Australian National University

Analyzing and Improving the Scalability of In-Memory Indices for Managed Search Engines

Aditya (Adi) Chilukuri
aditya.chilukuri@anu.edu.au

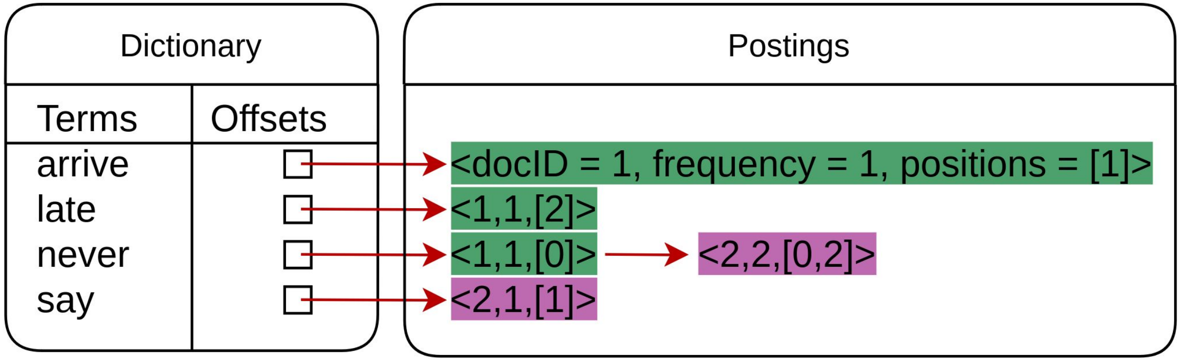
Shoaib Akram
shoaib.akram@anu.edu.au

Full text search is ubiquitous

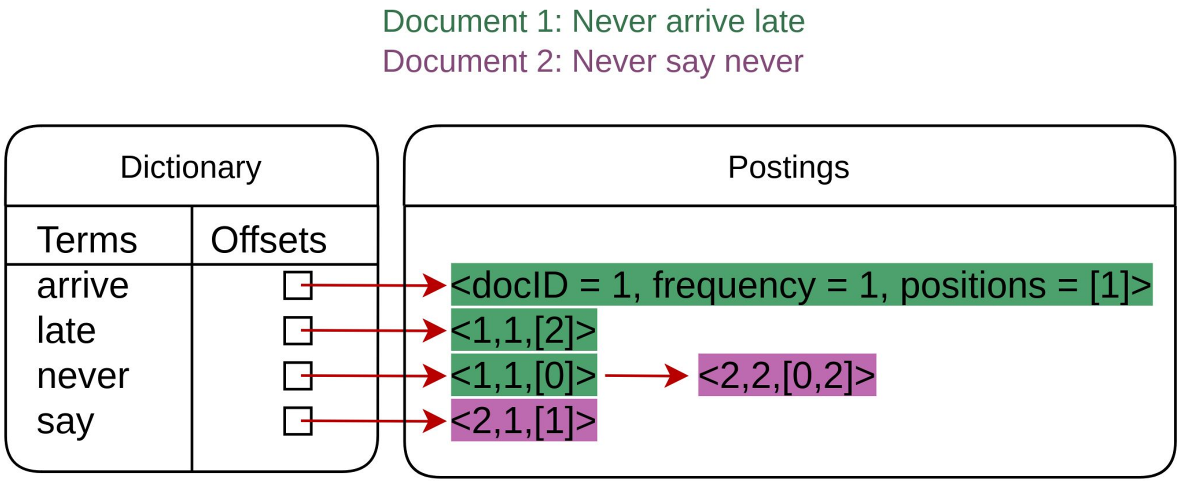
- Serve a large and impatient user base
- Tail latency impacts profit & loss
- **Goal:** High throughput and low response time

Inverted indices power search

Document 1: Never arrive late
Document 2: Never say never



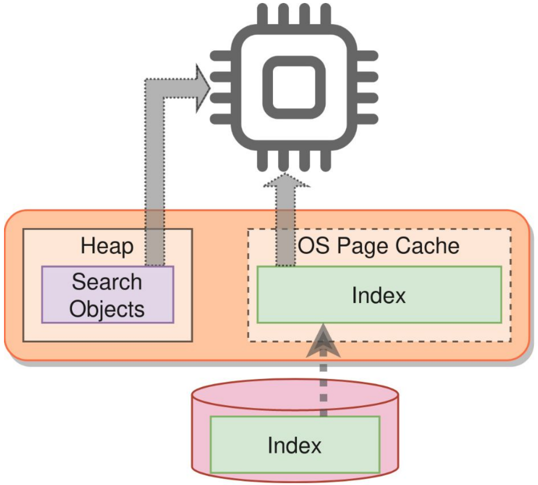
Inverted indices are outgrowing memory capacity



- Index is placed in DRAM (**fastest** storage resource available today)
- As datasets grow, indices grow proportionally
  - **Problem:** DRAM capacity is limited
  - **Problem:** Scalable devices (SSDs) have high latency

Index is typically placed in the page cache

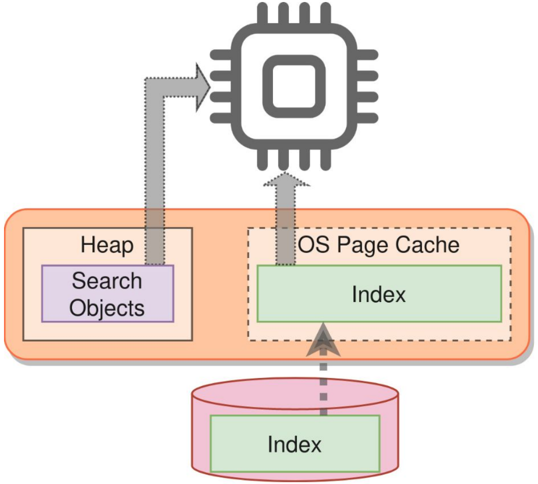
- Indices are archived on disks/SSDs
- Read index to DRAM to serve queries
- In a managed (**Java**) runtime, there are two options



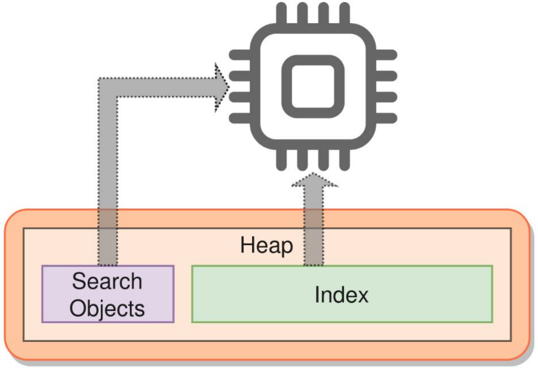
Page cache (**unsafe** accesses, typical)

Index is typically placed in the page cache

- Indices are archived on disks/SSDs
- Read index to DRAM to serve queries
- In a managed (**Java**) runtime, there are two options

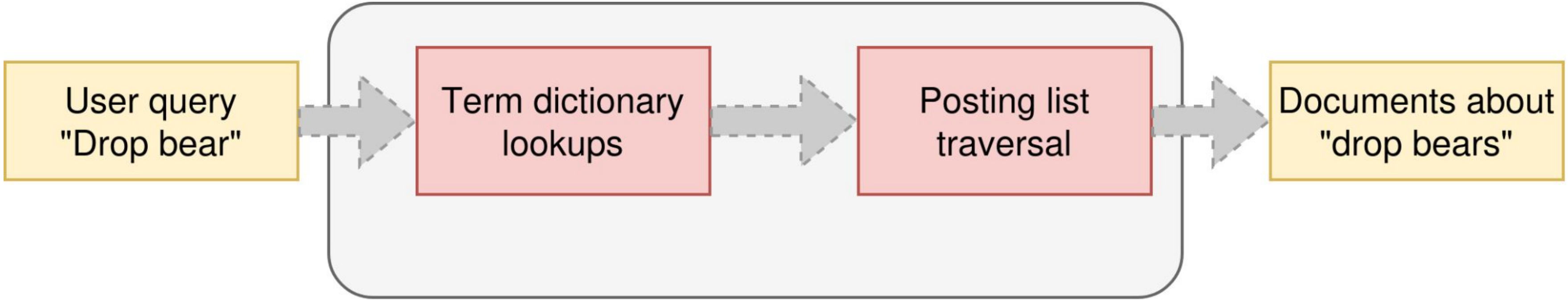


Page cache (**unsafe** accesses, typical)



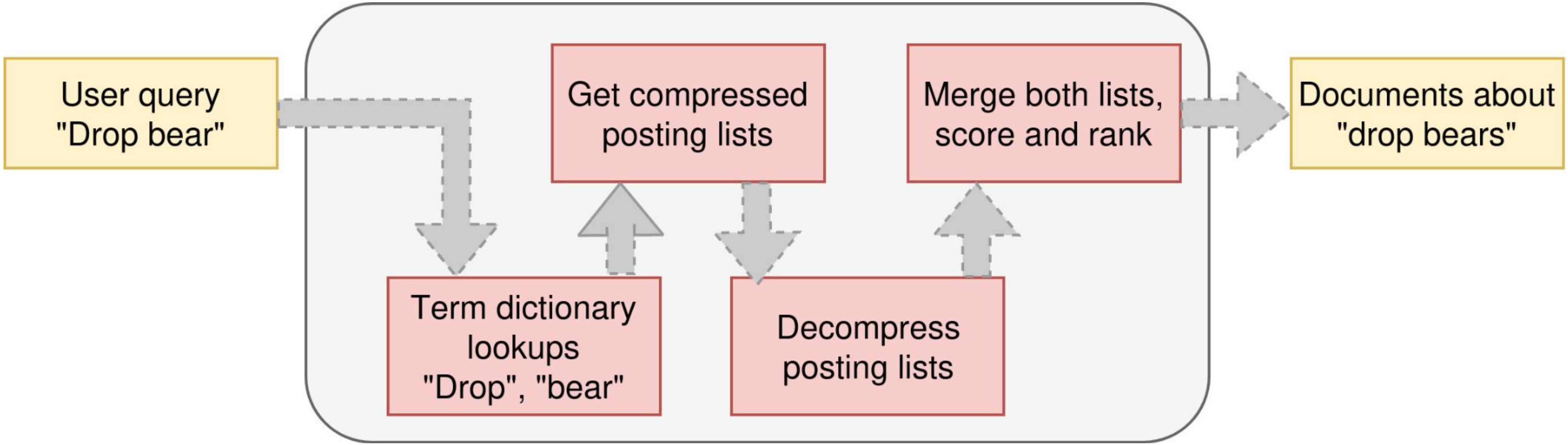
Managed heap (**GC** cost, avoided today)

Behaviour of query evaluation



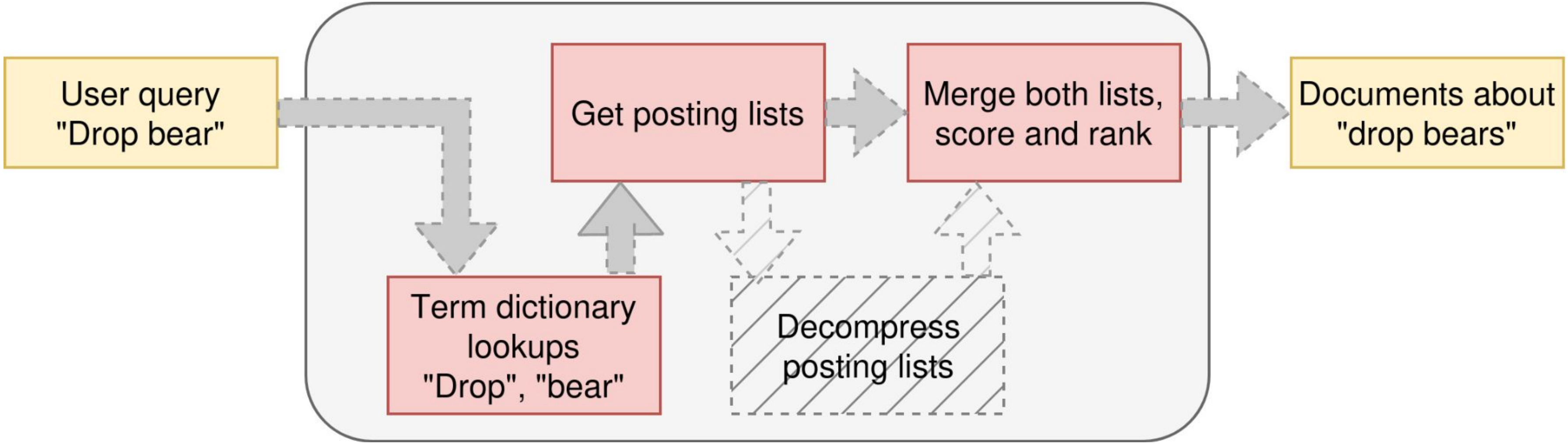
- Dictionary lookup is fast
- Posting traversal is slow (**especially for popular queries**)
- Postings traversal: **sequential** access pattern
- Posting lists are variable-sized (depends on term frequency)

Compression saves storage space but increases query latency



- Compress search indices to save space
- Decompress “on demand”
- Decompressing in-memory postings incurs a cost!

Let’s use an uncompressed search index

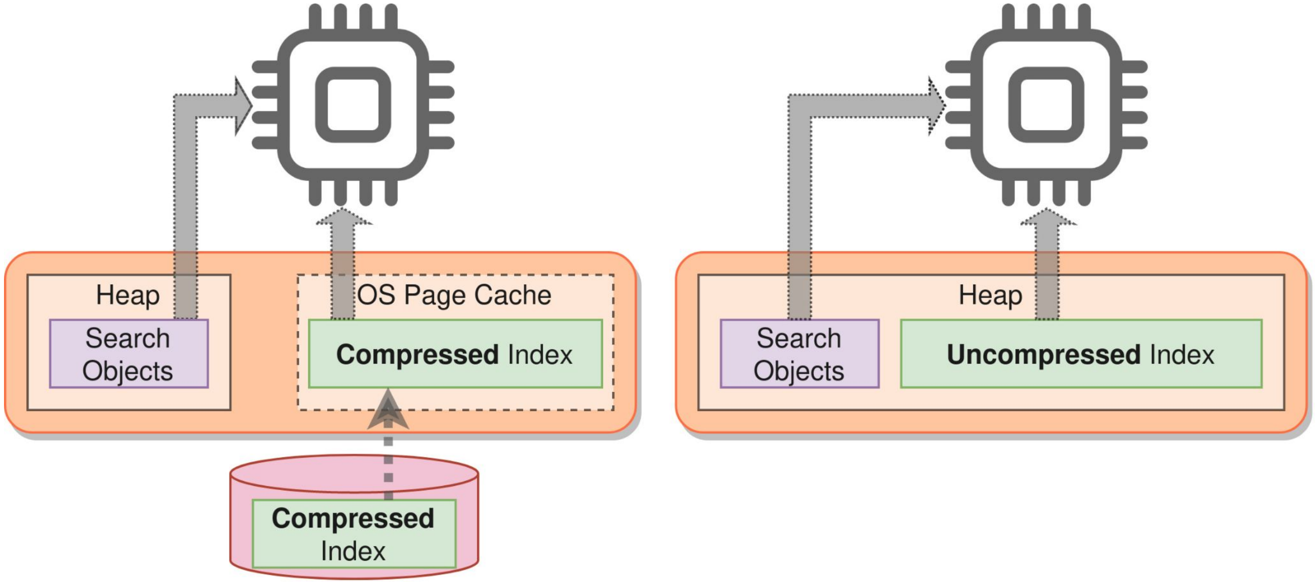


- Potential speed-up?
- Pressure on memory?

Baseline and proposed systems using DRAM

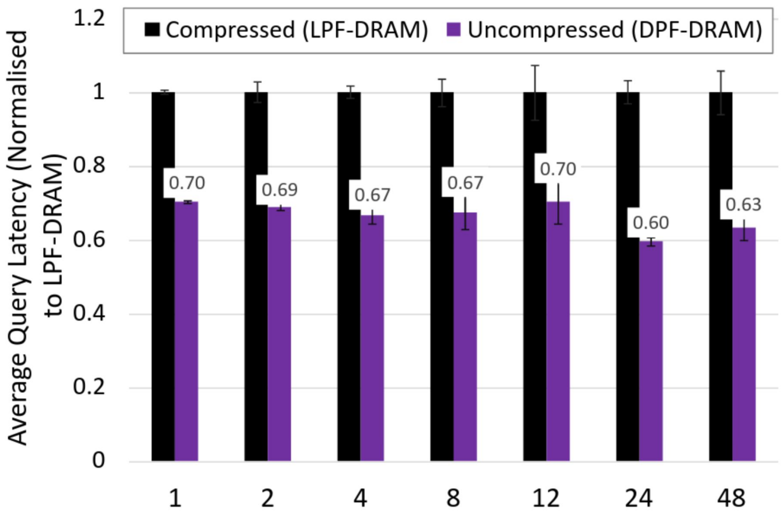
Baseline: Lucene Postings Format on DRAM (**LPF-DRAM**)

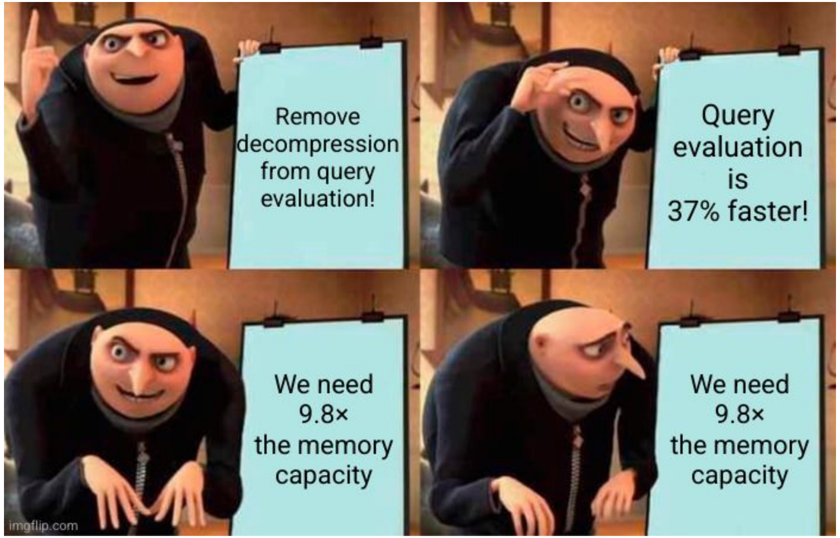
Proposed: Direct Postings Format on DRAM (**DPF-DRAM**)



- Use Apache Lucene (Java search engine library)
- Use existing code from the Lucene project

Search is 37% faster over an uncompressed index





- Data is normalised
- Mismatch between compute power and memory bandwidth exists
- Capacity can only come from **scalable** memory

Dealing with limited memory capacity

- **Non-Volatile Memory (NVM)**
  - Most promising complement to DRAM to build a large physical address space
  - Intel Optane persistent memory (discontinued but technology still promising)

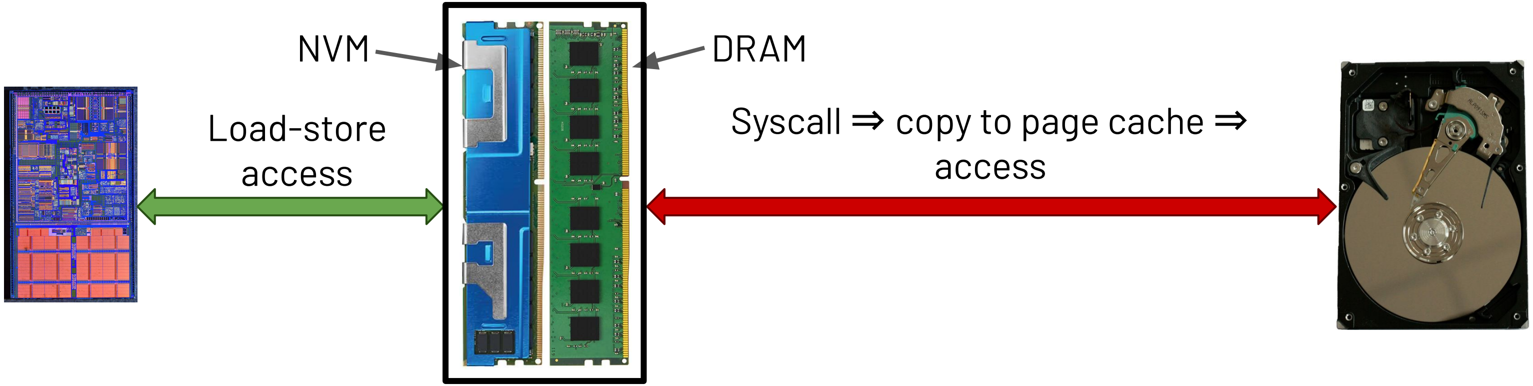
- **Other rapidly evolving options (promising but not focus of this work)**

  - Fast local storage (NVMe SSDs)

  - Remote disaggregated memory

Non-volatile Main Memory (NVM)

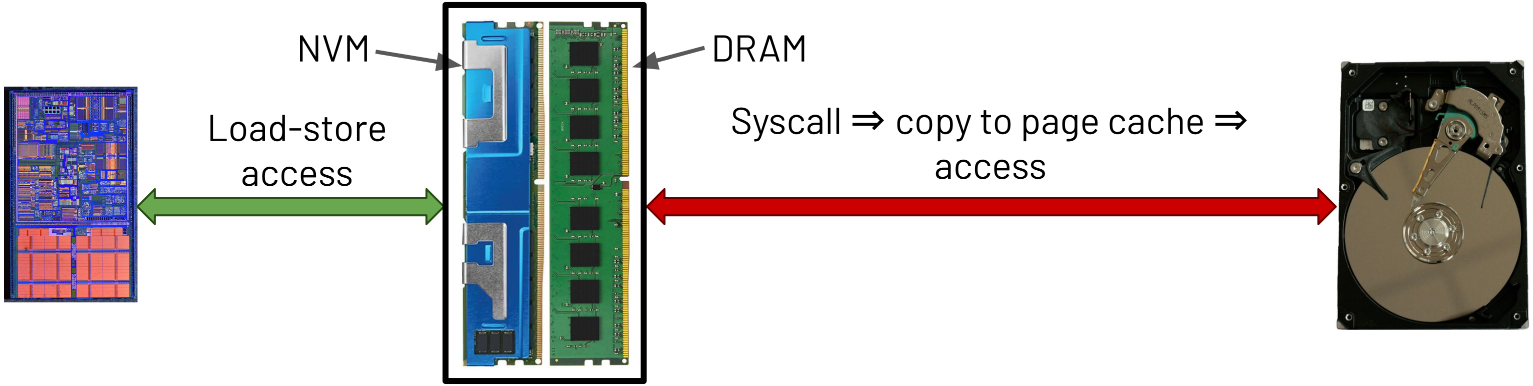
- Large capacity to complement DRAM
- Accessible on the (NV)DIMM interface
  - As a persistent storage device
  - As extension to DRAM
- Capacities/DIMM can scale up to many times DRAM DIMMs if technology follows the DRAM/SSD roadmap



Non-volatile Main Memory (NVM)

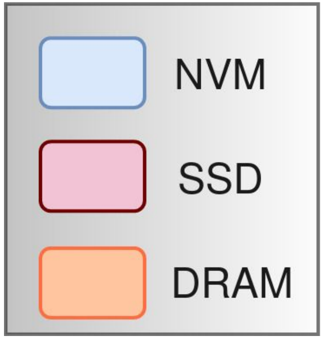
- Large capacity to complement DRAM
- Accessible on the (NV)DIMM interface
  - As a persistent storage device
  - As extension to DRAM
- Capacities/DIMM can scale up to many times DRAM DIMMs if technology follows the DRAM/SSD roadmap

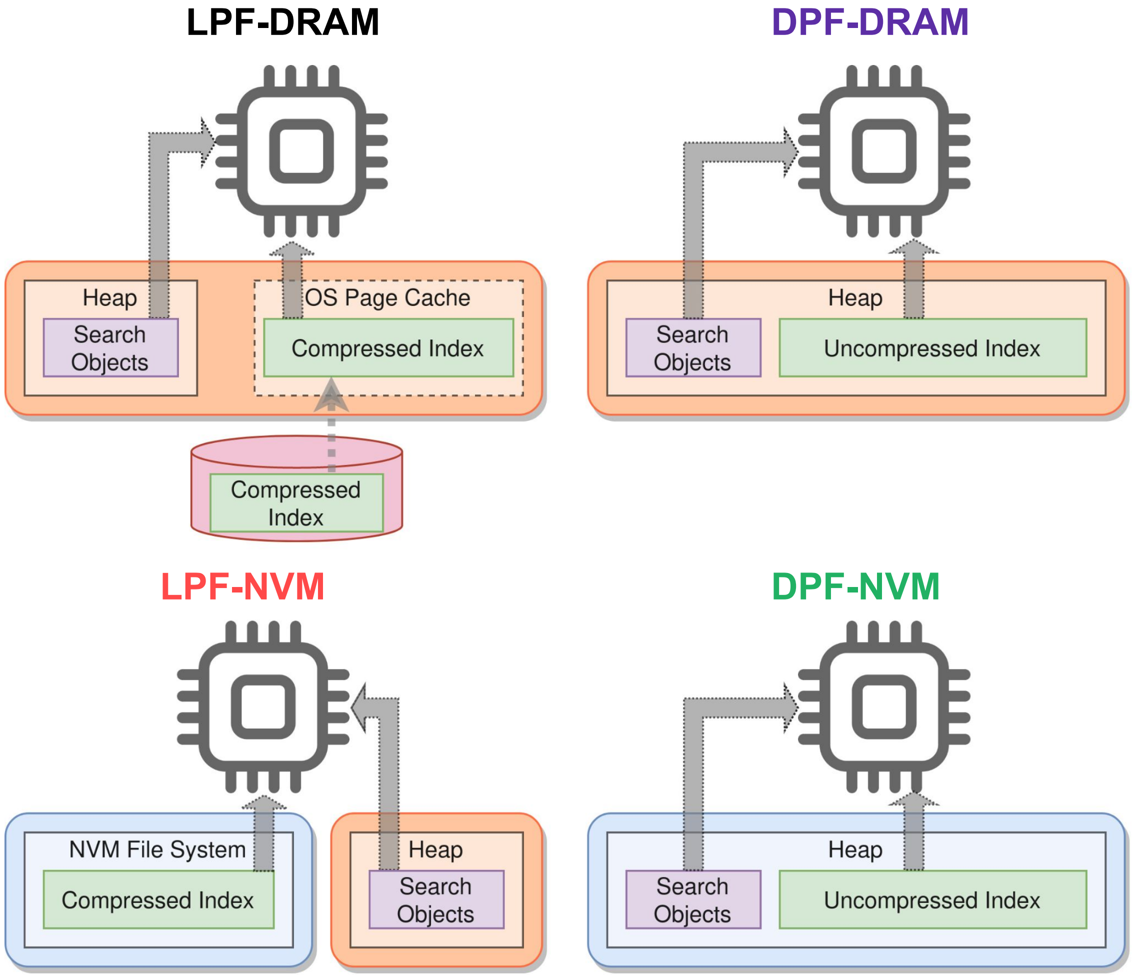
Microbenchmarks:
2-3x slower reads



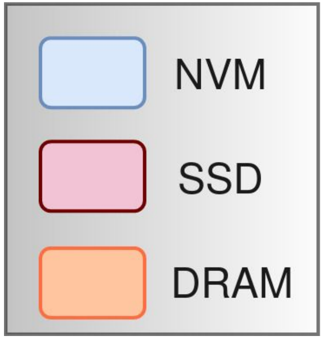
Can we place an **uncompressed** index on **NVM** and gain a similar speedup over a **compressed** index in **DRAM**?

Design space

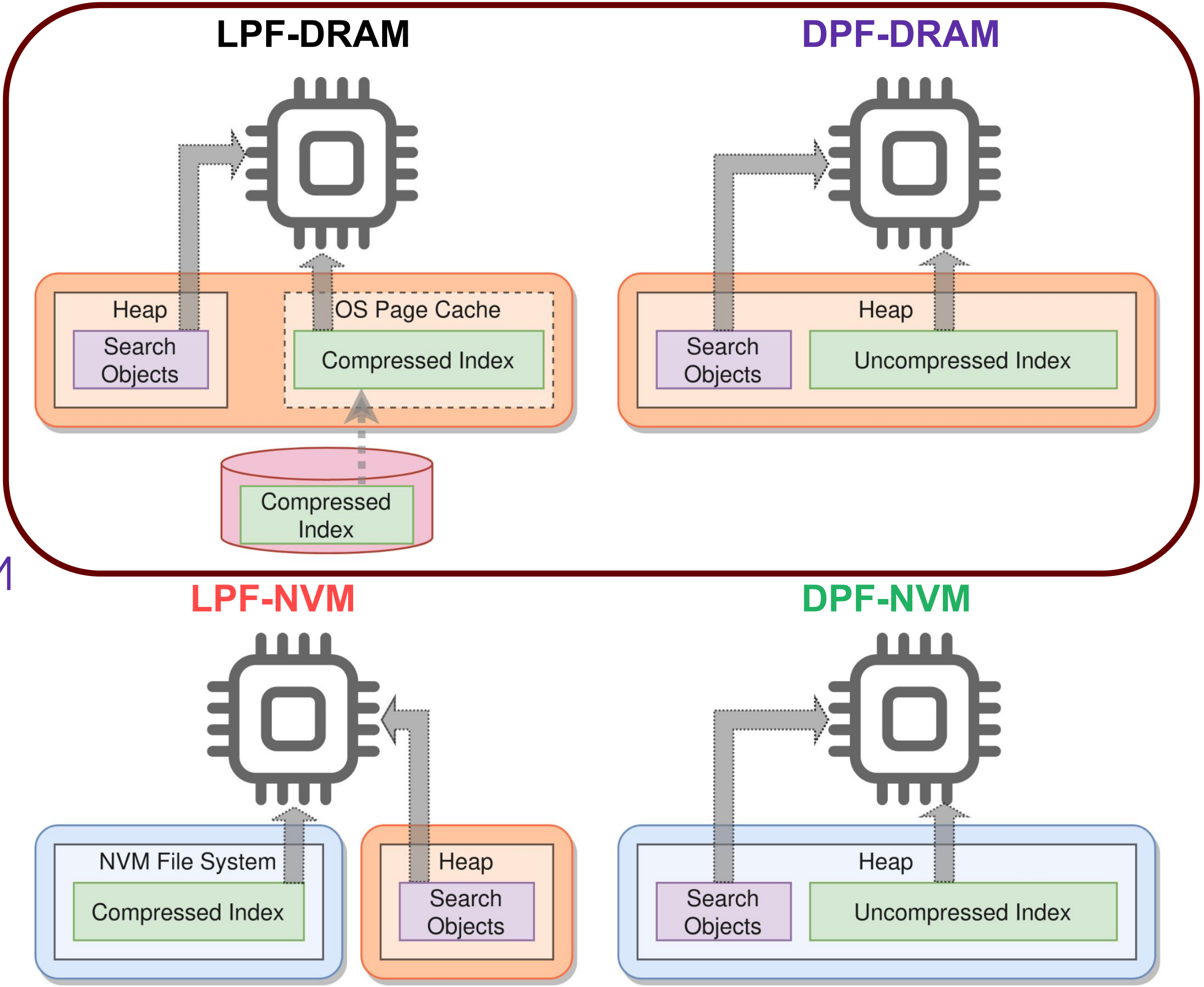




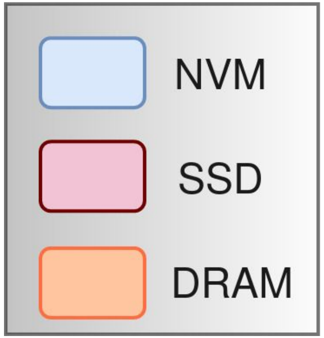
Design space



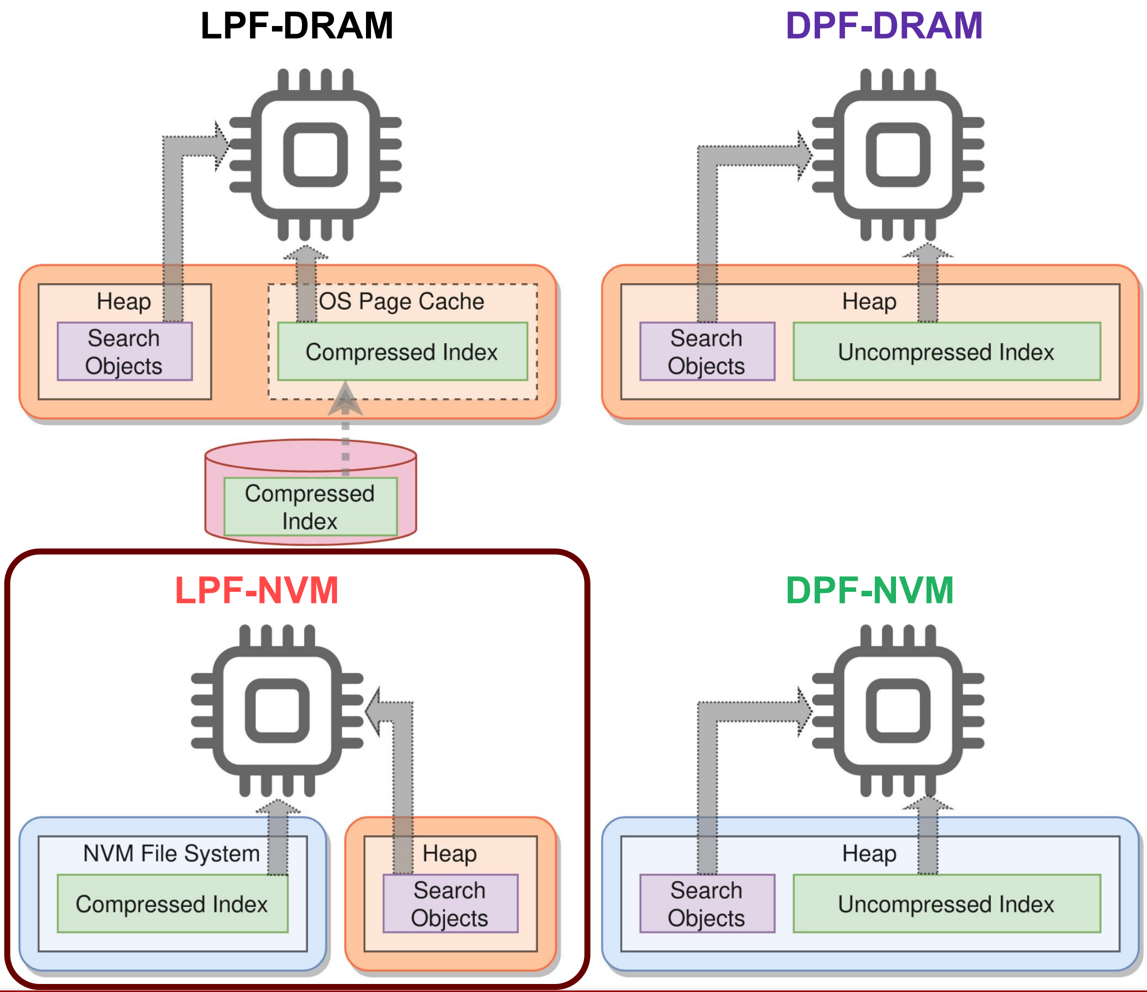
- LPF-DRAM and DPF-DRAM same as before.



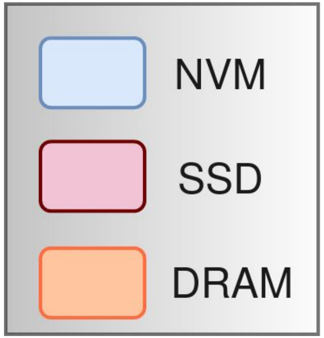
Design space

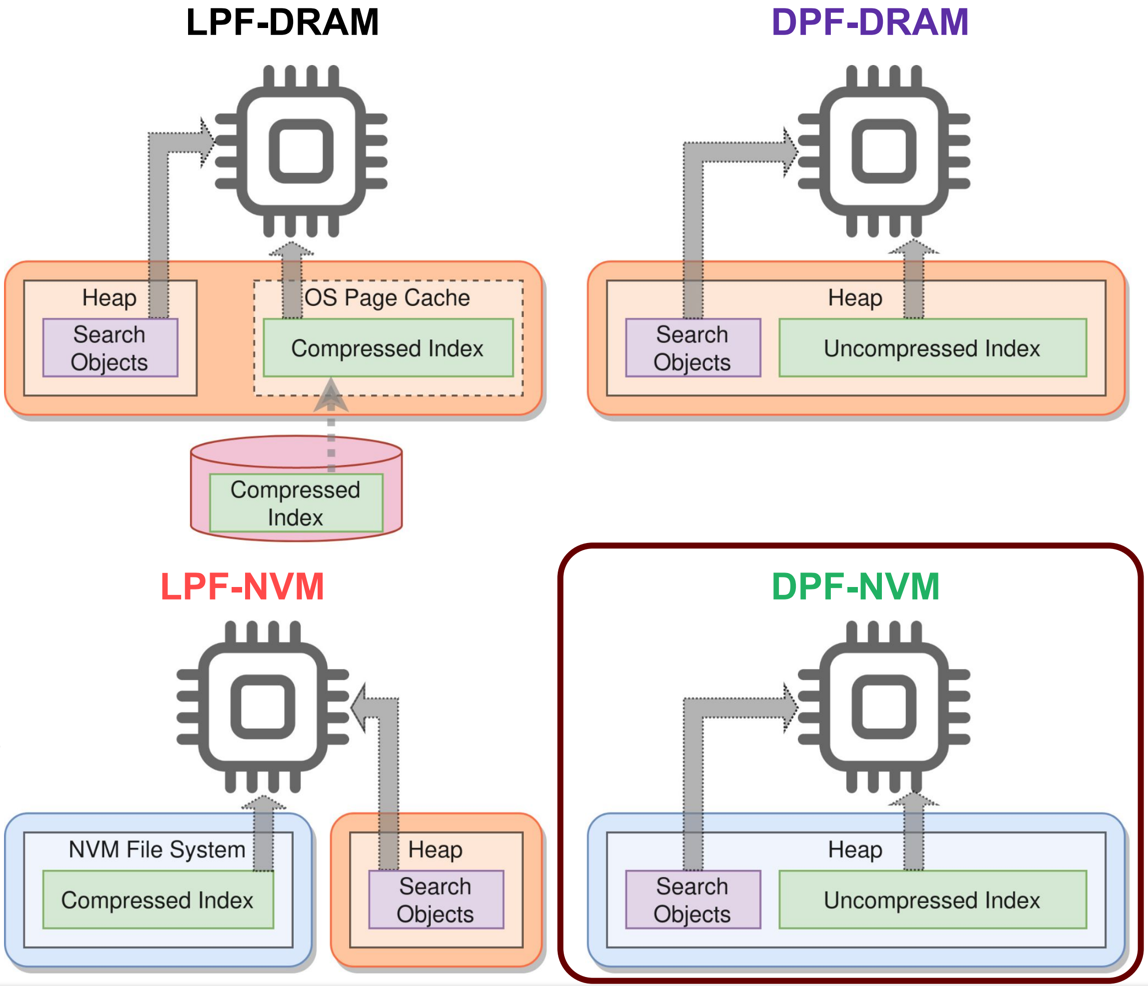


- LPF-DRAM and DPF-DRAM same as before.
- LPF-NVM places index on NVM file system, no OS page cache.



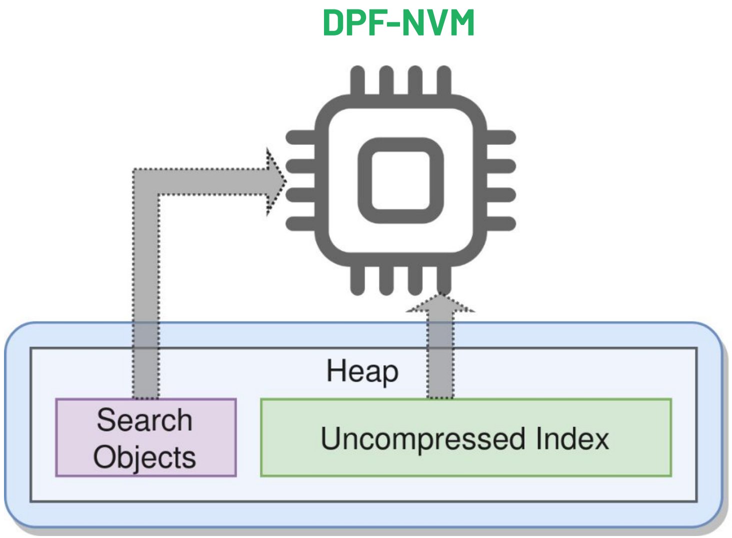
Design space





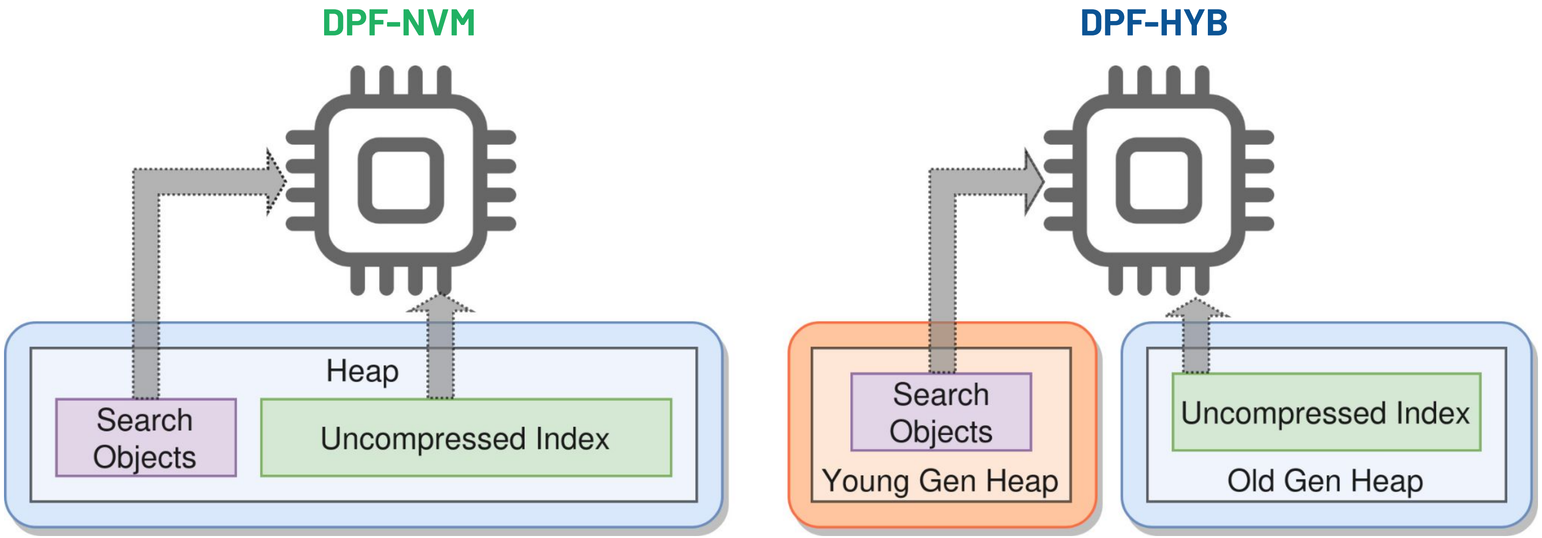
- LPF-DRAM and DPF-DRAM same as before
- LPF-NVM places index on DAX NVM file system (no OS page cache)
- DPF-NVM mmap Java heap to NVM

Hybrid DRAM-NVM setup



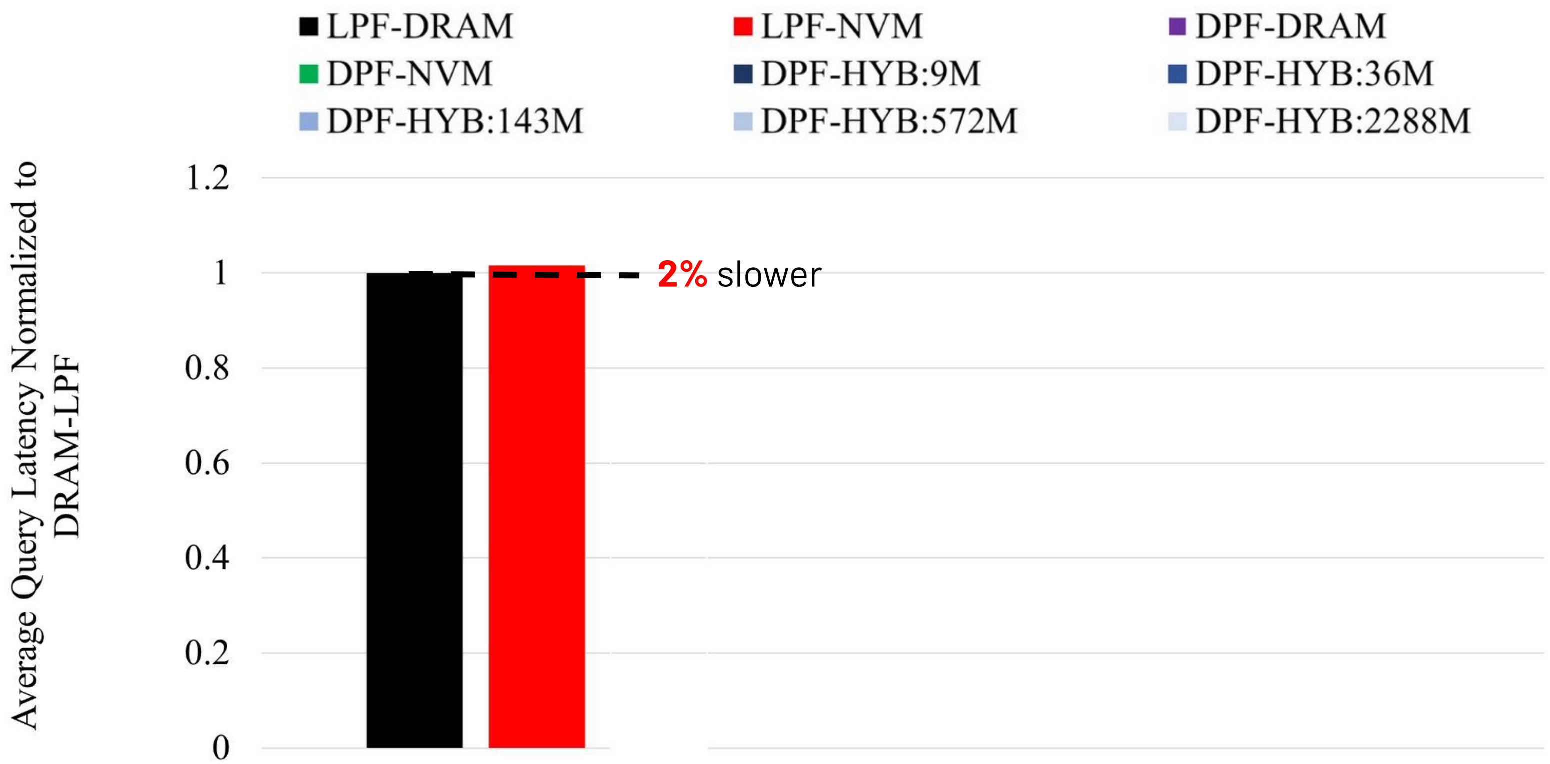
- DPF-NVM places search objects on NVM **(unnecessary slowdown)**

Hybrid DRAM-NVM setup

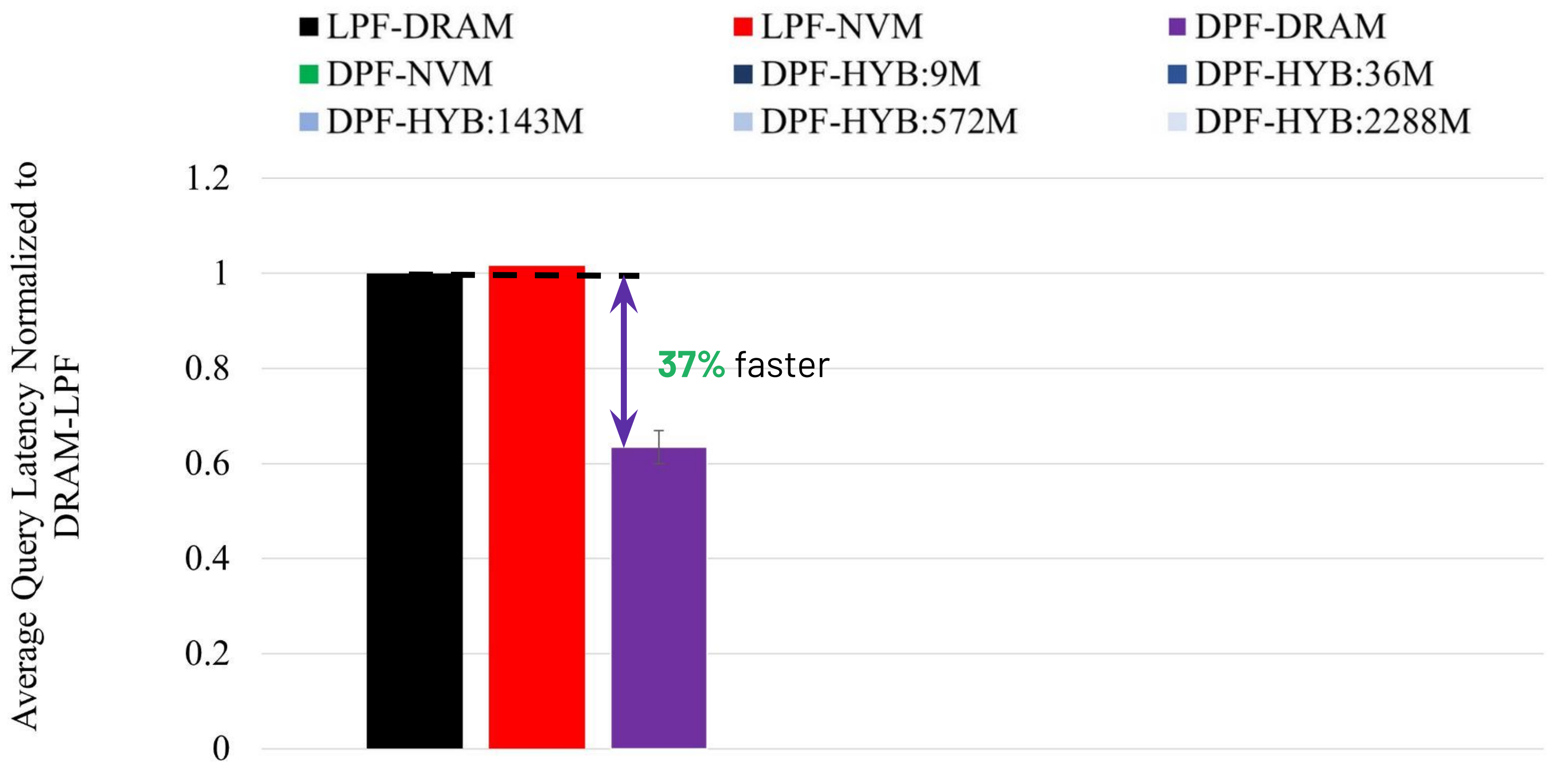


- DPF-NVM places search objects in NVM **(unnecessary slowdown)**
- **DPF-HYB:** place young generation in DRAM
- Ensure index is moved to old gen during setup
- Sensitivity analysis of young generation size
- Maximal DRAM use ≈ 2GB

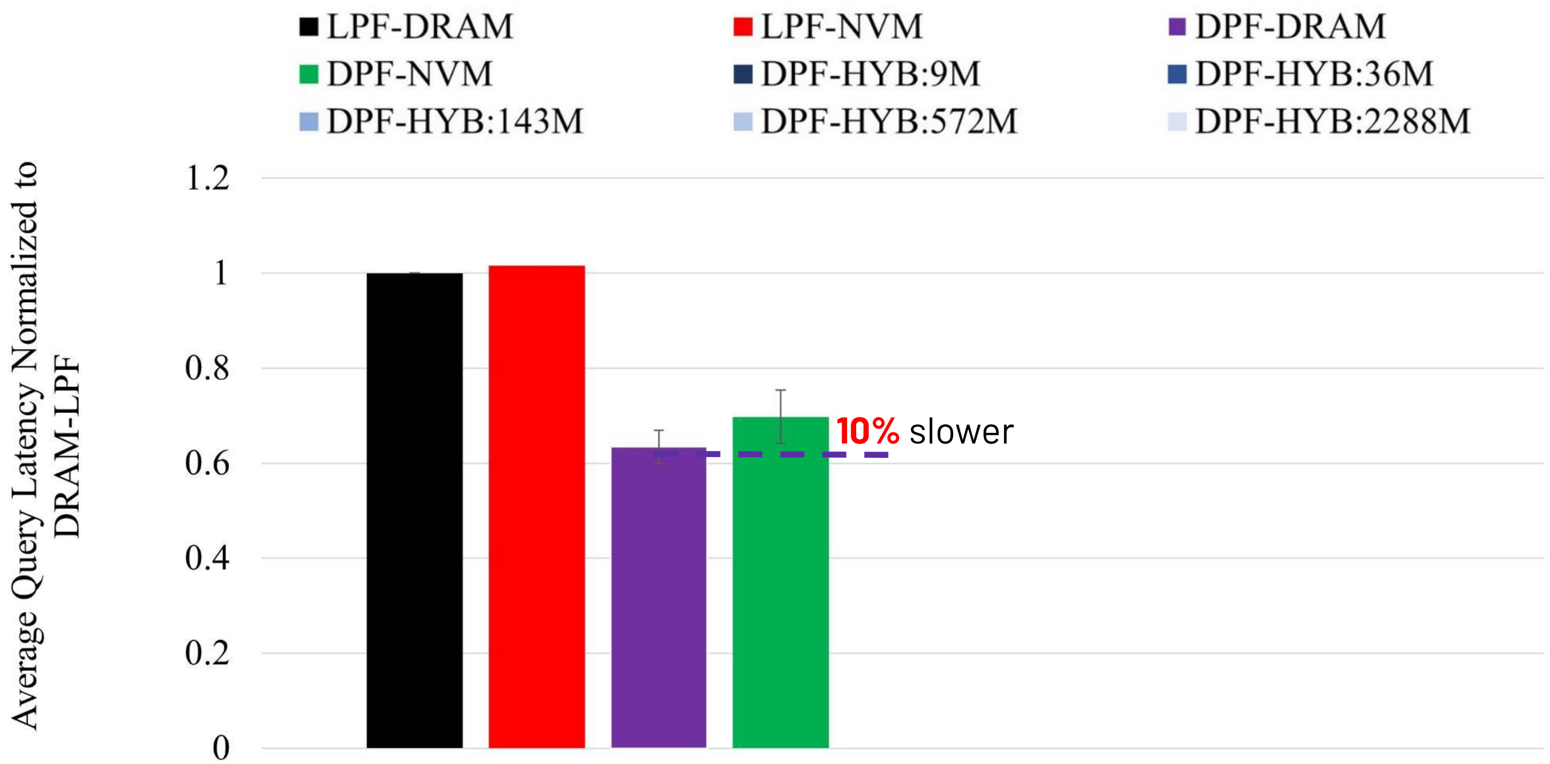
NVM only 2% slower than DRAM for compressed (LPF) index



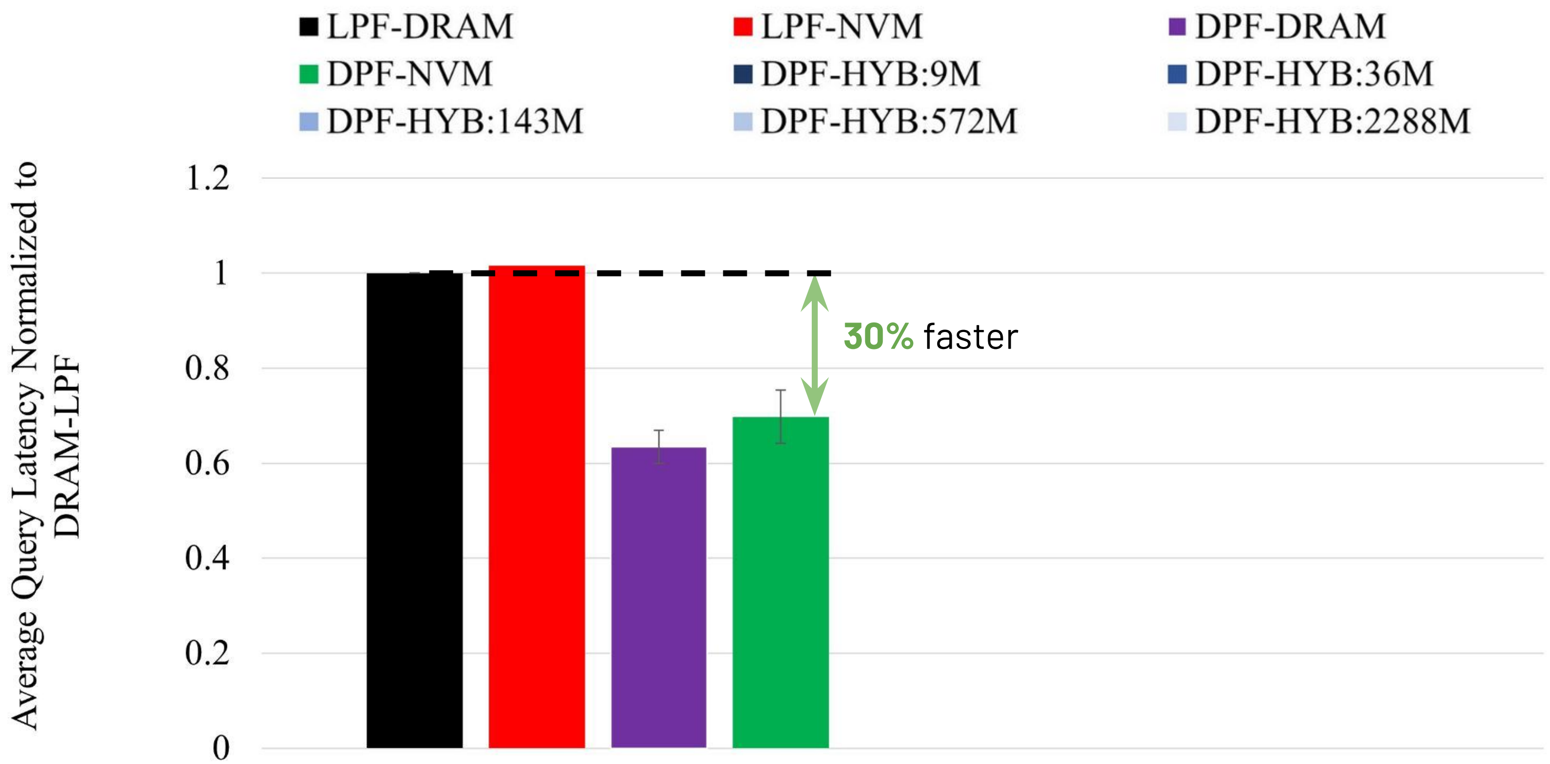
Uncompressed index (DPF) 37% faster than compressed (LPF)



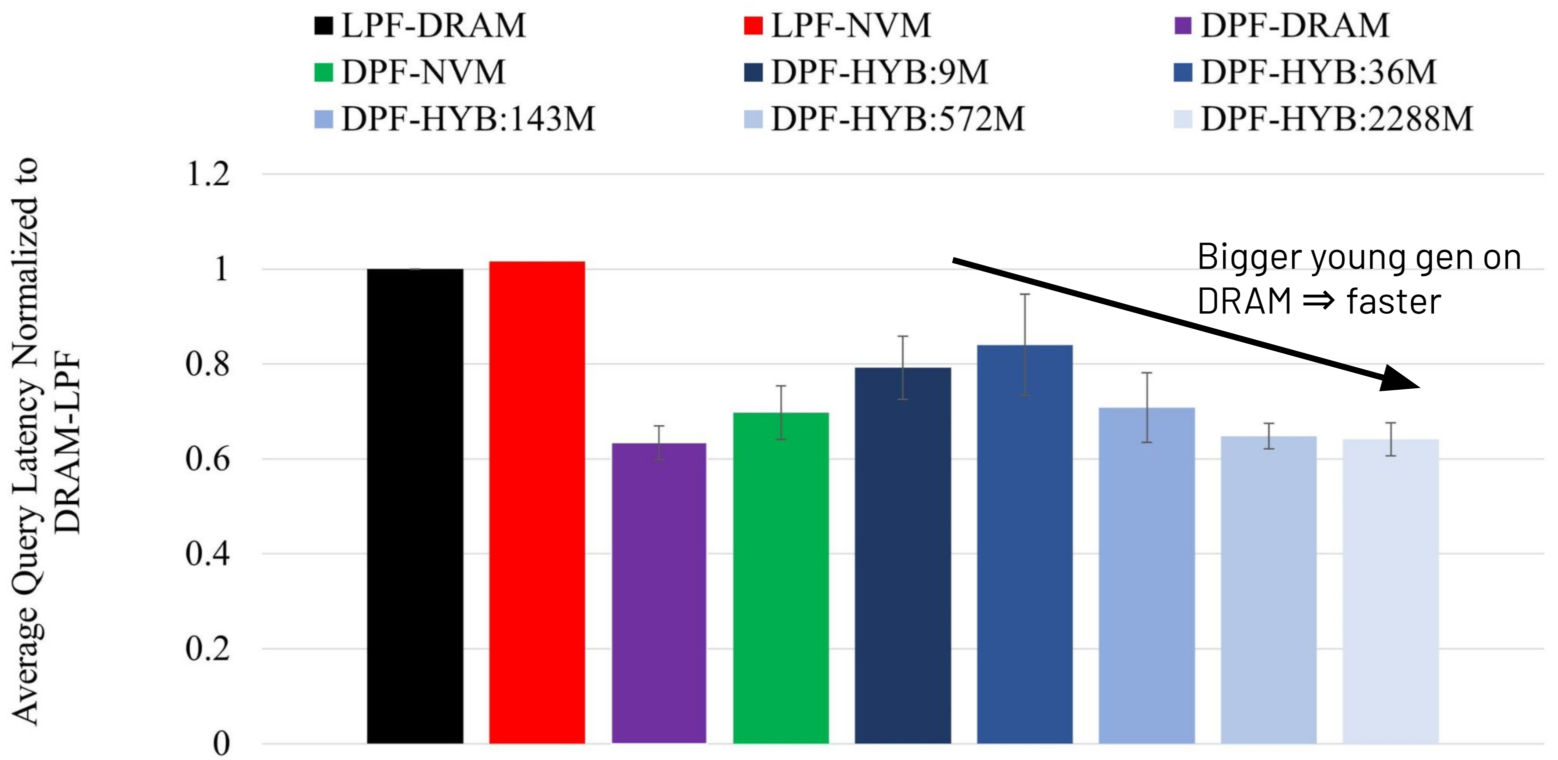
NVM only 10% slower than DRAM for uncompressed (DPF) index



NVM only 10% slower than DRAM for uncompressed (DPF) index



Hybrid setups ≈ DPF-DRAM (fastest system tested)



Surprising results!

- LPF-NVM only 2% slower than LPF-DRAM
- DPF-NVM only 10% slower than DPF-DRAM
- DPF-NVM 30% faster than LPF-DRAM (**SoA**)

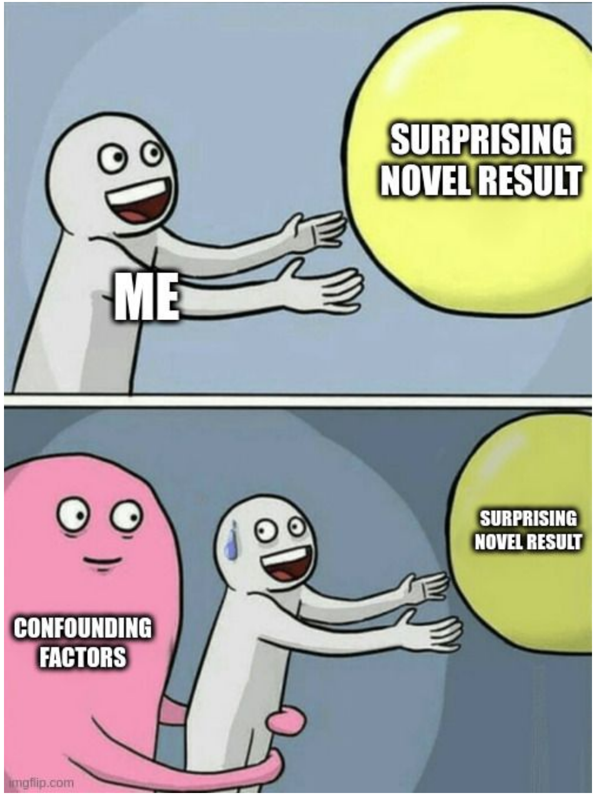
But we know NVM is 2-3x slower

Surprising results!

- LPF-NVM only 2% slower than LPF-DRAM
- DPF-NVM only 10% slower than DPF-DRAM
- DPF-NVM 30% faster than LPF-DRAM (**SoA**)

But we know NVM is 2-3x slower

Confounding factors?



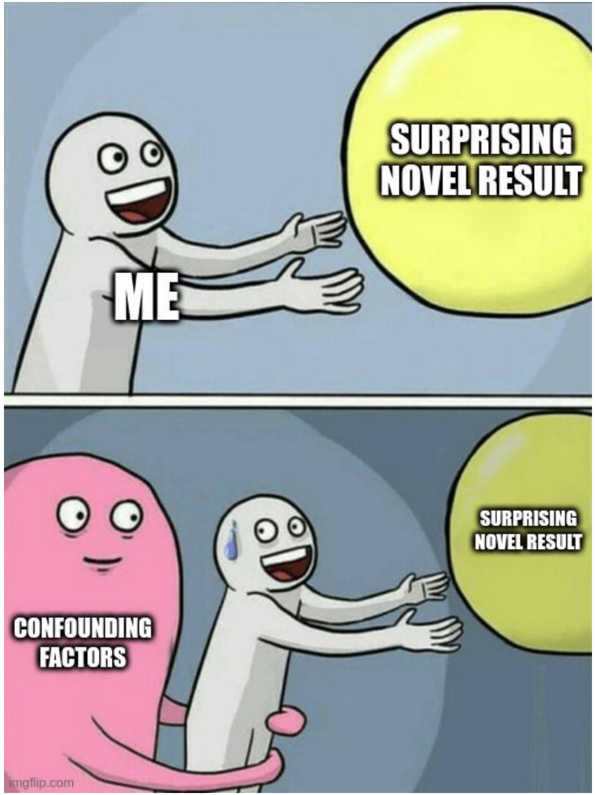
Surprising results!

- LPF-NVM only 2% slower than LPF-DRAM
- DPF-NVM only 10% slower than DPF-DRAM
- DPF-NVM 30% faster than LPF-DRAM (**SoA**)

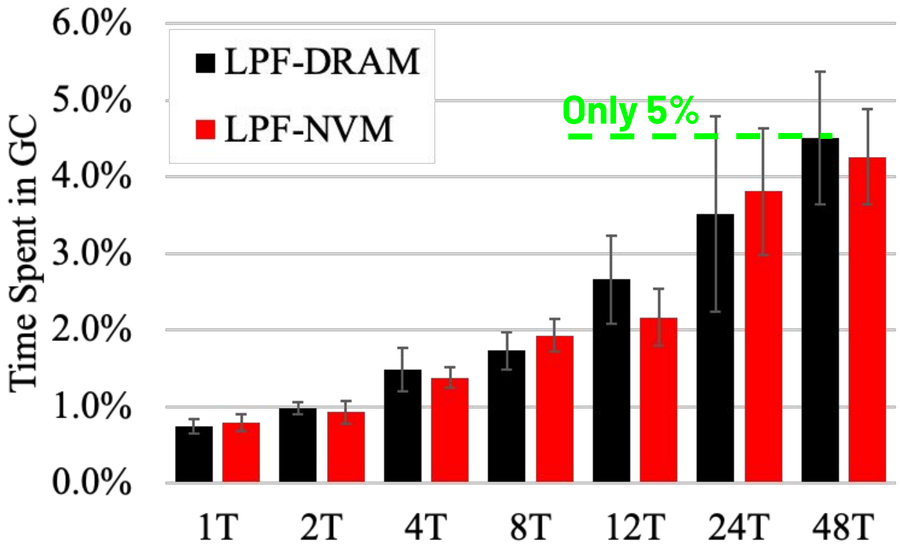
But we know NVM is 2-3x slower

Confounding factors?

- GC overhead?
- Other CPU hardware factors?



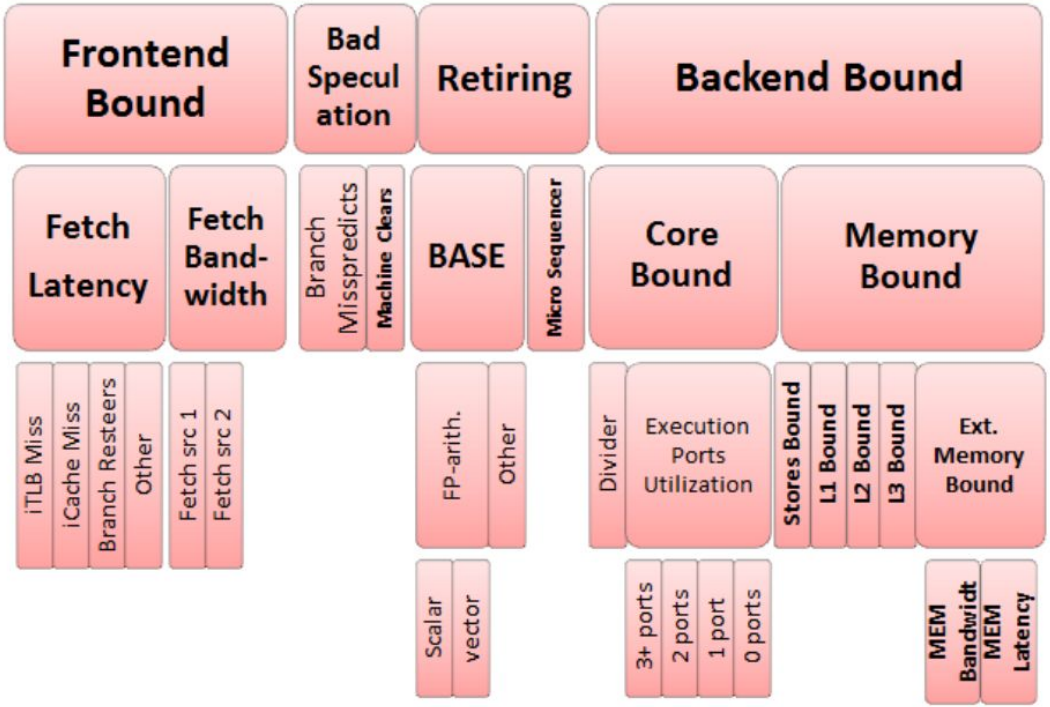
Is GC a confounding factor?



- DPF exhibits **negligible** GC cost
  - Only allocation is per-query objects that are **short-lived** and die in nursery
  - Old gen contains immutable index with primitive arrays (no scanning necessary)
  - **Today:** Big data apps (**try to**) avoid high GC cost by using primitive arrays

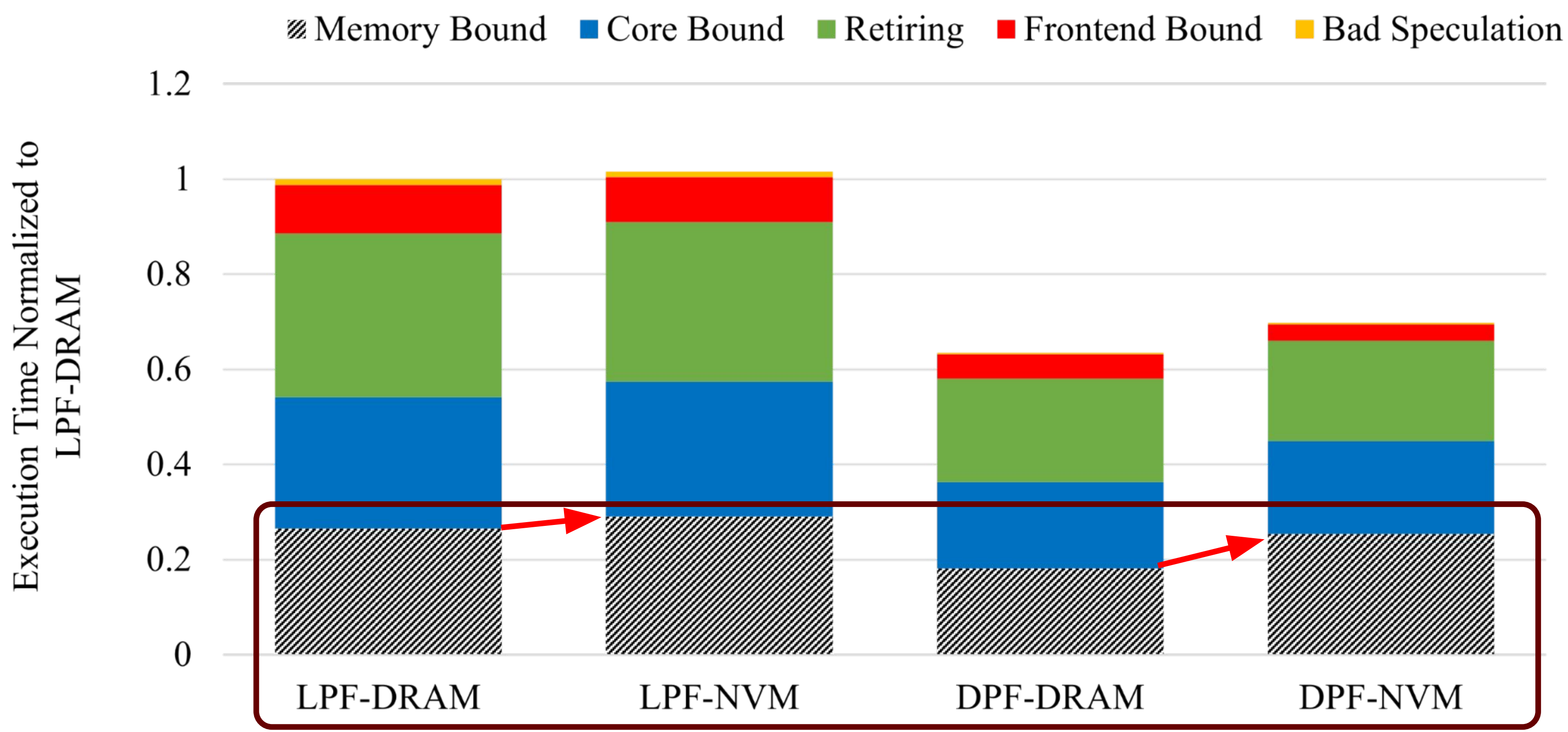
GC is **not** a confounding factor

Intel’s top down approach to performance analysis

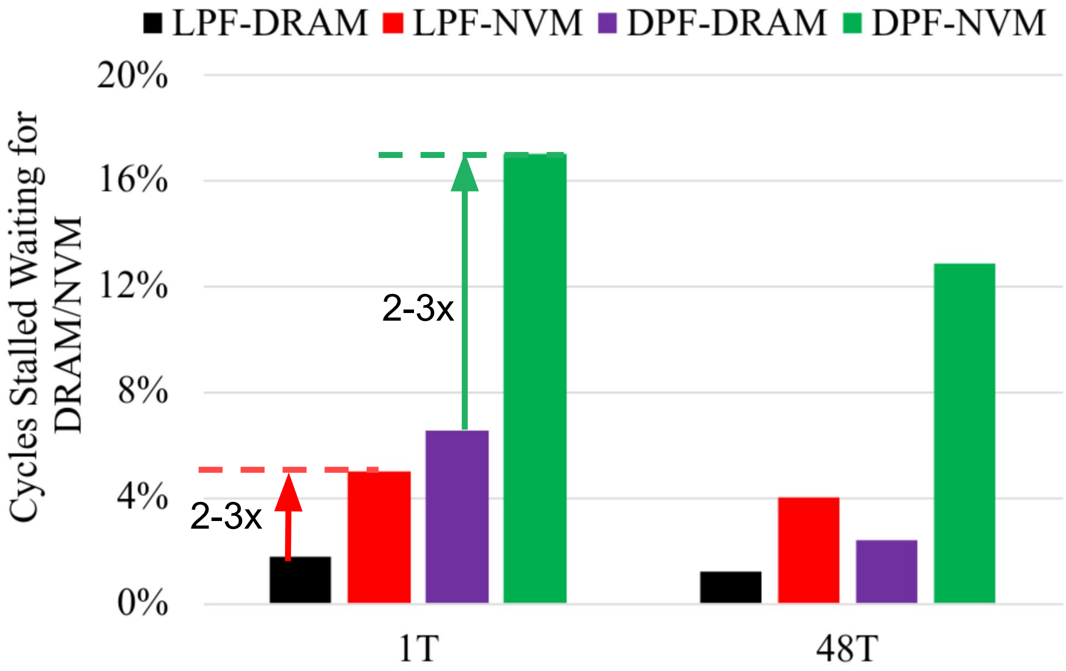


- ILP machinery makes it hard to pinpoint bottlenecks
- Need a systematic methodology to rule out events
- If an issue slot was not **utilized** in a cycle, who is to blame?
  - Memory response time
  - Mis-speculation
  - Overwhelmed decoder
  - Lack of physical reg.

High latency of NVM is not exposed in query execution

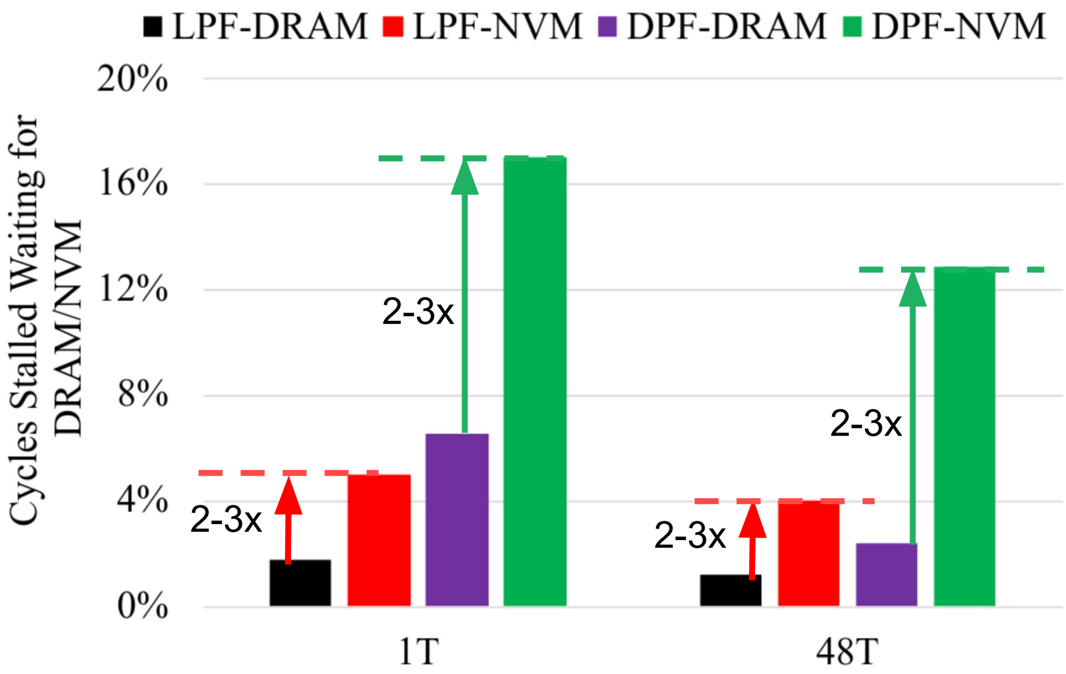


CPU wastes 2–3x more time waiting for NVM than DRAM



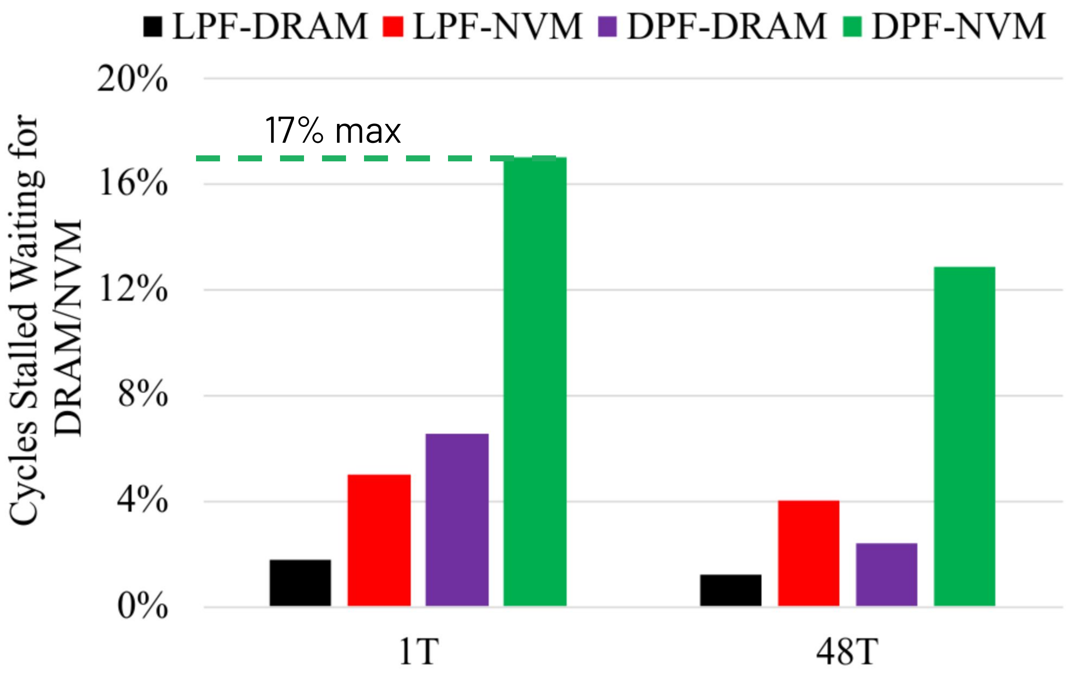
- Both LPF and DPF show 2-3x higher cycles stalled on NVM than DRAM

CPU wastes 2-3x more time waiting for NVM than DRAM



- Both LPF and DPF show 2-3x higher cycles stalled on NVM than DRAM
- Multi-core as well

Cycles stalled on NVM is low



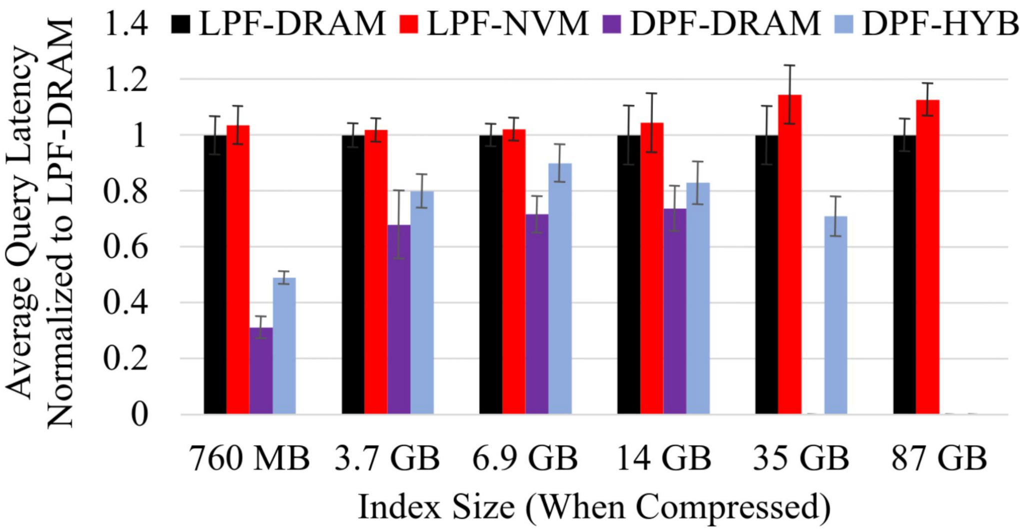
- Both LPF and DPF show 2-3x higher cycles stalled on NVM than DRAM
- Multi-core as well

Do results scale to larger indices?

- Build large indices using open web crawl data
- From now on: show results for **DPF-HYB (2288MB)**

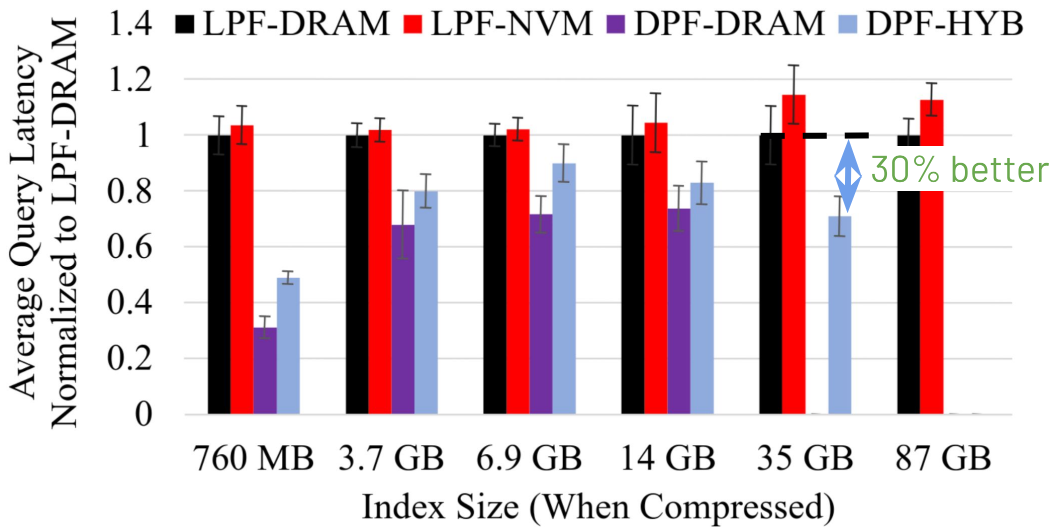
Common Crawl

Findings are applicable to (very) large index sizes



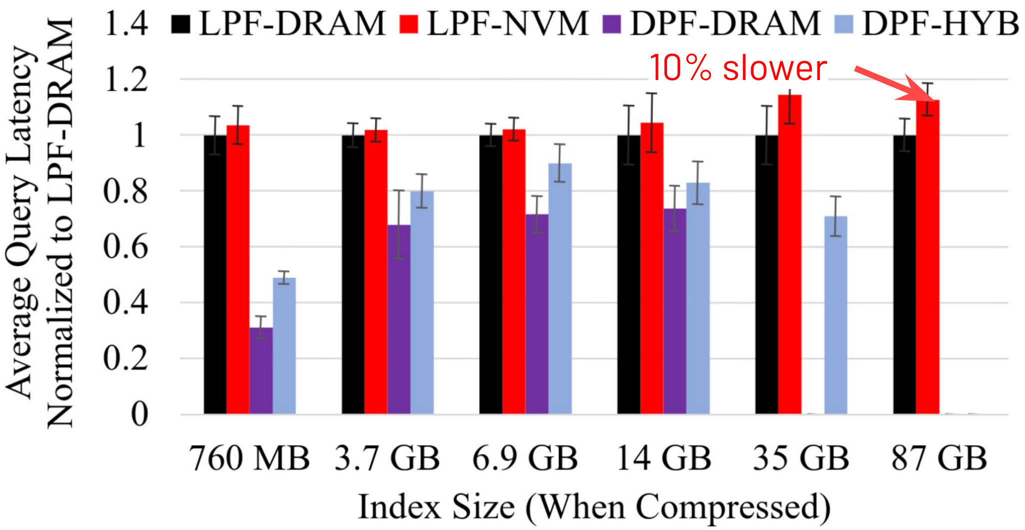
- Missing data: memory exhausted.

Findings are applicable to (very) large index sizes



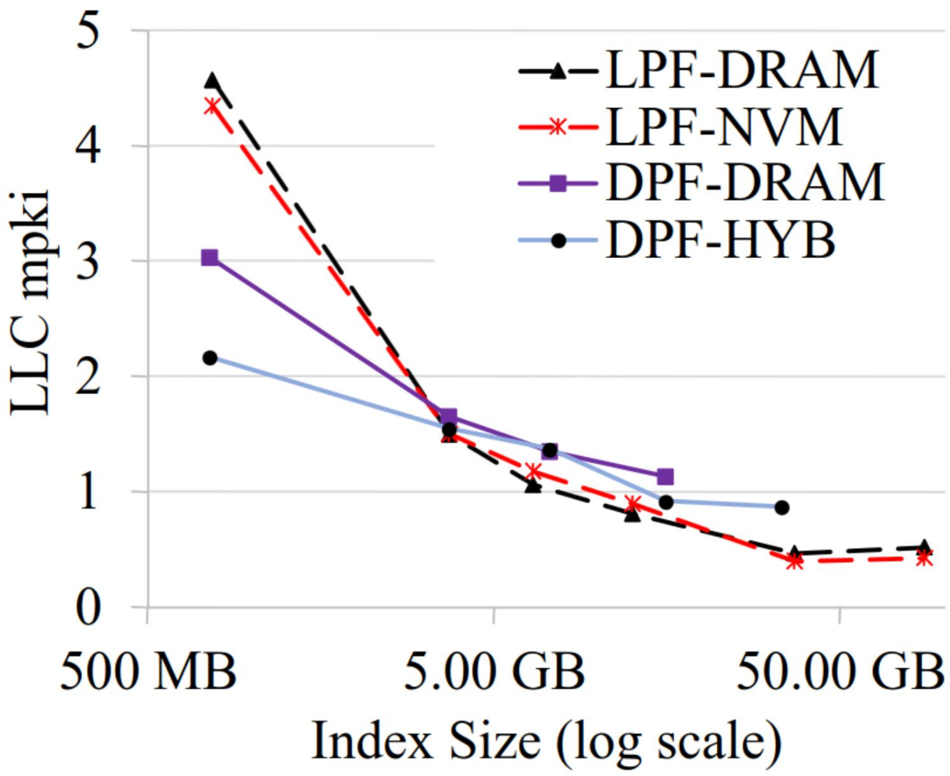
- Missing data: memory exhausted.
- DPF-Hybrid consistently better than LPF-DRAM (SoA)

Findings are applicable to (very) large index sizes

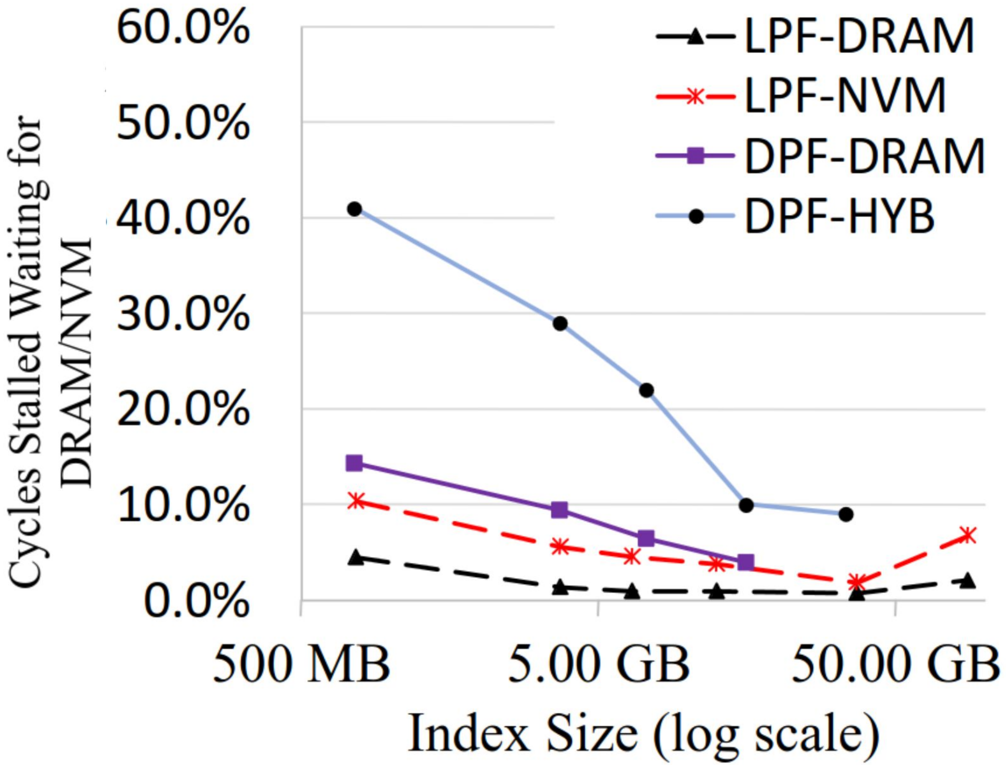


- Missing data: memory exhausted.
- DPF-Hybrid consistently better than LPF-DRAM (SoA)
- LPF-NVM modestly slower than LPF-DRAM.

Key insight: prefetchers more effective for larger indices



Key insight: NVM latency hidden by sequential access pattern and prefetching



Key takeaways

- Memory and storage is evolving
    - Space-time tradeoffs are changing

- Compression + off-heap is standard today for big data apps
    - Critical (but not all) data can have a new home in **uncompressed** format

- **Future Work**
    - **Hardware:** NVMe and remote memory
    - **Software:** Other frameworks + specialized Java heap