hides part of IO time. As a result, the remaining IO time after phase 2 becomes shorter leading to a shorter overlapping time.

Table 4.3: Overhead at Restart (seconds)

<table>
<thead>
<tr>
<th></th>
<th>Rebuild a checkpoint</th>
<th>Restart time</th>
</tr>
</thead>
<tbody>
<tr>
<td>LU.C.64</td>
<td>6.36</td>
<td>9.25</td>
</tr>
<tr>
<td>BT.C.64</td>
<td>10.72</td>
<td>9.52</td>
</tr>
</tbody>
</table>

4.3.4 WAI: Restart

In WAI, an IO thread writes each chunk of buffer to a separate file. The file name encodes the process id which generates this data, and the offset of this chunk in original checkpoint file. Reconstructing a checkpoint file requires concatenating the individual chunk files in order of their offset, the cost of which is linearly proportional to the original checkpoint file size. After recovering the checkpoint files, an application can be restarted using BLCR as usual. Table 4.3 measures the time cost to rebuild the checkpoint file for one application process. It also reports the time cost to restart an application. Take LU.C.64 for example. It takes 6.36 seconds for the application to rebuild checkpoint files for all its 64 processes. Each compute node takes care of rebuilding 8 checkpoint files. 9.25 seconds are required to restart the application LU.C.64. Cost to rebuild a
checkpoint is 68.8% of the restart cost. However the reconstruction overhead occurs only at restart after a failure. Therefore the cost of rebuilding checkpoint files is largely amortized in the lifespan of an application run.

4.4 CRFS: A Lightweight User-Level Filesystem for Generic Checkpoint/Restart

The WAI strategy in previous sections requires modifications in the MPI implementation and BLCR library to alleviate the IO contention. Although effective, this approach only works for a specific MPI stack and requires patching BLCR kernel module, which is not portable to be applied to generic environments.

We propose a scalable and portable solution that incorporates Write-Aggregation into Checkpoint/Restart File System (CRFS) which can transparently benefit a wide range of MPI stacks to reduce the IO bottleneck in Checkpoint/Restart. In contrast to other alternative optimizations implemented in MPI libraries or in checkpoint libraries, CRFS is implemented as a user-level filesystem based on FUSE [4] which is readily available in most of the clusters. A wide spectrum of upper layer components, including any MPI implementation and other generic I/O applications, can transparently benefit from CRFS’s capabilities. CRFS internally aggregates write streams from multiple processes into fewer bigger chunks, which are asynchronously written to back-end filesystem for more efficient IO. CRFS manages a flexible IO thread pool to throttle concurrent writing to alleviate the stress on back-end filesystems. By configuring these IO threads, it’s able to throttle concurrent write requests for better IO performance. CRFS can be mounted on top of any existing filesystem, such as ext3, PVFS2, NFS, and Lustre to leverage their capacity.

4.4.1 CRFS Design and Implementation

As depicted in Figure 4.8, CRFS is built on top of FUSE [4] and runs in user space as a stackable filesystem. CRFS relies on other filesystems to store the real file data, such as ext3/4, NFS, and
Lustre [14]. Users can perform any POSIX filesystem operations in CRFS as in any other regular filesystems. These system calls are intercepted by FUSE kernel module and then routed to CRFS, where proper actions are taken by calling corresponding functions from the underlying filesystem and returning the results. We will describe in detail how CRFS handles various filesystem calls in the following sections.

### 4.4.2 File Open

At the beginning of a checkpoint, each application process calls `open()` to create a new checkpoint file. This system call is caught by FUSE kernel module and redirected to CRFS. CRFS maintains a hash table to keep track of opened files. Each opened file is associated with an entry that contains metadata to be used in later I/O operations. A new entry is inserted into the table for a newly opened file. If the file is already opened, the reference counter in its table entry is incremented by one. After inserting the entry CRFS invokes the corresponding functions from the underlying filesystem to open/create the required file.
4.4.3 File Write

Concurrent file writing is a major performance bottleneck usually seen when checkpointing parallel applications [37, 93]. CRFS performs write aggregation to coalesce concurrent writes from parallel application processes into fewer larger chunks, and asynchronously write these bigger chunks to back-end storage system. With this, CRFS is able to improve file writing efficiency and reduce concurrent write contentions.

CRFS manages a buffer pool initialized at mount time. The buffer pool is divided into fixed-sized chunks. When an application writes data to CRFS, the write() system call is captured by FUSE and the control is handed to CRFS. CRFS goes to the buffer pool to grab a free buffer chunk, and associates this chunk to the file. The data to be written is copied into this chunk, the file’s metadata entry is updated and the write() returns. The metadata entry includes information such as: size of valid data in the chunk, size of the chunk, append point in this chunk, offset of this chunk in the original file, and ownership identities. All subsequent writes to the target file will be coalesced into this chunk until the chunk becomes full. This is possible because checkpoint data is written sequentially during a checkpoint. After the chunk becomes full, it is enqueued into the Work Queue. At this point we increment the “write chunk count” by one in the metadata entry to mark the outstanding full chunk writes for this file. After that, the next free chunk is allocated to this file to accommodate the following writes.

**Work Queue and IO Throttling:** Data chunks are eventually handed over to the Work Queue for actual writing. CRFS manipulates a pool of worker IO threads waiting on the work queue. Whenever a chunk is enqueued, an IO thread wakes up and fetches the chunk off the queue. Each chunk is tagged with metadata information including target file handler, offset into the file, valid data size in the chunk, etc.. The IO thread then calls a write() with the underlying filesystem to write the data to its actual file. Once completed, the “complete chunk count” in the file’s metadata
entry is incremented. Then the chunk is returned to the buffer pool to be reused, and the IO thread goes back to the work queue for the next chunk.

CRFS can configure the IO thread number to throttle the outstanding buffer chunk write requests. With that, we can mitigate the IO contentions at back-end filesystems to attain a better performance. In Section 4.5.2 we carry out experiments to find a proper IO thread level.

4.4.4 File Close

When a process has finished checkpointing, a close() is called on its checkpoint file and is eventually routed to CRFS. If there is any remaining data in the buffer chunk associated with this file it’s enqueued into the Work Queue. Then the calling thread is blocked until the “complete chunk count” becomes equivalent to “write chunk count”, which means all outstanding buffer chunk writes have been finished. After that the call returns.

4.4.5 Other Filesystem Operations

1) File Read: File reading is required when a process is to be restarted from a checkpoint file. For read() we directly pass it to the underlying filesystem without any additional operation.

2) File Sync: The fsync() system call flushes a file’s modified in-core data to the storage device where the file resides, which includes data coalesced in a file’s buffer chunk, and data in page cache for the underlying filesystems. Upon a fsync(), we first enqueue the current buffer chunk associated with the file, then wait for all outstanding chunk writes to complete. After that a fsync() is called on the underlying filesystem to flush all dirty data in page cache to stable storage.

3) Other File Operations: For a user-level filesystem to be usable, many other filesystem operations are required to be supported, such as mkdir, rmdir, link, truncate, chmod, utime, and so on. CRFS directly passes these calls to the underlying filesystem without additional processing.
4.5 Performance Evaluation of CRFS

In this section we conduct extensive experiments to evaluate CRFS performance from various perspectives including: (a) Raw performance of CRFS to aggregate concurrent write streams; (b) Checkpoint writing performance of CRFS with different MPI stacks (MVAPICH2 [18], MPICH2 [16], OpenMPI [23]) using different back-end filesystems (ext3, Lustre [14], NFS). We will also show some detailed profiling to reveal the reasons why CRFS leads to benefits.

4.5.1 Experimental Setup

In the evaluation, a 64-node InfiniBand Linux cluster was used. Each node has eight processor cores on two Intel Xeon 2.33 GHz Quad-core CPUs. Each node has 6 GB main memory and a 250 GB ST3250620NS disk drive. The nodes are connected with Mellanox MT25208 DDR InfiniBand HCAs for high performance MPI communication. The nodes are also connected with a 1 GigE network for interactive logging and maintenance purposes. Each node runs Linux 2.6.30 with FUSE library 2.8.1. We enable the “big_writes” option for FUSE to perform large writes to deliver full performance. CRFS is mounted upon different filesystems: ext3, Lustre 1.8.3, and NFS at different runs. Lustre 1.8.3 is configured using 1 Metadata server and 3 Object Storage Servers with InfiniBand transport. For the case with NFS, the single NFS server exposes its disk via NFSv3 protocol using IPoIB transport.

4.5.2 CRFS: Raw Performance

CRFS relies on FUSE to intercept filesystem operations. The write requests are coalesced into CRFS’s buffer pool, then the filled data chunks are handed to the work queue and processed by the IO threads asynchronously. Multiple factors can affect the overall performance of this pipeline, including FUSE internal overhead, buffer pool size, chunk size and IO thread numbers. In this
test we ran 8 parallel processes in a node each writing 1 GB data into CRFS. Once a filled chunk is picked up by an IO thread it is discarded without being written to a back-end filesystem. With this we can measure the raw performance of CRFS to aggregate write streams, precluding the impacts of different back-end filesystems. In real checkpoint writing we observe that too many IO threads tend to generate high level of contentions when they concurrently write chunks to backend filesystems, leading to degraded performance, while too few IO threads cannot unleash the full potential of the filesystem. After extensive experimental runs we find that 4 IO threads generally yield the best throughput for most of the situations. Due to space constraints we haven’t included the detailed studies in the paper. We stick to 4 IO threads to throttle a high degree of concurrent IO in all experiments hereafter.

![Figure 4.9: CRFS Raw Write Bandwidth (8 processes on a single node)](image)

Figure 4.9 shows the measured write throughput at different buffer pool sizes with varied chunk sizes. Given a 16 MB buffer pool, CRFS can always achieve more than 700 MB/s aggregation throughput on a single node for chunk sizes larger than 128 KB. This is sufficient to saturate most available parallel filesystems if multiple nodes concurrently write their checkpoints. Larger chunk size is generally more favorable for the underlying filesystems to exhibit full potential, therefore we would fix chunk size to be 4 MB in all experiments hereafter.
Table 4.4: Checkpoint Sizes of Different Applications with Varied MPI Stacks. MVAPICH2 and OpenMPI use InfiniBand Transport. MPICH2 uses TCP Transport.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MPI Library</th>
<th>Total Checkpoint Size (MB)</th>
<th>Process Image Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LU.B.128</td>
<td>MVAPICH2-IB</td>
<td>903.2</td>
<td>7.1</td>
</tr>
<tr>
<td></td>
<td>OpenMPI-IB</td>
<td>909.1</td>
<td>7.1</td>
</tr>
<tr>
<td></td>
<td>MPICH2-TCP</td>
<td>497.8</td>
<td>3.9</td>
</tr>
<tr>
<td>LU.C.128</td>
<td>MVAPICH2-IB</td>
<td>1,928.7</td>
<td>15.1</td>
</tr>
<tr>
<td></td>
<td>OpenMPI-IB</td>
<td>1,751.7</td>
<td>13.7</td>
</tr>
<tr>
<td></td>
<td>MPICH2-TCP</td>
<td>1,359.6</td>
<td>10.7</td>
</tr>
<tr>
<td>LU.D.128</td>
<td>MVAPICH2-IB</td>
<td>13,653.9</td>
<td>106.7</td>
</tr>
<tr>
<td></td>
<td>OpenMPI-IB</td>
<td>13,864.9</td>
<td>108.3</td>
</tr>
<tr>
<td></td>
<td>MPICH2-TCP</td>
<td>13,261.2</td>
<td>103.6</td>
</tr>
</tbody>
</table>

For a given chunk size=4 MB, it can be observed that write throughput rises as buffer pool becomes bigger and starts to flatten when buffer pool is greater than 32 MB. CRFS shouldn’t occupy too much memory since a real parallel application can use a large portion of the available memory. Hence we fix the buffer pool to be 16 MB in all experiments hereafter.

4.5.3 CRFS: Checkpointing Performance

In this section, we evaluate the performance of CRFS to checkpoint real applications. We ran NAS parallel benchmark LU of class B, C and D with 128 processes on 16 compute nodes. We chose ext3, NFS and Lustre-1.8.3 as the three underlying filesystems. The checkpoint was either directly written to native filesystems, or processed by CRFS before writing. In order to demonstrate the portability of CRFS that can benefit a wide range of applications, the experiments were repeated using three popular MPI implementations that support Checkpoint/Restart: MVAPICH2 1.6rc3, OpenMPI 1.5.1 and MPICH2 1.3.2p1.

Table 4.4 shows the checkpoint sizes at varied application scales. The three MPI stacks are tagged with “IB” (InfiniBand) or “TCP” to indicate the transport they use. Generally MVAPICH2 and OpenMPI produce checkpoint images slightly bigger than MPICH2. This is because they
use InfiniBand transport which requires more memory to maintain the communication channels. MPICH2, on the other hand, uses TCP transport and has a lower memory footprint.

The measured checkpoint time includes the time for BLCR to write the checkpointed data and the time to close the file (so there is no pending data in CRFS) for all the processes. The values plotted in the following figures are the average checkpoint time among all the processes for one given checkpoint, and the average for at least 5 checkpoints in the same conditions.

Figure 4.10 gives the checkpoint writing time for MVAPICH2. It clearly indicates that CRFS is able to diminish checkpoint writing overhead for different underlying filesystems at a wide range of application memory footprint. For example, for application class C, CRFS with Lustre reduces the writing time from 6.0 seconds to 1.1 seconds, which stands for a 5.5X speedup. With ext3 and NFS filesystems the improvements are 3.2X and 2.1X, respectively. For bigger problem size of class D the improvement is less dramatic because the majority of overhead is dominated by the absolute amount of data to dump. Even in this case CRFS is 30% faster than native Lustre and drives down the writing time from 29.3 seconds to 20.7 seconds. We observe that NFS becomes an outlier at this problem size. NFS isn’t a good candidate to store checkpoint since its single server design

42
doesn’t match the intensive concurrent IO requirements. The node-level optimizations carried out by CRFS cannot benefit NFS server to handle the high level of concurrent IO tension, and the additional overhead within CRFS (such as multiple buffer copies) starts to manifest. Therefore CRFS+NFS performs slightly worse than the native NFS.

Figure 4.11: Checkpoint Writing Time with MPICH2 (Lower is Better)

Figure 4.11 and Figure 4.12 exhibit the performance benefits gained by CRFS when running with MPICH2 and OpenMPI, respectively. A similar level of improvement is obtained here. In the case of MPICH2, for example, with application class C and Lustre filesystem, CRFS achieves a 9.3X speedup to complete checkpoint writing. The speedup is 2.4X with NFS for the same problem size. We see the same abnormality with NFS when problem size becomes larger at class D. At this problem size, CRFS optimizations cannot make a positive impact, instead its overhead appears more prominent for NFS.

In Figure 4.12(b), the bar for LU.C.128 with OpenMPI using native Lustre is missing. Despite several tries, the checkpoint in OpenMPI always failed for these conditions.
Figure 4.12: Checkpoint Writing Time with OpenMPI (Lower is Better). *We could not get the result of native Lustre for LU.C.128, despite many attempts.*

### 4.5.4 CRFS: Reasons of Improvements

The aforementioned experiments have demonstrated CRFS’s capability to improve checkpoint writing efficiency. We have also further explored the reasons why CRFS can bring about such benefits. We used “blktrace” to collect the block IO layer access traces during checkpoint writing to local ext3. We ran application LU.C.64 with 64 processes on 8 compute nodes using MVAPICH2. The results collected from one node are illustrated in Figure 4.13. The top part of Figure 4.13(a) shows the disk IO pattern of checkpoint writing. We see a high degree of randomness caused by concurrent writing from 8 processes on the same node. This enforces a lot of disk head seeks (the middle part of Figure 4.13(a)), and results in a lower effective write throughput. In contrast, CRFS is able to coalesce the concurrent write requests and perform relatively sequential writes, as can be seen in top part of Figure 4.13(b). Consequently it can avoid a lot of disk head seeks and deliver a better write throughput.
By merging concurrent write requests and reducing the level of IO contention, CRFS brings about another benefit to diminish the uncertainty of checkpoint writing completion time. Figure 4.14 compares the cumulative checkpoint write time for all the processes. We see a wide variation in the completion time if writing directly to ext3. In this mode, the slow writing processes hinder the overall progress of all processes in the application, resulting in a prolonged checkpoint completion time. On the contrary, CRFS effectively drives down the write contentions at back-end filesystem so as to significantly minimize this variation. As a consequence, all processes converge and finish their writing at about the same time, as shown in Figure 4.14. This helps achieve a quicker resumption of the application execution after a checkpoint is carried out.

### 4.5.5 Restart

During restart, BLCR library reads from checkpoint files and restores the in-memory context for every process. CRFS forwards every read request to the back-end filesystem, and does not impose any additional overhead on file reads. What’s more, CRFS doesn’t change any file layout
during checkpoint write phase. Consequently an application can be restarted directly from the back-end filesystem, without the need to mount CRFS. In our experiments, we did not observe any noticeable improvement in the application restart time when CRFS is mounted atop an underlying filesystem.

4.6 Related Work

Checkpointing an application and restarting it from the last checkpoint is a widely adopted mechanism for serve fault tolerance. Many works have been done to provide checkpoint/restart facilities for stand-alone applications [36, 49, 83, 95, 120]. Checkpoint/restart mechanisms have been incorporated into MPI libraries such as LAM/MPI [99], MVAPICH2 C/R [96], MPICH-V [40] and OpenMPI [23].

The overhead of checkpoint/restart on file IO has been studied by [63]. Milo etc. [29] proposes the use of log-based file structures at the server side to serialize all file writing requests for checkpoint. This structure is optimized for a checkpoint writing pattern where multiple processes write to a single file. The server has to be altered to adopt this file structure which makes it infeasible for many existing applications. Additionally this approach modifies the PVFS [25]
filesystem to serialize all file writing requests for checkpointing. This approach isn’t portable because it requires changing the filesystem, and impedes data reading throughput because additional re-mapping is needed for every read request to find its data. Stdchk \cite{33} tries to scavenge spare storage resources from all participating nodes to form a dedicated storage space for checkpoint data. Our work differs from it in that we focus on utilizing local residual memory as a buffer pool. \cite{79} proposes a CLL algorithm to reduce checkpoint overhead. It’s a user-level optimization, and its buffer management incurs significant overhead to synchronize the copier thread and application thread on every common page access. On the contrary, our work is purely in kernel level, and our algorithms synchronize at chunk level which can largely mitigate buffer management overhead. The authors of \cite{37} proposed a parallel log-structured filesystem (PLFS) to improve the writing throughput. However, this solution only deals with N-1 scenario where multiple processes write to the same shared file, hence it cannot handle MPI system-level checkpoint
Chapter 5: A HIERARCHICAL DATA STAGING FRAMEWORK TO ACCELERATE CHECKPOINT/RESTART

In this chapter we propose a hierarchical data staging architecture that uses a dedicated set of staging server nodes to offload the tedious checkpoint writing so as to hide the checkpoint latency from the application. This component is shown as the highlighted box in Section 5.1 of our proposed research framework. We start off by taking a close look at the existing checkpoint approach to reveal its drawbacks in Section 5.1. In Section 5.2 we propose a hierarchical data staging architecture to overcome the limitations. The performance evaluation is presented in Section 5.3. Related work is discussed in Section 5.4.

5.1 Problems with Sequential Checkpoint Writing

With existing checkpoint approach, all application processes write their checkpoint data directly to the shared filesystem. As a result, each application process independently issues a sequence of VFS writes to separate checkpoint files. If not optimized, the interference of intermixed VFS write streams can severely degrade aggregated write bandwidth.

The application is not allowed to resume until all processes finish writing. This sequential model is illustrated at the left side of Figure 5.2. Due to the heavy I/O burden imposed on the shared filesystem by the concurrent writing from many processes, the parallel writes get multiplexed and the achievable aggregate throughput is severely compromised [37, 93]. This increases the time for
which the application blocks, waiting for the checkpointing operation to complete. As a result we observe a high checkpoint time cost in terms of application execution [88].

With the rapid advances in technology, many clusters are being built with high performance commercial components such as high-speed low-latency networks and advanced storage devices such as Solid State Drives (SSDs). These advanced technologies provide an opportunity to redesign existing solutions to tackle the I/O challenges imposed by Checkpoint/Restart. We propose to insert an interposition layer between the application VFS writes and the actual data movement between client and data servers. This layer aggregates all VFS writes from the application, and takes care of moving data to data servers. In the next Section we propose a hierarchical data staging architecture to address the I/O bottleneck caused by Checkpoint/Restart.
Figure 5.2: Comparison between the direct checkpoint and the checkpoint staging approaches

5.2 Hierarchical Data Staging Framework

In this section we present a hierarchical staging I/O framework improve write bandwidth for checkpoint writing. As shown in the right part of Figure 5.2, with the staging approach, the staging nodes are able to quickly absorb the large amount of data thrust upon them by the client nodes during a checkpoint, with the help of the scratch space provided by the staging servers. Once the checkpoint data has been written to the staging nodes, the application on the compute node can resume. Meanwhile the data transfer between the staging servers and the shared filesystem takes place in background and overlaps with the computation. Hence, this approach reduces the idling time of application due to the checkpoint.

The central principle of our Hierarchical Data Staging Framework is to provide a fast and temporary storage area in order to absorb the I/O load burst induced by a checkpointing operation. This fast staging area is governed by a Staging Server. In addition to what a generic compute-node is configured with, staging servers are over-provisioned with high-throughput SSDs and high-bandwidth links. Given the fact that such hardware is expensive, this design avoids the need to install them on every compute-node.
Our proposed staging framework consists of two components: the computer nodes side (client side) and the staging server side (server side). As indicated in Figure 5.3, the shadowed boxes represent our new design on both client and server sides.

![Checkpoint Write Staging](image)

Figure 5.3: Checkpoint Write Staging

### 5.2.1 Write Aggregation on Client Side

In order to tackle the IO contention at storage servers caused by direct checkpointing, we propose to insert an interposition layer between the application VFS writes and the actual data movement between client and data servers. This layer aggregates all VFS writes from the application, and takes care of moving data to data servers.

In our new strategy, the “Write Aggregation” module prepares a buffer pool by registering the buffer pool to the InfiniBand hardware. Each application process will request the “Write Aggregation” module to allocate a memory chunk from the buffer pool. As shown in Figure 5.3 on client side, when a VFS write is called by application process, instead of following the usual path...
to store data to a VFS buffer cache, the data is directly copied to the memory chunk associated to that process. When the chunk is full, the process requests another free chunk from the “Write Aggregation” module, and continues with checkpoint writing. Meanwhile for every filled chunks in the buffer pool, “Write Aggregation” module delivers a “RDMA-Read request” to data server. The data server then pulls the data to server side through RDMA Read. By aggregating many VFS writes into a buffer pool at client side, we expect to harvest significant improvement in file writing bandwidth at the cost of additional memory usage. A recent study [28] suggests that even large scale parallel jobs seldom use all available local memory. Therefore we feel it is reasonable to assume that residential memory is available in the client side. Our experiments indicate that even a mildly-sized buffer pool (64MB) can greatly improve the write bandwidth.

Design alternatives exist to choose a proper buffer pool size and chunk size. Large buffer pool always help to improve the write bandwidth. Our experiments choose a medium size 64MB to be the buffer pool size. Once the chunk size is reasonably large (greater than 1MB), we find that the performance won’t change much. So we will stick to 4MB chunk size in this thesis.

5.2.2 Staging Area on Server Side

The data server maintains a queue to receive all requests from clients. Each request contains information such as: process ID, data size, offset of data in original checkpoint, buffer address to be used in RDMA-Read and remote key to be used in RDMA-Read. Once a request is enqueued, it’s dispatched to a free IO thread out of the IO thread pool. This IO thread grabs a free chunk of memory in the local buffer pool, then issues a RDMA-Read operation to pull data from client memory to server memory. Once the data is present in the memory chunk, the IO thread appends this data chunk to local storage in a log-based file structure [29]. The persistent storage can be a disk file, or even a raw block device. The metadata about this chunk (process ID, data size, offset
of data, physical offset in data server storage, etc.) is also saved for all IO threads. After the file
write finishes, the IO thread sends a completion message back to the client. The client will then
release the memory chunk to be used by other processes.

In our design, we choose to let the client expose its memory to server, so server can perform
RDMA-Read to pull the data. This approach owns the merit of better security over the alternative,
where server exposes its memory to client and client performs RDMA-Write to push the data to
server.

5.3 Performance Evaluation

In this section, we conduct various experiments to evaluate the performance of our design. A 64
node RedHat Enterprise Linux 5 cluster is used in the evaluation. Each node has 8 processor cores
on 2 Intel Xeon 2.33 GHz Quad-core CPUs. The nodes are connected with Mellanox MT25208
DDR InfiniBand HCAs. All our experiments are based on MVAPICH2 1.6 as the MPI library with
BLCR 0.8.0.

The client side “Write Aggregation” design can be implemented as a stackable file system to
intercept the VFS write system calls. In this experiment, we have modified BLCR kernel module
to redirect VFS writes to the aggregation module. Client buffer pool is fixed to 64MB with 4MB
chunk size. At the server side Staging Area design, 16 IO threads are used in the thread pool. The
same buffer pool size (64MB) and chunk size (4MB) are used. Both hard disk and Solid State Disk
(SSD) are tried as storage device at the server side. Table 5.1 gives the raw write bandwidth of
these two kinds of storage devices.
Table 5.1: Device Raw Write Bandwidth (MB/second)

<table>
<thead>
<tr>
<th>Device</th>
<th>Write</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard Disk</td>
<td>57</td>
<td>65</td>
</tr>
<tr>
<td>SSD (Intel X-25E 64GB)</td>
<td>179</td>
<td>202</td>
</tr>
</tbody>
</table>

5.3.1 Aggregated Write Bandwidth

We first run a synthetic benchmark to measure the aggregated write bandwidth achieved by our Aggregation Staging IO strategy. In this benchmark, 4 nodes act as data servers. All participating client nodes synchronize at a barrier, then start to write 1GB data at 4MB chunk size to the data servers. When all client nodes finish writing, they enter another barrier, after which the elapsed time $T$ is measured. The aggregated write bandwidth is derived as $(\text{TotalDataAmount})/T$. Both PVFS2 and Aggregated Staging IO strategy are evaluated as a comparison. Only buffered IO mode is tested for PVFS2, while both buffered IO and direct IO modes are evaluated with Aggregated Staging IO strategy. The data servers use hard disk and SSD as storage device at different runs. Figure 5.4 shows the aggregated write bandwidth using 1/2/4/8 client nodes.

PVFS2 with hard disk is used as a baseline for comparison. As client node number varies from 1/2/4/8, the aggregated write bandwidth is 313/560/497/262 MB/s, respectively. It’s clear that the interference between multiple data streams at data servers lead to a performance degradation when 8 client nodes are used. Since SSD has no mechanical seek latency as hard disk, the write stream interference should exhibit little impact on its write throughput. This is consolidated by our experiment to substitute disk with SSD. PVFS2+SSD achieves a aggregated write bandwidth of 310/556/601/643 MB/s, a big improvement over PVFS2+disk at 8 client nodes.

As a comparison we measured the performance of Aggregation Staging IO strategy. Staging IO with hard disk and buffered IO reaches aggregated write bandwidth of 1176/1468/1560/1654
MB/s when using 1/2/4/8 client nodes. Our strategy outperforms PVFS2+disk by 6.31 times at 8 client nodes. When replacing disk with SSD (Staging IO with SSD and buffered IO), we can achieve write bandwidth of 1266/2184/2744/2575 MB/s, 55% higher than Staging IO with disk at 8 clients. This improvement can be attributed to the better raw bandwidth of SSD over hard disks. If newer PCIe based SSD like Fusion-IO [7] is used, we will expect to gain more benefit in bandwidth. At 8 client nodes, Staging IO+SSD outperforms PVFS2+SSD by 4 times.

When direct IO is used instead of buffered IO, our staging IO with disk reaches write bandwidth of 145/151/180/180 MB/s. If using SSD as storage, the write bandwidth is 593/713/698/716 MB/s. Our strategy can saturate the 4 SSDs even with 2 client nodes.

5.3.2 Application Checkpoint Time

In this test we measure the time to checkpoint some parallel applications (LU and BT) from NAS parallel benchmark of class C and 64 processes. 8 compute nodes are used to run the application, and 4 storage nodes act as data servers. The benchmarks are compiled with MVAPICH2 1.6 Checkpoint/Restart and modified BLCR 0.8.0.
Figure 5.5 gives the time cost to write checkpoint data of application LU with different strategies. For PVFS2 with disk, 12.4 seconds are required to write the checkpoint data. PVFS2+SSD uses 7.3 seconds to complete the writing. If Staging IO with disk and buffered IO is used to write checkpoint data, the writing can be completed in 1.25 seconds, which is 9.9x improvement over PVFS2+disk. This time can be further driven down to 1.07 seconds if replacing disk with SSD. As with direct IO, Staging IO with disk can finish checkpoint writing in 8.83 seconds. Staging IO with SSD can finish writing in 2.75 seconds.
The similar trend can be observed for application BT in Figure 5.6. The checkpoint writing time can be reduced from 19.67 seconds (PVFS2+disk and buffered IO) to 2.02 seconds (Staging IO + disk and buffered IO), which represents a 9.7 times improvement. This time is driven down to 1.43 seconds by Staging IO with SSD and buffered IO.

5.4 Related Work

Checkpoint/Restart is supported by several MPI stacks [42, 53, 60] to achieve fault tolerance. Checkpoint is well known for its heavy I/O overhead to dump process images to stable storage [63, 94]. [88, 89] explored how to utilize write aggregation to improve checkpoint writing to a local file system. In this paper we extend the work to store checkpoint data to a shared parallel storage system.

SCR [85] is similar to our work in the sense that it builds a multi-level checkpoint storage system. SCR stores checkpoint data to the local storage on compute nodes to improve the aggregated write throughput. Additionally SCR stores redundant data on neighbor nodes to tolerate failures of a small portion of the system, and it periodically copies locally cached data to parallel filesystem to tolerate cluster-wide catastrophic failures. Our approach differs from SCR in that a compute node stages its checkpoint data to its associated staging server, such that the compute node can quickly resume execution while the staging server asynchronously moves checkpoint data to a parallel filesystem.

OpenMPI [59] proposes a feature to store process images to node-local filesystem, and later copies these files to a parallel filesystem. Dumping a memory-intensive job to a local filesystem is usually bounded by the local disk speed, and it is hard to work on disk-less clusters where RAM
disk is not feasible due to the high application memory footprint. Our approach aggregates node-local checkpoint data and stages it to a dedicated staging server, which takes advantages of high bandwidth network and advanced storage media such as SSD to achieve good throughput.

Isaila et al. [64] designed a two-level staging hierarchy to hide file access latency from applications. Their design is coupled with Blue Gene’s architecture where dedicated I/O nodes service a group of compute nodes, and not all clusters have such a hierarchical structure.

DataStager [31] is generic service for I/O staging which is also based on InfiniBand RDMA. However, our work is specialized for the Checkpoint/Restart. Thus, we can optimize the I/O scheduling for this scheme. For example, we give the priority to the data movement from the application to the staging nodes to shorten the checkpoint time from the application perspective.
Chapter 6: HIGH PERFORMANCE PROCESS MIGRATION OVER INFINIBAND

Although Checkpoint/Restart (C/R) is widely adopted as a practical solution for fault tolerance, it’s not capable enough to meet the demands of upcoming exascale systems, due to its heavy I/O overhead. Process migration has already been proposed as a pro-active fault-tolerance mechanism to complement C/R. Migration overcomes the two key drawbacks of C/R, namely the unnecessary dumping of all processes’ snapshots and the re-submit queuing latency during restart. Job Migration can work in synergy with C/R by significantly reducing the frequency of full checkpoint [114]. Additionally, process migration is also a desirable feature to meet many other demanding requirements such as cluster-wide load balancing, server consolidation, performance isolation and ease of management. Hence any progress to improve process migration performance will likely be perceived by a wide spectrum of demanding cluster applications.

Several popular MPI implementations have provided support for process migration, including MVAPICH2 [18] and OpenMPI [23]. But these existing solutions cannot yield a satisfactory performance. In this chapter we propose a new protocol that can drastically reduce the overhead in process migration. This work is shown as the highlighted box in Section 6.1 of our proposed research framework.
We first study different migration approaches in order to understand the root causes of overhead in process migration in Section 6.1. In Section 6.2 we introduce the detailed design. Performance evaluation is presented in Section 6.3. Section 6.4 discusses related work.
6.1 Profiling Process Migration

In this section we study different migration approaches in order to understand the root causes of overhead in process migration. We consider three non-optimal process migration approaches already implemented in MVAPICH2 [18], as illustrated in Figure 6.2.

**Local filesystem-based migration:** The processes are checkpointed into image files stored in a local filesystem (EXT3 in this paper). Then the image files are transferred via the 1 GigE to the local filesystem on the target node. After all data are received the processes are restarted on the target node. We denote this approach as “Local” for brevity.

**Shared filesystem-based migration:** The processes are checkpointed into image files stored in a shared filesystem (PVFS2 [25] in this paper). Then the target node restarts the processes by reading data from the shared filesystem. It’s denoted as “Shared”.

**RDMA-transfer with local filesystem:** The processes on a source node are checkpointed and the data is aggregated into a staging buffer pool. Meanwhile a set of IO threads transfer the data to the target node via RDMA. On target node the data is saved as checkpoint files in local filesystem. Later on, these files are used to restart the processes. It’s named as “RDMA+Local”.

6.1.1 Characterizing Process Migration Protocols

A Process Migration can be characterized with a 5-step series:

**Step 1: Suspend.** Once a migration is initiated, all processes of the application shall suspend their communication activities and drain all in-flight messages. If the transport utilizes high-performance network with RDMA support, their communication end-points shall be torn down. This step is necessary for all processes to reach a consistent global state in order to checkpoint individual processes, due to the native characteristics of RDMA-capable networks. The reasons are detailed in [119].
**Step 2: Process Snapshot (Write).** Once the application is suspended, a snapshot is taken for each process on the source node. MVAPICH2 uses BLCR [57] to checkpoint individual process images into files. For the “Local” and “Shared” approaches, BLCR directly writes the process images on the chosen filesystem. For “RDMA+Local” approach, the BLCR library has been modified to aggregate all writes into a staging buffer pool. Simultaneously a set of I/O threads transfer the data to the target node through RDMA (Step 3).

**Step 3: Process image transfer (Transfer).** This step consists in the transfer of the process images from the source node to the target node. Depending on the considered approach, this transfer can be in different forms. In “Local” approach the process images are transferred directly from the source node to the target node using the `scp` command. In “Shared” approach the data transfer is implicit: during Write operations in Step 2, data is transferred to the shared filesystem. Regarding the “RDMA+Local” approach, the process image data is transferred by chunks directly to the target node using RDMA capabilities of the InfiniBand network. On the target node, a file on the local disk is created for each process image that has been transferred.

**Step 4: Process Restart (Read).** This step loads the process images and restarts the application processes on target node. This task is achieved by the BLCR library by reading process image files from the local filesystem (for “Local” and “RDMA+Local” approaches) or from the network shared filesystem (for the “Shared” approach).

**Step 5: Re-connection.** Once the processes have been restarted on the target node, all processes of the application synchronize and rebuild their communication end-points and resume their communication activity. Once this is done the application has been successfully migrated.

The five steps above represent the elementary operations that need to be performed to realize process migration of an MPI application. However, depending on the considered approach, these steps can overlap with each other. Figure 6.3 shows how these steps overlap in different migration
Local filesystem-based migration

\[\text{Write} \quad \text{Transfer} \quad \text{Read}\]

Shared filesystem-based migration

\[\text{Write} \quad \text{Read} \quad \text{Transfer 1} \quad \text{Transfer 2}\]

RDMA+Local filesystem-based migration

\[\text{Write} \quad \text{Read} \quad \text{Transfer}\]

Time

Write (step 2) = Process snapshot (including writing process images)
Transfer (step 3) = Process image transfer from source node to target node
Read (step 4) = Process restart (including reading process images)

Figure 6.3: Step Overlaps in the Different Migration Approaches

approaches. In this figure, we only show steps 2, 3 and 4 for simplicity purpose (steps 1 and 5 don’t differ in these approaches).

In the case of Local based migration, these three steps are serialized. Each step waits for the previous one to complete before starting. In the Shared based approach, there are two data transfer steps: one from the source node to the shared filesystem server and another one from the server to the target node. These transfers overlap with the Write and Read steps. However, Step 2 (Write) and Step 4 (Read) are serialized because the process restart step has to wait for the process images to be fully written before restarting. Finally, the RDMA+Local based migration has only one transfer step which overlaps with the Write step. However, similar to the Shared based approach, the Step 4 (Read) waits for the process image files to be fully transferred before restarting.
6.1.2 Profiling Process Migration

We run the three non-optimal process migration protocols and collect detailed profiling about the time cost to complete one migration, which relocates 8 processes from one compute node to another spare node. Three applications LU/BT/SP with input class C are used in the profiling (detailed in Section 6.3). Figure 6.4 breaks down the time cost into different steps.

We find data write is responsible for part of the time cost during a migration for both Local and Shared strategies. RDMA+Local takes advantage of RDMA transport to minimize the cost to write and transfer data. Local pays some price to transfer data using scp command via 1 Gigabit Ethernet. Both Local and RDMA+Local incur heavy overhead during restart phase. With both strategies, the processes being restarted load data from local files concurrently, which results in severe contentions in the filesystem. The Shared approach, on the other hand, avoids this cost by fetching data from PVFS data servers via high performance InfiniBand network. However this efficiency is obtained because the processes are reading from page cache on data servers. In this profiling, up to 320MB data is migrated and stored in the 4 PVFS data servers. Hence all data can be buffered in page cache. In a production deployment, the shared filesystem is likely to be
servicing multiple intensive data streams concurrently. One cannot expect his data will be stored in page cache for fast retrieval.

The characterization and profiling results both indicate that all the three steps (Write, Transfer, Read) should be taken care of in order to improve process migration efficiency. In the next section, we propose our new design that can fully overlap the three major components to achieve a better performance.

6.2 Pipelined Process Migration with RDMA

In this section, we elucidate the design of our new Pipelined Process Migration with RDMA (PPMR) protocol implemented in MVAPICH2 [18], which addresses the performance issues of current migration approaches as presented in Section 6.1.

As mentioned earlier, the process migration framework in MVAPICH2 is based on the BLCR library [57] for checkpointing and restarting individual processes. BLCR uses files as a medium to dump the snapshot of a process. The image of a process being restarted is loaded from the same file. This constitutes an implicit barrier between Step 2 (Write) and Step 4 (Read) of the process migration model. As illustrated in Figure 6.3, current migration approaches wait until the complete process snapshot is dumped to a file (Step 2) before proceeding to the process restart phase (Step 4).

This sequential file-handling is a bottleneck during process migration. It can be resolved by overlapping process snapshot (Step 2) and the process restart (Step 4). This is possible because BLCR handles checkpoint files in a sequential manner, from beginning to end, like a stream.

In this manner, the entire process can be streamlined into three fundamental steps, Write to, Transfer across and Read from a pipeline that moves the process images from the source node to the target node, as shown in Figure 6.5. Previous studies [108,114] exploit TCP socket to build this
streamline. However, the TCP socket based design is subject to high protocol processing overhead of TCP/IP.

We propose the Pipelined Process Migration with RDMA (PPMR) protocol to fully streamline the data movement by leveraging high performance RDMA capabilities. The PPMR architecture is depicted in Figure 6.6. In the figure, the Application processes has BLCR library linked in, which will take care of checkpointing this process. We leverage FUSE library to intercept file I/O system calls and aggregate them into data streams directed to a buffer pool. Buffer mangers on both migration source node and target node cooperate to move data chunks between the two nodes.
Conceptually PPMR maintains the 5-step migration cycle as described in Section 6.1, but with steps 2 to 4 fully pipelined. Step 1 and 5 remains the same in this new design. Below we will discuss Step 2/3/4 in details.

**Step 2: Process Snapshot (Write).** In this step, the unmodified BLCR library writes the process image data by making write() system calls to a virtual filesystem which is backed by FUSE. These calls are intercepted by FUSE module. The data is coalesced into a buffer chunk that is taken from the shared buffer pool. When a buffer chunk is filled up, it’s returned to the pool marked as full, while the next free chunk is grazed to continue the aggregation.

**Step 3: Data Transfer (Transfer).** Once a buffer chunk is filled by Step 2, the buffer manager on the source node sends a RDMA-Read request to its counterpart on the target node. This request contains two types of information: (1) RDMA information for the target buffer manager to perform a RDMA Read to pull over the data, and (2) the information (such as process rank, data size, offset of the data) based on which the chunks belonging to the same process can be concatenated into the proper positions in a separate linked list. Upon such a request, the target buffer manager will grab a free chunk from the buffer pool and issue a RDMA Read request to pull over the data. Once the RDMA Read is complete, a reply is sent to tell the source buffer manager to recycle a buffer chunk. The newly filled chunk will be placed to a proper position in a linked list to be used by Step 4.

It’s worthwhile to note that PPMR protocol can utilize both RDMA-Read and RDMA-Write mechanisms to transfer data, with slight changes to the control message exchanges between the source/target buffer managers. Both can achieve approximately the same bandwidth given that we transfer data at big chunk sizes (128KB as indicated in later sections). Due to space constraints we only present the design and experiment results with RDMA-Read.
Step 4: Process restart (Read). In this step, the BLCR library restarts the process on the target node. In this purpose, it reads the process images from files in a virtual filesystem that is built on top of FUSE in a way similar to the Step 2. After these read() system calls are intercepted by FUSE, we scan the linked list consisting of received data chunks to locate the data being requested. If the data is found, it’s returned to the reading process. Otherwise the process is blocked till the demanded data chunk arrives. This is possible because the checkpoint data is generated sequentially at the source node, while BLCR reads the process image data sequentially and only once at the target node. When all data contained in a chunk has been read by a process, that chunk is recycled to receive new data coming from the source node.

PPMR enables the Write, Transfer and Read steps to be seamlessly pipelined. The other mechanisms discussed in Section 6.1, on the other hand, requires a temporarily storage to keep the whole process images. Since this data is potentially too large to fit in memory, a local or shared filesystem is used to stored the process images, which create a new bottleneck in the migration.

This pipelined design allows to synchronize the throughput of the three steps Write, Transfer and Read. It has the advantage of requiring only a small temporarily storage to stream the chunk, which corresponds to the buffer pool whose size is small (a few megabytes). The migration throughput for different buffer pool sizes and different chunk sizes is studied in the next section.

6.3 Performance Evaluation

We have implemented the Pipelined Process Migration with RDMA (PPMR) protocol into MVAPICH2 [18]. In this section we conduct extensive experiments to evaluate its performance from various perspectives including: (a) Raw performance of PPMR to pump data through the pipeline from source node to target node; (b) Time cost to perform a process migration using PPMR
in comparison to other existing mechanisms; (c) Scalability of PPMR protocol with applications of varied memory footprints, and with different levels of process multiplexing.

In the evaluation a 64-nodes InfiniBand Linux cluster is used. Each node has eight processor cores on two Intel Xeon 2.33 GHz Quad-core CPUs. Each node has 6GB main memory and a 250GB ST3250620NS disk drive. The nodes are connected with Mellanox MT25208 DDR InfiniBand HCAs for high performance MPI communication and process migration. The nodes are also connected with a 1 GigE network for interactive logging and maintenance purposes. Each node runs Linux 2.6.30 with FUSE library 2.8.1. We enable the “big_writes” mode for FUSE to perform large writes to deliver full performance. “Shared” migration protocol uses PVFS-2.8.2 [25] with InfiniBand transport and 4 dedicated nodes to store both data and metadata.

6.3.1 Raw Data Bandwidth

PPMR’s performance is eventually bounded by how fast it’s able to aggregate data at the source node and how quick the data can be pipelined to the target node. Multiple elements can play a role here. In this section we examine PPMR’s raw performance including:

**Aggregation Bandwidth**: how fast the data from user processes can be aggregated via FUSE module into the shared buffer pool at the source node, assuming the buffer pool is large enough to hold all data. This corresponds to letter “A” in Figure 6.6.

**Network Transfer Bandwidth**: how fast PPMR is able to transfer data from the source node’s buffer pool to target node’s buffer pool. This is letter “B” in Figure 6.6.

**Pipeline Bandwidth**: how fast the data from user processes on the source node can be pumped through the whole PPMR pipeline to the buffer pool on the target node. This is represented by letter “C” in Figure 6.6.
Since PPMR relies on RDMA Read to transfer data, we first measure the raw bandwidth of RDMA Read using a microbenchmark “ib_read_bw” which is part of the InfiniBand Driver stack OFED-1.5.1 [22]. This microbenchmark issues 100 back-to-back RDMA Read requests to read a certain sized data chunk from another node, and measures the attained bandwidth. Figure 6.7 shows the results at varied read sizes. The network is saturated with chunk size $\geq 16$KB. This indicates that PPMR’s data transfer chunk size should $\geq 16$KB to better utilize InfiniBand bandwidth.

![Figure 6.7: InfiniBand RDMA Read Bandwidth](image)

Figure 6.8: Aggregation Bandwidth (Higher is Better)
6.3.2 Aggregation Bandwidth

In this section we examine the *Aggregation Bandwidth* as defined before. In one compute node we start multiple I/O processes. Every process makes a series of write() system calls to write 1GB data. Each write() contains 128KB data since FUSE internally coalesce data into 128KB units in “big writes” mode. Once a buffer chunk is filled up with data, the data is discarded and the chunk is returned to the pool immediately to be reused.

Figure 6.8(a) and Figure 6.8(b) report the total *Aggregation Bandwidth* with 1MB/8MB buffer pool respectively at varied buffer chunk sizes. We observe poor bandwidth with scarce memory (1MB) and large chunk size (>256K). In this case the IO processes are frequently blocked waiting for a free chunk from buffer pool, which detrimentally affects the performance. When 8MB buffer pool is used in Figure 6.8(b), the total write bandwidth reaches the peak of about 800MB/s at 16 processes, and 128KB chunk size can yield the best throughput. This is understandable since 128KB chunk size matches FUSE’s internal 128KB write units. As number of IO processes increases to 32, the FUSE internal worker threads are overloaded with more frequent lock/unlock overhead, hence the performance begins to drop.

Figure 6.9: Network Transfer Bandwidth and Pipeline Bandwidth (Higher is Better)
In Figure 6.8(c) we vary the buffer pool from 1MB to 32MB with chunk size fixed to 128KB. We observe that *Aggregation Bandwidth* isn’t very sensitive to buffer size as long as there are reasonable number of buffer chunks in the pool. All these results strongly implies that PPMR will be more likely bounded by the capability of FUSE module to coalesce user processes’ write streams instead of amount of buffer, as long as a moderate buffer pool (8MB for example) is provisioned. FUSE intercepts write() system calls from the user processes and every such a call incurs multiple memory copy overhead to move user data through FUSE internal memory to the buffer pool. Recently FUSE developers have realized such a performance hit, and zero-copy protocol has been proposed [5]. Our design can transparently benefit from such a light-weight implementation once it’s available.

### 6.3.3 Network Transfer Bandwidth

In order to measure *Network Transfer Bandwidth* we run multiple IO processes on a source node. Each process grabs a free memory chunk from the buffer pool and sends a request to the target node. The latter will perform a RDMA Read to pull the chunk. Once RMDA Read is complete, the data is discarded and the target node sends a reply to the source node to recycle its data chunk. Figure 6.9(a) shows the obtained transfer bandwidth with fixed chunk size (128KB) and varied buffer pool size. With fewer IO process we see an under-utilization of the network bandwidth because of PPMR’s control message overhead. With >= 8 IO processes we are able to saturate the InfiniBand network with 8MB buffer pool.

### 6.3.4 Pipeline Bandwidth

It is bounded by the smaller one of *Aggregation Bandwidth* and *Network Transfer Bandwidth*. We run multiple IO processes on a source node, and each process performs write() system call to write 1GB data in different chunk sizes. These writes system call are coalesced by FUSE module
and redirected to the buffer pool. Then the data chunks in the buffer pool is RDMA Read by the target node, in a way similar to how we measure Network Transfer Bandwidth. As shown in Figure 6.9(b), Pipeline Bandwidth reaches a peak of around 750MB/s with 8 IO processes and chunk size=128KB using 8MB buffer pool. Figure 6.9(c) also asserts that Pipeline Bandwidth isn’t sensitive to buffer pool sizes.

Figure 6.10: Time to Complete One Migration (Lower is Better)

6.3.5 Process Migration Performance

In this section we evaluate the process migration performance on a set of applications taken from NAS parallel benchmark (NPB) [116] suite version 3.2. All experiments use MVAPICH2 1.6RC1 [18] as the MPI library and BLCR 0.8.2 [57]. The buffer pool is set to be 8MB on all nodes with chunk size 128KB. Due to space constraints we choose applications LU/BT/SP with class C input and 64 processes running on 8 compute nodes. Three spare nodes are prepared as migration targets. In one migration 8 processes are moved from a compute node to a spare node. We simulate the migration trigger by firing a user signal to the Job Manager. Other mechanism such as node health monitoring events can also be used to kick off a migration.
Table 6.1: Migration Time Cost with Different Memory Footprint (BT.C/D, 64 processes on 8 compute nodes)

<table>
<thead>
<tr>
<th>Application</th>
<th>Migrated Data</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>PPMR</td>
</tr>
<tr>
<td>BT.C.64</td>
<td>320 MB</td>
<td>3.5</td>
</tr>
<tr>
<td>BT.D.64</td>
<td>3472 MB</td>
<td>9.1</td>
</tr>
</tbody>
</table>

Figure 6.10 illustrates the time cost to perform a process migration with different strategies. In the example of application BT.C.64, PPMR is able to complete one migration in 3.1 seconds. This translates into 10.7X speedups over the “Local” approach (33.3 seconds). It’s also 2.3X faster than the “Shared” approach (7.1 seconds), and 4.3X faster than “RDMA+Local” mechanism (13.5 seconds).

We have also assessed the application execution time without any migration and with three migrations using different migration strategies. As shown in Figure 6.11, PPMR extends the base execution time by 5.1% for LU.C.64. As a contrast, “Shared” and "Local” strategies prolong LU.C.64’s execution by 9.2% and 38%, respectively.

Figure 6.11: Application Execution Time with and without 3 Migrations
6.3.6 Scalability of PPMR

In this section we evaluate PPMR’s scalability from two aspects: (1) Efficiency to migrate applications with different memory footprints (Memory scalability). (2) Efficiency to migrate varied number of process in a same node (Multiplex scalability).

First we run application BT with inputs C/D on 8 compute nodes (8 processes per node), with each node generating respectively 320MB/3472MB memory contents to be migrated in one migration (an 10.9X increase in memory footprint). As indicated in Table 6.1, PPMR finishes a migration in 3.5/9.1 seconds for input C/D respectively which represents a 2.6X increase in time cost. The “Shared” strategy spends 7.4/54.4 seconds to handle a node migration (an increase of 7.3 times).

We then run application LU.D on 8 compute nodes with $2^n$ processes (8 to 64). This leads to different number of processes (1 to 8) to be moved in one migration but with approximately the same amount of data (around 1500MB). As revealed by Section 6.1, process multiplexing is a major cause of IO overhead, and this experiment is able to expose the efficiency of different migration protocols to handle this bottleneck. Figure 6.12 illustrates the decomposed cost to complete a migration with PPMR. As processes increase from 1 to 4 the data movement cost(tagged as...
“Pipeline”) drops slightly because better *Pipeline Bandwidth* is achieved (as is seen in Figure 6.9(b) and 6.9(c)). The performance stays constant at 8 processes. With number of processes per node keeps increasing in multicore platforms, process multiplexing is becoming more challenging. Figure 6.12 indicates that the PPMR mechanism is able to deliver good performance to address this challenge.

### 6.4 Related work

In the field of High Performance Computing systems, many efforts have been carried out to provide fault tolerance to MPI applications. Generally, the application state is periodically saved and used to restart the application when a failure occurs and the checkpoint is coordinated among the processors [45]. CoCheck [108], Starfish [32], LAM/MPI [99], among others, implement this class of checkpointing. These coordinated checkpoint approaches share a downside in that all processes must save their process images in a coordinated manner, which imposes a heavy burden on the IO subsystem. On the other side, message logging fault-tolerance protocols have tried to re-build the state that the application had before the failure. This is achieved by replaying the nondeterministic events (usually messages) to processes that failed. The different variants are called optimistic, pessimistic or causal [50] and are notably implemented in MPICH-V [41]. However, message logging protocols have a large memory overhead and may cause latency in the message processing.

Recent research directions have focussed on proactive Fault Tolerance proposing to migrate a process on a spare node before the failure happens. This approach can reduce the frequency to take a checkpoint and alleviate the overhead of file IO. We propose to categorize these different migration strategies according to several metrics.
Data to migrate. The data transferred during migration can be user-specified data which implies application-level checkpoint [111], or it can be the whole process images [3, 114, 119] generated by transparent system-level checkpoint like BLCR [57]. In a similar way, AMPI proposes a mechanism based on processor virtualization [44] in which a classic MPI process is represented by a user-level thread that can be migrated when a failure is predicted. With Virtual Machine (VM) migration [58, 86, 101], the entire memory used by the VM is transmitted. However, this last approach comes with the inextricable cost to migrate the complete memory content used by the guest OS.

How the data is transfered from the source to the target. The checkpoint data can be stored as a file in local/shared filesystem [111]. Simple as it is, this approach has the drawback of additional filesystem IO overhead, especially in modern multicore architectures where multiple processes running on the same node generate checkpoint data simultaneously. In such a case the concurrent IO contentions can lead to degraded IO throughput [88, 89]. A more efficient alternative is to convert it into a data stream [108, 114] which is seamlessly transmitted over the network.

Network transport used to move data. The checkpoint data can be transferred using different network transports, such as conventional socket [114] or advance network transports such as RDMA [58].

According to this taxonomy, the new pipelined migration strategy that we presented in this paper corresponds to a process-based migration using data streaming through RDMA transport.
Chapter 7: ATOMIC WRITE WITH SSD

Many applications perform heavy parallel I/O during its execution, which imposes a colossal challenge to the underlying storage system. In addition to the sheer amount of I/O size, a lot of scientific applications conduct large number of small non-contiguous I/O accesses. It’s very difficult for a conventional storage system to deliver high performance while also guarantee the consistency and atomicity for such a non-contiguous access [48, 110].

The advent of Non Volatile Memory (NVM) based storage technique represents unique opportunities to optimize parallel I/O from the fundamentally bottom layer in the storage stack. The dominant NVM technology in use today, NAND Flash [71], show many benefits such as fast random access and low static power consumption, which provides the potentials to meet the demand of extensive non-contiguous access pattern typically seen in many scientific applications. However, NAND flash has performance characteristics that are dissimilar to prior storage media. Flash memory comes with some unique characteristics such as asymmetric read/write latency and low write-durability, therefore special care is needed when utilizing Flash memory as the storage media. To overcome the asymmetric read/write latency and limited write-durability, most Flash-based Solid State Disk (SSD) implements a logical to physical mapping within the device known as a Flash Translation Layer (FTL) [62]. The design of this FTL has direct implications on the performance and durability of the SSD device and significant effort [30, 39, 70, 73, 81] has gone into optimizing the FTL for performance, power, durability, or a combination of these properties.
The FTL layer sits in a unique stratum in the software stack right on top of the device firmware. Therefore optimizations performed in FTL can benefit any software layers above it. In this chapter we explore a new approach that implements atomicity guarantee for non-contiguous access [48, 110] within the FTL layer. We propose a new I/O primitive that batches multiple discrete I/O requests into a logical group which is issued as a single atomic unit with rollback support. This new primitive, called Atomic Write, is embedded into SSD’s Flash Translation Layer to guarantee the atomic completion of such a group request. This work is shown as the highlighted box in Section 7.1 of our proposed research framework.

![Figure 7.1: Atomic-Write Primitive in the Proposed Research Framework](image)

We choose a database management systems (MySQL) as a driving application in need of atomic-write and modify MySQL’s InnoDB storage engine to leverage this new functionality. Using atomic-write we are able to achieve speedups of as much as 33% for the industry standard TPC-C and TPC-H benchmarks. Atomic-write enables a dramatic change in MySQL’s I/O patterns required to implement ACID transaction semantics [56], reducing the need for write-bandwidth by
as much as 43%. In addition to improving performance, reducing unnecessary writes has the secondary effect of doubling device longevity due to wearout, eliminating a major barrier to solid state storage adoption in the enterprise market [87].

This chapter is organized as follows. In Section 7.1 we describe why existing FTL structure isn’t capable enough to provide write atomicity guarantee for non-contiguous write. In Section 7.2 we extend a log-structured FTL to implement atomic-write efficiently. Section 7.3 justifies our approach to embed transactional atomicity guarantee inside FTL layer. In Section 7.4 we choose a Database management system (MySQL) as an example to show how modifications can be made to existing storage engine to to take advantage of atomic-write. Experimental results are described in Section 7.5, showing the efficiency of our atomic-write implementation and how atomic-write affects several industry standard database benchmarks. Related work is covered in Section 7.6.

7.1 SSD Limitations for Atomicity Guarantee

Non-volatile memory (NVM) based Solid State Storage has been deemed as a promising storage technique. The dominant NVM technology in use today, NAND Flash [71], has many technique merits such as fast random access, lower power consumption, shock resistance and small form factor. However NAND flash also has two major drawbacks. First, a flash page cannot be overwritten unless it’s been erased, and an erasure can only be performed at a very large unit (Erase Block) which contains hundreds of flash pages. Second, a flash page can only sustain a limited amount of erase-program cycles (3k-100k), after which it becomes unreliable to store data.

In order to overcome these constraints, most SSD implements a logical to physical mapping layer called Flash Translation Layer (FTL) [62]. At the highest level, the input to the FTL is a logical block address (LBA) and the output is Physical Block Address (PBA) and commands to the NAND-flash media on which the data is stored.
Many of today’s advanced solid state storage devices employ a variation of a log structured file system [98] when implementing their flash translation layer (FTL). In log based designs all writes to the media are sequentially appended to the tail of the log and a separate garbage collection thread is responsible for reclaiming deleted/superseded sectors from the head of the log. Log based designs work well for NAND-flash based devices because the slow erase time of physical blocks is no longer on the critical path for write operations. To implement a log based system, the FTL manages a mapping of logical (LBA) to physical block addresses (PBA). Thus, when a logical block is overwritten, the mapping must be updated to point to the new physical block, and the old block must be marked in the log as available for grooming. The garbage collector will eventually erase the block for re-use. Generally, this mechanism works well to provide efficient read and write access to NAND-flash based devices. However it’s not able to provide an arbitrarily sized atomic-write guarantee for two reasons:

- A write request may contain data that spans multiple contiguous physical blocks within the NAND-flash. Each physical block within NAND-flash must be programmed as a separate unit, thus requiring iterative or parallel programming, which isn’t an atomic operation.

- If multiple sectors are being iteratively written and a system failure occurs, some blocks may be completely written, one block may be partially written, and others will be un-written. The failure recovery process must be able to identify both fully written blocks which should not have been committed, as well as partially written blocks. Incorrect identification of these blocks will result in them being marked as valid within the log, and the superseded data will be erased making future recovery impossible.
7.2 Design of Atomic-Write

We propose to extend FTL to implement a new primitive, atomic-write, that allows multiple I/O operations to be issued as a single atomic unit with rollback support. Specifically we extend the implementation of a log based FTL to support tracking of committed and uncommitted blocks and the necessary crash recovery semantics that utilize this tracking information.

7.2.1 Event Log Tracking of Atomic-Writes

Figure 7.2 provides an example of the tracking methodology we use within the log to identify physical blocks that are part of an atomic-write; we augment the PBA association with a single bit per block to track if any given block is part of an atomic-write operation. Because traditional single block writes are always atomic, this flag is set to “1” for any normal write operation. When the first sector of an atomic-write begins this flag is set to “0”, any subsequent physical blocks written as part of the atomic-write operation are also marked as “0”, until the final block is handled which will again be marked with the flag set to “1”. As a result, the bit tracking fields in the log for an
atomic-write form a sequence that’s very easy to identify. For example, if an atomic-write consists of 3 sectors, then the flag sequence is “001”, as shown in Figure 7.2.

For this implementation it is a requirement that all blocks belonging to an atomic-write are in contiguous locations within the event log. As a result, data blocks from other write requests are not allowed to interleave with atomic-writes. The benefit of this requirement is that any atomic transaction can be easily identified as incomplete if it is not ended by a physical block tagged as “1”. We recognize that serializing atomic-writes in the data stream is undesirable, however in practice we have found that there is very little downside to this design choice since applications using atomic-write semantics recognize that large transactions are costly, and thus try to minimize transaction size. Armed with a method of identifying atomic-writes, we must still guarantee that superseded data is not garbage collected from the log before blocks within an atomic-write have been fully committed, and that upon crash recovery the uncommitted atomic-write blocks are removed from the log.

7.2.2 Delayed Garbage Collection and Crash Recovery

Simply modifying the tracking within the log is not enough to allow rollback to the previous version of data should a write-failure occur. As illustrated in Figure 7.2, the LBA to PBA mapping table must also be aware of atomic-write semantics since this mapping defines what data is valid, discarded, and superseded, making it available for garbage collection. To prevent valid data from being garbage collected before an atomic-write is fully committed, we simply delay updating this range encoded mapping table until the physical data has been committed to the log. By delaying the mapping table update, previous versions of data will never be erased by the garbage collector until a fully committed atomic-write group is on physical media. In the event of a crash recovery in
which the physical blocks were written to disk but the mapping table was not updated, the mapping table can be completely recreated from the log.

During crash-recovery, the log is examined starting at its tail. If the first block contains a “1” then we can safely conclude the storage device was not left in an inconsistent state. If a failure happens in the middle of an atomic-write, we know the log will contain several blocks marked “0” with no “1” preceding it (on the tail). Thus, if the last block written to the log has a “0” flag, we have encountered an incomplete atomic-write. We must then scan backwards, finding all blocks with flag “0” until we encounter the first block with the flag set to “1” which marks the last previous successful completion of either a normal write, or previous atomic-write. All blocks marked with “0” flag must be discarded from the log. Once the tail of the log has been cleaned of any failed atomic-write, a full scan of the log beginning at the head allows us to rebuild the most recent valid data.

Combining the log bitmask, delayed mapping table update/invalidate, and log tail examination upon crash recovery allows us to fully implement atomic-write semantics within our log based FTL.

7.3 Rational of Placement Within The Storage Stack

The concepts we use to implement atomic-write in Sections 7.2.1 and 7.2.2 have been explored in many other contexts [98, 102, 117]. There are also alternative ways one might implement atomic-write within the storage stack. For instance, ZFS [27] provides a strong guarantee that a write to the filesystem is always atomic by using a copy-on-write model. Other filesystems, such as ext2/3/4 allow files to be opened in append-only mode. Append-only allows the filesystem to guarantee that data in the file will never be overwritten or truncated. These files also grow indefinitely, requiring application control to open the file in a different mode (RW) to eliminate old copies of data which
will no longer be referenced. An application could then implement its own tracking of data, much like our log based implementation, to track the most recent copy of data structures written within the file.

The common thread among these high level implementations of atomic-write is that they fundamentally rely on creating multiple copies of on-disk storage, so that previous versions are not over-written. We identify the key insight in this work: Log based Flash Translation Layers already maintain multiple copies of data within the storage device, thus there is no need to duplicate this effort to implement atomic-write at higher levels in the storage stack. As we will show in Section 7.5, by moving atomicity semantics into the log based FTL the amount of data being written to disk decreases substantially and as a result can substantially improve performance for applications relying on transactional semantics.

7.4 Adopting Atomic-Write: an Example with DBMS

Database management systems are one class of applications that typically require strong I/O atomicity guarantees. The atomic guarantee on high level logical pages are implemented by systematic control of logs, buffers, and locks on the underlying storage. In this section, we demonstrate how the popular InnoDB [11] database engine for MySQL [21] can leverage the atomic-write primitive to achieve a performance improvement and simplified implementation, without modifying its ACID compliance.

7.4.1 InnoDB Double-write

InnoDB uses a transaction log to track changes made to a data page. Frequently the dirty pages need to be flushed to stable storage. If a failure happens during a page writing that page will be corrupted and InnoDB is unable to recover such a partial page write.
To overcome the partial-write consistency issues when updating the tablespace, InnoDB utilizes a two phase page update technique known as double-write. Figure 7.3 illustrates the two phases required by double-write to guarantee page consistency.

- In Phase I, InnoDB copies discrete dirty pages from its in memory buffer pool into an additional dedicated in memory buffer area called double-write buffer. This contiguous group of memory pages is then written sequentially and synchronously to a dedicated area within the tablespace file, called the double-write area. If write buffering is being used, a fsync, or flush, is called to force the data through all buffering onto persistent media.

- In Phase II, InnoDB re-writes these same individual dirty data pages to their final locations in the tablespace using synchronous random writes since these pages can be scattered throughout the table space file. If write buffering is being used a fsync is again called to force the data through all buffering onto persistent media.

With this two-phase double-write strategy, InnoDB can guarantee a complete base data page (to which the transaction deltas can be applied) always exists in persistent storage even in the face of a system failure. Should a failure happen that leaves any tablespace data in an inconsistent state, InnoDB will check double-write area, the tablespace area and the transaction log. If a page in the double-write area (Phase I) is found to be partially written, it’s simply discarded since the most recent correct copy still exists in the tablespace. If a page in tablespace is inconsistent, which implies a failure in Phase II, it is recovered using the copy of page in double-write area.

### 7.4.2 Double-write Implications on Storage

Double-write is an effective solution to solving the partial-page write issue but it has significant implications on solid state storage.
• Firstly, double-write imposes an additional write phase (Phase I) that is serialized with the in-place update of tablespace data in Phase II. When working with conventional mechanical disks, Phase I, dominated by sequential-write, is much more efficient compared to random-writes in Phase II. Thus the 100% write overhead only results in a small performance degradation. However, advanced solid state storage can achieve random-write performance very close to the performance of sequential-write, shown in Tables 7.3 and 7.2. Therefore the overhead of this additional write phase is now much more costly in the era of solid state storage.

• Secondly, double-write is named appropriately because it literally writes every data page twice to stable storage. One of the major functions of a FTL layer is to allow re-mapping of LBA to PBA addresses so that wear-leveling can occur transparently to the application. By performing two writes for every singular data page that is intended to persist on media in one location, the double-write approach effectively halves the useful life of a NAND-flash device which is subject to wear-out effects.

Figure 7.3: MySQL Disk Accesses to Guarantee Data Integrity
7.4.3 Replacing Double-write with Atomic-write

InnoDB relies on double-write to protect itself from partial-write of a page (which is made up of multiple physical device blocks). We propose that InnoDB can be modified to replace its complex double-write strategy with the atomic-write primitive described in Section 7.2.

Figure 7.3 shows the natural fit of an atomic-write primitive into MySQL. Rather than performing Phase I of the double-write procedure, pages within the tablespace can be overwritten directly using the atomic-write primitive which guarantees that, this compound update will succeed or fails in entirety. If this atomic-write commits, the transaction delta can simply be removed from the transaction log. If it fails, no explicit recovery is required by InnoDB because the storage subsystem will recover the original pages in place and it will appear to InnoDB that no write to the tablespace ever occurred. By implementing atomic-write within the storage subsystem, we *remove the possibility that partial page writes can occur*.

While the InnoDB recovery process is greatly simplified, there is a substantial performance benefit as well. Atomic-write has replaced a series of serialized sequential and random writes, with a single operation containing half the data payload compared to the original implementation. This reduces the backing store bandwidth required by MySQL by half and doubles the effective wear-out life of the solid state storage device.

7.5 Performance Evaluation

The baseline for all results in this section utilizes an unmodified FusionIO 320GB MLC NAND-flash based device and the most recent production driver available. For this work we implement atomic-write within a research branch of the recently shipped version 2.1 of the FusionIO
driver/firmware [9]. We have extended the InnoDB storage engine for MySQL to leverage atomic-write support as described in section 7.4. All tests are performed on a real machine for which the specification is shown in Table 7.1, none of our results are simulated.

Table 7.1: Experimental Machine Configuration

<table>
<thead>
<tr>
<th>Processor</th>
<th>Xeon X3210 @ 2.13GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM</td>
<td>8GB DDR2 677MHz 4x2GB DIMMs</td>
</tr>
<tr>
<td>Boot Device</td>
<td>250GB SATA-II 3.0Gb/s</td>
</tr>
<tr>
<td>DB Storage Device</td>
<td>FusionIO ioDrive 320GB PCIe 1.0 4x lanes</td>
</tr>
<tr>
<td>Operating System</td>
<td>Ubuntu 9.10 - Linux Kernel 2.6.33</td>
</tr>
</tbody>
</table>

7.5.1 I/O Microbenchmarks

In this section we measure the bandwidth and latency achieved by our atomic-write primitive. **Latency** is the round trip time required for an I/O operation to be durably recorded to storage. **Bandwidth** is the maximum sustained data rate that a storage device can achieve by pipelining commands and maximizing the amount of data transferred per control operation.

7.5.1.1 Write Latency

To evaluate the control overhead required by various I/O methods available in Linux, we test the total time required to perform a compound write which consists of 64x512B blocks to storage (averaged over 100 iterations). For atomic-write (A-Write) we encapsulate all blocks in a single atomic-write request, issue the request to FTL, then wait for its completion. Since atomic-write does not buffer data there is no need to perform a post write buffer flush.

For synchronous I/O, we serialize the block writes by issuing a fsync following each write. For asynchronous I/O, we utilize the Linux native asynchronous I/O library, *libaio*, to submit all
blocks via one I/O request, wait for the operation to complete, and then do a fsync() to flush data to the media if buffering was enabled. Latency is measured from the beginning of the first I/O issued until the completion of all writes, including the fsync if used.

Three different write patterns are tested:

- **Random** - Blocks are randomly scattered within a 1 GB range and aligned to 512B boundaries.
- **Strided** - Blocks start at position \(N\) and are separated by fixed 64KB increments.
- **Sequential** - Blocks are positioned sequentially from position \(N\).

Table 7.2: Write Latency in Microseconds

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Buffering</th>
<th>I/O Type</th>
<th>Sync.</th>
<th>Async.</th>
<th>A-Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>Buffered directIO</td>
<td></td>
<td>4,042</td>
<td>1,112</td>
<td>671</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3,542</td>
<td>851</td>
<td></td>
</tr>
<tr>
<td>Strided</td>
<td>Buffered directIO</td>
<td></td>
<td>4,006</td>
<td>1,146</td>
<td>669</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3,447</td>
<td>857</td>
<td></td>
</tr>
<tr>
<td>Sequential</td>
<td>Buffered directIO</td>
<td></td>
<td>3,955</td>
<td>330</td>
<td>685</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3,402</td>
<td>898</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.2 shows the average latency to complete these writes using different I/O mechanisms. Asynchronous directIO is the fastest traditional storage method because it avoids an unnecessary copy from user space into the operating system page cache before flushing the data back to disk upon fsync. Atomic-write is able to slightly outperform directIO because it natively takes multiple write ranges and avoid the overhead of performing multiple system calls. Sequential buffered+async I/O appears to be an outlier to the general trends - this is because libaio is capable
of detecting and merging multiple contiguous IOPs when using buffered I/O, consolidating the I/O into a single, more efficient, IOP at lower levels in the storage stack. Such an optimization is not possible when using directIO. Atomic-write could make a similar optimization but is beyond the scope of this work.

7.5.1.2 Write Bandwidth

To test achievable bandwidth we utilize the same methodology as in section 7.5.1.1, however we increase the individual block size from 512B to 16KB to maximize the ratio of data transferred per control word and do not require fsyncs after each write, only a single fsync at the completion of buffered I/O.

In Table 7.3, much like Table 7.2, we find that async directIO is able to achieve the highest throughput for traditional storage methods. Again, this is due to being able to pipeline I/O operations and not performing an extraneous copy into the page cache before flushing the data to media. Atomic-write appears to slightly outperform directIO in all cases, but the difference is simply in implementation, atomic-write should have no fundamental performance advantage over asynchronous directIO. It is worth noting that MySQL does not natively use async directIO. Instead, it chooses to use synchronous directIO but implement its own I/O offload thread which approximates the behavior of asynchronous I/O while providing better control over I/O re-ordering for ACID compliance.

7.5.2 Database Workloads

The microbenchmark results in Section 7.5.1 show that atomic-write can be implemented within a log based FTL and provide performance that meets or exceeds that of the legacy I/O interfaces. To test the real world impact that atomic-write can have on application performance, we evaluate two industry standard transaction processing workloads DBT-2 [24] and DBT-3 [20]
which are fair use implementations of TPC-C and TPC-H respectively, and *SysBench* [26], which is another transaction-processing benchmark. The performance metrics we evaluate are: *Transaction Throughput* which is the number of transactions completed per unit time; *Data Written* which is amount of data written to storage during workload execution; and *Latency* which is the average time required for a single transaction to complete.

For this work we configure MySQL to run on a non-shared machine seen in Table 7.1. InnoDB’s buffer pool is set at 4GB and both its transactional log and tablespace files co-exist on the FusionIO device. MySQL’s binary log is stored to a separate hard disk drive and is not on the critical path for performance. DBT-2 is configured to use 500 warehouses with a resulting database size of 47GB including indices. DBT-3 is configured with scale factor of 3 resulting in a total database size of 8.5 GB with indices. SysBench uses a table of size 11GB with 20 million tuples in it. The driver for each workload was run on a separate machine connected by 1Gbps Ethernet to avoid polluting the database host system. Each benchmark was run for a minimum of 10 minutes and warmed up once before collecting statistics.

Performance is measured for three distinct test cases:
MySQL - The unmodified InnoDB engine with double-write enabled. This mode provides full ACID compliance, but shows the performance penalty incurred on a SSS device by having to perform twice the number of writes. This mode is used as the baseline in all results.

Double-Write Disabled - InnoDB with Phase I of the double-write simply disabled. This is an unsafe mode that may suffer from the “partial-write” problem, but highlights the potential gains of eliminating Phase I of InnoDB’s ACID compliant implementation.

Atomic-Write - InnoDB optimized to use atomic-write as described in Section 7.4. Using atomic-write provides the same level of ACID compliance as the baseline InnoDB implementation.

7.5.2.1 Transaction Throughput

Figure 7.4 shows the transaction throughput of our three test cases normalized to the baseline InnoDB implementation. Simply disabling phase I of the InnoDB double-write implementation results in a maximum throughput improvement of 9%. Atomic-write is able to outperform the baseline InnoDB implementation by as much as 23%. Both Atomic-write and double-write-disabled write the same amount of data to storage, but InnoDB offloads tablespace writing to an I/O thread which in turn performs synchronous writes. As seen in Table 7.3, our atomic-write implementation is able to sustain 142% more bandwidth than the native synchronous directIO methodology used by MySQL. As a result, utilizing atomic-write can further improve throughput over simply disabling double-write.

The throughput improvement achievable by using atomic-write within InnoDB is fundamentally limited by the amount of time spent waiting on write I/O within the workload. There are
two factors that affect this wait time: the percentage of read vs. write operations that the workload natively requests - and the amount of memory in use by the database.

7.5.2.2 Amount of Data Written to Storage

InnoDB writes both its transaction log and tablespace data to stable storage. Using atomic-write, we are able to optimize the tablespace data storage process reducing the total writes by one half, but the amount of data written to the transaction log is unaffected by either disabling double-write or leveraging atomic-write. Figure 7.5 shows the relative amount of data written to the underlying storage during workload execution. Disabling double-write from MySQL reduces the total data written to the backing store by up to 46%, while atomic-write reduces total data written by up to 43%. Because atomic-write can process more transactions and generate more write requests during the fixed time execution of the benchmarks, it has slightly higher write-bandwidth. On a per transaction basis, double-write-disabled and atomic-write require the same amount of total I/O.
In our experimental configuration, each database workload was run in isolation on a single high throughput (>500MB/s) solid state storage device. In enterprise installations the storage subsystem is often shared between one or many applications in a NAS or SAN environment. In these situations, storage subsystem bandwidth is often the single largest bottleneck in database performance; by reducing the write rate to a database by 43%, we also help extend the value of shared storage and network infrastructure. A by-product of reducing the number of writes to storage is that for solid state devices, the useable life of the device is almost doubled. Device wearout has been a major barrier to enterprise adoption, so this significant improvement should not be overlooked.

### 7.5.2.3 Transaction Latency

Another important metric in many database driven applications is the average response time per query or transaction. Transaction latency is dominated more by the synchronous write to the transaction log, but write bandwidth also plays an important role. In a memory constraint environment, the database has to frequently flush out dirty data pages to make room for newly accessed
pages. This effectively serializes transaction processing with table space writes when they are occurring. Full database checkpointing, which is convenient for crash recovery, effectively blocks all transactions until the tablespace write has finished. Thus, by reducing the amount of data that must be written to storage in both these cases, atomic-write helps decrease both the variation and average latency of transactions. For DBT2 and SysBench, we show the 90th percentile latency in Figure 7.6. For DBT3 we show the average latency of the queries performed during the execution. Atomic-write is able to reduce 90th percentile latency of DBT2 and SysBench by 20% and 24% respectively. The average latency in DBT3 is reduced by 9%. Many improvements in database throughput often come at the expense of transaction latency. For many interactive database workloads, such as Web 2.0 sites, maintaining a worst case latency is extremely important for usability of the system. Improving both throughput and transaction latency makes atomic-write an ideal database optimization for these types of systems.

Figure 7.6: Transaction Latency (Lower is Better)
7.6 Related Work

Flash translation layers have received significant study because the LBA to PBA mapping layer is on the critical path for both read and write operations. There have been several efforts to compare the efficiency of block mapping versus page mapping FTL designs [30, 39, 70, 73]. Lim et al. [81] specifically try to improve the performance of a block mapping system to that of a page mapping scheme without requisite memory overhead. Shim et al. [106] attempt to partition the on-board DRAM cache between mapping and data buffering to optimize performance. Seppanen et al. [105] focus on how to optimize the operating system to maximize performance of solid state devices. Our work differs from these in that we are providing a new primitive and additional functionality, not just optimizing performance within the existing paradigm.

Choi et al. [61] and Josephson et al. [69] have both investigated how filesystems might be able to integrate more closely with log based flash translation layers. While closest to our work, both of these require integrating with functionality within the FTL that is not exported for general use. The atomic-write primitive proposed in this work could be leveraged by both studies to help decouple themselves from the FTL. Filesystems such as ZFS [27] and EXT3cow [91] implement a copy-on-write technique to preserve data atomicity which is functionally similar to InnoDB’s double-write methodology. Seltzer et al. [102–104] describe how to support atomicity and transactions within a log structured filesystem. All these studies assume that the basic atomic primitive provided by the lowest level of storage is a single fixed 512B block. We differ from these works by showing that it is fundamentally more efficient to support multiple block atomicity in the FTL than build atomicity guarantees at higher levels within the storage stack.
Chapter 8: SSD-ASSISTED HYBRID MEMORY TO ACCELERATE OBJECT CACHING

Many Datacenter applications depend on large RAM size to cache a huge quantity of frequently accessed objects (indexes, lookup tables, key-value pairs, etc.) in RAM to deliver high performance. These applications range from social networks, financial real-time trading, online gaming and storage de-duplication [47, 90].

In this chapter we propose a SSD-Assisted Hybrid Memory that augments RAM with SSD to achieve a huge program memory size to be used by an object-caching software. This Hybrid Memory design is shown as the highlighted box in Figure 8.1 as our proposed framework. Unlike virtual memory swap system that manipulates memory and SSD at page level, we manage resource allocation at object granularity. SSD is used to store all objects, with a small amount of RAM to cache recently accessed objects in a LRU manner. An in-memory lookup table transparently maps a given key to the objects location in SSD and to its RAM address if cached in RAM. The SSD is organized as a log-structured sequence of blocks to overcome SSD limitations such as asymmetric read/write latency and limited write endurance. We have integrated Hybrid memory into Memcached as its memory allocator.

This chapter is organized as follows. Section 8.1 gives the motivation of our work. In Section 8.2 we discuss possible ways to leverage SSD in Memcached. In Section 8.3.2 we propose the hybrid memory design and discuss how to integrate it into Memcached. An analytical model
Figure 8.1: SSD-Assisted Hybrid Memory in the Proposed Research Framework

is presented in Section 8.4 to derive possible performance gains. In section 8.5, we present our experiments and evaluation. Related work is discussed in Section 8.6,

8.1 Motivation

Object-caching layer is very important for datacenter application performance. A very common software in object-caching layer is Memcached [15], a distributed-memory object-caching system. Memcached stores the key-value pairs of database queries, API calls or any other data in the aggregated memory pool formed by participating servers, such that later accesses can be satisfied by reading the objects from memory at the corresponding server without expensive accesses to secondary storage.

Aggregated memory pool size has a direct impact on Memcached performance. In datacenters, the server workload is highly volatile [112] and requires a very large memory pool to cache the recently accessed objects. If an object is not found in the memory pool, a price has to be paid to fetch it from the backend database storage which is orders of magnitude slower than the in-memory
fetch. Due to constraints of hardware cost, power/thermal concerns and floor plan limits [82], it is becoming difficult to further scale the memory pool size by packing more RAM into individual servers, or by expanding the server arrays.

![Figure 8.2: Memcached Get Latency at 1KB Object Size. “IB-RAM” is Memcached Running on Native InfiniBand with Data in RAM. “IB-VirtualMem” is Similar to “IB-RAM”, but Data is Stored at Virtual Memory Mapped SSD. The SSD is preloaded with 30 GB Data. “SSD-Read” Gives the SSD Read Latency. The SSD used is Fusion-io ioDrive 80 GB SLC.](image)

NAND-flash based Solid State Disk (SSD) has attracted a lot of attention recently as an alternative storage device [76]. There have been proposals [6] to expand effective memory size by mapping SSD into the virtual memory. These approaches effectively treat SSD as a swap device, and allow applications to transparently access a huge virtual memory space that is managed by the operating system. However, these approaches suffer from heavy overhead at virtual memory management layer [72, 84], which manipulates SSD at memory-page granularity. Conventional virtual memory mechanisms are highly tuned toward mechanical hard drives, and they do not fit well with SSD that behaves rather differently [84]. Additionally, they cause excessive amount of write traffic to SSD that undermines SSD lifetime [46, 118].
Simply mapping SSD into virtual memory can lead to substantial performance loss, as is demonstrated in our evaluation shown in Figure 8.2. In this test, we perform query to a Memcached server via InfiniBand transport, and let the Memcached server store the data either in RAM (“IB-RAM”) or in a chunk of virtual memory mapped by SSD (“IB-VirtualMem”). As shown with the label “IB-RAM”, we can fetch a 1KB object from Memcached RAM in 10 µs. This latency rises to 347 µs if data is stored in SSD-mapped virtual memory, as marked by “IB-VirtualMem”. Given that SSD random read latency is 68 µs, the majority of performance loss is attributed to the additional overhead at virtual memory management [46] to treat SSD as a swap device. Our finding is consistent with what has been reported by the authors in [72, 84].

we explore approaches to expand available memory size with SSD that can fully capture the performance potentials of SSD. We want to address several challenges:

1. How can SSDs be used in an efficient manner to expand available memory size and in the meantime to overcome the inherent constraints imposed by SSD, such as asymmetric read/write latency, and limited write durability?

2. How can such an envisioned design be integrated with Memcached to unleash the potentials of both SSD and high bandwidth low latency networks to improve Memcached performance?

3. Can an analytical model be developed that not only matches with experimental results, but also can predict performance trends of designing Memcached servers with next generation networks and SSDs?

8.2 Design Alternatives to Extend Memcached with SSD

In this section we discuss possible approaches to exploit SSD to expand available memory size for Memcached.
Figure 8.3: Existing Memcached Deployment (Basic)

Figure 8.3 illustrates a typical Memcached deployment in datacenter environment to accelerate dynamic web applications [90]. A Memcached client runs on the Proxy Server which receives requests from the Internet, and tries to get the desired objects from the corresponding Memcached server based on the keys (Step 1). If such objects are not found in the memory pool, the client queries the database server to find them (Step 2). Those objects are then stored to Memcached server (Step 3) for future usage and the replies transmitted to Internet in the meanwhile. This diagram indicates that Memcached performance is dependent on several factors, including the network latency for the components to communicate between each other, the in-RAM hit ratio at Memcached server, and database query latency.

Datacenters usually contain gigantic amount of data, and the volatile server workload requires to cache huge amount of data that is even bigger than the aggregated main memory size on Memcached servers. In such scenarios Memcached may not yield a high in-RAM hit ratio. Every miss comes with the price to perform an expensive database query to retrieve the data. We have measured the database query latency using MySQL in contrast with Memcached latency over different
Table 8.1: Latency of Memcached and MySQL with Different Network (1KB Record Size). SSD Latency is Measured with Fusion-Io ioDrive [7] 80GB at 1 KB Access Size.

<table>
<thead>
<tr>
<th></th>
<th>InfiniBand-Verbs</th>
<th>InfiniBand-IPoIB</th>
<th>10 GigE</th>
<th>1 GigE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL Query Latency (µs)</td>
<td>N/A</td>
<td>10763</td>
<td>10724</td>
<td>11220</td>
</tr>
<tr>
<td>Memcached Get Latency(in RAM) (µs)</td>
<td>10</td>
<td>60</td>
<td>40</td>
<td>150</td>
</tr>
<tr>
<td>Memcached Get Latency(in SSD mapped Virtual Memory) (µs)</td>
<td>347</td>
<td>387</td>
<td>362</td>
<td>455</td>
</tr>
<tr>
<td>SSD Latency (µs)</td>
<td>Random Read</td>
<td>Random Write</td>
<td>103</td>
<td></td>
</tr>
</tbody>
</table>

networks. The results are shown in Table 8.1. In this evaluation the database contains 40 million records of 1 KB size (40 GB data in total) in a 250 GB ST3250310NS 7200 RPM hard drive, and the MySQL server uses 3 GB memory as its buffer pool. A database client performs random queries of the records over different networks and measures the latency. As is shown in the figure, a database query is orders of magnitude slower than an Memcached access.

It is virtually impractical to significantly reduce database query latency at a low cost. One can install an SSD in the database server as a fast disk cache. This may reduce the cost of database query, but additional network round trips are still needed. Another possible solution is to install more RAM at Memcached servers to cache more data. However the RAM cost rises dramatically at large sizes [97]. Power consumption and thermal concerns are the additional issues [82] that prevent packing a lot of RAM modules into the servers. One can add more servers to increase the total RAM size, but the cost rises proportionally and more servers also means more power consumption and higher floor plan usage.

The left part of Figure 8.4 depicts a straightforward way to exploit SSD as a swap device to enlarge the virtual memory (VMS). However, this approach incurs substantial overhead as is shown
in Table 8.1. With native InfiniBand as transport, for example, a Memcached get operation costs 10 μs if data is stored in RAM. This number becomes 347 μs if data is stored in SSD-mapped virtual memory, even if SSD read latency is 68 μs. This radical growth in query latency can be attributed to the inefficient virtual memory management, as has been studied in previous literatures [46, 72]. The operating system operates virtual memory (VM) at page granularity, and every object read/write will cause an entire flash page to be read/written no matter how small the object is [65, 84]. This not only wastes the SSD IO bandwidth, but also results in excessive write access to SSD which leads to premature wearing out of SSD devices [118].

Figure 8.4: Possible Approaches to Exploit SSD in Memcached

We propose to augment RAM with SSD to build an efficient memory stack to store key-value pairs. This structure, called Hybrid Memory, can be plugged into Memcached to expand the effective memory size to applications. This architecture is illustrated in the right part of Figure 8.4. Hybrid memory is designed with SSD characteristics and Memcached requirements in mind. With
such an approach we effectively surmount SSD limitations and require very little changes in Memcached to accommodate hybrid memory. The detailed design is presented in the next section.

8.3 SSD-Assisted Hybrid Memory Design

In this section we describe the design of *SSD-Assisted Hybrid Memory*, which combines RAM and SSD into a unified memory stack for fast allocation and access. We first identify several challenges to overcome in the hybrid memory design. Then we elucidate the key components of hybrid memory. We also discuss how Memcached can be extended to leverage the hybrid memory.

8.3.1 Design Challenges

We aim at a SSD-based object store that can efficiently handle random read/write of large quantity of varied-sized key-value objects. Flash memory cannot deal with this type of operations efficiently for two reasons. (1) Small random writes cause update to portions within a flash page, and in order to avoid the expensive erasure operation (a constraint of flash memory), the valid data in the original flash page needs to be read out and merged with the new data before writing to a new flash page [66]. This compound of read-merge-write wastes the SSD IO bandwidth. (2) Frequent read-merge-write cycles cause unnecessarily large amount write accesses, which generates additional wearing on flash pages and reduces SSD lifespan [118]. In addition to the constraints due to SSD characteristics, varied-sized objects also lead to wastage in RAM usage because: (1) Mixing objects of different sizes in a RAM page causes memory fragmentation which wastes RAM resource. (2) Each in-RAM index entry may have to store the size of the associated object.

8.3.2 SSD-Assisted Hybrid Memory

We have proposed and designed the SSD-Assisted Hybrid Memory to address these challenges. As illustrated in Figure 8.5, the hybrid memory consists of three major components: a *Read/Write*
Buffer to cache recently accessed objects which also copes with varied-sized objects; an In-memory Lookup Table that stores the index of all key-value pairs in hybrid memory; and a Log-Structured SSD Storage which is the persistent location for all key-value pairs.

![Diagram of SSD-Assisted Hybrid Memory Architecture](image)

**Figure 8.5: SSD-Assisted Hybrid Memory Architecture**

1. **Slab-Structured Read/Write Buffer.**

   The hybrid memory caches newly-inserted and recently accessed objects in Read/Write Buffer in LRU manner to accelerate reuse in near future. This buffer consists of multiple memory slabs each composed of a list of chunks (shown as a bold box in Figure 8.5). The chunk size can be adjusted and we use 1 MB chunk size in our experiments. Each slab is dedicated to store objects of a predetermined size, such as 64 bytes to 4 KB. An object is associated with a unique slab that best matches its size. Objects belonging to a given slab are packed into memory chunks for that slab. This dense format improves RAM efficiency to hold more objects with a given RAM size, while it also provides the flexibility to handle varied-sized objects. It achieves a proper trade-off between RAM efficiency and flexibility. When the total RAM usage of the slab exceeds a threshold, object eviction is performed for a slab by coalescing dirty objects into a temporary buffer chunk and writing that buffer chunk to SSD. Details of eviction are discussed below.
(2) Lookup Table

The lookup table contains one index entry per object stored in the hybrid memory. Each index entry points to the SSD and/or Read/Write Buffer locations where the object is stored, together with a flag bit to indicate the object status. It also contains LRU pointers to organize all objects into a LRU list which is used during object eviction.

When a new object is created ("New"), it is cached in the Read/Write Buffer and deemed dirty. A new index entry is allocated for this object and points to its in-RAM location. When an object is to be accessed, a search operation hashes the given key into the lookup table and finds the corresponding index entry. If this object is cached in Read/Write Buffer, it can be directly accessed by following the pointer in its index entry. If the object is not in Read/Write Buffer, it is loaded into the Read/Write Buffer and its index entry is updated with the new RAM address. At this time this object is in “Active” state and is regarded clean. Overwriting an existing object makes it dirty.

When the Read/Write Buffer becomes full, the cached objects in “dirty” status are compacted and evicted into SSD. This process is performed in a per slab basis. Since all index entries form a LRU list, we scan this LRU list from the tail and retire those objects cached in Read/Write Buffer. Clean objects are discarded because they have an up to date in-SSD copy, and dirty objects are compacted into a coalescing buffer (1 MB size). This buffer is written to SSD when it is full. Once the writing finishes, the corresponding memory chunks in Read/Write Buffer are released to be reused. This batched writing of multiple objects overcomes the write anomaly of SSD [76]. In the meantime these objects’ index entries are updated to reflect the objects’ new locations in SSD, and their Read/Write Buffer pointers are cleared. Those retired objects are in “Evicted” state. When an object is to be deleted, its index entry and the copy in Read/Write Buffer is released, but its in-SSD copy is left over for lazy reclamation (in “Invalid” state), where special considerations are required. This is covered in the next section.
RAM usage is an important aspect in the Lookup Table design. Our initial design stores the full key in the index entry for quick search. We have designed a dynamic memory management scheme that allocates memory slices for the literal keys only as much as is actually needed. We choose this structure of the index entry for several reasons: (1) This format matches very well with the index format used by Memcached, which greatly smooths the integration. (2) At present it is common that a server is equipped with large RAM from 16 GB to 64 GB. Assuming 64 GB RAM, 20 bytes key and 1 KB object size, we can easily store over 1 billion index entries which represents 1 TB data in SSD. As part of the next step, we are investigating a more memory-frugal design that stores only the fingerprint of key name in the index entry [38].

(3) Log Structured SSD Storage

The SSD storage is organized into a Log-Structured sequence of blocks [104]. Upon eviction from the Read/Write Buffer, dirty objects are aggregated into a temporary buffer chunk and appended to the log tail. This append-only mode leverages SSD’s efficient sequential write at large size to amortize the writing cost over many smaller objects. A timestamp and backward pointer to the index entry are embedded into each in-SSD object. These elements are used during garbage collection.

(4) Garbage Collection (GC)

Our append-only design leaves old object copies in SSD when an object is deleted or overwritten to avoid the expensive in-place update in SSD [74]. A garbage collector is needed to reclaim these obsolete objects in the background to release SSD space for incoming writes. There is a straightforward approach [35] that scans the SSD from beginning of the block sequence (log head), skips dead objects and merges the live objects into a temporary buffer chunk, then appends this chunk to log tail. After appending, the log head is ready to be erased and becomes free. This
Basic Scan and Compact (BSC) approach has some drawbacks. If the dead objects are not concentrated at the starting area, BSC ends up reading a lot of live objects from many SSD blocks and rewriting them to the log tail. This leads to unnecessary read/write access to SSD. In order to overcome this blind scan-and-compact behavior, we propose a Freshness based Scan and Compact (FSC) algorithm. We maintain a freshness index for each used SSD block to mark how much portion of a block is occupied by live objects. Upon eviction, we append a data chunk to a fresh SSD block at log tail, and set this block’s freshness index to 1. During the eviction scan, we examine each evicted object’s in-SSD location, and decrease the corresponding SSD block’s freshness accordingly. When GC starts, it checks the SSD blocks’ freshness and always scans the block with the most dead objects data, so as to minimize the amount of live data that needs to be written to SSD. Therefore it can reduce the total write volume to reclaim certain amount of SSD blocks. We will demonstrate FSC performance in the next section.

8.3.3 Integrate Hybrid Memory into Memcached

The hybrid memory is designed as an efficient object allocator, and it fits very well with the memory-object model used in Memcached. Memcached defines an item to encapsulate a key-value object as a whole in a piece of memory. Memcached item header shares many common fields with hybrid memory index entry. We slightly extend the Memcached item definition to accommodate the index entry needed by hybrid memory, and replace its memory manager with hybrid memory. With minimum amount of changes we are able to adopt hybrid memory into Memcached.

8.4 Analytical Model of Memcached

We design an analytical model to estimate the Memcached access latencies with Basic (Figure 8.3) and Hybrid (Figure 8.4) approaches, respectively, in order to better understand the possible
performance gains. The concept is similar to the one used in memory hierarchy design [67]. Assume network round trip time is $t$, database query latency is $q$, and the in-RAM hit ratio is $h_1$, in-SSD hit ratio is $h_2$, SSD read/write latency to be $r$ and $w$. The operations on Memcached server are mainly to manipulate objects in memory which are drastically faster than network traffic or database query, therefore we will ignore them in the calculation.

In the Basic approach, the latency of a search seen by the Memcached client is:

$$L_b = h_1 t + (1 - h_1)(t + q + t)$$

$$= (1 - h_1)q + 2t - h_1 t$$

Referring to Table 8.1, we assume the following values for the parameters: $h_1=0.2$, $q=11,000 \mu s$, and $t=10 \mu s$. This leads to $L_b=8,818 \mu s$.

In the Hybrid approach, the latency of a query is:

$$L_h = h_1 t + (1 - h_1)[h_2(t + r) + (1 - h_2)(t + q + t)]$$

$$= (1 - h_2)(q + t) + t + h_2 r$$

With Hybrid approach we assume $h_1$ is small enough to be ignored since hybrid memory can tolerate a small Read/Write Buffer size that is very tiny compared to the working data set, and the server-style workload we are targeting at has very little data locality. SSD size can be big enough to yield a larger hit ratio, so we let $h_2=0.8$, SSD access latency $r=68 \mu s$, $w=70 \mu s$, $q=11,000 \mu s$, and $t=10 \mu s$. This leads to $L_h = 2,266 \mu s$, a 3.9X speedup over the Basic approach.

Although a rough estimation, this model matches well with the real experimental results as presented in Section 8.5.7. This model reveals the potential benefit of Hybrid memory to diminish object query latency which comes from two aspects: (1) SSD is much larger than RAM in terms of
size which results in a higher hit ratio at Memcached server; (2) SSD fast random access property doesn’t impose significant overhead when accessing data. In Section 8.5.7 we use this model to predict the possible latency as technology progress changes the value of different components.

8.5 Performance Evaluation

We have designed the SSD-Assisted Hybrid Memory, and have integrated it into Memcached-1.4.5. In this section we conduct experiments to evaluate hybrid memory performance from three perspectives including:

(1) Microbenchmark level testings to measure hybrid memory’s basic performance including random read/write latencies, operation throughput, effectiveness to reduce read/write traffic to SSD, and Garbage Collection performance.

(2) Evaluate Memcached performance that stores its key-value pairs at hybrid memory. Different networks (IB-Verbs, IB-IPoIB, 10GigE, and 1GigE) are used to measure their relative impacts.

(3) Evaluation in a data-center environment with huge dataset stored in a database server, and Memcached workload generated with a zipf distribution.

Figure 8.6: Hybrid Memory Basic Operation Latency. Have Preloaded 30 GB data, Hybrid Memory Buffer Size=256 MB.
8.5.1 Experimental Setup

We use a Linux cluster in the experiments. Each node in the cluster has eight processor cores on two Intel Xeon 2.33 GHz Quad-core CPUs, 6 GB Main memory, and a 250 GB ST3250310NS Hard drive. Each node is equipped with a Mellanox MT25208 DDR (16 Gbps) HCA, 10GigE adapter by Chelsio Communications (T320) and the standard 1GigE adapter. TCP offload engine is enabled for the 10GigE adapter. We integrated the proposed hybrid memory as the memory allocator into Memcached-1.4.5 which has been extended to support both InfiniBand [68] and socket transports (10GigE and 1GigE). A Fusion-io ioDrive [7] 80GB SSD is plugged into the Memcached server as part of the SSD-assisted hybrid memory. Hybrid memory Read/Write Buffer is fixed to be 256 MB, if not particularly mentioned.

8.5.2 Hybrid Memory: Basic Performance

We first measure the basic performance to read/write/create objects of varied sizes using hybrid memory as a stand-alone module. In this microbenchmark we pre-create objects with sizes ranging from 256 bytes to 4 KB into hybrid memory so that it contains 30 GB data in SSD, which is much bigger than the 6 GB RAM in a node. The Read/Write Buffer is fixed to be 256 MB so that in-RAM hit ratio is very low for random accesses. Then we read/write/create random objects of certain sizes with hybrid memory and measure the average latency. This corresponds to the Hybrid approach in Figure 8.4. As a contrast we also evaluate an alternative approach where SSD is mapped to a huge virtual memory space that is much bigger than RAM size, and we create/read/write objects into this virtual memory. This alternative represents the VMS approach in Figure 8.4. We perform the same initialization with VMS approach so that it contains 30 GB data in SSD, and repeat the same testing to see the read/write/create latency.
8.5.2.1 Random Read Latency

Figure 8.6(a) gives the average latency of random read at different object sizes. Hybrid consistently outperforms VMS approach by large margin. At 1KB object size VMS needs 271 µs to find an object and load it to RAM, while Hybrid can load an object to RAM in 76 µs, a 3.6X speedup. Hybrid can read an 4KB object in 95 µs in contrast to 358 µs by VMS, which is a 3.8X improvement. The OS manages virtual memory at page granularity. When desired data is not found in memory, the OS loads multiple pages from SSD to page cache and then copies the desired data to user. The excessive data reading and memory copy is not beneficial for the random access workload in question. Hybrid approach, on the contrary, loads the exact portion of data that is really needed and avoids the excessive reading from SSD and memory copy overhead.

8.5.2.2 Random Write Latency

Figure 8.6(b) shows the average latency to overwrite an random existing object in hybrid memory. Hybrid greatly reduces the latency to write an object. It outperforms VMS by 3.2X to 3.4X for object sizes ranging from 256 bytes to 4 KB. With VMS, writing to a random object that is not in memory causes read accesses to load multiple consecutive pages from SSD that contains the desired object. During subsequent writing which needs clean memory pages to hold data, the dirty pages containing modified data have to be flushed to SSD. This page-granularity resource management conducted by the OS is transparent to users, but it unnecessarily wastes SSD bandwidth. On the contrary, Hybrid only loads exact amount of data that is needed during read phase, and it coalesces multiple dirty objects for eviction during writing. Therefore it substantially improves the IO efficiency.
8.5.2.3 Create New Object Latency

Figure 8.6(c) shows the average latency to insert a new object into hybrid memory. *Hybrid* packs dirty objects at Read/Write Buffer into big chunks (1 MB chunk) and flushes a whole chunk to SSD, so the writing cost is amortized across many objects. *VMS* manages the memory object at page granularity. At small object size, a page can hold multiple objects so the writing cost can be averaged. As object becomes larger, it is more and more possible that each single new object results in a whole page write to SSD. Therefore *Hybrid* outperforms *VMS* by 1.1X to 2.6X for object sizes ranging from 256 bytes to 4 KB.

![Graph](image)

Figure 8.7: Hybrid Memory: Random Read Latency at Different in-RAM Hit Ratio. Have Preloaded 15 GB Data, Object Size=1KB.

8.5.2.4 Random Read Latency at Different in-RAM Hit Ratio

We have also measured random read latency at different in-RAM hit ratio using hybrid memory. We populate hybrid memory with 15 GB data with object size=1 KB. Then we vary the buffer size for random read to achieve different in-RAM hit ratio. The maximum hit ratio we can get is about 30% given the 6 GB main memory. Figure 8.7 shows the results. At very low in-RAM hit ratio the
random read latency is largely bounded by SSD random access latency. As buffer size increases and hit ratio grows, more requests end up finding the target data in RAM, so the average access latency drops.

Figure 8.8: Hybrid Memory: Operation Throughput. Have Preloaded 30 GB Data, Object Size=1KB, Read/Write Buffer=256 MB.

### 8.5.3 Hybrid Memory: Operation Throughput

In this test we run multiple processes to perform concurrent operations to hybrid memory. The hybrid memory has been populated with 30 GB data (object size=1 KB). Each operation either reads or modifies a random object of 1 KB size. We start 1/2/4/8/16 processes and vary the mixture ratio of read and write accesses in total operations. The total operation throughput is reported in Figure 8.8.

Hybrid memory attains a steady increase in total operation throughput with increased number of processes, and its throughput is much higher than what can be achieved by VMS. With 16 processes and read-only workload, hybrid memory reaches a throughput of 75,000 operations/second.
which is 15.3X of what can be obtained by VMS. The results in Figure 8.8 clearly demonstrates the capability of hybrid memory to fully exploit the SSD potentials.

It is observed that, the read-only workload with hybrid memory has a slightly higher throughput than those mixed with read and write accesses. Given the stringent Read/Write Buffer size (256 MB) in this configuration in contrast to the huge amount of working set data (30 GB), a write operation almost always results in a read access to load the target data object into buffer before the data is modified and become dirty. Soon this dirty object is evicted to SSD in order to make space for subsequent operations. On the contrary, an object that has been loaded by a read operation can be discarded without incurring any writing cost because it is still clean. However, our hybrid memory amortizes the writing cost by batching multiple dirty objects into a big chunk. As a result we only observe a minimum drop in the write throughput in contrast to read throughput.

![Figure 8.9: Actual Read/Write Volume to SSD, Normalized to VMS. 50% Read, 50% Write. Have Preloaded 30 GB Data, Object Size=1KB.](image)

8.5.4 Hybrid Memory: Read/Write Volume to SSD

By managing resources at object granularity, Hybrid avoids the wastage of SSD IO bandwidth. In this test we measure the actual read/write traffic to SSD for a given workload using either Hybrid.
or VMS. The SSD is preloaded with 30 million objects of 1 KB size each (30 GB data in total). Then we perform a 50%-read-50%-write workload that accesses random objects 10 million times. We use “blktrace” to measure the total amount of read/write traffic directed to SSD. The results are reported in Figure 8.9. The traffic volume is normalized to VMS case. It clearly demonstrates the advantages of Hybrid to substantially reduce the IO traffic. Read volume is reduced by 97%, and write volume by 81%. Because of SSD write durability issue, 81% less write traffic represents a 5.3X improvement in SSD lifespan. This drastic improvement is obtained by bypassing the inefficient virtual memory stack, which accesses whole pages upon a miss in addition to aggressive read-ahead to load excessive amount of data.

Figure 8.10: Hybrid Memory Read/Write Throughput with GC in Background. 1 Reader, 1 Writer.
8.5.5 Hybrid Memory: Garbage Collector with FSC

As described in Section 8.3.2, Garbage Collector (GC) is an important component in hybrid memory. When GC is activated, it competes with other tasks for SSD IO bandwidth, hence it may lead to performance degradation.

In this test we run a workload that resembles a production deployment to understand the GC implications. We confine the available SSD size to 10 GB, and preload it with 5 GB data in 1 KB object size. A worker thread runs constantly to perform random read, and a concurrent thread keeps overwriting all objects in a randomly picked SSD block, causing the SSD space usage to grow. When the usage ratio hits 60%, a GC task kicks in and it runs FSC algorithm (as described in Section 8.3.2) to reclaim the obsolete data till the SSD usage ratio drops to 50%. We measure the worker thread’s read throughput over a period of 10 minutes. The results are shown in Figure 8.10. For most of the time the GC task is dormant and the throughput remains relatively steady. At some intervals we observe radical fluctuation. These anomalies are caused by GC which aggressively consumes the SSD bandwidth to reclaim free space. However FSC uses SSD bandwidth efficiently, and it induces only very brief period of degradation. The average throughput over this 10 minutes period reaches 13,500 operations/second. The same experiment without GC achieves a throughput of 14,000 operations/second (as shown in Figure 8.8, 1 thread read-only). FSC algorithm can effectively minimize the IO interference on user workload, therefore it only causes 3.6% performance degradation.

8.5.6 Memcached Performance

We have extended Memcached-1.4.5 to use SSD-Assisted Hybrid Memory as its memory allocator. The same changes done in [68] have been incorporated so that our Memcached can support both socket and IB-Verbs as its transport. We start Memcached server in one node and Memcached
Different networks (IB-Verbs, IB-IPoIB, 10GigE and 1GigE) are used to evaluate the impact caused by network transport. Section 8.5.2 has demonstrated the advantages of hybrid memory over the straightforward approach of treating SSD as a swap device. Therefore in this section we only report the results using hybrid memory as the memory manager at Memcached server due to space constraints.

Figure 8.11: Memcached Performance with Hybrid Memory. Have Preloaded 30 GB data, Hybrid Memory Buffer Size=256 MB.
8.5.6.1 Memcached Latency

In this testing we let a Memcached client perform \textit{Get} (read) and \textit{Set} (insert) operations to Memcached server and measure the average latency. The \textit{Get} operations read object from hybrid memory, and \textit{Put} operations insert new objects into hybrid memory in the Memcached server. The Memcached server has been initialized with 30 GB data in its hybrid memory at mixed object sizes from 256 bytes to 4 KB. Hybrid memory’s read/write buffer is set to be 256 MB. Figure 8.11(a) and Figure 8.11(b) present Memcached get and set latencies, respectively, at varied object sizes when different network transports are used to connect the Memcached client and server.

Using InfiniBand transport, a \textit{Get} fetches a 1 KB object in 93 µs. Referring to the 347 µs number in Figure 8.2 when SSD is mapped to virtual memory, this represents a 3.7X improvement. We also observe that IB-Verbs-level design achieves the best latency for both get and set operations in all object sizes amongst different networks used. In the \textit{Set} operations of 1 KB object, IB-Verbs is 4.4X faster than IPoIB. Compared to 10 GigE and 1 GigE, IB-Verbs wins by 2.9X and 9.4X, respectively. The \textit{Get} is more constrained by SSD random read latency, so the improvements are relatively smaller. At 1 KB object size, IB-Verbs outperforms IPoIB, 10 GigE and 1 GigE by 1.6X, 1.4X and 2.3X, respectively.

8.5.6.2 Memcached Throughput

We run 8 Memcached clients each on a separate machine. Each client performs \textit{Get} operation to read random objects from Memcached server connected via IB-Verbs or 10 GigE transport. The hybrid memory at Memcached server has been populated with 30 GB data with object size=1 KB. We vary the number of Memcached server processes on the server node from 1 to 8 to change the concurrency level of parallel access to the hybrid memory. Figure 8.11(c) reports the aggregated read throughput. The throughput numbers for read-only workload on hybrid memory are also
included in the figure for reference. With high speed networks, Memcached operation throughput approaches the hybrid memory throughput very closely. It is also observed that IB-verbs transport slightly surpasses 10 GigE because of its better latency.

8.5.7 Performance in Datacenter Environment

We have deployed hybrid memory into our testbed that resembles a production datacenter environment as depicted by Figure 8.4 Hybrid part. This environment includes a Memcached client, a Memcached server and a MySQL 5.0 database server with InnoDB storage engine. The database contains 40 million key-value pairs 1 KB each (40 GB data in total). MySQL server uses 3 GB buffer pool size and a 250GB ST3250310NS 7200 RPM hard drive as its storage. The database uses 10 GigE network. The client performs key-value pair query following the three steps as described in Figure 8.3. The requests follow a Zipf distribution [115] such that 80% of requests go to 20% of the data. We vary the available SSD size at Memcached server to yield different hit ratio in hybrid memory. If a desired key-value pair is not found in hybrid memory, the object is fetched from database and inserted into hybrid memory at Memcached server. At the beginning of a test we run queries for sufficiently long time to warm up the hybrid memory and reach a steady hit ratio.

Table 8.2 lists the average query latencies in $\mu$s with different networks connecting the client and Memcached server. Take IB-Verbs for example. With “Basic” approach the Memcached server is constrained by the relatively small RAM size compared to the working set size, and achieves a low hit ratio (33%). This results in more frequent expensive database queries and a high average latency (7349 $\mu$s). On the contrary, Hybrid can afford to cache more data in the larger SSD to obtain much higher hit ratio, and the cost to load data from SSD is fractional compared to a database query. These two factors combined help attain a large gain over the “Basic” approach.
Table 8.2: Resemble a Datacenter Environment: Key-Value Query Latency (µs) with Zipf Distribution of 80/20.

<table>
<thead>
<tr>
<th>Hybrid</th>
<th>Basic</th>
<th>Hit Ratio</th>
<th>Hybrid</th>
<th>Basic</th>
<th>Hit Ratio</th>
<th>Hybrid</th>
<th>Basic</th>
<th>Hit Ratio</th>
<th>Hybrid</th>
<th>Basic</th>
<th>Hit Ratio</th>
<th>Hybrid</th>
<th>Basic</th>
<th>Hit Ratio</th>
<th>Hybrid</th>
<th>Basic</th>
<th>Hit Ratio</th>
<th>Hybrid</th>
<th>Basic</th>
<th>Hit Ratio</th>
<th>Hybrid</th>
<th>Basic</th>
<th>Hit Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0.33</td>
<td></td>
<td></td>
<td>0.50</td>
<td></td>
<td></td>
<td>0.76</td>
<td></td>
<td></td>
<td>0.90</td>
<td></td>
<td></td>
<td>0.95</td>
<td></td>
<td></td>
<td>0.99</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IB-Verbs</td>
<td>7349</td>
<td>5628</td>
<td></td>
<td>2692</td>
<td>1154</td>
<td></td>
<td>650</td>
<td>200</td>
<td></td>
<td>IB-IPOIB</td>
<td>7431</td>
<td>5678</td>
<td></td>
<td>2781</td>
<td>1189</td>
<td></td>
<td>701</td>
<td>261</td>
<td></td>
<td>10GigE</td>
<td>7426</td>
<td>5652</td>
<td></td>
</tr>
<tr>
<td>Analytical Model Error</td>
<td>1.4%</td>
<td>2.4%</td>
<td></td>
<td>2.6%</td>
<td>5.0%</td>
<td></td>
<td>3.6%</td>
<td>6.7%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Hybrid* approach can easily achieve a 95% hit ratio when 30 GB SSD is used to cache data. As a consequence it minimizes average query latency to 650 µs, a 11.3X speedup. Higher hit ratio is possible if more SSD space is allocated into hybrid memory to cache the data.

### 8.5.8 Validation of the Analytical Model and Projections

We also use the models described in Section 8.4 to compute the theoretical latency according to parameters given in Table 8.1. The calculation outcome matches very well with the actual experiment results. The percentage error is given in the last row in Table 8.2.

![Bar chart showing predicted Memcached Query Latency](image)

*Figure 8.12: Predicted Memcached Query Latency by using the Analytical Model from Section 8.4. Latency Normalized to InfiniBand-Verbs. Hit Ratio=0.95.*
In the experiments conducted in previous sections we use a low-end machine as the database server, and the database query latency is around 11,000 µs, which is orders of magnitude higher than network latency and SSD access cost. Therefore the overall query latency is dominated by the database. As a result we do not observe major difference amongst different networks in table 8.2 at a given hit ratio. However if advanced platforms and algorithms are deployed that can dramatically change the relative cost of database operations, the profile will be altered significantly. We use the model from Section 8.4 to derive the Memcached query latencies at a variety of database latency and hybrid memory hit ratio. Figure 8.12 shows an example when hybrid memory hit ratio is fixed to be 0.95 and database latency varies amongst 5,000/1,000/500 µs. The results of different networks are normalized to IB-Verbs. The numbers with database latency=11,000 µs are real experimental results from Table 8.2. A clear trend is observed that, as the database latency drops, the impact caused by different network connections starts to dominate. If database latency can be driven down to 500 µs, for instance, IB-Verbs will accelerate a key-value query by 1.5X, 1.3X, 2.5X over IPoIB, 10GigE and 1GigE, respectively.

8.6 Related Work

Memcached performance is dominated by both network latency and aggregated memory pool size. There have been studies [34,68] to explore high performance interconnect to reduce communication cost. In order to enlarge the memory size, there have been proposals [6] to utilize SSD as a swap device to expand the effective memory size. Although transparent to applications, these approaches incur heavy overhead at virtual memory software stack [72,84] which manipulates SSD at memory-page granularity. Relying on existing virtual memory system also leads to excessive amount of read/write traffic to SSD, which undermines the write durability of SSD. Our hybrid memory design manages SSD resource at object granularity, and effective solves the SSD
wearing concern. With our hybrid memory, Memcached delivers a throughput of around 60,000 read/second which significantly outperforms the 8,000 read/second result reported in [6].

Flash memory has been poised as a revolution in the storage systems [43]. Recently NAND-flash based SSD has been widely deployed in both personal computing and high performance computing (HPC) domains. A large number of deployment treats SSD as a direct replacement of hard disks [76]. Additionally an alternative is to use SSD to extend the virtual memory (VM). One straightforward approach is to use SSD as a VM swap device and let OS manage it [65, 84]. This approach can transparently benefit end user applications, but it leads to severe under-utilization of SSD capabilities because, operating system operates virtual memory (VM) at page granularity, and every object read/write will cause an entire flash page to be read/written no matter how small the object is [65, 84]. This not only wastes the IO bandwidth, but also results in unnecessary flash wearing and undermine its lifespan [118].

FlashStore [38] is proposed for storing large quantity of key-value pairs in local SSD. Although similar to our design, our Hybrid memory differs from it in the index structure used to store the object locations, and Memcached doesn’t require descending additional objects to hard disk if no space is available in SSD. SSDAlloc [35] allows SSD to be used as an extension to RAM, and provides a general object allocation interface to applications. Our hybrid memory design shares a similar concept, but hybrid memory is designed with an indexing structure that is well tailored into Memcached structure for a better match. Due to SSDAlloc’s proprietary nature, we are unable to conduct a performance comparison between hybrid memory and SSDAlloc.
Chapter 9: SOFTWARE DISTRIBUTION

The major portion of the studies described in this dissertation has been incorporated into several software releases.

The optimizations related to HPC fault-tolerance features have been included into MVAPICH2 [18], a high-performance MPI implementation over InfiniBand, 10GigE/iWARP and RoCE. MVAPICH2 is in active use by over 1,820 organizations worldwide, and are also incorporated into a number of different vendor distributions. It is also distributed in the OpenFabrics Enterprise Edition (OFED). Checkpoint/Restart support is included into MVAPICH2 0.9.8 in 2006 [96]. However the basic CR strategy incurs heavy I/O overhead related to concurrent checkpoint writing. Part of the studies in this dissertation is aimed to reduce the CR overhead. These optimizations have already been released with MVAPICH2.

- MVAPICH2 1.6 (March 2011): The Write-Aggregation optimization proposed in Chapter 4 is included in this release to improve I/O efficiency. The CRFS designed in Section 4.4 is also incorporated into this release as a user-level virtual filesystem to benefit generic checkpoint/restart.

- MVAPICH2 1.7 (October 2011): The Pipelined Process Migration over RDMA protocol (PPMR), as proposed in Chapter 6, is incorporated into this release. In addition to this
advanced PPMR algorithm, this release also supports several file-based migration protocols which are described in Chapter 6.

The research conducted in Chapter 7 leads to a new I/O primitive, namely Atomic-Write, which dramatically improves non-contiguous I/O performance. This work has been adopted by Fusion-io ioDrive SSD products [7]. Recently the Atomic-Write primitive has been released by Fusionio in their SSD driver stack to enhance database storage performance [8].
Chapter 10: CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

This chapter includes a summary of the research contributions of this dissertation and concludes with future research directions.

10.1 Summary of Research Contributions

This dissertation aims to design an efficient I/O middleware for High End Computing (HEC) clusters to address the I/O bottleneck that exists in both High Performance Computing (HPC) and Cloud computing domains. The successful research conducted in this dissertation leads to a drastic reduction in the I/O overhead for HEC clusters. Our work focuses on multiple aspects of the I/O stack, and has a significant impact on both industry and academia by contributing novel designs and concepts.

There are multiple research contributions from this dissertation. We propose several technologies in the I/O middleware layer, including:

- CRFS: a light-weight user-level filesystem with Write-Aggregation optimization.

- A Hierarchical Data Staging Framework that leverages advanced network features such as RDMA and new storage technique such as SSD to accelerate checkpointing operations.

- PPMR: Pipelined Process Migration over RDMA which is able to minimize overhead of a process migration.
• Atomic-Write: a new IO primitive for non-contiguous write with atomicity guarantee.

• SSD-Assisted Hybrid Memory that uses SSD to expand program memory size to improve object store performance.

We summarize our research contributions in detail in the rest of this section.

10.1.1 Write-Aggregation to Reduce I/O Overhead in Checkpoint

In Chapter 4 we conduct extensive profiling to identify the dominant factors that determine the cost of Checkpoint-Restart. Based on the findings, we propose an optimization, called Write-Aggregation with Dynamic Interleaving (WAI), to mitigate the checkpoint writing on a multicore platform for MPI checkpoint. We have designed and implemented CRFS, a user-space filesystem, which incorporates the principal of write aggregation. The generic filesystem based architecture enables a wide range of software components, including any MPI stack and general IO application, to transparently benefit from CRFSs optimizations.

10.1.2 A Hierarchical Data Staging Framework

Although the Write-Aggregation algorithm is effective in reducing IO contention on a node-local level, poor performance is observed when multiple compute nodes simultaneously save their checkpoint data to a shared filesystem. In Chapter 5 we explored several design alternatives to develop a hierarchical data staging framework to alleviate the bottleneck caused by heavy I/O contention at the shared storage when multiple processes in an application dump their respective checkpointed data. Using the proposed framework, we are able to decouple the application
progress from the lengthy data IO to the shared filesystem, and drastically drive down the check-
point latency seen by an application. We have also studied the scalability and throughput of hierar-
chical data staging and the merits it offers when it comes to handling large amounts of Checkpoint
data.

10.1.3 High Performance Pipelined Process Migration with RDMA

In Fault Tolerance area, Process/Job migration complements the Checkpoint/Restart approach
that saves full snapshot for each process. In Chapter 6 we conduct extensive profiling on sev-
eral process migration implementations, and reveal that inefficient IO and network transfer are the
principal factors responsible for the high overhead. We have proposed and implemented Pipelined
Process Migration with RDMA (PPMR) strategy into MVAPICH2 [18] to optimize the inefficient
data IO and network transfer at various aspects in a process migration. Our new protocol fully
pipelines data writing, data transfer, and data read on all aspects of a migration. PPMR aggre-
gates data writings on migration source node and transfers data to target node via high throughput
RDMA transport. PPMR implements an efficient process restart mechanism in the target node to
restart processes from RDMA data streams.

10.1.4 Improving Non-Contiguous Access with Atomic-Write

Non-contiguous access is a common data access pattern in many scientific applications. They
perform a lot of small random access which may overlap, and data consistency has to be guaranteed
for those concurrent overlapping accesses. SSD provides outstanding random access feature, but
providing data consistency imposes significant overhead in the existing software stack. In Chap-
ter 7 we design a new storage primitive for SSD, Atomic-Write, within a log-structured FTL that
allows multiple I/O operations to commit or rollback as a group in an atomic manner. We have
shown that the FTL is a natural placement for atomic-write semantics because they can utilize
the already existing FTL block tracking mechanisms for commit, rollback, and recovery. Atomic Write primitive not only improves the non-contiguous access performance by reducing the duplicated writing, but also leads to a much better write endurance of SSD devices. Atomic Write can also benefit a wide range of transactional storage manager with ACID compliance requirement.

10.1.5 SSD-Assisted Hybrid Memory

SSD has received a wide deployment in both HPC and Cloud Computing domains. In addition to being a direct replacement to hard drives, SSD can also be utilized in virtual memory system. In Chapter 8 we propose SSD-Assisted Hybrid Memory that augments RAM with SSD to enlarge the available memory size to applications. We have implemented a prototype of Hybrid memory to be used as a huge object cache to store key-value pairs. Unlike VM swap system that manipulates memory and SSD at page granularity, we manage memory allocation at object level. We organize SSD into a log-structured sequence of blocks to avoid expensive in-place update of SSD. With this strategy we can efficiently support random update and also achieve even wear leveling. We have integrated Hybrid memory into Memcached, and deployed it into a datacenter environment with various networking technologies.

We have also developed an analytical model that matches very well with real experiments results. Using this model we are able to foresee the future performance trends when technology advancements alter the various parameters in the system.

10.2 Future Research Directions

This dissertation focuses on an IO middleware that contains optimizations in many aspects of the IO stack to relieve the bottleneck related to storage IO in HEC clusters. In addition to the efforts in the dissertation, there are still several research areas to explore.
10.2.1 Improvements on Checkpoint/Restart

The optimizations we have proposed in the dissertation can improve checkpoint performance by minimizing the concurrent writing overhead. Additional research is possible to explore strategies that reduce the amount of checkpoint data by performing inline compression. This approach yields less amount of checkpoint data dumped to the storage system, which translates into minimized IO burden imposed on the storage system, and it will work in synergy with our existing research to further reduce the checkpoint delay seen by an application. However, data compression and de-compression consumes CPU cycles. CPU consumption may not seem to be a significant issue in checkpoint/restart scenario which is totally IO bounded. Studies are needed to evaluate the effects of data compression during checkpoint, and decide a proper trade off between more CPU usage and less data IO.

10.2.2 Improvements in PPMR protocol

Inline data compression can also be beneficial to PPMR by reducing the on-wire data to be transferred during a process migration. However, the additional CPU cost at source-side packing and target-side unpacking should be investigated to justify the validity of this approach.

10.2.3 Improvements on SSD-Assisted Hybrid Memory

The hybrid memory combines RAM and SSD into a compound program memory. Hybrid memory can be improved from multiple aspects:

- **RAM Usage:**

  In the SSD-Assisted Hybrid Memory, RAM usage is an important aspect in the Lookup design. Our initial design contains the full key in the index entry for quick search. Better strategies are needed to alleviate the RAM usage in the lookup table. Additional efforts can
be taken to investigate more memory-frugal mechanisms that stores only the fingerprint of key name in the index entry [38].

• **Light-weight Checkpoint:**

Since SSD is a non-volatile storage, majority of the data in hybrid memory has already been stored persistently in SSD and a small portion of recently accessed data is cached in Read/Write buffer. This creates the potentials to perform a light-cost transparent checkpoint to reliably save an application’s working set data before the application aborts, and reconstruct the working set by rebuilding the index table from the contents in SSD. This light-weight checkpoint will be much cheaper than a conventional checkpoint which dumps all in-memory data to persistent storage and later restore by loading from the storage to in-memory data.

• **Virtual address based hybrid memory:**

As of now, the hybrid memory provides a key-value access interface which fits well with an object caching layer such as Memcached. Further work is needed to expand this interface to support virtual memory access capability such that hybrid memory be utilized as a transparent virtual memory by a wide range of applications. Memory-intensive applications can access hybrid memory via standard pointers without any changes in the user code. This can be very useful in out-of-core algorithms where the application explicitly move data between memory and external storage to keep the working set data in memory. Given virtual-address access support, it becomes possible to replace out-of-core algorithms with hybrid memory.
Bibliography


[17] MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/.
[18] MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/.

134


[38] Biplob Debnath, Sudipta Sengupta and Jin Li. FlashStore: High Throughput Persistent Key-Value Store. PVLDB, 2010.


