The Australian National University
2600 ACT | Canberra | Australia

Australian National University

School of Computing

College of Systems and Society

ZTera: Extending Java Heaps Using Compressed Memory

— Honours project (S1/S2 2025)

A thesis submitted for the degree
Bachelor of Advanced Computing R&D

By:
Adnan Hasnat

Supervisor:
Dr. Shoaib Akram

November 2025

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the Academic Integrity Rule;
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or LMS course site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

I acknowledge that I am expected to have undertaken Academic Integrity training through the Epigeum Academic Integrity modules prior to submitting an assessment, and so acknowledge that ignorance of the rules around academic integrity cannot be an excuse for any breach.

November (2025), Adnan Hasnat

Acknowledgements

There are many people I could not have reached the end of this journey without.

Thanks to my supervisor Shoaib Akram for the many late-night discussions and constantly checking up on me.

Thanks to Jack Kolokakis, who helped me considerably with the TeraHeap portion of this research as one of the developers of the original system, and whose patient help and advice saved me many hours of frustration. Best of luck to you, as well, on your own PhD thesis.

Thanks to the friends who gave me sound advice and support even while they were conquering challenges of their own: Alex Mirrlees-Black, Jasper Quirk, Brightony Yap, Kshama Patel, Alyssa Herald and Hayley Patton. Thanks especially to Jan Powell-Davies, for all of the wise words you gave me from your years of research experience, and helping me believe I could make it to the end. I hope all of you can reach your ends too.

Thanks to Deeksha Nagpal and Deborah Short. You guys do amazing work.

And thanks to my parents, for everything.

Abstract

Recent work at Google and Meta has explored the concept of using compressed memory as a tier for offloading cold data. Prior literature claims that compressed memory provides a middle ground between DRAM and SSD with regard to performance, and that the overhead of compression and decompression is far lower than accessing storage.

In this work, we explore the application of compressed memory tiers to the Java heap. We extend the existing work TeraHeap, which demonstrates adding a second heap space H2 designed for use on block devices. TeraHeap fences H2 from tracing to avoid expensive GC scans over the device. However, the cost of accesses to H2 can be expensive if it is backed by a slow medium, and may be mitigated using compressed memory.

Firstly, we perform synthetic benchmarking on a ZRAM compressed memory device. We find that ZRAM outperforms SSDs in read performance, but that extracting competitive write performance while maintaining good compression ratios requires up to 16 threads. Secondly, we propose a design called ZTera for integrating ZRAM into TeraHeap, using the writeback mechanism to offload incompressible and cold data to overcome the capacity limitations of ZRAM. A naive two-tier version of ZTera is evaluated. We find that even when H2 is able to fit within ZRAM only, ZTera does not perform well due to parallelism bottlenecks in TeraHeap's H2 migration. We conclude that ZRAM is not a straightforward solution for extending heap capacity in Java.

Table of Contents

Chapter 1

Introduction and Motivation

Main memory is a critical resource in modern datacenters. Demand for memory continues to increase due to its high performance capabilities, which incentivises applications to retain data in memory as much as possible. Conversely, hardware architects have faced difficulties in shrinking dynamic random access memory (DRAM) cells further (Lee (2016)), which has kept DRAM cost efficiency from improving. These effects have made DRAM expensive in today's world. Meta reports that 33% of its server costs and 38% of server power costs are expected to be attributable to DRAM in the near future (Weiner et al. (2022)). Finding opportunities to save memory without sacrificing application performance is hence immensely valuable.

The classical mechanism called swapping uses hard disks and other persistent block devices to store data when memory runs out. However, swapping is known to significantly degrade application performance as storage devices are extremely slow compared to memory. The recent development of fast solid state drives (SSDs) and Peripheral Component Interconnect Express (PCIe), which provides high bandwidth links to the CPU, mitigates access cost significantly. However, modern DRAM access latency remains 500× faster than SSD (Cheong et al. (2018)), and improvements in memory and double data rate (DDR) transfer speeds have kept pace with SSD developments.

One method of mitigating the effect of swapping is to prioritise evicting data which is not accessed frequently by applications. This can be achieved by refining the eviction policy of the swapper, otherwise called the memory reclamation logic. The Linux kernel's default policy is an approximate version of a least-recently-used (LRU) algorithm (Torvalds et al. (n.d.), `/mm/workingset.c:21-180`). However, Lagar-Cavilla et al. (2019) indicate that swapping is flawed due to being *reactive*. It only activates when memory is almost full and does not attempt to ensure that swapping occurs at a time which is not performance-sensitive for applications. It also chooses how much to swap based on requirements for creating additional memory capacity, neglecting performance impacts.

Lagar-Cavilla et al. (2019) and Weiner et al. (2022) have recently proposed systems that address these issues, called gswap and TMO respectively. Both systems rely on profiling application memory usage at a more detailed level, and make more informed and proactive decisions for offloading cold memory. The primary contribution of both systems is a method for automatically detecting which memory is cold and deciding how much to offload. We particularly note the choice of medium for storing offloaded memory in these systems. Rather than use SSD, gswap chooses to use compressed memory, which allows more logical data to fit within the same physical space at the added cost of compression and decompression. Lagar-Cavilla et al. implicitly assume that SSD transfer costs are too high for it to be effective for offloading without sacrificing application performance, instead seeking a tier they call "far memory" that can strike a balance between the capacity of SSD and the performance of memory. They indicate that using the Linux kernel mechanisms zswap and zram, which can act transparently to the application, compressed memory becomes an appealing option which is easy to deploy with no additional hardware cost. TMO supports the use of both SSD and zswap, with the system dynamically scaling the amount to offload based on the performance impact of the backend. They note in a case study with a specific application that SSD only allows 4% of data to be offloaded without unacceptable sacrifices to application performance, while zswap allows 13% of data to be offloaded when it is compressible. Hence, compressed memory is promising as a slow memory tier.

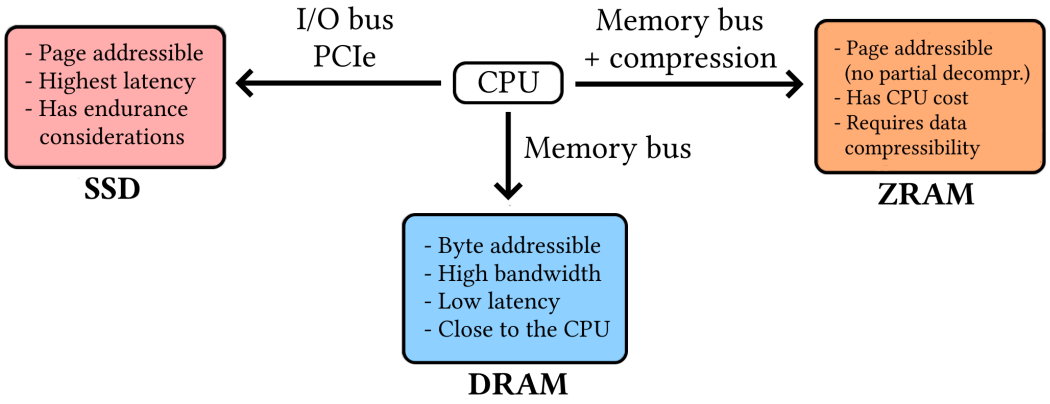

Figure 1.1: Characteristics of DRAM, SSD and ZRAM (compressed memory) as memory tiers

While gswap and TMO work well for general applications, they have limitations when used for Java applications. These systems work transparently to applications, which benefits usability as developers do not need to make any changes to existing programs to benefit from these systems. However, this comes at the tradeoff of losing insight into the behavior of applications. In a managed runtime like Java, garbage collection (GC) scans run periodically to remove objects which the program will definitely not access again. GC involves traversing references from a selected set of root objects such as

local function variables, and calculating the set of all objects which are reachable from the roots and hence may be accessed. This process poses two problems for transparent offloading:

1. If full garbage collection is frequent, then the GC accesses every object in the heap frequently. Transparent offloading concludes that all application data is hot, and avoids offloading. However, the mutator (non-GC) thread may in fact only access some data rarely, leading to missed opportunities.

2. If full garbage collection is infrequent, some objects may be offloaded. However, when GC is once again triggered, it traverses all objects **including offloaded ones**. Both SSD and compressed memory require data to be accessed at a certain granularity:

   - For the former, SSD is internally physically organised as blocks that must be accessed atomically.

   - For the latter, compressing data requires exposing redundancy across different data and any individual byte of compressed data might be associated with many different independent bytes of uncompressed data. Partial decompression is hence typically not possible and retaining granular access requires compressing in blocks.

   On the other hand, Java objects are variable sized and may be small or occur across block boundaries. Additionally, GC accesses can be highly random and it is unlikely for the GC to access the other objects within a block immediately after accessing one of them. These accesses ultimately results in significant read/write amplification and diminishes performance. This phenomenon is illustrated in Figure 1.2.

These two issues ultimately make TMO and gswap inappropriate for use with Java applications.

A new solution to this issue is proposed in Kolokasis et al. (2023) in a system called TeraHeap. The key contribution of this system is to define a second memory tier within the Java Virtual Machine (JVM) as a second heap space called H2, and carefully manage garbage collection over this new space to avoid amplification. H2 is managed as a region heap with only light reference-counting garbage collection. The regular parallel scavenge GC still runs in H1 (the memory tier), but is fenced from scanning H2. This design leverages JVM-specific knowledge to effectively combat the problems faced by fully transparent offloading.

However, TeraHeap only partially mitigates the costs of accessing H2, which typically resides on SSD. In applications which already move data off heap to avoid running out of space, TeraHeap eliminates serialisation costs by allowing data to be retained in the memory address space directly. Additionally, H2 data is only lightly garbage collected, and TeraHeap's base design reserves long-lived data for H2 which would otherwise incur

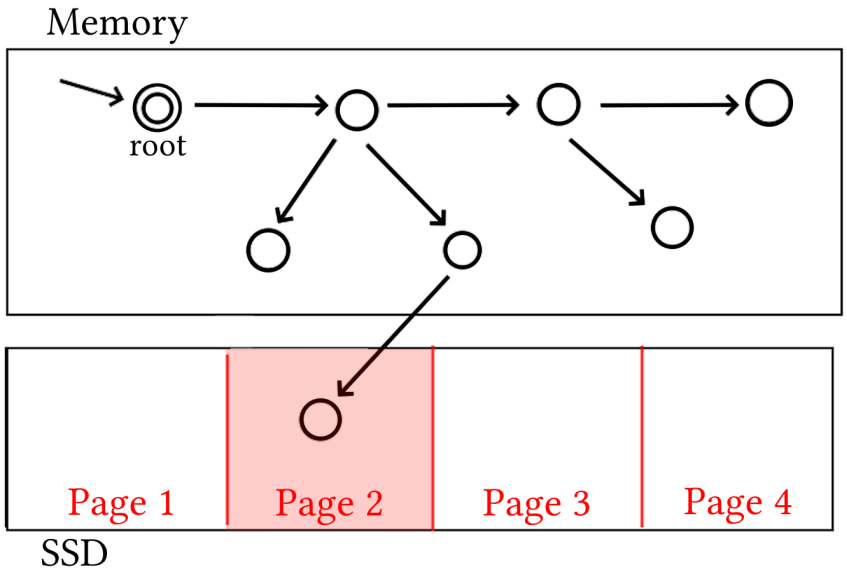

Figure 1.2: Illustrating the problem with naive garbage collection over Java heaps split across memory tiers. GC scan will reach the single small object shown in SSD, and result in reads/writes that affect the entire page.

redundant GC scans. Hence, TeraHeap's migration policy and design is focused on GC overheads, ignoring the performance penalties that may occur from frequent accesses to hot data in H2. The Linux kernel's page cache stores some of H2's data in memory and mitigates access costs, but does not fully eliminate them.

There is potential in introducing compressed memory as a medium for H2. If data is compressible, compressed memory may provide an alternative method of expanding total heap space while incurring a more acceptable penalty for H2 accesses. However, compressed memory is significantly limited in capacity compared to SSD. Recent work (Kolokasis et al. (2025)) has added facilities to TeraHeap to more efficiently manage H2 by allowing regions to be moved back to H1 for garbage collection. Nevertheless, if H2 contains a large volume of live data, compressed memory will still be exhausted long before SSD, and simply replacing H2 with compressed memory limits the size of applications which can use TeraHeap. Hence, SSD will likely need to remain in the system to provide capacity, and one must be judicious in choosing the subset of data that is cached in compressed memory.

In this thesis, we explore how to integrate compressed memory effectively into TeraHeap. We approach this problem from an analytical perspective by developing an understanding of compressed memory in more simple settings before building up to a full system, so as to leverage insight to inform our design. The first half of the thesis' contribution is a detailed

benchmark of a ZRAM compressed block device compared to SSDs using synthetic I/O workloads. These experiments include testing with data from dumps of real Java heaps, specifically from executions of the DaCapo Chopin microbenchmarks (Blackburn et al. (2025)). The second half of the contribution concerns integration of compressed memory into TeraHeap to form a system we call ZTera. We consider various design aspects associated with using compressed memory, including where to place incompressible data and how to prioritise data for compressed space. We then perform an evaluation of a simple two-heap version of the system, which discovers bottlenecks even if one assumes that compressed memory as H2 has sufficient capacity to support the application. We find that despite literature indicating the efficacy of ZRAM as a middle tier, it is more challenging than it appears to apply it effectively.

Chapter 2

Background

2.1 Garbage Collection

We provide some minor background on garbage collection in Java. Modern garbage collectors typically use a generational heap divided into a young generation/nursery, and an old generation of mature objects. This design is based on the insight that programs tend to create a large amount of temporary data which can be collected quickly, which is called a minor GC. Objects which repeatedly survive minor GC demonstrate that they are long lived, and are moved into the mature space to avoid being checked during minor GCs. When the total heap including mature objects begins to become full, this triggers a major GC which includes both generations.

There are additionally many choices of garbage collector with varying levels of sophistication and associated policies. The simplest type is sequential stop-the-world allocators. In this regime, normal non-GC application threads, called the mutators since they act on data and change the liveness of objects, are completely stopped during GC. These allocators also avoid synchronisation challenges by only using a single thread for GC. The next step is to introduce parallelised GC with multiple threads, as seen in Java's Parallel Scavenge allocator. Finally, one can also allow the mutator to run concurrently with GC to avoid application progress being completely stopped from GC, as seen in Java's G1 allocator (Oracle Corporation (n.d. (1)).

Parallel and concurrent collectors often introduce read barriers and write barriers, code which is set to trigger whenever a read or write occurs in the JVM. Such barriers are expensive due to the frequency with which they activate (every object read/write) and hence avoided when possible, but are sometimes necessary to detect mutator operations immediately for the purpose of ensuring correctness (Jones et al. (2023)).

The current implementation of TeraHeap, which we build on, uses the parallel scavenge collector. G1 integration into TeraHeap is an ongoing project (Kolokasis et al. (2024)), and may be added to this work when available.

2.2 TeraHeap Architecture

This section elaborates on the details of the base TeraHeap architecture.

TeraHeap's design considers three primary problems:

1. How to identify objects suitable for moving to H2?
2. How to reclaim space in H2 while minimising garbage collection overhead?
3. How to avoid reclaiming H1 objects which are kept live by references from H2 objects (backward references) without scanning H2?

2.2.1 How to Identify Objects Suitable for Moving to H2?

TeraHeap ultimately opts for manual identification of H2 objects, but with two important design choices: moving the identification decisions into a software framework that runs within the JVM, and minimising the manual effort required by needing only roots to be tagged. Firstly, TeraHeap is designed for use with large software analytics frameworks such as Spark and Neo4J, which provide rich libraries and APIs to Java programmers for specialised tasks such as scientific computation and graph analysis. Since these frameworks often manage much of the data created by the applications, while manually deciding which objects are best to offload is costly for a specific framework, it is amortised by the fact that the same framework can be used by a programmer for many different applications. Secondly, the interface exposed by TeraHeap consists of two simple commands: `h2_tag_root(object, label)`, which tags a small selection of root objects with label identifiers, and `h2_move(label)`, which hints to the system to move all roots with the associated label as well as all objects reachable from those roots (the transitive closure) to H2. The ability to handle roots in bulk atomically, and to trace references to identify associated objects automatically, greatly eases the burden on the system developer.

The manual identification used by TeraHeap currently prioritises long-lived objects, since garbage collection over H2 is infrequent and it should be reserved for objects unlikely to require GC. Memory is split between H1 and the page cache, with the latter providing fast access to hot objects in H2.

In addition to explicit move instruction with `h2_move`, TeraHeap also monitors the long-lived object content of H1 by considering how many objects survive full GC. If more than 85% of objects remain, automatic movement of marked objects to H2 is triggered in order to avoid H1 from becoming full.

2.2.2 How to Garbage Collect H2?

The key insight is that H2 can be organised into regions that are collected in bulk with only light garbage collection in place, as shown in Figure 2.1. The labelling based interface complements this design by allowing labels to be assigned to objects with similar lifetimes, which are then grouped into the same region. TeraHeap manages a dependency list for each region which tracks which other regions it contains references to. This list is created when moving a new region to H2 by considering existing references to other H2 objects, and tracks all H2 cross-region references. The other source of references that may keep a H2 region alive is from objects in H1 (forward references), which are identified when major GC occurs in H1. Upon discovering a forward reference, the GC avoids following it and instead just sets the live bit of the region and any other regions reachable from it from cross region references. At the end of major GC, all regions which have not been marked are cleared.

Managing data in bulk suits the block-based structure of H2 mediums and retains performance while retaining the ability to collect some garbage. There is a tradeoff here as not all dead objects in H2 are collected, and incorrectly choosing allocation of objects to regions will result in mostly dead regions which are not collected due to containing a small number of live objects. This problem must be considered when using TeraHeap.

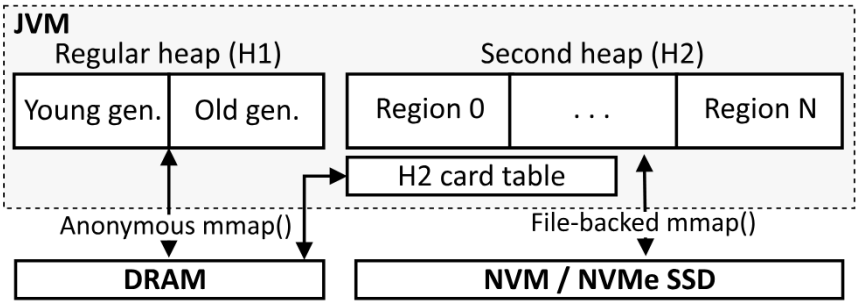

Figure 2.1: High level overview of base TeraHeap architecture. Figure by Kolokasis et al. (2023) (Section 3.1).

2.2.3 How to Handle Backwards References from H2 to H1?

While H1 objects can be traversed and garbage collected mostly identically to a base JVM, the introduction of H2 introduces the possibility that H1 objects are kept live by references from live H2 objects. To handle this case, TeraHeap maintains a card table, which allows it to minimise scanning for backwards references to only H2 regions which have been modified by the mutator. The card table has an entry for each region and stores one of four states: (1) clean, (2) dirty, (3) youngGen and (4) oldGen. All regions begin as clean. A post-write barrier is introduced which intercepts the mutator when it writes to objects in H2, and marks the relevant region's card table entry as dirty. Then, during GC, only dirty H2 regions are checked as follows:

1. If the region contains no backward references, mark as clean.
2. If the region contains references to young generation objects, mark as youngGen.
3. Otherwise, if the region contains references to old generation objects, mark as oldGen.

The youngGen/oldGen distinction is made to reduce the number of H2 regions that need to be considered during minor GC, i.e. GC of young generation H1 objects, which occurs often and is hence worth optimising for. The card table transition diagram is shown in Figure 2.2. There are additional design aspects involved in ensuring that the card table can be accessed in parallel by GC threads, but these are outside the scope of this thesis.

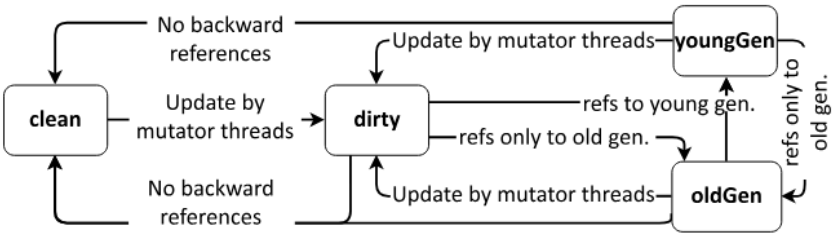

Figure 2.2: Card table transitions in TeraHeap. Figure by Kolokasis et al. (2023) (Section 3.3).

Card tables allow retaining the correctness of GC by preventing live objects from being reclaimed erroneously, while balancing performance considerations.

2.2.4 Major GC Stages

We have discussed the major data structures and architectural aspects involved in the design of base TeraHeap. We now proceed to outline the steps involved in Major GC in TeraHeap, specifically focusing on the version implemented for JDK 17 as outlined by Kolokasis et al. (2024). These steps will be relevant to our performance analysis, and to designing how to integrate compressed memory into the system.

The original parallel scavenge major GC consists of three phases: marking, summary and compact. In the marking phase, all objects in the heap are scanned starting from the roots and live objects are marked as they are encountered. Bitmaps are used to enable fast and cheap marking with minimal CPU operations. In the summary phase, the heap is divided into 512 KB regions and a destination address is calculated for each region. In the compact phase, objects that point to regions to be moved have their references updated to the new locations, and the regions are copied in parallel to their destinations. The summary and compact phases copy live objects and overwrite spaces which originally belonged to now dead objects, thus reclaiming their space while ensuring that the heap does not become fragmented from gaps created by dead objects. These stages together form a parallel mark-and-compact algorithm.

TeraHeap provides the following extensions to the phases, adding one extra precompact phase and extending operations during the other phases:

- Marking phase: objects designated to be moved to H2, either by previous `h2_move` hint or chosen to alleviate memory pressure, are marked by following references from the roots. Candidate objects are stored in an additional bitmap.
- Summary phase: a forwarding table is constructed which stores a destination address for each H2 candidate, which is used to update the references in H1 objects that point to candidates. Destination addresses are calculated using a custom allocator optimised for this use case. H1 metadata associated with H2 candidates is adjusted to account for the object no longer being there.
- Precompact phase: candidate objects are moved to H2, and backwards references from H2 candidates to H1 objects that are moved as part of the regular H1 compaction are updated.
- Compact phase: proceeds as normal.

2.3 ZRAM and ZSwap for Transparent Compression

We elaborate on the mechanisms for transparent compression in the Linux kernel.

There are two systems which have rather different interfaces and methods of invocation: ZRAM and Zswap. Zswap allows compressed memory to be used as a swap space and is implemented by directly interfacing with the Linux memory controller's page reclaim functions (Torvalds et al. (n.d.), `/mm/page_io.c:275` among others). Zswap's execution is handled by the `kswapd` kernel daemon, and its functions are invoked when the kernel is swapping out memory to compress the data instead of writing to a swap disk. On the other hand, ZRAM exists as a virtual block device. The kernel implements it within the virtual block layer by defining block I/O (BIO) functions such as `__zram_bvec_read` in the ZRAM driver (Torvalds et al. (n.d.), `/drivers/block/zram/zram_drv.c:1227`). The user creates a ZRAM block device and mounts a filesystem on it (Senozhatsky et al. (n.d.a)), and can thereupon perform I/O on files on the device identically to other block devices (e.g. SSD). We choose ZRAM as our focus for integration into TeraHeap for the following reasons:

1. It is highly flexible with regard to choosing what data to write to the compressed memory pool. Conversely, Zswap relies on the existing page reclaim (eviction policy) which is implemented within the kernel. ZRAM instead allows us to choose from user space.
    - Zswap does have additional policies where it can choose to write data which is incompressible or does not fit within the compressed memory pool to a backing device, e.g. SSD. However, these features are matched by ZRAM's writeback features (Pham et al. (n.d.); Senozhatsky et al. (n.d.a)).

2. ZRAM's interface is identical to SSD, allowing for little modification to systems already designed for block devices.

It is notable that ZRAM only compresses data in units of pages, which is because the underlying allocator implementation `zsmalloc` manages data as linked lists of minimum size pages (Senozhatsky et al. (n.d.b)). Zsmalloc accepts the compression losses from not being able to compress across page boundaries for the benefit of avoiding failures from trying to allocate contiguous blocks when space is low and memory pressure is high. Other allocator choices such as Z3fold and Zbud existed in the past, but were deprecated recently due to poor performance (Larabel (2025)).

2.4 SSD Architecture

We provide an overview of SSD architecture, with a focus on performance considerations that will impact our I/O benchmarks, such as read/write asymmetry and internal parallelism.

An SSD consists of an SSD controller and a flash array, with the array being divided into a hierarchy of levels which is shown in Figures 2.3 and 2.4. Each cell is a MOSFET floating gate transistor, and stores data by persistently storing a charge that dictates whether it draws current or not when applying a voltage below a certain threshold. Reading cells is relatively simple by applying voltages to transistor gates using wordlines, and reading outputs on bitlines. However, while the operation of "programming" (i.e. writing 0s to cells by injecting charges) is easy, erasing cells by ejecting their charges requires applying high voltages. Ejection is an intensive operation that cannot be done as precisely as reading or programming, and over time causes damage to cells. Reading and programming hence occur at a different granularity compared to erasing, specifically at sub-blocks compared to blocks. SSD controllers maintain a logical-to-physical address mapping to allow out-of-place updates, i.e. moving the updated data to a new location and updating its mapping. They also apply wear levelling by distributing writes across the device, and occasionally conduct garbage collection to compact gaps and dead sub-blocks caused by updates. While these mechanisms mitigate the latency of writes, the asymmetry ultimately causes write performance to be generally worse than reads (Chen et al. (2016)).

Beyond pages and blocks, there are several further levels of hierarchy which enable various forms of internal parallelism. In fact, Chen et al. (2016) argue that these are fundamentally required for SSDs to deliver competitive performance. SSDs exhibit:

- Plane-level parallelism: SSDs can perform the same operation (e.g. writes or reads) across multiple different planes at once. The parallelism in this case is limited by the fact that planes share instruction decoders (Mutlu (2023)).
- Die-level parallelism: SSDs have multiple dies which can perform commands completely independently.

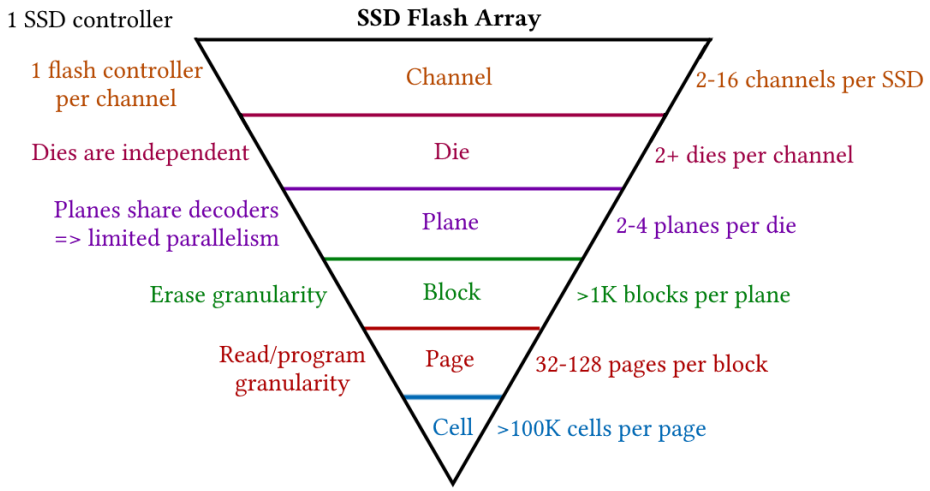

Figure 2.3: Hierarchy of levels in an SSD flash array. Labels on the left side give descriptions of various properties of each level, and on the right side describe how many form the next level.

- Channel-level parallelism: Dies are both interleaved within channels to saturate their bandwidth, and parallelised across multiple channels to combine their bandwidth. Each channel also has its own error correction control by the use of separate hardware flash controllers.

SSDs have internal SRAM and DRAM caches for various operations. While DRAM is shared between both data and metadata, the logical to physical translations which are stored there tend to take up most of the available space (Mutlu (2023)). Nevertheless, these may alleviate some pressure on SSD cells.

There are further performance considerations associated with modern SSD developments, such as the use of 3D architectures and cells which can store several bits by maintaining multiple different voltage thresholds (Yoon et al. (2017)). Such cells are distinguished from single-level cells (SLCs) and range from double level (MLC) to quadruple level (QLC). There are tradeoffs involved where increasing the number of levels affects reliability and requires more error correction mechanisms, which must be offset by raw bandwidth gains (Mutlu (2023)). Detailed consideration of these factors is outside of our scope.

It is notable that sub-blocks are sometimes called "pages", which usually do not correspond with the Linux unit also called "pages". When discussing pages or page sizes in any following text, we refer to the Linux default unit of 4 KB.

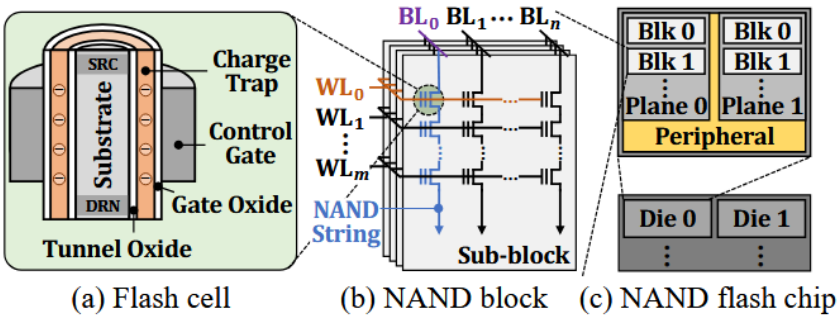

Figure 2.4: Physical view of SSD hierarchy. Pages are called "sub-blocks" in (b). WL and BL refer to wordlines and bitlines, which are used as select and data lines respectively. Figure by Park et al. (2021).

2.5 I/O Mechanisms and Performance Considerations

We provide a brief overview of various choices of synchronous and asynchronous I/O mechanisms for block devices in Linux, and some description of the associated performance considerations. This background will be relevant especially in our raw I/O benchmarks of ZRAM and SSD in the first half of the thesis.

2.5.1 Synchronous I/O

The two most basic methods of invoking I/O are system calls (syscalls) and memory mapping (mmap).

Syscalls generate signals that trigger a context switch into the kernel, which takes care of handling the I/O request and interfacing with the actual device. The `open` syscall is used to specify a path to a file to open for I/O, which is converted to a file descriptor that identifies the file object within the kernel. The user then invokes `read` and `write` syscalls, providing a pointer to a user space buffer containing or to contain the data to be transferred. If the `O_DIRECT` option is provided, the kernel directly transfers this data to/from the device. If not, it additionally copies the data into a kernel space buffer which is retained after the I/O is complete, so that if the user asks for the same data again it can be served from memory instead of having to go to the device. Data cached from files in this way is said to be stored in the page cache. Use of the page cache is important for performance, but direct I/O serves two important purposes: skipping the overhead of the extra kernel space copies to minimise latency or otherwise, and benchmarking device I/O directly. We often control the amount of memory available to the application using Linux `cgroups` to control the size of the page cache.

Memory-mapped I/O instead relies on the virtual memory mechanism of modern CPUs. The user invokes the `mmap` system call with a file descriptor to create a memory mapping. Thereafter, reading and writing from addresses within the memory mapping (i.e. with

load and store instructions) will return data directly from the file, which allows the user to interface with file data identically to memory data. In reality, the program is accessing virtual addresses which are translated by a combination of hardware and software mechanisms to physical locations in memory, and the "page fault" mechanism is used to inform the kernel to fill memory (specifically page cache) with data from the file. All device I/O hence only happens on demand. Additional mechanisms exist to handle multiple process retaining several mappings of the same file.

2.5.2 Asynchronous I/O

Syscall and mmap I/O are notably both synchronous mechanisms, and the executing thread is stalled waiting for the I/O to complete. There are additional mechanisms in Linux to execute I/O asynchronously.

The LibAIO library provides wrappers for two system calls `io_submit` and `io_getevents`, for submitting async requests and retrieving completed requests respectively. The CPU is free to do other work in between submitting and waiting for a request. LibAIO suffers, however, from two issues (Didona et al. (2022)):

1. While multiple I/O requests can be batched into one invocation of `io_submit`, submitting and waiting for a request always involves two separate system calls which can be expensive.
2. LibAIO only supports direct I/O.

As of Linux version 5.1, a new interface called `io_uring` has also been introduced which provides powerful facilities for minimising system call overhead (Corbet (2019)). IO_uring exposes two queues which are shared between kernel and user space: the submission queue and the completion queue. The user directly updates the submission queue from user space and uses the `io_uring_enter` system call to notify the kernel of new submitted requests. The user can then either use the same system call invocation to request the kernel to poll the completion queue, or itself poll the completion queue from user space. Alternatively, the kernel can be requested to spawn a thread to poll the submission queue automatically, resulting in no system calls being required outside of setup (Didona et al. (2022)).

Chapter 3

Related Work

Memory tiering is extensively studied in literature, but generally in settings which are different from ours. Much of related work is either implemented at a different layer in the software stack (kernel/hardware), or focuses on byte-addressable tiering backends. Kernel and hardware level tiering systems lack awareness of the JVM and GC, which diminishes their effectiveness for Java applications. Additionally, byte-addressable backends allow the GC to access individual objects, while coping with the read/write amplification associated with block-addressible backends is a core part of our motivation. Hence, while these backends include promising technologies such as Non-Volatile Memory (NVM) and Compute Express Link (CXL) for coherent access to remote memory, work focusing on them is orthogonal to our focus.

To the best of our knowledge, there are no works that extend the heap with block addressable mediums which are directly comparable to ours. However, we will discuss similar systems in other settings and extract aspects that are relevant to our work.

3.1 JVM-Level Tiering

There is some relatively old work that considers storing heap objects on block storage devices, but for the purpose of handling memory leaks. The systems LeakSurvivor and Melt use hard disks to swap out data identified as potentially leaked (Bond and McKinley (2008); Tang et al. (2008)). Correctly predicted leaks do not incur disk access costs since the program never accesses this data again, while mispredicted leaks can be recovered from by swapping data back in from disk. This work does demonstrate the interaction of offloading and garbage collection overhead which is also important in TeraHeap, but is not appropriate for capacity expansion.

There is another recent group of papers focusing on disaggregated systems, an architecture used in cloud settings where compute and memory resources are both split across

several servers. A primary theme in this work is that in compute nodes running Java applications, data may have to be retrieved from a remote node. However, local memory caches do not perform well because the garbage collector ruins locality of accesses. Wang et al. (2022) also observe that the garbage collector complicates the pattern of memory accesses, which can disrupt OS prefetching algorithms. Semeru and Mako resolve this issue by offloading garbage collection tasks onto memory servers, avoiding the need to bring data to local nodes for GC and eliminating cache pollution (Wang et al. (2020); Ma et al. (2022)). Memliner takes a different approach by changing the order of garbage collection so that mutator and GC accesses have better locality compared to each other (Wang et al. (2022)). While the problem which GC poses for the naive hotness identification algorithms used by OS remote memory caches is similar to our own motivation, these system are generally not applicable to our problem because they also rely on remote compute resources. In our setting, we only have remote memory. A newer system called Polar focuses more heavily on the memory aspect and primarily focuses on hotness/coldness identification for Java objects (Nguyen and Nguyen (2024)). Polar will be discussed later alongside similar algorithms used in kernel-level tiering.

An additional category of work is tiering which uses NVM as the second medium. Crystal Gazer uses write-rationing GC, a technique which prioritises objects that are updated less often for the second tier, in order to account for the limited write endurance of NVM mediums (Akram et al. (2019)). Espresso provides the `pnew` keyword to programmers to allow defining persistent objects, and supports both coarse and fine grained persistence (Wu et al. (2018)). As discussed prior, this work is orthogonal to ours due to using byte addressable mediums that do not pose the same challenges we tackle. Additionally, this work focuses on supporting persistence rather than volatile data capacity.

Considering our focus on ZRAM and compression, there is some work that discusses compression of Java heaps. Sartor et al. (2008) develop a taxonomy of object-specific compression algorithms, which work with the existing structure of objects (i.e. structs containing fields). Object-based compression includes eliding object instances that are identical in all fields, compressing fields using dominant value compression by storing a small subset of potential values instead of the original value, and many other methods. The benefit of object-level compression is retaining the ability to access and update individual objects, allowing these strategies to work well within the JVM without incurring amplification from having to decompress unrelated data to access specific objects. Sartor et al demonstrate that object-level compression can achieve 50% compression while retaining fine granularity access, and provide a baseline suggesting agnostic compression with `bzip2` can achieve even higher ratios of 72–99%. We focus on applying ZRAM transparently using infrastructure already designed to handle page-level granularity access. However, object-level compression can complement our work well and is a consideration for extending the schemes we propose to achieve better write performance over compressed memory in the JVM

3.2 Kernel-Level Tiering

There are two problems commonly focused on by kernel-level systems which are relevant to our work:

1. Identifying hot and cold data to choose which data to prioritise for the fast tier and which data to relegate to slower tiers, which includes associated demotion/promotion policies.
2. Controlling migration across tiers. One aspect is choosing how much to migrate to limit the impact on applications, both directly from migration and indirectly through increased access costs to migrated data. Another aspect is how to make migration itself efficient.

We also discuss how page sizing and the use of Linux huge pages interacts with tiering, which is a common focus in kernel-level systems.

3.2.1 Identifying Hot and Cold Data

Most systems evaluate the hotness of data at a page granularity and use either the frequency of accesses or the time since last access to each page as the primary underlying metric for hotness. This metric requires tracking accesses to pages in the kernel, for which there are three primary mechanisms: scanning the page table, using page faults and using hardware events-based sampling.

Page table entries (PTEs) contain an accessed bit which is typically set by the hardware memory management unit, and is cleared by software. These are used by the Linux kernel already to implement the page cache eviction algorithm (Gorman (2007)), but can also be used by custom kernel mechanisms. TMTS uses the accessed bit to track the time since last access to each page, adding an age field within each PTE's metadata and updating it using a kernel daemon. This field directly indicates how cold a page is in the past, and is used to estimate chance of it being accessed again in the future (Duraisamy et al. (2023)). Lee et al. (2023) and Maruf et al. (2023) note that PTE mechanisms are costly due to requiring scans over the entire page table. Additionally, accessed bits for some pages may have only been updated in the translation lookaside buffer (TLB) which caches PTEs for fast address translation. Hence full accuracy of hotness identification using accessed bits requires invalidating TLB entries. Several works have proposed mechanisms for reducing these costs. Thermostat aggregates page access counts over 2MB granularities by using a small sample of 4KB pages within each huge page, while retaining additional mechanisms to correct misclassified pages (Agarwal and Wenisch (2017)). Telescope exploits multi-level page tables, which organise PTEs into a hierarchy of progressively finer granularity tables and is required in modern systems to reduce the size of the page table. By tracking hotness at higher levels in the hierarchy, one can approximate hotness for sublevels (Nair et al. (2024)). TMTS chooses to avoid invalidating TLB entries, claiming that the loss of accuracy is negligible compared to the performance improvement (Duraisamy et al. (2023)).

An alternative mechanism is to make use of page faults. In a swap-tiering system, swapped data is accessible from the virtual address space but is not present in memory, instead residing in the slower tier. Accessing swapped data triggers a major page fault where it is copied immediately back into RAM and promoted. TMTS notes that its predecessor gswap relied on this mechanism, but that it is flawed due to incurring a significant immediate latency. Additionally, infrequently accessed data is incorrectly assumed to be hot as soon as a new access arrives. Other systems such as TPP and AutoTiering resolve the latency problem by using minor page faults associated with a Linux kernel feature called AutoNUMA. In a NUMA system, remote pages are periodically marked as inaccessible in order to test accesses to nonlocal data. Accessing the pages triggers a minor page fault which requires only updating the PTE and does not require data copy, providing a hint to the system to migrate the pages locally for faster access. Since this feature provides a promotion hint instead of directly initiating promotion, it is much more lightweight. TPP and AutoTiering apply NUMA hints in the context of CXL memory tiering (Maruf et al. (2023), Kim et al. (2021)).

Yet another alternative is to use hardware event-based sampling. Intel's Processor Event-Based Sampling (PEBS) and AMD's Instruction-Based Sampling (IBS) systems provide facilities for tracking cache miss events and tracing them back to the memory accesses that triggered them (Intel Corporation (n.d.), AMD Corporation (n.d.)). By extracting the addresses of accesses, hot pages can be identified. By controlling the sampling frequency, event based sampling can become cheaper than other methods. However, sampling resolution becomes a tradeoff against CPU overhead. TMTS filters events to loads accessing the slow tier specifically and exclusively uses this method for hot page promotion (Duraisamy et al. (2023)). Additionally, TMTS only samples 1% of loads, and uses periodic page table walks to supplement this method. HeMem limits the sample period to 5000 memory accesses, indicating that this frequency strikes a sufficient balance between fidelity and overhead (Raybuck et al. (2021)).

Many tiering solutions in fact employ several of these access tracking methods simultaneously. TMTS notes that identifying pages which are in the slow tier but are now becoming hot must be done quickly to allow the system to adapt to sudden changes and avoid bursts of accesses from damaging performance. Conversely, identifying cold data can be performed lazily. Hence, sampling methods are used to make promotion policies more proactive, while demotion is controlled via low frequency page table walks.

There also exist hotness identification methods designed to work in the JVM. Polar creates a hotness-segregated heap with local and remote memory as the two heap mediums, and measures hotness using the metric of number of GCs since last access to an object (Nguyen and Nguyen (2024)). This metric allows the aggressiveness of offloading to automatically scale with memory pressure, as more frequent GC will result in objects aging more quickly. Hotness is tracked for each object in the object header, and is updated via a combination of GC-level and mutator-level operations. More specifically, the GC increases each object's count as it scans, while the mutator uses existing read/write barriers to clear an object's count if it is accessed. Polar is unique compared to ker-

nel systems for tracking hotness on an object level basis, arguing that page-granularity tracking suffers from inefficiencies due to possibility for hotness to be skewed between objects within the same page.

New work proposes methods of identifying objects that go beyond hotness by considering the performance impact of accesses in addition to number/time of accesses. Liu et al. (2025) note that modern systems use memory level parallelism (MLP) to hide the latency of some accesses when the memory is under load, which is not considered by traditional hotness identification. Liu et al propose the metric of amortised offcore latency (AOL) which is calculated using existing performance counters. They present two systems SOAR and ALTO which apply AOL to suggest page migrations and can be integrated into existing tiering systems.

Comparing this aspect with our own work, we are also interested in adding a coldness/hotness mechanism in ZTera to prioritise data for compressed memory. Base TeraHeap does not have automatic mechanisms for identifying objects to move to H2, instead relying on manual tagging. This method does not work well for coldness/hotness because these parameters are dynamic and can change across the course of the program. However, static mechanisms may be appropriate for identifying compressible data, which can be relegated for compressed memory. Many kernel-level mechanisms are not appropriate for the JVM due to being unable to consider individual objects, but a technique similar to Polar's may be applicable.

3.2.2 Controlling Migration Across tiers

Many related works focus on controlling the amount migrated to slower tiers to limit the cost incurred by applications accessing data on slower mediums to meet a particular performance objective. Gswap and TMTS both define a metric measuring the rate of accesses to the second tier, called promotion rate in the former and Secondary Tier Access Ratio (STAR) in the latter. TMTS notes that it is desirable to offload as much data as possible to increase space for more applications to fit in the same memory, while limiting STAR to reduce the impact of slow accesses. They perform a systemwide analysis to measure the effect of various levels of STAR on performance to find a static threshold for acceptable STAR, which is 0.5% for their setup. TMTS then keeps track of a promotion histogram that allows calculating the STAR which results from a particular choice of cold age threshold, i.e. how long since last access to consider a page to be cold. This analysis is then used to inform the choice of cold age threshold to be applied in the future (Duraisamy et al. (2023)). Polar also scales its cold threshold to control performance impact, though uses a much rougher heuristic by taking the average coldness across all objects (Nguyen and Nguyen (2024)).

Weiner et al note that STAR is only a proxy for the actual performance costs incurred by applications and instead introduce the Pressure-Stall Information (PSI) metric, which is currently integrated into the Linux kernel and accessible via `/proc/pressure` (Weiner (2018)). PSI considers the time spent by CPUs stalled on events that are directly caused

by shortage of application memory, such as I/O from accessing data recently evicted from the page cache and page reclaims triggered by the kernel. At low memory pressure, TMO proactively offloads memory at a certain ratio of the currently available memory amount. As PSI increases up to a threshold, TMO gradually backs off and reduces the aggressiveness of offloading until it stops altogether (Weiner et al. (2022)).

In addition to application performance, the system Colloid considers the impact of memory contention (Vuppalapati and Agarwal (2024)). While other works retain as much hot data in memory as possible, this policy assumes that the cost of accessing slow and fast tiers is static and that accesses to data in the fast tier will always be faster. However under high memory bandwidth load, latency of accesses to the fast tier can grow. Colloid additionally triggers migration for the purpose of controlling contention and optimises performance holistically (Vuppalapati and Agarwal (2024)).

Other works directly optimise the migration operation itself. Nimble observes based on profiling that the actual copy of pages across tiers is a significant cost. They focus on page copy routines used by NUMA migration, and perform a number of optimisations on the `move_pages` system call. One of these is to parallelise the data copy across multiple threads spawned by the kernel, including across several pages. Another is to eliminate unidirectional copy overheads such as allocating space in the second medium by allowing symmetric exchange of pages when they are promoted from slower tiers into fast ones, combining eviction and promotion into one step.

TeraHeap's `h2_move` hint provides manual control over migration, supplemented by automatic policies for controlling memory pressure. The base implementation's hints prioritise moving immutable data and avoids updates in H2, which is sensible for both performance and endurance reasons in SSD and NVM. Additionally, automatic movement of objects is triggered when H1 is close to capacity. This mechanism also includes a policy which limits the number of moved objects to avoid excessive device traffic (Kolokasis et al. (2024)). These policies are quite different from related work and do not consider the impact of read performance from the slow tier, which is of concern when choosing between compressed memory and SSD in ZTera.

3.2.3 Page Sizing in Tiering

Several works in kernel-level tiering also consider the effect of page sizes. Virtual memory systems incur the cost of address translation, which scales with the size of address space. Linux supports transparent huge pages (THPs) which combines 4KB pages into 2MB huge pages. This feature benefits address translation by allowing a single page table entry to capture a larger amount of data, improving translation cache performance and reducing the size of page tables. However, Lee et al. (2023) note that huge pages pose issues for tiering because subpages may not be equally hot, and propose a system called MEMTIS that smartly chooses when to split huge pages to fix skewed hotness. Other works that focus on this aspect include HotBox and Thermostat (Bergman et al. (2022); Agarwal and Wenisch (2017))

TeraHeap backs H2 data on huge pages to reduce the cost of address space scaling (Kolokasis et al. (2024)). Otherwise, this work is not highly relevant for us as huge pages are a kernel level detail and entirely JVM agnostic.

3.3 Hardware-Level Tiering

There is some work on directly integrating tiering into the hardware for NVM systems. Intel Optane persistent memory has a "memory mode" in which NVM can be used as memory with DRAM acting as a direct-mapped cache (Raybuck et al. (2021)). Additionally, 3D die stacking allows placing a large DRAM tier extremely close to the CPU that can act as a cache for regular DRAM (Gulur et al. (2014)), and Intel has developed a hardware tiering system for CXL called Flat Memory Mode (Zhong et al. (2024)). However, this work is entirely designed to work with byte-addressable mediums and is not relevant for us.

Chapter 4

Evaluation: Synthetic Benchmarking of ZRAM and SSD

We conduct an evaluation with synthetic I/O benchmarks to first gain some basic understanding of the offloading tiers we are using. This section has several goals:

- **Goal 1**: Establish baseline performance numbers, such as bandwidth and latency, for various I/O patterns and parameters.
- **Goal 2**: Investigate methods of scaling I/O performance, such as applying more processes or increasing async I/O depth, and how they impact each device.
- **Goal 3**: Investigate the CPU overhead of compression associated with ZRAM and whether it is a limiting concern.
- **Goal 4**: Estimate how ZRAM performs when storing Java data, including compressibility and compression/decompression latency.

4.1 Hardware Setup

We use three servers with varying CPUs, memories and SSDs for the purpose of generalising our observations to a variety of hardware architectures. These consist of:

- Intel Server (Table 4.1): Dual-Socket Intel Xeon processor, 56 cores. Ice Lake architecture (2019). PCIe 4.0 x8 Samsung SSD. DDR4 memory.
- Small AMD Server (Table 4.2): AMD EPYC processor, 64 cores. Zen 4 architecture (2022). Two PCIe 4.0 x4 Samsung SSDs. DDR5 memory.
- Large AMD Server (Table 4.1): Similar features to small server, except 256 cores and 4 SSDs.

Table 4.1: Device parameters for Intel server

CPU	
Processor	Dual-Socket Intel Xeon Gold 6330N
Number of cores/threads	56C/112T
Core frequency	2.2 GHz (3.4 GHz turbo)
L3 cache	84 MB
NUMA domains	2
Memory channels	16 (8 per socket)
Memory	
DIMM model name	HMAA4GR7CJR8N-XN
DIMM vendor	SK-Hynix
Capacity	512 GB
DIMMs	16 DIMMs (1 per memory channel)
DIMM technology	DDR4-3200
Rank organisation	2Rx8
Bus speed	3200 MT/s
Bus width	64 bits
Bus bandwidth	25.6 GB/s
Total peak bandwidth	409.6 GB/s
ECC	Multi-bit ECC
SSD	
Model name	MZPLJ3T2HBJR-00007 (PM1735)
Vendor	Samsung
Number	1
Capacity	3.2 TB
Bus speed	16 GT/s (PCIe 4.0)
Bus width	8 bits (x8)
Bus bandwidth	16 GB/s

As the tables show, the AMD servers are more recent and have stronger compute capabilities and better memories, but the Intel server's SSD has a wider bus. We also set up a software Redundant Array of Inexpensive Disks (RAID) with the four SSDs on the large server, using RAID0 striped allocation to maximise parallelism. The RAID provides another baseline to compare ZRAM against.

The Intel server is run with Ubuntu 22.04 LTS (Jammy), while the AMD servers were run with Ubuntu 24.04 LTS (Noble).

4.2 Exploration With Fio

Our first set of experiments focus on using the Flexible I/O (Fio) benchmarking tool, which provides detailed support for customising I/O workloads and reporting I/O metrics (Axboe (n.d.)). While Fio by itself is limited with respect to customising the data being written (as is to be discussed), it provides enough options for us to complete a basic profile of ZRAM and SSD.

Table 4.2: Device parameters for Small AMD Server

CPU	
Processor	AMD EPYC 9374F 32-Core Processor
Number of cores/threads	32C/64T
Core frequency	3.85 GHz (4.1 GHz boost)
L3 cache	256 MB
NUMA domains	4
Memory	
DIMM model name	M321R2GA3BB6-CQK
DIMM vendor	Samsung
Capacity	192 GB
DIMMs	12 DIMMs (1 per memory channel)
DIMM technology	DDR5-4800
Ranks	1Rx8
Bus speed	4800 MT/s
Bus width	64 bits
Bus bandwidth	38.4 GB/s
Total peak B/W	461 GB/s
ECC	Multi-bit ECC
SSD	
Model name	MZQL21T9HCJR-00B7C (PM9A3)
Vendor	Samsung
Number	2
Capacity	1.92 TB
Bus speed	16 GT/s (PCIe 4.0)
Bus width	4 bits (x4)
Bus bandwidth	8 GB/s

We explore the following parameters exposed by Fio:

1. I/O engine: we mainly focus on `sync` (syscall I/O) and `io_uring`.
   - Fio provides the options `aio` for LibAIO and `mmap` for memory mapped I/O, but `io_uring` is strictly better than LibAIO, and memory mapped I/O must use the page cache; it does not have a `O_DIRECT` option, making `sync` a superior choice for analysing direct device performance.
   - Fio does allow specifying `direct=1` for mmap as well and internally implements an approximation to direct I/O by using the `posix_madvise` syscall to hint to the OS to drop the pages from page cache (Axboe et al. (n.d.), `/fio/engines/mmap.c:125-128`). However, the OS is free to ignore this hint and in our experiments we found that it does not sufficiently mitigate page cache effects. Fixing this issue would require modifying the Fio source, which we opted not to do.
2. I/O type: Fio provides six options which combine whether the I/O is sequential or random, and whether it consists of reads or writes:

Table 4.3: Device parameters for Large AMD Server

CPU	
Processor	AMD EPYC 9754 128-Core Processor
Number of cores/threads	128C/256T
Core frequency	2.25 GHz (3.1 GHz boost)
L3 cache	256 MB
NUMA domains	4
Memory channels	12
Memory	
DIMM model name	M321R2GA3BB6-CQK
DIMM vendor	Samsung
Capacity	384 GB
DIMMs	12 DIMMs (1 per memory channel)
Ranks	1Rx8
Bus speed	4800 MT/s
Bus width	64 bits
Bus bandwidth	38.4 GB/s
Total peak B/W	461 GB/s
ECC	Multi-bit ECC
SSD	
Model name	MZQL21T9HCJR-00B7C (PM9A3)
Vendor	Samsung
Number	4
Capacity	1.92 TB
Bus speed	16 GT/s (PCIe 4.0)
Bus width	4 bits (x4)
Bus bandwidth	8 GB/s

- `read`, `write`: sequential reads and writes
- `randread`, `randwrite`: random reads and writes
- `rw`, `randrw`: mixed workloads with 50% reads and 50% writes. There is an additional option to control the mix of reads and writes.

We explore all of these options.

3. Block size, i.e. the size per I/O request: we use a minimum of 4 KB. Sub-page sizes are possible, but excluded from consideration as neither device is designed for them.
4. Number of processes issuing requests: we control this using the `numjobs` parameter to instruct Fio to run several instances of the same job in multiple processes.
5. I/O depth for async engines, i.e. the size of the submission/completion queue, dictating how many requests can be in flight at once per process.
6. Compressibility of data written by Fio. The `buffer_compress_percentage` pa-

rameter allows instructing Fio to fill a certain percentage of write buffers with random incompressible data, and the rest with compressible data. This method is a relatively rough approximation of varying compressibility of data. We observe the sensitivity of ZRAM to this parameter.

We also explore a selection of compression algorithms for ZRAM, specifically lzo, lz4 and zstd (the main algorithms described to have been considered in TMO (Weiner et al. (2022))). Per-algorithm options (e.g. the option `fast` for zstd) exist, but were not investigated in depth.

This section is organised into general methodology, followed by results from various sub-experiments designed to address the goals outlined earlier. Each of these forms a case study with a particular narrow focus. We then conclude by bringing together the observations across each set of results to form a general view.

4.2.1 Methodology

Three ZRAM devices are created by enabling the `zram` kernel module. The `zramctl` utility, which provides a neat interface to sysfs control files, is used to configure a memory limit and disksize for each device. Each device is also configured to use a different compression algorithm as described earlier, using default algorithm settings. The maximum number of compression streams (limiting how many threads can compress at once) is set to the number of cores on the respective machine. An ext4 filesystem is created on the device. Importantly, the filesystem is mounted with the `discard` option to instruct ZRAM to drop blocks as soon as files are deleted. This option must be passed explicitly as otherwise the ZRAM devices will not drop deleted blocks until an explicit TRIM command is provided. This default behaviour exists for devices where deletion can cause internal compaction or garbage collection that may cause lag, which is not a concern for ZRAM. Discard also avoids out-of-memory (OOM) errors. The SSD is also set up with an ext4 filesystem.

Basic fio config files are set up, and a script is created which used the environment variable expansion feature of Fio (Axboe (n.d.), section 1.12.1) to generate runs with varying parameters. Each run of `fio` is accompanied by various monitoring processes:

- Instances of the Linux tools `iostat` and `mpstat`, run with a modified version of the scripts in Kolokakis (2024). These processes report the underlying device statistics (as collected by virtual block layer) and system CPU utilisation statistics respectively. The tools are instructed to print a report every second. Read-only workloads in Fio have a setup phase where the file is "laid out", which we manually eliminate from consideration by removing entries where either number of reads to the device is close to zero (for `iostat`) or only one CPU is being utilised (for `mpstat`).
- A `zram_usage` script which polls various ZRAM sysfs statistics files every second, namely `io_stat`, `mm_stat` and `bd_stat` which are stored in the directory

Table 4.4: Default values for each parameter

Parameter	Default value
I/O engine	`sync`
Block size	64K
Number of processes	1
Compressibility	50%
Device	Small AMD Server

`/sys/block/zramX`. The most important is `mm_stat` which provides information regarding the amount of data stored and compressed, how many pages were deemed incompressible and how many were trivially compressed as "same pages". We decide to pull the statistics directly rather than use `zramctl`, which does not include important metrics even when using the `--output-all` flag.

The script also copies the contents of `/proc/diskstats` to file, once before Fio runs and once after, to provide further device statistics.

We also set a number of constant parameters for `fio`. The `direct=1` option is provided to eliminate page cache effects, along with the default `invalidate=1` option for dropping the page cache before runs. There are also effects from non-uniform memory access (NUMA), a necessity for memory scaling on large servers, which must be mitigated. If using fewer cores than are present on one NUMA node, the `numa_cpu_nodes` option is set to bind all cores to one node.

During experiments where a particular parameter is not the main focus, it is necessary to set a default. I/O type and compression algorithm were considered important enough parameters to try all options as variants in each experiment. The size of files used by Fio varies somewhat across experiments, and is scaled based on expected runtimes. The file size is set to keep experiments quick while ensuring that they run for enough time to overcome possible SSD internal caching effects and allow `iostat`/`mpstat` to provide multiple reports. Otherwise, defaults are generally chosen as either a middle ground between extreme options or to prioritise the simplest case (e.g. single process over multiple). Some experiments do include variants on each of the 3 devices, but if not explicitly mentioned data presented is from the Small AMD Server. Defaults are shown in Table 4.4.

There are various other minor methodological aspects. The scripts contain simple validation checks before runs (e.g. check that devices are mounted). Fio is set to output results in JSON format, balancing human readability for quick checking with machine readability for bulk processing. Post processing scripts combine reports from Fio and monitoring tools, both within and across runs/configurations, into CSVs for viewing and analysing. To avoid interference from shell processes, all runs are done in the background with `nohup` and the shell is exited, with logs being printed to a remote device for checking experiment status.

4.2.2 Effect of Data Compressibility and Compression Algorithm

We first vary the compressibility of data using the `buffer_compress_percentage` option mentioned earlier, which splits written data between a highly random/incompressible part and a compressible part. The additional options `buffer_compress_chunk` and `buffer_pattern` provide further control. The former allows specifying how interleaved compressible/incompressible parts should be within I/O blocks in terms of the size of "chunks", which we set to 256 B to ensure a good distribution within pages. The latter specifies what data to place for the compressible part, which is zeroes by default but can be set to a specific repeating string such as "ABCD". Since use of `buffer_pattern` may skew the compression ratio controlling `buffer_compress_percentage` is only an approximation to controlling compressibility, we use the default.

Our hypotheses are as follows:

1. More incompressible data should experience higher performance overhead, since the compressor has to do more work to find redundancies.
2. There should be a sensitivity to the type of compression algorithm used. Since zstd is a heavier algorithm that generally achieves higher compression ratios, and hence does more work to extract compression, it should perform worse than the other two algorithms.
3. We also run with SSD, and expect that the SSD performance will not vary with compressibility, though we admitted the possibility for internal hardware compression effects.

The results for sequential read and write bandwidth are shown in Figure 4.1. The data for latency was analysed but showed similar results. Hypothesis 3 is clearly validated as SSD performance remains constant. However, there are some interesting exceptions to the other two hypotheses.

Firstly, regarding hypothesis 1, it is especially apparent in the write bandwidth case, and for compressibilities ranging from 10–90%, that performance does improve as data becomes more compressible. For both read and write performance, 100% compressibility results in a significant increase in performance (around 5–10×) compared to 90%. When all pages are zero, it is likely that the amount of work required to compress becomes more or less trivial. However, the 0% compressible case in fact shows a significant *increase* in both read and write performance, especially in the former. Looking further into the ZRAM driver source code, we found that while ZRAM attempts to compress all pages, if the resulting compression ratio is sufficiently poor the page is marked as incompressible, and in fact the original data is stored instead of the compressed one (Torvalds et al. (n.d.), `/drivers/block/zram/zram_drv.c:1323-1438`). Reading incompressible pages is hence only a simple `memcpy` (Torvalds et al. (n.d.), `/drivers/block/zram/zram_drv.c:1270-1272`), explaining the high read bandwidth. The write operation still incurs the compression overhead, but skips the `zsmalloc` invocation as the page is simply kept in its original memory location.

Another point regarding hypothesis 1 is that the effect of compressibility is much more markedly seen in write performance compared to read performance. In fact, we observe a fundamental asymmetry in ZRAM read/write performance, especially for the stronger compression algorithm ZRAM. A similar asymmetry is visible for SSD (1.5 GB/s write vs 2 GB/s read), as is expected given the discussion from the background. However, it is much more apparent for ZRAM (3 – 4 GB/s VS 0.1 – 1 GB/s). It is in fact classically known that compression is more expensive than decompression, since compression is more complex and involves analysing redundancies and building data structures to capture them, while decompression reads off of pre-built structures. This also explains why reads/decompression is not as affected by compressibility, since the effect on how difficult it is to find redundancies is much greater than the time to read from pre-built tables.

Coming to hypothesis 2, the expected gap between compression algorithms is also seen, especially in write B/W. In fact, the gap is more stark than we expected; while lz4 and lzo perform similarly at approximately 1 GB/s write bandwidth at medium compressibility (50%), zstd is much worse and only allows 0.1 GB/s bandwidth. Interestingly though, while a gap does exist in read bandwidth between the algorithms as well, it is much less pronounced (3.5 GB/s VS 4 GB/s). This result suggests that even though zstd may build more complex data structures to capture redundancies, it still retains high read performance over them. In the next few experiments, we will sometimes show the zstd write performance compared to SSD separately to the other two algorithms compared to SSD, as they tend to lead to different conclusions regarding how the two technologies compare.

4.2.3 Scaling Async I/O Depth

Our next experiment concerns the effect of asynchronous I/O. Using the `io_uring` engine, we scale the I/O depth. We use one process to issue asynchronous I/O requests with varying I/O queue sizes. Increasing queue size should improve performance for both devices as the execution of different I/O requests can be overlapped.

We in fact observe in Figure 4.2 that while SSD performance scales with I/O depth, ZRAM does not benefit from asynchronous I/O whatsoever. SSD read and write bandwidth both end up exceeding the single-process bandwidth extracted by ZRAM. Observing `mpstat` data reveals that all CPUs except one remain idle when using ZRAM, and hence it is only using one process for compression despite being provided requests that may be fulfilled by more processes. Figure 4.3 displays the average device-side I/O queue lengths, which shows that the SSD queue becomes very large while the ZRAM queue only holds 1 or 2 requests at any given time. This indicates that ZRAM is only handling one request at a time.

These results can be attributed to the fact that since ZRAM is implemented in the block layer, compression is handled directly by the I/O-issuing process. As a result, asynchronous I/O does not work as ZRAM cannot use additional CPU resources to sat-

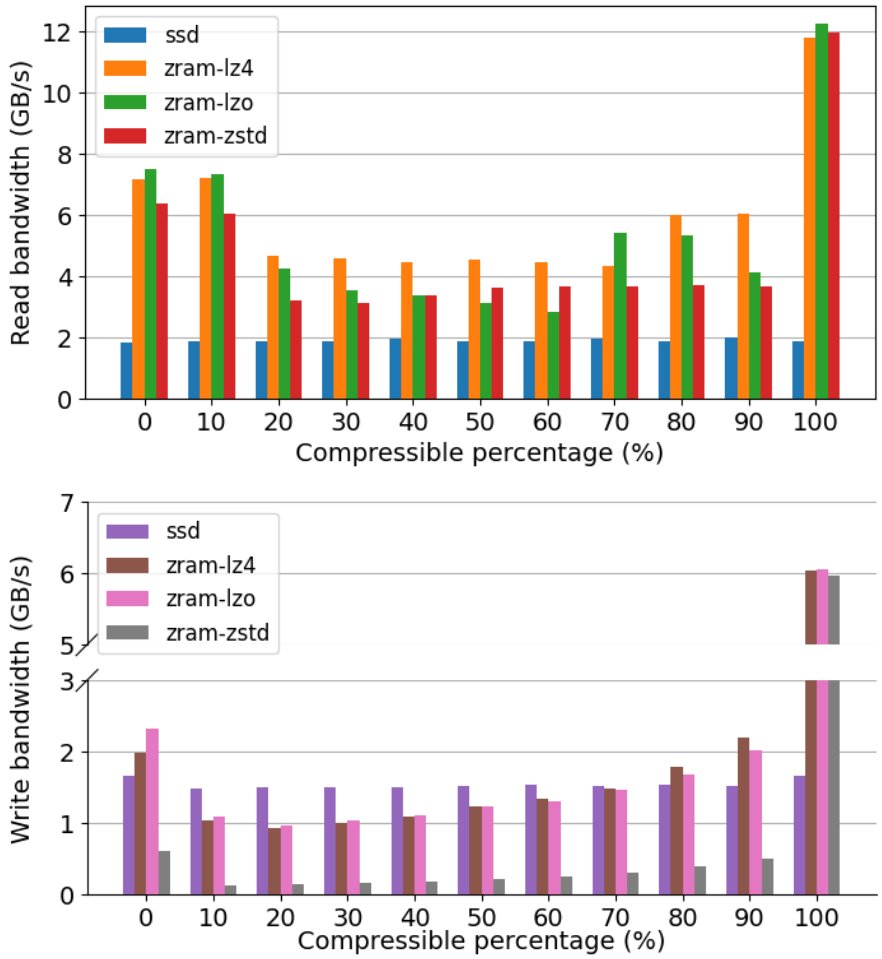

Figure 4.1: Sequential read (top) and write (bottom) bandwidth for varying compressibilities for each compression algorithm

isfy the requests. This is in contrast to Zswap, which can use its own external thread pool in the `kswapd` kernel daemon for compression. Conversely, the SSD can process asynchronous requests freely using direct-memory-access (DMA) semantics for I/O requests. SSD internal parallelism also allows its performance to scale with the volume of requests. Hence, asynchronous I/O allows SSDs to be a vastly superior choice to ZRAM when limited to using a single I/O-issuing thread.

4.2.4 Random/Sequential Gap

We proceed to examining whether there is a gap between random and sequential I/O in SSD and ZRAM. Our hypothesis was that there should not be one for both devices. SSDs already apply logical to physical mapping and hence internally break the locality

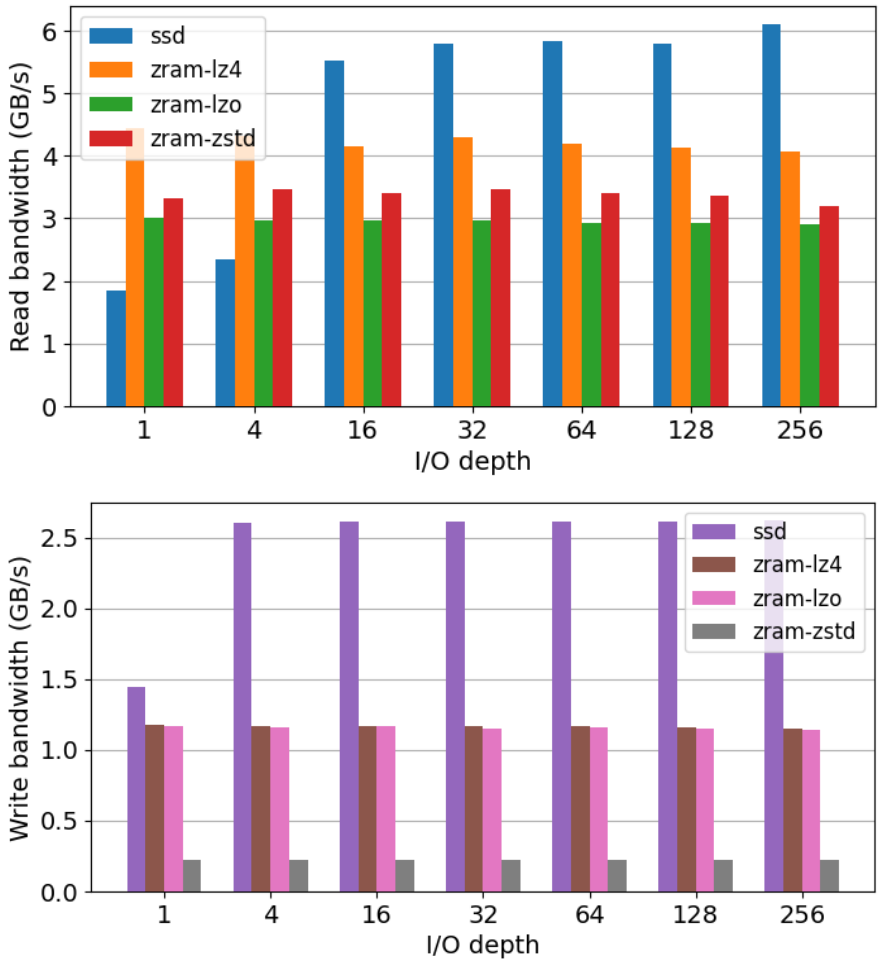

Figure 4.2: Sequential read (top) and write (bottom) bandwidth for varying I/O depths using `io_uring` asynchronous I/O

otherwise available from streamed accesses. RAM is usually treated as if it has no random/sequential gap either, though some minor effects do actually occur from the fact that RAM must precharge internal row buffers after each access, which can be skipped if an access hits an "open" row. These are not expected to be a major factor.

The results are shown in Figure 4.4. We observe that our hypothesis is mostly validated, though in most cases sequential does achieve slightly higher bandwidth (1.1–1.2×), and there is one case which exhibits a significant gap of 2.7×: SSD on reads. We observed also that the SSD on the Intel server does not exhibit this read gap, and the gap disappears on all devices when scaling the number of processes or using async I/O with large I/O depth.

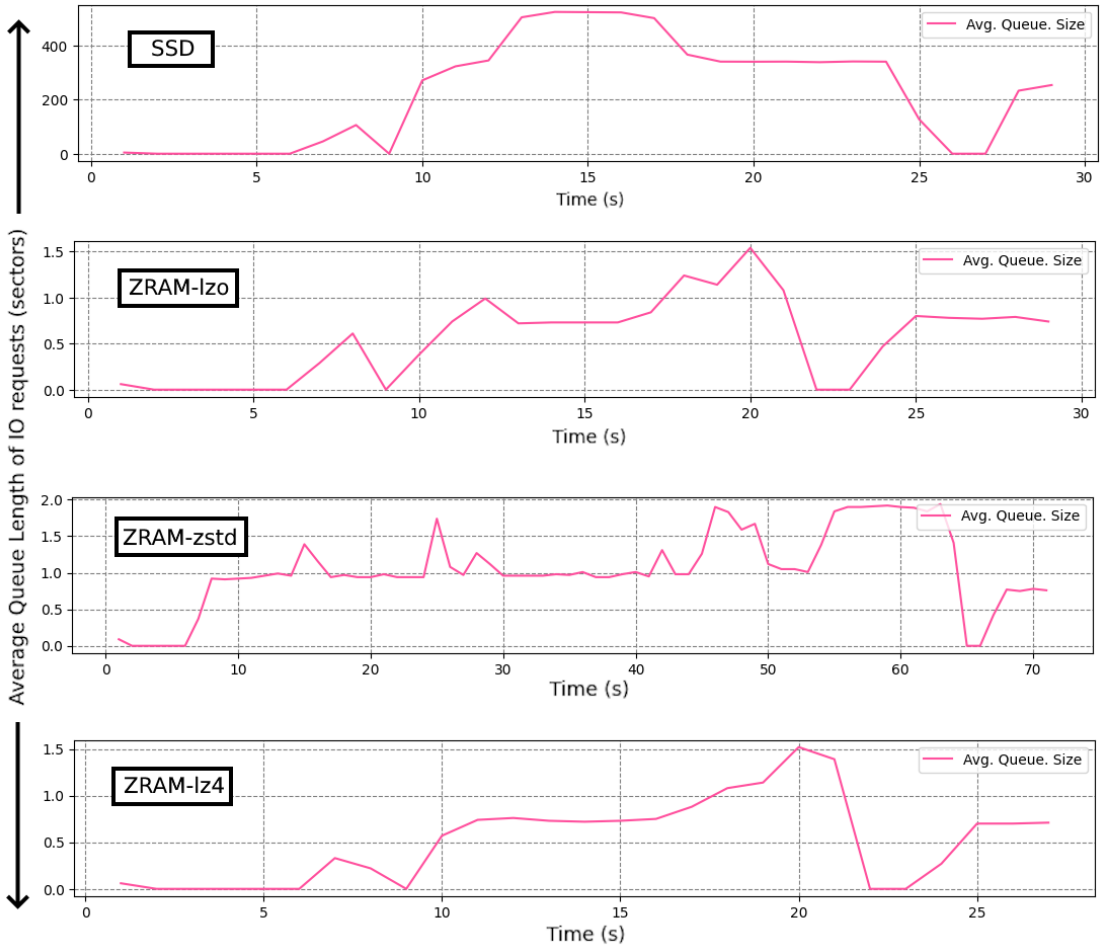

Figure 4.3: Graphs of device side queue lengths over time for each device, for sequential read benchmark and I/O depth 128. Data measured by `iostat`. Sequential writes display similar behaviour.

Since the gap is generally minor, except for the case of SSD reads in which case it disappears after scaling, subsequent data tends to exclude showing the random I/O case.

4.2.5 Scaling I/O Block Size

We now scale the I/O block size for a single process, which provides the device more work to do per I/O request and reduces the effect of per-request startup and shutdown operations. SSDs are expected to benefit significantly from larger I/Os, as they can divide up the I/O into SSD sub-block/block granularities and execute parts of it in parallel. On the other hand, ZRAM with zsmalloc is always restricted to compressing and decompressing within pages (see background), and hence larger I/Os are not expected

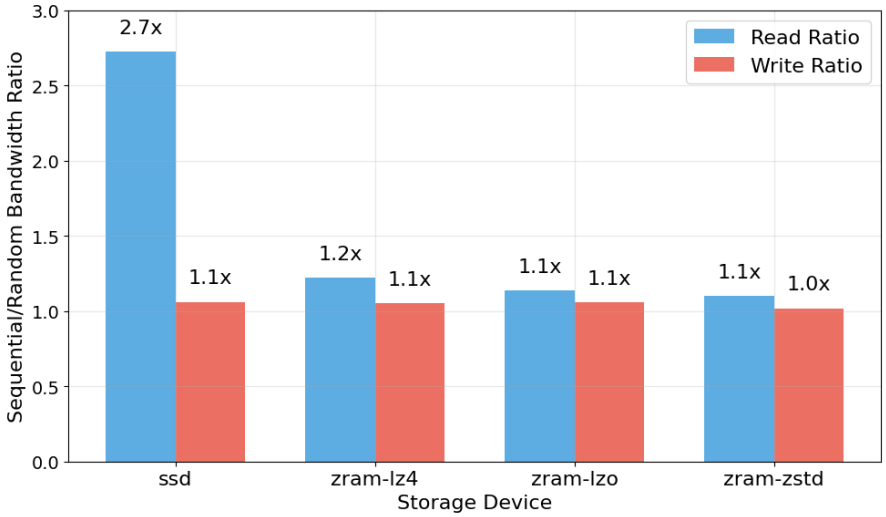

Figure 4.4: Sequential/random bandwidth ratio for each device, 1 process

to greatly benefit it.

The results are shown in Figure 4.5. SSD does indeed scale well for both read and write bandwidth, reaching 3 GB/s for reads and 2.5 GB/s for writes. Conversely, SSD performs quite poorly when using very small block sizes (4 KB), only having 200 MB/s read/write bandwidth. However, contrary to expectations, we observe a number of different trends with ZRAM. For instance, read bandwidth does in fact scale with block size. For small and medium block sizes, ZRAM read performance is greater, though it becomes only similar for large sizes as SSD's scaling is better. As for write performance, we observe a difference in behaviour between the compression algorithms. Zstd write performance does not scale at all with block size, which is in line with our expectations. However, lzo and lz4 do actually scale up to a block size of 64 KB, after which the scaling stalls and SSD write performance overtakes ZRAM.

Since ZRAM compresses page by page, and the `zsmalloc` allocator is also invoked once per page unit, there does not appear to be a kernel reason that ZRAM can sometimes scale with block size. Reducing block size does reduce system call overhead, but should affect all of the mediums uniformly. It is true however that RAM benefits from larger block sizes due to locality and internal parallelism. Our best updated hypothesis is hence that for small enough block sizes and weaker algorithms, compression/decompression is not a significant part of I/O execution compared to RAM access and hence it scales with block size.

These results indicate that we need a moderate block size (64K) to properly expose the compression/decompression cost of ZRAM.

4.2.6 Scaling Processes

We now move on to scaling the number of executors issuing I/O, which is one of the most important experiments as it is often required to extract the maximum bandwidth from I/O devices. For this experiment in particular we introduce a more significant focus on the other two servers (Intel and Large AMD), since they have different amounts of CPU resources and we can push scaling further. Additionally, we examine the scaling potential of the SSD RAID on Large AMD. On all servers, we try the case of applying every single core available.

We have three hypotheses:

1. Given that SSDs naturally exhibit significant internal parallelism, we expect it to scale well. It is unclear whether process scaling for SSDs will be more effective than either async I/O depth or block scaling, which shall be an interesting result.
2. As for ZRAM, it is evidently clear that applying more CPUs to compress will significantly scale performance. Since each Fio process is reading/writing separate data and hence handled by ZRAM separately, intuitively it appears to be an embarassingly parallel problem and we expect linear bandwidth scaling.
3. The SSD RAID enables even further parallelism than the SSD, and since it is striped for full parallelism (RAID0) across 4 SSDs we expect a maximum of 4× the bandwidth.

The results of scaling are shown in Figure 4.6 and Figure 4.7, showing bandwidth and average latency respectively.

In fact, hypothesis 2 has some quite interesting exceptions. While both ZRAM read and write bandwidth do scale and reach to around 14 GB/s and 3 GB/s respectively, they both hit constant caps that they cannot scale beyond. Most interestingly, the cap appears to depend on the compression algorithm. In the write bandwidth graphs, zstd is able to scale up to 16 processes while lz4 and lzo hit the cap at 4–8. As a result, even though single-process zstd write performance is quite poor, with enough processes all three algorithms deliver the same write bandwidth. We also observe in the average latency graphs that scaling processes does not affect the latency of writes for zstd up to 16 processes and lz4/lzo up to 4–8 processes, but then begins to increase dramatically beyond that.

We can hence conclude that compression/decompression bandwidth IS indeed scaling, but there are additional synchronisation bottlenecks inside ZRAM which are inhibiting it. These bottlenecks cause the latency of accesses to start increasing rapidly with number of processes at a certain point, negating the bandwidth benefits. There are in fact several layers of locks inside the `zsmalloc` allocator (Torvalds et al. (n.d.), `/mm/zsmalloc.c:18-24`), including across ZRAM pools and virtual `zspage` higher order pages that span several actual pages. These locks account for the results we observe.

As for SSD/SSD RAID scaling, we do in fact observe significant scaling for both. In

fact, the RAID scaling is perfectly linear with enough processes, e.g. read B/W at 256 processes is ≈ 20 GB/s for 4 SSDs compared to ≈ 5 GB/s for a single SSDs. SSD RAIDs eclipse *both* ZRAM read and ZRAM write bandwidth with enough processes, and hence in fact form a performance tier *above* compressed memory. We have optimised our RAID setup for performance exclusively (full striping/RAID0), and it would be an interesting datapoint to also consider setups which balance parallelism with redundancy/fault tolerance such as RAID5/6 and RAID10 (Arpaci-Dusseau and Arpaci-Dusseau (2023), Chapter 38). However, this is sufficient to demonstrate the ability to create physical SSD tiers that are more performant than software tiers.

Comparing only single SSDs and ZRAM, sufficient process scaling allows ZRAM write performance to be similar to SSD for all compression algorithms, and read performance is significantly better even with moderate or no scaling. The interesting point to note is the amount of scaling required for comparable write performance depends on the algorithm and can be anywhere from 4–8 to 16.

4.2.7 CPU Overhead of Compression

While the prior experiments either test sensitivity to particular I/O parameters or focused on different methods of scaling performance, we shift our focus in this sub-part to consider the CPU cost of ZRAM. This includes both the direct latency cost which ZRAM incurs though the use of CPU to compress data, as well as the utilisation of CPU it requires which takes away resources from other processes.

To check the compression latency overhead, we conduct an experiment using a `tmpfs` ramdisk, which creates a filesystem that stores data directly in system memory. Ramdisk allows accessing RAM while emulating the same I/O method as SSD/ZRAM, and factoring in associated costs such as system call overhead. Thus, we expect that the difference between ramdisk and ZRAM access should reflect only the overhead of ZRAM, i.e. compression and decompression. This experiment used one process and 1M block size to expose the compression overhead as much as possible, minimising synchronisation and syscall elements.

The latency results are shown in Figure 4.9. While RAM incurs a moderate read latency of 81.8μs and write latency of 110μs, ZRAM incurs read latency varying between 200–300 μs, and write latency varying between 680μs for lzo/lz4 to 4250μs for zstd. Thus, the decompression latency of ZRAM is around 100–200 μs, while the compression latency is ≈ 570μs for lzo/lz4 and ≈ 4000μs for zstd. We hence find that the compression/decompression component of ZRAM operation runtime is significant compared to the RAM access component

Moving on to CPU utilisation, Figure 4.8 displays a breakdown of the CPU usage for I/O to each device, as measured by `mpstat`, in one particular experiment. We use 64 processes here, but the results are similar for less processes if one adjusts for the number of processes left always idle. There are four components reported: user time, system time, I/O wait time and idle time. We observe that SSD runtime is mostly I/O

wait, leaving the relevant CPUs free to be used for other processes. Meanwhile, ZRAM runtime is 99% system time, and applying compression requires the active use of all of the I/O issuing threads. There is almost no idle time here to fit in other processes.

Thus, while processes used for SSD I/O can be scheduled out to overlap I/O wait with other thread executions, the same cannot be done for ZRAM I/O, which causes a significant CPU bottleneck that may affect application performance in resource-limited scenarios.

4.2.8 Effect of Read/Writes Mix

We next observe the effect of mixed read/write workloads on the two devices. SSD is generally known to be sensitive to even a small proportion of writes, as both reads and writes are affected by write operations such as erasing blocks. ZRAM does not have these issues, and we expected it to be less sensitive to writes as a result.

The total bandwidth for various read/write mixes is shown in Figures 4.10 and 4.11. Fio in fact reports read and write bandwidth separately, but the proportion between read and write bandwidth for each individual run is exactly given by to the proportion of read and write I/Os, and hence only total bandwidth (reads + writes) is interesting. We found differing results when considering 1 process VS 64 processes, and hence present both.

For 1 process, we in fact observe the exact opposite of what we expected: SSD is not impacted much by writes even up to a fully write workload, while ZRAM is highly affected. The zstd algorithm's total performance drops by over half after introducing only 10% writes. For 64 processes, SSD and ZRAM bandwidth are both affects by adding writes, even for small proportions. While this result is expected for SSD, we again did not expect ZRAM to also be affected. The lack of performance difference for 1 process in SSD may be explained by the fact that the small amount of I/O being done allows operations to be sufficiently distributed around the device that write and read operations do not conflict with each other. Meanwhile, the fact that ZRAM is so heavily affected by write operations can be explained by the fact that there is already a large read/write asymmetry for ZRAM, and synchronisation effects caused by writes + adding new pages to the `zsmalloc` pools causes the write performance to also affect read performance disproportionately.

Hence, even a small amount of writes can significantly impact performance for both ZRAM and SSD (when using enough processes to reach device saturation for the latter).

4.2.9 Performance Baselines

We now apply the knowledge from our scaling experiments to find the maximum performance possible with SSD and ZRAM. We separately consider singlethreaded and multithreaded performance, and maximise scaling of asynchronous I/O depth, processes and block sizes to find the best performance possible. The run configuration involved a

scan over a selection of block sizes (4K, 64K, 1M) and both sync/async with I/O depth 128. However, we find that in the multithreaded case, scaling block sizes and async I/O depth did not affect the performance of SSDs nor ZRAM, which is itself an interesting result. Hence, the figures reported for multithreaded are irrespective of block size and I/O depth. As for singlethreaded, we use the maximum block size of 1M and report both synchronous and asynchronous numbers for SSD; ZRAM on the other hand is not affected by async I/O and hence we report just one number. Ramdisk is also included as an additional datapoint.

The results for each server are shown in Figures 4.5, 4.6 and 4.7. We observe that in singlethreaded performance, using large enough block sizes allows SSD read B/W to be comparable to ZRAM and write B/W to completely eclipse ZRAM. Additionally, asynchronous I/O scaling allows SSD to handily defeat ZRAM in maximum singlethreaded performance. Given the explicit reliance of ZRAM on CPU resources, this is not an overly unexpected result. Conversely, after applying multithreading, ZRAM read performance is significantly better than a single SSD (×4 on the Intel server and ×2 on the small AMD server), and write performance is comparable. However, SSD RAID's enormous capability for delivering bandwidth allows it to become the best option in almost every case.

Table 4.5: Maximum performance numbers for devices on Intel server. Singlethreaded uses 1M block size; multithreaded is agnostic to block size and I/O depth. As a sidenote, Ramdisk figures for this experiment use `direct=0` due to Fio being unable to run with `direct=1` on `tmpfs` on this server.

Intel	**Singlethreaded read (GB/s)**		**Singlethreaded write (GB/s)**		**Multithreaded read (GB/s)**		**Multithreaded write (GB/s)**	
	Sync	**Async**	**Sync**	**Async**	**Sync**	**Async**	**Sync**	**Async**
SSD	1.9	8.4	2.9	3.7	8.2		3.2	
ZRAM lz4	3.4		0.84		36.6		3.7	
ZRAM lzo	1.5		0.78		37.7		3.2	
ZRAM zstd	2.6		0.11		36.1		2.9	
Ramdisk	5.0		4.2		56.7		36.6	

4.2.10 Conclusions

We have thus demonstrated that despite literature predominantly using ZRAM as a middle memory tier in between RAM and SSD, there are cases where SSD can in fact deliver higher bandwidth than ZRAM. For singlethreaded performance, ZRAM beats SSD if I/O depth is small (and/or I/O is synchronous) or I/O is done in small block sizes. However, scaling either of these parameters allows SSD to clearly come out on top. Moving to multithreaded performance provides ZRAM enough resources to overcome SSD in read performance, although write performance is similar and in fact even

Table 4.6: Maximum performance numbers for devices on Small AMD server. Singlethreaded uses 1M block size; multithreaded is agnostic to block size and I/O depth.

Small AMD	**Singlethreaded read (GB/s)**		**Singlethreaded write (GB/s)**		**Multithreaded read (GB/s)**		**Multithreaded write (GB/s)**	
	Sync	**Async**	**Sync**	**Async**	**Sync**	**Async**	**Sync**	**Async**
SSD	3.3	6.6	2.6	2.6	6.0		2.6	
ZRAM lz4	4.3		1.2		14.3		3.1	
ZRAM lzo	3.0		1.2		13.7		3.1	
ZRAM zstd	3.5		0.2		14.3		3.1	
Ramdisk	11.3		6.1		103.8		36.7	

Table 4.7: Maximum performance numbers for devices on Large AMD server. Singlethreaded uses 1M block size; multithreaded is agnostic to block size and I/O depth. Asynchronous I/O figures for SSD Raid are missing due to the server becoming unavailable from experiencing hardware errors.

Large AMD	**Singlethreaded read (GB/s)**		**Singlethreaded write (GB/s)**		**Multithreaded read (GB/s)**		**Multithreaded write (GB/s)**	
	Sync	**Async**	**Sync**	**Async**	**Sync**	**Async**	**Sync**	**Async**
SSD Raid	4.45	—	3.6	—	20.9		10.2	
ZRAM lz4	3.6		1.0		11.8		2.5	
ZRAM lzo	2.4		1.0		11.5		2.5	
ZRAM zstd	3.0		0.16		12.4		2.5	
Ramdisk	8.3		4.5		83.3		40.6	

a small write component to the workload can greatly diminish ZRAM's read performance benefits. Additionally, using RAID striping allows creating a physical tier that is significantly more capable than ZRAM in terms of achievable bandwidth.

Factoring in the cost of additional SSDs in the RAID for providing redundancy, it is an interesting question as to whether an SSD RAID is more expensive in terms of hardware cost than compressed memory. RAM cost is high, but compression ratios decrease the cost per byte of data. On the other hand, while SSD cost per byte is low, RAID redundancy increases the cost per byte by requiring each logical byte to take up multiple physical bytes. Balancing these cost factors against the associated mediums' performance can be a direction for research.

4.3 Testing with Java heap data

One major issue with the results thus far is that Fio's model of partially compressible data is very rough. The fact that part of the data is extremely incompressible while part of it is trivially compressible does not reflect most actual data, and especially not data which Java applications may place on the heap.

This section examines the efficacy of ZRAM on real Java data from the DaCapo Chopin microbenchmarks (Blackburn et al. (2025)), analysing the compression ratios and performance achieved by ZRAM in this setting.

4.3.1 Methodology

The general process we use is to capture snapshots of the Java heap's contents at various points in the benchmarks' executions, followed by running Fio to read/write data from these files in place of its own synthetic data. There are three metrics we are interested in: compression ratios, read performance and write performance.

There were two methods we considered for getting snapshots:

1. Create HPROF binary dumps of the heap using the `jcmd` utility's `GC.heap_dump` option, passing to it the process ID (PID) of running Java processes
2. Use the extra JVM option `-XX:AllocateHeapAt` to run the benchmarks with heap allocated over a file on SSD, and create backups of this file. Requires using `SIGSTOP`/`SIGCONT` signals to pause the benchmark execution to avoid race conditions.

The snapshots from these two methods will be referred to as HPROF dumps and direct dumps respectively.

The direct option proves surprisingly difficult, since the JVM unlinks the heap file on SSD as soon as it creates it. The data from the file is retained by the kernel because the JVM keeps a memory mapping of it, but the file becomes inaccessible from the filesystem, and is not persisted beyond the end of the benchmark execution. While

normally a temporary version of the file is still possible to access from `/proc/<pid>/fd`, we could not find the file there, which may be attributed to the fact that JVM uses mmap rather than refer to the file descriptor directly. The only possible option we found was to find the name of the heap file from `sudo lsof`, find the virtual address range that is allocated over the heap file from `/proc/<pid>/dumps` and directly read out the memory contents of the process from `/proc/<pid>/mem`. Since in practice there are actually several discontiguous memory mappings, implementing this method is rather complicated.

Hence, we prefer the HPROF option for generating dumps. However, both methods have flaws with respect to accurately capturing Java data:

1. The HPROF format documentation (Oracle Corporation (1993)) notes that the raw data is modified in several ways, including replacing all raw pointers with integer IDs. Our direct examination of the HPROF dumps using `hexdump -C` revealed that they contain a large amount of ASCII text, including the full names of various classes and interfaces. These names were missing from corresponding direct dumps. We hypothesise that the HPROF format describes the class membership of objects using direct ASCII names. Since the raw Java object layout condenses this information into the object header in the form of a reference to the class object (Oracle Corporation (n.d. (2)), the HPROF format here is significantly more verbose. Changes like these may make HPROF dumps artificially more compressible compared to the raw data.

2. While HPROF dumps restrict to including only live data on the heap, direct dumps necessarily capture the entire heap. The snapshot hence includes empty spaces that were unused during execution, and dead objects yet to be reclaimed. While the empty spaces can be excluded from consideration by applying zero compression to eliminate them, there remains potentially large amounts of young (dead) data polluting the heap. Since we would like to reserve compressed memory for long lived data, direct dumps may not reflect the data we are interested in.

In all of the following, the DaCapo Chopin benchmarks are run in the large size to maximise the amount of data considered. It is apparent that some benchmarks are missing from the data; in particular, `cassandra` did not run due to deprecation issues with JDK 17, and `fop`/`zxing` did not have a large size setting. We decide to accept the exclusion of these benchmarks and contend that our sample is nonetheless mostly representative.

For measuring compression ratios, we use two methods based on each method of dumping and compared the results. For HPROF dumps, we copy each dump individually to each ZRAM device and measured the data sizes and compressed sizes before and after. For direct dumps, while it is impractical to get persistent copies of these dumps, allocating the heap directly onto ZRAM during live benchmark execution allows measuring compression ratio regardless. This method has the added benefit that unused pages in the heap do not matter, because `mmap` only writes dirty pages to file. Hence,

even though the JVM allocates a large and initially empty file on the ZRAM device, the ZRAM device statistics only reflect the data actually written by the JVM. For both methods, we use the `zram_usage` monitoring script from the previous section to extract ZRAM statistics. We note that it is important to get statistics directly from `sysfs`, as the `zramctl` utility does not have support to change the granularity of units. We also note that ZRAM reports some data on the device even when the filesystem is empty due to storing filesystem metadata, which must be subtracted from the final data size.

For measuring read and write performance on the dumps, we use the filepath option for the `buffer_pattern` parameter in Fio to instruct it to fill I/O buffers with data from dump files. Fio will then use data from the file for each I/O. There is the associated limitation that Fio does not support changing the data used across different I/O requests, and uses the first `block_size` bytes of the file for every I/O. If the block size is set to be large, Fio uses more data from the file, but provides fewer I/Os to average results over. We hence choose a middle ground of `100M` as the block size and instruct Fio to perform 32 GB of I/O. Some heap dumps are smaller than 100M, in which case the data is duplicated to fill the 100M I/O buffer. Duplication appears to pose a concern due to its impact on compressibility which may in turn impact read/write performance.However, ZRAM only compresses within pages and cannot identify duplication across pages, and we find experimentally that compression ratios are not improved by duplication. Hence, the methodology remains sound.

For read experiments, there is an alternative method using the Fio `filename` parameter that allows specifying existing files to read. To allow this method to work for multiple processes simultaneously performing I/O, one must manually split the file into multiple files and specify each to a separate Fio job in the configuration. The setup for this method exists in our codebase, and it may be preferred since it allows using all of the data in each dump rather than restricting to a beginning chunk. However, write experiments cannot use this methodology because Fio write workloads will overwrite the file provided with the `filename` option rather than use data from the file. On the other hand, read experiments do support the `buffer_pattern` methodology as Fio lays out the file beforehand, which uses data that respects the `buffer_pattern` option. For the sake of consistency between experiments, we opt to use the former methodology for both read and write experiments.

We cover some minor experimental details. In both compression ratio experiments, we observe errors from data in previous experiments lingering for some time despite ZRAM being mounted with discard, which should delete data once the associated file is deleted. Hence, ZRAM device usage is reported incorrectly for the first 10 seconds of execution. This is handled by discarding the first 10 seconds of data for direct heap experiments, and waiting 10 seconds after deleting previous dumps before in HPROF experiments. `sync` is called on each dump after copy in HPROF experiments. The last dump for each benchmark is also discarded for HPROF experiments, as it tends to be smaller than previous ones and demonstrate aberrant behaviour.

4.3.2 Compression Ratios

Compression ratio results are shown in Figures 4.12 and 4.13. Average compression ratios across benchmarks are shown in Table 4.8.

Comparing compression algorithms, in almost all cases zstd proves to extract more compression than the other algorithms. In fact, lz4 and lzo perform similarly in ratio, while zstd achieves 1.5× better compression ratios on average. The benchmarks jme, sunflow and pmd in Figure 4.12 appear to contradict this claim, but these benchmarks wrote very little data to the ZRAM device. In the case of pmd, some of the ZRAM devices register 0 data written at all. This may be due to page cache effects inhibiting the data from reaching the device or similar buffer effects. Hence, these benchmarks can be excluded from consideration. For all other benchmarks across both dump methods, zstd consistently shows higher compression ratios. We expect as such, though previous experiments displaying that zstd also incurs the highest latency and CPU cost indicate that there is a tradeoff present here.

Focusing exclusively on Figure 4.13 for the moment, we observe that compressibility varies across different benchmarks. Kafka, Sunflow, PMD, Xalan and Biojava demonstrate compression ratios above 5× using zstd, while luindex and graphchi do not exceed 2.5×. These differences can mostly be explained by considering the content of the benchmarks (Blackburn (2024)). Kafka involves the use of a messaging framework, Xalan works with XML documents, Sunflow renders images, PMD analyses source code and Biojava analyses protein sequences. All of these tasks include data that is known to demonstrate compressibility, e.g. XML is highly repetitive in structure. As for incompressible benchmarks, Graphchi applies ALS matrix factorisation which is used to decompose matrices for recommendation systems, which may involve dense numeric data. Luindex is surprising to some extent as it involves indexing text documents for the search engine Lucene, which should be compressible. Our working hypothesis however is that the raw document data is dominated by index segment data held in memory, which involves raw numeric data in the form of posting lists and may be less compressible.

Comparing the two dump methods, we had anticipated that HPROF dumps may inflate compression ratios by altering the underlying data. However, Table 4.8 indicates that direct dumps in fact turn out to be MORE compressible. Additionally, while many benchmarks have similar compressibilities in both dump methods, there are exceptions. For instance, the H2O benchmark which performs machine learning demonstrates moderate compressibility for HPROF dumps, but significant compressibility for direct dumps. This inconsistency can be understood to be caused by the fact that HPROF dumps only include live data, while direct dumps include all data on the heap. Long-lived dead objects may inflate compression ratios in some cases for direct benchmarks. There may additionally be gaps in the heap layout in the direct case that do not appear in the dumps case.

Overall, both methods indicate that there are significant opportunities for compressing Java object data, and that compressed memory is effective in a Java setting.

Table 4.8: Average compression ratio for each compression algorithm across dumps and benchmarks

Dump method	lz4	lzo	zstd
Direct	3.44	3.49	4.82
HPROF	2.84	3.07	4.51

4.3.3 Read and Write Performance

Read and write performance on representative dumps for each benchmark are shown in Figures 4.14 and 4.15. SSD performance does not change depending on underlying data and is hence reported separately as a single reference bar. The SSD experiment uses identical methodology to the ZRAM ones, i.e. 100M block size with prefilled data, to provide a fair comparison.

The overall comparison to SSD is similar to previous experiments. SSD generally outperforms ZRAM in write bandwidth for single process experiments. We also observe SSD read bandwidth being greater in this experiment, which is different to prior data. However, this is because the methodology uses extremely large I/O block sizes and SSD is able to retain multiprocess level performance even when using single processes. Conversely, in multiprocess experiments ZRAM generally performs significantly better than SSD.

However, the individual benchmark performance has surprising variations. Kafka, sunflow and lusearch, three of the more compressible benchmarks from the previous experiment, demonstrate higher read and write bandwidth compared to the other benchmarks. Interestingly, graphchi also displays slightly better read and write performance when using single processes despite involving highly incompressible data. We are aware from the compressibility experiment that compression/decompression overhead is affected by compressibility levels. More specifically, overhead increases with difficulty compressing up to full incompressibility, at which point data is stored in uncompressed form and decompression is skipped. This behaviour appears to generally explain the observed trends. Figure 4.16 shows the ratio of incompressible data in each benchmark and indicates that graphchi has a significant portion, even when using the stronger zstd algorithm.

However, there is a conflict in our theories if we attempt to explain the multiprocessing data via compression overhead alone. Our earlier data shows that the three compression algorithms perform similarly when using enough processes, which is also observed in this experiment's data. Our prior reasoning indicates that this result is due to compression no longer being the primary performance bottleneck when applying enough processes, instead being capped by synchronisation bottlenecks in `zsmalloc`. If this is the case, it should not matter how compressible data is since improved compression should not affect total performance, which remains capped by synchronisation and `zsmalloc` overhead. Yet, we observe that the more compressible benchmarks kafka, xalan, lusearch and sunflow still retain high read and write bandwidth even when applying many processes.

This phenomenon is explained by considering another mechanism in ZRAM we have yet to discuss, which is the ability to detect "same pages". For pages which are identically zero or otherwise contain the same byte repeated, there is a special mechanism where ZRAM avoids storing *any* data for the page in its compressed memory pool. For clarity we call these pages trivially compressible. Using sysfs-exported metrics we can identify the number of trivially compressible pages in each dump, which is reported as a percentage of total data in Figure 4.17. We observe in fact that the well performing benchmarks have a significant proportion of trivially compressible data. As this data is not stored by `zsmalloc`, it does not experience the same synchronisation bottlenecks and thus performance can scale beyond what we previously observed was possible.

This analysis also leads to the interesting result that some Java heaps, even when limited to only live data, contain a significant amount of trivially compressible space. Additionally, ZRAM can handle trivial compression extremely efficiently.

4.3.4 Conclusions

We observe that Java heap data demonstrates good potential for compression, which includes trivial compression. Our read and write performance results do not significantly change compared to before as we again find that SSD's single process performance scaling is superior, but that using multiple processes allows ZRAM to perform better than SSD. However, we have now verified that this result holds true for more realistic data as well.

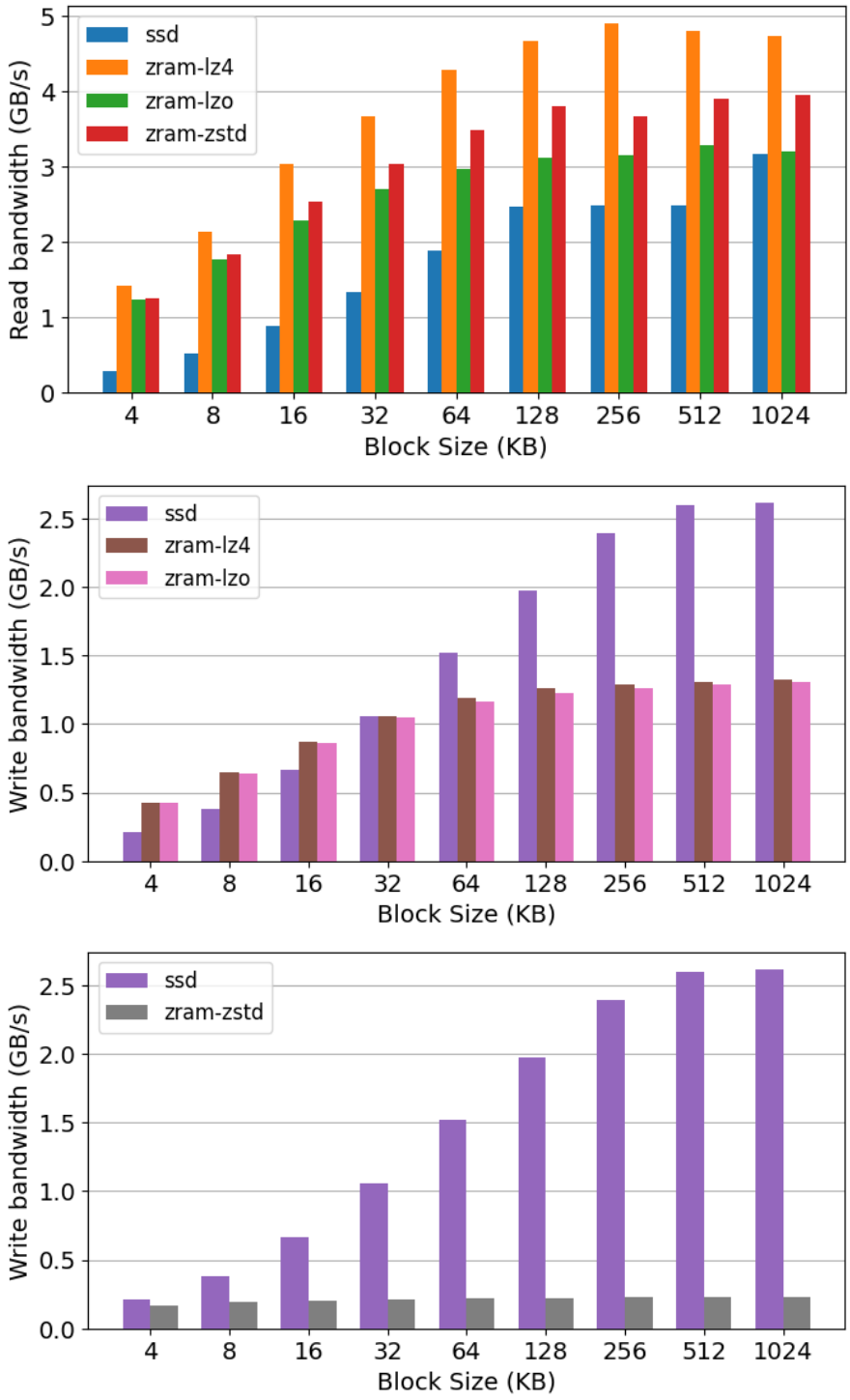

Figure 4.5: Read and write bandwidth for various I/O block sizes. Top graph is reads, middle graph is writes comparing lzo/lz4 with SSD, bottom graph is writes comparing zstd with SSD.

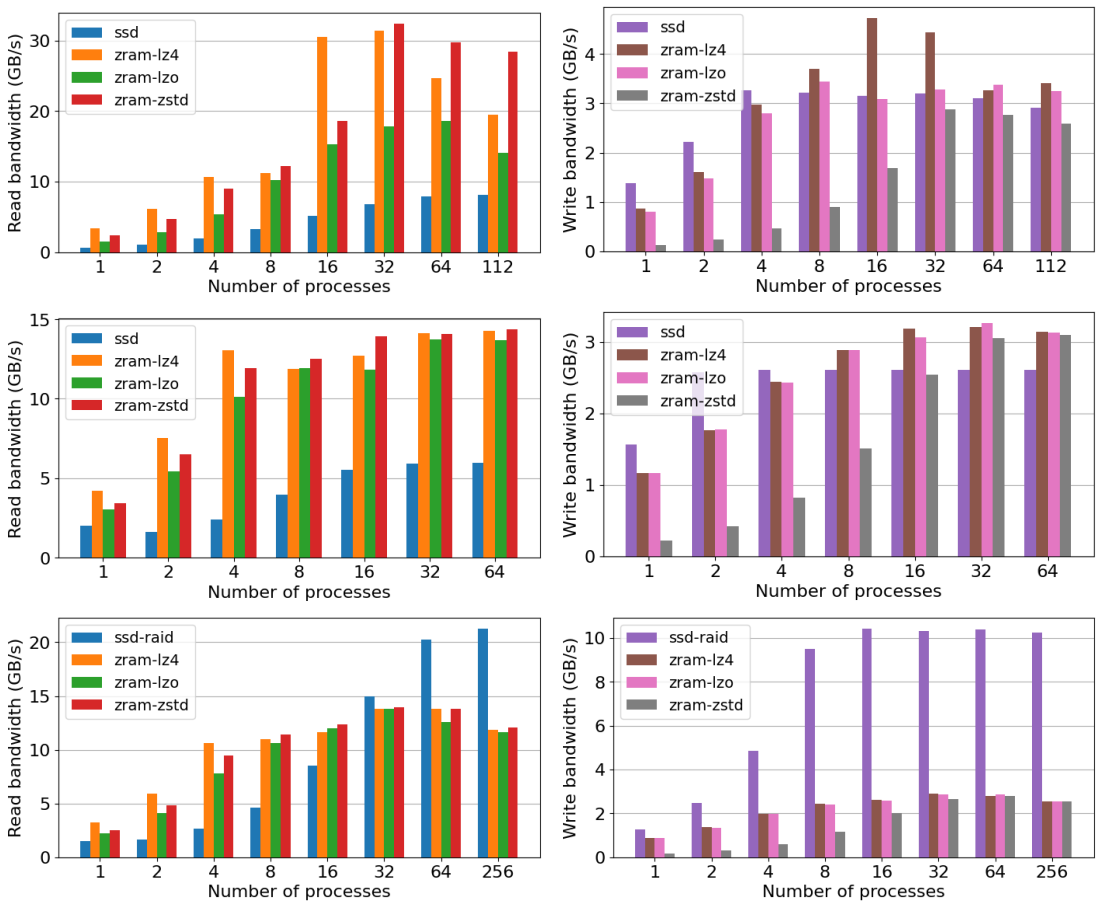

Figure 4.6: Bandwidth results from process scaling. Left column: sequential reads, right column: sequential writes. Each row is a different one of the three servers; top is Intel, middle is Small AMD and bottom is Large AMD + SSD RAID

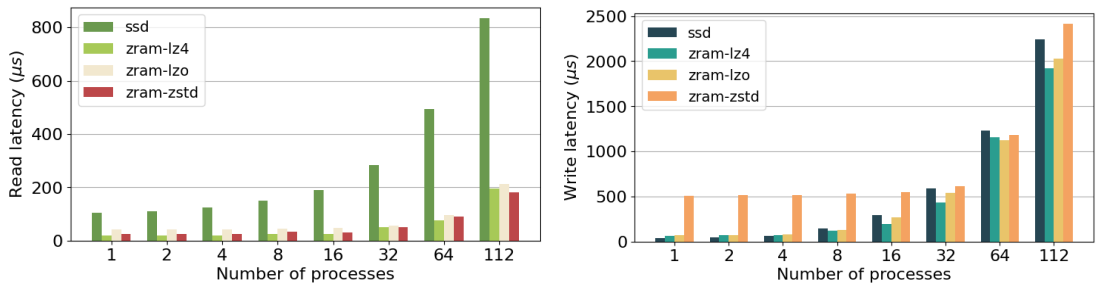

Figure 4.7: Subset of average latency results from process scaling, specifically those from the Intel server. Left is reads and right is writes.

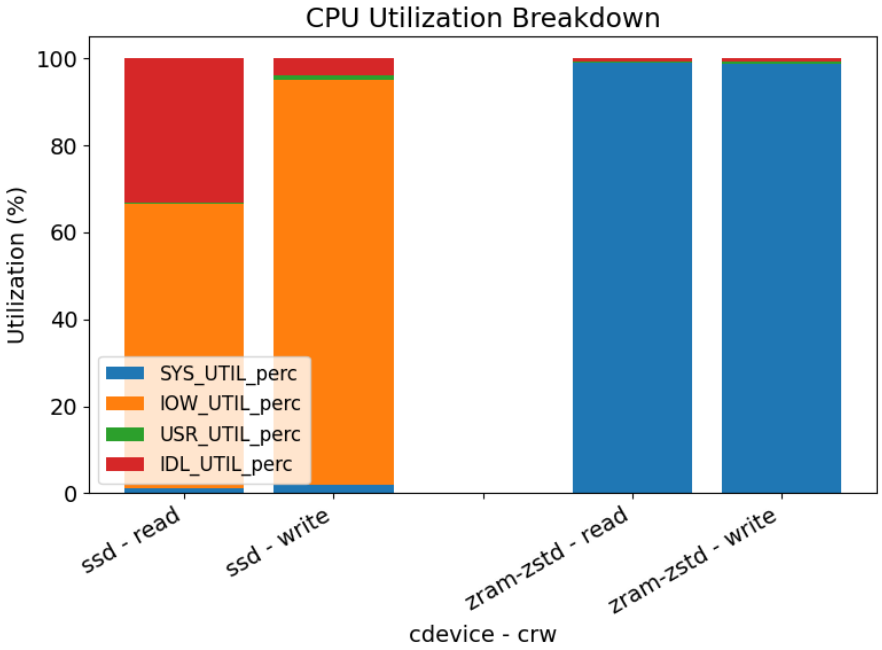

Figure 4.8: CPU utilisation for 64 processes on RAM, SSD and ZRAM respectively. The labels' meanings are as follows: `SYS` is system time, `IOW` is I/O wait time, `USR` is user time and `IDL` is idle time.

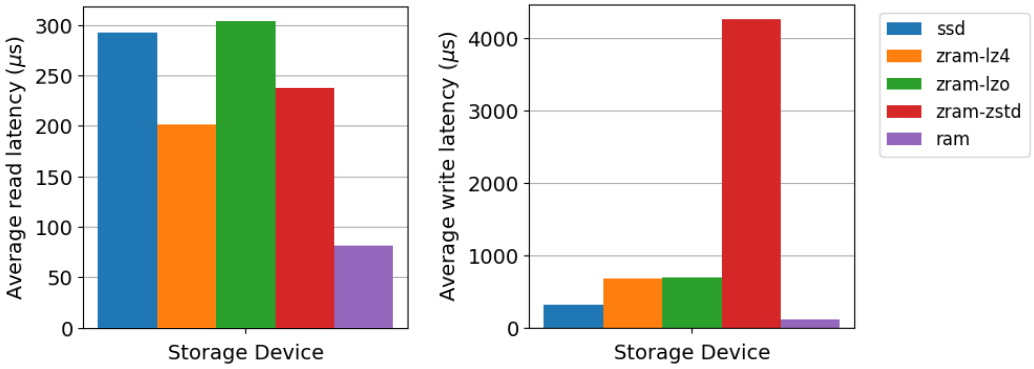

Figure 4.9: Latency for each device (1 process, 1M block size). Left is reads, right is writes. The difference between ZRAM and RAM corresponds to compression/decompression overhead.

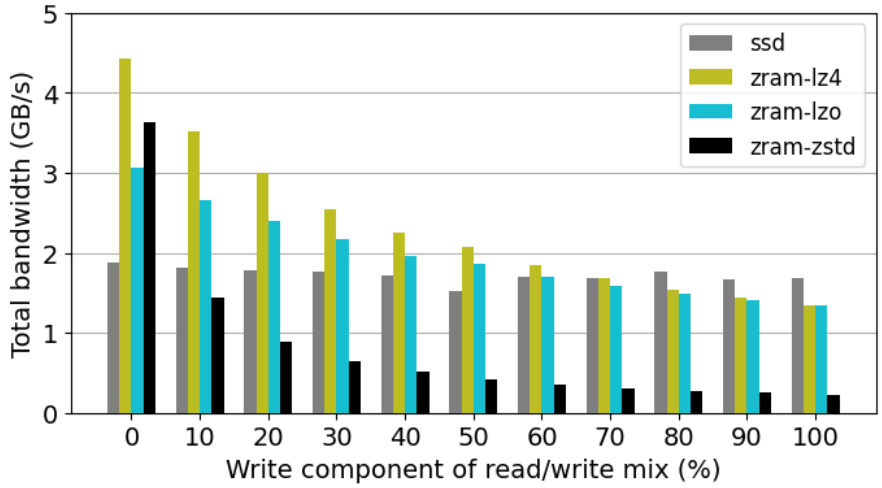

Figure 4.10: Total bandwidth for various read-write mixes, one process. Leftmost side is reads-only, rightmost side is writes-only.

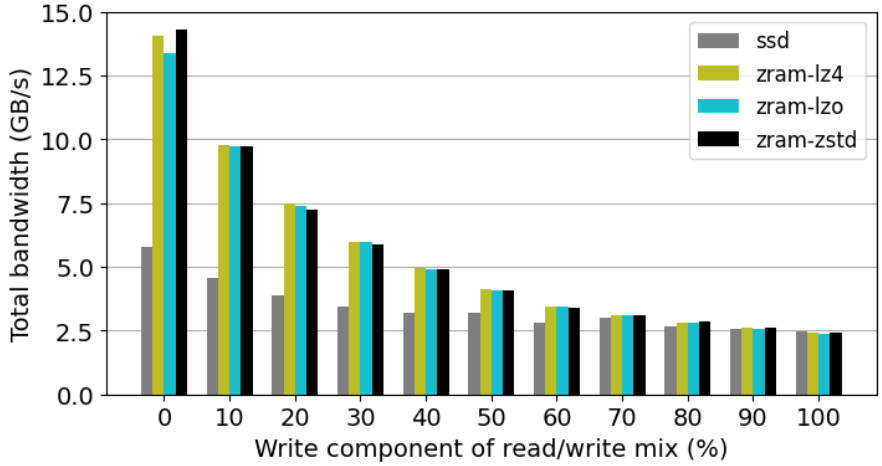

Figure 4.11: Total bandwidth for various read-write mixes, 64 processes. Leftmost side is reads-only, rightmost side is writes-only.

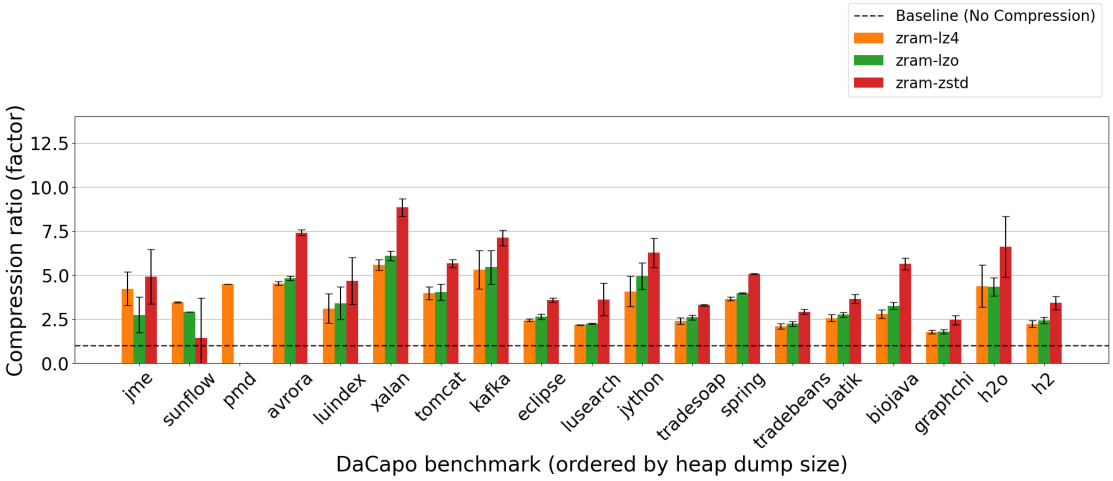

Figure 4.12: Compression ratios for each DaCapo benchmark, measured on the direct heap using `-XX:AllocateHeapAt` option. Error bars display variation across 1 second sampling intervals.

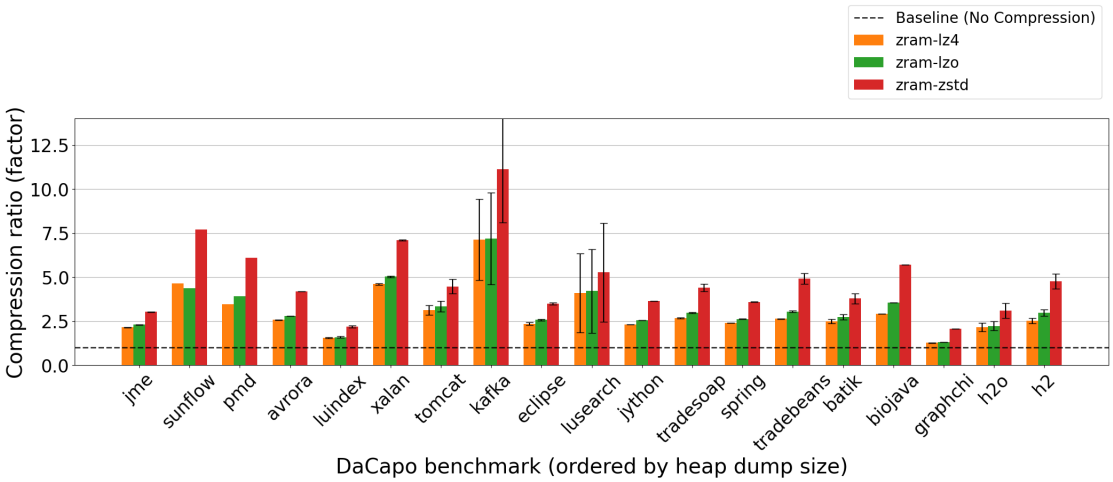

Figure 4.13: Compression ratios with each DaCapo benchmark, measured on the HPROF heap dumps. HPROF dump sizes in fact order differently compared to direct dump sizes, but both this graph and Figure 4.12 use the same ordering for ease of reading. Error bars display variation across each dump (10 second intervals).

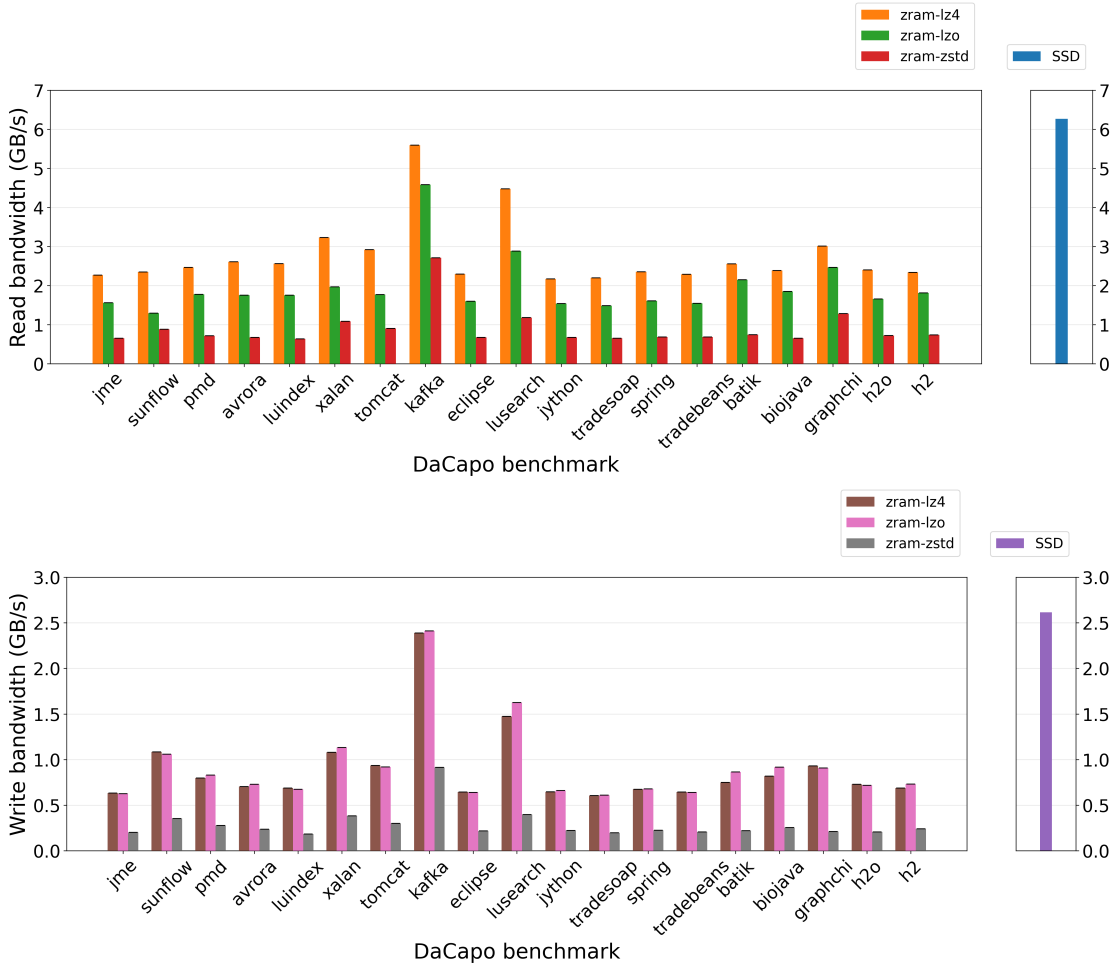

Figure 4.14: Single-process bandwidth achieved on Java heap dumps. Top graph is reads, bottom graph is writes.

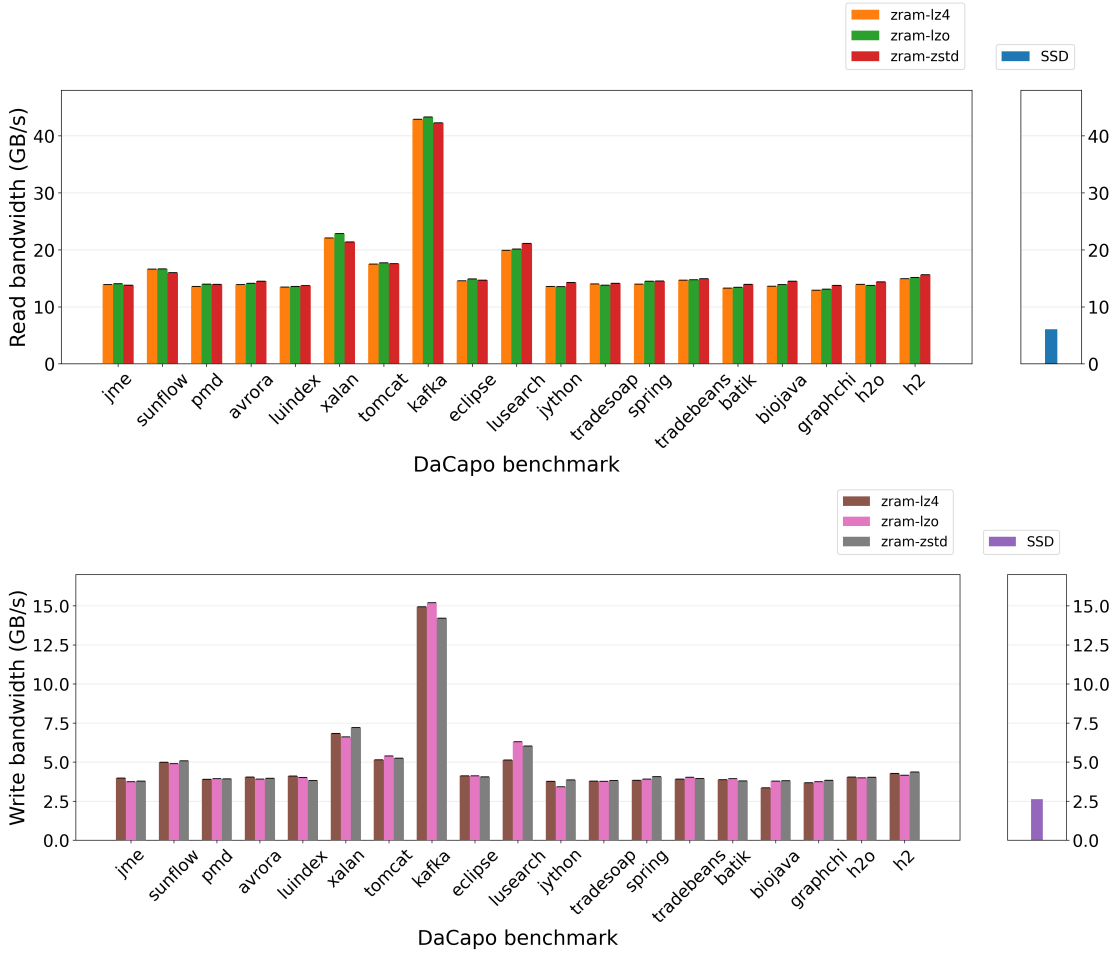

Figure 4.15: Maxmimum multi-process bandwidth achieved on Java heap dumps. Experiments use 64 processes. Top graph is reads, bottom graph is writes.

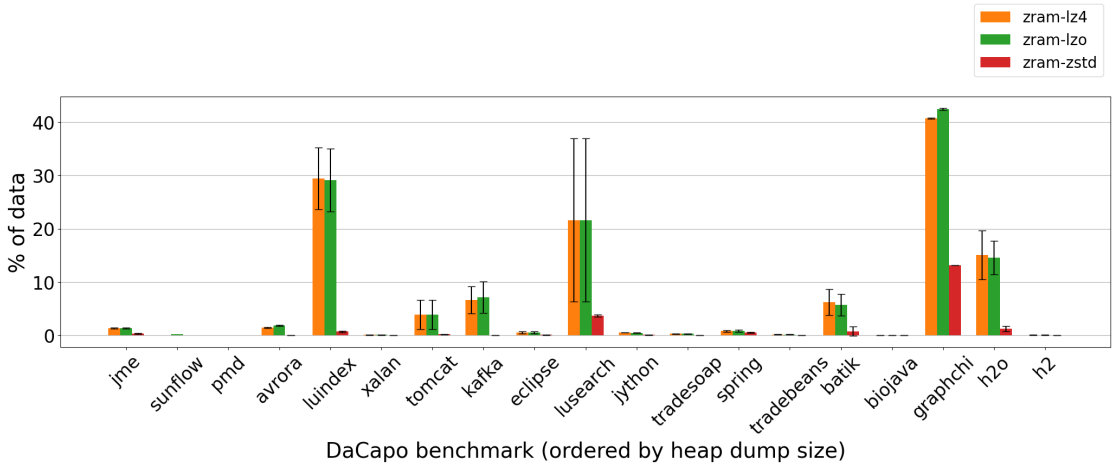

Figure 4.16: Percentage of each benchmark's data found to be incompressible and stored in uncompressed form.

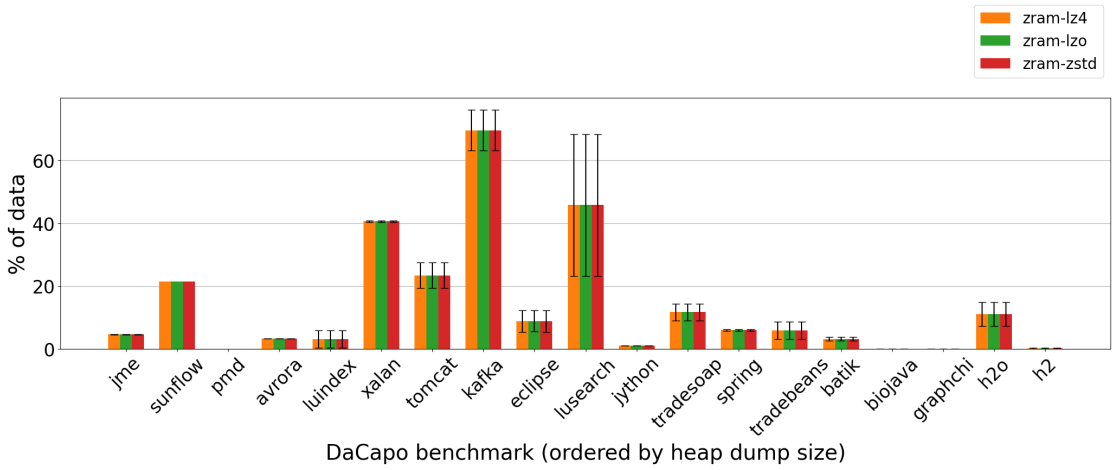

Figure 4.17: Percentage of each benchmark's data found to be trivially compressible (entire page is identical bytes), and not stored at all.

Chapter 5

Design of ZTera: Integrating ZRAM Into TeraHeap

Having evaluated the properties of ZRAM as a medium, we now move to consider its integration in TeraHeap. We first discuss the properties of data which may affect suitability of objects for storing in ZRAM. We then move to proposals for designs of ZTera systems, increasing in complexity as we introduce elements to handle potential problems. While our implementation and evaluation does not include all of the considerations discussed here, this theory would have guided our further work and is presented for completeness.

5.1 Factors Affecting Suitability of ZRAM Candidates

Given the presence of a ZRAM tier, we consider the following factors that can be used to decide whether to store an object in it:

1. **Compressibility**: The space allocated for ZRAM takes away from the total memory capacity of the system. To extract any space-usage benefits from ZRAM, the data stored must be compressible. Additionally, improved compressibility reduces compression/decompression times and hence ZRAM read/write latency, as observed in synthetic benchmarking.

2. **Hotness**: ZRAM is slower to access than RAM, but faster to read than SSD. We should hence preserve hot objects to store in ZRAM rather than SSD, but the hottest objects should be kept in RAM.

3. **Update frequency**: We observed in our evaluation that ZRAM has a significant read/write performance asymmetry. SSD also has an asymmetry and the additional consideration of write endurance, but while reads behave far better on ZRAM than SSD, writes perform worse or similar depending on the number of pro-

cesses. Hence, placing update heavy objects in ZRAM may negate its performance benefits.

4. **Longevity**: TeraHeap migrates long lived objects out of the main heap in order to reduce GC costs. SSD has relatively high capacity and can sustain long-living objects. However, very long lived objects may need to be moved out of ZRAM to avoid running out of compressed memory.

5.2 Naive ZTera

The simplest ZTera system is to simply allocate the H2 heap over ZRAM instead of SSD, which requires zero modifications to the underlying system. This setup allows TeraHeap to provide heap expansion with compression trivially. Additionally, the existing page cache for H2 handles avoiding decompression costs for hotter objects. However, there are two clear flaws with this system. Firstly, if the existing manual selection of objects for H2 does not choose particularly compressible ones, then this system fails to provide any heap capacity expansion. Secondly, ZRAM capacity is limited compared to SSD and there is no fallback option provided if ZRAM is exhausted.

We may be able to rectify the compressibility issue by adjusting the manual allocation of objects to choose compressible data only. This solution however leads to the further problem of deciding where to place incompressible data. Retaining long lived but incompressible data in H1 results in a large heap which incurs high costs on each major GC, which runs counter to one of the motivational goals of TeraHeap. On the other hand, such data cannot be placed in H2 ZRAM either.

Both of these two issues suggest reintroducing SSD as a third tier, both for storing incompressible data and allowing expansion beyond the combined capabilities of RAM and ZRAM. The challenge then becomes how to implement this additional tier.

5.3 Three-Tier ZTera With ZRAM Writeback

One method for enabling a three-tiered system that requires minimal modifications to TeraHeap involves making use of ZRAM's writeback feature. By specifying a device or partition, ZRAM can be instructed to set a backing device (Senozhatsky et al. (n.d.a)). There are three types of writeback mechanisms that can be used to instruct ZRAM to write pages to the backing device:

1. **Incompressible writeback**: Specify to writeback all incompressible pages, which are already identified by ZRAM due to its ability to store such pages in uncompressed form.

2. **Idle writeback**: Specify to writeback idle pages. Idle writeback is a two-step process: (1) mark all pages as idle, which sets the idle bit for every page which is cleared if the page is accessed, and (2) trigger writeback at a later time, which

includes any page whose idle bit remains set. It is also possible to instruct ZRAM to keep track of time since last access and set idle only for pages last accessed before a certain threshold.

3. **Manual writeback**: Specify ranges of page numbers manually to directly trigger writeback.

Incompressible writeback is an easy and obviously beneficial addition to ZTera. It is always desirable to avoid keeping incompressible data in ZRAM. The system can simply trigger writeback immediately after completing migration to H2. Hot pages will be brought back from SSD to page cache if required.

Idle writeback is desirable for reserving ZRAM for hotter objects, but is more complicated to integrate. There are several crucial choices involved including when to mark pages as idle, how to set the idle threshold and when to trigger writeback. One choice could involve checking idleness in periods of N garbage collection cycles, beginning by setting all data as idle and ending by triggering writeback, which is the same metric used by the related work Polar (Nguyen and Nguyen (2024)). We could also monitor the current free capacity of the ZRAM tier and trigger idle writeback more aggressively the closer it becomes to full, though the aggressiveness must be controlled to avoid writing overly hot data to SSD. These policies must be explored experimentally to balance their various tradeoffs.

There are two major problems that are not adequately covered by these automatic writeback mechanisms: controlling data promotion and avoiding ZRAM from running out of capacity.

As far as we understand, writeback in ZRAM is currently not reversible, which means there is no mechanism to bring written-back data from SSD back into ZRAM. As a result, idle writeback can handle demotion, but there is no mechanism to control promotion. This behaviour appears to be an oversight in the design of ZRAM itself as permanent demotion of idle data does not generally make sense, since applications' data access behaviour is always subject to change and previously cold data may become hot again. It is notable that promotion from SSD to RAM still occurs through the page cache, but upon eviction the data will return to SSD and not ZRAM despite no longer being idle. Future work should consider fixing this problem within the ZRAM kernel module itself. However, a workaround can be applied by deleting and rewriting data to ZRAM.

If ZRAM becomes full with compressible and hot data, there is no automatic mechanism to evict data to SSD to make space. This omission is reasonable to some extent as ZRAM provides the abstraction of a block device with a fixed capacity. In fact, initialising ZRAM requires setting a "disk size" parameter limiting the logical size of data stored on the device, even though the true capacity constraint is decided by the size of the physical data. We may consider manually copying the data out of the device for eviction, but this solution leads to data on the SSD tier being stored in two different places and inhibits unified management. An alternative method is to overallocate the disk size to be very

large and include the SSD tier as well, and handle eviction for making space by triggering manual writeback.

5.4 Further Challenges

ZRAM writeback allows handling several of our common problems with Naive ZTera without modifying TeraHeap itself. However, it is entirely transparent to the JVM and does not directly consider application behaviour. Regular GC is fenced from accessing H2, but the GC still accesses regions containing backward references to H1 which may reside on SSD, resulting in read/write amplification. The effect of GC should be considered in policies for choosing regions to evict from ZRAM.

Another issue created by transparency is that ZRAM writeback can only track idleness/hotness on a page granularity, while different objects in the same page may have differing hotnesses. This problem also applies to the page cache, which is currently used to cache accesses to H2. These problems require additional mechanisms within the JVM, which could include an explicit off-heap RAM cache for H2 objects and per-object hotness identification. H2 regions may then be split and recombined to balance hotness across objects in the same region. It is interesting to consider how such a solution should interact with the existing policy of ensuring objects in the same region have similar lifespans.

Overall, there is a spectrum of potential ZTera systems with varying degrees of involvement from the kernel and the JVM. The 3-tier system proposed above with ZRAM aims for the least JVM modification by exploiting as many existing kernel mechanisms as possible. Conversely, we may consider designing the system to involve the JVM more heavily. Taking this idea to the extreme would involve entirely removing ZRAM and instead using direct compression routines inside the JVM, which may allow introducing object-oriented compression algorithms as well. Exploring this space in detail is potential for future work.

Chapter 6

Evaluation: Naive ZTera

In this section we evaluate the most naive version of ZTera, in which the H2 heap is allocated over ZRAM instead of SSD. While the capacity of this version of ZTera is limited, we expect that ZRAM should be able to provide high performance compared to SSD. Another goal is to check the compressibility of data which is currently moved to H2, which is required for ZRAM to provide a benefit.

6.1 Methodology

We use a version of the Apache Spark 3.3.0 analytics engine which is modified to work with TeraHeap. Spark provides the abstraction of a Resilient Distributed Dataset (RDD) to programs. The main feature of RDDs is the ability to easily distribute data across several nodes and compute in parallel, which is supported by backing the data on storage with a distributed filesystem such as Hadoop (Kwon et al. (n.d.)). The key aspect that is relevant for us is the ability to "persist" data in an RDD. Persisting objects causes them to be cached in memory, which is often used for intermediate results in computations to reduce the costs of accessing storage. In the Spark implementation for TeraHeap, cached objects are automatically selected as H2 candidates due to their longevity and immutability. When data is cached, it is grouped by partition and immediately hinted for movement to H2 (Kolokasis et al. (2024)).

We run a selection of the SparkBench 2.0 benchmarks on the Spark engine, which represent common categories of applications that run on the Spark engine (Li et al. (2015)). The following benchmarks are run:

1. **Machine Learning (ML)**: Linear Regression (LinR), Logistic Regression (LogR), Support Vector Machine (SVM)
2. **Graph Processing**: PageRank (PR), Connected Components (CC), Triangle

Table 6.1: Size of input data for each SparkBench benchmark

Benchmark	Number of vertices	Size × # of vectors	Dataset size (GB)
Graph benchmarks			
PageRank	6M	—	13
ConnectedComponents	5.5M	—	12
TriangleCounts	1M	—	2
ShortestPaths	5.5M	—	12
SVDPlusPlus	1M	—	1.7
ML benchmarks			
LinearRegression	—	177K × 20K	66
LogisticRegression	—	177K × 20K	66
SVM	—	350K × 10K	65

Counts (TC), Shortest Paths (SSSP), SVD++

In order for Naive ZTera to be able to run without crashing, we must limit the size of the working set for each benchmark's data so that it can fit on ZRAM. This size can be controlled using the size of the input dataset for each program. The size of graph based benchmarks is primarily controlled via the number of vertices and edge density, while the size of ML based benchmarks is controlled by the dimensionality and number of vectors. We balance these parameters to create "small" sizes for each benchmark and report the associated values, in addition to the dataset size on disk, in Figure 6.1.

We now discuss the setup used to run benchmarks. Experiments are entirely conducted on the Small AMD Server. Datasets for each benchmark are pre-generated and stored on a secondary SSD, which is also used by Spark for the shuffle operation used to distribute data in the RDD across nodes (Kwon et al. (n.d.)). This data is kept separate from the primary SSD, which is used as a baseline medium for H2, in order to allow `iostat` and disk utilisation statistics to isolate H2 I/O. Spark is run inside of a cgroup with a limited memory size to control the size of the page cache and ensure that the effect of H2 medium I/O is observed. It is important to turn off the OS swapper so that no memory is swapped out to SSD/hard disk, as this results in severe performance interference. The page cache is also invalidated prior to runs.

Several monitoring scripts are run to generate statistics during each Spark run. Scripts begin reporting after Spark has completed setup, which is detected by checking whether all of the worker processes have been created. Monitoring processes include:

- `iostat`, `mpstat` and `zram_usage`, as used for Fio synthetic benchmarking
- `mem_usage`, which runs the `free` command once per second and provides a report of memory usage. Each report includes used/free memory space, memory usage of kernel buffers and page cache, and swap disk usage.
- `async_profiler`, a third party tool which enables generating flame graphs for

breaking down Java application execution (Pangin et al. (n.d.)).

- `jstat`, a JVM monitoring tool which provides information regarding Java heap utilisation and garbage collection events.

We discuss some other miscellaneous details of the overall methodology. Each benchmark run is repeated 3 times to account for noise within the JVM, and metrics are averaged. If processes fail due to memory underutilisation, Spark's default behaviour is to start a new executor and try the benchmark again. Although the Spark parameter `spark.task.maxFailures` exists to limit the number of retries (Zaharia et al. (n.d.)), it appears to not work as intended for our use case. We hence set a timeout for Spark runs, which varies based on benchmark between 10 and 30 minutes. Runs which encounter retries are discarded even if the run eventually succeeded, as monitoring processes do not reset their counters upon retry.

Our script allows setting run configurations at a high level, which are applied automatically by the script. Configuration parameters are exported into Spark's configuration files and environment variables. Using configuration files aids reproducibility by retaining a record of parameters for each experiment. To run variations of the same experiment with different parameters, another script is set up to generate temporary configuration files that modify particular parameters in a base file. `nohup` and remote monitoring were again used for experiments to limit login shell interference.

The constant parameters used by runs are shown in Table 6.2. These are categorised into parameters used by TeraHeap, parameters used by Spark and JVM parameters. The H2 file size is set to 200 GB. ZRAM is set up with a memory limit of 130 GB to leave 50 GB remaining space for H1/page cache, so this H2 size assumes a compression ratio of around 2:1. The main parameters which were varied in our experiments were the H1 size, total memory budget and H2 medium, which will be discussed individually for each experiment.

There are also a number of compile-time parameters in TeraHeap. We mainly use the defaults, though opt to turn on TeraHeap statistics tracking and timers. These statistics include time spent in each GC phase and bookkeeping operation by TeraHeap, which is useful for understanding results. Statistics tracking incurs a slight performance overhead, but one which is consistent across experiments and hence does not affect our results.

6.2 Using Overallocated Memory

We first ran each benchmark with a naively chosen H1 and page cache size, simply making it large enough to avoid the benchmark from crashing from out of memory. Based on the values set in TeraHeap's evaluation for larger dataset sizes (Kolokasis et al. (2024)), we set H1 size to 64 GB and total memory budget to 80 GB. This leaves 16 GB for additional application memory and page cache. We ran each benchmark with H2 allocated over SSD, and ZRAM devices with each of the compression algorithms

Table 6.2: Constant parameters for Naive ZTera experiments

Parameter	Value
TeraHeap	
Card size	8 KB
Region size	256 MB
Stripe size	32K
H2 file size	200 GB
Spark and JVM	
Storage memory fraction	0.9
Worker cores	8
Worker instances	1
Data partitions	256
GC threads	8

evaluated prior (lzo, lz4, zstd).

We first comment on the compression ratios observed for data which is moved to H2, as shown in Figure 6.1. Here, we find that the graph benchmarks report compression ratios of 2–3× while the ML benchmarks are actually *incompressible* for the weaker algorithms. Only zstd extracts any compression at all, and even then only 1.6×. Hence, despite our Java heap dump experiments suggesting that Java heap data is generally compressible, we find that this is no longer necessarily the case for data stored to H2. This result can be attributed to the fact that TeraHeap stores only long lived data to H2. Short-lived data uses write-optimised formats which demonstrate more opportunities for compression than read-optimised, such as gaps left in arrays for inserting more data. At the same time, such data is not acceptable for offloading since compression automatically makes them less write-optimised, which degrades performance. This result indicates a caveat to our earlier result that Java heaps are compressible, which suggests that long-lived data which is suitable for offloading may display different compressibility behaviour to general Java heap data.

The difference in compressibility between graph and ML benchmarks can be explained by considering that ML involves a large number of dense vectors and matrices that may be difficult to compress. In contrast, graph benchmarks may use dependency lists or even separate objects for each vertex, creating more opportunities for compression.

Moving on to runtime, we display the time spent by each benchmark in Figure 6.2, divided into three categories: mutator time, minor GC and major GC. Time spent in major and minor GC is calculated from `jstat` output, and these are subtracted from the total runtime to find the time spent by the mutator. This is a valid calculation in our case because the GC is not concurrent and mutator execution does not overlap with GC.

Our expectation with Naive ZTera was that using ZRAM sacrifices capacity of H2, but should retain superior performance compared to SSD-based TeraHeap. However, our

runtime results indicate that this is **not the case**. In every benchmark, ZTera performs similar or worse than SSD-TeraHeap, which is especially the case when using zstd as the compression algorithm. Drilling further into runtime, we find that while mutator time is similar between SSD-TeraHeap and ZTera, the major GC time is greater for ZTera in almost every case. Using TeraHeap's timers and the flame graph outputted by our profiling, we are able to narrow down the cause of the difference to the H2 compaction stage of GC in which TeraHeap writes data to H2, where we observe a clear difference between the benchmarks as seen in Figure 6.3. Thus, the bottleneck in ZTera is writes to ZRAM H2, and it degrades performance so significantly as to negate any potential benefit of ZTera.

In fact, the core reason for poor Naive ZTera performance is that TeraHeap only uses **a single thread to handle writes to H2**. It uses asynchronous I/O and batched writing to retain high performance of writes even when only using a single process. However, this method does not work for ZRAM as we observed in our synthetic benchmarking; ZRAM requires multiprocessed I/O and the use of multiple CPUs to extract performance. Increasing the number of GC threads alone does not resolve this bottleneck, as TeraHeap always uses a single thread regardless of the total number of threads. Hence, proceeding further requires explicitly parallelising H2 compaction.

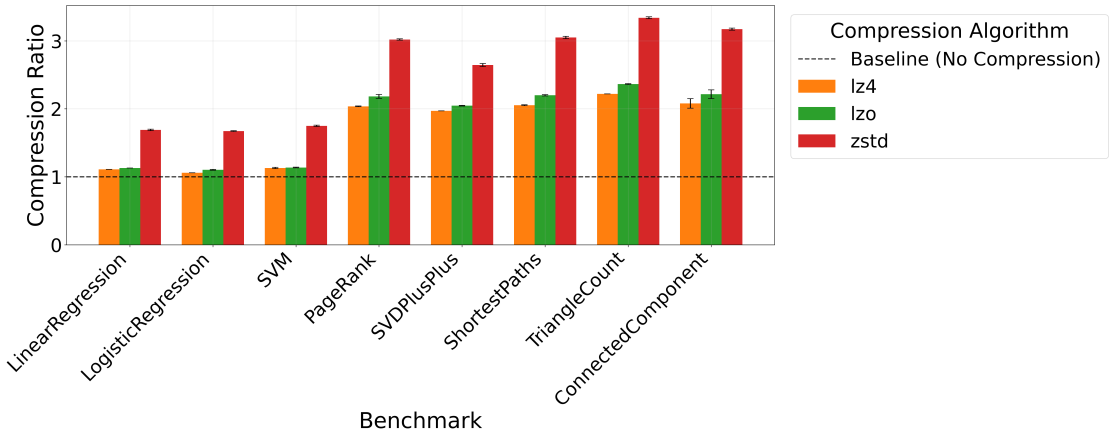

Figure 6.1: Compression ratio of H2 data for each SparkBench benchmark and compression algorithm

6.3 Using Limited H1 and Page Cache

We attempt one minor change in order to find whether Naive ZTera can still *theoretically* provide performance gains. If the write bottleneck is fixed, then we expect based on existing results that read performance of ZRAM should be far superior to SSD, which should allow ZTera to improve upon SSD-TeraHeap. In order to verify this, we adjust our experimental methodology to reduce the amount of H1 and page cache allowed for each program. Reducing H1 size triggers garbage collection more often and allows more

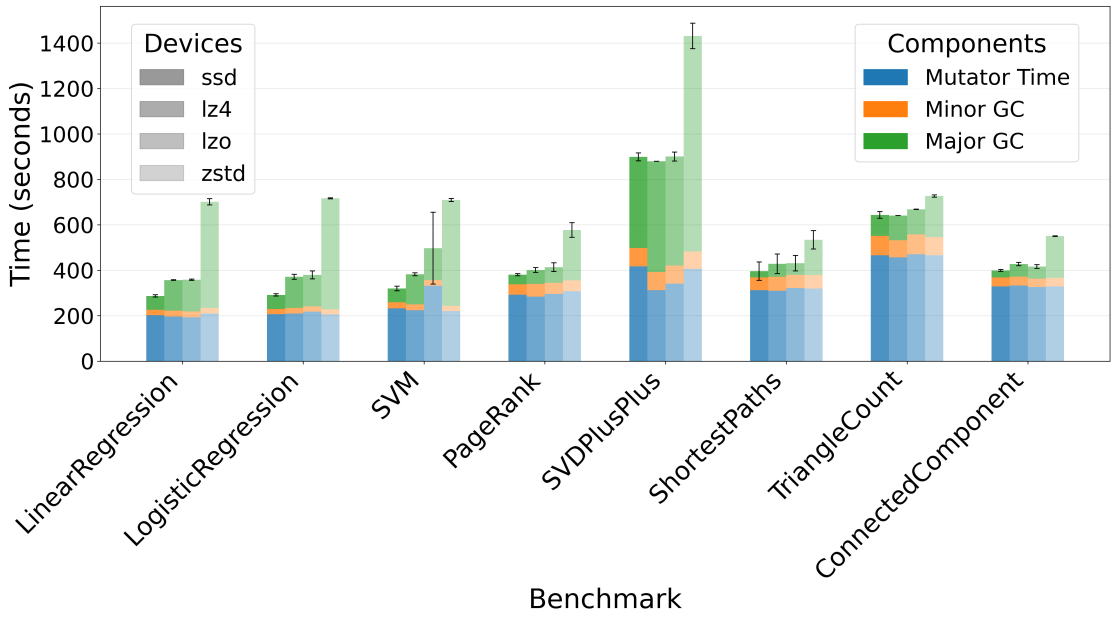

Figure 6.2: Runtime stack for each benchmark, showing mutator time, major GC and minor GC time

data to be moved to H2, and reducing page cache size increases the effect of the raw latency and bandwidth of the H2 devices on performance. Both of these choices are designed to expose the performance gap between SSD and ZRAM as much as possible to create an observable difference between ZTera and SSD-TeraHeap.

The results for H1 size of 16 GB and total memory budget size of 26 GB (allowing 10 GB of buffer and page cache) are shown in Figure 6.4. The benchmarks not shown in this graph did not run due to OOM errors. The machine learning benchmarks with lz4 also did not run due to running out of H2 space. We observe from the remaining data that in this case, the mutator time in Linear Regression and Logistic Regression is slightly lower for ZTera than SSD-TeraHeap. In particular, it is about 5% lower. These are the benchmarks with incompressible data and hence this result does not represent a real improvement for lzo and lz4. However, zstd is able to achieve both compression and lower mutator time. On the other hand, major GC time easily outweighs this benefit. Additionally, we do not see an improvement in mutator time in the graph benchmarks, which is attributed to these benchmarks not being particularly I/O bound due to reading less data from H2. We hence see some potential for improvement, albeit limited.

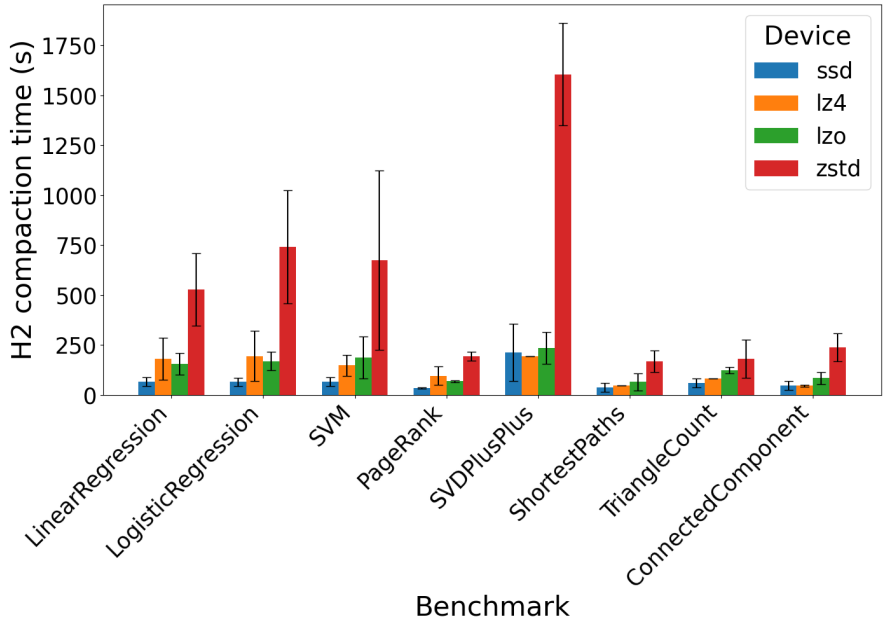

Figure 6.3: H2 compaction time for each benchmark

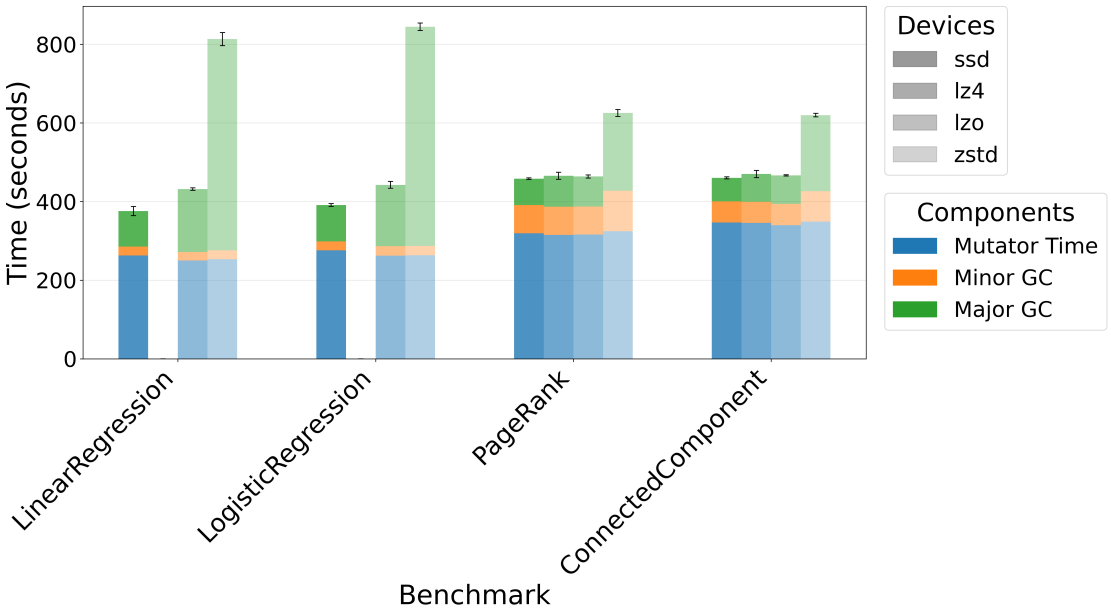

Figure 6.4: Runtime stacks for limited H1/page cache (16 GB H1, 26 GB total budget)

Chapter 7

Threats to Validity

We discuss threats to validity for our results, including both synthetic benchmarking of ZRAM and TeraHeap evaluation.

It is possible that ZRAM performance can be improved by using stronger compilation options for the kernel. The `-march=native` flag instructs the compiler to allow using instructions specific to the architecture, which can include stronger vectorisation options such as Intel AVX512. This compiler option sacrifices the portability of the binary for performance. Compression tasks have a significant compute component and can be benefitted by such optimisations. Hence, it is possible that production systems can achieve better performance from ZRAM compression modules by adjusting the kernel's compilation flags, which may change our comparisons between ZRAM and SSD. We did not investigate this option and used a default Ubuntu distribution for our servers. One can argue our results are not valid due to missing an opportunity for extracting maximum performance from our devices. However, we maintain that our data is still useful as the majority of kernels are not compiled with specialised options, and that individual studies on the effect of parameters such as number of processes would still yield the same results. Nevertheless, exploring compiler optimisations for ZRAM is an avenue for future work.

Some of our Fio experiments are relatively short in runtime (less than 5 seconds), which poses issues due to the potential impact of momentary device operations such as SSD garbage collection. We opted to use file size to control runtime, but this is flawed as different configurations in the same experiment may have highly varying runtimes. However, we performed additional validation runs with larger file sizes for various experiments such as process scaling, and concluded that our results remain consistent for longer runtimes. Our experiments may have also used the `time_based` option in Fio to ensure experiments run for a specific amount of time, but we avoided using this due to potential caching effects from accessing data in the same file more than once.

Chapter 8

Conclusion and Future Work

In the course of this work we have extensively analysed and understood the benefits and limitations of ZRAM, both in general settings as a block device and as a tier for extending the JVM heap. ZRAM demonstrates exceptional read performance which is slower than RAM but much faster than SSD. However, in CPU-constrained environments and especially with a limited number of I/O issuing threads, ZRAM write performance is poor compared to SSD. Achieving acceptable write bandwidth when using the best compression algorithm we tested requires having 8–16 processes simultaneously issuing I/O. On the other hand, SSD performance can scale to its maximum with only one process issuing requests if large block sizes or async I/O is used. We have applied ZRAM to compress heap data and identified opportunities for compression in the JVM, but later found that long-lived data may be far less compressible than general data. Finally, we design a system using ZRAM with SSD writeback to create a three-tier version of TeraHeap that is able to handle incompressible data fully and manage hotness to some extent. However, even when testing a naive version without writeback and assuming that ZRAM has sufficient capacity to contain H2, we find a critical performance bottleneck that negates all benefits of ZRAM in our system.

The major takeaway from this work is that ZRAM is not a trivial solution for performance in memory tiering systems. In fact, applying ZRAM naively in our own system resulted in a significant slowdown. While this problem can be fixed with some additional effort by parallelising writes to ZRAM, the limited capacity of ZRAM remains a problem, and takes careful allocation of data between ZRAM and SSD to manage. Additionally, the fact that it is the application's responsibility to issue I/O with multiple processes to achieve acceptable performance with ZRAM limits the extent to which ZRAM can be used in existing systems without modification. Existing literature does not explore this issue because most other systems use Zswap, in which the I/O issuing process is the kernel daemon `kswapd` and hence this aspect is transparent to the application. Retaining control over which data is sent to compressed memory requires using ZRAM, and

system developers who desire this aspect will benefit from understanding the limitations we have exposed in our work.

Future work may proceed along many different directions. One may consider fixing the bottlenecks in ZRAM directly at the kernel level by adding a mechanism for asynchronous I/O and compression. The introduction of hardware accelerators such as Intel IAA which allow offloading compression tasks complement this approach. A more classical approach would be to allow instructing the kernel to create its own thread pool for compression. One can also consider addressing the issue with ZRAM idle writeback where data accessed from the SSD tier is not promoted automatically despite no longer being idle.

Another direction is to consider alternatives to ZRAM for compression in JVM memory tiering. We have discovered several limitations caused by ZRAM itself, such as the limitation of only being able to compress at a page granularity and synchronisation bottlenecks. Using larger granularities could enable better compression with lower overhead. One approach could be to leverage compression capabilities directly in hardware. Another is to consider leveraging filesystem-level compression, for instance using XFS and BTRFS. There is also the direct approach of applying compression as an explicit routine in the JVM. Explicit compression comes with the benefits of leveraging JVM object awareness and using object-specific compression algorithms, which may be crucial for efficiently designing a compression tiered system.

Finally, our design for ZTera goes largely unimplemented, and it is an obvious direction to simply proceed further here by first parallelising compaction and then introducing writeback. We hope that our work clearly demonstrates that such an implementation has to be well considered and carefully designed to represent an improvement on existing systems.

Bibliography

Agarwal, N. and Wenisch, T. F., 2017. Thermostat: Application-transparent page management for two-tiered main memory. *SIGARCH Comput. Archit. News*, 45, 1 (Apr. 2017), 631–644. doi:10.1145/3093337.3037706. https://doi.org/10.1145/3093337.3037706. [Cited on pages 19 and 22.]

Akram, S.; Sartor, J.; McKinley, K.; and Eeckhout, L., 2019. Crystal Gazer: Profile-driven write-rationing garbage collection for hybrid memories. *Proc. ACM Meas. Anal. Comput. Syst.*, 3, 1 (Mar. 2019). doi:10.1145/3322205.3311080. https://doi.org/10.1145/3322205.3311080. [Cited on page 18.]

AMD Corporation, n.d. Introduction to IBS (instruction-based sampling). Online manual. https://docs.amd.com/r/en-US/68658-uProf-getting-started-guide/Introduction-to-IBS-Instruction-Based-Sampling. [Cited on page 20.]

Arpaci-Dusseau, R. H. and Arpaci-Dusseau, A. C., 2023. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.10 edn. [Cited on page 38.]

Axboe, J., n.d. fio - Flexible I/O tester rev. 3.3. Online manual. https://fio.readthedocs.io/en/latest/fio_doc.html. Last accessed 11th October 2025. [Cited on pages 26 and 29.]

Axboe, J.; Fu, V. K.; Kusumi, T.; Wheeler, S.; and et al., n.d. Flexible I/O tool (source code). Github repository. https://github.com/axboe/fio. Last accessed 16/10/2025. [Cited on page 27.]

Bergman, S.; Faldu, P.; Grot, B.; Vilanova, L.; and Silberstein, M., 2022. Reconsidering os memory optimizations in the presence of disaggregated memory. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2022 (San Diego, CA, USA, 2022), 1–14. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3520263.3534650. https://doi.org/10.1145/3520263.3534650. [Cited on page 22.]

Blackburn, S., 2024. Dacapo chopin benchmark descriptions and statistics. Documentation in DaCapo Chopin Github repository. https://github.com/dacapoben

ch/dacapobench/blob/main/benchmarks/doc/dacapo-descriptions-and-statistics.pdf. [Cited on page 45.]

Blackburn, S. M.; Cai, Z.; Chen, R.; Yang, X.; Zhang, J.; and Zigman, J., 2025. Rethinking Java performance analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*. ACM. doi:10.1145/3669940.3707217. https://doi.org/10.1145/3669940.3707217. [Cited on pages 5 and 42.]

Bond, M. D. and McKinley, K. S., 2008. Tolerating memory leaks. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA '08 (Nashville, TN, USA, 2008), 109–126. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1449764.1449774. https://doi.org/10.1145/1449764.1449774. [Cited on page 17.]

Chen, F.; Hou, B.; and Lee, R., 2016. Internal parallelism of flash memory-based solid-state drives. *ACM Trans. Storage*, 12, 3 (May 2016). doi:10.1145/2818376. https://doi.org/10.1145/2818376. [Cited on page 12.]

Cheong, W.; Yoon, C.; Woo, S.; Han, K.; Kim, D.; Lee, C.; Choi, Y.; Kim, S.; Kang, D.; Yu, G.; Kim, J.; Park, J.; Song, K.-W.; Park, K.-T.; Cho, S.; Oh, H.; Lee, D. D.; Choi, J.-H.; and Jeong, J., 2018. A flash memory controller for 15μs ultra-low-latency SSD using high-speed 3d NAND flash with 3μs read time. In *2018 IEEE International Solid-State Circuits Conference - (ISSCC)*, 338–340. doi:10.1109/ISSCC.2018.8310322. [Cited on page 1.]

Corbet, J., 2019. Ringing in a new asynchronous I/O API. Online. https://lwn.net/Articles/776703/. [Cited on page 15.]

Didona, D.; Pfefferle, J.; Ioannou, N.; Metzler, B.; and Trivedi, A., 2022. Understanding modern storage APIs: a systematic study of `libaio`, `SPDK`, and `io_uring`. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, SYSTOR '22, 120–127. ACM. doi:10.1145/3534056.3534945. [Cited on page 15.]

Duraisamy, P.; Xu, W.; Hare, S.; Rajwar, R.; Culler, D.; Xu, Z.; Fan, J.; Kennelly, C.; McCloskey, B.; Mijailovic, D.; Morris, B.; Mukherjee, C.; Ren, J.; Thelen, G.; Turner, P.; Villavieja, C.; Ranganathan, P.; and Vahdat, A., 2023. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023 (Vancouver, BC, Canada, 2023), 727–741. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3582016.3582031. https://doi.org/10.1145/3582016.3582031. [Cited on pages 19, 20, and 21.]

Gorman, M., 2007. Understanding the linux virtual memory manager. https://www.kernel.org/doc/gorman/pdf/. [Cited on page 19.]

Gulur, N.; Mehendale, M.; Manikantan, R.; and Govindarajan, R., 2014. Bi-modal DRAM cache: A scalable and effective die-stacked DRAM cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47 (Cambridge, United Kingdom, 2014), 38–50. IEEE Computer Society, USA. doi:10.1109/MICRO.2014.36. https://doi.org/10.1109/MICRO.2014.36. [Cited on page 23.]

Intel Corporation, n.d. Timed process event based sampling (TPEBS). Online manual. https://www.intel.com/content/www/us/en/developer/articles/technical/timed-process-event-based-sampling-tpebs.html. [Cited on page 20.]

Jones, R.; Hosking, A.; and Moss, E., 2023. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman and Hall/CRC. ISBN 9781003276142. doi:10.1201/9781003276142. https://learning.oreilly.com/library/view/the-garbage-collection/9781000883688/chapter-18.html. [Cited on page 7.]

Kim, J.; Choe, W.; and Ahn, J., 2021. Exploring the design space of page management for Multi-Tiered memory systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 715–728. USENIX Association. https://www.usenix.org/conference/atc21/presentation/kim-jonghyeon. [Cited on page 20.]

Kolokakis, I., 2024. System util (source code). Github repository. https://github.com/jackkolokasis/system_util. [Cited on page 29.]

Kolokasis, I. G.; Delis, K.; Akram, S.; Zakkak, F. S.; Pratikakis, P.; and Bilas, A., 2025. SmartSweep: Efficient space reclamation in tiered managed heaps. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR '25 (Singapore, Singapore, 2025), 71–78. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3759426.3760981. https://doi.org/10.1145/3759426.3760981. [Cited on page 4.]

Kolokasis, I. G.; Evdorou, G.; Akram, S.; Kozanitis, C.; Papagiannis, A.; Zakkak, F. S.; Pratikakis, P.; and Bilas, A., 2023. TeraHeap: Reducing memory pressure in managed big data frameworks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '23, 694–709. ACM. doi:10.1145/3582016.3582045. [Cited on pages 3, 9, and 10.]

Kolokasis, I. G.; Evdorou, G.; Akram, S.; Kozanitis, C.; Papagiannis, A.; Zakkak, F. S.; Pratikakis, P.; and Bilas, A., 2024. TeraHeap: Exploiting flash storage for mitigating DRAM pressure in managed big data frameworks. *ACM*

Transactions on Programming Languages and Systems, 46, 4 (Dec. 2024), 1–37. doi: 10.1145/3700593. [Cited on pages 8, 10, 22, 23, 61, and 63.]

Kwon, H.; Owen, S.; Dongjoon, H.; Wang, Y.; and Yao, K., n.d. RDD programming guide (Spark documentation). Online manual. https://spark.apache.org/docs/latest/rdd-programming-guide. Last accessed 29th October 2025. [Cited on pages 61 and 62.]

Lagar-Cavilla, A.; Ahn, J.; Souhlal, S.; Agarwal, N.; Burny, R.; Butt, S.; Chang, J.; Chaugule, A.; Deng, N.; Shahid, J.; Thelen, G.; Yurtsever, K. A.; Zhao, Y.; and Ranganathan, P., 2019. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19. ACM. doi:10.1145/3297858.3304053. [Cited on page 1.]

Larabel, M., 2025. Z3fold & Zbud allocators likely to be removed in linux 6.15. Online (Phoronix article). https://www.phoronix.com/news/Linux-Dropping-Zbud-Z3fold. [Cited on page 12.]

Lee, S.-H., 2016. Technology scaling challenges and opportunities of memory devices. *IEEE International Electron Devices Meeting*, (Dec. 2016). doi:10.1109/iedm.2016.7838026. [Cited on page 1.]

Lee, T.; Monga, S. K.; Min, C.; and Eom, Y. I., 2023. MEMTIS: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23 (Koblenz, Germany, 2023), 17–34. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3600006.3613167. https://doi.org/10.1145/3600006.3613167. [Cited on pages 19 and 22.]

Li, M.; Tan, J.; Wang, Y.; Zhang, L.; and Salapura, V., 2015. SparkBench: a comprehensive benchmarking suite for in memory data analytic platform Spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15 (Ischia, Italy, 2015). Association for Computing Machinery, New York, NY, USA. doi:10.1145/2742854.2747283. https://doi.org/10.1145/2742854.2747283. [Cited on page 61.]

Liu, J.; Hadian, H.; Xu, H.; and Li, H., 2025. Tiered memory management beyond hotness. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25 (Boston, MA, USA, 2025). USENIX Association, USA. [Cited on page 21.]

Ma, H.; Liu, S.; Wang, C.; Qiao, Y.; Bond, M. D.; Blackburn, S. M.; Kim, M.; and Xu, G. H., 2022. Mako: a low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and*

Implementation, PLDI 2022 (San Diego, CA, USA, 2022), 92–107. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3519939.3523441. https://doi.org/10.1145/3519939.3523441. [Cited on page 18.]

Maruf, H. A.; Wang, H.; Dhanotia, A.; Weiner, J.; Agarwal, N.; Bhattacharya, P.; Petersen, C.; Chowdhury, M.; Kanaujia, S.; and Chauhan, P., 2023. TPP: Transparent page placement for CXL-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023 (Vancouver, BC, Canada, 2023), 742–755. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3582016.3582063. https://doi.org/10.1145/3582016.3582063. [Cited on pages 19 and 20.]

Mutlu, O., 2023. Understanding and designing modern NAND flash-based SSDs (solid-state drives). Lecture series, online (videos + slides). https://safari.ethz.ch/projects_and_seminars/spring2023/doku.php?id=modern_ssds#learning_materials. [Cited on pages 12 and 13.]

Nair, A.; Kumar, S.; and Prasad, A., 2024. Telescope: Telemetry for gargantuan memory footprint applications. In *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference*, 409–424. doi:10.5555/3691992.3692018. https://www.usenix.org/system/files/atc24-nair.pdf. [Cited on page 19.]

Nguyen, D. and Nguyen, K., 2024. Polar: A managed runtime with hotness-segregated heap for far memory. In *Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '24 (Kyoto, Japan, 2024), 15–22. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3678015.3680490. https://doi.org/10.1145/3678015.3680490. [Cited on pages 18, 20, 21, and 59.]

Oracle Corporation, 1993. Java platform, standard edition troubleshooting guide, section 2.8: HPROF. Online documentation. https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr008.html#BABDFAGH. [Cited on page 43.]

Oracle Corporation, n.d. (1). GC tuning guide: Available collectors (in Java). Online. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/collectors.html#sthref27. Last accessed on 12/10/2025. [Cited on page 7.]

Oracle Corporation, n.d. (2). Storage management. Online. https://openjdk.org/groups/hotspot/docs/StorageManagement.html. Documentation associated with OpenJDK Hotspot VM. [Cited on page 43.]

Pangin, A.; Andreuzzi, F.; Kat, K.; and Hasheesh, B., n.d. Java Async Profiler. Github repository. https://github.com/async-profiler/async-profiler. [Cited on page 63.]

Park, J.; Kim, M.; Chun, M.; Orosa, L.; Kim, J.; and Mutlu, O., 2021. Reducing solid-state drive read latency by optimizing read-retry. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21 (Virtual, USA, 2021), 702–716. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3445814.3446719. https://doi.org/10.1145/3445814.3446719. [Cited on page 14.]

Pham, N.; Ahmed, Y.; and Weiner, J., n.d. ZSwap. Linux kernel documentation. https://docs.kernel.org/admin-guide/mm/zswap.html. Last accessed 14th October 2025. [Cited on page 11.]

Raybuck, A.; Stamler, T.; Zhang, W.; Erez, M.; and Peter, S., 2021. HeMem: Scalable tiered memory management for big data applications and real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21 (Virtual Event, Germany, 2021), 392–407. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3477132.3483550. https://doi.org/10.1145/3477132.3483550. [Cited on pages 20 and 23.]

Sartor, J. B.; Hirzel, M.; and McKinley, K. S., 2008. No bit left behind: the limits of heap data compression. In *Proceedings of the 7th international symposium on Memory management*, ISMM '08, 111–120. ACM. doi:10.1145/1375634.1375651. [Cited on page 18.]

Senozhatsky, S.; Kim, M.; Geffon, B.; and Gupta, N., n.d.a. ZRAM: Compressed RAM-based block devices. Linux kernel documentation. https://docs.kernel.org/admin-guide/blockdev/zram.html. Last accessed 14th October 2025. [Cited on pages 11 and 58.]

Senozhatsky, S.; Rapoport, M.; and Ahmed, Y., n.d.b. ZSMalloc. Linux kernel documentation. https://www.kernel.org/doc/html/v5.0/vm/zsmalloc.html. Last accessed on 14th October 2025. [Cited on page 12.]

Tang, Y.; Gao, Q.; and Qin, F., 2008. LeakSurvivor: towards safely tolerating memory leaks for garbage-collected languages. In *USENIX 2008 Annual Technical Conference*, ATC'08 (Boston, Massachusetts, 2008), 307–320. USENIX Association, USA. [Cited on page 17.]

Torvalds, L.; Miller, D. S.; Bergmann, A.; Brown, M.; et al., n.d. Linux kernel source code. Git repository (also available as mirror on Github). https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/. Last accessed 15th October 2025. [Cited on pages 1, 11, 31, and 37.]

Vuppalapati, M. and Agarwal, R., 2024. Tiered memory management: Access latency is the key! In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24 (Austin, TX, USA, 2024), 79–94. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3694715.3695968. https://doi.org/10.1145/3694715.3695968. [Cited on page 22.]

Wang, C.; Ma, H.; Liu, S.; Li, Y.; Ruan, Z.; Nguyen, K.; Bond, M. D.; Netravali, R.; Kim, M.; and Xu, G. H., 2020. Semeru: A Memory-Disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 261–280. USENIX Association. https://www.usenix.org/conference/osdi20/presentation/wang. [Cited on page 18.]

Wang, C.; Ma, H.; Liu, S.; Qiao, Y.; Eyolfson, J.; Navasca, C.; Lu, S.; and Xu, G. H., 2022. MemLiner: Lining up tracing and application for a Far-Memory-Friendly runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 35–53. USENIX Association, Carlsbad, CA. https://www.usenix.org/conference/osdi22/presentation/wang. [Cited on page 18.]

Weiner, J., 2018. Pressure Stall Information. Linux kernel documentation. https://docs.kernel.org/accounting/psi.html. [Cited on page 21.]

Weiner, J.; Agarwal, N.; Schatzberg, D.; Yang, L.; Wang, H.; Sanouillet, B.; Sharma, B.; Heo, T.; Jain, M.; Tang, C.; and Skarlatos, D., 2022. TMO: transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, 609–621. ACM. doi:10.1145/3503222.3507731. [Cited on pages 1, 2, 22, and 29.]

Wu, M.; Zhao, Z.; Li, H.; Li, H.; Chen, H.; Zang, B.; and Guan, H., 2018. Espresso: Brewing java for more non-volatility with non-volatile memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, 70–83. ACM. doi:10.1145/3173162.3173201. [Cited on page 18.]

Yoon, J. H.; Godse, R.; Tressler, G.; and Hunter, H., 2017. 3D-NAND scaling & 3D-SCM – implications to enterprise storage. Presentation slides from 2017 Flash Memory Summit. https://files.futurememorystorage.com/proceedings/2017/20170809_FD21_IBM.pdf. IBM Corporation. [Cited on page 13.]

Zaharia, M.; Hyun, D.; Wendell, P.; Geng, J.; and Xin, R., n.d. Spark configuration (Spark documentation). Online manual. https://spark.apache.org/docs/latest/configuration.html. Last accessed 29th October 2025. [Cited on page 63.]

Zhong, Y.; Berger, D. S.; Waldspurger, C.; Wee, R.; Agarwal, I.; Agarwal, R.; Hady, F.; Kumar, K.; Hill, M. D.; Chowdhury, M.; and Cidon, A., 2024. Managing memory tiers with CXL in virtualized environments. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24 (Santa Clara, CA, USA, 2024). USENIX Association, USA. [Cited on page 23.]