The Academic College of Tel Aviv-Yaffo

THE SCHOOL OF COMPUTER SCIENCE

Dictionary-Based Cache Line Compression

April 2024

Thesis submitted in partial fulfilment of the requirements for the M.Sc. degree in the School of Computer Science of the Academic College of Tel Aviv-Yaffo

By

Daniel Cohen

The research work for the thesis has been carried out under the supervision of
Dr. Sarel Cohen

Contents

1 Abstract

Active-standby mechanisms for VM high-availability demand frequent synchronization of memory and CPU state, involving the identification and transfer of "dirty" memory pages to a standby target. Building upon the granularity offered by CXL-enabled memory devices, as discussed by Waddington et al. [24], this thesis proposes a dictionary-based compression method operating on 64-byte cache lines to minimize snapshot volume and synchronization latency. The method aims to transmit only necessary information required to reconstruct the memory state at the standby machine, augmented by byte grouping and cache-line partitioning techniques. We assess the compression benefits on memory access patterns across 20 benchmarks snapshots and compare our approach to standard off-the-shelf compression methods. Our findings reveal significant improvements across nearly all benchmarks, with some experiencing over a twofold enhancement compared to standard compression, while others show more moderate gains. We conduct an in-depth experimental analysis on the contribution of each method and examine the nature of the benchmarks. We ascertain that the repeating nature of cache lines across snapshots and their concise representation contributes most to the size reduction, accounting for 92% of the gains. Our work paves the way for further reduction in the data transferred to standby machines, thereby enhancing VM high-availability and reducing synchronization latency.

The research results conducted during the course of the work on this thesis were published in HotStorage'24 conference

2 Introduction

In the realm of cloud computing and data management, ensuring the resilience of applications is crucial, not only for maintaining service continuity but also for safeguarding against data loss and operational disruptions. Our work addresses the technique of *continuous snapshotting* of a VM's memory space, where snapshots are taken at intervals of seconds or lower [20], employing an active-standby architecture to bolster high availability. The active-standby architecture for snapshotting plays a crucial role in ensuring application resilience by maintaining a real-time replica of the active system. In the event of a system failure, the standby system can quickly take over, minimizing disruption and maintaining continuous service availability. Leading solutions such as Zert∅ [25], Remus [8], rRVN [12], and Kemari [21] exemplify the deployment of continuous snapshotting within this architecture. These systems are designed to minimize downtime and enhance the resilience of applications through rapid recovery mechanisms, setting a standard for reliability in the industry. Traditionally, capturing and replicating entire memory spaces to create a comprehensive snapshot of the VM's state has been executed at a 4KB page level. However, this method often results in the transmission of a substantial amount of redundant data, i.e. memory segments which haven't changed between snapshots. The advent of Compute Express Link (CXL) technology has ushered in a paradigm shift in how these snapshots can be created. By leveraging CXL's capabilities, recent advancements have moved away from page-level snapshotting to a more granular approach utilizing 64-byte cache lines [24]. This technique focuses on transmitting only the cache lines deltas — namely, the differences between consecutive snapshots — rather than replicating the entire snapshot. This state-of-the-art method significantly reduces the volume of data that needs to be transferred, enhancing the efficiency of the snapshotting process.

Despite these advancements, the challenge of data transfer still persists in large-scale virtualized environments, where hundreds of VMs may be running simultaneously [1]. In such settings, the sheer volume of data that needs to be synchronized across the network can become a critical bottleneck, impacting the overall performance and reliability of the system. The motivation behind our research stems from this challenge: our objective is to further minimize the network data transfer during the snapshotting process, thereby alleviating the strain on network resources and enhancing the operational efficiency of large-scale VM-based systems.

Focusing on this challenge, we start out by examining a number of widely used compression algorithms to determine which one can offer the best compression factor and compression speed. Following that, we developed a novel solution that employs a *dictionary-based* compression technique that is specifically tailored to the context of VM snapshotting. This solution builds upon the fine granularity offered by CXL; it uses 64-byte cache lines and introduces an innovative method for compressing the repeating cache line deltas over time. Our approach further improves the data reduction ratio by rearranging the data before the compression and by partitioning the cache lines into groups.

By choosing the optimal off-the-shelf compression algorithm we observed a drastic reduction in snapshot volume of 70% to 91% in various workloads. On top of that, our dictionary compression technique reduces the data volume by 20% to 70% of the previous results depending on the workload, with an average reduction of 30%.

This substantial reduction in the volume of the data transferred not only alleviates the network bottleneck but also contributes to more efficient use of network resources, ultimately supporting the high-availability requirements of modern cloud computing and data center operations.

This thesis is organized as follows. Section 3 gives background and presents concepts and notations within this domain needed to understand this work. Section 4 describes our exploration of popular compression libraries, followed by our new methods: dictionary-based, byte grouping and partitioning. Section 5 outlines the experiments and the results. In Section 6, we discuss related work. Finally, in Section 7, we conclude and suggest future research directions.

3 Preliminaries

In this chapter we will introduce the concepts of Virtual Machines, Snapshots, Compression and VM Synchronization in a cloud based environment and provide the necessary background for understanding the research.

3.1 Virtual Machine (VM)

A Virtual Machine (VM) can be succinctly described as a software-based simulation of a physical computer. Within the context of cloud computing, VMs are the cornerstone that allows the provisioning of multiple isolated computing environments on a single physical host. Below are the key concepts and features of VMs:

- **Isolation:** Each VM operates in an isolated environment. This means that the actions in one VM do not affect other VMs on the same host. This isolation guarantees security and fault separation among the VMs.

- **Hardware Independence:** VMs are not directly bound to the physical hardware they run on. Instead, they interact with a virtualized representation of the hardware resources, presented to them by the underlying hypervisor or virtualization software. This abstraction allows VMs to be easily moved, copied, or migrated across different physical hosts.

- **Snapshot Capability:** One of the most powerful features of VMs in the context of cloud systems is the ability to take a snapshot, which is a point-in-time representation of a VM's state. This includes its memory content, virtual machine disk file, and all associated metadata. Snapshots enable functionalities like recovery, cloning, and stateful migration of VMs.

- **Resource Overcommitment:** Given that not all VMs utilize their allocated resources fully at all times, cloud providers often overcommit resources, allocating more virtual resources than there are physical ones available. The VM's perception of available resources is managed by the hypervisor, which can reallocate them dynamically as needed among the VMs.

In cloud-based systems, VMs play a pivotal role in ensuring scalability, flexibility, and efficient resource utilization. As data centers transition to serve global audiences with stringent demands for high availability, there's an increasing need to optimize operations like VM snapshot transfer to ensure rapid recovery and reduced downtime.

3.2 Virtual Machine Snapshotting

Virtual machine snapshotting is a fundamental feature in virtualized environments, allowing the capture of the state of a VM at a specific point in time. This snapshot can be used for various purposes ranging from backup and recovery to testing and development. Below we delve deeper into the nature and utility of VM snapshots:

- **Definition:** A VM snapshot comprises the current state of the VM's CPU, memory, the virtual machine disk file, and all associated metadata. Essentially, it's a comprehensive representation of the VM's state at the moment the snapshot is taken.
- **Use Cases:**
  - *Backup and Recovery:* Snapshots can serve as a backup mechanism. In the event of a failure or a software bug, administrators can revert the VM to a previous known good state using snapshots.
  - *Testing and Development:* Before making changes to a VM, be it software updates or configuration tweaks, a snapshot can be taken. If the changes result in unforeseen issues, the VM can easily be reverted to its pre-change state.
  - *VM Migration and Cloning:* Snapshots can assist in creating clones of VMs or migrating VMs across different hosts or data centers without needing to duplicate the entire base VM.
- **Implementation Details:** When a snapshot is taken, the original disk file becomes read-only, and all subsequent writes are directed to a new delta disk file. This ensures that the original state is preserved, allowing for reversion if necessary.
- **Performance Considerations:** While snapshots are undeniably useful, they're not devoid of overhead. Prolonged reliance on delta disks or having a long chain of snapshots can degrade VM performance. As such,

snapshots are typically not intended as a long-term storage solution but rather a temporary point-in-time capture.

- **Storage Implications:** Snapshots can quickly consume storage space, especially when the VM sees a lot of write activity. Efficient storage management and regular snapshot consolidation are crucial to ensure optimal system performance and storage utilization.

In the realm of cloud computing and with the pressing need for high availability and disaster recovery, VM snapshotting stands as a vital tool.

3.3 Memory Snapshotting

Memory snapshotting, often part of the broader VM snapshotting process, captures the content of a VM's RAM. This is pivotal for preserving the running state and active processes of a VM. Below, we discuss the key facets and implications of memory snapshotting:

- **Definition:** A memory snapshot captures the entirety of a VM's RAM content at a specific moment. This includes active processes, cached data, and even unsaved changes to applications or files.
- **State Preservation:** The primary advantage of including memory in a snapshot is the ability to restore a VM not just to a consistent disk state, but also to a specific running state. Without memory snapshotting, a restored VM would need to undergo a cold boot, losing the context of any running applications or unsaved data at the time the snapshot was taken.
- **Snapshotting Process:** Memory snapshotting, given its need to capture volatile data, is more time-sensitive compared to disk snapshotting. Typically, the VM's execution is momentarily paused to ensure a consistent memory state. This pause is usually brief but can lead to a short period of unresponsiveness from the VM.
- **Storage Implications:** Memory snapshots can be sizeable, often corresponding to the allocated RAM of the VM. Therefore, they can significantly increase the storage requirements of a snapshot, especially for VMs with large memory footprints.
- **Performance Considerations:** When a memory snapshot is restored, the VM's RAM content is loaded from the snapshot file. Depending on storage speed and the VM's memory size, this can introduce a noticeable delay in the restoration process. Additionally, frequent memory snapshotting can impact the performance of a VM, given the need to periodically pause execution.
- **Application Continuity:** For applications that require continuous uptime or have stringent real-time requirements, the brief interruption during

memory snapshotting might be non-trivial. In such cases, it's essential to balance the need for stateful backups with application performance requirements.

Memory snapshotting offers a deeper level of state preservation, allowing VMs to be restored to their exact running state at the time of the snapshot.

3.4 Dirty Pages and Efficient Snapshot Deltas

One of the primary challenges in generating snapshot deltas lies in efficiently identifying and capturing only the changes (deltas) since the previous snapshot. To address this, a prevalent technique revolves around monitoring "dirty" pages. Here's a deeper explanation:

- **Definition:** *Diry Page***.** In the context of virtualization and snapshotting, a "dirty" page refers to a page, either in memory or on disk, that has been modified (or written to) since the last snapshot or checkpoint was taken. By contrast, "clean" pages remain unchanged and, therefore, don't need to be captured in a new delta.

- **Definition:** *Snapshot Delta***.** A snapshot delta captures the differences or changes made to a VM's state or data between two points in time. Instead of storing a complete copy of the VM's state at each snapshot, delta snapshots record only the modifications since the last snapshot. This method significantly reduces the amount of data that needs to be saved and transferred, enhancing the efficiency of snapshot operations and minimizing the impact on storage and network resources.

- **Tracking Mechanism:** Hypervisors or virtualization platforms typically employ a bitmap or similar data structure to track dirty pages. When a write operation occurs to a page, its corresponding bit in the bitmap is set (marked as dirty). This allows for a quick identification of modified pages when generating snapshot deltas.

- **Efficiency Gains:** By focusing only on dirty pages, the system can ignore large swathes of unmodified data, leading to quicker snapshot operations and smaller delta files. This ensures that only truly necessary data is processed and stored, optimizing both performance and storage utilization.

- **Snapshot Consistency:** Dirty page tracking not only improves efficiency but also aids in ensuring snapshot consistency. By capturing all modified pages since the last checkpoint, the system guarantees that the snapshot delta accurately reflects all changes, ensuring that no data inconsistencies arise during restoration.

- **Challenges and Considerations:** While dirty page tracking is efficient, it's not without challenges. High write activity can lead to rapid growth of the dirty bitmap, potentially causing overhead.

Dirty page tracking, is an effective approach to snapshot delta generation. It strikes a balance between data consistency and operational efficiency, making it a preferred method in many modern virtualization platforms.

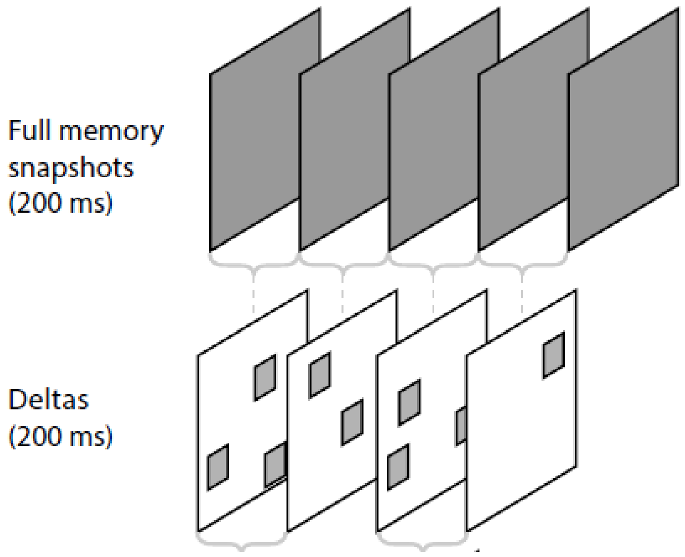

Figure 1: Snapshot Deltas

3.5 Synchronization of Active-Standby Virtual Machines

In cloud environments, synchronization of Active-Standby Virtual Machines (VMs) is crucial for achieving high availability and fault tolerance. The essence of this synchronization lies in keeping the standby VM—a replica awaiting activation—in a state that mirrors the active VM as closely as possible. Periodic synchronization processes are initiated, wherein changes from the active VM, often encapsulated as snapshot deltas or logs, are transmitted and applied to the standby VM. This involves both disk state and, in cases requiring high fidelity, memory state as well. Advanced mechanisms, such as *dirty page tracking*, can be employed to enhance efficiency by transmitting only modified data. The cloud's underlying infrastructure often leverages high-speed networks to minimize latency during synchronization, ensuring a near-real-time reflection of the active VM's state in the standby. This synchronization ensures that, should the active VM falter, the standby can seamlessly assume its role with minimal service disruption.

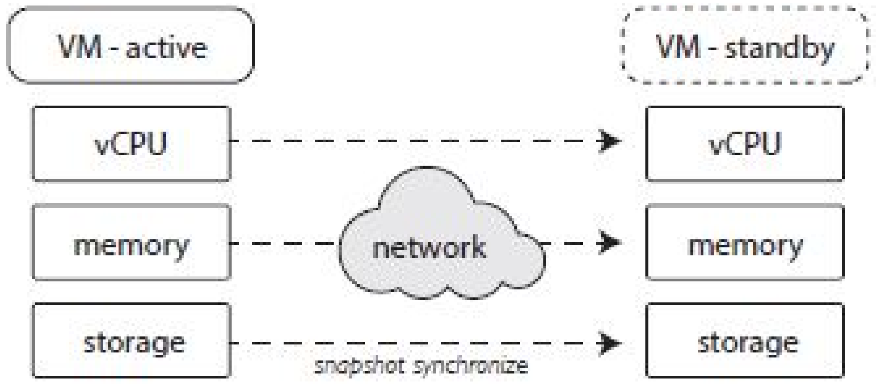

Figure 2: VM Synchronization

3.6 Compression

Compression refers to the technique of reducing the size of data by encoding it in a more efficient manner, allowing for optimal storage or faster transmission. In the realm of virtualization, compression is of paramount importance, especially when dealing with virtual machines and their memory snapshots. These snapshots, capturing the VM's current state, can grow significantly large, especially for VMs with substantial allocated resources. Compressing these snapshots not only conserves storage space but also expedites processes like backup, migration, and synchronization between active and standby VMs. Employing advanced compression algorithms tailored to the characteristics of VM data ensures minimal loss of data fidelity while achieving size reductions.

4 Compressing Cache Lines

Recent work[24] highlights the advantage of using cache line granularity over traditional page-based approaches in VM snapshotting. By leveraging the finer granularity of 64-byte cache lines, the volume of data that needs to be replicated and transferred can be substantially reduced. This approach minimizes unnecessary data replication that arises in conventional page-based methods.

Motivated by these findings, in this section we investigate how to reduce the cache line data even further. Our approach is to exploit the specific characteristics of the cache line data and employ various compression techniques on cache line deltas. We start with the baseline standard compression, followed by new methods tailored specifically to the nature of VM memory data.

4.1 Compressibility Potential

Initially we examine the use of standard off-the-shelf compression techniques on the workloads tested in [24]. We evaluated some of the most popular compression algorithms when applied to cache line deltas (CLDs). We compared the performance of the following algorithms: Zlib[10], ZSTD[7], LZ4[6], Gzip[9], and Snappy[19], in terms of compression factor and compression speed.

Based on this initial observation, we propose several methods to further compress the data beyond that achieved by the optimal compression algorithm alone (which will be our baseline). The first technique we propose is a *dictionary-based* compression method that leverages the repetitive nature of many data patterns in VM snapshots.

4.2 Proposed Compression Enhancements

4.2.1 Dictionary Based Compression

Recall that in the context of the active-standby VM system architecture, a memory snapshot is comprised of all the cache lines that changed over the current epoch and their corresponding memory addresses. The dictionary-based compression method represents the cache lines that changed over time in a **dictionary**. The idea is to exploit the fact that cache line values repeat across memory addresses over time (see figure 7). Instead of listing these repeated cache line values many times, we assign them a unique number, which we call a "serial number", that is smaller in size than the original cache line value of 64 bytes. To achieve that, we keep a table (i.e., the dictionary) of all the cache line values that have been changed over the epochs. We represent a given cache line value by its serial number in the dictionary. The serial number is a 4-byte running counter, starting from zero. This counter increments by one for every new unique cache line processed, providing a maximum range defined by the upper limit of a 4-byte integer. As a new cache line is processed, it is assigned current value of the running counter. This assigned value is then used as a key in our dictionary, with the value representing the corresponding cache line data.

The original VM memory snapshot on the active machine, which consists of cache lines and their memory addresses, is now transformed into (i) a list of addresses and their corresponding serial numbers within the dictionary, along with (ii) the newly added keys to the dictionary, namely the new cache lines that were inserted into the dictionary in the current epoch. We call the list of serial numbers *symbolised cache lines.* See an example in Figure 3.

In summary, the transformed snapshot that is transmitted to the stand-by machine is composed of a *compressed version* of the following components: (1) Symbolized cache lines that changed during this epoch; (2) Memory addresses that correspond to these cache lines; (3) Changes (deltas) in the dictionary, namely the new values that were modified and added to the dictionary in this epoch, ordered by serial number. This transformed snapshot is called the 'new snapshot'. See the example in Figure 3.

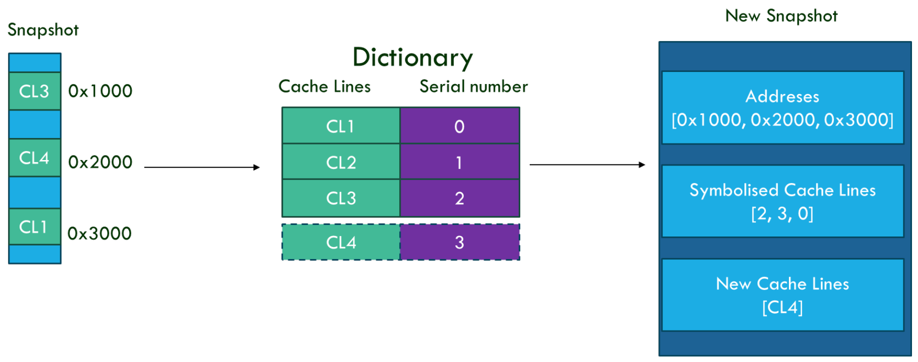

Figure 3: Dictionary Based Compression Architecture. *The current epoch consists of a list ¡cache lines, memory addresses¿; only C4 is new and did not appear in previous epochs. The transformation creates: memory addresses of current epoch, a list of symbolized cache lines, and new dictionary values , all compressed.*

Upon transfer to the standby machine, the recovery process begins. The standby machine decompresses all the components of the new snapshot. To reconstruct the original snapshot, we first update the dictionary with the new values (deltas). Then, each symbolized cache line is decoded using the dictionary to retrieve its original value. These cache lines are then paired with their respective memory addresses, effectively reconstructing the original VM memory state.

The key observation in our dictionary-based compression is that in order to reconstruct the standby VM, there is no need to build a dictionary at the stand-by server; it's enough to reconstruct a list of cache lines identified by their serial number, i.e., from the symbol array. This is in contrast to the active VM; there, a dictionary that supports efficient lookup is needed in order to efficiently determine whether a cache line had already appeared in previous snapshots. Since our objective is to minimize the transferred data between the two machines, this observation is critical; we simply transfer the newly added cache line in the order it was added (the serial number is implicit) which are simpler, and easier to compress. Work by Tian et al. [22] is also based on identifying duplicated cache lines and storing only one copy of the data so that it can be accessed from multiple physical addresses. However, these duplicated data blocks are identified by hash values rather than by their serial number in the dictionary which is more succinct.

4.2.2 Compression with Byte Grouping

Recall that the symbolized cache lines are a list of incrementing 4-byte serial numbers and as such can be represented more efficiently. In particular, we take advantage of the fact that many leading (most significant) bits in these numbers are repeated and therefore compress them in a ”column-based” manner. In

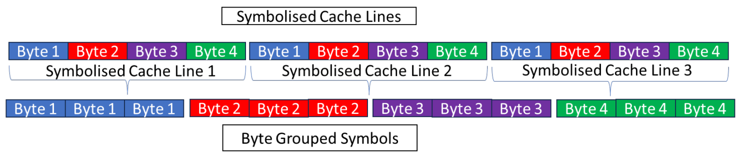
Symbolised Cache Lines
Byte 1 Byte 2 Byte 3 Byte 4 Byte 1 Byte 2 Byte 3 Byte 4 Byte 1 Byte 2 Byte 3 Byte 4
Symbolised Cache Line 1
Symbolised Cache Line 2
Symbolised Cache Line 3
Byte 1 Byte 1 Byte 1 Byte 2 Byte 2 Byte 2 Byte 3 Byte 3 Byte 3 Byte 4 Byte 4 Byte 4
Byte Grouped Symbols

Figure 4: Byte Grouping.

Section 6 we discuss the motivation behind choosing this encoding as opposed to Huffman encoding.

Byte Grouping operates by dissecting these serial numbers into bytes and grouping the corresponding bytes into groups. Within these groups, bytes at identical positions are extracted and assembled together into a contiguous group. For example, the first byte from each serial number forms a new group, the second byte from each serial number forms another, and so forth. This reorganization facilitates the compression algorithm's ability to more effectively reduce the size of the data, since bytes that are alike are now adjacent. [13]

As shown in Figure 4, the symbolized cache lines form an array of 4-byte serial numbers. We create 4 separate groups that are comprised of bytes from each relative position in this array, i.e. byte 0, byte 1, byte 2 and byte 3, and compress them group by group. By compressing these groups instead of the symbolised cache lines array the compression factor of the symbolised cache lines is greatly improved.

4.2.3 Cache Line partitioning

Cache line partitioning involves dividing each 64-byte cache line into smaller partitions and representing partitions in the dictionary rather than the full 64-byte cache line. These partitions vary in size. For example, a 64-byte cache line can be split into the following: 2x partitions of 32 bytes, 4x partitions of 16 bytes or 8x partitions of 8 bytes. The primary motivation behind cache line partitioning is that smaller cache line partitions may exhibit more frequent and identifiable patterns compared to larger 64-byte blocks. This results in a more compact and efficient dictionary, where each entry represents a commonly occurring data segment. The trade off is that we will have to symbolise more cache line segments depending on the partition size and that will result in a larger symbolised cache lines footprint. We tested this partitioning idea with different values of granularity.

5 Experimental Results

5.1 Experimental Setting

We experimentally evaluate our approach across a diverse range of workloads from various domains[1]. Our test workloads includes a broad spectrum of real-world data with distinct characteristics, such as neural networks from applications like Rnnoise and Onednn, databases such as LevelDB and Sqlite, cryptographic algorithms such as nettle-aes and many others. Snapshots were taken every 200 msec; each dataset is a single application execution and includes all the snapshots within this execution. The Snapshot Descriptions Table 1 provides a comprehensive explanation of these datasets.

[1]The dataset will be publicly available soon.

Table 1: Benchmark test descriptions

Test	**Uncompressed Dictionary Size (MB)**	**Description**
sqlite	57	Simple SQLite database benchmark
ebizzy	18	Workload resembling web-server
leveldb	127	Key-value store that uses Snappy compression
influxdb	13932	InfluxDB time-series database
memcache	515	Memcache in-memory cache put workload
build-gcc	4432	Compile the GCC compiler
quantlib	202	Quantitative finance for modeling, trading, and risk management
ngspice	8099	SPICE circuit emulator
py-imgseg	4928	Python image segmentation (skimage)
dolfyn	619	Computational Fluid Dynamics (CFD) simulation
himeno	2139	Linear solver of pressure Poisson
py-3drotate	446	Python 3D matrix rotation (numpy)
nettle-aes	18	AES cryptography from the Nettle library
py-graph-spn	7634	Python weighted graph spanning tree
py-feature	1561	Python logistic regression feature selection
py-faces	904	Python face recognition using eigenfaces and SVMs
als	10240	MLlib Alternating Least Squares (ALS) matrix factorization
rnnoise	343	Recurrent neural network for audio noise reduction
genetic	9574	Genetic algorithm using the Jenetics library
onednn	2694	Deep neural network training

Our evaluation focuses primarily on *compression factor* as a key metric, reporting on the proportion of the original size; for instance, a factor of 0.4 indicates that the compressed data is 40% of the original size. In this section we present the overall compression factor of the complete snapshot data consisting of cache lines and addresses , which we call cache line deltas (CLD), utilizing all the methods presented in Section 4.

5.2 Compression Algorithms Evaluation

We begin by assessing the performance of each algorithms' compression factor and compression speed on each of our workloads. Compression is run on every epoch separately and the results shown are the sums of all the compressed sizes and compression times across all the epochs recorded in a workload.

Table 2: *This table showcases the performance of each algorithm mentioned in Section 4 across all our workloads. Each cell contains a tuple showing the compression factor and speed (in sec).*

Workload	Zlib	Zstd	Lz4	Gzip	Snappy
als	(0.18, 846s)	(0.13, 255s)	(0.26, 260s)	(0.18, 5128s)	(0.29, 264s)
genetic	(0.23, 403s)	(0.22, 127s)	(0.35, 119s)	(0.22, 1524s)	(0.36, 117s)
build-gcc	(0.17, 253s)	(0.15, 93s)	(0.26, 89s)	(0.17, 1121s)	(0.27, 89s)
dolfyn	(0.58, 56s)	(0.53, 26s)	(0.70, 22s)	(0.58, 183s)	(0.73, 21s)
ebizzy	(0.21, 16s)	(0.14, 14s)	(0.42, 13s)	(0.21, 54s)	(0.42, 13s)
himeno	(0.56, 97s)	(0.47, 32s)	(0.78, 20s)	(0.56, 108s)	(0.82, 18s)
influxdb	(0.13, 764s)	(0.09, 260s)	(0.25, 269s)	(0.13, 4595s)	(0.25, 270s)
leveldb	(0.42, 29s)	(0.26, 18s)	(0.52, 17s)	(0.42, 52s)	(0.52, 17s)
memcache	(0.11, 17s)	(0.12, 8s)	(0.19, 5s)	(0.11, 74s)	(0.19, 5s)
nettle-aes	(0.24, 1s)	(0.23, 0.26s)	(0.35, 0.18s)	(0.23, 6s)	(0.34, 0.19s)
ngspice	(0.49, 440s)	(0.48, 109s)	(0.58, 98s)	(0.49, 1976s)	(0.57, 98s)
onednn	(0.40, 427s)	(0.36, 86s)	(0.51, 76s)	(0.40, 453s)	(0.54, 73s)
quantlib	(0.33, 13s)	(0.25, 4s)	(0.39, 3s)	(0.32, 82s)	(0.42, 3s)
rnnoise	(0.76, 35s)	(0.77, 8s)	(0.90, 4s)	(0.76, 54s)	(0.91, 4s)
sqlite	(0.20, 4s)	(0.17, 1s)	(0.28, 1s)	(0.19, 27s)	(0.30, 1s)
py-3drotate	(0.23, 27s)	(0.21, 7s)	(0.33, 7s)	(0.22, 135s)	(0.34, 7s)
py-faces	(0.35, 78s)	(0.32, 17s)	(0.46, 15s)	(0.35, 413s)	(0.47, 14s)
py-feature	(0.37, 138s)	(0.33, 34s)	(0.48, 30s)	(0.37, 742s)	(0.49, 30s)
py-graph-spn	(0.23, 495s)	(0.22, 143s)	(0.33, 144s)	(0.23, 2519s)	(0.33, 142s)
py-imgseg	(0.50, 264s)	(0.43, 70s)	(0.58, 67s)	(0.50, 578s)	(0.62, 64s)

Our results show the following: In terms of compression factor, ZSTD achieves the best results in nearly all workloads except in *rnnoise* and *memcache* when compressed by ZLIB which were smaller by a negligible amount (1%). In terms of compression speed the results are mixed; ZSTD, SNAPPY and LZ4 generally perform the best with their results being the most comparable to each other, while ZLIB and especially GZIP were slower by an order of magnitude in nearly every scenario.

Ultimately, the algorithm we'll choose going forward will be ZSTD, seeing as it has the best compression factor by a margin in a number of scenarios and its compression time is on par with the best performers.

The following results shown in Figure 5 achieved by using ZSTD compression will serve as the baseline for our additional experiments.

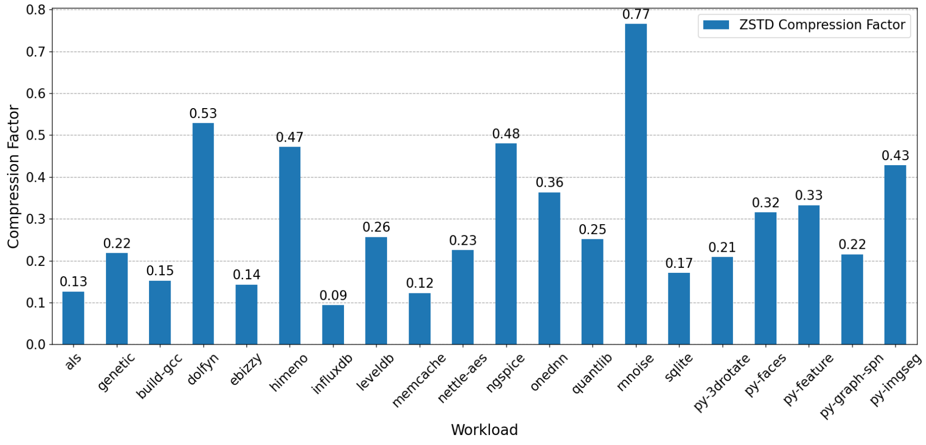

Figure 5: ZSTD Compression on Cache Lines.

5.3 Dictionary Based Compression Evaluation

In this section we evaluate the compression improvements between regular ZSTD and dictionary-based ZSTD on the snapshot data. We now present the compression factor achieved by applying dictionary-based compression, as described previously in Section 4, on the entirety of each snapshot (cache lines and addresses), followed by the standard ZSTD compression.

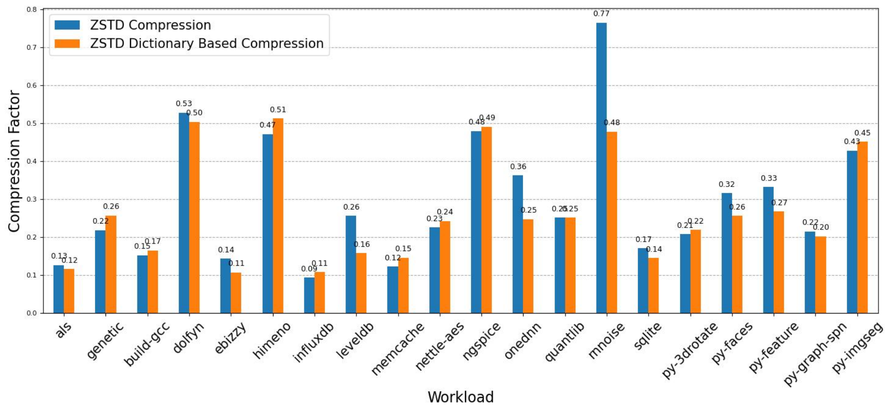

Figure 6: *Compression Factor with Dictionary Based Method.*

Figure 6 shows the compression factors achievable on different workloads when applying our Dictionary based method. On some snapshots, we see that overall there was an improvement for most of the application snapshots but this improvement is not uniform. For instance the **rnnoise** and **leveldb** application snapshots exhibit an improved compression factor by 37%and 38% respectively. Conversely the **nettle-aes** snapshot and others exhibited an increase from their original compressed size, suggesting that those applications are not susceptible to any further compression.

To explain this phenomena, we count the occurrences of each cache line within a number of snapshots. The rationale is that applications which have cache lines that are highly replicated would benefit greatly from our symbolic encoding. In Figure 7 we divided the cache lines into five distinct occurrences

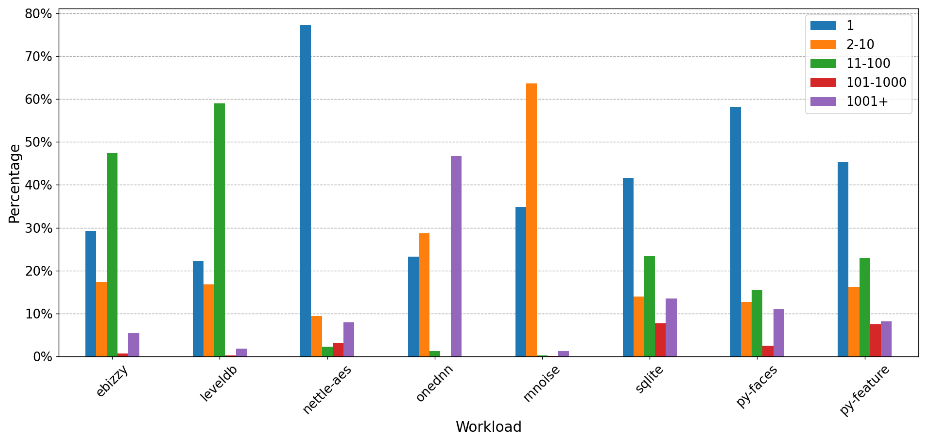

Figure 7: Histogram of Cache Line Occurrences

groups to see the cache lines distribution. We notice that in the majority of snapshots most of the cache lines are singletons, i.e. they appear only once. In the case of **nettle-aes** nearly 80% of cache lines are singletons which explains why the dictionary based approach had an adverse effect on ZSTD compression. Conversely in the case of **rnnoise** most cache lines appeared multiple times (2-10), so the dictionary-based method works more effectively.

5.4 Dictionary Compression with Byte Grouping

In this section we will evaluate the benefits of byte grouping. Recall that byte grouping groups together many bytes with the same values. Our assumption is that cache lines that appear often will receive a low encoding value with leading zeros and by applying byte grouping to the symbolized cache lines we are able to reorganize the data in such a way that the reoccurring zeros will be grouped together.

Initially, let's observe the compression results of byte grouping when applied solely to the symbolized cache lines (without the dictionary or cache line addresses).

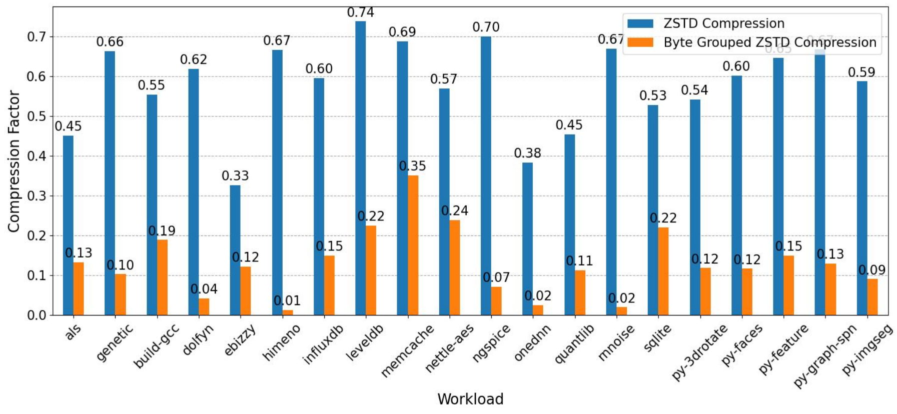

Figure 8: Compression with Byte Grouping on Symbolized Cache Lines

The results in Figure 8 , clearly show us that Byte Grouping greatly improves the compressiblity of the symbolised cache lines suggesting that many bytes with similar values were indeed grouped together leading to improved compression. Additionally we also applied Byte Grouping on to the Cache Line addresses which is also a group of 4 byte integers with a high number of repeating characters.

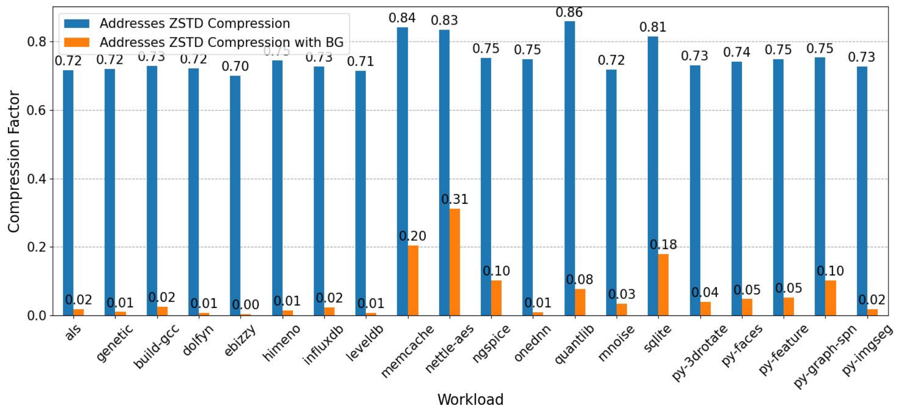

Figure 9: Compression with Byte Grouping on Cache Line Addresses

As we can see in Figure 9, Byte Grouping also has a very beneficial effect when compressing the cache line addresses.

Now, let's examine this enhancement improves the overall compression factor when combined with the rest of the new snapshot components.

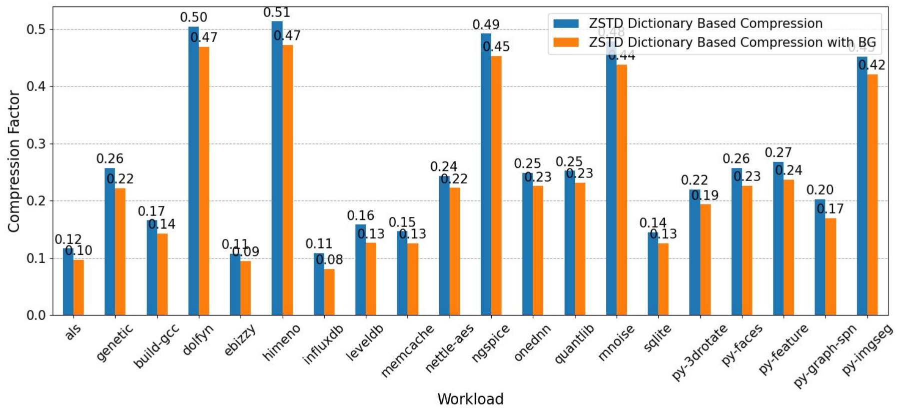

Figure 10: Compression Factor with Dictionary and Byte Grouping

Figure 10 shows that, despite the improvements offered by applying byte grouping prior to compressing the symbolised cache lines and the cache line addresses, the dominating factor by far on the snapshot size is the dictionary size.

5.5 Dictionary Compression with Cache Line Partitioning

In order to reduce the most prominent factor on the overall footprint (which is the dictionary) even further we utilize cache line partitioning

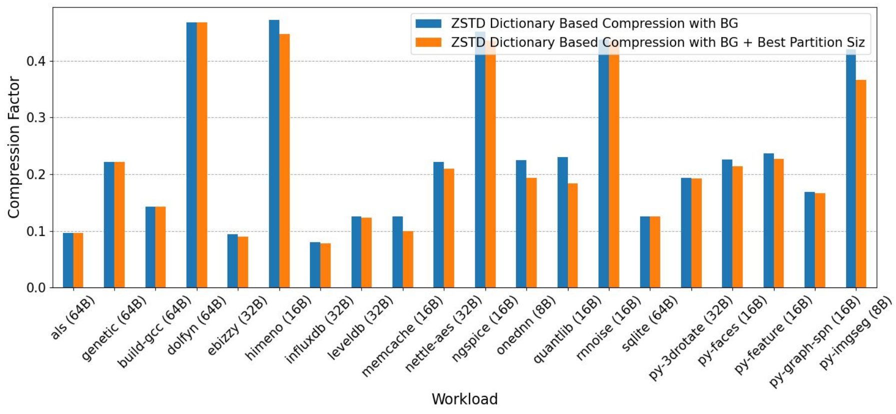

Figure 11: Compression Factor with Dictionary and Byte Grouping and Cache Line partitioning

. Figure 11 shows the compression factor when introducing partitioning (the orange bar). Note that partitioning can be done at various granularities (8B, 16B, 32B, 64B). The orange bar in the graph depicts the compression with the *optimal* partition granularity across all options; the number in brackets is the best partition size for this workload (Up until now all the experiments were conducted with the default partition size which is 64B). Overall we can observe minor improvements to the compression factor in most workloads when utilizing a better partition size. For instance in *py-imgseg* workload there is an 8% reduction in compression factor and in the *quantlib* workload there was a 5% reduction. In other workloads the improvements are less impactful but overall there is always a reduction in size.

To gain a some additional insight into this enhancement we decided to measure which partition size tends to yield the best results, to that end we measured how each partition size performed on every workload and created the following ranking.

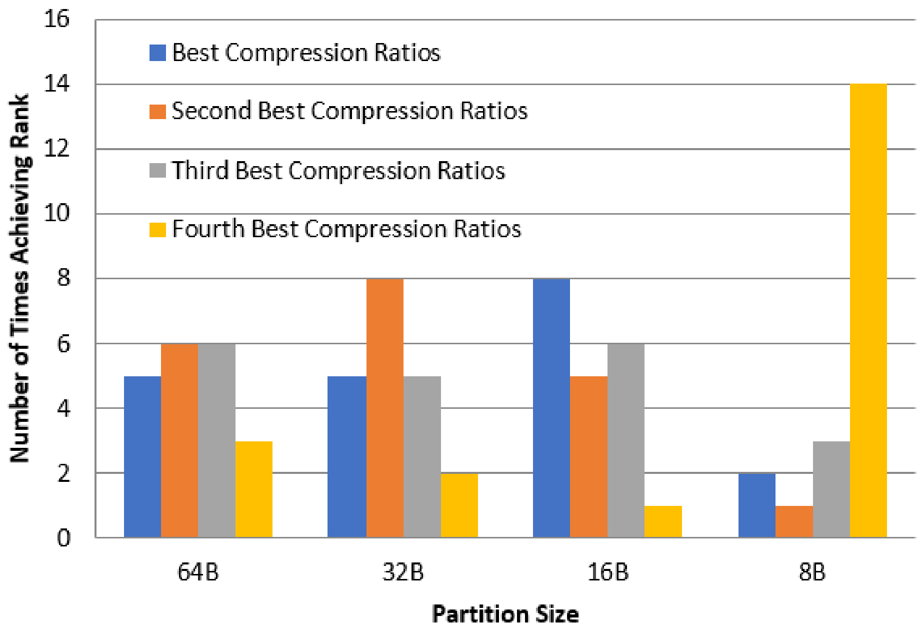

Figure 12: Cache Line Partition Rankings

Among the four possible partition sizes, 16B is the best choice for most workloads, while using 8B partition is the worst choice almost every time.

5.6 Detailed Compression Factor Evaluation

We would like to understand and explain the effectiveness of the dictionary based method and its enhancements (byte grouping and cache line partitioning) on the overall compression ratio. Recall that when utilizing our dictionary based compression, data is divided into three distinct entities: dictionary, symbolised cache lines and cache line addresses. Dictionary-based compression captures repeating cache lines within the snapshots. Byte grouping can reduce the size of the symbolised cache lines and of the cache line addresses, and cache line partitioning is used to efficiently represent the dictionary values. We extracted the compression ratio of each method separately and on each of the separate parts of the data in order to gain a better understanding of the total compression ratio. By averaging over all workloads, we see that the dictionary makes up 92% percent of the snapshot footprint, while the symbolized cache lines and the addresses comprise 5% and 3% respectively. See Figure 13.

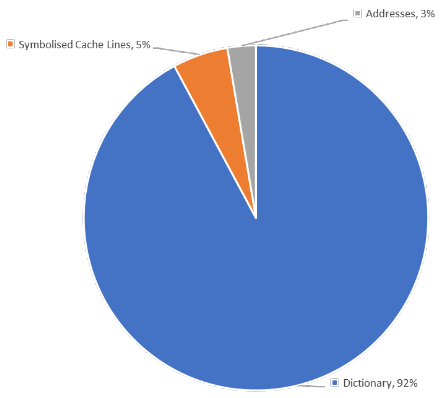

Figure 13: New Snapshot Components sizes. *Taken as an average across all workloads*

Caveat: Note that on some corner-cases, our method is not practical since the size of the dictionary grows beyond a manageable size, e.g. beyond 10GB. Nevertheless, only 2 out of the 20 workloads exhibited such behavior. Across the 20 workloads, the median dictionary size is 1.2 GB.

6 Related Work

There is a body of work dealing with the efficiency of live migration that operates at 4KiB page granularity [2, 26, 17, 23, 11, 18, 14] . These typically build on the problem of VM replication which, similar to VM snapshots, also aims at reducing the volume of data for network transfer. For example, Jin et al. [14] introduced a scheme for compressing memory at page level by examining the characteristics of pages, e.g word similarity.

Other work has examined high-frequency VM replication [8, 24]. Waddington et al. [24] transfer only the difference between two consecutive snapshots (i.e., the delta) to the standby machine at cache line (64B) granularity and apply off-the-shelf compression solutions including ZLIB and RLE. The contributions of our work, over [24], is that we focus on improving the compression of the cache line deltas further and reducing the network bandwidth to the standby machine.

This is achieved by developing a new dictionary-based compression technique tailored to this framework, which leverages the fact that in the standby machine no inserts to the dictionary are needed, thus there is no need to maintain a full dictionary in the standby machine that maps cache lines back to their symbol; instead, a simpler structure consisting of a consecutive array of unique cache lines is sufficient (where the position of a cache line in the array equals to its symbol).

Mittal et al. [16] provides a comprehensive survey on the compression of cache and main memory systems. More relevant to our work is Alameldeen and Wood [3] , who suggest a frequent pattern compression for L2 caches based on the insight that many data types require fewer bits than the maximum allocated. Zhang et al. [27] proposed an additional data cache structure called *frequent value cache*, that stores only frequently occurring cache values in compact encoding, thereby allowing the cache to store more data and improve the overall performance. Benini et al. [5] proposed a hardware-assisted data compression to reduce the traffic from cache-to-memory caused by cache write-back; cache lines are compressed before being written back to main memory, and decompressed upon cache refill.

6.1 Huffman Encoding

A dictionary-based Huffman encoding has been proposed as a method of compressing cache [4] and memory [15]. However, Huffman encoding emphasizes the challenge of fluctuating value-locality over time. Thus, the statistical prerequisites for Huffman encoding limit its widespread adoption, as noted by Mittal et al.[16]. In contrast, our proposed approach employs Byte Grouping with monotonic symbols, offering a simpler encoding mechanism that doesn't require a learning phase. By leveraging a monotonic order, our method eliminates the need for transferring symbol Huffman encoding. Notably, the compressed symbol array constitutes less than 5% of the total data transferred, further justifying our choice of the Byte Grouping technique.

6.2 Deduplication

Tian et al. [22] suggested *last level cache deduplication.* Their method is contrasted with our dictionary-based method in Section 4.2. The main difference in our approach as opposed to deduplication is that we use consecutive ordering for the symbols and not a hash fingerprint. This omits the need to transfer the fingerprint to the standby machine and reduces the bandwidth of the traffic transferred to the standby machine. Moreover, this allows us to forgo the construction of a dictionary in the standby machine.

7 Conclusions and Future Work

This thesis builds on the idea of representing periodical memory snapshots using cache line and addresses that have been modified between epochs instead of pages. We investigated how compressible cache lines are by using off the shelf compression algorithms and whether the cache lines are compressible beyond that by devising a dictionary-based method with additional enhancements to compress memory snapshot data. Our results show that the dictionary-based approach significantly reduces the snapshot footprint. The dominating factor in the snapshot footprint reduction is the minimal representation of newly changed cache lines in the dictionary, taking advantage of the fact the cache line bit patterns are often repeated.

Reducing the footprint of snapshots is critical - it reduces network bandwidth and recovery time. However there is a trade-off between computation time and footprint reduction. A natural future direction is to investigate this trade-off. In addition, our study shows that there is significant variability across workloads - not all methods perform equally well on all workloads. Hence, another future direction is to develop approaches that investigate which methods to use on a particular workload.

Our method relies on the fact that the dictionary size remains relatively steady over all the snapshots. This may not always be the case. In some corner-cases, we noticed that the accumulated number of different cache lines over time (and therefore in the dictionary size) grows beyond a manageable size (e.g. beyond 10GB). In future work we will investigate methods to handle these cases by limiting the dictionary size.

8 References

[1] Alexandru Agache et al. “Firecracker: Lightweight virtualization for serverless applications”. In: *17th {usenix} symposium on networked systems design and implementation ({nsdi} 20)*. 2020, pp. 419–434.

[2] Raja Wasim Ahmad et al. “Virtual Machine Migration in Cloud Data Centers: A Review, Taxonomy, and Open Research Issues”. In: *J. Supercomput.* 71.7 (July 2015), pp. 2473–2515. ISSN: 0920-8542. DOI: `10.1007/s11227-015-1400-5`. URL: `https://doi.org/10.1007/s11227-015-1400-5`.

[3] Alaa Alameldeen and David Wood. *Frequent pattern compression: A significance-based compression scheme for L2 caches.* Tech. rep. University of Wisconsin-Madison Department of Computer Sciences, 2004.

[4] Angelos Arelakis and Per Stenstrom. “SC2: A statistical compression cache scheme”. In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 145–156.

[5] Luca Benini et al. “Memory energy minimization by data compression: algorithms, architectures and implementation”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12.3 (2004), pp. 255–268.

[6] Y Collet. *LZ4 lossless compression algorithm.* 2013.

[7] Yann Collet and Murray Kucherawy. *Zstandard Compression and the application/zstd Media Type.* Tech. rep. 2018.

[8] Brendan Cully et al. “Remus: High Availability via Asynchronous Virtual Machine Replication”. In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation.* NSDI’08. San Francisco, California: USENIX Association, 2008, pp. 161–174. ISBN: 1119995555221.

[9] Peter Deutsch. *GZIP file format specification version 4.3.* Tech. rep. 1996.

[10] Peter Deutsch and Jean-Loup Gailly. *Zlib compressed data format specification version 3.3.* Tech. rep. 1996.

[11] Stuart Hacking and Benoıt Hudzia. “Improving the Live Migration Process of Large Enterprise Applications”. In: *Proceedings of the 3rd International Workshop on Virtualization Technologies in Distributed Computing.* VTDC ’09. Barcelona, Spain: Association for Computing Machinery, 2009, pp. 51–58. ISBN: 9781605585802. DOI: `10.1145/1555336.1555346`. URL: `https://doi.org/10.1145/1555336.1555346`.

[12] Muyang He et al. “Reverse Replication of Virtual Machines (rRVM) for Low Latency and High Availability Services”. In: *Proceedings of the 9th International Conference on Utility and Cloud Computing.* UCC ’16. Shanghai, China: Association for Computing Machinery, 2016, pp. 118–127. ISBN: 9781450346160. DOI: `10.1145/2996890.2996894`. URL: `https://doi.org/10.1145/2996890.2996894`.

[13] Moshik Hershcovitch et al. "Lossless and Near-Lossless Compression for Foundation Models". In: *arXiv preprint arXiv:2404.15198* (2024).

[14] Hai Jin et al. "Live virtual machine migration with adaptive, memory compression". In: *2009 IEEE International Conference on Cluster Computing and Workshops*. 2009, pp. 1–10. DOI: `10.1109/CLUSTR.2009.5289170`.

[15] Morten Kjelso, Mark Gooch, and Simon Jones. "Design and performance of a main memory hardware data compressor". In: *Proceedings of EUROMICRO 96. 22nd Euromicro Conference. Beyond 2000: Hardware and Software Design Strategies*. IEEE. 1996, pp. 423–430.

[16] Sparsh Mittal and Jeffrey S Vetter. "A survey of architectural approaches for data compression in cache and main memory systems". In: *IEEE Transactions on Parallel and Distributed Systems* 27.5 (2015), pp. 1524–1536.

[17] Fereydoun Farrahi Moghaddam and Mohamed Cheriet. "Decreasing live virtual machine migration down-time using a memory page selection based on memory change PDF". In: *2010 International Conference on Networking, Sensing and Control (ICNSC)*. 2010, pp. 355–359. DOI: `10.1109/ICNSC.2010.5461517`.

[18] Guangyong Piao et al. "Efficient Pre-copy Live Migration with Memory Compaction and Adaptive VM Downtime Control". In: *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*. 2014, pp. 85–90. DOI: `10.1109/BDCloud.2014.57`.

[19] Horst Samulowitz et al. "Snappy: A simple algorithm portfolio". In: *Theory and Applications of Satisfiability Testing–SAT 2013: 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings 16*. Springer. 2013, pp. 422–428.

[20] Janis Schoetterl-Glausch. "Intel page modification logging for lightweight continuous checkpointing". In: *Bachelor thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, October31* (2016).

[21] Yoshiaki Tamura et al. "Kemari: virtual machine synchronization for fault tolerance". In: (Jan. 2008).

[22] Yingying Tian et al. "Last-level cache deduplication". In: *Proceedings of the 28th ACM international conference on Supercomputing*. 2014, pp. 53–62.

[23] Franco Travostino et al. "Seamless live migration of virtual machines over the MAN/WAN". In: *Future Generation Computer Systems* 22.8 (2006), pp. 901–907. ISSN: 0167-739X. DOI: `https://doi.org/10.1016/j.future.2006.03.007`. URL: `https://www.sciencedirect.com/science/article/pii/S0167739X06000483`.

[24] Daniel Waddington et al. "A case for using cache line deltas for high frequency VM snapshotting". In: *Proceedings of the 13th Symposium on Cloud Computing*. 2022, pp. 526–539.

[25] *Zert0 : Architecture Guide for the IT Resilience Platform.* Feb. 2019.

[26] Fei Zhang et al. “A Survey on Virtual Machine Migration: Challenges, Techniques, and Open Issues”. In: *IEEE Communications Surveys Tutorials* 20.2 (2018), pp. 1206–1243. DOI: 10.1109/COMST.2018.2794881.

[27] Youtao Zhang, Jun Yang, and Rajiv Gupta. “Frequent value locality and value-centric data cache design”. In: *ACM SIGARCH Computer Architecture News* 28.5 (2000), pp. 150–159.