

# TOWARDS SCALABLE DEEP LEARNING VIA I/O ANALYSIS AND OPTIMIZATION

Sarunya Pumma,\* Min Si,<sup>†</sup> Wu-chun Feng,\* and Pavan Balaji<sup>†</sup>

\*Virginia Tech, USA; {sarunya, wfeng}@vt.edu

<sup>†</sup>Argonne National Laboratory, USA; {msi, balaji}@anl.gov

**Abstract**—Deep learning systems have been growing in prominence as a way to automatically characterize objects, trends, and anomalies. Given the importance of deep learning systems, researchers have been investigating techniques to optimize such systems. One area of particular interest has been using large supercomputing systems to quickly generate effective deep learning networks: a phase often referred to as “training” of a deep learning neural network. As we scale existing deep learning frameworks—such as Caffe—in these large supercomputing systems, we notice that the parallelism can help improve computation tremendously, leading data I/O as the major bottleneck limiting the overall system scalability. In this paper, we first present a detailed analysis of the performance bottlenecks of Caffe on large supercomputing systems. Our analysis shows that the I/O subsystem of Caffe—LMDB—reliably memory-mapped I/O to access its database, which can be highly inefficient on large scale systems because of its interaction with the process scheduling system and the network-based parallel filesystem. Based on this analysis, we then present LMDBIO, our optimized I/O plugin for Caffe that takes into account the data access pattern of Caffe in order to vastly improve I/O performance. Our experimental results show that LMDBIO can improve the overall execution time of Caffe by nearly 20-fold in some cases.

## I. INTRODUCTION

Deep learning has quickly grown in prominence as a key technology to analyze and characterize large volumes of data. Deep learning systems utilize a neural network to represent a complex mathematical equation that can precisely predict outputs based on given inputs. Training a deep neural network (DNN) is nontrivial because it iteratively processes a tremendous amount of training data that requires considerable computation power and memory. Consequently, several researchers have parallelized deep learning frameworks that can harness the performance of large supercomputing systems to efficiently train the DNN on large volumes of training data. For instance, Caffe [8] is a well-known deep learning framework that has been parallelized for efficient execution on large supercomputing systems [12], [2], [4].

Due to the need to train the DNN with an increasingly large amount of data, researchers have been pushing the boundaries of parallelism and scalability that can be achieved by such frameworks. Common techniques in this area include sophisticated preconditioning approaches that enable the network training process to begin at a better starting point, rather than a fully random network, thus allowing for improved parallelism and, in turn, performance. Such a push in parallelism and scalability, however, has started to expose new bottlenecks in the I/O subsystem of deep learning frameworks, such as Caffe, that urgently need to be addressed.

To put this situation into context, deep learning frameworks such as Caffe, iteratively process data in three steps: (1) receiving data from the I/O subsystem, (2) performing computation to identify deviation errors in the network, and (3) performing backpropagation in order to modify the network appropriately. This process occurs iteratively over a large number of iterations and training datasets in order to converge to the desired network accuracy. As this deep learning training process is scaled up to large supercomputing systems or to use highly efficient computational units such as NVIDIA GPUs, Intel Xeon Phi, Google TPU processors,<sup>1</sup> computation time continues to decrease, making data I/O the primary bottleneck in the system, limiting overall scalability. In fact, our analysis of two large datasets, CIFAR10-Large [10] (on a medium-sized network called AlexNet [9]) and ImageNet [6] (on a large-sized network called CaffeNet [11]) showed that even with a small amount of asynchrony in the network processing of approximately 4,096 images in each batch, the performance of Caffe effectively flattened out even at a relatively small scale of 512 processes, with I/O taking up a dominant fraction of the overall execution time.

To understand this problem, we first present a detailed analysis of the I/O subsystem in Caffe. Our analysis shows that Caffe’s I/O subsystem, called LMDB, relies on memory-mapped I/O to access its database, which can be highly inefficient in large-scale systems because of its interaction with the process scheduling system and the network-based parallel filesystem. Based on this analysis, we present LMDBIO, our optimized I/O plugin for Caffe that takes into account the data access pattern of deep learning frameworks, such as Caffe, in order to significantly improve I/O performance. We present performance results comparing Caffe-LMDBIO with the original Caffe with various datasets and networks and show that LMDBIO can improve the overall execution time of Caffe by nearly 20-fold in some cases.

The rest of the paper is organized as follows. Section II presents an overview of Caffe, LMDB, mmap, and the Completely Fair Scheduler (CFS) to frame our subsequent discussion. Section III provides a detailed analysis of the I/O subsystem of Caffe, pointing out causes of its low performance. Section IV describes LMDBIO and how it addresses the shortcomings of Caffe’s current I/O subsystem. Experimental results comparing Caffe-LMDBIO with the original Caffe are presented in Section V while related work is discussed in Section VI and concluding remarks in Section VII.

<sup>1</sup>[https://en.wikipedia.org/wiki/Tensor\\_processing\\_unit](https://en.wikipedia.org/wiki/Tensor_processing_unit)

## II. BACKGROUND

Here we present an overview of the Caffe deep learning tool, LMDB and mmap, and the Completely Fair Scheduler.

### A. Caffe Overview

Caffe is a well-known convolutional neural network training framework developed by the Berkeley Vision and Learning Center. The original framework was written in C++ with CUDA for highly optimized GPU computation, although subsequent variants of Caffe have included support for generic CPU architectures as well.

The general workflow of Caffe is as follows. At the beginning of each training iteration, each worker reads a subset of a batch from a dataset. The framework provides a variety of data reading options via different types of databases. The default and the most widely used option is to use the Lightweight Memory-mapped Database (LMDB) to access the dataset. Once the data is read, it starts performing a forward training by computing a weighted sum of inputs in each layer until it reaches the last layer where a classification error is calculated.

The backward pass, where error is propagated back into the network to calculate the network's parameter gradients. When the backward pass is done, the gradients computed by each worker are accumulated and used for updating the parameters in the network. During the training, the quality of the network parameters is periodically tested.

The training is repeated until termination criteria are satisfied, that is, until the desired accuracy or the maximum number of iterations is reached.

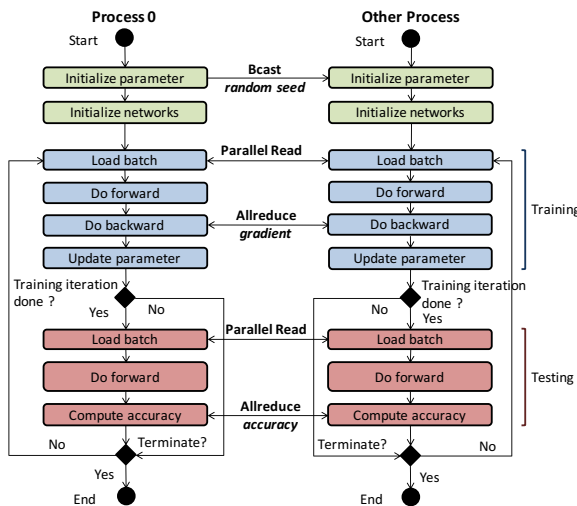


Fig. 1. Caffe workflow

Since sequential DNN training is not practical, researchers have begun investigating parallel training for derivatives of Caffe [12], [2], [4], using either supercomputing systems or accelerators, where a batch is divided into multiple chunks

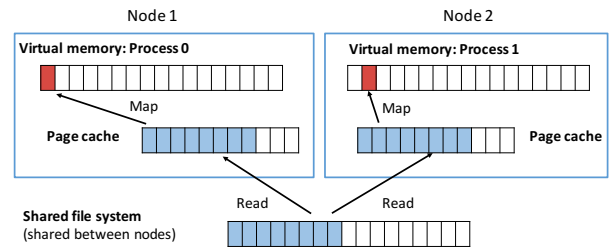


Fig. 2. Workflow of mmap

that are processed concurrently on multiple workers (i.e., processes and threads). The workflow of parallel training in Caffe is illustrated in Figure 1. At the beginning of each training iteration, each worker reads a subset of a batch from a dataset. The framework provides a variety of data reading options via different types of databases. The default and the most widely used option is to use the Lightweight Memory-mapped Database (LMDB) to access the dataset.

Once the data is read, it starts performing a forward training by computing a weighted sum of inputs in each layer until it reaches the last layer where a classification error is calculated. The backward pass, where error is propagated back into the network to calculate the network's parameter gradients. When the backward pass is done, the gradients computed by each worker are accumulated and used for updating the parameters in the network. During the training, the quality of the network parameters is periodically tested. The training is repeated until termination criteria are satisfied, that is, until the desired accuracy or the maximum number of iterations is reached.

### B. LMDB and Mmap

Caffe uses the LMDB database format and library for accessing the training dataset. LMDB represents the data samples in a B+ tree format due to its ability to allow for block-based I/O to improve I/O performance. LMDB internally uses mmap to map the entire dataset into memory and parse through the dataset as if it fully resided in memory.

The Unix system call mmap exposes the layout of a file from the file system into the virtual address space of a process and enables accesses to the file as if it were a memory buffer. With mmap, segments of a file, at a page granularity, are loaded only when they are accessed. Therefore, the mmap file access method is beneficial for partial file accesses where an entire file is not accessed at once or when the file represents a complex database rather than a dataset and might require nontrivial access patterns. When the file segments are loaded to memory, they can be mapped by multiple processes.

Three components are associated with mmap, as shown in Figure 2: the shared file system (which can be local or shared across machines), the page cache (which is shared across processes), and the virtual address space (which is private to a process). When mmap is called, a process receives a virtual address range that the file will be mapped to. The process can access the file buffer freely as if all the data were in memory. If the process attempts to access a part of the buffer where the corresponding portion of the file has not yet been mapped

115-120

6 notes:

added to memory, a page fault happens, and the page-fault handler is invoked. The page-fault handler will initially try to find this page in the page cache to determine whether the page has already been read to memory.

121-123

3 notes:

memory, the handler will map the physical address of the page to the corresponding virtual address of the process.

124-126

3 notes:

If the page is not found in memory, the handler will issue an I/O request to read the page and its neighboring pages to memory.

While waiting for completion of the I/O operation that is fetching the data from the file system, the process is put to sleep and context switched out. It will be woken up to check whether its I/O request has been completed by the operating system process scheduler when I/O completion interrupts come in.

### C. Completely Fair Scheduler

127-129

3 notes:

Linux kernel 2.6 introduced the Completely Fair Scheduler (CFS) as the default process scheduler. CFS guarantees fairness of CPU usage between processes and attempts to maximize CPU utilization. The scheduler schedules processes to execute on the CPUs from a red-black tree where a process with the least-used CPU time will be chosen to run first. The scheduler does not take into account the order in which the processes are enqueued.

130-131

2 notes:

In case that processes are waiting for I/O operations to complete, they are put to sleep. Once an I/O completion interrupt comes in, all the processes that are waiting for I/O are marked as runnable. The CFS scheduler then wakes up the runnable processes in the red-black tree to let the processes check whether the I/O completion corresponds to the operation on which they are waiting. If the I/O operation that completed is not the one they were waiting for, they go back to sleep.

132-134

3 notes:

135-138

4 notes:

139-141

3 notes:

## III. ANALYSIS OF I/O IN CAFFE

In this section, we present an analysis of the I/O performance of Caffe. Caffe is read-intensive and does not perform any significant file writes, so much of the analysis is on the file read accesses in Caffe.

142-146

5 notes:

### A. Caffe Scalability Analysis

Before we consider the I/O performance in Caffe, we study the overall execution performance of Caffe in order to understand the state of I/O in the current framework. Here we first analyze Caffe's scalability by training the CIFAR10-Large dataset using the AlexNet DNN model. We use a batch size of 4,096 and perform scaling experiments on the Argonne's Blues cluster (testbed details are provided in Section V-A).

147-150

4 notes:

151-156

6 notes:

Let us first consider the overall execution time scalability (strong scaling) of Caffe compared with ideal scaling in Figure 3(a). We notice in the graph that the actual training time starts to differ from the ideal scaling time after just two processes and that the difference increases with the number of processes. In fact, with just 512 processes, the performance of Caffe is only 20-fold worse than the ideal scaling performance. Understand this result better, we analyzed the breakdown in time taken by the various components of Caffe

157-159

3 notes:

in Figure 3(b). We note that the data I/O time (represented as "Read time" in the figure) becomes highly significant when training a network on a large number of processes. It takes approximately 70% of the overall training time when using 512 processes and tends to increase when using a larger number of processes.

Since I/O is a major bottleneck in DNN training, we now analyze Caffe's data reading performance. The default data reading approach of Caffe is to read the dataset from a memory-mapped file (using `mmap`) via LMDB. To understand the data reading performance in Caffe, we measured the data reading bandwidth of Caffe using different numbers of readers and compared the results with those of POSIX I/O read using the IOR benchmark.<sup>2</sup> The IOR performance is typically considered the best case for POSIX I/O that is achievable on a given platform. Figure 4 shows that the read bandwidth of Caffe is much worse than that of POSIX I/O read from the IOR benchmark. This shows that while I/O takes a significant amount of time in Caffe, most of this performance loss is due not to hardware limitations in the system but to the inefficient usage of the available I/O bandwidth.

### B. Mmap Analysis

As discussed in Section II-B, `mmap` exposes the layout of a file from the file system into the virtual address space of a process, which enables accesses to the file as if it were a memory. Caffe's fundamental workflow relies on dynamically fetching data from the file system to physical memory and mapping pages from the physical memory to the corresponding virtual address space of the process.

Despite its obvious benefits, `mmap` suffers from a few shortcomings, specifically in the way it handles I/O requests. When a user process accesses a page, if that page is not already in memory, a page fault handler is triggered to fetch the data from the file system. This I/O request is then passed down to the hardware controller (e.g., SCSI for local storage or a network adapter for network-based file systems) and the user process goes to sleep while waiting for the I/O request to complete.

When the I/O operation completes, the hardware controller takes an interrupt informing the file system of completion. One important aspect to note here is that this interrupt handler is a *bottom-half* handler in Linux. This interrupt is not associated with any particular user process in the file system but is a generic event informing the file system that an I/O operation was issued by one or more processes has now completed. This interrupt handler then marks as complete all processes that were sleeping while waiting for an event.

At this point, the CFS scheduler takes over. The next time the scheduler is triggered, it traverses all processes in its red-black tree and schedules the runnable processes one at a time. In general, however, since only some of the processes were waiting for that specific I/O operation that just completed, most processes will see that their I/O operation has not completed

<sup>2</sup><https://github.com/LLNL/ior>

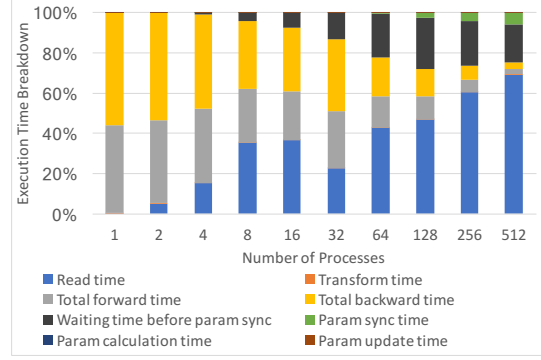
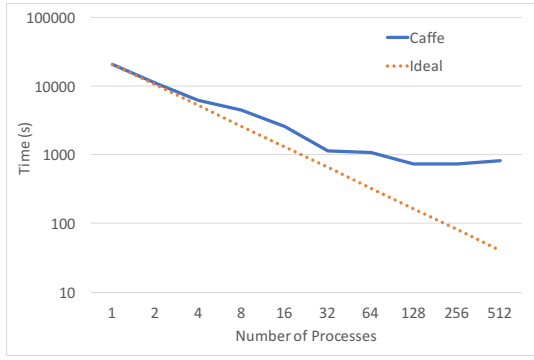


Fig. 3. Caffe scalability (CIFAR10-Large dataset): (a) scaling analysis; (b) scaling time breakdown

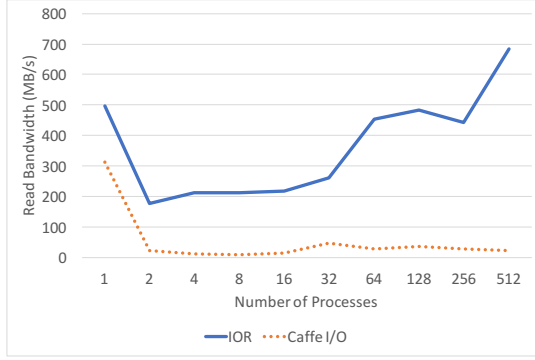


Fig. 4. Caffe I/O performance analysis (CIFAR10-Large dataset)

#### IV. LMDBIO: DESIGN AND IMPLEMENTATION

As discussed in Section III, the current I/O infrastructure in Caffe is highly inefficient in the way that it accesses data. The primary inefficiency comes from the way `mmap`-based I/O requests are generated and handled by multiple processes. To alleviate this issue, we propose LMDBIO—an I/O plugin for Caffe that takes the data access characteristics of Caffe into account to optimize the I/O performance.

##### A. Overview of LMDBIO

One of the major drawbacks in the current data I/O model in Caffe is the lack of true parallelism. Data fetching for each process is independent of the other processes even though it is a parallel application and the I/O could, in principle, be coordinated, thus losing several potential optimization opportunities. Often, this approach results in unnecessary data to be read and discarded. Moreover, since all processes are performing data I/O and relying on the underlying bottom-half handler to wake them up, the wakeup model is imprecise and leads to unnecessary context switches.

The general idea of LMDBIO is to utilize what we refer to as “localized `mmap`.” In this model, a single process is chosen on each node as the root process. The root process reads data from the file system and distributes it to the remaining processes on the node using MPI-3 shared memory. This approach aims to reduce I/O parallelism to give `mmap` a more sequential view of I/O and to minimize interprocess contention. The `mmap` localization approach also allows the traditional Linux bottom-half handler for I/O to wake up the exact process that is waiting on I/O, since only one process is performing I/O. This strategy minimizes the number of context switches and helps improve performance.

LMDBIO is written in C++ and utilizes MPI and LMDB as core engines. We assume the availability of MPI-3<sup>3</sup> in order to allow LMDBIO to detect process colocation automatically, perform reader assignment, and utilize a shared-memory buffer. LMDBIO adopts LMDB’s API for accessing

<sup>3</sup>Most supercomputers in the world already support MPI-3. The only notable exception to this claim is the IBM Blue Gene series of supercomputers that do not yet support MPI-3. However, these supercomputers are nearing their end of life, and the next generation of supercomputers from IBM do plan to support MPI-3 and later MPI standards.

160-165 6 notes: and back to sleep. Only one or a few processes will be able to use this completion to perform further processing.

166-168 3 notes: Consequently, this model significantly increases the number of context switches that get triggered with most of the switches coming in no real work. This approach thus increases the amount of “sleep time” associated with each process as well.

This problem is not present when performing simple `mmap` with a single process accessing the data. In such cases, the completion handler wakes up only one process, and every completion corresponds to the exact process that is waiting for that I/O operation. But, the larger the number of readers, the greater the chance that the processes will be woken up without having any real work to do. Therefore, we expect the number of context switches in the processes to grow as we increase the number of readers. Moreover, we expect that more than one process, the processes will spend most of their data reading time waiting for I/O (i.e., sleeping).

169-171 3 notes: This problem is not present when performing simple `mmap` with a single process accessing the data. In such cases, the completion handler wakes up only one process, and every completion corresponds to the exact process that is waiting for that I/O operation. But, the larger the number of readers, the greater the chance that the processes will be woken up without having any real work to do. Therefore, we expect the number of context switches in the processes to grow as we increase the number of readers. Moreover, we expect that more than one process, the processes will spend most of their data reading time waiting for I/O (i.e., sleeping).

172-176 5 notes: This problem is not present when performing simple `mmap` with a single process accessing the data. In such cases, the completion handler wakes up only one process, and every completion corresponds to the exact process that is waiting for that I/O operation. But, the larger the number of readers, the greater the chance that the processes will be woken up without having any real work to do. Therefore, we expect the number of context switches in the processes to grow as we increase the number of readers. Moreover, we expect that more than one process, the processes will spend most of their data reading time waiting for I/O (i.e., sleeping).

177 chen ping We demonstrate this behavior in Figures 5(a) and 5(b). As expected, the context switches increase as we increase the number of processes, and the sleep time when using more than one process takes approximately 90% of the overall read time. These results show that `mmap`-based file access is highly inefficient with more than one process because processes are wasting time in getting context switched in and out for I/O interrupts that do not belong to them.



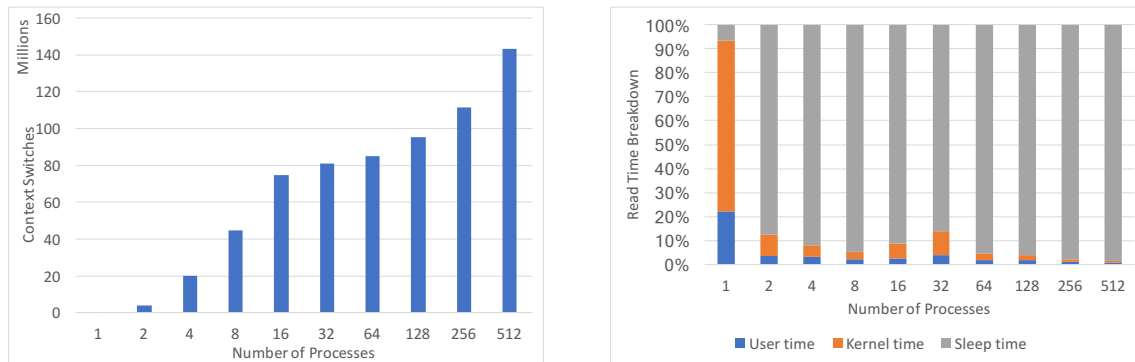


Fig. 5. Caffe mmap analysis (CIFAR10-Large dataset): (a) context switches; (b) sleep time

the database file efficiently. Moreover, LMDBIO abstracts parallel data reading from applications and provides a convenient C++ API that the applications can utilize to obtain the data for each process.

### B. Detecting Colocated Processes

One of the first aspects that we need to solve in order to achieve localized mmap as described in Section IV-A is to detect which processes reside on the same node (or more precisely, which processes share the same mmap page cache and bottom-half interrupt handlers).

Achieving this objective portably is hard. LMDBIO adopts a feature in MPI-3 to split a global MPI communicator into multiple local communicators (`MPI_Comm_split_type` with `MPI_COMM_TYPE_SHARED`). The general idea of this feature is to inform the user of the group of processes that are capable of allocating a shared-memory buffer. In theory, the MPI implementation can provide any group of processes that are capable of creating a shared-memory buffer together. For example, this could be all the processes on the same NUMA domain or same socket. In practice, however, this group is often the processes that reside on the same node and thus share the same page cache and bottom-half interrupt handlers. This gives us a semi-portable way to detect processes on the same node with the added convenience that in case the approach does not give the right set of processes, we can gracefully degrade performance rather than failing outright.

Once the communicators are set up, LMDBIO chooses one reader from each local group as the “root.”

### C. Inner Workings of LMDBIO

LMDBIO consists of two phases: an initialization phase and a data reading phase.

**Initialization phase:** In the initialization phase, LMDBIO assigns one reader per node using the approach mentioned in Section IV-B. Then, each reader opens the LMDB database that internally maps the database to that process’s virtual address space using `mmap`. All processes on the node also allocate a shared-memory buffer that is directly accessible by all of them.

**Data reading phase:** In the data reading phase (shown in Figure 6), each reader in LMDBIO (one process per node)

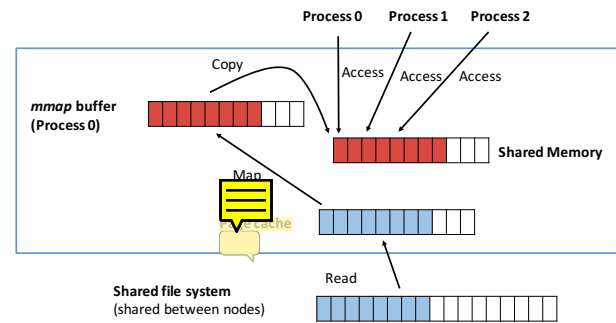


Fig. 6. LMDBIO overview

reads  $B/R$  data samples ( $B$  is the batch size and  $R$  is the number of readers) from the database. The data is read from the file system to page cache and mapped to the virtual address space of the reader process. The reader process then copies the data to the shared-memory buffer that was allocated during the initialization phase in order to allow other processes to access this data. LMDBIO then synchronizes the processes within the local communicator (to ensure that the read has completed), after which the other processes on the node are allowed to access the shared-memory buffer. Even though each process has full access to all of the shared-memory buffer, LMDBIO internally limits such access so that each process can access only  $B/P$  samples of data ( $P$  is a total number of processes).

### D. Shortcomings of LMDBIO

Despite the various advantages of LMDBIO it still suffers from some shortcomings.

**Serialized I/O:** LMDBIO serializes I/O on each node so only one process per node is doing the I/O rather than all processes. While this approach is beneficial for minimizing the amount of inter-process I/O contention that occurs with `mmap`, we miss the opportunity to maximize a read bandwidth of the file system. To utilize the I/O bandwidth more efficiently with multiple processes, we have to use other approaches, for example, direct I/O, in order to avoid the problems with `mmap`.

**Buffer Aliasing:** In general, when a buffer is allocated, the allocated memory can contain pointers to other pieces of memory. Thus, any access on such memory could inadvertently modify other memory regions: a problem referred to as

178-180

3 notes:

181-183

3 notes:

buffer aliasing. Most compilers are conservative in computing on aliased buffers since they need to be aware of such side effects and consequently generate less-optimized code. Malloc and malloc-like memory allocation calls are special in that they pass a special attribute to the compiler assuring it that any buffer allocated through malloc is not aliased. Thus, the compiler can perform more aggressive optimizations on this buffer, leading to better performance. Unfortunately, this “no aliasing” attribute can be passed to the compiler only when the return value of the allocation function is the memory buffer, not when the memory buffer is a function parameter. The MPI-3 shared-memory buffer allocation function misses this optimization opportunity: the shared-memory buffer that is allocated is not the return value of the function, but rather a function parameter. This is a flaw in the MPI-3 interface design and one that can cause degradation in the compiler optimization. This issue can be worked around by using `restrict` pointers to access the buffer, which provides the compiler with equivalent information as an unaliased buffer, thus achieving the same level of performance. But this is an extra step that the application needs to be aware of.

**TLB Misses:** In LMDBIO, the data samples are copied from the `mmap` buffer into a shared-memory buffer. In traditional Caffe, this data is copied into a regular malloc buffer. Unfortunately, malloc buffers and shared-memory buffers differ significantly in the way the OS assigns physical pages to them. Buffers allocated with malloc use large 2 MB pages on most processors, whereas shared-memory buffers use regular 4 KB pages. Thus, computing directly on this buffer might cause a larger number of TLB misses when using shared memory than when using malloc. In Caffe, we are not affected by this issue because of a second transformation that copies the data again into another malloc buffer before any core computation is done. Thus, the core computation itself does not suffer from the increased TLB misses. Nevertheless, this is an issue that future variants of Caffe need to be aware of.

## V. EXPERIMENTS AND ANALYSIS

In this section we present and analyze several experimental results to showcase the capability of LMDBIO.

### A. Experimental Platform

The experimental evaluation for this paper was performed on Argonne’s “Blues” cluster.<sup>4</sup> Blues consists of 310 computing nodes connected via InfiniBand Qlogic QDR. Each node has 64 GB of memory and two Sandy Bridge 2.6 GHz Pentium Xeon processors (16 cores, hyperthreading disabled). The storage is 110 TB of clusterwide space provided by GPFS and 15 GB of on-node ramdisk. We built both the original Caffe and Caffe-LMDBIO by using the Intel ICC compiler (version 13.1.3). We used MVAPICH-2.2 with the PSM netmod [17] for all experiments. All experiments were run three times and the average performance is shown.

We used two datasets for our experiments. The first dataset was the CIFAR10-Large dataset that was trained by using the AlexNet DNN model. The CIFAR10-Large dataset consists of 50 million sample images, each approximately 3 KB. The total dataset size, including the raw images and some metadata corresponding to the images, is approximately 190 GB. The second dataset was the ImageNet dataset that was trained by using the CaffeNet DNN model. The ImageNet dataset consists of 1.2 million sample images, each approximately 192 KB. The total dataset is 240 GB. Although both datasets can be I/O intensive, the ImageNet dataset is particularly so, given the size of the images that need to be processed.

For our experiments we used a batch size of 4,096 for both datasets. We trained the network for the CIFAR10-Large dataset over 1,024 iterations (4 million images) and the ImageNet dataset over 32 iterations (128K images).

### B. Performance Comparison

Figure 7(a) compares the performance of the original Caffe with that of Caffe-LMDBIO for the CIFAR10-Large dataset. The general trend that we notice is that Caffe-LMDBIO performs better than Caffe by up to a factor of 1.75. Figure 7(b) shows the breakdown of performance for Caffe-LMDBIO where we notice that time taken by the data I/O (represented as “Read time” in the figure) has reduced significantly compared with that of the original Caffe implementation (shown in Figure 3(b)), specifically from 70% to 30%. We note that our work is not fully done yet; I/O still takes a large portion of the overall time even in the LMDBIO optimized version. Nevertheless, the proposed approach is still a significant step toward improving I/O performance for Caffe. Moreover, this improvement is despite the reduced I/O parallelism and the extra data copy that we perform within Caffe-LMDBIO.

We performed a similar analysis on the ImageNet dataset, as shown in Figures 8(a) and 8(b). The general trend for ImageNet is similar to that of CIFAR10-Large. However, we notice that the performance improvement with Caffe-LMDBIO is significantly larger for ImageNet, showing improvements up to 20-fold in some cases. This larger improvement for ImageNet is attributed to the larger images in its dataset.

### C. Analysis of Context Switches

Apart from overall performance, we also studied the number of process context switches with Caffe-LMDBIO. Again, our analysis was performed with both CIFAR10-Large and ImageNet. Figure 9 shows the improvement in context switches for the two datasets. We see that the number of context switches is significantly better for Caffe-LMDBIO compared with that of the original Caffe. For the CIFAR10-Large dataset, LMDBIO reduces the number of context switches by nearly 120-fold. For ImageNet, the improvement is even more dramatic, with close to a 700-fold reduction in the number of context switches.

This improvement is expected. Since LMDBIO has a single process performing `mmap` it ensures that no contention occurs between `mmap` calls performed by multiple processes. This serialization reduces the number of unnecessary wakeups

<sup>4</sup><http://www.lrcr.anl.gov/about/blues>

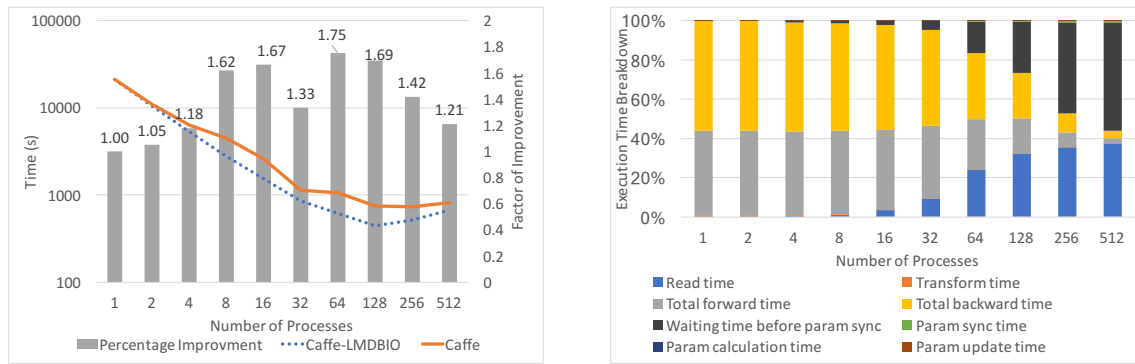


Fig. 7. CIFAR10-Large dataset: (a) performance comparison of Caffe and Caffe-LMDBIO; (b) performance breakdown of Caffe-LMDBIO

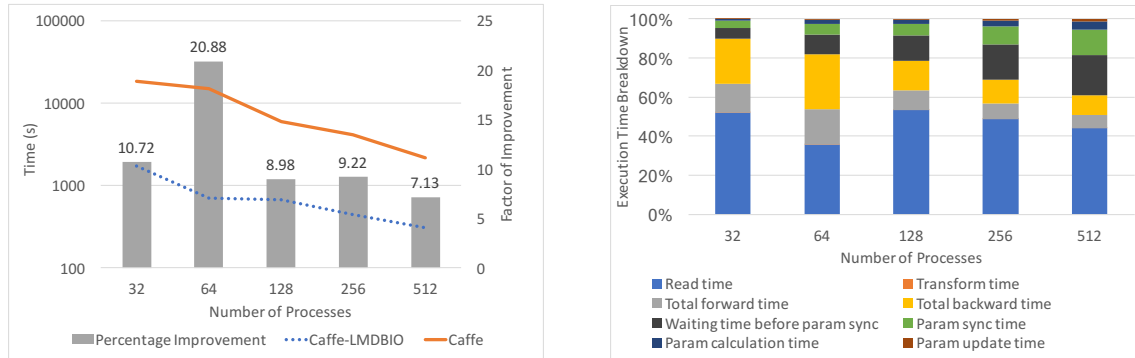


Fig. 8. ImageNet dataset: (a) performance comparison of Caffe and Caffe-LMDBIO; (b) performance breakdown of Caffe-LMDBIO

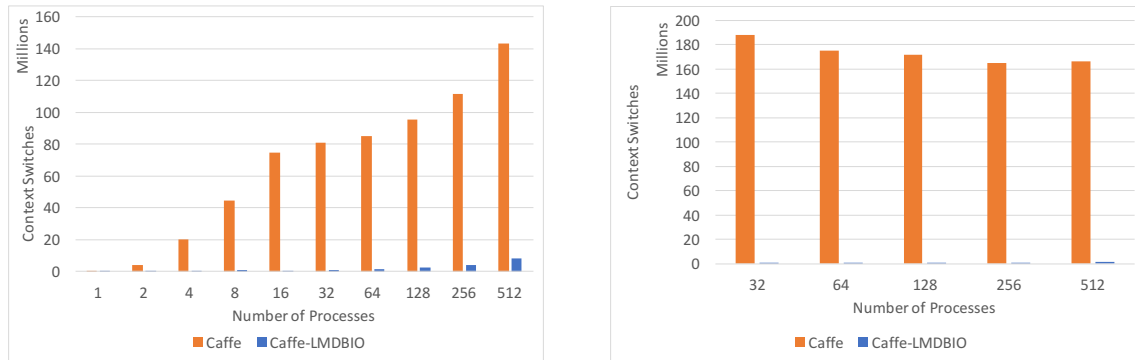


Fig. 9. LMBIO context switches: (a) CIFAR10-Large dataset; (b) ImageNet dataset

created by the interrupt handler thus reducing the number of context switches.

## VI. RELATED WORK

Given the immense popularity of deep learning, considerable research has been conducted addressing various aspects of deep learning.

**184-185** **Other Deep Learning Frameworks:** While this paper focuses on Caffe, it is not the only deep learning framework used in the community. Theano [18] was one of the first deep learning toolkits which also has a parallel version [14]. Google's TensorFlow [3] is another highly flexible dataflow-like architecture where nodes in the graph represent mathematical operations and an edge between nodes represents a multidimensional data array communicated between them, called a tensor. Recently,

a distributed version of TensorFlow has been released, where tasks can be distributed over multiple cluster nodes with data transferred through the gRPC protocol [1]. Vishnu et al. contributed to the distributed version of TensorFlow using MPI [19] to scale it on supercomputing systems. While all these frameworks allow for different forms of parallelization and network generation, however, their core I/O infrastructure is not that different. **186** Particular, using LMDB datasets for storing data samples is a common practice in the community; and regardless of which deep learning framework we use, as long as it accesses LMDB datasets it will suffer from the same. Thus, our work is broadly applicable to these other deep learning frameworks as well.

Many parallel variants of Caffe itself have been published. Of these, MPI-Caffe [12] and Caffe-MPI [2] target only the

compute portions of the framework and do little to optimize I/O. S-Caffe [4] performs parallel data read but does not analyze the issues with Caffe-IO and inherits many of its shortcomings. Our work aims at fixing the underlying cause of the performance degradation in Caffe-IO, making it applicable to all Caffe derivatives.

**Parallel I/O:** In HPC systems, I/O subsystems are typically the slowest components. Researchers are looking into ways to parallelize data access in order to improve the I/O bandwidth. MPI-IO [15], [16] provides a low-level interface to carry out parallel I/O for generic unstructured data. Other higher-level I/O libraries such as HDF5<sup>5</sup> and NetCDF [5] also exist. These libraries abstract various structured scientific application data into portable file formats and provide feature-rich programming interfaces. Parallel HDF5 [7] and PnetCDF [13] provide parallel access and storage for files with those formats based on MPI-IO.

Although these I/O frameworks are more efficient than `mmap`, they are all based on explicit I/O. That is, the user must provide the exact bytes in the file that would be accessed before actually accessing the file. On the other hand, `mmap` forms implicit I/O. That is, it maps the entire file into the virtual address space of the process; and depending on what part of the file is being accessed, it dynamically fetches the corresponding data. This implicit model is more convenient for complex datasets that require I/O access that is not simple sequential reading (e.g., LMDB uses a B+ tree format to store its data). Thus, while in the long term it is valuable to migrate the I/O model of Caffe and other deep learning systems to use explicit I/O, our approach provides the shortest path solution to improve I/O performance without requiring all existing datasets to be migrated away from the LMDB format.

## VII. CONCLUDING REMARKS

In this paper, we presented a scalable I/O plugin, called LMDBIO, for the Caffe deep learning framework. We first performed a detailed analysis of I/O in Caffe, showcased the problems associated with it, and discussed the cause of these problems. We then presented LMDBIO, which alleviates these problems to improve the I/O performance. We presented experimental results with two datasets and network models, demonstrating nearly a 20-fold improvement in the overall performance of Caffe in some cases.

## ACKNOWLEDGMENTS

This material was based upon work supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357, and in part by the NSF XPS program via CCF-1337131. The experimental resources for this paper were provided by the Laboratory Computing Resource Center at Argonne National Laboratory.

## REFERENCES

- [1] GRPC: A High Performance, Open-Source Universal RPC Framework. <http://www.grpc.io>.
- [2] Caffe-MPI for Deep Learning. <https://github.com/Caffe-MPI/Caffe-MPI.github.io>, September 2015.
- [3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from [tensorflow.org](http://tensorflow.org).
- [4] Ammar Ahmad Awan, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K Panda. S-Caffe: Co-designing MPI Runtimes and Caffe for Scalable Deep Learning on Modern GPU Clusters. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 193–205. ACM, 2017.
- [5] Glenn Davis and Russ Rew. Data Management: NetCDF: An Interface for Scientific Data Access. *IEEE Computer Graphics and Applications*, 10:76–82, 1990.
- [6] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A Large-Scale Hierarchical Image Database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.
- [7] The HDF Group. Enabling a Strict Consistency Semantics Model in Parallel HDF5. <https://support.hdfgroup.org/HDF5/doc/Advanced/PHDF5FileConsistencySemantics/PHDF5FileConsistencySemantics.pdf>, 2012.
- [8] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.
- [9] Alex Krizhevsky. `cuda-convnet`. <https://code.google.com/p/cuda-convnet/>, 2012.
- [10] Alex Krizhevsky and Geoffrey Hinton. Learning Multiple Layers of Features from Tiny Images. Technical report, University of Toronto, 2009.
- [11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.
- [12] Stefan Lee, Senthil Purushwalkam, Michael Cogswell, David J. Crandall, and Dhruv Batra. Why M Heads Are Better than One: Training a Diverse Ensemble of Deep Networks. *arXiv*, 2015.
- [13] Jianwei Li, Wei keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netCDF: A High-Performance Scientific I/O Interface. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, SC '03*, pages 39–, New York, NY, USA, 2003. ACM.
- [14] He Ma, Fei Mao, and Graham W. Taylor. Theano-MPI: A Theano-Based Distributed Training Framework. *CoRR*, abs/1605.08325, 2016.
- [15] R. Thakur, W. Gropp, and E. Lusk. A Case for Using MPI's Derived Datatypes to Improve I/O Performance. In *Supercomputing, 1998.SC98. IEEE/ACM Conference on*, pages 1–1, November 1998.
- [16] Rajeev Thakur, William Gropp, and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation, FRONTIERS '99*, pages 182–189, Washington, DC, USA, 1999. IEEE Computer Society.
- [17] The Ohio State University. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. <http://mvapich.cse.ohio-state.edu>, 2014.
- [18] Theano Development Team. Theano: A Python Framework for Fast Computation of Mathematical Expressions. *arXiv e-prints*, abs/1605.02688, May 2016.
- [19] Abhinav Vishnu, Charles Siegel, and Jeffrey Daily. Distributed TensorFlow with MPI. *CoRR*, abs/1603.02339, 2016.

<sup>5</sup><https://support.hdfgroup.org/HDF5>



The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory (“Argonne”). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan. <http://energy.gov/downloads/doe-public-access-plan>.

# Towards Scalable Deep Learning via I/O Analysis and Optimization

Sirbu, Adina

01	chen ping	Page 1
9/8/2019 7:41		
02	chen ping	Page 1
9/8/2019 7:41		
03	chen ping	Page 1
9/8/2019 7:41		
04	chen ping	Page 1
9/8/2019 7:41		
05	chen ping	Page 1
9/8/2019 7:41		
06	chen ping	Page 1
9/8/2019 7:41		
07	chen ping	Page 1
9/8/2019 7:41		
08	chen ping	Page 1
9/8/2019 7:41		
09	chen ping	Page 1
9/8/2019 7:20		

10	chen ping	Page 1
9/8/2019 7:20		
11	chen ping	Page 1
9/8/2019 7:41		
12	chen ping	Page 1
9/8/2019 7:20		
13	chen ping	Page 1
9/8/2019 7:26		
14	chen ping	Page 1
9/8/2019 7:26		
15	chen ping	Page 1
9/8/2019 7:26		
16	chen ping	Page 1
9/8/2019 8:59		
17	chen ping	Page 1
9/8/2019 8:59		
18	chen ping	Page 1
9/8/2019 8:59		
19	chen ping	Page 1
9/8/2019 8:59		
20	chen ping	Page 1
9/8/2019 8:59		
21	chen ping	Page 1
9/8/2019 7:25		

22	chen ping	Page 1
9/8/2019 7:25		
23	chen ping	Page 1
9/8/2019 8:59		
24	chen ping	Page 1
9/8/2019 8:59		
25	chen ping	Page 1
9/8/2019 8:59		
26	chen ping	Page 1
9/8/2019 8:59		
27	chen ping	Page 1
9/8/2019 7:25		
28	chen ping	Page 1
9/8/2019 7:25		
29	chen ping	Page 1
9/8/2019 7:25		
30	chen ping	Page 1
9/8/2019 7:25		
31	chen ping	Page 1
9/8/2019 9:03		
32	chen ping	Page 1
9/8/2019 9:03		
33	chen ping	Page 1
9/8/2019 9:03		



34	chen ping	Page 1
9/8/2019 9:06		
35	chen ping	Page 1
9/8/2019 9:06		
36	chen ping	Page 1
9/8/2019 7:28		
37	chen ping	Page 1
9/8/2019 7:28		
38	chen ping	Page 1
9/8/2019 9:06		
39	chen ping	Page 1
9/8/2019 7:28		
40	chen ping	Page 2
9/8/2019 9:12		
41	chen ping	Page 2
9/8/2019 9:12		
42	chen ping	Page 2
9/8/2019 9:12		
43	chen ping	Page 2
9/8/2019 9:13		
44	chen ping	Page 2
9/8/2019 9:13		
45	chen ping	Page 2
9/8/2019 9:25		

46	chen ping	Page 2
9/8/2019 9:25		
47	chen ping	Page 2
9/8/2019 9:13		
48	chen ping	Page 2
9/8/2019 9:25		
49	chen ping	Page 2
9/8/2019 9:14		
50	chen ping	Page 2
9/8/2019 9:14		
51	chen ping	Page 2
9/8/2019 9:25		
52	chen ping	Page 2
9/8/2019 9:25		
53	chen ping	Page 2
9/8/2019 9:14		
54	chen ping	Page 2
9/8/2019 9:14		
55	chen ping	Page 2
9/8/2019 9:14		
56	chen ping	Page 2
9/8/2019 9:25		
57	chen ping	Page 2
9/8/2019 9:26		

58	chen ping	Page 2
9/8/2019 9:26		
59	chen ping	Page 2
9/8/2019 9:26		
60	chen ping	Page 2
9/8/2019 9:26		
61	chen ping	Page 2
9/8/2019 9:26		
62	chen ping	Page 2
9/8/2019 9:57		
这里使用过的LMDB是不是可以进行扩展研究？		
63	chen ping	Page 2
9/8/2019 9:57		
这里使用过的LMDB是不是可以进行扩展研究？		
64	chen ping	Page 2
9/8/2019 9:57		
这里使用过的LMDB是不是可以进行扩展研究？		
65	chen ping	Page 2
9/8/2019 9:26		
66	chen ping	Page 2
9/8/2019 9:26		
67	chen ping	Page 2
9/8/2019 9:26		
68	chen ping	Page 2
9/8/2019 9:14		

69	chen ping	Page 2
9/8/2019 9:26		
70	chen ping	Page 2
9/8/2019 9:19		
标准的做法		
71	chen ping	Page 2
9/8/2019 9:19		
标准的做法		
72	chen ping	Page 2
9/8/2019 9:19		
标准的做法		
73	chen ping	Page 2
9/8/2019 9:19		
74	chen ping	Page 2
9/8/2019 9:19		
75	chen ping	Page 2
9/8/2019 9:19		
76	chen ping	Page 2
9/8/2019 9:29		
77	chen ping	Page 2
9/8/2019 9:29		
78	chen ping	Page 2
9/8/2019 9:29		
79	chen ping	Page 2
9/8/2019 9:21		



80	chen ping	Page 2
9/8/2019 9:21		
81	chen ping	Page 2
9/8/2019 9:21		
82	chen ping	Page 2
9/8/2019 9:30		
83	chen ping	Page 2
9/8/2019 9:30		
84	chen ping	Page 2
9/8/2019 9:30		
85	chen ping	Page 2
9/8/2019 9:59		
LMDB使用mmap来将数据map到内存中。		
86	chen ping	Page 2
9/8/2019 9:59		
LMDB使用mmap来将数据map到内存中。		
87	chen ping	Page 2
9/8/2019 9:59		
LMDB使用mmap来将数据map到内存中。		
88	chen ping	Page 2
9/8/2019 10:01		
类似于一种内存缓冲区！		
89	chen ping	Page 2
9/8/2019 10:01		
类似于一种内存缓冲区！		

90	chen ping	Page 2
	9/8/2019 10:01 类似于一种内存缓冲区！	
91	chen ping	Page 2
	9/8/2019 10:00	
92	chen ping	Page 2
	9/8/2019 10:00	
93	chen ping	Page 2
	9/8/2019 10:00	
94	chen ping	Page 2
	9/8/2019 10:02	
95	chen ping	Page 2
	9/8/2019 10:02	
96	chen ping	Page 2
	9/8/2019 10:02	
97	chen ping	Page 2
	9/8/2019 10:03	
98	chen ping	Page 2
	9/8/2019 10:03	
99	chen ping	Page 2
	9/8/2019 10:03	
100	chen ping	Page 2
	9/8/2019 10:03	

101	chen ping	Page 2
9/8/2019 10:03		
102	chen ping	Page 2
9/8/2019 10:03		
103	chen ping	Page 2
9/8/2019 10:03		
104	chen ping	Page 2
9/8/2019 10:03		
105	chen ping	Page 2
9/8/2019 10:03		
106	chen ping	Page 2
9/8/2019 10:21		
107	chen ping	Page 2
9/8/2019 10:21		
108	chen ping	Page 2
9/8/2019 10:21		
109	chen ping	Page 2
9/8/2019 10:21		
110	chen ping	Page 2
9/8/2019 10:21		
111	chen ping	Page 2
9/8/2019 10:31		
处理器可以自由的访问这个buffer		

112	chen ping	Page 2
	9/8/2019 10:31 处理器可以自由的访问这个buffer	
113	chen ping	Page 2
	9/8/2019 10:31 处理器可以自由的访问这个buffer	
114	chen ping	Page 2
	9/8/2019 10:21	
115	chen ping	Page 3
	9/8/2019 10:32 文件不在buffer的情况	
116	chen ping	Page 3
	9/8/2019 10:32 文件不在buffer的情况	
117	chen ping	Page 3
	9/8/2019 10:32 文件不在buffer的情况	
118	chen ping	Page 3
	9/8/2019 10:22	
119	chen ping	Page 3
	9/8/2019 10:22	
120	chen ping	Page 3
	9/8/2019 10:22	
121	chen ping	Page 3
	9/8/2019 10:26	



122	chen ping	Page 3
9/8/2019 10:26		
123	chen ping	Page 3
9/8/2019 10:26		
124	chen ping	Page 3
9/8/2019 10:27		
125	chen ping	Page 3
9/8/2019 10:27		
126	chen ping	Page 3
9/8/2019 10:27		
127	chen ping	Page 3
9/8/2019 10:54		
128	chen ping	Page 3
9/8/2019 10:54		
129	chen ping	Page 3
9/8/2019 10:54		
130	chen ping	Page 3
9/8/2019 11:10		
131	chen ping	Page 3
9/8/2019 11:10		
132	chen ping	Page 3
9/8/2019 11:10		
133	chen ping	Page 3
9/8/2019 11:10		

134	chen ping	Page 3
9/8/2019 11:10		
135	chen ping	Page 3
9/8/2019 11:10		
136	chen ping	Page 3
9/8/2019 11:11		
137	chen ping	Page 3
9/8/2019 11:11		
138	chen ping	Page 3
9/8/2019 11:11		
139	chen ping	Page 3
9/8/2019 11:11		
140	chen ping	Page 3
9/8/2019 11:11		
141	chen ping	Page 3
9/8/2019 11:11		
142	chen ping	Page 3
9/8/2019 11:12		
143	chen ping	Page 3
9/8/2019 11:12		
144	chen ping	Page 3
9/8/2019 11:12		
145	chen ping	Page 3
9/8/2019 11:12		

146	chen ping	Page 3
	9/8/2019 11:12	
147	chen ping	Page 3
	9/8/2019 11:12	
148	chen ping	Page 3
	9/8/2019 11:30	
149	chen ping	Page 3
	9/8/2019 11:30	
150	chen ping	Page 3
	9/8/2019 11:30	
151	chen ping	Page 3
	9/8/2019 11:13	
152	chen ping	Page 3
	9/8/2019 11:13	
153	chen ping	Page 3
	9/8/2019 11:14	
	这里存在由于此事件发送的消息是一个通用类的，所以所有的处理器都要被唤醒。浪费资源	
154	chen ping	Page 3
	9/8/2019 11:14	
	这里存在由于此事件发送的消息是一个通用类的，所以所有的处理器都要被唤醒。浪费资源	
155	chen ping	Page 3
	9/8/2019 11:14	
	这里存在由于此事件发送的消息是一个通用类的，所以所有的处理器都要被唤醒。浪费资源	
156	chen ping	Page 3
	9/8/2019 11:13	

157	chen ping	Page 3
9/8/2019 11:05		
158	chen ping	Page 3
9/8/2019 11:05		
159	chen ping	Page 3
9/8/2019 11:05		
160	chen ping	Page 4
9/8/2019 11:16		
缺点暴露了！		
161	chen ping	Page 4
9/8/2019 11:16		
缺点暴露了！		
162	chen ping	Page 4
9/8/2019 11:16		
缺点暴露了！		
163	chen ping	Page 4
9/8/2019 11:16		
164	chen ping	Page 4
9/8/2019 11:16		
165	chen ping	Page 4
9/8/2019 11:16		
166	chen shuaiben	Page 4
15/8/2019 11:53		
167	chen shuaiben	Page 4
15/8/2019 11:53		



168	chen shuaiben	Page 4
	15/8/2019 11:53	
169	chen ping	Page 4
	9/8/2019 11:23	
170	chen ping	Page 4
	9/8/2019 11:23	
171	chen ping	Page 4
	9/8/2019 11:23	
172	chen ping	Page 4
	9/8/2019 11:24	
173	chen ping	Page 4
	9/8/2019 11:24	
174	chen ping	Page 4
	9/8/2019 11:24	
175	chen ping	Page 4
	9/8/2019 11:24	
176	chen ping	Page 4
	9/8/2019 11:24	
177	chen ping	Page 4
	9/8/2019 11:24	
178	chen shuaiben	Page 5
	15/8/2019 11:53	
179	chen shuaiben	Page 5
	15/8/2019 11:53	

180	chen shuaiben	Page 5
15/8/2019 11:53		
181	chen shuaiben	Page 5
15/8/2019 11:54		
182	chen shuaiben	Page 5
15/8/2019 11:54		
183	chen shuaiben	Page 5
15/8/2019 11:54		
184	chen shuaiben	Page 7
15/8/2019 11:53		
185	chen shuaiben	Page 7
15/8/2019 11:53		
186	chen shuaiben	Page 7
15/8/2019 11:53		
187	chen shuaiben	Page 8
15/8/2019 11:54		
188	chen shuaiben	Page 8
15/8/2019 11:54		
189	chen shuaiben	Page 8
15/8/2019 11:54		