The Australian National University
2600 ACT | Canberra | Australia

Australian
National
University

School of Computing

College of Engineering and
Computer Science (CECS)

Efficient Caching For Non-Volatile Memory-Backed Search Indices

— 24 pt Honours project (S1/S2 2022)

A thesis submitted for the degree
Bachelor of Software Engineering (Honours)

By:
Jackson Kilrain-Mottram

Supervisor:
Dr. Shoaib Akram

November 2022

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the University Academic Misconduct Rules;
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or Wattle 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.

November, Jackson Kilrain-Mottram

Abstract

Our lives and responsibilities heavily rely on real-time access to information from anywhere. Search engines make this possible through enormous corpora in varying formats and target audiences. Irrespective of the intent of the content, scalability and performance are paramount to ensuring this requirement is met. A substantial overhead in search is the IO operations to stored documents, specifically on disk. Intel's Optane DC Persistent Memory (PMEM) looks to provide the scalability of disk with relative performance of DRAM. Using a search engine that exploits this technology, we look to characterise the performance difference between persistent memory and DRAM. Utilising this characterisation, we improve the viability of persistent memory as an index medium by determine effective caching strategies to employ. We present optimisation for both DRAM usage in the caches, result formats, key structures and also matching behaviour to exploit the caches to a high degree. Beyond the layer of direct contact, we also explore the applicability of direct access (DAX) and buffered IO configurations for PMEM. We detail the performance of page caching and virtual memory with the buffered IO configurations for PMEM. We also examine the behaviour of hardware caching and the impact of our chosen configurations at an architectural level. Our results show a significant improvement over the baseline characterisation of PMEM, and pave the way for future research to further improve its viability for bulk index storage and efficient search.

Table of Contents

1 Introduction 1
1.1 Search Engines . . . 1
1.2 Storage Mediums . . . 2
1.3 PMEM Applications . . . 2
1.4 Contributions . . . 3

2 Background 5
2.1 Search Engine Architecture . . . 5
2.1.1 Inverted Indexes . . . 6
2.1.2 Acceptors . . . 6
2.1.3 Query Servers . . . 6
2.1.4 Query Evaluator . . . 7
2.2 LSIP Engine . . . 7
2.3 Caches . . . 8
2.3.1 Policies . . . 8
2.3.2 Structure . . . 9
2.3.3 Memory Allocation . . . 9
2.4 Intel Optane DC Persistent Memory . . . 10

3 Result Caching for PMEM Indexes 11
3.1 LSIP Corpus . . . 11
3.2 Cache Design . . . 11
3.2.1 Cache-Thread Taxonomy . . . 12
3.2.2 Buffer Cache . . . 12
3.2.3 Allocators . . . 13
3.2.3.1 Baseline: Simplified GNU Libc . . . 14
3.2.3.2 Optimised: Two-Level Segregated Fit (TLSF) . . . 16
3.2.4 Cache Policies . . . 17
3.2.4.1 Least Recently Used (LRU) . . . 17
3.2.4.2 Dynamic Low Inter-Reference Recency Set (DLIRS) . . . 17
3.3 Entry Matching Behaviour . . . 18
3.3.1 Key Format . . . 18

3.3.2 Results as Entries . . . 19
3.3.3 Match Conditions . . . 19
3.3.3.1 Isomorphic . . . 20
3.3.3.2 Partial Type I . . . 20
3.3.3.3 Partial Type II . . . 20

4 Methodology 23
4.1 Platform . . . 23
4.2 Test Configurations . . . 23
4.2.1 Query Workloads . . . 25
4.3 Non-Uniform Memory Architecture (NUMA) . . . 25
4.4 Limiting Buffer Cache Size . . . 25
4.5 Analysing Cached Pages . . . 26
4.5.1 VMTouch . . . 26
4.5.2 VMProbe . . . 27
4.6 DRAM Backed Indexes . . . 27
4.7 System Performance Metrics . . . 28

5 Evaluation 29
5.1 Baseline . . . 29
5.2 Threaded Access Overhead . . . 30
5.3 Allocator Performance . . . 31
5.4 Contribution of Key Match Conditions . . . 32
5.4.1 Isomorphic and Partial Type I . . . 32
5.4.2 Partial Type II . . . 33
5.5 Cache Policy Performance . . . 34
5.5.1 Conjunctive (AND) Queries . . . 34
5.5.2 Disjunctive (OR) Queries . . . 37
5.6 Storage Device Access . . . 40
5.6.1 Conjunctive Queries . . . 40
5.6.2 Disjunctive Queries . . . 41
5.6.3 TLB & Page Cache Behaviour . . . 42
5.6.3.1 Page Temperature . . . 43
5.7 Hardware Prefetching . . . 44
5.7.1 L2 Prefetching . . . 45

6 Concluding Remarks 47
6.1 Hit Rates . . . 47
6.2 Inter-dependent Hybrid Type I/II Matching . . . 47
6.3 Huge Pages . . . 48
6.4 Conclusion . . . 48

A Appendix: Cache Key Matching Conditions **51**
A.1 Key Matching Conditions 51
A.1.1 Isomorphic Conditions 51
A.1.2 Partial Type I 52
A.1.3 Partial Type II 52

Bibliography **53**

Chapter 1

Introduction

The popularity of the web and online content has exploded since its introduction to the world beyond the borders of universities. Given the potential for mass availability of information, the demand for fast and reliable access has risen alongside reliance on it as the major source. In collective availability of what is essentially the entirety of human knowledge, the necessity for searching it is prime. In recently years, new technologies have begun to evolve that present new frontiers for scalability and reliability in big data contexts. Similar to how SSDs revolutionised the performance of storage critical services (Wang et al., 2013), newer persistent memory technologies are paving the way for a new era of scalability and efficiency. Search is a prime area for exploring the applicability of these technologies to improve the availability and scalability of such a fundamental and widely used service.

1.1 Search Engines

Search engines provide the means to do precisely this, classifying and structuring the information in query-able format. In order to achieve reasonable performance with large corpora, data structures and techniques are employed. Commonly, inverted indexes are used to map terms to documents (specifically document IDs, of internal importance), known as a posting list (Zobel and Moffat, 2006a).

Term based indexes allows for full use of combinatorial term queries with set operations performed on relevant posting lists. Computationally, this is more efficient and allows for structural parallelism (Zobel and Moffat, 2006b). In terms of spatial complexity, compression techniques are employed to reduce the overhead of storage capacity required for indexed corpora. Typically, indexed entries will balance the cost of partially uncompressed entries for initial lookup with the rest of the posting list entries being compressed (Zhang et al., 2008).

1.2 Storage Mediums

A major issue for search engines is requiring fast storage mediums to reduce the overall temporal cost of queries. Traditionally, there have been two major controllable storage mediums, Dynamic Random Access Memory (DRAM) and NAND flash solid state drives (SSD). The cost of DRAM is growing, and hardware support for large quantities cannot support the desired capacities that would be needed for DRAM only backed search engines. In contrast, SSDs provide the volume needed at a fraction of the cost, but with a significant increase in latency as a medium of storage when compared to DRAM.

In the last few of years, the availability of a new technology called Persistent Memory (PMEM) has given rise to potential for a medium that sits between DRAM and SSD (Intel, c). Most notably that provides relative performance of DRAM with the scalability of SSD, with two modes of interaction. One configuration supports direct access to the data (DAX) and the other via buffered IO with the kernel page cache.

Of this new storage medium technology, Intel's Optane drives are the primary source of PMEM technology in use in industry and research fields. Prior work by Akram explored the potential of PMEM in a search context, specifically as a index backing medium. While measured performance is noticeably better, it was noted that there was still a significant gap. Prior work by Xiang et al. (2022a), showed that the read-write latency difference between DRAM and PMEM is quite significant, in some cases of 2100+ cycles difference, through suffering with higher parallel thread access above 8.

1.3 PMEM Applications

PMEM provides an opportunity to scale beyond the limits of SSD. One of the largest drawbacks in large-scale processing systems, is the cost of DRAM for fast access storage in order to compensate for the differences in throughput and latency with SSDs (Rodriguez et al., 2021; Tsai et al., 2020). In order to scale to the degree that current state-of-the-art storage is at, considerations of distribution, caching and general scaling issues are forefront (Shan et al., 2017). Application wise, data centres present a prime opportunity for this technology. Generalised search falls into this category for many target audiences, such as Twitter, Meta, Google and more.

Addressing the contention of throughput in a highly distributed and active system becomes paramount to the success of a technology such as persistent memory. Much of the existing work in persistent memory application has looked into precisely this. Expanding the potential horizon of applicability in various distributed contexts. Our focus is primarily on the fundamental operation of storage with large file systems in a high throughput environment. Searching over this data presents a prime chance for characterising the configurations and usages of PMEM.

1.4 Contributions

Caching is seen as an effective means of improving performance, being widely used in search contexts, with two-level (Saraiva et al., 2001) and three-level (Long and Suel, 2005) approaches well researched and reasoned about. In particular is the barrier between the two main mediums, DRAM and SSD or DRAM and PMEM within the query processor. At this border,, caching is employed to alleviate the overhead of querying to the slower large-scale store of the full indices.

Our research focuses on caching for PMEM, where we look to:

- Improve the viability of PMEM as an index medium
- Determine effective caching policies to apply
- Optimise DRAM usage in the backing of caches
- Present optimisation to results formats for cache entries
- Evaluate result matching strategies to maximise cache usage
- Characterise cache-to-thread taxonomy and optimal ratio
- Explore the applicability of DAX against buffered IO for interacting with PMEM
- Evaluate the hardware caching impacts of the proposed caching architecture
- Detail the performance impacts on virtual memory translation and cache behaviour via PMEM interactions
- Identify the behaviour of kernel and hardware page caching with non-DAX PMEM configurations
- Examine the impact of result sizes in cache behaviour and architectural contexts

We present these through implementation of our chosen designs and optimisations over a fine-tuned search engine. In doing so, we look to evaluate the effectiveness of our chosen cache designs, and the resulting performance improvements measured.

Chapter 2

Background

In brief we present an overview of search engine architectures, query formats and processing stages. In addition, we provide an introduction to our baseline search engine, caching in search and Intel Optane DC Persistent Memory.

2.1 Search Engine Architecture

Search engines are designed to utilise instance/machine resources to produce the highest throughput with lowest query latencies. At a high level, engines are composed of index stores, query evaluators, query servers and acceptors.

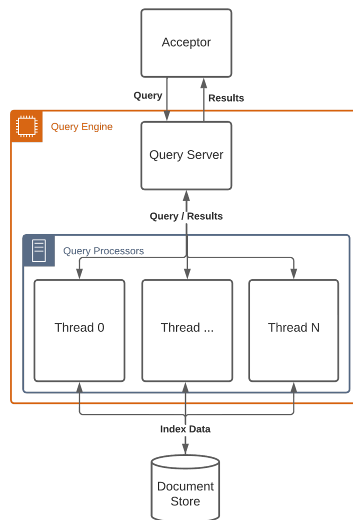


Figure 2.1: High-level overview of search engine architecture.

2.1.1 Inverted Indexes

Full text search requires efficient lookup of constituent terms to potential entries in corpora. Inverted indexes provide this exact relation as a mapping of terms to documents. More completely, a full inverted index also provides the position of the term within a document. Originally presented by (Vo and Moffat, 1998)(Zobel and Moffat, 2006a), they represent a mapping of terms to posting lists. Posting lists store term frequency, term position and document IDs. Each document ID maps to an entry in a table, containing a set of available documents. Inverted indexes are designed to accelerate search, which is driven by terms and operations between terms, by providing efficient look-ups into documents of the corpus/corpora.

Term	Posting List (Doc Ids)
Cat	53, 12, 2, 37, 41
Liverpool	96, 21, 16, 74, 85
⋮	⋮
Bridge	22, 43, 62, 2, 68

Figure 2.2: Mappings of terms to document IDs as a simplified inverted index.

Typically, these indexes are sharded, where each shard is responsible for a portion of the index, allowing for physical parallelism when interfacing with the index stores during query processing. Many industry grade engines such as Elasticsearch/OpenSearch, Lucene, Solr etc. employ this technique when interfacing with indexes, though will be done through High Availability (HA) constructs such as replicas, load-balancing and routing.

2.1.2 Acceptors

When a user submits a query, the literal query format is parsed by the acceptor. The purpose of this is to translate user interpret-able syntax into a structured format that evaluators can handle. At minimum this will split and hierarchically organise the query based on the connectives present. This organisation optimises the query for usage in processing, reducing overhead for determining operations to be performed on indexes. Internal formats of queries will differ engine to engine, where design considerations and optimisations will affect the desired structure.

2.1.3 Query Servers

A query that has been accepted and formatted will be passed to a server. The purpose of a query server is to determine the resource allocations for handling the query. Specifically,

it will queue it for query evaluators to handle. Depending on the design of the engine architecture, this may set the affinity of particular query evaluators for the query if it is trivial to know which shard is responsible for the query context. Affinity between a query and a(n) processor(s) may also be set for load balancing purposes.

2.1.4 Query Evaluator

Query evaluators interface on a one-to-one basis with shards, enacting a query on a subset of the indices. The intent of a query is defined by the use of connectives and dependent structure between terms. Before a query is passed to a relevant processor, it is pre-processed according to specific characterisations of intent to make querying an index easier. These characterisations are set operations between terms, where conjunctive (and), disjunctive (or) and negation (not) operations describe the relation between terms.

Processing a query involves looking up each constituent term on a query and performing the relevant set operation between the returned posting lists. Conjunctive queries performs intersection, disjunctive queries perform union and not performs the inversion of the next strongest binding operation. For example, `term1 AND term2` will query for `term1`, then `term2` and perform a set intersection on the returned posting lists.

2.2 LSIP Engine

Our baseline search engine is a modified version Psearchy (Wang and Lin, 2015) (Magdy et al., 2014) known as Log-Structured Inverted Index for Persistent Memory (LSIP). Psearchy is available with the MOSBENCH (Boyd-Wickizer et al.) tooling suite. The engine consists of a C/C++ based parallel indexer and query evaluator. Previous work has explored the usage of Pserachy has been used to power the open-source implementation of CiteSeer (Giles et al., 1998), known as OverCite (Stribling et al., 2006). In prior work, we see that the use of distributed hash-tables (DHTs) in Psearchy, have been used in evaluation of peer-to-peer search (Wang and Lin, 2015).

LSIP follows a simplified architecture, consisting of index stores in DRAM, SSD and PMEM mediums, scalable and parallel query evaluators and a core query server. We do not concern ourselves with query acceptors in this model, our research focuses on query evaluation and processing. LSIP uses unified indexes, with uniquely partitioned query evaluation, indexes are therefore not sharded in this model. This allows us to focus on fully-parallel query evaluation for fine-grained evaluation on a per-query basis.

In Psearchy, postings are stored in unordered lists, bound only to the order they were indexed in. We utilise Berkeley Database (BDB) to store a mapping of document names and file system location, to document IDs (figure 2.3).

As a baseline Psearchy provides two key benefits over a production tier engine. Being written in C/C++ allows for native integration with the Intel Persistent Memory Development Kit (PMDK) libraries. This reduces the overhead of design constructs re-

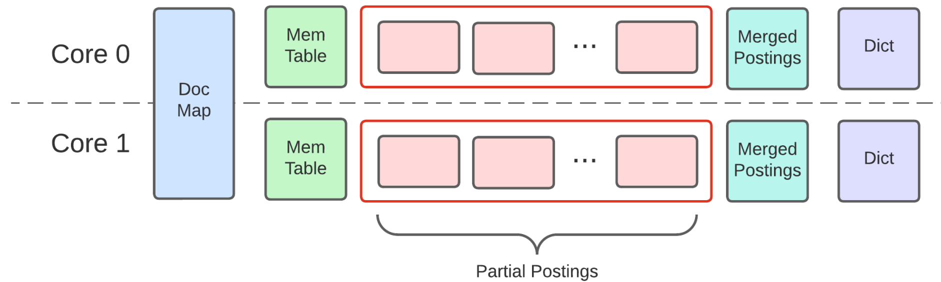


Figure 2.3: . The global and per-core data structures for indexing in Psearchy.

quired to work with the library. Psearchy does not utilise hybrid memories like that of Elasticsearch/OpenSearch, Lucene or Solr. In doing so, avoids the complications for performance analysis with garbage collected memory backing. Perturbations seen with garbage collection complicate performance results, requiring addition work to distinguish performance characteristics in recorded metrics.

2.3 Caches

Caching is a well explored topic in search, used to accelerate query evaluation. The core principle is to retain a smaller portion of the inverted index, with entries of high relevance and frequent access. Caching policies aim to exploit characteristics of search regularity, results and trends in order to keep the freshest possible results. In doing so, caches aim to reduce search latency by avoiding full inverted index lookup on every query.

Prior work has explored the potential for caching in different areas and stages of search. This also consists of multi-level caching, whereby multiple caches are employed in hierarchical order from the earliest query acceptance to latest query processing stages. Result caching, is considered one of, if not the most, fundamental component of caching in search engines (Baeza-Yates and Jonassen, 2012) (Saraiva et al., 2001) (Long and Suel, 2005) (Ozcan et al., 2012). Given our research focus is on improving the performance of the underlying medium, we identify result caching as the target point for performance improvements.

2.3.1 Policies

Replacement policies determine when and what the cache should evict based on a set of criteria, that constitute an invalid cache entry. Designs vary in the context of these metrics, some consider periodicity of usage, such as LRU, some consider recency, such as LIRS (Jiang and Zhang, 2002), others prefer to focus on frequency such as Hawkeye (Jain and Lin, 2016). Considerations for the types of data and access patterns must be made to most effectively inform on which policies are most effective for a particular use case. In search, LRU is widely known and was held highly for a long time, improvements with more advanced techniques such as those employed in LIRS (Jiang and Zhang,

2002) and DLIRS (Li, 2018) have since overtaken utilising measures of recency and reuse distance. One particular issue that has been improved upon in recent works, is multi-metric policies. Avoiding the use of only one static measure for ranking entries such as least-recent, least-frequent, and instead moving to mixed metric bases.

2.3.2 Structure

Caches are limited to a fixed memory footprint that entries can be allocated and managed within. Maximising usage of the limited space requires consideration of only the most relevant data points to track and minimising overhead of both the cache structure and entry structure (Berger et al., 2001). For example, caches such as LRU (Johnson and Shasha, 1994) will prefer to use hybrid data structures, such as dequeue hash-tables, for backing entries, in order to reduce the overhead required for implementing a dequeue structure on top of a hash-table.

2.3.3 Memory Allocation

In-memory caches are considered "hot", in terms of how often they are accessed and updated. Minimising all possible avenues of overhead is advantageous to search contexts. In addition to this, consistent operations timings are also key, in that insertion or eviction actions are bounded as low as possible in the time required to execute (Johnstone and Wilson, 1998). Underpinning these areas is allocation and freeing of memory. Optimising the allocation mechanisms can lead to steadier cache performance and potentially more entries available to be inserted (Berger et al., 2001). Allocators should be chosen well to limit several factors; fragmentation, operation complexity and structural overhead.

Fragmentation encapsulates the idea of unusable portions of memory that are created from freed blocks that cannot be merged with surrounding free blocks or are too small to allocate on their own. As they introduce un-allocatable space this means the total memory area that is usable for a cache will degrade overtime if not managed well (fig 2.4).

Spatial complexity in allocators refers to the overhead that the allocation structure imposes on the total memory available via the allocation/freeing abstractions. Reducing the overhead allows for more space to be utilised by the application, maximising the total amount of cache entries that can be held.

We note temporal complexity refers to the time taken to perform relevant allocation and freeing operations. Well designed allocators will have low bounds defined for their operations, reducing the penalty on the application while waiting for allocation/free operations to complete. In order to manage blocks, allocators will utilise a variety of structuring techniques and header formats for the required metadata for block management. The larger these structures are, the more space in the heap is not available to the application.

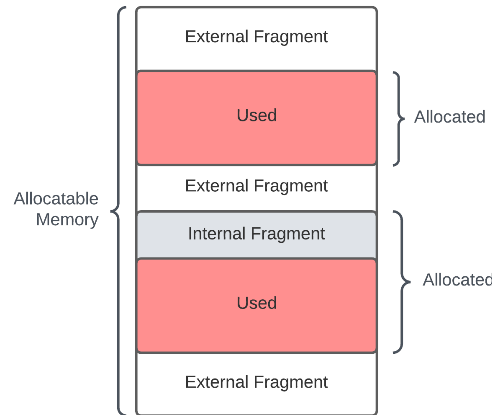


Figure 2.4: Memory fragmentation.

2.4 Intel Optane DC Persistent Memory

Intel's Optane Persistent Memory (PMEM) DCPMM modules utilise a modified version of 288-pin DDR4 DIMM interface known as DDR-T (Intel, c). Optane is non-volatile in nature, as such written data is retained without active cycling. It is however, distinct from non-volatile random access memory (NVRAM) in its use of a new technology called 3D XPoint. Optane uses an approach coined "bulk-switching" which is based on a change of bulk resistance in the material, known as Phase Change Memory (PCM). This is in conjunction with a stackable cross-grid data access array allows for byte-addressability (Xiang et al., 2022b). Cell access is performed via 64-byte granularity cache lines, however 3D XPoint supports 256-byte granularity, meaning smaller write sizes result in less efficient read-modify-write operations, an on-DIMM write-combining buffer of 16KB is used to merge small temporally local writes to mitigate this (Yang et al., 2020).

Capacity wise, Optane scales much higher than DRAM, supporting modules of 128GB, 256GB or 512GB. NVDIMMs and DIMMs are designed to be balanced in ratio depending on the configuration desired. Optane supports two configuration in Memory mode or App Direct mode. Memory mode utilises the DRR4 modules as cache for the PMEM modules. In App direct mode, both the PMEM and DDR4 modules are available to applications. In addition to this, PMEM can be accessed in either direct access (DAX) module or a mounted block device.
DAX supports byte-addressability via PMEM-aware APIs, such as those in the Intel Persistent Memory Development Kit (PMDK) (PMem.io). Utilising PMEM as a block storage device will behave like an SSD, though requiring the OS (more specifically the kernel) to be PMEM module aware. Note that this does not require applications to use PMEM aware APIs.

Chapter 3

Result Caching for PMEM Indexes

3.1 LSIP Corpus

LSIP has been indexed with the 02/05/2012 English Wikipedia corpus (Wikimedia; EnWiki) as the chosen data set. It provides a wide diversity of document content, formats and term regularity, and is well used data set in search engine performance evaluation. It is often chosen as a corpus for search engine research for these reasons. For our tests, we index the first 1,000,000 lines of the randomised corpus into the engine.

3.2 Cache Design

In real-time systems, caches are designed to handle high velocity workloads in large volumes with high variance. Cache designs address these points with consideration to the type of cache, complexity of policy, memory management behaviour and locality in the architecture.

In the LSIP search engine, we focus on result caching the query processor level. Locality specific to query processors allows for optimisations around the format of results and specific dependent behaviour of the caches, that would otherwise be difficult to achieve or introduce additional overhead. We explore this further in a later section, 3.3.

Memory management behaviour is a key consideration when addressing potentially implicit overheads in dependent systems. In particular, the memory management systems of a standard Ubuntu 20.04 distribution with GLibC. We look to consider improvements in this regard, utilising real-time specific memory management techniques, including allocator design. In section 3.2.3, we explore this further in determining baselines and optimised allocator design choices.

Many caching policies have been utilised and improved over the years. We look to

analyse the performance of two user-space caching policies (explored further in section 3.2.4) and the kernel buffer cache:

- Kernel buffer cache
- Previously well-used policy of Least Recently Used (LRU)
- State-of-the-art Dynamic Low Inter-Reference Recency (DLIRS)

3.2.1 Cache-Thread Taxonomy

Caches are rarely singular in their usage for search applications. That being said, their placement and associativity with targeted infrastructure is a contentious point, hinging on application and requirements. In the case of results caches, their placement and association is bound to that of query processor threads. Two main configurations constitute the taxonomy of cache placement in this scenario. Figure 3.1 outlines these configurations as single-cache-multiple-thread (SCMT) and multiple-cache-multiple-thread (MCMT).

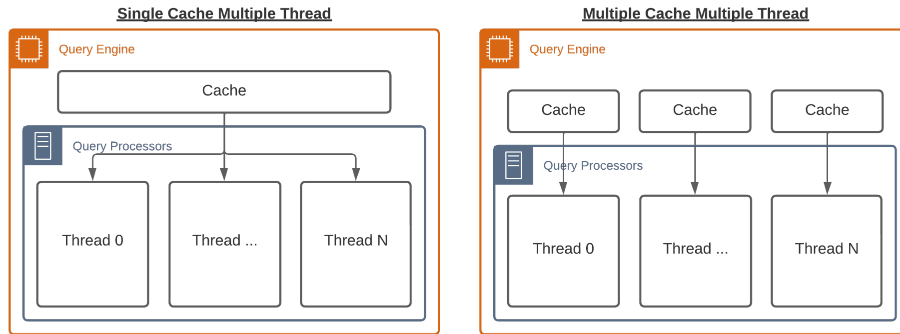


Figure 3.1: Cache-thread taxonomy.

The MCMT configuration is not limited to just a one-to-one association. It is a generalisation that allows for any combination of caches to threads, exclusive of SCMT model. In order to best utilise the locality of queries between query processors, the cache-to-thread ratio should be chosen to exploit this as much as possible. Generally speaking, this model fits the MCMT configuration. During our testing, we will consider potential configurations and their performance impacts.

3.2.2 Buffer Cache

Given the main portion of indexes are on disk, the query processors often need to bring postings from disk into memory. This is assisted by the linux kernel with a buffer cache. The core idea of the buffer cache is to retain a store of recently written or read data to improve access latency with accesses to the same data with relatively low temporal locality. Each entry in the buffer cache stores blocks of 1KB size (this may vary kernel

to kernel, which are the smallest units of disk I/O. This allows for everything from filesystem objects, to super blocks and non-filesystem content to be cached.

For our system and tests, PMEM can be configured to use either Direct Access (DAX) or buffered IO via the buffer cache. In terms of query processing, the buffer cache can be utilised underneath user level caches to provide additional acceleration for disk operations. Compared to user level caches, there is no application specific context being applied in the cache policy for the buffer cache. Its behaviour is therefore more general in regards to disk interaction than that of query state. We compare against this behaviour in order to determine the effectiveness of user level caching against with and with in-built kernel level caching for search applications using DAX enabled and disabled PMEM configurations.

Given the nature of query processing against structured inverted indices, the buffer cache could provide significant performance improvements over that of DAX. This is due to the ability to cache pages of index content to avoid the latency penalty of always accessing directly to PMEM.

3.2.3 Allocators

Several allocator designs were considered as the memory management model for the pre-allocated cache heap memory. The memory backing is a pre-allocated fixed size heap from a POSIX `mmap` call. Two allocators were chosen, one as a baseline and another as an optimised variation. Determining the best choice was done on the basis of

- Fragmentation
- Operation complexity (temporal bounds)
- Structure overhead (spatial bounds)

Previous work by Masmano et al. (2006), provided characterisations of several allocators; First Fit (Brent, 1989), Best Fit, Half Fit (Ogasawara, 1995), Binary Buddy (Ogasawara, 1995), Douglas-Lea malloc (DLmalloc) (Lea, 1996), and Two Level Segregated Fit (TLSF) (Masmano et al., 2004). Their work focused on execution time, instruction count, and fragmentation. They show that TLSF, Binary Buddy and Half Fit are the most stable in term of standard deviation of processor cycles consumed and efficiency as overall processor cycles consumed. With regards to instructions, TLSF and Half Fit have the lowest average count as well as the smallest standard deviation.

In terms of fragmentation, TLSF performs above the other allocators, followed by Best Fit. Masmano et al. (2006) note the lower fragmentation is due to TLSF's consistent allocation behaviour and ability to reuse blocks that are freed for the same size. Best Fit struggles in this regard, they cite an example of TLSF allocating 132 bytes for a request of 130 bytes, which when freed can be reused for the same allocation size. Comparatively, Best Fit allocates 130 bytes for a 130 byte allocation request but has an inability to reuse the freed block for the same 130 byte allocation. In real-time systems such as search,

maximisation of memory usage, whilst minimising overhead and fragmentation are key. Specific to caching, allowing for larger cache footprints due to a lower percentage of unusable memory present in the heap.

Table 3.1: Comparison of allocator complexity.

	Allocation	**Deallocation**
First Fit/Best Fit	$\mathcal{O}\left(\frac{\mathcal{H}}{2\mathcal{M}}\right)$	$\mathcal{O}(1)$
Binary Buddy	$\mathcal{O}\left(\log_2\left(\frac{\mathcal{H}}{\mathcal{M}}\right)\right)$	$\mathcal{O}\left(\log_2\left(\frac{\mathcal{H}}{\mathcal{M}}\right)\right)$
DLmalloc	$\mathcal{O}\left(\frac{\mathcal{H}}{\mathcal{M}}\right)$	$\mathcal{O}(1)$
Half Fit	$\mathcal{O}(1)$	$\mathcal{O}(1)$
TLSF	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Summarised in table 3.1 are the comparisons between complexity for each allocator. TLSF and Half Fit both exhibit $\mathcal{O}(1)$ bounded allocation and deallocation performance. The importance here is on a bounded maximum of instructions in the worst case, as well as a stable execution basis. DLmalloc, First Fit and Best Fit all have $\mathcal{O}(1)$ deallocation complexity, but exhibit non-linear allocation bounds of $\mathcal{O}\left(\frac{\mathcal{H}}{\mathcal{M}}\right)$ and $\mathcal{O}\left(\frac{\mathcal{H}}{2\mathcal{M}}\right)$ respectively, making them less desirable for real-time systems. Binary Buddy performs better than First Fit and Best Fit, but still undesirably with $\mathcal{O}\left(\log_2\left(\frac{\mathcal{H}}{\mathcal{M}}\right)\right)$ [BB REF] in both allocation and deallocation performance.

Implementations of chosen allocators are backed by `mmap`ed regions of fixed heap. This is done to avoid the need to override the existing allocator implementation as part of GLibC. In doing so, allows us to separate the behaviour of the chosen allocator when considering performance characteristics in test suites. Implementation wise, utilising `mmap`ed heap as the backing allows for full control over the usage of the memory in the area, without complications of otherwise needing to managed allocations distributed through the program heap.

3.2.3.1 Baseline: Simplified GNU Libc

Many linux distributions utilise an allocator derivative of ptmalloc (Gloger, 2006) which is itself a variant of DLmalloc (Lea, 1996). As such we use a simplified free-list allocator as a baseline for this characterisation. The core structure revolves around a free list, chained through headers at the top of free blocks. Headers also store used/allocated block sizes once marked as allocated via request.

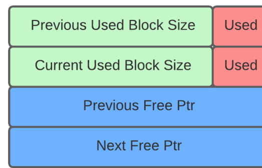


Figure 3.2: Heap header layout.

Allocations will attempt to find the first-fit free block in a free list (see fig 3.3). If a free block is found, it will be split if it is above the requested size and the resulting split blocks are both above the minimum allocatable size. Otherwise the whole block is returned. In the case that no block of adequate size if found, or this is the first allocation, the allocator will invoke a logical shift of the break in the pre-allocated heap. Heap break will only be moved if the distance between the current break and the top of the heap is at least that of the requested size + header, aligned. In this implementation, the alignment is chosen to be 4096 bytes.

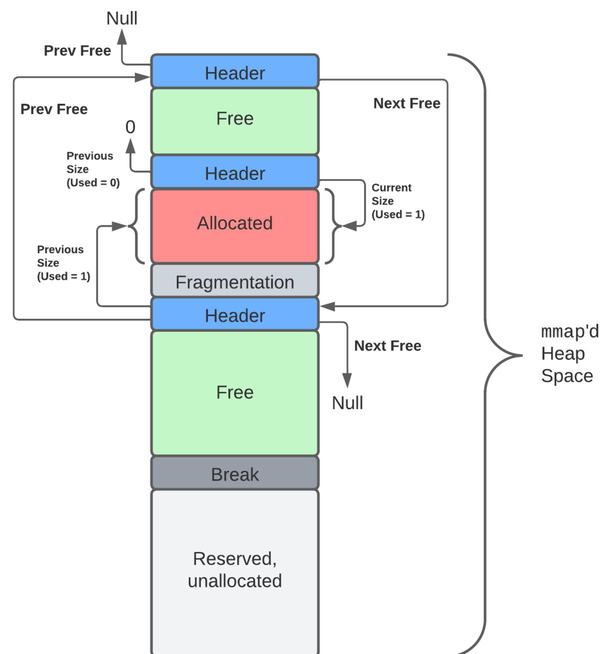


Figure 3.3: Free list heap structure.

During a free operation, the header of the block will be marked as free, and attempted to be coalesced with neighbouring blocks. Block coalescing occurs in both directions, where forward merges will clear the header of the next block and extended the size of the current and backward merges will remove the current header and extended the size of the previous free block.

3.2.3.2 Optimised: Two-Level Segregated Fit (TLSF)

TLSF (Masmano et al., 2004) utilises two levels of classification of free blocks. The first level (fl bitmap) classifies blocks under powers of 2 in the range $[2^i, 2^{i+1})$, illustrated in 3.4. Within each top level category, the range of sizes is split into $\mathcal{L}$ evenly distributed sizes where free blocks of proportional size are tracked (sl bitmap). Both the first and second level categorisations use bit masks, which are proportional to the bit-size of the system architecture. The two level categorisation of free blocks lends to the $\mathcal{O}(1)$ bound for all operations with TLSF.

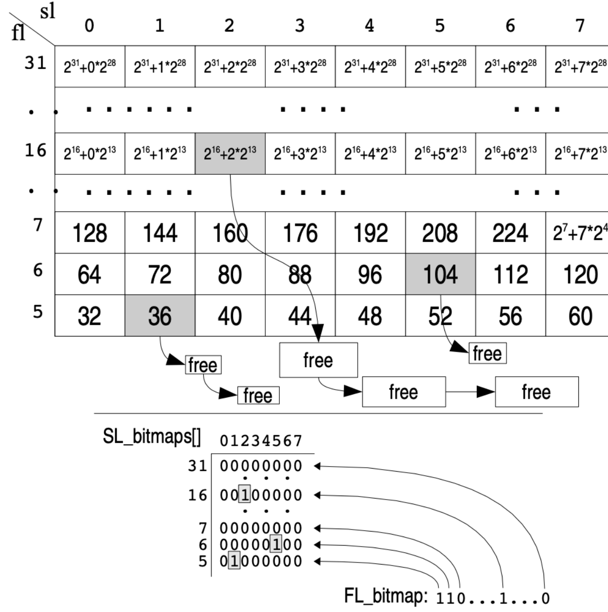


Figure 3.4: TLSF `sl` and `fl` bitmaps. (Masmano et al., 2006)

Allocations first index through the first bitmap level, finding the next smallest fitting category power of 2. If no matching category can be found in the first level, allocation fails. Otherwise, TLSF indexes the second level bitmap associated with the category found in the first. Specifically, TLSF will perform log based indexing with bit searching, as described in 3.1 and 3.2.

$$\texttt{mapping}(size) \rightarrow (f, s) \tag{3.1}$$

$$\texttt{mapping}(size) = \begin{cases} f := \lfloor \log_2(size) \rfloor \\ s := (size - 2^f) \cdot \frac{2^{SLI}}{2^f} \end{cases} \tag{3.2}$$

A similar search procedure is performed to find the most suitable block in the list of free blocks from a second level match. Internal heuristics are used to determine the most appropriate block size to minimise internal, external and structural fragmentation. All

indexing operations are bound by values set at compile time, ensuring this operation in the worst case is asymptotically bounded as $\mathcal{O}(1)$ in all cases.

Freeing a block will first attempt to coalesce with neighbouring blocks and re-categorise the resulting block as needed. In the case that this is not possible, the free block will be re-categorised within the bitmaps and associated second level list of free blocks. Under the same reasoning as the allocation, this ensures in the worst case it is asymptotically bounded as $\mathcal{O}(1)$.

For further detail on complexity bounds and algorithm details, the reader is referred to the original paper on TLSF Masmano et al. (2004).

3.2.4 Cache Policies

3.2.4.1 Least Recently Used (LRU)

Originally proposed by Johnson and Shasha (1994) and refined through years of use ((O'Neil et al., 1993; Bilal and Kang, 2014), and many more), LRU has been a staple of caching for a long time. Its structure is that of a LIFO queue, keeping entries in order of least recently used. If a cache hit occurs, the entry is re-inserted to the end of the queue. Similarly, when a new entry is inserted, it is appended to the end. Before an insertion, if the cache is at capacity, the first element in the queue is popped and returned as the evicted entry.

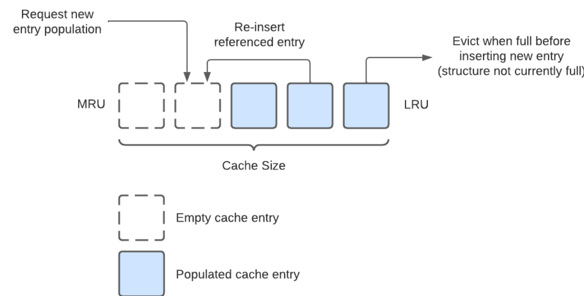


Figure 3.5: Layout of LRU structure and operations.

3.2.4.2 Dynamic Low Inter-Reference Recency Set (DLIRS)

DLIRS (Li, 2018) employs inter-reference recency (IRR) as a measure of priority for predicting future cache requests. DLIRS considers two periods of references for entries between 3 access times. The distance between the first two access times is known as reuse distance and the distance between the last two is known as recency. Reuse distance . IRR is defined as the number of unique blocks accessed between two consecutive accesses of the block, this is calculated as the maximum of either reuse distance or recency. Entries are assumed to be accessed with a higher likelihood when IRR is lower.

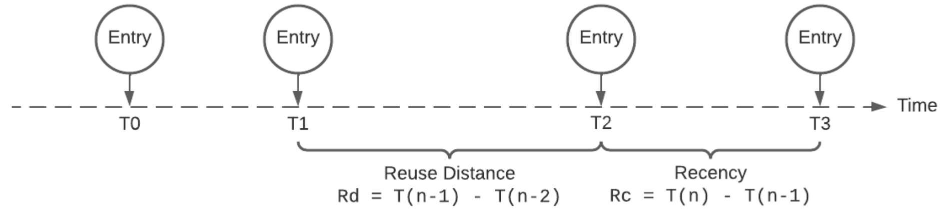


Figure 3.6: Illustration of reuse distance and recency.
(E = cache entry)
(Tn = access time)

A value is used during initialisation to determine the partitioning of the cache entry space into low IRR (LIR) and high IRR (HIR) areas. Li (2018) determined the initialising split to be optimal with 99% allocation for LIR entries and 1% for resident HIR entries. LIRs entries have additional metadata for tracking the LRU LIRs entries, known as non-resident HIRs entries.

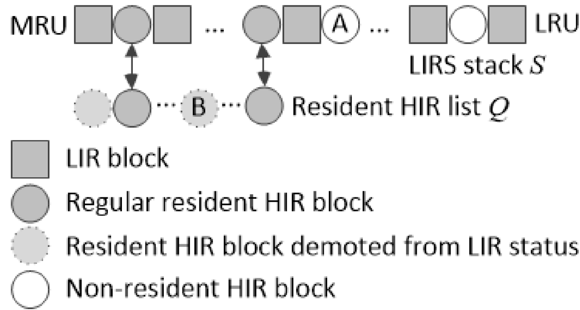


Figure 3.7: Layout of DLIRS structure. Next access on either Block A or Block B leads to dynamics in cache partitioning.(Li, 2018)

In figure 3.7, we can see that DLIRS adapts the partition of space for LIRs entries against HIRs entries by approximating the capability of non-resident HIRs entries to predict future accesses. The space for resident HIRs entries is expanded or reduced according to the determination made, which are internally marked as demoted entries.

3.3 Entry Matching Behaviour

3.3.1 Key Format

Cache keys utilised packed structuring for space efficiency (fig 3.8). A header is always present, containing the operation character (AND: ^, OR: `|`, Single: `0`), term 1 size and term 2 size. Terms are appended linearly after the header, including terminating null bytes. For single term queries, term 2 is omitted and the size is zeroed. Keys are stored

as a contiguous char array, for more efficient hashing. Immediate offsets into the key can be calculated to retrieve each component using pointer arithmetic.

Figure 3.8: Key memory layout. Top structure has both terms present, and bottom structure has only term 1 present.

3.3.2 Results as Entries

We expand on an idea proposed by Trinh et al., whereby either the result or term specific posting lists are cached based on the smaller of the two. In our scheme we always cache results, however query results contain postings in appended format, for fast term-independent lookup. A separate array of contains the ordering of postings by index.

We keep track of whether a term had results via two flags; `term1_found` and `term2_found`. Single term queries will only use term 1 associated fields. Note that found flags and offsets are irrelevant for conjunction queries, as both terms are present in every posting list entry. Disjunctive queries will make complete use of the fields, with appended results and potentially conditional term presence. The structure is packed to be as small as possible, ensuring minimal memory footprint.

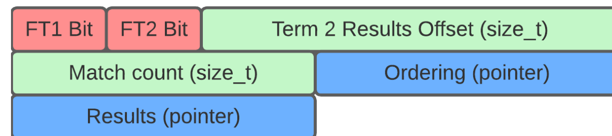


Figure 3.9: Result struct memory layout. FTn denotes the found bit for term 1 or 2.

For example, a disjunctive result with only 12 entries for term 2, orderings at `0x159c9e000` and results at `0x399c9e000` has the following layout `0 0 0x0 0xB 0x159c9e000 0x399c9e000`

3.3.3 Match Conditions

Prior work by Tolosa et al. (2014) evaluated a merged, list + intersection cache where queries can utilise previously cached results to compute the required result. We perform a similar computation on the reliance of matching conditions between keys and cached

results. There are three matching conditions that are evaluated; isomorphic, partial type I and partial type II. Each being more relaxed than the last, where partial type II allows for potentially degraded result quality.

3.3.3.1 Isomorphic

Isomorphic matches will attempt to match keys based on positional variation of terms. For example the query `other|something` will match to both `other|something` and `something|other`. Queries with conjunction will perform the same isomorphic matching behaviour against conjunctive entries. See A.1.1 for the precise matching conditions.

3.3.3.2 Partial Type I

There are two levels of partial matching, the first will match any entry that contains relevant results as a subset. For example, a single term query `melon` will match against `cheese|melon` and return the subset of results for term 2. This guarantees that the results will indeed be only melon, avoid the need for an additional cache entry exclusively for the term `melon`.

This first type will only match entries that guarantee a full subset of entries for the given query. Cases such as `test|more` tested against `test^more` will not match, since the intersection of the result sets for each is not necessarily the full set of results. See A.1.2 for the precise matching conditions.

3.3.3.3 Partial Type II

The second level of partial matching will attempt an extended set of matches. This will match entries that are any kind of subset of the query. For conjunctions, this is not trivial as further processing of entry posting lists would be needed. In order to maintain optimal behaviour, this is not performed as performance is bounded to posting list length, which degrades performance for larger corpora and larger inclusion thresholds.

Consider the following example, the current entry (e) from the cache in context is the result of a conjunctive entry between the terms `egg` and `cake`. Suppose we have a disjunctive query (q) of with the terms `egg` and `milk`. Allowing matches between these keys, would provide a subset of potential results that match the query. More formally, the set of results for the entry, and query are defined as

$$\texttt{get_postings}(term) = postings_{term} \tag{3.3}$$

$$e_{results} \subseteq \bigcap_{i=1}^{2} \texttt{get_postings}(term_i) \tag{3.4}$$

$$q_{results} \subseteq \bigcup_{i=1}^{2} \texttt{get_postings}(term_i) \tag{3.5}$$

Note that $q_{results} \subseteq e_{results}$ with regards to the example. However, the overlap in matches for the entry is potentially $|e_{results}| \lll |q_{results}|$. In this regard, the $e_{results}$ is highly unlikely to be an adequate subset of the potential full results of $q_{results}$. For disjunctive and singular subset matches are performed in full, though result accuracy is potentially lower. See A.1.3 for the precise matching conditions.

Chapter 4

Methodology

4.1 Platform

All test configurations are performed over the same physical system. We are running a Ubuntu 20.04.4 distribution of Linux as the operating system with the 5.4.0 kernel. Onboard processors are Intel Xeon Gold 6252N clocked at 2.3 GHz at base, with 3.6 GHz boosted. Each processor has 48 physical and 96 logical cores, with 71.5 MB L3 cache.

The system supports six DDR4 channels over each host iMC. Each channel is occupied by a 32 GB Micron DDR4 DIMM and a 128 GB Intel Optane (over DDR-T interface) running in a 1:4 configuration. In total, 400 GB of DRAM and 1.5 TB of Optane PMEM is avilable to the system.

Disk space is provided by 1.5 TB of Intel Optane PCI-E NVMe SSD (SD P4800X) using the same 3d XPoint technology as the Optane DIMMs via a PCI-E 3.0 interface. In addition to the SSD, a 1 TB 7200 RPM 3.5-Inch Seagate SATA-3 (6 Gbps) hard drive is available.

4.2 Test Configurations

In order to properly examine the behaviour of our chosen allocators and caching policies, we devise a set of configurations to explore the contribution of key aspects in their design. Firstly, we note that all configurations will utilise 50 threads to meet that of the baseline characterisation in the previous chapter. Additionally, our full set of configurations is tested on both allocators described in the previous chapter. We focus on cache specific tuning in these configurations, our available set of parameters that can be tuned are:

- Type (DLIRS and LRU)

- Count (at most equal to thread count, distributed evenly between threads)
- Heap size (bytes)
- HIRS ratio (DLIRS specific)
- Cache size (max entry count)
- Top K result inclusion
- Partial matching type
- New entry inclusion threshold (min length of postings in result)

Each configuration looks to evaluate various relevant combinations of the above. We do not evaluate all exhaustive combinations due to the complexity of doing so. We prioritise configurations that are most likely to provide insight into the key points of evaluating the caching behaviour and any improvements seen within. Three particular areas we do not focus on are varying top-k entries, HIRS ratio and max entry count.

For top-k, we fix this value at 20 as per a reasonable set of entries available with a given search to a popular engine such as Google. For the DLIRS configurations, the HIRS ratio is fixed at 0.01 as per the findings by (Li, 2018), discussed in the previous chapter. Our chosen heap-entry count is 1000, this is chosen with regards to the diversity of unique queries in each type. Given there is is roughly 5900 unique queries per workload, this strikes a balance between cache usage vs total query space.

Each configuration is performed on three different heap sizes, which are 5%, 10% and 20% of the size of the index. As values, these are, respectively, `210346770`, `420693540` and `841387080` bytes. For inclusion threshold, we look at minimum posting list lengths of 1, 70 and 120. Given we have a top-k of 20, this range considers whether the cache should include only results with longer posting lists or not. We explore all three key matching types, where we look to note the most optimal type without losing quality. The exception to this is partial type II, which is included as an extended characterisation of behaviour in looser matching scenarios.

All testing configurations are executed 20 times, holding distinct results for every iteration that are averaged in the final breakdown. For configurations that utilise caches, 20 executions are performed initially to warm the caches. No statistics are captured from the warming phase, we only capture during a second phase of 20 executions after the caches have warmed. This provides more accurate data, without pollution from non-typical behaviour during cache warming.

Table 4.1: Caption

Query Type	Unique Queries
AND	5886
OR	5884
SINGLE	1490

4.2.1 Query Workloads

We utilise a pre-scored list of terms with relative frequencies to determine our workloads. The terms for the Wikipedia 02/05/2012 English corpus have already been scored in the luceneutil repository (McCandless, 2012), where each are categorised into one of three categories; `HIGH`, `MED`, and `LOW`. We generate a randomised workload for each of the query types (AND, OR, SINGLE), by selecting randomly from the high terms category. Each of the generated workloads consists of 100,000 queries.

4.3 Non-Uniform Memory Architecture (NUMA)

NUMA allows the assignment of memory to particular CPUs via NUMA nodes. Memory associated with a node is considered to be local for the attached CPUs. Any memory DIMM present in the system is available to NUMA for assignment to CPUs, this is done via a mapped IO bus between the two. Localised memory ensures the CPUs have priority to it and is faster to access than remote or non-associated memory. NUMA allows access to non-local memory via other nodes, however with high latencies, since the CPUs do not have direct affinity with that memory, nor a direct IO mapping. This model extends beyond regular DRAM and into PMEM as well, since PMEM is mapped over a DDR-T interface via DDR4 DIMM slots. As such, it is can be considered memory by the NUMA system.

Depending on the support for NUMA, different systems and architectures will support different configurations. For our system we have two NUMA nodes, each with 48 CPUs and half of the available DRAM and PMEM. The Linux tool `numactl` provides control over this system, providing user level access to configuring particular processes to have affinity with particular CPUs and memory. When launching a test configuration, it is done via the `numactl` command, binding to a single node for memory and CPU. See listing 4.3 for syntax details.

```
1 numactl --cpunodebind=0 --membind=0 -- <PROGRAM AND ARGS>
```

4.4 Limiting Buffer Cache Size

For our PMEM tests, we look at both DAX and non-DAX mounted file systems. In the case of DAX no adjustments are made to kernel caching behaviour as direct write

through is supported. However, in the case of the non-DAX mount, we employ techniques to limit the size of the page cache (known more commonly as buffer cache, which we will use to refer to from here on out) to evaluate the buffer cache in comparison to the user level caches. We specifically limit the size of buffer cache using the same quantity as the heap sizes 5%, 10% and 20% variants in the user level cache tests.

In order to compare against the performance of the kernel buffer cache, we need to limit the available memory to processes. Utilising cgroups, we can set limits on the memory and swap used for processes within the group. The `cgroup-tools` package provides utilities for managing cgroups, which we use here. We create a new group under the memory controller with the `cgcreate` command (Listing 4.4 line 1). Then set a hard limit on memory + swap usage with the `memory.memsw.limit_in_bytes` variable (Listing 4.4 line 2). Performing tests within this group is done by using `cgexec` with a program, targeting the `limited_buffer_cache` cgroup (Listing 4.4 line 3). The `--sticky` flag ensures that all child processes are bound to the group as well, which accounts for threads spawned within the application context, which in our case are query processors. For configurations requiring the buffer cache with PMEM, the cgroup is created with the amount of memory for the cache in the configuration plus an additional 3KB for minimum execution guarantees (crashes occur without the extra room).

```
1 cgcreate -t $USER -a $USER -g memory:/limited_buffer_cache
2 cgset -r memory.memsw.limit_in_bytes=<BYTES> memory:/limited_buffer_cache
3 cgexec --sticky -g memory:/limited_buffer_cache <PROGRAM AND ARGS>
```

4.5 Analysing Cached Pages

Determining the contribution that the buffer cache has over using DAX with PMEM requires tooling to analyse the contents of the cache. We utilise the `vmtouch` (Hoyte, 2009) utility in order to retrieve metrics of the state of the buffer cache at a particular time. This tool will be run at an interval of 200ms and written to a file for analysis. In the following sub-sections we briefly describe the intent `vmtouch` and our usage.

4.5.1 VMTouch

`vmtouch` is a Linux utility designed to provide insight into the state of a file or directory in cache at any given time. It also allows for one to forcibly push files or directories into the cache and have them persist. For our usage, we intend only to view the state of the search indices in the cache. Results from `stdout` when running `vmtouch` are designed to be human readable, so these will be parsed externally before being analysed (See the `vmtouch` site for further details on this format).

For our purpose we utilise only the verbose flag `-v` alongside the base command. This provides the total page count and resident page count that are managed by the cache for every file that is crawled by the tool. See listing 4.5.1 for exact syntax details.

```
1 vmtouch -v <INDEX FILE>
```

4.5.2 VMProbe

Analysing the behaviour of the page cache in page level granularity requires inspection of the Linux `pagemap` and `maps` structures for specific processes in `/proc/<id>/pagemap` (Linux, a,c). In order to improve the interfacing with these structures, the `vmprobe` tool provides inspection, detailing, snapshot-ting and residency control. Utilising this we can gain insight into the page cache residency at a page level for the index, allowing for inspection over the lifetime of queries.

```
1 vmprobe cache show -w <BUCKET SIZE> -f <METRICS> <INDEX PATH>
2 vmprobe db init
3 vmprobe cache show -r <RATE> <INDEX FILE PATH> --save
```

`vmprobe` groups pages of a target into buckets when showing the residency of pages. In doing so we can adjust the granularity for easier parsing and analysis (see listing 4.5.2 line 1). `vmprobe` also provides the means to snapshot the cache state in order to inspect or restore from. It also provides a localised database structure to store these snapshots, so that we can generate a full picture of the cache state in restore-able form over a period of time (see listing 4.5.2, lines 2 & 3).

4.6 DRAM Backed Indexes

Measuring the performance of DRAM backed indices requires the allocation and mounting of memory as an accessible file system. Linux provides three main ways of doing this, each with their own particularities; block RAM disk (`brd`), `ramfs` and `tempfs`. Block RAM disks are very much a legacy structure. They allocated a fixed chunk of memory to use as the backing for a synthetic block device that is mounted as a file system (Gortmaker, 1995). The main drawback for our use case is that it requires extensive copying of data to-and-from the page cache in addition to the creation and destruction of dentries. These operations occur in the critical path during interactions with the synthetic block device, impacting performance and efficiency.

`ramfs` and `tmpfs` both directly expose the page and dentry cache as a mount point. Given that Linux caches data that is read and written already, this merely makes this directly available without requiring a device backing to forward cached entries to and from. In doing so, entries written to a `ramfs` or `tmpfs` file system are page and dentry cache entries for files and directories respectively (Rohland et al., 2001). This allows for little to no overhead when interacting with the file systems, and are guaranteed to access directly into memory. One key difference separates the two, in that `tmpfs` ensures hard limits on the usage of memory based on the initial size. Comparatively, `ramfs` allows unbounded usage of memory with can potentially lead to out-of-memory issues. For this reason we prefer `tmpfs`.

```
1 mount -t tmpfs -o size=<SIZE IN KB>k tmpfs /mnt/ramdisk
```

Creating a mounted `tmpfs` file system is easily done with the inbuilt `mount` utility. Once

completed, we re-index the search engine onto this mounted file system and perform tests targeting it. listing 4.6 details the the specific command syntax for creating a mounted `tmpfs` file system.

4.7 System Performance Metrics

To fully grasp the impact of query processing at all levels of caching, from user space to hardware, we need to be able to obtain accurate statistics with regards to the system state. Utilising the `perf` tool, we can gain insight into hot-spots in the system in order to trace their sources. `perf` provides performance counters as hardware registers that are incremented when certain conditions are met, such as a cache hit or CPU cycle completion. In addition to this, it also provides trace-points for kernel and user-level events such as page-cache events or file IO. For our purposes, we utilise the metrics detailed in table 4.2.

Table 4.2: Performance metrics captured for testing and their descriptions.

Metric	Description
`dTLB-loads`	Count of data TLB loads
`dTLB-load-misses`	Count of load misses to data TLB
`LLC-loads`	L3 cache loads
`LLC-load-misses`	L3 cache load misses
`L1-dcache-load-misses`	Count of load misses to L1 data cache
`L1-dcache-loads`	Count of L1 data cache loads
`l2_rqsts.pf_hit`	L2 cache prefetch hits
`l2_rqsts.pf_miss`	L2 cache prefetch misses to L3 or DRAM
`filemap:mm_filemap_add_to_page_cache`	Count of pages mapped to page cache
`filemap:mm_filemap_delete_from_page_cache`	Count of pages evicted from page cache

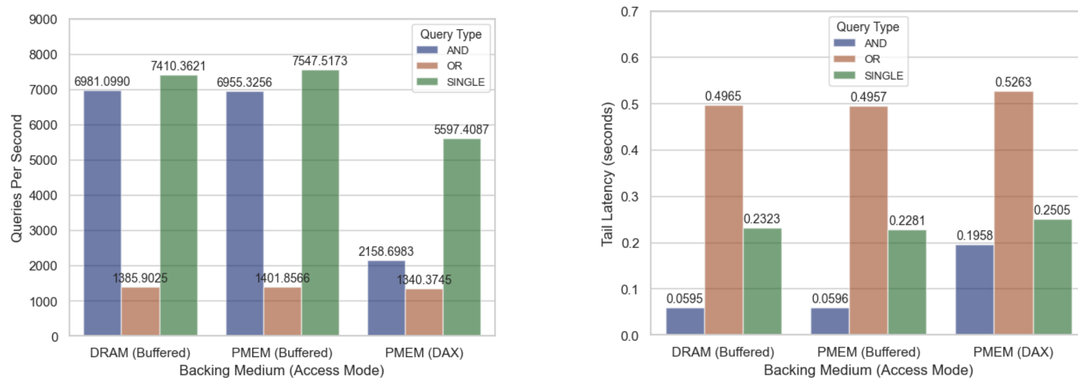
Capturing these performance metrics is done via the `perf stat` command, where the target program is provided inline via the `--` delimiter or by process ID (PID) with `-p <PID>`. We choose to use the direct invocation via delimiter as we execute the test configurations with NUMA controller and cgroup executor, to ensure that resources are allocated correctly to the processes associated with the query processors.

Chapter 5

Evaluation

5.1 Baseline

In this section we characterise the LSIP search engine across two areas, DRAM and PMEM backed indices. Each configuration is tested with three query types; OR, AND and SINGLE query performance. In doing so, we are able to identify the query types that result in the greatest performance decreases and the comparison between query performance. In our comparison, we focus on multi-threaded performance in particular as this provides the most throughput and highest load (as read and write operations) onto the backing medium. Our particular configuration is 50 threads, each of which has single core affinity with a single, resident query processor.



(a) Query Throughput between DRAM and PMEM with no caching

(b) Tail latency between DRAM and PMEM with no caching

Figure 5.1: Performance characterisation of DRAM vs PMEM as an index backing medium

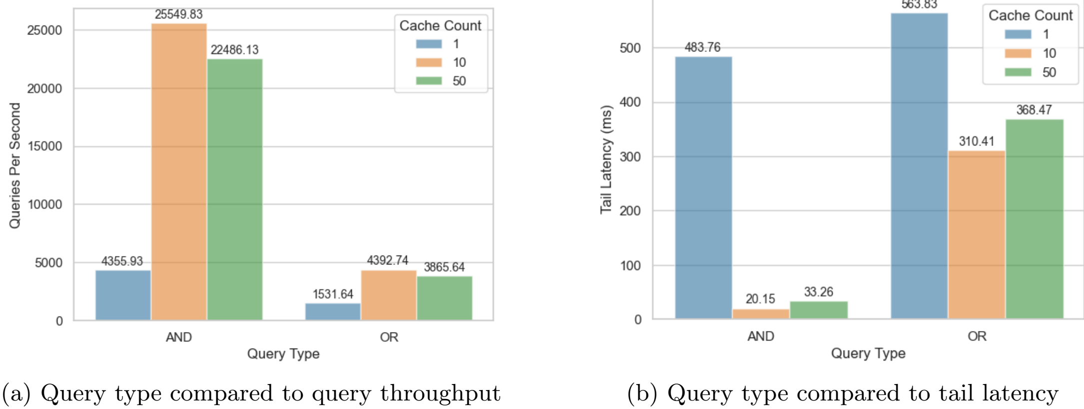
In figure 5.1a and 5.1b we see that there is a uniform performance slowdown of ~3.2× between DRAM and PMEM in the disjunctive (AND) queries. Interestingly, conjunctive (OR) and single term queries perform similarly in both mediums.

Through this initial characterisation of DRAM and PMEM, we see a significant performance gap. In order to address this, we look towards the proposed caching schemes as a means of achieving improvements in the performance of PMEM. In doing so, improving the viability of PMEM as a medium for index storage, and reasoning about the memory required to balance the performance.

5.2 Threaded Access Overhead

An initial observation is that the cache and allocator implementations utilise mutex and read-write lock (RW-lock) mechanisms for thread safety. Higher cache-to-thread ratios can impose larger performance impacts, given that lock ownership is held and passed between a greater number of contenders. In light of this, we focus on a balance of 10 caches spread evenly between the 50 threads. This lends to more stable results, being less influenced by threaded access management, rather than algorithmic and structural performance.

In addition to reducing lock-induced overhead, the choice of cache count, targets a more reasonable threshold for the percentage of potential queries and indexed documents that are managed via a cache. In doing so, we reduce the variance allowing for more efficient usage of reuse distance and recency in determining priority within the caching algorithms.



(a) Query type compared to query throughput

(b) Query type compared to tail latency

Figure 5.2: Performance of cache-to-thread ratios for 20% heap size. Each configuration utilises 50 threads (not mentioned in the graph).

Experimentally in figure 5.2, we can see that the 10:50 ratio between caches and threads yields a higher throughput and low tail latency in comparison to 1:50 and 50:50. For all

further tests, we focus on the 10:50 cache-to-thread configuration.

5.3 Allocator Performance

To gain insight into the contribution of allocators to the performance of query processing, we use a configuration that allows for all query results to be considered against a range of heap sizes with relative cache sizes.

Parameter	Configuration 1	Configuration 2	Configuration 3
Cache count	10	10	10
Heap size (% of index size)	5%	10%	20%
HIRS ratio (DLIRS only).	0.01	0.01	0.01
Cache size	1000	2000	4000
Top K	20	20	20
Matching type	Partial Type I	Partial Type I	Partial Type I
Inclusion threshold	1	1	1

We choose to vary only the heap size, as the only relevant parameter to the allocators. Utilising a fixed top-k value of 20 is inline with our initial description of the test boundaries as set out in chapter 4. We use partial type I matching to provide the most relaxed key matching criteria without impacting result quality, providing the most realistic cache interaction conditions possible within our constraints. These configurations are run for all caching policies to aggregate a complete view of all configurations. The results will consider the results of all policies simultaneously for a more complete view of performance as the specific cache policy and its contributions to performance is not the intent of this section.

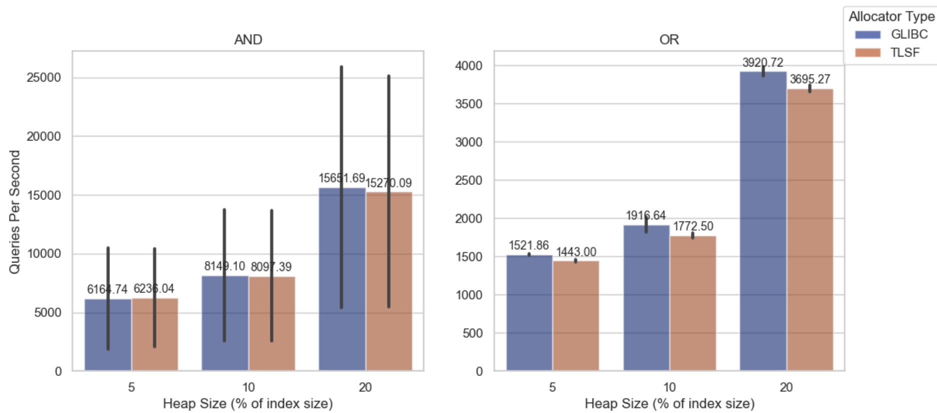


Figure 5.3: Performance impact on query throughput on varying heap sizes with allocators

Comparing allocator contribution for conjunctive queries in throughput (figure 5.3), it

is clear that there is a constant difference between the GLIBC allocator and that of TLSF. In all cases, TLSF has a higher performance than that of the GLIBC allocator. Interestingly, the standard deviation is similar between both, suggesting that the choice of allocator does not significantly contribute directly to performance for smaller heap sizes, but a gap is present with larger heap sizes. In this regard, TLSF should be preferred over GLIBC.

Conjunctive queries exhibit more diverse behaviour in the results. We can see that the GLIBC allocator shows greater variance between heap sizes than that of TLSF with regards to query throughput. This is easily accounted for as conjunctive results vary more greatly in their posting list lengths that that of disjunctive queries. Given that GLIBC is less effective at managing fragmentation and artificial overhead, the performance impacts of such structural handling is clear here. TLSF performs better with more consistent performance in all three heap sizes and well as variance in standard deviation.

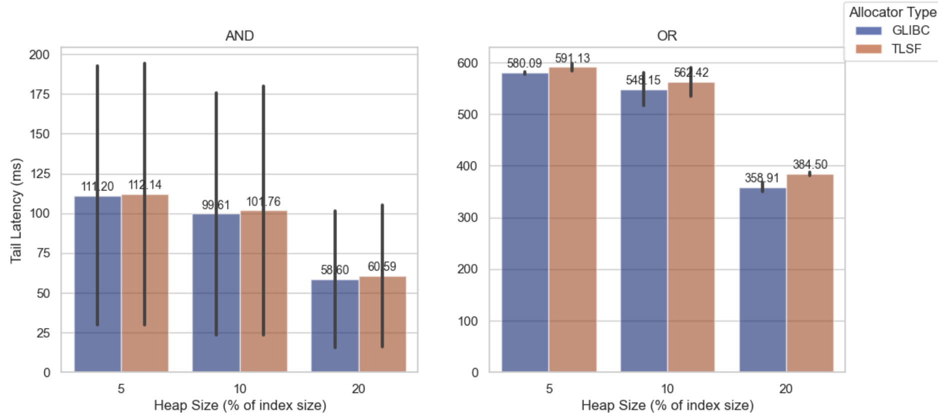


Figure 5.4: Performance impact on query throughput on varying heap sizes with allocators

Figure 5.4 showcases similar variance in the disjunctive queries compared to conjunctive with regards to tail latency. We explore this further later in section 5.5.2, identifying the details of memory access irregularity and posting list lengths.

5.4 Contribution of Key Match Conditions

As described in section 3.3.3, we employed three types of match conditions. As a result of our limited time,, we exclude partial type II. as it provides unrealistic matching conditions and does not contribute well in terms of results returned from the cache.

5.4.1 Isomorphic and Partial Type I

The most interesting criteria to compare is that of isomorphic and partial type I matching behaviour. In all query types, there is a noticeable improvement in the performance when

using the extended matching behaviour of partial type I. At peak, we see 1.35% greater hit rate for DLIRS and 0.27% for LRU. Given that partial type I guarantees absolute subsets, in the same regard as isomorphic, this suggests that the extended conditions should always be favoured.

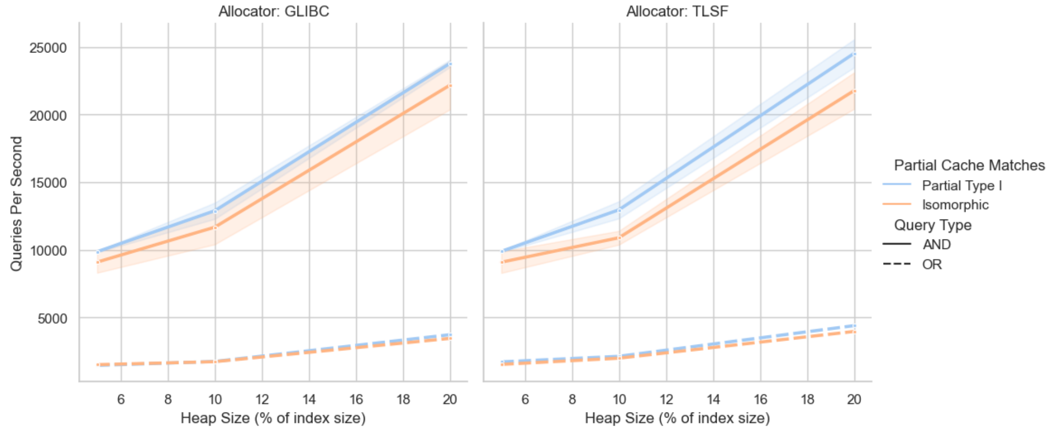


Figure 5.5: Comparison of isomorphic and partial type I matching behaviour. DLIRS is not present in the disjunctive query type, due to time constraints.

A particular insight is the variance in performance gains for conjunctive queries always exceeds that of the isomorphic category. This indicates that additional behaviour of the conjunctive cases in partial type I contributes to a significant portion of underutilised key matching scenarios not captured within the isomorphic category.

This indicates that the utilisation of fine-grained key matching behaviour does indeed contribute to measurable performance improvements. One should note that this does require a larger result size per cache entry, but overall can contribute to more efficient cache usage. In the best case for this result scheme, three entries can be stored as one. This exploits the relation between a conjunctive query being comprised on two single term queries and the isomorphism of key ordering.

As shown, partial type I provides the highest performance gains over all the key matching behaviours, without degrading result quality. Due to the limitations of time and in spirit of analysing the highest quality behaviours, we will utilise partial type I matching in all tests from here on out. A more complete view of the contribution that partial type I matching provides, would need to be done over a mixed query set, given the time constraints, this was not possible however. As an avenue of future research, this should be evaluated to complete the characterisation presented here.

5.4.2 Partial Type II

Utilising partial type II comes with the consequence of needing to do excessive pre-processing and continual searching on the basis of the quality of results found. This is

due to the fact that it allows matches between subsets that don't guarantee overlap. For example, that of a disjunctive query matching a subset of a conjunctive query.

By itself, this logic is problematic, however, with further deliberation over implementation could prove to be useful. More specifically that utilising partial type II to create an aggregation of approximate results as an "early preview" of sorts. This preview could be refined by more precise behaviour such as isomorphic and partial type I matching. One could imagine an approach of separating potential matches by layers of approximation. Similar to the structure of the TLSF allocator, albeit rather abstractly. However, given that this is not the topic of our research, we leave it as a potential avenue for further exploration into matching behaviour.

5.5 Cache Policy Performance

[1] From here on out, we will refer to configurations in figures by initialism of the configuration properties. This format is defined as `<Medium>-<Access>-<Cache>-<Allocator>` where

- Medium: `D`=DRAM, `P`=PMEM
- Access: `D`=DAX, `B`=Buffered
- Cache: `N`=None, `D`=DLIRS, `L`=LRU, `R`=Random
- Allocator: `N`=None, `G`=GLIBC, `T`=TLSF

5.5.1 Conjunctive (AND) Queries

Between the caches, DLIRS outperforms LRU in terms of hit rate, which can be attributed to the more advanced prioritisation and de-prioritisation scheme. In tables 5.1 & 5.2 we can see that with the lower 5% heap size, LRU out performs DLIRS in hit rate. However, with higher heap scaling this quickly becomes the opposite, with DLIRS scaling higher for mean, min and max hit rates. It should be noted, however, that given these are conjunctive queries, matching behaviour does not take affect, given that both terms in the query are tightly coupled which we do not extrapolate between cache entries in our scheme. We can see this here with both matching behaviours being relatively identical.

The relative hit rate improvements from exponentially increased heap size appear to be roughly linear in nature. This suggests that in the case of conjunctive queries, the performance benefits of a larger heap must be balanced against the relative resource usage. However, given that conjunctive queries are just a subset of the potential query space, this may yet balance out. In order to fully characterise this behaviour, a full

[1] Due to time constraints and implementation issues in C, we weren't able to acquire the DLIRS results for OR queries in time for this publication. Similarly, SINGLE term queries are only present for the baselines and not for LRU or DLIRS.

Table 5.1: Hit rate aggregate comparison

Cache Type	**Heap Size**	**Mean**	**Min**	**Max**
DLIRS	5	16.90%	16.81%	16.95%
DLIRS	10	36.80%	36.39%	37.25%
DLIRS	20	71.54%	70.05%	73.10%
LRU	5	17.44%	17.39%	17.47%
LRU	10	34.28%	34.27%	34.29%
LRU	20	68.20%	68.08%	68.46%

Table 5.2: Percentage change between DLIRS and LRU hit rates

Heap Size	**Mean**	**Min**	**Max**
5	3.19%	3.46%	3.06%
10	-6.84%	-5.81%	-7.94%
20	-4.66%	-2.82%	-6.35%

enumeration of all test suites including single term queries would need to be performed to fully grasp the impact of this behaviour.

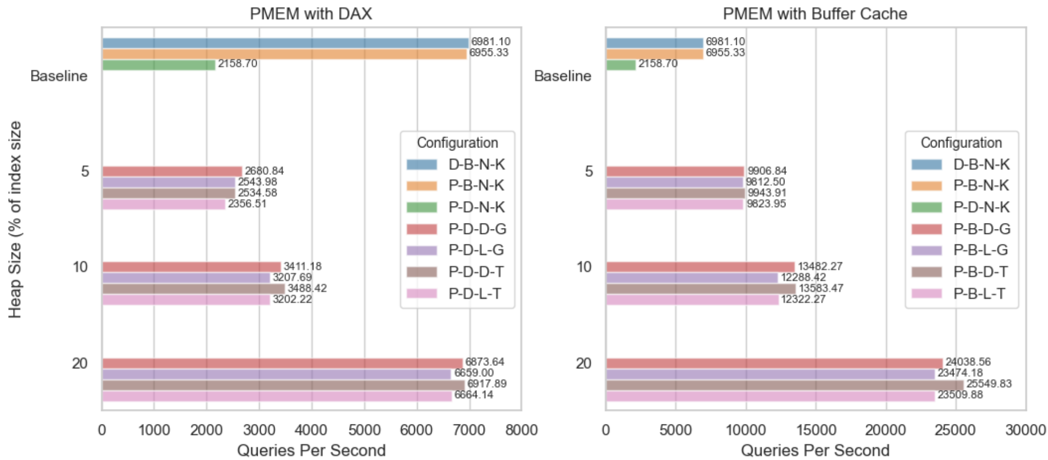


Figure 5.6: Query throughput of varying heap sizes of each cache + backing combination.

In the results of figure 5.6, we can see that mounting PMEM with DAX significantly inhibits the performance of search. Note that the user-level caches within the query processors effectively bridge the gap left by the absence of the buffer cache. In doing so, we don't gain any performance improvements that would not otherwise be present

at the baseline by simply disabling DAX. Utilising the buffer cache provides signifcant performance improvements when paired with caching in the query processors. To that end we see up to 265.986% increase over DRAM and 267.342% increase over PMEM with buffer caching with DLIRS paired with the TLSF allocator.

It is clear that the improvements DLIRS makes with tracing recency and reuse distance provide a consistent performance improvement over LRU. As the heap size grows, this difference becomes more apparent. In addition to this, considering that the implementation of DLIRS ensures that

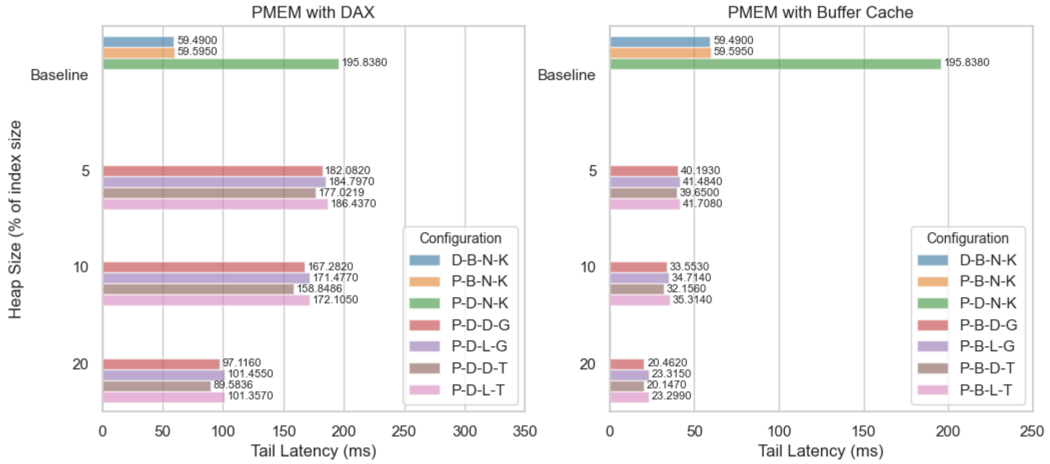


Figure 5.7: Latency of tail queries in milliseconds of varying heap sizes of each cache + backing combination.

DLIRS provides the greatest performance improvement in all test configurations. In the most optimal scenario of 20% heap, we see that it outperforms DRAM only search. This is not exclusive to a particular allocator either, both GLIBC and TLSF allocators provide this performance, with TLSF slightly outperforming GLIBC configurations.

The triple categorisation of entries as LIRS, non-resident HIRS and resident HIRS, lends well to being able to temporally adapt to content varying in recency and reuse distance over the lifetime of the query set. This extra room for adapting to changing query targets, is clearly shown when comparing to LRU with only a single recency metric. Despite the obvious advantage of DLIRS' extended structure, LRU does not fall far behind. In the DAX enabled configuration the difference is more pronounced than with the buffer cache enabled configurations

Increasing the heap size leads to greater divergence between the query throughput between DLIRS and LRU. However, interestingly, the latency tends to converge between DLIRS and LRU with respect to increased heap size. In terms of cache specific operation, the optimal scenario for DLIRS is precisely that of LRU, where a single entry is moved in priority. In A broader scheme, when considering an optimal set of queries that

are given to the cache, the majority of them will perform the exact process. As such, the converging behaviour in tail latency is accountable. It should be noted, that there is an insurmountable difference in algorithmic complexity between DLIRS and LRU, by design. So this convergence is only approximate, since the minimum instruction count in the most optimal scenario will always been higher for DLIRS.

5.5.2 Disjunctive (OR) Queries

Regardless of backing medium, disjunctive queries are substantially less impacted by caching optimisations. We can see from the outset that between the baselines (figures 5.10 & 5.8), there is very little difference in performance. Tail latency is virtually identical between DRAM and PMEM, with a difference of ~0.16%. We do a consistency of improvement when using the buffer cache over DAX. This produces a slight improvement of 5.84% over the DRAM baseline.

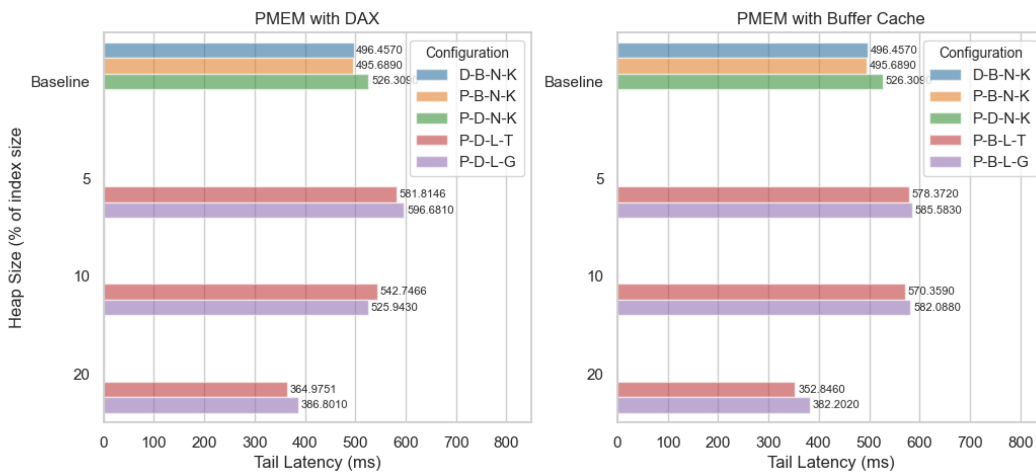


Figure 5.8: Latency of tail queries in milliseconds of varying heap sizes of each cache + backing combination.

Disjunctive queries are by necessity much larger than that of conjunctive queries, given the potential to match a larger subset of documents. We can see that this contributes significantly to the performance in both DAX and non-DAX configurations. This highlights several performance bottlenecks. In particular, relative processing time of these queries is substantially longer than that of conjunctive queries, the tail latency difference between figures 5.7 and 5.8 indicate this, with a 175.621% increase in the most optimal category of 20% heap (20.147ms vs 310.413ms).

Mapping out the posting list lengths for the conjunctive queries against disjunctive (figure 5.9), we immediately see a stark difference between the two. Conjunctive queries exhibit a considerably smaller footprint than that of disjunctive queries, on average ~827787 compared to ~11117563, a difference of 172.281%. In terms of critical path,

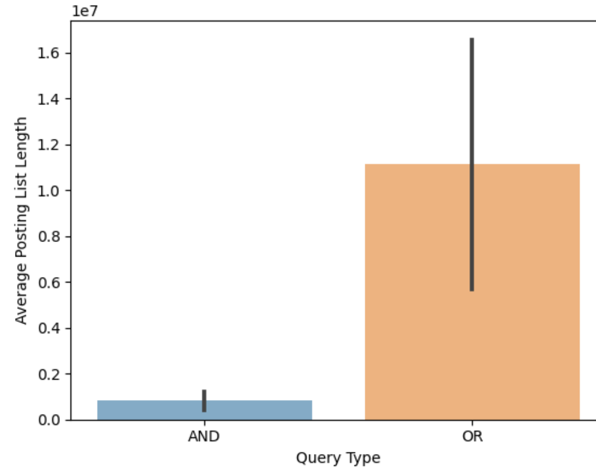


Figure 5.9: Sizes of posting lists in bytes averaged over each cache, with 10 affiliated query processors.

memory operations are forefront, implying considerable overhead when interacting with longer postings lists.

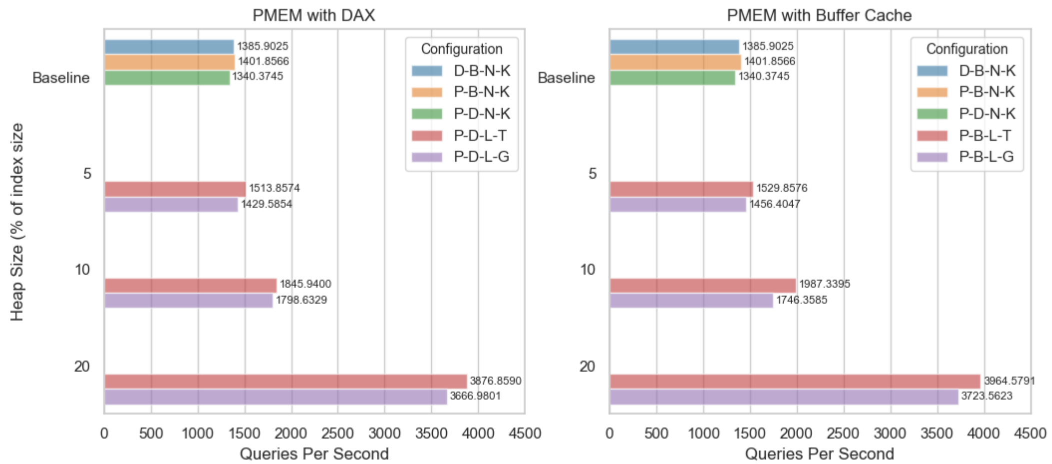


Figure 5.10: Query throughput of varying heap sizes of each cache + backing combination.

As mentioned before in section 5.3, the disparity between conjunctive and disjunctive queries will be explored further here. this leads to irregular memory access behaviour. When handling disjunctive queries, terms are matched through parallel iteration of both terms in a query. In doing so, the iterator index for each positing list is updated irregularly. Extrapolating this further, we look at the commonality bewteen conjunctive and disjunctive query processing logic, alongside the separating characterisation; posting list

length.

```
1  while(iterator1 < post_count1 && iterator2 < post_count2) {
2      if (post1[iterator1].docNumber == post2[iterator2].docNumber) {
3          match_array[post_match] = post1[iterator1];
4          post_match++;
5          iterator1++;
6          iterator2++;
7      } else if (post1[iterator1].docNumber < post2[iterator2].docNumber) {
8          iterator1 += 1;
9      } else {
10         iterator2 += 1;
11     }
12 }
```

Listing 5.5.2 is taken directly from the `two_term_and` processor method in LSIP, this performs a merge of both posting lists, sorted by document ID with parallel iteration. When looking at the L3 (LLC) cache load misses, for both disjunctive and conjunctive queries, there is a substantial difference between the two. Figure 5.11 showcases this disparity, where we can see that disjunctive queries exhibit 64.69% of all L3 cache load accesses are misses, compared to only 39.72% miss rate for L3 cache load accesses with conjunctive queries. It is important to note that the volume of L3 loads is substantially different between the two, with a difference of 40677141912 total load misses.

Despite the higher volume of L3 loads in conjunctive query workloads, the locality of misses is much broader. In terms of handlers in LSIP, the `two_term_or` method handles three cases, two of which are more optimal in load behaviour than the other. The third case overlaps with the behaviour of the `two_term_and` handler for conjunctive queries;

- Case 1: The first element of term 1's posting list, has a document ID greater than the last element of term 2's posting list. Resulting combined posting list is term 1's postings concatenated with term 2's.
- Case 2: The first element of term 2's posting list, has a document ID greater than the last element of term 1's posting list. Resulting combined posting list is term 2's postings concatenated with term 1's.
- Case 3: Otherwise, merge both posting lists, sorted by document ID with parallel iteration

It should be noted that the first two cases listed above are rare, in that postings lists are almost never perfectly aligned to allow for such optimsiations to make substantial difference. Given the substantial logical overlap between the two, and the results presented in figures 5.9 & 5.11, we can conclude that the separating property is indeed posting list length. It is interesting to note, that with a higher posting list length the disjunctive queries exhibit a higher L3 hit rate. Additionally, over time L3 to DRAM lookups will result in better performance due to the page cache become more efficiency saturated for DAX configurations. We discuss this behaviour further in section 5.6 and consider the behavioural differences of DAX compared to buffered IO with PMEM.

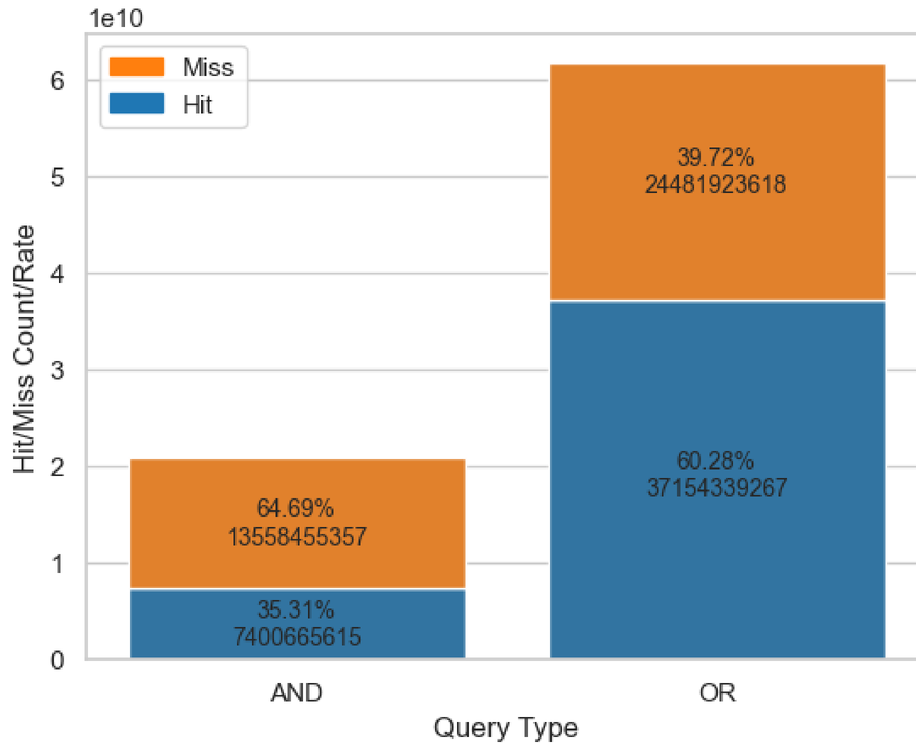


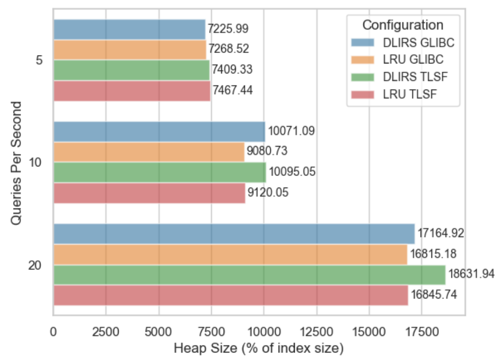
Figure 5.11: L3 load hit rates by query type. These are collected from the 20% heap configuration. There is little to no variance between GLIBC and TLSF in these metrics, as such only the TLSF are used in this figure.

5.6 Storage Device Access

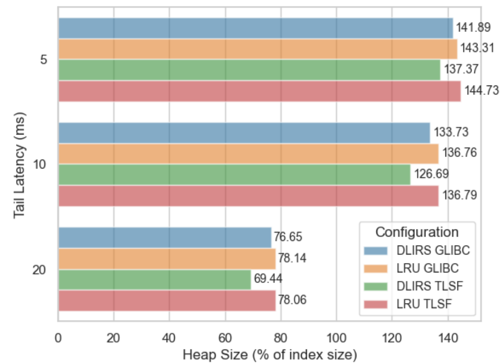
As previously mentioned, all test configurations are evaluated with PMEM configured to use DAX or the kernel buffer cache. Inline with the hypothesis presented in section 3.2.2, DAX increases the latency for accessing indexed data. In all test cases we can see that DAX impedes the performance of search. In figures 5.6 & 5.10, the impact of DAX vs buffered access is visible in both query throughput and tail latency. In figures 5.12 & 5.15 the difference has been plotted to ease in the analysis.

5.6.1 Conjunctive Queries

Conjunctive queries are hit the hardest compared to disjunctive. In figure 5.10 we can see that at the peak, there is a ~18632 different in total query throughput. Even in the smallest case of 5% heap size, we see ~7467 less queries per second. Without considering the impact that DAX vs buffered has on conjunctive queries, the values seen here substantially discredit then use of DAX with search. The performance penalty of doing so far outweighs the potential benefits that have been desired for it's use. However, it



(a) Difference in AND query throughput with and without DAX



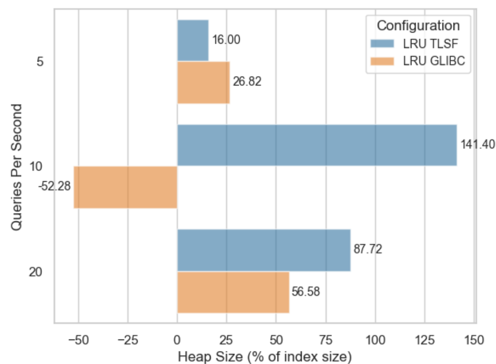
(b) Difference in AND query tail latency with and without DAX

s

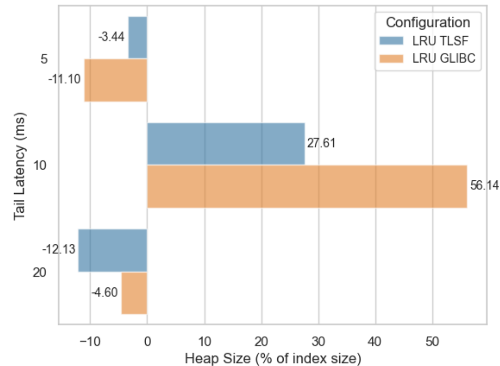
Figure 5.12: Impact of switching from DAX to buffered access with PMEM.

is important to acknowledge that our indexing size is only 1,000,000 and as such the value of this result should be considered in tadem with this. In future a more complete characterisation of this impact this has on larger indexes would be desirable. In doing so, one could either confirm the behaviour of DAX to be deterimental at all size contexts or alternatively provide a new path of research into the cause of what may the a threshold for the use of DAX.

5.6.2 Disjunctive Queries



(a) Difference in OR query throughput with and without DAX



(b) Difference in OR query tail latency with and without DAX

Figure 5.13: Impact of switching from DAX to buffered access with PMEM.

disjunctive queries fair a lot better than their counterparts. Noting that we see an increase of ~52 queries per second with the use of DAX in a 10% heap configuration. However,

given the state of all other configurations here, it is highly likely that is is an outlier, and should be discarded as such. In light of this, the impact of using DAX over buffered access does not benefit conjunctive queries. Although, comparatively, the peak difference is substantially less than that of Conjunctive, at ~141.

Analysing the tail latency difference suggests that there are benefits to be had with the use of DAX, reducing tail latency by ~12 ms at peak. Though, remarking on the inconsistency of these results, with regards to the throughput, would suggest that there is a significant basis of error in these measurements. We were unable to quantify precisely where this error originates from, which suggests that it is more intrinsic in the state of the entire software and hardware stack. As such the results for tail latency difference are not reliable enough to make a recommendation of usage for just the context of disjunctive queries.

5.6.3 TLB & Page Cache Behaviour

In order to further quantify the behaviour of page caching as an optimal configuration for PMEM with search indicies, we look to examine the behaviour of TLBs with virtual memory for these pages. Utilising `perf`, we capture page cache events for the non-DAX PMEM configurations via the `dTLB-load-misses` event.

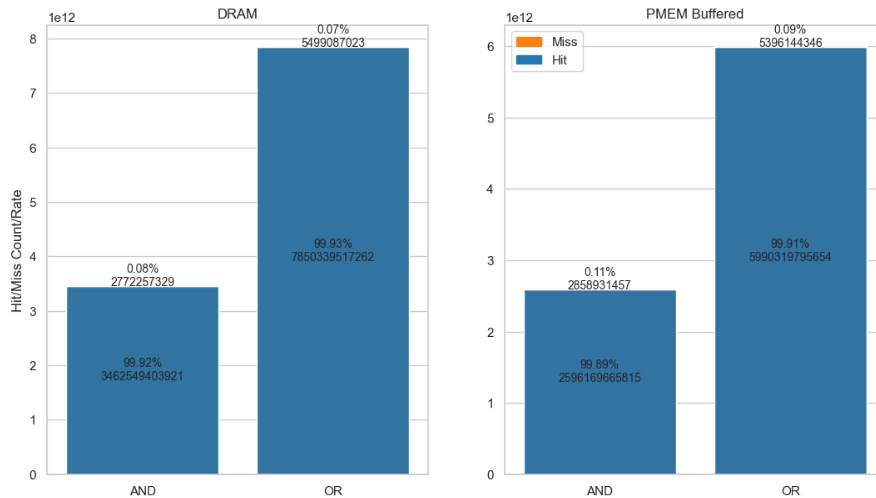
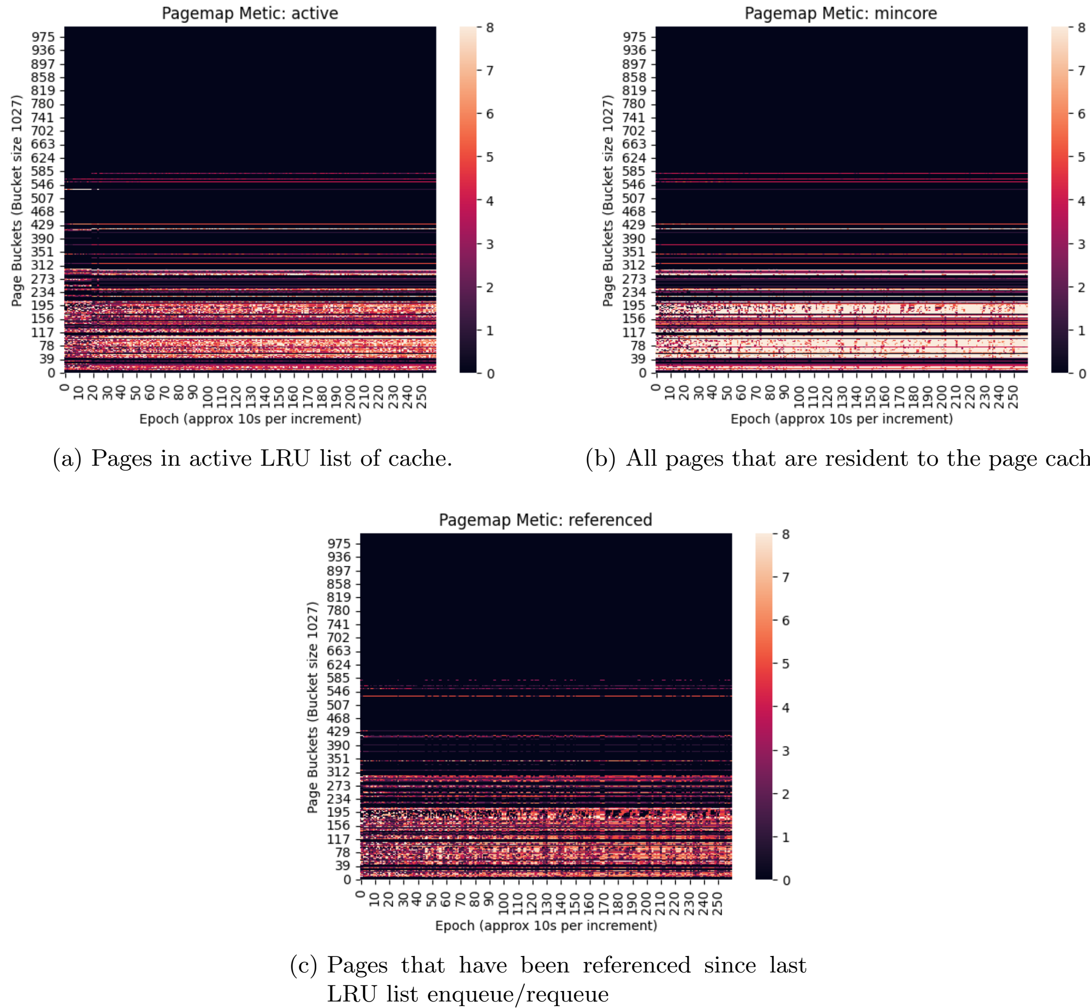


Figure 5.14: Data TLB hit/miss rates.

Analysing the figure 5.14, we can see that the hit rates suggest that. At first glance this may seen coincidental, however as verified in an earlier section (see figure 5.9), the posting list lengths are beyond regular page sizes. Most notably for disjunctive queries, posting lists reach into the range of huge pages (up to 1GB). As such the conclusion that can be drawn here is an extension to that of the page cache analysis, pages are effectively cached for interactions between PMEM and the query evaluators. However, this is limited by the index size, given that we have only 1,000,000 documents indexed,

this threshold can quickly be exceed. In a later section (6.3), We will discuss the potential for huge pages to be used in this context as an optimisation for larger query sets.

5.6.3.1 Page Temperature



(a) Pages in active LRU list of cache.

(b) All pages that are resident to the page cache

(c) Pages that have been referenced since last LRU list enqueue/requeue

Figure 5.15: Heatmaps of index file in page cache over the lifetime of the optimal conjunctive query set.

Examining the heatmaps of the page cache over the period of a conjunctive query set suggests that pages are well grouped together. Their global locality to the start of the file indicates that the start of the index contains the most relevant results for high terms (figure 5.15b). Interestingly, there is a clear transition period at the beginning with the active pages (figure 5.15a), where the caches are yet to fully warm and the caching behaviour is less regular. It appears to be within the first two iterations of the query set.

It is clear that as time goes on, the cache becomes more stable, with less fluctuation in the resident accesses (figure 5.15c), where the resident pages that are frequently referenced become finer grained in their residency.

By extension, we can say that the performance of user space caching has not impeded the page cache in any way, rather it seems to complement it well. Given that we had restricted the available memory to both buffer cache and processes, the memory mapped backings for allocators used the caches are also included in mapped pages. Given that there is no contention in the TLBs with regards to loads for either pages of origin in PMEM or allocator backing space, this indicates a coupling that should be exploited as much as possible in search systems.

5.7 Hardware Prefetching

During lookup of indexes, the processor will attempt to prefetch memory according to predictions made around the regularity and relation of previous memory accesses. This prefetching In particular, our system's Cascade Lake Xeon Gold 6252 uses 64 byte stride prefetching (Intel, a,b) with 4KB pages, through 64-entry 4-way Translation Look aside Buffers (TLBs). In this particular configuration, the prefetching behaviour is determined by stride. This is a measure of relative access distance between virtual memory addresses. Stride prefetching will not only look at a single dimension, such as an array, but also in more complex views of relation between separated chunks of memory. However, the latter is often algorithmically complex and does not lend well to being predicted. Figure 5.16 illustrates these behaviours in summary.

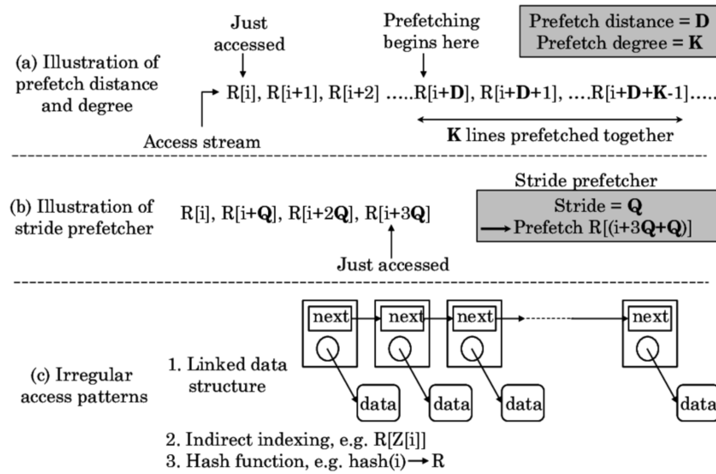


Figure 5.16: Behaviour of stride prefetcher in all distinct scenarios (Mittal, 2016)

5.7.1 L2 Prefetching

In the context of query processing, the most common cases for stride evaluation will be on elements within a posting list and the next posting list in a term evaluation. The CPU will attempt to prefetch the next section of a posting list based on the current behaviour and distance between consecutive elements in the list. In the second case, prefetching will be done for the next potential posting list based on previous previous non-matches for a particular term to posting lists.

Figure 5.17: L2 cache load prefetching hit ratios.

Between cores, the prefetching behaviour is specific to the search space for a particular term. This extends to the size of the data being brought into cache. As the size of the posting lists grow, such as disjunctive queries, the hit rate will improve. However, there is a threshold where cached data exceeds the cache capacity and begins to cause thrashing. As a result, the cache performance degrades with more frequent misses and pollution of the cache contents, leading to an unhealthy feedback cycle. The afflicted cache can recover if the data sets being requested fit within the cache once again. In doing so the polluted entries are evicted and hit rates return to higher values (Mittal, 2016). This relies on the polluting processes to better manage their data cycles.

Comparing the L2 load hit rates of figure 5.17 against the L3 laod hit rates in figure 5.11, we can see that despite the high volume of misses for prefetch, the follow up full loads through L3 have a much tamer miss rate. Quantitatively, this equates to about 2.98% of prefetch misses being full misses for disjunctive queries. Interestingly, despite the appearance of the miss rates, the combination of L2 and L3 misses, equates to roughly 2.40% of L2 prefetch misses for conjunctive queries.

In prior work by Monil et al., Cascade Lake processors exhibited inconsistent performance behaviours for that of intialised arrays with strides of 64 and above for prefetch-

ing. More specifically in our case, given the posting list irregularity of conjunctive and single term queries over disjunctive (see figure 5.9), the stride approximation is regularly broken leading to more hardware cache misses and full look ups to DRAM. Given that the posting lists exceed the cache sizes, this leads to the majority of look ups becoming bypasses (Mittal, 2016), essentially causing thrashing which degrades processor performance.

Chapter 6

Concluding Remarks

6.1 Hit Rates

Through our testing we have characterised the performance of caches in many aspects. One of particular note is the hit rates observed in each configuration. It is easy to see that the hit rates in all cases are less than desirable, reaching a peak of ~73%. This does not, however, signal the end of improvements, rather we see that our implementation and characterisation has limitations that are bounded by the constraints of time. Further development work and research is needed in order to refine and improve the caching models developed as part of this work. We see this as an opportunity for future research to, which would lend further to the viability of PMEM as a backing medium in search.

6.2 Inter-dependent Hybrid Type I/II Matching

In the evaluation chapter 5, we analysed the matching behaviours presented in the content chapter 3. Of particular note was not evaluating partial type II matching conditions as part of this body of work. However, is it not without its potential benefits. As mentioned there are avenues for research into progressive query processing with iterative or phased refinement of results.

In addition to this, there are other potential research points to analyse. One could restructure the cache entries to hold micro-inverted indexes. Whereby, the posting lists of similar (term wise) queries, could be merged such that on-entry look-ups could be performed. For example, such entries could map common overlaps between queries, where the operation or a single term differs. This would allow for optimisation in search behaviour that can intelligently restructure entries on a closely relational basis.

In prior, work by Feuerstein et al. proposed a 3D indexing strategy with replication across processors with intersection and results caches. Although not directly compatible

with the concept presented here, their separation of intersection caching as a standalone optimisation would be extended. More specifically, to be a posting list combination cache for similar queries, where topical and structural similarities of queries characterise merge operations between queries in context of their results. It should be noted that, a balance would need to be drawn between the sizes of these micro-inverted indexes and the quality of queries with respect to the cache and heap space required for them.

Recently, work by Trinh et al. looked at intelligent caching using mutual dependency between result and posting list caches. Their proposed algorithms explore exploiting redundancy between results and optimising entry utilisation based on current state of the cache. A proposed optimisation here would be to merge the ideas presented in these two areas of research with the historical data available with the DLIRS caching algorithm. In stating this, we identify this as an avenue of potential future research.

6.3 Huge Pages

In section 5.6.3, we briefly mentioned the usage of huge pages for search. In Linux kernels of version 2.6 and above a feature called huge pages is available (Linux, b,d). This allows the operating system to support page sizes greater than the standard size of 4KB. Huge pages are between 2MB and 256MB in size, depending on the kernel and hardware support. For search contexts, huge pages provide the means to wrangle data sets with large memory footprint while reducing the impact on caches and TLB lookups.

More specifically, queries that return long posting lists that span significant portions of memory, can be brought into working memory with relatively less TLB misses for virtual address translations. This reduces the impact of the average cost in cycles for TLB misses, given regular page sizes would need to be swapped out more often to accommodate working with the large data set. As we noted previously in section 5.6.3, our test configurations were in the range of huge pages, but not at their maximum. However, as noted can easily become substantially larger. As such, evaluating the impact of huge pages in searching PMEM indexes would lend to a better characterisation in high performance contexts that were otherwise not possible to consider in this body of work. Examining its performance contribution in the presented result caching context could potentially provide additional performance benefits. We see this as a potential point for future research into the viability of huge pages in search contexts.

6.4 Conclusion

We have presented a characterisation of PMEM as the backing medium for indices in a fine-tuned search engine. Through our testing and analysis we see great potential for the use of PMEM in this context. During the exploring of behaviours exhibited by the search engine, we have characterised the impacts beyond the immediate contexts and into the kernel and hardware layers. In our analysis we found that given sufficient consideration and fine-tuning, persistent memory can provide significant performance

benefits for search engines. Through our research, we have identified many potential avenues for future research and have laid the groundwork for more in-depth analysis of PMEM in high-performance environments.

Appendix A

Appendix: Cache Key Matching Conditions

A.1 Key Matching Conditions

A.1.1 Isomorphic Conditions

$$\exists e \in E \mid \begin{cases} \texttt{match_or_iso}(q,e) & q_{op} = \texttt{'|'} \ \texttt{\&\&} \ e_{op} = \texttt{'|'} \\ \texttt{match_and_iso}(q,e) & q_{op} = \texttt{'\^{}'} \ \texttt{\&\&} \ e_{op} = \texttt{'\^{}'} \\ \texttt{match_single_iso}(q,e) & q_{op} = \texttt{'0'} \ \texttt{\&\&} \ e_{op} = \texttt{'0'} \\ \texttt{false} & \textit{otherwise} \end{cases} \tag{A.1}$$

$$\texttt{match_or_iso}(q,e) = \begin{cases} q_{t1} = e_{t1} \ \texttt{\&\&} \ q_{t2} = e_{t2} \\ q_{t1} = e_{t2} \ \texttt{\&\&} \ q_{t2} = e_{t1} \\ \texttt{false} & \textit{otherwise} \end{cases} \tag{A.2}$$

$$\texttt{match_and_iso}(q,e) = \begin{cases} q_{t1} = e_{t1} \ \texttt{\&\&} \ q_{t2} = e_{t2} \\ q_{t1} = e_{t2} \ \texttt{\&\&} \ q_{t2} = e_{t1} \\ \texttt{false} & \textit{otherwise} \end{cases} \tag{A.3}$$

$$\texttt{match_single_iso}(q,e) = \begin{cases} q_{t1} = e_{t1} \\ q_{t1} = e_{t2} \\ \texttt{false} & \textit{otherwise} \end{cases} \tag{A.4}$$

A.1.2 Partial Type I

$$\exists e \in E \mid \begin{cases} \texttt{match_or_iso}(q,e) & q_{op} = \texttt{'|'} \ \texttt{\&\&} \ e_{op} = \texttt{'|'} \\ \texttt{match_and_iso}(q,e) & q_{op} = \texttt{'\^{}'} \ \texttt{\&\&} \ e_{op} = \texttt{'\^{}'} \\ \texttt{match_single_partial1}(q,e) & q_{op} = \texttt{'0'} \ \texttt{\&\&} \ e_{op} \neq \texttt{'\^{}'} \\ \texttt{false} & \textit{otherwise} \end{cases} \tag{A.5}$$

$$\texttt{match_single_partial1}(q,e) = \begin{cases} \texttt{match_single_iso}(q,e) & e_{op} = \texttt{'0'} \\ q_{t1} = e_{t1} & e_{\textit{term1Found}} \\ q_{t1} = e_{t2} & e_{\textit{term2Found}} \\ \texttt{false} & \textit{otherwise} \end{cases} \tag{A.6}$$

A.1.3 Partial Type II

$$\exists e \in E \mid \begin{cases} \texttt{match_or_partial2}(q,e) & q_{op} = \texttt{'|'} \\ \texttt{match_and_partial1}(q,e) & q_{op} = \texttt{'\^{}'} \\ \texttt{match_single_partial2}(q,e) & q_{op} = \texttt{'0'} \\ \texttt{false} & \textit{otherwise} \end{cases} \tag{A.7}$$

$$\texttt{match_or_partial2}(q,e) = \begin{cases} q_{t1} = e_{t1} \ \texttt{||} \ q_{t1} = e_{t2} & q_{t2} == \texttt{NULL} \ \texttt{\&\&} \ \texttt{'\^{}'} \\ q_{t2} = e_{t1} \ \texttt{||} \ q_{t2} = e_{t2} & q_{t1} == \texttt{NULL} \ \texttt{\&\&} \ \texttt{'\^{}'} \\ q_{t1} = e_{t1} \ \texttt{||} \ q_{t2} = e_{t1} & e_{op} = \texttt{'0'} \\ q_{t1} = e_{t1} \ \texttt{||} \ q_{t2} = e_{t1} & e_{\textit{term1Found}} \\ q_{t1} = e_{t2} \ \texttt{||} \ q_{t2} = e_{t2} & e_{\textit{term2Found}} \\ \texttt{match_or_iso}(q,e) & \textit{otherwise} \end{cases} \tag{A.8}$$

$$\texttt{match_single_partial2}(q,e) = \begin{cases} \texttt{match_single_partial1}(q,e) & e_{op} \neq \texttt{'\^{}'} \\ q_{t1} = e_{t1} \ \texttt{||} \ q_{t1} = e_{t2} & \textit{otherwise} \end{cases} \tag{A.9}$$

Bibliography

Akram, S., 2021. Exploiting intel optane persistent memory for full text search. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2021 (Virtual, Canada, 2021), 80–93. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3459898.3463906. https://doi.org/10.1145/3459898.3463906. [Cited on page 2.]

Baeza-Yates, R. and Jonassen, S., 2012. Modeling static caching in web search engines. In *Advances in Information Retrieval*, 436–446. Springer Berlin Heidelberg, Berlin, Heidelberg. [Cited on page 8.]

Berger, E. D.; Zorn, B. G.; and McKinley, K. S., 2001. Composing high-performance memory allocators. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01 (Snowbird, Utah, USA, 2001), 114–124. Association for Computing Machinery, New York, NY, USA. doi:10.1145/378795.378821. https://doi.org/10.1145/378795.378821. [Cited on page 9.]

Bilal, M. and Kang, S.-G., 2014. Time aware least recent used (tlru) cache management policy in icn. In *16th International Conference on Advanced Communication Technology*, 528–532. doi:10.1109/ICACT.2014.6779016. [Cited on page 17.]

Boyd-Wickizer, S.; T. Clements, A.; Mao, Y.; Pesterev, A.; Kaashoek, M. F.; Morris, R.; and Zeldovich, N. Mosbench suite. https://pdos.csail.mit.edu/archive/mosbench/. [Cited on page 7.]

Brent, R., 1989. Efficient implementation of a first-fit strategy for dynamic storage allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11 (07 1989), 388–403. doi:10.1145/65979.65981. [Cited on page 13.]

Elasticsearch. https://www.elastic.co/. [Cited on pages 6 and 8.]

EnWiki. https://dumps.wikimedia.org/enwiki/. [Cited on page 11.]

Feuerstein, E.; Gil-Costa, V.; Marin, M.; Tolosa, G.; and Baeza-Yates, R., 2012. 3d inverted index with cache sharing for web search engines. In *Proceedings of*

the 18th International Conference on Parallel Processing, Euro-Par'12 (Rhodes Island, Greece, 2012), 272–284. Springer-Verlag, Berlin, Heidelberg. doi:10.1007/978-3-642-32820-6_28. https://doi.org/10.1007/978-3-642-32820-6_28. [Cited on page 47.]

Giles, C. L.; Bollacker, K. D.; and Lawrence, S., 1998. Citeseer: An automatic citation indexing system. In *Proceedings of the Third ACM Conference on Digital Libraries*, DL '98 (Pittsburgh, Pennsylvania, USA, 1998), 89–98. Association for Computing Machinery, New York, NY, USA. doi:10.1145/276675.276685. https://doi.org/10.1145/276675.276685. [Cited on page 7.]

Gloger, W., 2006. Wolfram gloger's malloc homepage. http://www.malloc.de/. [Cited on page 14.]

Gortmaker, P., 1995. Using the ram disk block device with linux. https://www.kernel.org/doc/html/latest/admin-guide/blockdev/ramdisk.html. [Cited on page 27.]

Hoyte, D., 2009. Vmtouch - the virtual memory toucher. https://hoytech.com/vmtouch/. [Cited on page 26.]

Intel, a. https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html. [Cited on page 44.]

Intel, b. https://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%206252.html. [Cited on page 44.]

Intel, c. Intel® optane™ persistent memory product brief. https://www.intel.com.au/content/www/au/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html. [Cited on pages 2 and 10.]

Jain, A. and Lin, C., 2016. Back to the future: Leveraging belady's algorithm for improved cache replacement. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 78–89. doi:10.1109/ISCA.2016.17. [Cited on page 8.]

Jiang, S. and Zhang, X., 2002. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *SIGMETRICS Perform. Eval. Rev.*, 30, 1 (jun 2002), 31–42. doi:10.1145/511399.511340. https://doi-org.virtual.anu.edu.au/10.1145/511399.511340. [Cited on page 8.]

Johnson, T. and Shasha, D., 1994. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, 439–450. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. [Cited on pages 9 and 17.]

Johnstone, M. S. and Wilson, P. R., 1998. The memory fragmentation problem: Solved? In *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98 (Vancouver, British Columbia, Canada, 1998), 26–36. Association for Computing Machinery, New York, NY, USA. doi:10.1145/286860.286864. https://doi.org/10.1145/286860.286864. [Cited on page 9.]

Lea, D., 1996. A memory allocator. *unix/mail*, (12 1996). [Cited on pages 13 and 14.]

Li, C., 2018. Dlirs: Improving low inter-reference recency set cache replacement policy with dynamics. In *Proceedings of the 11th ACM International Systems and Storage Conference*, SYSTOR '18 (Haifa, Israel, 2018), 59–64. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3211890.3211891. https://doi-org.virtual.anu.edu.au/10.1145/3211890.3211891. [Cited on pages 9, 17, 18, and 24.]

Linux, a. Examining process page tables. https://www.kernel.org/doc/html/v4.18/admin-guide/mm/pagemap.html. [Cited on page 27.]

Linux, b. Linux huge tlb pages. https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt. [Cited on page 48.]

Linux, c. proc(5) — linux manual page. https://man7.org/linux/man-pages/man5/proc.5.html. [Cited on page 27.]

Linux, R., d. 5.2.nbsp;huge pages and transparent huge pages red hat enterprise linux 6. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-memory-transhuge. [Cited on page 48.]

Long, X. and Suel, T., 2005. Three-level caching for efficient query processing in large web search engines. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05 (Chiba, Japan, 2005), 257–266. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1060745.1060785. https://doi.org/10.1145/1060745.1060785. [Cited on pages 3 and 8.]

Lucene, A. https://lucene.apache.org/. [Cited on pages 6 and 8.]

Magdy, A.; Mokbel, M. F.; Elnikety, S.; Nath, S.; and He, Y., 2014. Mercury: A memory-constrained spatio-temporal real-time search on microblogs. In *2014 IEEE 30th International Conference on Data Engineering*, 172–183. doi:10.1109/ICDE.2014.6816649. [Cited on page 7.]

Masmano, M.; Ripoll, I.; and Crespo, A., 2006. A comparison of memory allocators for real-time applications. 177 (01 2006), 68–76. doi:10.1145/1167999.1168012. [Cited on pages 13 and 16.]

Masmano, M.; Ripoll, I.; Crespo, A.; and Real, J., 2004. Tlsf: A new dynamic memory allocator for real-time systems. vol. 16, 79– 88. doi:10.1109/EMRTS.2004.1311009. [Cited on pages 13, 16, and 17.]

McCandless, M., 2012. https://github.com/mikemccand/luceneutil. [Cited on page 25.]

Mittal, S., 2016. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys*, 49 (08 2016). doi:10.1145/2907071. [Cited on pages 44, 45, and 46.]

Monil, M. A. H.; Lee, S.; Vetter, J. S.; and Malony, A. D., 2020. Understanding the impact of memory access patterns in intel processors. In *2020 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, 52–61. doi:10.1109/MCHPC51950.2020.00012. [Cited on page 45.]

Ogasawara, T., 1995. An algorithm with constant execution time for dynamic storage allocation. *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 0 (10 1995), 21. doi:10.1109/RTCSA.1995.528746. [Cited on page 13.]

O'Neil, E. J.; O'Neil, P. E.; and Weikum, G., 1993. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93 (Washington, D.C., USA, 1993), 297–306. Association for Computing Machinery, New York, NY, USA. doi:10.1145/170035.170081. https://doi.org/10.1145/170035.170081. [Cited on page 17.]

OpenSearch. https://opensearch.org/. [Cited on pages 6 and 8.]

Ozcan, R.; Altingövde, I.; Cambazoglu, B.; Junqueira, F.; and Ulusoy, , 2012. A five-level static cache architecture for web search engines. *Information Processing and Management - IPM*, 48 (09 2012). doi:10.1016/j.ipm.2010.12.007. [Cited on page 8.]

PMem.io. Persistent memory development kit. https://pmem.io/pmdk/. [Cited on page 10.]

Rodriguez, L. V.; Gonzalez, A.; Poudel, P.; Rangaswami, R.; and Liu, J., 2021. Unifying the data center caching layer: Feasible? profitable? In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '21 (Virtual, USA, 2021), 50–57. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3465332.3470884. https://doi.org/10.1145/3465332.3470884. [Cited on page 2.]

Rohland, C.; Dickins, H.; Motohiro, M.; and Down, C., 2001. Tmpfs. https://www.kernel.org/doc/html/latest/filesystems/tmpfs.html. [Cited on page 27.]

Saraiva, P. C.; Silva de Moura, E.; Ziviani, N.; Meira, W.; Fonseca, R.; and Ribeiro-Neto, B., 2001. Rank-preserving two-level caching for scalable search

engines. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '01 (New Orleans, Louisiana, USA, 2001), 51–58. Association for Computing Machinery, New York, NY, USA. doi:10.1145/383952.383959. https://doi.org/10.1145/383952.383959. [Cited on pages 3 and 8.]

Shan, Y.; Tsai, S.-Y.; and Zhang, Y., 2017. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17 (Santa Clara, California, 2017), 323–337. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3127479.3128610. https://doi.org/10.1145/3127479.3128610. [Cited on page 2.]

Solr, A. https://solr.apache.org/. [Cited on pages 6 and 8.]

Stribling, J.; Li, J.; Councill, I.; Kaashoek, M.; and Morris, R., 2006. Overcite: A distributed, cooperative citeseer. [Cited on page 7.]

Tolosa, G.; Becchetti, L.; Feuerstein, E.; and Marchetti-Spaccamela, A., 2014. Performance improvements for search systems using an integrated cache of lists+intersections. In *String Processing and Information Retrieval*, 227–235. Springer International Publishing, Cham. [Cited on page 19.]

Trinh, T.; Wu, D.; and Huang, J., 2017. A new static web caching mechanism based on mutual dependency between result cache and posting list cache. 148–156. doi:10.1007/978-3-319-68786-5_12. [Cited on pages 19 and 48.]

Tsai, S.-Y.; Shan, Y.; and Zhang, Y., 2020. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *USENIX Annual Technical Conference*. [Cited on page 2.]

Vo, A. N. and Moffat, A., 1998. Compressed inverted files with reduced decoding overheads. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, 290–297. [Cited on page 6.]

Wang, J.; Lo, E.; Yiu, M. L.; Tong, J.; Wang, G.; and Liu, X., 2013. The impact of solid state drive on search engine cache management. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13 (Dublin, Ireland, 2013), 693–702. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2484028.2484046. https://doi.org/10.1145/2484028.2484046. [Cited on page 1.]

Wang, Y. and Lin, J., 2015. The feasibility of brute force scans for real-time tweet search. In *Proceedings of the 2015 International Conference on The Theory of Information Retrieval*, ICTIR '15 (Northampton, Massachusetts, USA, 2015), 321–324. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2808194.2809489. https://doi.org/10.1145/2808194.2809489. [Cited on page 7.]

Wikimedia. Wikimedia downloads. https://dumps.wikimedia.org/. [Cited on page 11.]

Xiang, L.; Zhao, X.; Rao, J.; Jiang, S.; and Jiang, H., 2022a. Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22 (Rennes, France, 2022), 495. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3492321.3519556. https://doi.org/10.1145/3492321.3519556. [Cited on page 2.]

Xiang, L.; Zhao, X.; Rao, J.; Jiang, S.; and Jiang, H., 2022b. Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22 (Rennes, France, 2022), 490. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3492321.3519556. https://doi.org/10.1145/3492321.3519556. [Cited on page 10.]

Yang, J.; Li, B.; and Lilja, D. J., 2020. Exploring performance characteristics of the optane 3d xpoint storage technology. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 5, 1 (feb 2020). doi:10.1145/3372783. https://doi.org/10.1145/3372783. [Cited on page 10.]

Zhang, J.; Long, X.; and Suel, T., 2008. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08 (Beijing, China, 2008), 387–396. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1367497.1367550. https://doi.org/10.1145/1367497.1367550. [Cited on page 1.]

Zobel, J. and Moffat, A., 2006a. Inverted files for text search engines. *ACM Comput. Surv.*, 38, 2 (jul 2006), 6–es. doi:10.1145/1132956.1132959. https://doi.org/10.1145/1132956.1132959. [Cited on pages 1 and 6.]

Zobel, J. and Moffat, A., 2006b. Inverted files for text search engines. *ACM Comput. Surv.*, 38, 2 (jul 2006), 17. doi:10.1145/1132956.1132959. https://doi.org/10.1145/1132956.1132959. [Cited on page 1.]