NVM Compression -Hybrid Flash-Aware Application Level Compression

Conference Paper · October 2014

CITATIONS
12

READS
137

5 authors, including:

Dhananjoy Das
SanDisk
5 PUBLICATIONS **25** CITATIONS
SEE PROFILE

Dulcardo Arteaga
Florida International University
11 PUBLICATIONS **75** CITATIONS
SEE PROFILE

Nisha Talagala
Fusion-io
27 PUBLICATIONS **339** CITATIONS
SEE PROFILE

Jan Lindström
MariaDB Corporation
34 PUBLICATIONS **234** CITATIONS
SEE PROFILE

NVM Compression - Hybrid Flash-Aware Application Level Compression

Dhananjoy Das[1], Dulcardo Arteaga[2], Nisha Talagala[1], Torben Mathiasen[1], and Jan Lindström[3]

[1]{*Dhananjoy.Das, Nisha.Talagala, Torben.Mathiasen*}*@SanDisk.com*

[2]*darte003@fiu.edu (Intern at SanDisk)*

[3]*Jan.Lindstrom@skysql.com*

ABSTRACT

This paper describes NVM Compression, a novel hybrid technique that combines application level compression with flash awareness for optimal performance and storage efficiency. Utilizing new interface primitives exported by Flash Translation Layers (FTLs), we leverage the garbage collection available in flash devices to optimize the capacity management required by compression systems. We implement NVM Compression in the popular open source database MariaDB based on Oracle MySQL™and use variants of commonly available POSIX file system interfaces to provide the extended FTL capabilities to the user space application. The experimental results show that the hybrid approach of NVM Compression can improve compression performance by 2-3x, deliver compression performance for flash devices that is within 5% of uncompressed performance (and sometimes exceed uncompressed performance due to less data writes), improve storage efficiency by 19% compared to legacy Row compression method, reduce data writes by up to 4x when combined with other flash aware techniques such as Atomic Writes, and deliver further advantages in power efficiency and CPU utilization.

1. INTRODUCTION

Data compression is a well known capacity optimization method and used extensively in storage systems. Compression has historically been applied at various layers in the I/O stack, from user space applications like databases [26] to some file systems [35, 11, 30, 8, 29] and block level implementations [20]. Some storage devices also have built in compression using both software and hardware methods to accelerate compression operations [5]. Compression can arguably be best suited for application implementation, as the data layout and application access pattern details are well known [16, 17, 28, 15, 18, 40]. However, compression designs can add complexity since variable data sizes must be accommodated. Implementing compression at application level therefore implies that the application must now take on the burden of variable data size management in addition to its other duties.

Compression is also extremely attractive for flash based systems to both improve cost/capacity and to improve lifetime by reducing media writes. In the case of flash specifically, the Flash Translation Layer (FTL) is an appealing location to integrate compression due to its already sophisticated data management [25, 37]. A Flash Translation Layer (FTL) is responsible for the mapping of Logical Block Address (LBA) to physical block locations. One can thus extend existing mapping structures and garbage collection to perform the block management required by compression. While compression integrated solely at an FTL or within a flash device can be ideal for flash, the benefits of application level compression that come from better understanding of application patterns can be lost.

In this paper we explore NVM Compression, a flash aware high speed compression technique. NVM Compression employs a hybrid design that allows applications to retain control of compression techniques while integrating FTL exported primitives to gain the benefits that can come from flash awareness. We explore this hybrid design in the context of the MySQL open source relational database. The legacy MySQL compression was designed for spinning media. When applied to flash, several issues emerge which have resulted in poor performance and limited adoption of MySQL compression across the broad MySQL deployment base. We seek to improve MySQL's compressed performance on flash with NVM Compression.

The paper makes the following contributions:

- We describe the design and implementation of a novel compression mechanism that combines the advantages of application level awareness and FTL integration.
- We describe ways to integrate such an approach into an operating system stack through a combination of new interface primitives and file system support.

INFLOW '14, October 5, Broomfield, CO USA.

- We include a detailed performance study showing that this approach can improve compression performance by 2-3x, deliver compression performance for flash devices that is within 5% of uncompressed performance (and sometimes exceed uncompressed performance due to less data writes),improve storage efficiency by 19% compared to legacy Row compression method, reduce data writes by upto 4x when combined with other flash aware techniques such as Atomic Writes, and deliver further advantages in power efficiency and CPU utilization.

The rest of the paper is organized as follows. In Section 2 we describe Background and Related Work. Section 3 outlines the existing compression design in MySQL and our high level approach. Section 4 describes NVM Compression in detail. Section 5 contains a performance evaluation. Section 6 discusses the results and future work. Section 7 provides a brief summary. Section 8 concludes with acknowledgments for the paper.

2. BACKGROUND

Compression has been widely studied and implemented in storage systems, memory systems and many forms of applications [16, 17, 28, 15, 18, 40]. In this section we focus on two areas of background, compression implementations leveraging Flash Translation Layers, and the existing default compression implementation in MySQL, our target for our hybrid NVM Compression work.

2.1 FTL Integrated Compression

Some flash devices implement compression internally [5] and a number of studies have demonstrated how compression can be integrated into the FTL [25, 37]. These studies show that the indirection maps inside the FTL can be a convenient place to add the block management required for compression. They also show that that data compression can reduce data writes and improve write amplification. However, choosing an appropriate compression block size can be difficult since the application knowledge is not accessible at the device level and all types of blocks look identical. Larger block sizes generate more efficient compression but can cause excessive reads or add garbage collection cost if the application access size turns out to be smaller than the compression block size chosen by the FTL. Internal fragmentation can also be a problem if the application I/O pattern does not match the assumptions made by the FTL when mapping compressed blocks into flash pages.

2.2 MySQL ROW-Compression

In contrast to FTL based compression, application level compression presents a different set of benefits and challenges. We explore these through a detailed description of MySQL's existing compression mechanism, Row-Compression.

MySQL is a very popular open source database. It provides rich set of RDBMS features and supports various different storage-engine options. InnoDB and XtraDB are the two most popular storage engines used by the MySQL community today. MySQL data records are stored in one or more files in preconfigured page size units with the default

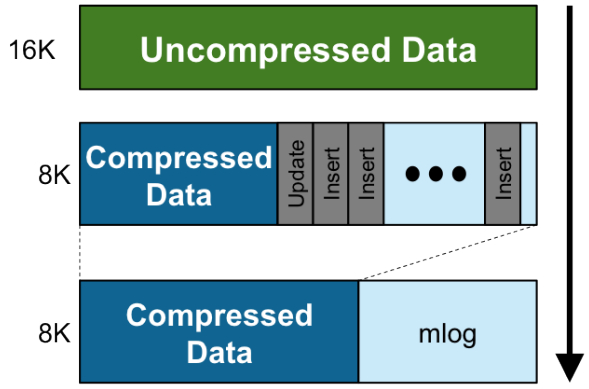

Figure 1: Traditional MySQL row-compression

at 16KB per page. As shown in Figure 1, Row Compression compresses data rows by replacing repeated patterns across the rows with shorter symbols. Uncompressed 16KB pages are stored in memory in the buffer pool while compressed pages are stored in both memory and on media. Each of the compressed blocks contains compressed data and a Modify-log-region where further block changes (inserts and updates) are logged. The size of the compressed block is set by an admin command and is fixed at database table-create time. As the table content changes, deltas are appended to the *mlog* until it runs out of space, at which point the compressed data is de-compressed, the *mlog* entries applied, and the resulting data-block is re-compressed.

The re-compression operation opens up the possibility of an Compression Insert Failure, where the newly compressed block is too big to fit in the predefined fixed compressed block size. This failure leads to a node (page) split where attempts are made to fit this block into an alternate location in the block map. This can lead to a need to rebalance the block map. Figure 2 shows the steps involved to handle a compression insert failure. First, an adjacent page needs to be read and decompressed. Entries are then inserted into these pages and the allocation maps are rebalanced until all the data is successfully inserted.

Depending on the workload and the frequency of insert failures, performance may suffer since insert failures consume CPU cycles and add other data management overhead. This complexity leads to much of the performance issues seen today in MySQL Row-Compression deployments. Substantial work has been done in the last several years to attempt to solve this problem [23, 12, 13, 27].

3. NVM COMPRESSION: APPROACH

The goals of our compression solution, NVM Compression, are to deliver high speed and high efficiency flash compression that combines the benefits of application level data knowledge and the FTL's awareness and operation of flash management. Application level solutions benefit from knowing the meaningful units of data access. In the case of MySQL, this is a database block. FTLs benefit from the inherent presence of an indirection map that translates log-

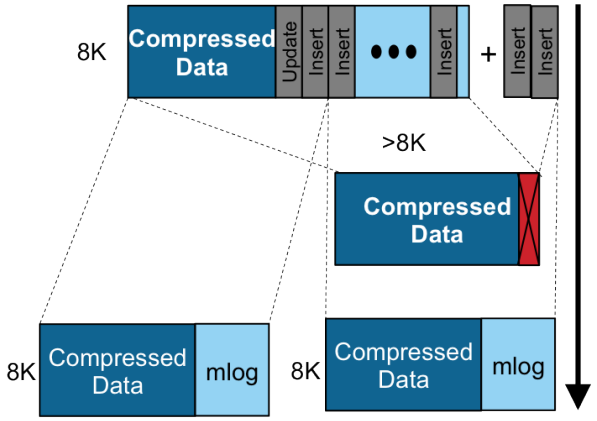

Figure 2: Row Compression: Insert Failures

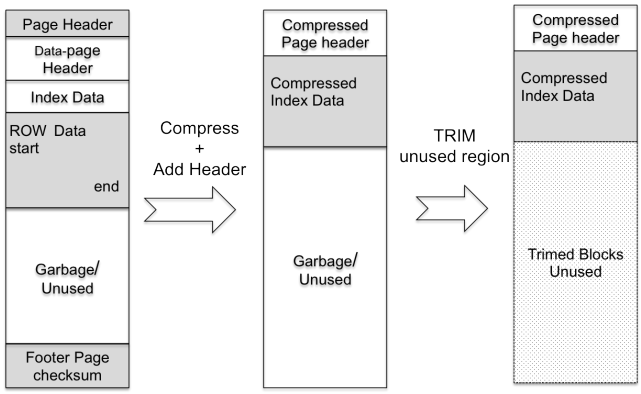

Figure 3: NVM Compression on Spare Address Space

ical addresses to physical addresses, and the existing high performance garbage collection that coalesces available free space. To combine these benefits, we create an architecture where the compression is performed at the application level while the free space and the compressed data block are managed at the FTL level.

Figure 3 shows the high level approach. The database operates on a virtual storage space which is always the size of the uncompressed data. Data is compressed at the natural application unit - i.e. a database block. As data is compressed, it is written back in place at the same virtual address, leaving a "hole", i.e. an empty location, in the remainder of the space allocated for the fully uncompressed block. For example, if a 16KB database block was expected to be stored at storage address A, and after compression becomes 3.5KB, 3.5KB of data will be stored at address A, and the remaining 12.5KB of allocated virtual space will contain a hole and be empty.

NVM compression model relies upon the FTL to perform thin provisioning, i.e. allocate physical capacity only to the populated virtual addresses. The virtual address "holes" are unpopulated, i.e. no physical media blocks are mapped to the unpopulated virtual address space. This is done by exporting a "Sparse Address Space" from the FTL similar to that used in [19, 31, 36, 22, 33]. Under this model, the FTL will have populated virtual addresses and some empty virtual addresses. FTL garbage collection will naturally coalesce the populated addresses, thus allow for re-provisioning of the available physical space to be used for new writes.

As database records are overwritten, the size of compressed data content can change. In particular, blocks may compress to smaller sizes than they did previously. Since, in the NVM Compression design, re-compressed blocks are always rewritten to the same virtual locations, it becomes necessary to "punch a hole" in the address space, i.e. create a hole in the virtual address space where previously there may have been data. In our above example, if, after record updates, the database block which was previously 3.5KB (7 512B sectors) now compresses to 2.5KB (5 512B sectors), the operations will be as follows. First the updated content will be written to addresses A through A+4, reflecting the new 2.5KB or 5 sectors of data. Second, a hole will be punched at address A+5 to A+6, reflecting the fact that those virtual addresses no longer contain valid data. Once this is done, the FTL can then garbage collect the 1KB of data previously stored to those locations.

To benefit from the FTL thin provisioning as discussed above, we need a way to expose that capability to layer above, in this case, a user space application:

- We expose native characteristics of flash translation layers through a series of primitives that are exported by the FTL. This is similar in approach to other flash specific optimizations for KV stores, File Systems and Caches as described in [19, 31, 36, 22].
- We modify the MySQL database to utilize these flash primitives to create a new compression design.
- In order to allow the database to operate on regular files (as is the norm for nearly all MySQL deployments), we modify a file system to effectively export the primitives from the flash translation layer to the user space application.
- To ease the integration process, we also map these primitives to existing Linux system calls.

Specifically, NVM Compression uses three FTL exposed primitives: a Sparse address space, Persistent TRIMs, and Atomic writes. These flash characteristics are then exported by the NVMFS file system (Non-Volatile Memory File System)[19, 34], a POSIX compliant file system that is developed specifically to efficiently access flash and export native flash control to user space applications. NVMFS is an extension of the work done in [19]. MySQL is then modified to support a new compression mode, NVM compression, which uses the primitives on files stored in the NVMFS file system.

Prior work in MySQL had utilized a flash specific primitive, Atomic Writes, to remove double writes to database files and improve performance and endurance. NVM Compression

builds upon this work which is described in more detail in [24].

4. NVM COMPRESSION: DESIGN

As described in the previous section, NVM Compression design has three elements:

- The primitives exported by the FTL that enable access to its thin-provisioning.
- Export primitives through NVMFS file system.
- Use of the primitives in an application level compression engine.

Figure 4 shows the layered architecture. The FTL exports a sparse address space for thin provisioning. The NVMFS file system then offers sparse files to the user space application, in addition FTL primitives are plumbed to user space through system calls (existing Linux system calls). The MySQL storage engine then implements compression using these capabilities.

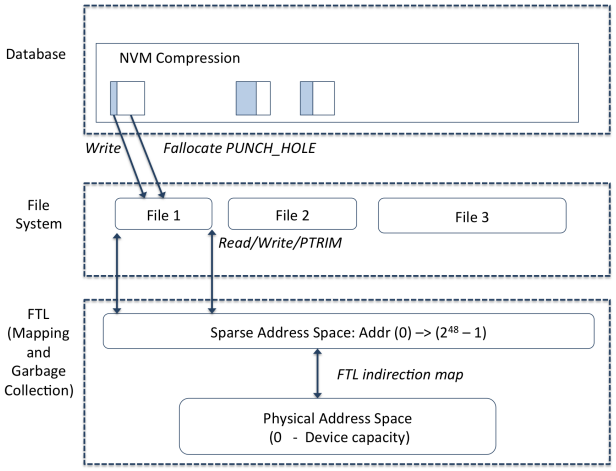

Figure 4: NVM Compression Design

4.1 FTL Primitives

The primitives exported by the FTL enable upstream software to use its indirection maps and garbage collection for variable length data management. The exported capabilities, summarized in table 1, are the sparse address space and the commands to allocate and delete contents of virtual addresses. We also leverage previous work in Atomic Writes to further improve efficiency and performance.

The sparse address space exported by the FTL works as follows. While SSDs normally expose a block device of the same size as the SSD capacity, a sparse address space allows the user to store data at a much larger range of addresses than the physical capacity allows. In our specific implementation, the FTL exposes an address space of 2PB over a physical space of single digit TB. Data is inserted at specified address by using a conventional *WRITE()* operation and deleted using a persistent trim operation *PTRIM()*.

Primitive	Detail
Sparse	Sparse address space allows consuming applications to offload data allocation management to the FTL.
PTRIM	Persistent TRIMs enable guaranteed deletes of a data block at a given virtual address.
Atomics	Atomic-write guarantees transactional writes, removing the need for applications to double buffer writes to ensure atomicity.

Table 1: NVM Primitives

System Call Interfaces	Functionality
fallocate(offset, len)	Pre-allocation and extending files/table space.
fallocate(PUNCH_HOLE)	Unmap/Punch hole operation (issue persistent TRIMs).
io_submit()	AIO Transparent atomic writes.

Table 2: System Call Interfaces

4.2 File System Export of Primitives

The above mentioned primitives are exported by the FTL as low level block device IOCTLs. The NVMFS file system re-exports the functionality as file level interfaces as described in Table 2. NVMFS [19] is a POSIX compliant file system that relies upon the same primitives described in Table 1. In this paper we only focus on the elements of NVMFS that support NVM Compression. Other design details of NVMFS can be found here [34].

The sparseness required for NVM Compression is provided through sparse files in NVMFS. The files are pre-allocated to be the size required for uncompressed data. As data is compressed during database operation, holes are created within the file through the *fallocate(PUNCH_HOLE)* operation.

Its important to note that the NVM Compression design does not specifically require NVMFS and also work on other file systems such as EXT4 and XFS. However, NVMFS is able to process operations like Persistent TRIM very efficiently since its design is also based on the FTL sparse address space (see Figure 5). NVMFS offloads the complexity of remapping and translation to the FTL, leveraging the capabilities of the underlying flash device. This architecture implies NVMFS does not need any additional data operations to optimize file system layouts for sparse files.

NVMFS exports the Persistent TRIM capability through the *fallocate(PUNCH_HOLE)* Linux system call operation. This operation is converted by NVMFS into a PTRIM primitive to the underlying FTL. Atomic Writes are exported as conventional writes which are configured to be atomic by default when issued to specific files. Such writes issued by the application are passed by NVMFS directly into the FTL as atomic operations.

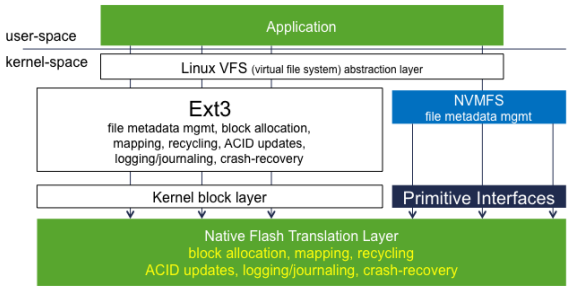

Figure 5: Non-Volatile Memory File System (NVMFS)

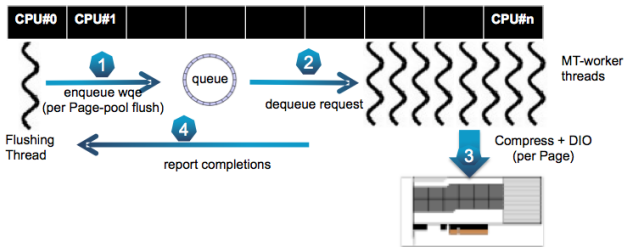

Figure 6: MT-Flush Framework

4.3 MySQL NVM Compression Design

The final component of NVM Compression is within the InnoDB/XtraDB storage engine. Unlike Row Compression where deltas are applied within the page in an Mlog, in NVM Compression mode, InnoDB compresses the page as a whole and stores it on media in a new page format. Much like the uncompressed format, the NVM Compression format includes checksums, page numbers, log sequence numbers etc. A new field is defined to indicate the page is "page compressed" and to indicate which compression algorithm is used (we have implemented LZ4 and LZ0 in addition to the LZ77 originally used by Row Compression) [38, 39].

The compressed page is stored with the resulting size in the virtual address allocated to the uncompressed block and a PTRIM operation is issued for the remainder of the address range (using *fallocate(PUNCH_HOLE)*) to inform NVMFS and the FTL that the remainder of the virtual addresses are now empty.

4.3.1 TRIM operation

As data becomes compressed and holes are generated in virtual space, PTRIM operations are issued to the flash device through NVMFS using the *fallocate(PUNCH_HOLE)* operation. This is done in the asynchronous I/O callback handler for the database block write. We study this aspect of NVM Compression performance further in the Evaluation in Section 5.

4.3.2 MT-Flush operation

The MySQL storage engines cache database blocks in an in-memory buffer pool. The buffer-pool hosts parts of the on-flash table-spaces and the system-space and is key to the performance of query and update requests. Buffer pool pages are flushed asynchronously using POSIX AIO, with various flush selection policies in effect including LRU and percentile dirty policies.

NVM page compression is designed to enable compression only at point of dirty page flush. Since compression can add to the latency of an individual block flush, the Multiple Threaded Flush (MT-Flush) framework was added to improve page flushing performance. The framework considers the number of available CPU cores and splits the buffer pool across a series or worker threads to optimally perform flush operation in parallel (and consequent data compression) across all available CPU cores. Figure 6 depicts the producer consumer framework used to perform parallel compress and flush operations.

4.3.3 Double Write Buffer

Finally, using NVMFS exported atomic writes, MySQL InnoDB flushes dirty data only once to media in an atomic operation eliminating usage of the double write buffer. Atomic writes in MySQL is covered in detail here [1].

5. EVALUATION

In this section we evaluate NVM Compression across various metrics ranging from performance to energy efficiency.

5.1 System level Benchmarks

5.1.1 LinkBench

LinkBench [7] is a database benchmark developed to emulate the workload of Facebook's production MySQL deployment. Linkbench conducts a range of operations to add, modify, delete, retrieve and perform other operations on graph nodes. The default LinkBench dataset (referred to as a $1x$ workload) comprises 10 million node ids with a dataset size of 10GB on media. A dataset with 100 million nodes is referred to as a $10x$ workload. Since the $10x$ workload allows only a partial amount of the data to fit in the buffer pool, it is a more realistic test of the I/O subsystem. The Linkbench experiments were conducted on dual socket server with 12 physical 3.4GHz cores (24 with HT-enabled) and 128GB RAM, running RHEL 6.4 kernel 2.6.32. The flash device used is a *SanDisk ioDrive*.

Figure 7 shows the throughput for the Linkbench $10x$ workload with NVM Compression compared to both Row Compression and Uncompressed. The results show a $2.25x$ improvement using NVM Compression compared to the default Row Compression. We can also see that the throughput of NVM Compression is within 5% of the throughput of Uncompressed.

Figure 8 and 9 shows the average and 99th percentile latencies for each Linkbench operation type when run with a $10x$ workload. The average latencies of NVM Compression are generally comparable to those of Uncompressed. Row Compression throughput is much lower compared to NVM Compression and Uncompressed. The 99th percentile latencies for NVM Compression are $2.5x$ to $5.5x$ lower than that for the default Row Compression. The NVM Compression latencies are also close to that of the Uncompressed workload, with lower latencies than the Uncompressed workload

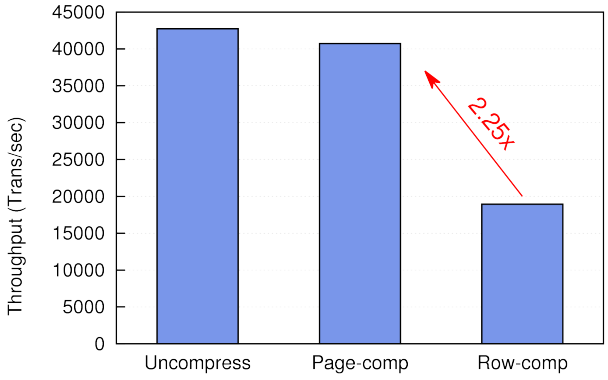

Figure 7: Linkbench $10x$ Throughput

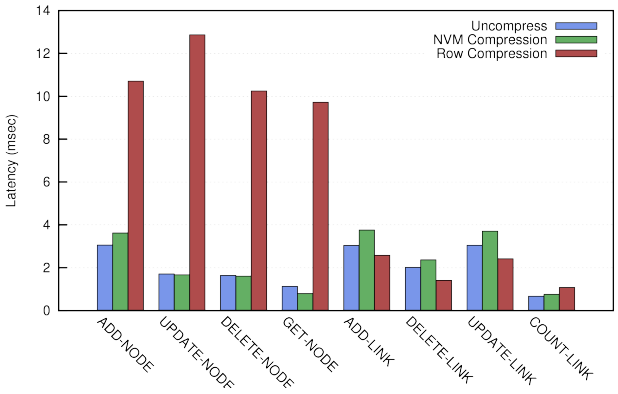

Figure 8: Average Latencies ($10x$ Workload)

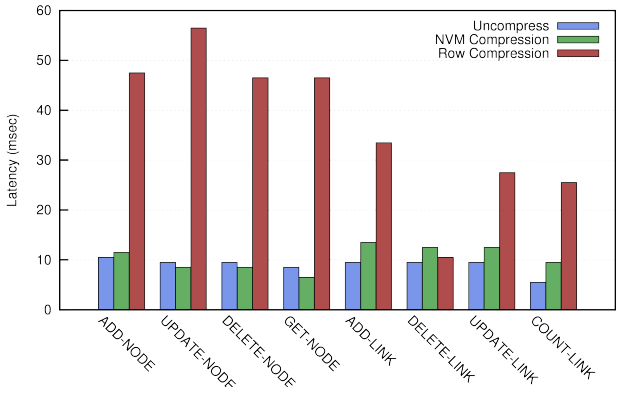

Figure 9: 99^{th}percentile Latencies ($10x$ Workload)

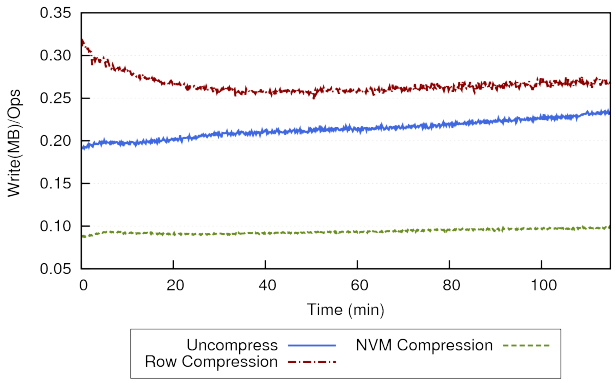

Figure 10: Cost of Write MB/OPs ($10x$ Workload: Buffer pool 50GB)

for some operations (Update Node, Deleted Node and Get Node).

The above results confirm that the hybrid NVM Compression approach can deliver benefits by removing some of the complexities inherent in the Row Compression scheme. However, NVM Compression is also able to keep up with, and occasionally outperform, the Uncompressed configuration. To better understand this, we measured the write behavior of all three configurations. Figure 10 highlights the data written per linkbench operation measured across 30 second intervals during a six hour run. NVM Compression writes far less data than uncompressed, which leads to better performance in the flash device over time since less garbage collection has to occur. We also found, that although Row-Compression stores less total data than Uncompressed, it also generates more data writes per unit of operation than Uncompressed or NVM Compression, which likely is a result of insert failures leading to node splits and allocation map rebalance 2 requiring additional writes thereby further contributes to its poor performance.

5.1.2 OLTP Workload

Next we study compression performance for an On-Line Transaction Processing (OLTP) benchmark. This TPCC-like workload involves a mix of five concurrent transaction types executed on-line or queued for deferred execution. The database is comprised of nine tables with a wide range of record and population sizes. Results are measured in terms of transactions per minute (tpmC). Tests were performed on a dual socket server with 16 2.4 GHz cores (32 with HT-enabled) and 128GB RAM running RHEL 6.4 kernel 2.6.32. The flash device used is a *SanDisk ioDrive*. MariaDB was configured to use a 75GB buffer pool with 1000 warehouses. The test ran for 1 hour with measurements every 10s.

The results are as seen in Figure 11. We observed 86% improvement in new order transaction performance between Row compression and NVM compression. The difference between NVM Compression and Uncompressed was 12% for the relatively short run.

5.2 Storage Capacity Efficiency

Storage efficiency is extremely dependent on the data content. However we can make some general observations regarding NVM Compression storage efficiency as compared to Row Compression. Figure 12 shows a sample of the storage efficiency of the two compression techniques when running the Linkbench 10x workload. In this case, NVM Compression generates 19% better storage efficiency. Since the compression algorithm used is the same in both cases (LZ77),

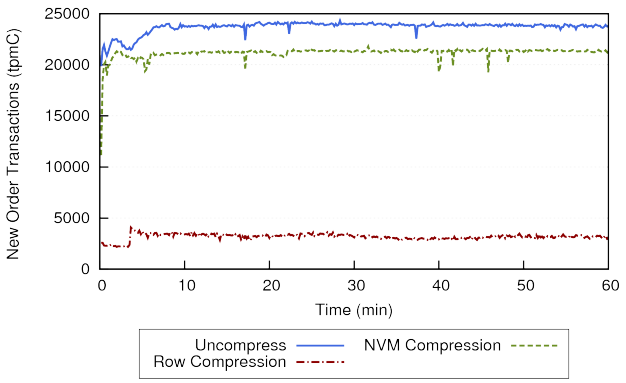

Figure 11: TPCc-like 1000 warehouses

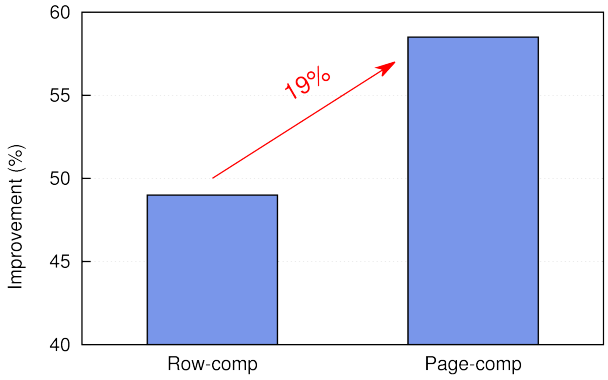

Figure 12: Storage Savings

and the starting database block size and block content is the same, the benefit comes from the design difference. Row Compression forces a compressed block size of 8KB for a 16KB database block, limiting the maximum possible storage efficiency to 2x. NVM Compression will allow block sizes as small as a single 512B sector. Given this, NVM Compression will always have storage efficiencies as good or better than Row Compression.

5.3 Power and CPU Efficiency

Recent reports [6, 21] show the importance of lowering energy consumption and its role in managing Total Cost of Ownership (TCO) [9]. Both direct power consumption and cooling costs contribute to TCO. For example [10] argues for energy consumption proportional to work accomplished. Since NVM-Compression enables lower data volumes to be written to flash, there is an opportunity to reduce power consumption through both lower application writes to the flash device and lower garbage collection writes within the flash device.

SanDisk cards have internal statistics that capture the bus-watts consumed and internal-temp of the card. Using these stats, we compute the temperature and power consumption for both NVM-Compressed and Uncompressed workloads.

Figure 13 shows the normalized data using z-score of the

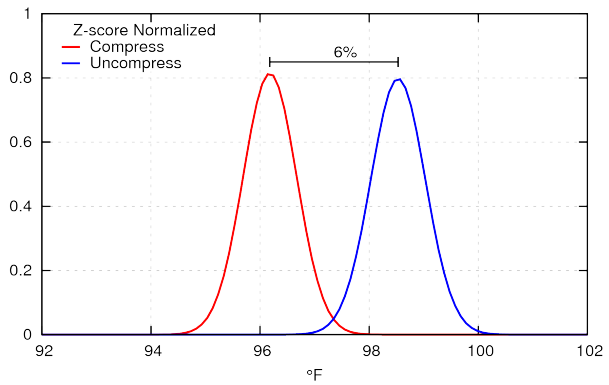

Figure 13: Normalized Temperature

	Watt Percentile	Temp Avg-F	OP/sec
Page Compression	56.3^{th}	96.18	60,526
Uncompress	61.1^{th}	98.53	57,045

Table 3: Power Consumption

current temperature of the flash drives during the workload run. When NVM compression is used, the average temperature is reduced up to 16% as compared to Uncompressed while the performance is nearly the same.

Table 3 shows a brief summary of the power consumed when running the same Linkbench 10x workload for six hours. NVM-Compressed consistently outperforms UnCompressed by a small amount, 6% OP/sec. Meanwhile, the Uncompressed workload consumes slightly more power and dissipates slightly more heat, about 3-16%.

We also measured the CPU utilization of both runs, as a means of assessing the power efficiency of the rest of the system under each workload. The CPU utilization was similar on both workloads, with the uncompressed generating slightly higher CPU consumption. This is likely due to the trade-off between CPU consumption for compression computations vs. CPU consumption for additional write activity generated by the uncompressed workload.

5.4 File System Impact

Due to lack of time we were unable to repeat the above experiments on file systems other than NVMFS. However - in order to understand the potential value of the NVMFS approach to exporting sparse and primitives, we performed some micro-benchmark experiments. The micro-benchmark evaluates the performance of different file systems when a workload like MySQL NVM compression is running on top of them. This workload comprises series of writes of size 16KB (default page size) follow by trim operation of smaller size.

We evaluate NVMFS, XFS, and EXT4 on a microbenchmark that simulates the I/O patterns of NVM Compression. The test fills the entire file with zeros and then spawns two threads with the first thread issuing a series of 16KB writes

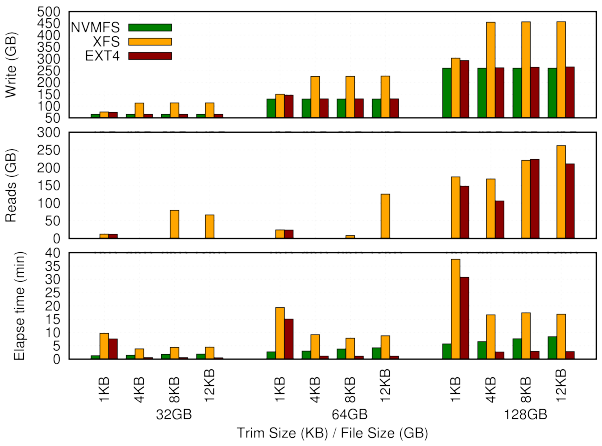

Figure 14: File System Impact

and the second thread issuing a series of trims of size 1KB to 12KB. Figure 14 shows the total runtime of the benchmark. We can see that NVMFS performance increases linearly proportional to the size of the trim operation. The other file systems perform poorly when trim size is 1KB because they do read modify writes for trims smaller than 4KB. The figure also shows the amount of reads generated to the flash device. XFS and EXT4 perform some read operations due to read modify trims while NVMFS does not show any reads at all. The graph with the amount of writes to the physical device, we can see the fact NVMFS issues less amount of writes for the same number of TRIM operations. This data suggests that NVMFS performs TRIMs more efficiently than EXT4 and XFS.

6. DISCUSSION

The measurement results show that NVM Compression can substantially outperform Row Compression and in many cases deliver performance similar to that of Uncompressed. NVM Compression also opens up a number of other optimization opportunities which we have yet to explore.

First, the only aspect of application awareness which is leveraged in NVM Compression is knowledge of the application block size. The MySQL implementation for the NVM Compression is a thin layer at the very bottom of the database storage engine with the rest of the database being largely agnostic. While the paper focused on the InnoDB implementation, all three primary forms of MySQL (SkySQL, Oracle and Percona) have now integrated NVM Compression into their respective distributions [2, 3, 4], demonstrating that the technique is extensible beyond InnoDB. We believe that the technique should be extensible to other apps beyond MySQL since most applications have a well defined notion of storage block size and block format. It may also be possible to integrate a single NVM Compression capability into the I/O stack with appropriate hints from applications.

Another interesting and encouraging observation is that the FTL Primitives leveraged for NVM Compression have also been used to optimize KV stores, file systems and caches for flash [19, 31, 36, 22]. These primitives are also included in a number of pending standards proposals [14, 32] here. The related work also covers details on different ways to implement such primitives within an FTL and demonstrates that costs within the FTL are far less than the overall benefits achieved.

Our approach involved using FTL primitives which were re-exported to the user space application by the file system through existing Linux system calls. This simplified application development, but also opens up several possibilities. Other file systems can support the existing NVM Compression code in MySQL simply by offering suitably efficient implementations of these system calls.

In fact, a file system only implementation, where the FTL offers no primitives, is also possible. The MySQL portion of NVM Compression will also function on devices that do not provide all of the primitives, albeit at reduced efficiency and at some performance cost. Since compression generates variable block sizes, the block management has to be performed at one of the three layers. The MySQL component of NVM Compression can also function above sparse files offered by file systems that do their own block management. This alternative is less efficient since the file system would then perform redundant work above the FTL. By using NVMFS which itself relies upon the FTL for block management, we optimize each layer for best combined efficiency.

Finally, the decoupling of compression from capacity management suggests that different compression algorithms can be intermixed within the same database with relative ease since the same generic FTL garbage collection techniques would manage the capacity. We believe this is a fruitful area for future work.

7. SUMMARY

NVM-Compression is designed to combine the best of application level compression and flash aware integration. The compression itself is performed at application level, enabling application workload intelligence, like ideal block size, to be used. The block management and high speed garbage collection of flash are leveraged through FTL primitives. File system support ensures that standard database deployment practices, such as placing tables in files, are preserved. The performance results are dramatic, with performance close to or sometimes exceeding uncompressed on both Linkbench and OLTP workloads. NVM Compression used in combination with Atomic Writes also improves endurance by about $4x$.

NVM compression has been released by all three MySQL distributions, MariaDB, Percona and Oracle and is available at [2, 4, 3]. The code for the MySQL storage engine implementations of NVM Compression is available in open source from all of the three distributions.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for the valuable feedback and comments, which has helped guide and improve the content and presentation of this paper. We also thank the members of the Advanced Development Group at SanDisk Corp. for their insightful comments.

9. REFERENCES

[1] Configuration of the doublewrite buffer. http://www.percona.com/doc/percona-server/5.5/performance/innodb_doublewrite_path.html.

[2] Mariadb 10.0 fusion-io. http://bazaar.launchpad.net/~maria-captains/maria/10.0-FusionIO.

[3] Mysql with innodb page-io compression. http://labs.mysql.com/.

[4] Percona server with xtradb, page compression mtflush. http://code.launchpad.net/~gl-az/percona-server/5.6-pagecomp_mtflush.

[5] Sandforce: Durawrite data reduction. http://www.lsi.com/company/technology/duraclass/pages/durawrite.aspx.

[6] Agency. Report to Congress on server and data center energy efficiency public law 109-431. Tech. rep., United States Environmental Protection Agency, 2007.

[7] Armstrong, T. G., Ponnekanti, V., Borthakur, D., and Callaghan, M. Linkbench: A database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 1185–1196.

[8] Ayers, L. E2compr: Transparent file compression for linux. http://e2compr.sourceforge.net(1997).

[9] Barroso, L. A. The Price of Performance: An Economic Case for Chip Multiprocessing. *ACM Queue* (Sept. 2005), 48–53.

[10] Barroso, L. A., and Hölzle, U. The case for energy-proportional computing. *Computer 40*, 12 (Dec. 2007), 33–37.

[11] Bonwick, J., and Moore, B. ZFS - The Last Word in File Systems.

[12] Callaghan, M. The effect of page size on innodb compression. https://www.facebook.com/note.php?note_id=10150348315455933.

[13] Callaghan, M. Making innodb compression adaptive. https://www.facebook.com/note.php?note_id=10150345355665933.

[14] Committee, T. T. Sbc-4 spc-5 atomic writes and reads. http://www.t10.org/cgi-bin/ac.pl?t=d\&f=14-043r4.pdf.

[15] Goldstein, J., Ramakrishnan, R., and Shaft, U. Compressing relations and indexes. In *Data Engineering, 1998. Proceedings., 14th International Conference on* (Feb 1998), pp. 370–379.

[16] Graefe, G., and Shapiro, L. D. Data compression and database performance. In *In Proc. ACM/IEEE-CS Symp. On Applied Computing* (1991), pp. 22–27.

[17] Iyer, B. R. Data compression support in databases. In *In Proceedings of the 20th International Conference on Very Large Data Bases* (1994), pp. 695–704.

[18] Johnson, T. Performance measurements of compressed bitmap indices. In *Proceedings of the 25th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1999), VLDB '99, Morgan Kaufmann Publishers Inc., pp. 278–289.

[19] Josephson, W. K., Bongo, L. A., and Flynn, David Li, K. Dfs: A file system for virtualized flash storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (2010), FAST '10.

[20] Klonatos, Y., Makatos, T., Marazakis, M., Flouris, M. D., and Bilas, A. Transparent online storage compression at the block-level. *Trans. Storage 8*, 2 (May 2012), 5:1–5:33.

[21] Koomey, J. G. Estimating total power consumption by servers in the U.S. and the world. Tech. rep., Lawrence Derkley National Laboratory, Feb. 2007.

[22] Marmol, L., Sundararaman, S., Talagala, N., Rangaswami, R., Devendrappa, S., Ramsundar, B., and Ganesan, S. Nvmkv: A scalable and lightweight flash aware key-value store. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)* (Philadelphia, PA, June 2014), USENIX Association.

[23] Ordulu, N. Getting innodb compression ready for facebook scale. http://www.percona.com/live/mysql-conference-2012/sessions/getting-innodb-compression-ready-facebook-scale.

[24] Ouyang, X., Nellans, D. W., Wipfel, R., Flynn, D., and Panda, D. K. Beyond block i/o: Rethinking traditional storage primitives. In *HPCA* (2011), pp. 301–311.

[25] Park, Y., and Kim, J.-S. zftl: Power-efficient data compression support for nand flash-based consumer electronics devices. In *Consumer Electronics, IEEE Transactions on (Volume:57 , Issue: 3)* (2011), IEEE, pp. 1148–1156.

[26] Poess, M., and Potapov, D. Data compression in oracle. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29* (2003), VLDB '03, VLDB Endowment, pp. 937–947.

[27] Rana, I. Innodb compression improvements in mysql 5.6. https://blogs.oracle.com/mysqlinnodb/entry/innodb_compression_improvements_in_mysql.

[28] Ray, G., Haritsa, J. R., and Seshadri, S. Database compression: A performance enhancement tool. In *Proc. of 7th Intl. Conf. on Management of Data (COMAD)* (1995).

[29] Richard Russon, R., and Fledel, Y. Ntfs documentation.

[30] Rodeh, O., Bacik, J., and Mason, C. Btrfs: The linux b-tree filesystem. *Trans. Storage 9*, 3 (Aug. 2013), 9:1–9:32.

[31] Saxena, M., Swift, M. M., and Zhang, Y. Flashtier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 267–280.

[32] SNIA, A. s. . i. t. Nvm programming model (npm). http://snia.org/sites/default/files/NVMProgrammingModel_v1.pdf.

[33] Subramanian, S., Sundararaman, S., Talagala, N., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Snapshots in a flash with iosnap. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 23:1–23:14.

[34] Talagala, N. Native flash support for applications. http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2012/20120823_S304B_

Talagala.pdf.

[35] Woodhouse, D. JFFS : The Journalling Flash File System, 2001.

[36] Yang, J., Plasson, N., Gillis, G., and Talagala, N. Hec: Improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference* (New York, NY, USA, 2013), SYSTOR '13, ACM, pp. 10:1–10:11.

[37] Yim, K. S., Bahn, H., and Koh, K. A flash compression layer for smartmedia card systems. In *Transactions on Consumer Electronics, Vol. 50, No. 1* (2004), IEEE.

[38] Ziv, J., and Lempel, A. A universal algorithm for sequential data compression. *IEEE TRANSACTIONS ON INFORMATION THEORY 23*, 3 (1977), 337–343.

[39] Ziv, J., and Lempel, A. Compression of individual sequences via variable-rate coding, 1978.

[40] Zukowski, M., Heman, S., Nes, N., and Boncz, P. Super-scalar ram-cpu cache compression. In *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on* (April 2006), pp. 59–59.