Understanding Index Structures for Tiered Memory

Roopam Taneja
RS3 Lab, EPFL

Contents

Abstract

This report details the process of benchmarking and understanding the behavior of index structures (using PiBench) on tiered memory. We evaluate two primary tiering schemes: the kernel's built-in TPP (Transparent Page Placement) coupled with AutoNUMA (`numa_balancing=2`), and the more recent DAMON (Data Access Monitor) framework. The objective is to understand how quickly a small, hot working set of a large index can be promoted to the faster memory tier and what impact this has on application throughput. We analyze results from MASIM, Memtis-B-Tree, and PiBench under various configurations, memory ratios, and access patterns. Please find results at CXL-Index GitHub. Please do read `qemu-cxl-hmsdk.pdf` and `README.md` files in `experiments` and `experiments/final_results`.

1 Introduction

The goal of this project is to benchmark and understand the behavior of index structures, particularly using PiBench, in a tiered memory environment. The experiments in this report were conducted on srv4 configured with local and remote DRAM, where the remote DRAM simulates a slower, high-latency memory tier. This setup serves as a doundation for upcoming experiments on hardware with true local/remote CXL memory (on srv9).

We compare following memory tiering schemes:

1. **TPP (Transparent Page Placement) + AutoNUMA:** This scheme uses the kernel's built-in NUMA balancing features. It is configured with `numa_balancing = 2` (allowing hot page promotion) and `demotion_enabled=true` (enabling TPP's LRU-based demotion).
2. **DAMON (Data Access Monitor):** This scheme utilizes the `damo` userspace tool to implement memory tiering policies, specifically using `migrate-hot` and `migrate-cold` actions.
3. **Baseline (No Tiering):** For comparison, we also run benchmarks without any tiering scheme, using `numa_balancing = 0`.

The primary aim is to observe how effectively a small, hot working set within a large index (spanning both memory nodes) is identified and promoted to the faster tier (Node 0). We hypothesize that an effective tiering scheme will quickly promote the hot set, after which the number of promotions should stabilize, leading to improved application throughput.

2 System Setup and Configuration

2.1 Kernel Configuration for Tiered Memory

A critical prerequisite for AutoNUMA (`numa_balancing=2`) to function correctly is the system's recognition of multiple memory tiers, visible in `/sys/devices/virtual/memory_tiering/memory_tier*`.

When both nodes are of the same memory type (e.g., DRAM), the kernel may not differentiate them. Initial attempts to patch the kernel source based on an HMSDK GitHub issue failed to trigger successful page promotions (`pgpromote_success`). I still don't know the exact reason. Even increased CXL_MEM_DIST to `10 $\times $ DRAM\_DIST` from `5 $\times $ DRAM\_DIST` in the issue, but that didn't help.

A working solution was found using the `tierinit-patch` from the colloid-tb repository tierinit-patch. After patching the kernel source and loading as a kernel module, it successfully creates two visible memory tiers and enables AutoNUMA promotions and demotions. So this works and is also flexible (the module can be loaded/unloaded at will).

Alongside this patch, DAMON support was enabled in the kernel configuration. This is how `.config` looked for compiling the kernel with DAMON support:

```
$ grep -i damon .config

CONFIG_DAMON=y
CONFIG_DAMON_VADDR=y
CONFIG_DAMON_PADDR=y
CONFIG_DAMON_SYSFS=y
```

```
CONFIG_DAMON_RECLAIM=y
CONFIG_DAMON_LRU_SORT=y
CONFIG_DAMON_STAT=y
CONFIG_DAMON_STAT_ENABLED_DEFAULT=y
# DAMON Samples
# CONFIG_SAMPLE_DAMON_WSSE is not set
# CONFIG_SAMPLE_DAMON_PRCL is not set
# CONFIG_SAMPLE_DAMON_MTIER is not set
# end of DAMON Samples
```

Listing 1: Kernel .config for DAMON

Finally, two other system settings were configured::

- `lru_gen_enabled = 0x0007`
- `zone_reclaim_mode = 0`

The experiments were conducted on the `srv4` machine running kernel version `6.17.1`.

2.2 Tiering Scheme Details

2.2.1 AutoNUMA + TPP

This scheme was enabled by setting `numa_balancing = 2` and `demotion_enabled=true`. The primary metrics monitored for this scheme were:

- **Promotions:** `pgpromote_success`
- **Demotions:** `pgdemote_kswapd`

2.2.2 DAMON

Initially, the HMSDK framework was considered, as it was developed to support CXL devices and provided memory tiering based on DAMON. HMSDK originally extended DAMOS with `migrate_hot` and `migrate_cold` actions. However, these actions have since been upstreamed into the mainline kernel. The latest versions of DAMON now offer more advanced features than the original HMSDK, including applying actions per-NUMA-node via the command line and auto-tuning of monitoring intervals and quotas. Therefore, the latest DAMON version was used directly.

NOTE : We have `demotions_enabled=false` with DAMON, since we are focussing on DAMON demotions only.

The DAMON-based tiering was implemented using a userspace script, `damon-mem-tier.sh`. It's taken from here: sjp-damon-patch.

The core `damo` command is as follows:

```
"$damo_bin" start \
        --numa_node 0 --monitoring_intervals_goal 4% 3 5ms 10s \
                --damos_action migrate_cold 1 --damos_access_rate 0% 0% \
                --damos_apply_interval 1s \
                --damos_quota_interval 1s --damos_quota_space 200MB \
                --damos_quota_goal node_mem_free_bp 0.5% 0 \
                --damos_filter reject young \
        --numa_node 1 --monitoring_intervals_goal 4% 3 5ms 10s \
                --damos_action migrate_hot 0 --damos_access_rate 5% max \
                --damos_apply_interval 1s \
                --damos_quota_interval 1s --damos_quota_space 200MB \
                --damos_quota_goal node_mem_used_bp 99.7% 0 \
                --damos_filter allow young \
                --damos_nr_quota_goals 1 1 --damos_nr_filters 1 1 \
        --nr_targets 1 1 --nr_schemes 1 1 --nr_ctxs 1 1
```

Listing 2: damon-mem-tier.sh command

A brief explanation of the key parameters:

`--monitoring_intervals_goal 4\% 3 5ms 10s` This tweaks monitoring intervals to capture 4% of accesses in 3 aggregate intervals, with a `sample\_interval` allowed to vary between 5ms and 10s. This can likely be left as the default.

`--damos_action migrate_cold 1 --damos_access_rate 0\% 0\%` This applies `migrate\_cold` (demotion), sending all pages with an access rate of 0% (not touched in the last aggregate interval) to Node 1. This also likely does not need modification.

`--damos_action migrate_hot 0 --damos_access_rate 5\% max` This applies `migrate\_hot` (promotion), sending all pages with an access rate of 5% or more to Node 0. 5% is a good default.

`--damos_apply_interval 1s` Actions are applied every 1 second, which seems reasonable, can be tweaked for more aggressive tuning.

`--damos_quota...` These are the key values that **can be modified**. The `damos_quota_goal` settings aim for a target memory usage on Node 0 (e.g., 99.7% used). However, the static `damos_quota_space 200MB` puts an upper cap (200 MB/s in this case) on the migration overhead. This is the main value that can be tweaked; the others look good as defaults.

Metrics for DAMON were obtained by querying sysfs. After setting `kdamonds/N/refresh_ms` to 100ms, the number of pages migrated was calculated from `/kdamonds/N/contexts/0/schemes/0/stats/sz_applied` divided by the page size. (refresh-damon-patch)

Thus, these numbers get us:

- **Promotions:** `migrate_hot_pages`
- **Demotions:** `migrate_cold_pages`

Further discussion on DAMON metrics can be found in damo-issue-34 and hmsdk-issue-6.

2.3 DAMON/DAMOS Internals

DAMON and DAMOS work in concert. For our use case, we wish to monitor using DAMON and apply actions using DAMOS.

- **DAMON (Monitor):** Deals with region adjustments and auto-tuning of monitoring intervals.
- **DAMOS (Operations):** Deals with applying actions (like migration) based on quotas, priorities, filters, and watermarks.

Recent DAMON features include auto-tuning for monitoring intervals (Figure 1) and quotas (Figure 2), as detailed in damon-kernel-recipes-2025 and damon-kernel-design.

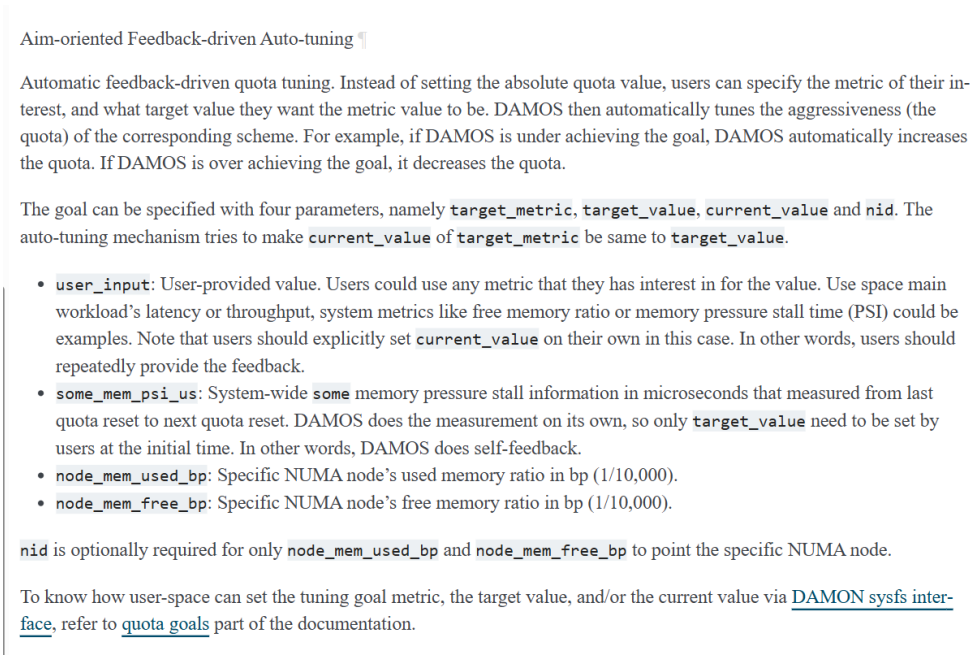
Aimed Monitoring Output-oriented Intervals Auto-tuning

- Change Question: How to do? (mechanism) → What to achieve? (final goal, policy)
- Let users specify
  - Desired amount of access events to capture in each snapshot
  - Minimum and maximum sampling intervals
- Find sampling/aggregation intervals for the desire using a feedback loop
  - Increase intervals if less than desired events are captured in current snapshot
  - Decrease intervals if more than desired events are captured in current snapshot

Monitoring Intervals Auto-tuning Parameters

- Parameters for parameters auto-tuning, but easy to set
- Suggestion
  - Desired access events per snapshot: 4% of per-snapshot maximum capturable events
  - Min/max sampling intervals: 5ms and 10s
  - Sampling:aggregation intervals ratio: 1:20
  - Proven to be useful on multiple real-world production workloads
  - Isn't this another heuristic? Yes, but the maintainer will be there to support this

Figure 1: DAMON Monitoring Interval Auto-tuning damon-kernel-recipes-2025



Aim-oriented Feedback-driven Auto-tuning ¶

Automatic feedback-driven quota tuning. Instead of setting the absolute quota value, users can specify the metric of their interest, and what target value they want the metric value to be. DAMOS then automatically tunes the aggressiveness (the quota) of the corresponding scheme. For example, if DAMOS is under achieving the goal, DAMOS automatically increases the quota. If DAMOS is over achieving the goal, it decreases the quota.

The goal can be specified with four parameters, namely `target_metric`, `target_value`, `current_value` and `nid`. The auto-tuning mechanism tries to make `current_value` of `target_metric` be same to `target_value`.

- `user_input`: User-provided value. Users could use any metric that they has interest in for the value. Use space main workload's latency or throughput, system metrics like free memory ratio or memory pressure stall time (PSI) could be examples. Note that users should explicitly set `current_value` on their own in this case. In other words, users should repeatedly provide the feedback.
- `some_mem_psi_us`: System-wide `some` memory pressure stall information in microseconds that measured from last quota reset to next quota reset. DAMOS does the measurement on its own, so only `target_value` need to be set by users at the initial time. In other words, DAMOS does self-feedback.
- `node_mem_used_bp`: Specific NUMA node's used memory ratio in bp (1/10,000).
- `node_mem_free_bp`: Specific NUMA node's free memory ratio in bp (1/10,000).

`nid` is optionally required for only `node_mem_used_bp` and `node_mem_free_bp` to point the specific NUMA node.

To know how user-space can set the tuning goal metric, the target value, and/or the current value via DAMON sysfs interface, refer to quota goals part of the documentation.

Figure 2: DAMON Quota Auto-tuning damon-kernel-design

schemes/<N>/stats/

DAMON counts statistics for each scheme. This statistics can be used for online analysis or tuning of the schemes. Refer to design doc for more details about the stats.

The statistics can be retrieved by reading the files under `stats` directory (`nr_tried`, `sz_tried`, `nr_applied`, `sz_applied`, `sz_ops_filter_passed`, and `qt_exceeds`), respectively. The files are not updated in real time, so you should ask DAMON sysfs interface to update the content of the files for the stats by writing a special keyword, `update_schemes_stats` to the relevant `kdamonds/<N>/state` file.

Figure 3: DAMON stats sysfs interface

3 Experimental Methodology

3.1 Workload Configuration

All benchmarks were run with a 1 billion key configuration. The workloads included a mix of read-heavy and write-heavy operations, using both `UNIFORM` (no skew) and `SELFSIMILAR` (variable skew) data distributions. The desired memory ratio between the fast and slow tiers was achieved using a userspace memeater application.

3.2 Benchmarks

A suite of benchmarks was used to evaluate the tiering schemes:

- **PiBench (CXL-Index):** The primary benchmark for evaluating index structures.
- **MASIM:** A synthetic workload generator.
- **Memtis-Btree:** A B-Tree implementation.
- **GUPS:** Used for initial validation and hotset experiments.

All benchmarks are run with `--cpunodebind=0` to ensure CPU affinity to Node 0 cores.

3.3 Code Modifications

Several modifications were made to the benchmarking framework:

1. Added monitoring support for `vmstat` and DAMON stats to `run.py`.
2. Added benchmark-specific arguments to run different benchmarks from `run.py`.
3. Added support for easily using existing options (`memhog`, `mon-perf` etc.) in `run.py`.
4. `run.py` was updated to automatically calculate and hog required memory on Node 0 using userspace memeater based on the desired memory ratio and benchmark RSS.
5. Added basic PiBench stdout parsing in `launch_helpers.py`.
6. Modified `index-benchmarks/latches/OMCSOffset.h` to use `malloc()` instead of `numa_alloc()`:

```
//base_qnode = (QNode *)numa_alloc_interleaved(npages * PAGE_SIZE);
base_qnode = (QNode *)malloc(npages * PAGE_SIZE);
//base_qnode = (QNode *)numa_alloc_onnode(sizeof(QNode) * Lock::kNumQueueNodes,
    node);
base_qnode = (QNode *)malloc(sizeof(QNode) * Lock::kNumQueueNodes);

```

Listing 3: Modification in OMCSOffset.h

7. For the Memtis-Btree benchmark, the code was modified to restrict the find operation to the first 25% of keys, creating a smaller, defined hotset (`NELEMENTS / 4`).

8. The `launch.py` script was modified to `LD_PRELOAD libjemalloc.so` for `cxl-index` and set `OMP_NUM_THREADS` for `btree`.

9. An option was added to `run.py` to pass an interleave ratio with `--numa_interleave` to make use of numactl's `--weighted-interleave` mode. However, since this is restricted to allocation and wouldn't help our purpose, it was not explored further.

10. As a side-quest, the PCM version in the PiBench repository was updated to make it work for newer architectures. The `pibench` submodule in `index-benchmarks/` points to this updated fork.

3.4 Initial Challenges

Initial experiments with PiBench were unreliable. Problems included:

1. Failure to see two distinct memory tiers, which prevented AutoNUMA promotions (fixed by the `tierinit-patch`).

2. A cap on processor frequency, which skewed performance results (`MEMBIND1` having better or same throughput as `MEMBIND0`).

Due to these issues, most early experimental results are not useful though stored in `experiments/old_experimental`

There were also issues with realising the multi-threaded nature of Memtis-BTree benchmark, which were fixed by setting `OMP_NUM_THREADS`, however since the BTree code itself was revamped, it is not an issue now.

For MASIM as well, it took some time to arrive at a correct configuration script suitable for our experiments.

3.5 PiBench WSS Analysis

A WSS (Working Set Size) analysis was performed for PiBench (30G RSS, 1s sample time) using the `wss.pl` script from wss-script to understand the memory footprint of different access patterns using default method of `sudo ./wss.pl <pid> <time>`.

Table 1: PiBench WSS (1s) for 30G RSS

Distribution (Skew)	**WSS (1s)**
UNIFORM	28.3 G
SELFSIMILAR (0.5)	28.2 G
SELFSIMILAR (0.2)	26.4 G
SELFSIMILAR (0.1)	24.6 G
SELFSIMILAR (0.01)	15.7–16 G
SELFSIMILAR (0.005)	11.7 G
SELFSIMILAR (0.004)	10.4 G
SELFSIMILAR (0.002)	7.5 G
SELFSIMILAR (0.0015)	6.3 G
SELFSIMILAR (0.001)	4.9 G
ZIPFIAN (0.5)	27.7 G
ZIPFIAN (0.9)	26 G
ZIPFIAN (0.99)	23.7 G
ZIPFIAN (0.999)	23.6 G
ZIPFIAN (0.9999)	23.4 G

4 Results and Analysis

4.1 MASIM Benchmark

MASIM is single-threaded. More info at MASIM GitHub Repo.

Goal: Validate autonuma promotions performing better than baseline for a known hotset on remote memory.

A config was designed to create two regions, `r0` (40G) and `r1` (10G), with `r1` acting as the hotset. Settled on this config script after several trials and errors.

```
r0, 40000000000, none
r1, 10000000000, none

p0
20000
r0, 0, 4096, 1, wo

p1
50000
r1, 0, 4096, 1, wo

p2
100000
r1, 0, 4096, 1, ro
```

Listing 4: MASIM Configuration Script

RSS is 47.6G (since it is 5e9 bytes). With a 1:1 memory ratio ensured by userspace memeater (24G on Node 0), we want `r0` to fill Node 0 and spill to Node 1, guaranteeing the hotset `r1` is initially allocated on Node 1.

If two regions are accessed in same phase, their accesses are interleaved. Thus to achieve our desired allocation, it is important for their allocation phases (p0, p1) to be different. They must have writes to actually increase RSS.

Our measurement phase `p2` then sequentially accesses this hotset `r1`. Sequential access allows a predictable WSS (checked using wss tool, its close to 9.4G for 1s).

Throughput values time-series at 1-second intervals were collected using `--log_interval=1000` flag in MASIM command. The values were averaged over 5 runs for each config.

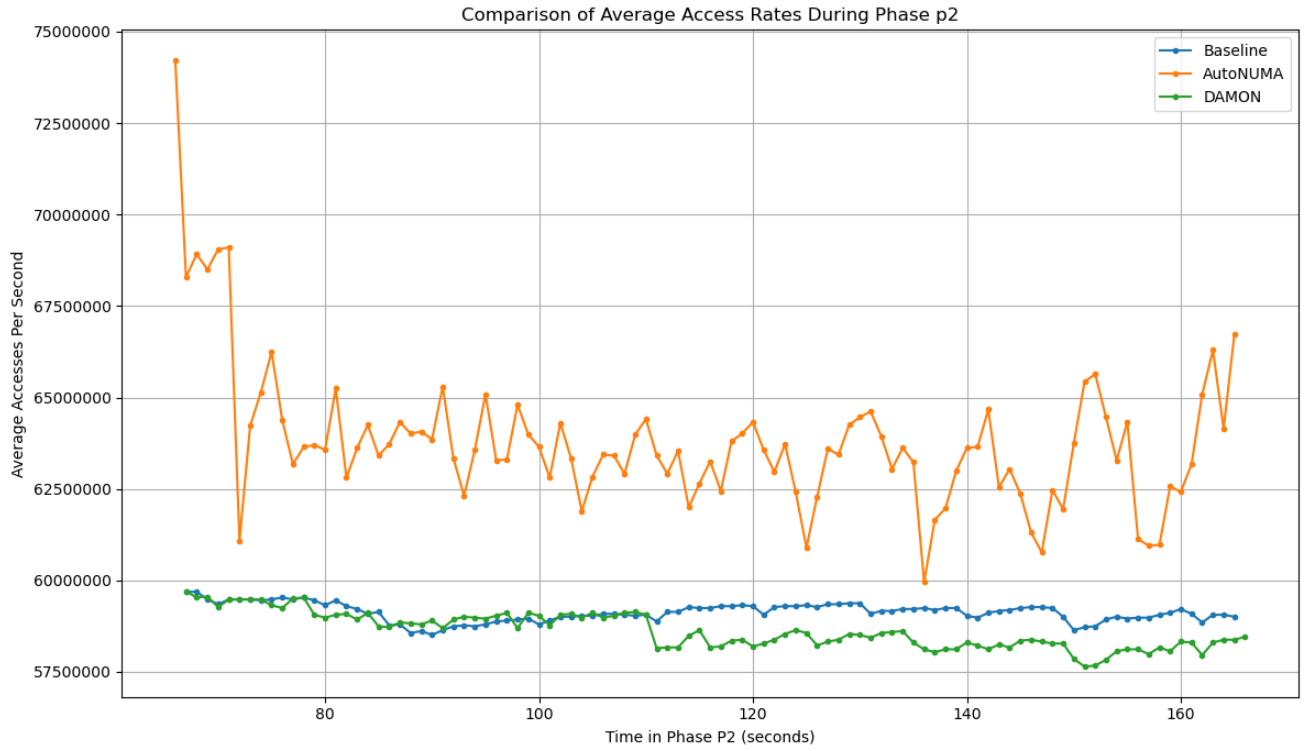


Figure 4: MASIM 1:1 Ratio Time Series (Avg. 5 runs)

As shown in Figure 4, for the 1:1 case, AutoNUMA shows a clear improvement in throughput over the baseline. DAMON's performance is similar to the baseline. The number of migrated pages was also

significantly higher for AutoNUMA than for DAMON in this workload.

The results for the above config along with plots are saved in `final_results/masim_correct_run`. Further details in `masim_correct_run/README.md`.

4.2 Memtis-BTree Benchmark

The B-Tree benchmark using updated code which tracks throughput was executed on srv4.

- No. of Elements: 400M
- No. of Lookups: 5000000000M
- Secs to run: 120

The RSS was 39.8GB. To create a smaller WSS, the find operation was restricted to the first 25% of keys, resulting in a WSS of 17-18G. Hence results for `MEMBIND0`, `MEMBIND1`, `1:1` and `2:1` for 8, 16 and 28 threads. Further details in `final_results/README.md`

Figure 5 shows the throughput and migration comparison for all configurations. Figures 6 through 8 show the time-series throughput logs for 8, 16, and 28 threads, respectively. All values have been **averaged over 5 runs**.

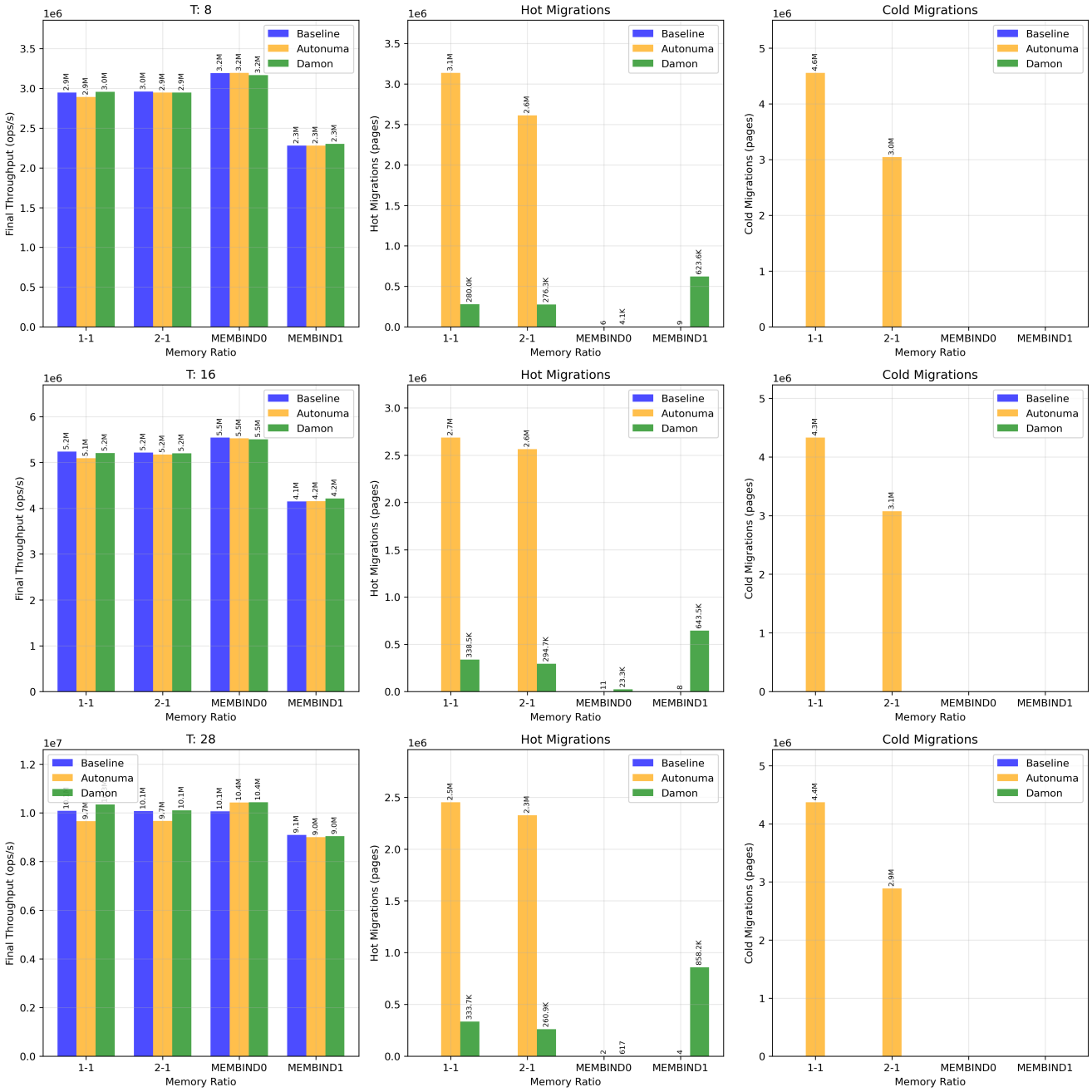
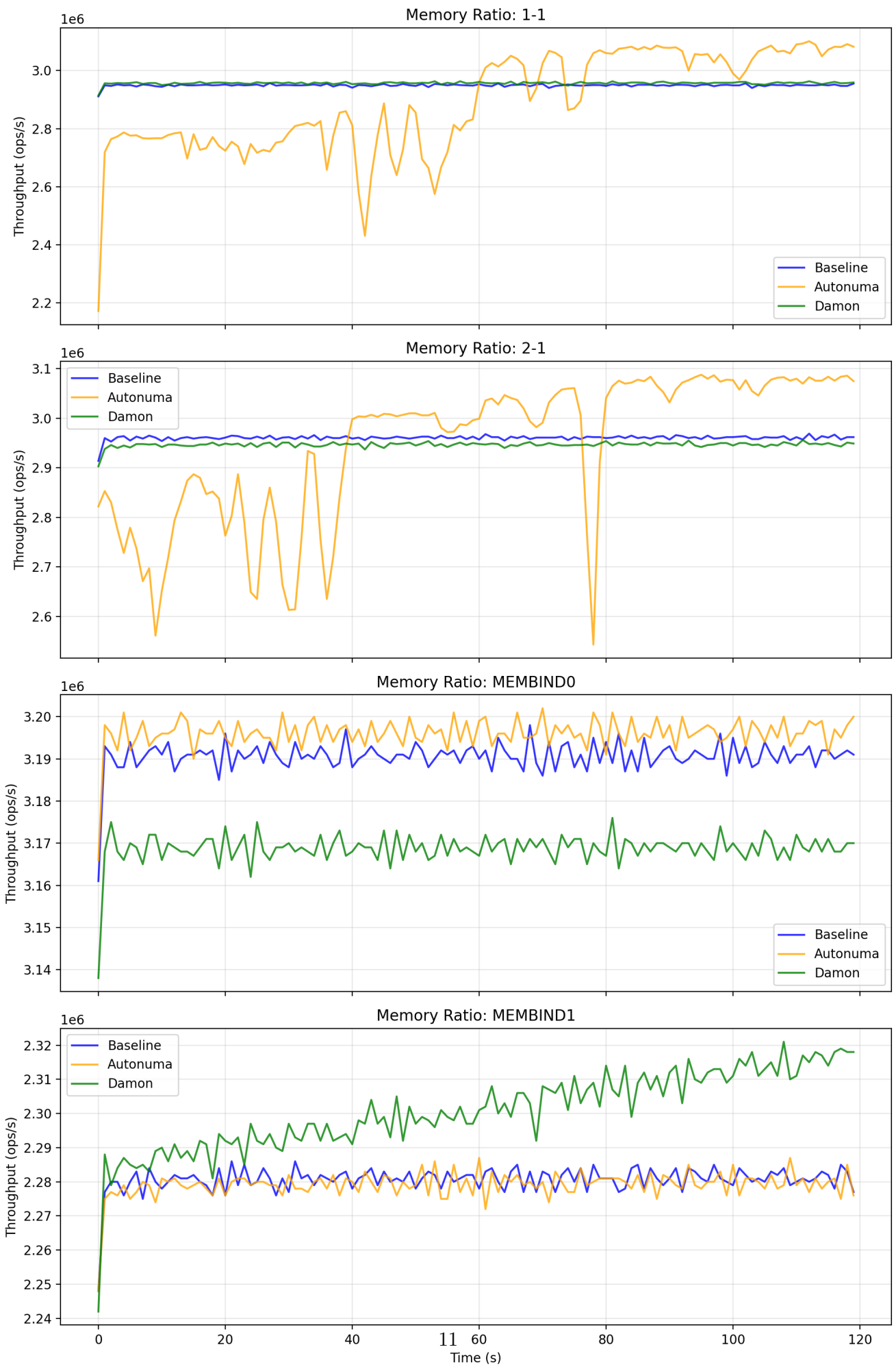
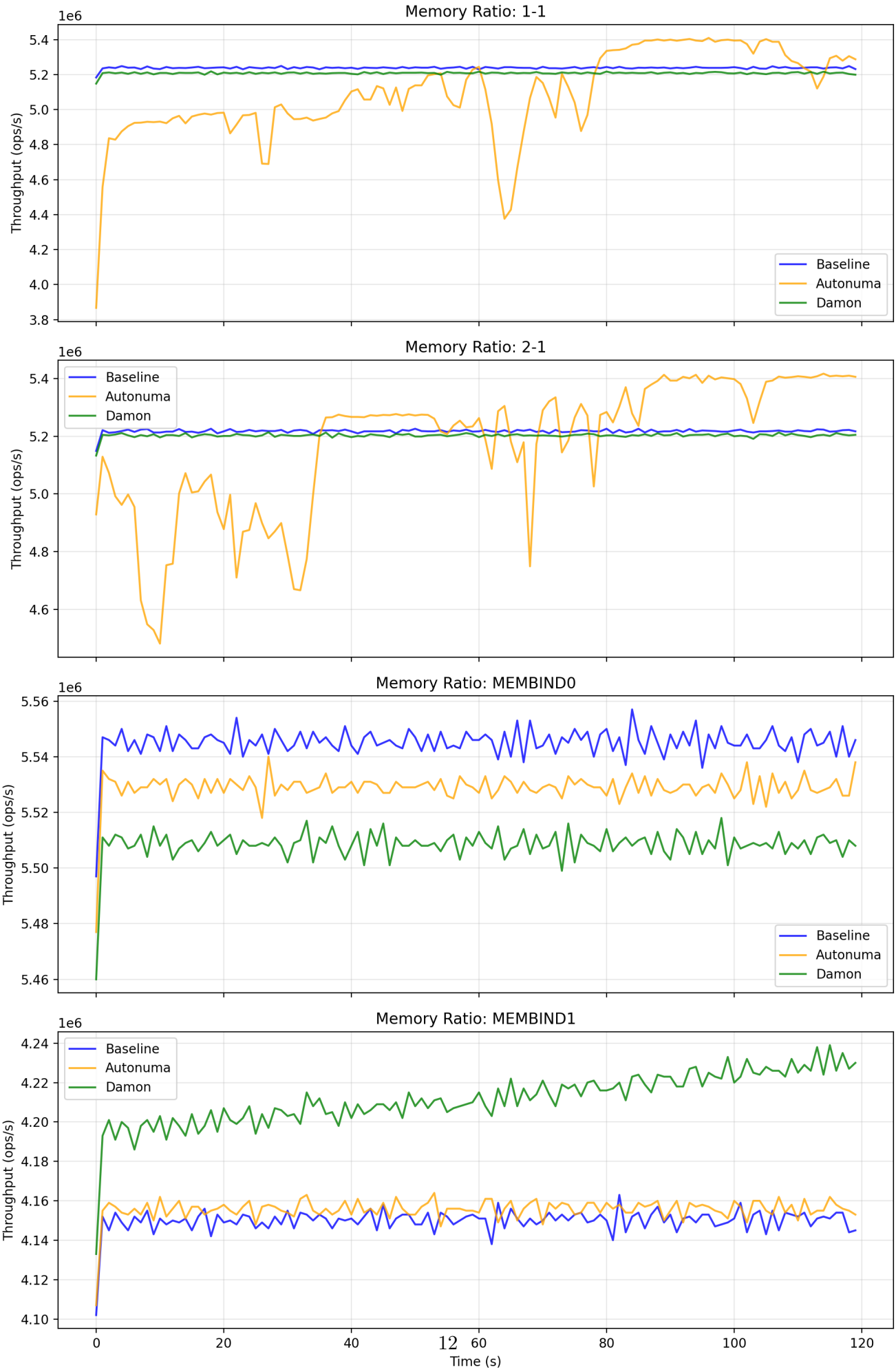
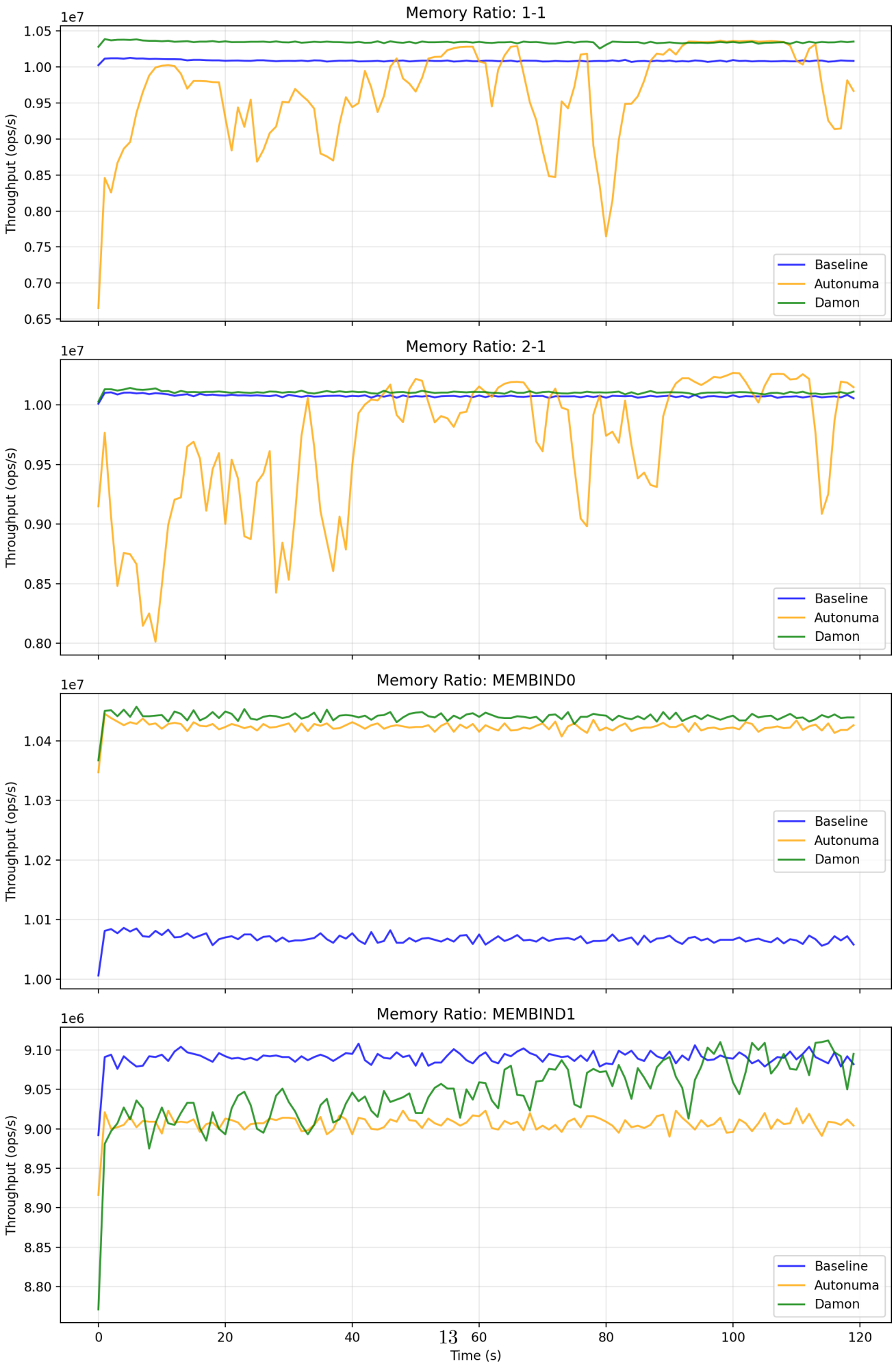


Figure 5: B-Tree Throughput Comparison (Avg. 5 runs)


Per-second Throughput (avg over iterations) - T:8
Memory Ratio: 1-1
Memory Ratio: 2-1
Memory Ratio: MEMBIND0
Memory Ratio: MEMBIND1
1e6
Throughput (ops/s)
Time (s)
Baseline
Autonuma
Damon


Per-second Throughput (avg over iterations) - T:16
Memory Ratio: 1-1
Memory Ratio: 2-1
Memory Ratio: MEMBIND0
Memory Ratio: MEMBIND1
Throughput (ops/s)
1e6
Time (s)
Baseline
Autonuma
Damon


Per-second Throughput (avg over iterations) - T:28
Memory Ratio: 1-1
1e7
Throughput (ops/s)
Baseline
Autonuma
Damon
Memory Ratio: 2-1
1e7
Throughput (ops/s)
Baseline
Autonuma
Damon
Memory Ratio: MEMBIND0
1e7
Throughput (ops/s)
Baseline
Autonuma
Damon
Memory Ratio: MEMBIND1
1e6
Throughput (ops/s)
Baseline
Autonuma
Damon
Time (s)

4.3 PiBench (Index Benchmark)

For more info on all PiBench experiments conducted on srv4, do check `final_results/README.md`.
Common parameters:

- **Keys:** 1e9
- **Duration:** 120s
- **Replicates:** 3 (All values averaged over 3 runs)
- **RSS:** 30G

4.3.1 Initial Run (Skew 0.2)

The first proper run of the PiBench benchmark (`index_first_proper_run`) used a `SELFSIMILAR` distribution with a skew of 0.2 compared with `UNIFORM` distribution for 16 and 28 threads. All files in `final_results/index_first_proper_run`.

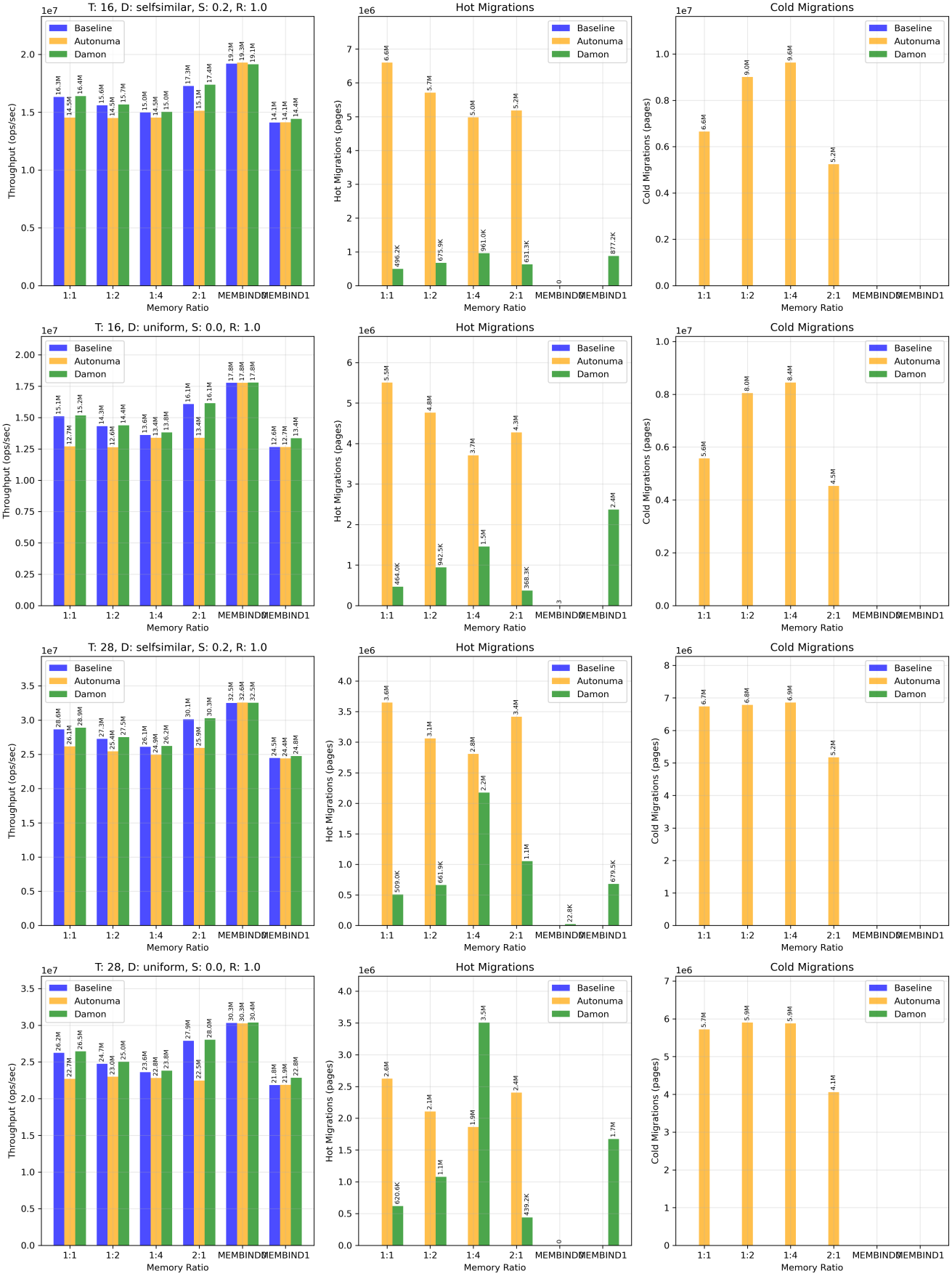


Figure 9: PiBench (Skew 0.2)

Key observations from this run (Figure 9):

- **Throughput:** Throughput slightly degraded for AutoNUMA compared to baseline, while it remained same or slightly improved for DAMON. This is possibly due to the synchronous nature of

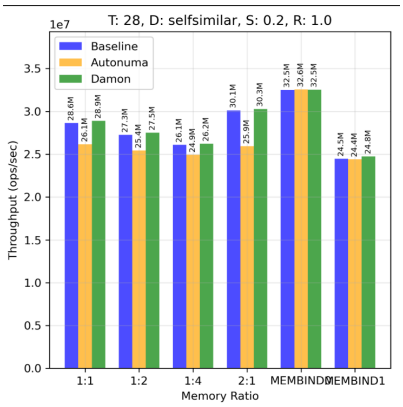
AutoNUMA migrations versus the asynchronous nature of DAMON sjp-damon-patch.

- **Migrations:** AutoNUMA migrated significantly more pages than DAMON (perhaps can be changed by tuning parameters). DAMON performed almost no demotions. **This is seen in all runs.**
- **Memory Ratio:** The expected throughput trend (`MEMBIND0 > 2:1 > 1:1 > 1:2 > 1:4 > MEMBIND1`) was generally respected except a couple of cases.
- **Threads:** If other things kept same, 28 threads gives an increase of 1.7-1.8x (expected since 28 threads are 1.75 times of 16, so almost linear increase).
- **Skew:** Compared to `UNIFORM` distribution, the skewed access pattern (skew 0.2) resulted in slightly better throughput for all schemes (perhaps due to lower cache misses, not sure).
- Throughput across tiering schemes is almost same for `MEMBIND0` and `MEMBIND1` cases, with slight improvements for DAMON (DAMON does not respect memory binding and still migrates hot pages to Node 0).

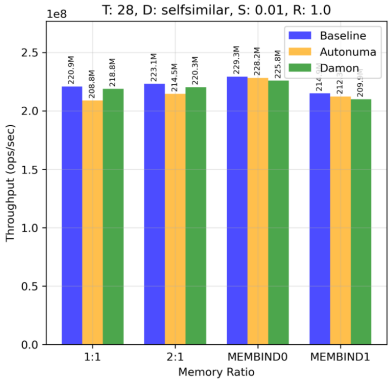
4.3.2 Small WSS Runs (Skew 0.01 & 0.001)

Realizing a smaller WSS that fits entirely on Node 0 would be a better test, runs were performed with `SELFSIMILAR` skews of 0.01 (WSS ≈ 16G) and 0.001 (WSS ≈ 4.9G), using only those memory ratios that would ensure the hotset can fit on Node 0. All files in `final_results/index_skew_1e-2_run` and `final_results/index_skew_1e-3_run`.

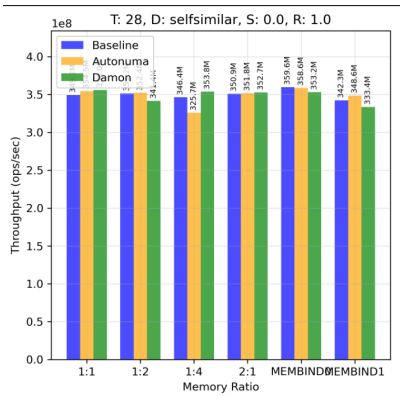
Comparing throughputs for different schemes for `SELFSIMILAR` with skews 0.2, 0.01 and 0.001 :



(a) PiBench Throughput (Skew 0.2)



(b) PiBench Throughput (Skew 0.01)



(c) PiBench Throughput (Skew 0.001)

Observations from smaller WSS runs:

- Throughput increased manifold with more skewed access (smaller WSS). Skew 0.2 vs Skew 0.01 : throughput becomes 7-8x. Skew 0.2 vs 0.001 : throughput becomes 12-13x.
- For skew 0.01 (Figure 10b), baseline performed the best while DAMON lost its slight advantage.
- For skew 0.001 (Figure 10c), there were no clear trends, and the performance gap between schemes was very low.

Full graphs for these runs are shown in Figures 11 and 12.

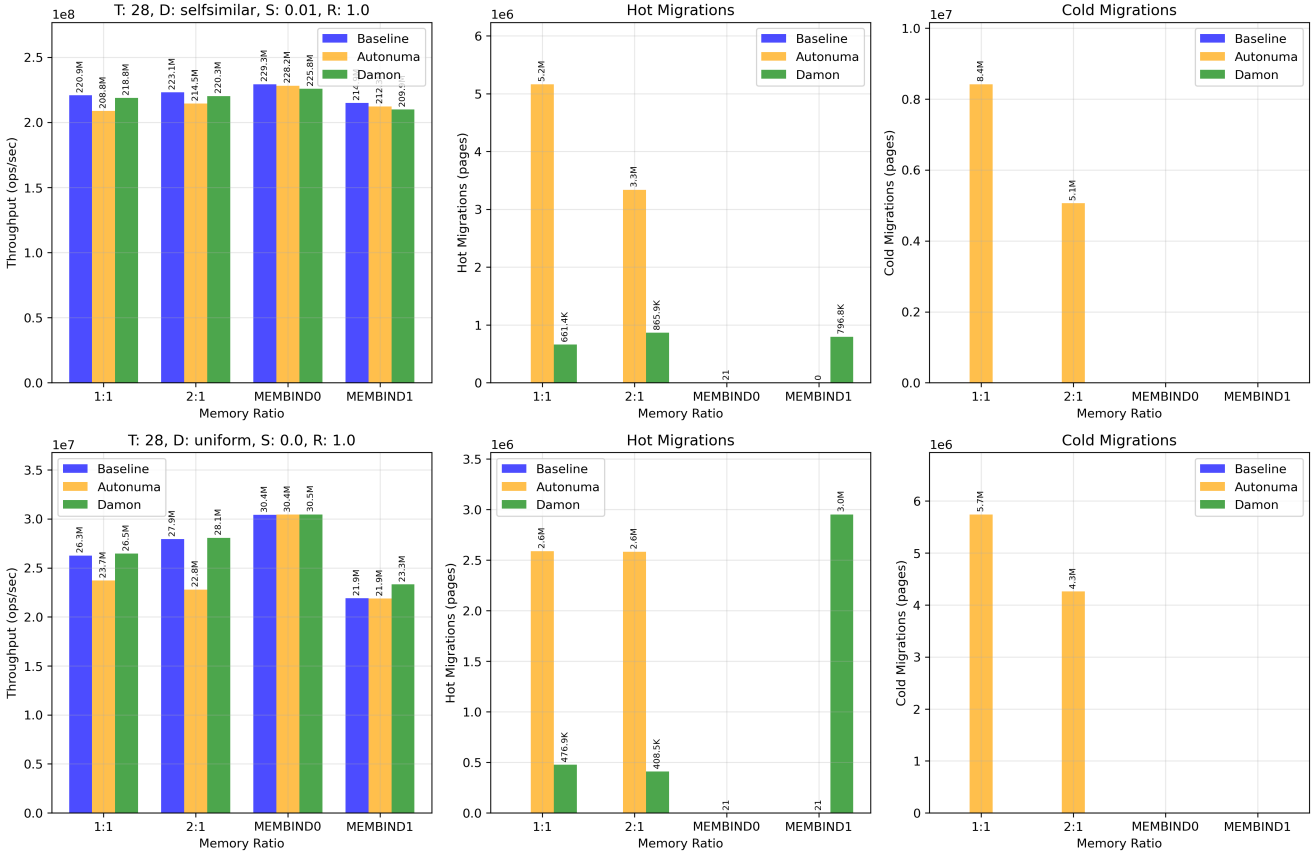


Figure 11: PiBench Full Run (Skew 0.01)

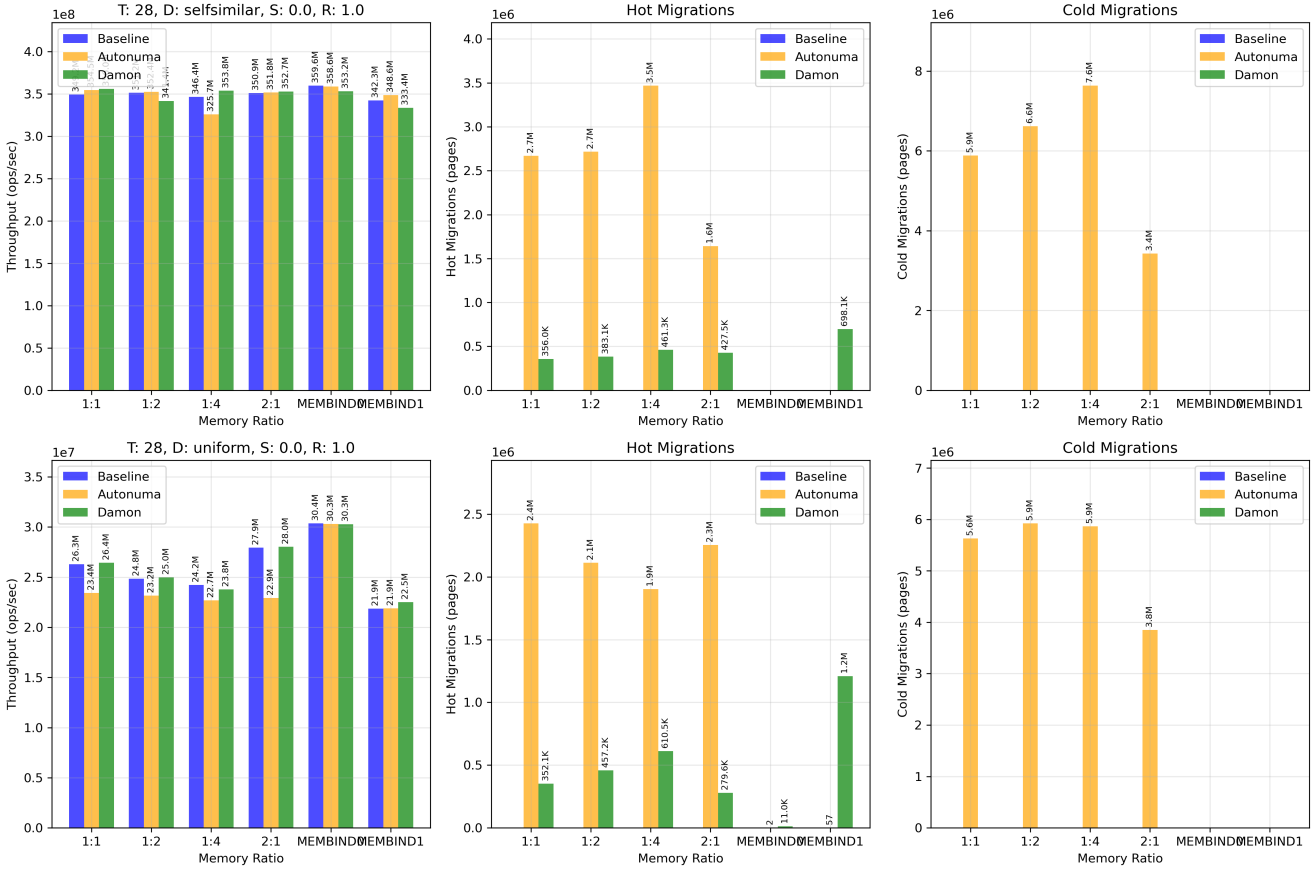


Figure 12: PiBench Full Run (Skew 0.001)

4.3.3 Partial Long Run Analysis (AutoNUMA)

A full run (all combinations of threads, read/write ratios, skews, and memory ratios) was started but interrupted after 2 days (since complete run would have taken 18 days). This completed all AutoNUMA configs for 16 and 28 threads.

All configuration parameters:

- **Threads:** [28, 16, 8, 4, 2, 1]
- **Read Ratios:** 0.0, 0.2, 0.5, 0.8, 1.0
- **Skews:** `SELFSIMILAR - [0.01, 0.1, 0.2, 0.5]`, `UNIFORM`
- **Memory Ratios:** 1:4, 1:2, 1:1, 2:1, `MEMBIND0`, `MEMBIND1`

All files in final_results/cxl_index_final_long_run_partial-autonuma_t16_28. Check out `individual_graphs`, `vary_read_ratio`, `vary_threads` and `vary_skew` folders for plots.

Observations from varying only one parameter at a time:

- **Varying Read Ratio:** General trend is of course, `MEMBIND0 >> shared > MEMBIND1`. For large WSS, throughput was similar across memory ratios and only discernible trend is `MEMBIND0 >> rest`. Also, there is not much difference across read ratios for a particular memory ratio.
- As WSS decreased (e.g., skew 0.2), the gap across read ratios increased, with read-heavy workloads (1.0) showing higher throughput and respecting the memory ratio trends (`2:1 > 1:1 > 1:2...`) more faithfully, while others may have some exceptions (including `MEMBIND0 > MEMBIND1` not being respected sometimes, probably due to very small WSS).
- Also, the gap between `MEMBIND0` and rest decreases with smaller WSS. I hope that's desirable.
- **Varying Threads:** Throughput for 28 threads was consistently much higher than 16 threads (though gap may vary). However, the number of migrations (hot and cold) was much higher for 16 threads than 28. Exceptions may arise in order of no of migrations for small WSS (`skew=0.01`) and non-read-heavy workloads (`read ratio<1.0`).
- **Varying Skew:** The throughput improvement from smaller WSS (more skew) was most visible for high read ratios. For write-heavy workloads, the gap was smaller, and trends were less consistent. Like check 28 threads, `read_ratio=0.2` and `read_ratio=0.0` cases : `skew=0.1,0.01` have lower throughput than larger skews.
- As mentioned, difference in throughput across mem ratios also becomes less visible for smaller read ratios (specially smaller skews, probably small WSS stops changing the throughput).

5 Related and Future Work

5.1 Related Work: SINLK Paper

The SINLK paper presents a method for tiered memory management with a focus on node granularity and promoting entire paths.

- **Method:**
  - It uses heuristics, such as giving preference to upper B-Tree nodes for fast memory.
  - Suggests using leaf nodes to track path access frequency. (Seems obvious upon reading but isn't at first read.)
  - Also single point of transition from fast to slow memory along a path is logical as well, and the methods used to achieve that are straightforward but make sense.
  - It employs asynchronous promotions/demotions and dynamic watermark adjustment.

  - Since acting at node granularity, it requires node metadata bookkeeping (part of frontend module). It requires application modification to include node metadata.
  - Apart from that, uses common techniques :
    * background asynchronous promotions/demotions
    * dynamic watermark adjustment
    * migration trigerred on breach of watermark and at some interval
    * halves access frequency periodically to give preference to recent accesses

- **Evaluation:**
  - Microbenchmark: Focusses on skewed accesses (90% requests to 5% keys)
  - Macrobenchmark : YCSB (with Zipfian)
  - Also mixes reads and updates.
  - Baseline : Weighted interleave NUMA allocation
  - Uses TPP, MEMTIS, Caption for comparisons (along with some other optimized index variants).
  - For microbenchmark, creates an ideal case to compare with all hot paths in fast memory initially (I think they can do this because it's a synthetic workload for microbenchmark so they can control the hot paths created and their allocation with their module).
  - Across cases, they show it to be having a better throughput, lower average and tail latencies, nice scalability with threads. Esp. good on read-heavy workloads (perhaps due to keeping a lock on writes while migrating).
  - Lastly, picks two traces with largest WSS from Alibaba Block Traces for real-world evaluation.

- **Interesting Ideas:** The paper includes a sensitivity analysis with a dynamic workload (shifting the hot region) and an analysis of the contribution of each individual factor both of which are interesting directions for future work.

5.2 Future Scope

Based on this project, future work could explore:

- Replicate the B+Tree (OptiQL) and MASIM benchmarks on the CXL-enabled server (srv9) using its local (Node 2) and remote CXL (Node 3) memory, comparing Baseline and AutoNUMA performance. This will validate the simulated remote-DRAM results against true CXL hardware.
- Trying different combinations of tiering schemes, as suggested in damo-issue-comment by SeongJae Park.
- Further tuning of DAMON command parameters (e.g., `damos_quota_space`) and AutoNUMA aggressiveness (e.g., `scan_size`, `scan_interval`).
- Implementing a dynamic workload benchmark, similar to the SINLK paper, to test the agility of each tiering scheme.
- numactl has introduced `--weighted-interleave` mode, however that is just restricted to allocation. DAMON is also bringing updates to have dynamic weights for weighted interleave, using migration actions to match the weights : damon-patch. They can be explored in future.

6 Conclusion

This project successfully set up a tiered memory testbed and evaluated the performance of AutoNUMA+TPP and DAMON for index-structure workloads. We found that the performance is highly dependent on the workload (WSS, read/write ratio) and the tiering scheme's design. DAMON generally provided more stable and sometimes improved throughput, likely due to its asynchronous, quota-managed migrations. AutoNUMA, while aggressive in promoting pages, sometimes suffered from performance degradation, possibly due to its synchronous nature. For highly skewed workloads with small, stable hotsets, the difference between schemes diminished.

Our findings indicate that DAMON, due to its asynchronous nature, often provides stable or improved throughput, whereas AutoNUMA's synchronous promotions can sometimes degrade performance.