# Tezos Context Lima Performance Audit

Tarides Irmin Team

2023-07-07

## Contents

# 1 Introduction

The Octez `tezos-context` package maintains an on-disk representation of Tezos blocks. It is implemented using Irmin and the purpose-built Irmin backend `irmin-pack`.

Performance of `tezos-context` is important for overall performance of Octez and previous improvments in `irmin-pack` have led to major performance improvements in Octez.

This report contains an audit of the `irmin-pack` storage backend used by Tezos Octez. We investigate the performance of individual sub-components as well as access patterns used by Tezos.

## 1.1 Overview of `irmin-pack`

Irmin is a content-addressable data store similar to Git. Data is stored in trees where nodes and leaves of the trees are identified by their cryptographic hash. At it's core Irmin is a store of content-addressed objects (nodes and leaves).

`irmin-pack` is a space-optimized Irmin backend inspired by Git packfiles that is currently used in Octez.

Conceptually `irmin-pack` stores content-addressed objects to an append-only file. A persistent datastructure called the index is used to map hashes to positions in the append-only file.

## 1.2 Methodology

If not noted all tests were run on StarLabs Starbook Mk VI with a AMD Ryzen 7 5800U and a PCIe-3 connected 960GiB SSD running Debain Linux 11 and the Linux Kernel version 6.1.0.

For replays a recording of 100000 blocks starting at block level 2981990 was used.

We use the fio tool to simulate disk access patterns and measure performance. A baseline, based on observed access patterns of `irmin-pack` is provided in the section Baseline `fio` job.

# 2 Sub-components

## 2.1 Index

The index is a persistent datastructure that maps the hash of an object to its offset in the append-only pack file.

Since Irmin version 3.0.0 objects in the pack file hold direct references to other objects in the pack file [irmin #1659] this removes the necessity to always query the index when traversing objects. This also allows a smaller index as now only top-level objects (i.e. commits) need to be in the index [irmin #1664]. This is called `minimal` indexing and has allowed considerable performance improvements for the Octez storage layer.

The index datastructure is split into two major parts: the log and the data. The log is a small and bounded structure that holds recently-added bindings. The log is kept in memory after initialization. The data part holds older bindings. When a certain number of bindings are added to the log, the bindings are merged with the bindings in the data part. The datastructure is similar to a two-level Log-structured merge-tree.

The default number of bindings to keep in the log before moving them to the data part in Octez and in `irmin-pack` is set to 2500000.

### 2.1.1 Archive node

Tezos archive nodes hold the full history since genesis. When using minimal indexing references to at most 2500000 commits/blocks are kept in the log of the index. In July 2022 the Tezos block chain reached that block height. So since at least July 2022 the index of archive nodes will also have a `data` part.

Archive nodes that were bootstrapped before the new minimal indexing strategy was adopted have many more entries in the index:

```
dune exec audits/index/count_bindings.exe -- inputs/archive-node-index/

Number of bindings in Index: 2036584177
```

Archive nodes bootstrapped before the new minimal indexing strategy was adopted use around 90GiB of space just for the index structure. With the new minimal indexing the size of a freshly bootstrapped archive node is about 2.4MiB.

### 2.1.2 Rolling and full node

Tezos rolling and full nodes keep a default of 6 cycles which corresponds to about 98304 blocks (16834 blocks per cycle) or, in Irmin terminology, to about 98304 commits.

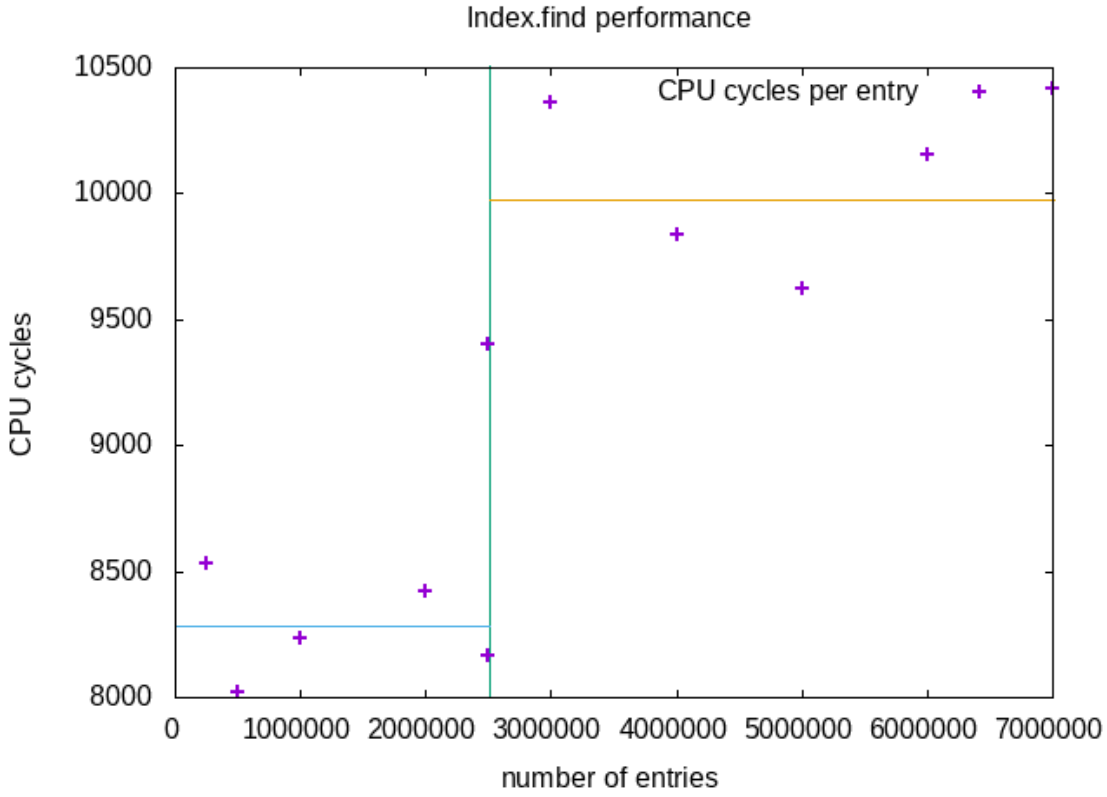
If for every commit a single index entry is maintained, rolling and full nodes will by default never create a data part and every index lookup performed is an in-memory operation.

However, currently entries in the index are never removed. So size of the index is not a function of the history mode and settings of the node, but the time since it was bootstrapped.

### 2.1.3 Find Performance

We measure the performance of index lookups by creating an empty index structure with same parameters as used in Tezos Octez (key size of 30 bytes, value size of 13 bytes and log size of 2500000), adding `c` random bindings and performing `c` lookups for random bindings. We use the LexiFi/landmarks library to measure performance in CPU cycles (as well as system time).

| count | cpu time | sys time | cpu time per entry |
|---|---|---|---|
| 250000 | 2132773826.000000 | 1.126439 | 8531.0953 |
| 500000 | 4009158441.000000 | 2.119049 | 8018.3169 |
| 1000000 | 8235106179.000000 | 4.351689 | 8235.1062 |
| 2000000 | 16838605030.000000 | 8.893822 | 8419.3025 |
| 2499999 | 20421445765.000000 | 10.791509 | 8168.5816 |
| 2500001 | 23511451504.000000 | 12.446721 | 9404.5768 |
| 3000000 | 31077192851.000000 | 16.442429 | 10359.064 |
| 4000000 | 39358430251.000000 | 20.823590 | 9839.6076 |
| 5000000 | 48122118368.000000 | 25.448949 | 9624.4237 |
| 6000000 | 60941841080.000000 | 32.247097 | 10156.974 |
| 7000000 | 72898690458.000000 | 38.564382 | 10414.099 |




We note a sharp increase in CPU cycles needed to lookup an entry when the number of bindings jumps over the log size (2500000). The lookup performance stays relatively constant at a higher level for up to 7000000 entries.

### 2.1.4 Conclusion

With the currently implemented minimal indexing scheme no performance issues are expected when using the default Tezos Octez configurations. No considerable performance degradation is expected up to at least block level 7000000.

For nodes running in history modes `rolling` and `full` a mechanism should be implemented to remove entries for commits that were garbage collected, this would allow the index to remain a purely in-memory structure for such nodes.

Archive nodes that were bootstrapped using non-minimal indexing have a very large index structure. For better disk-usage it is recommended to re-bootstrap these nodes using minimal indexing.

## 2.2 Dict

Irmin stores values in a tree where paths are sequences of strings. In order to de-duplicate commonly occurring path segments, some path segments are stored in an auxiliary persistent structure called the `Dict`.

The `Dict` maintains a mapping from path segments (strings) to integer identifiers. In the Irmin tree the integer identifiers can be used instead of the string path segment. As integer identifiers are in general smaller than the string path segments this results in a more compact on-disk representation.

In `irmin-pack` the `Dict` is limited to 100000 entries that were added on a first-come-first-serve basis. The `Dict` is currently full.

### 2.2.1 Occurrences of Dict elements in store

We count the number of occurrences of path segments that appear in the `Dict` in the store of the Tezos block 3081990.

It seems like the some common path segments are de-duplicated a considerable amount of times:

| Path segment | Occurrences in store |
|---|---|
| 4a1cf11667fa0165eac9963333b883a80bcfdfebde09b79bfc740680e986bab6 | 108903 |
| 053f610929e2b6ea458c54dfd8b29716d379c13f5c8fd82d5c793a9e31271743 | 90893 |
| 00d0158265571a474bc6ed02547db51416ab2228327e66332117ea7b587aca94 | 13912 |
| 1ffdaf1cd7574b72f933a9d5e102143f3e4d761a6e51b4019ed821b7b99b097a | 2313 |
| 04034e4a6228fdb92e8978fb85d9c2d1f79501b0c509f24b0f1eede3ca7cb234 | 1611 |
| 10f21b2eacdf858cf9824d29e9c0d09bf666d3d900fbc54b6438f67e63831d4d | 1157 |
| 2966fdb0cb953d94e959dcee2b2c3238c42bf0d1e0991a5a51609059aaa04080 | 890 |
| 1f33bf814d191cc602888479ef371d13082f19718f63737e685cc76110e323d9 | 813 |
| 02e5f95f2c3a3ccfa5ce71d0f11ad70f4746b8e0f3fe7bcecc63dbc8cfba71d1 | 731 |
| 438c52065d4605460b12d1b9446876a1c922b416103a20d44e994a9fd2b8ed07 | 477 |
| 00642bcad8681caf0f45f195cc1483f8366f155d83e272c9ff93fe3840a61dcb | 396 |
| 4001852857ca1c5dad1c1275f766fc5208e63c48ba0289127591ffef3c440d53 | 388 |
| 27e1640238d07d569852b3e4fe784f5adce0e6649673ea587aa7389d72b855af | 381 |

However we also note that most entries in the `Dict` do not occur in the store. We count 96394 `Dict` entries that do not occur in the store (96%).

Furthermore, we observe that most entries in the `Dict` are hashes. Commonly used short keywords like `total_bytes`, `missed_endorsement` or `index` are not `Dict` entries.

### 2.2.2 Increasing `Dict` capacity

In order to evaluate potential performance improvement we run benchmarks with following changes to the `Dict`:

- Increase capacity to 500000 entries.
- Remove limitations on the number of entries that the `Dict` can hold (see metanivek/irmin-92d910b). Path segments are always added to the dictionary before writing.
- Don't use `Dict` for any newly written content

| | 3.7.1 | 500k-dict | infinite-dict | ignore-dict |
|---|---|---|---|---|
| CPU time elapsed | 104m07s | 109m42s (105%) | 130m36s (125%) | 101m39s (98%) |
| total bytes read | 66.466G | 64.976G (98%) | 60.747G (91%) | 67.489G (102%) |
| total bytes written | 46.360G | 41.671G (90%) | 37.903G (82%) | 46.891G (101%) |
| max memory usage (bytes) | 0.394G | 0.508G (129%) | 2.534G (643%) | 0.396G (100%) |

When increasing the capacity of the `Dict`, the amount of bytes read and written decreases as improved usage of the `Dict` results in more compact representations. Memory usage increases as expected. Unfortunately it seems that overall performance degrades.

When disabling the `Dict` for newly written content we see that slightly more content is written and read. This small change again indicates that the `Dict` is underused. However, we see that disabling the `Dict`, even at the cost of causing more I/O, increases performance slightly.

The performance decrease when increasing `Dict` capacity could be due to the in-memory Dict (a `Hashtbl`) being taken to it's limits. A more efficient in-memory datastructure might offset this and result in performance increase.

### 2.2.3 Conclusion

We conclude that the `Dict` is underused. This observation has been previously made (see mirage/irmin#1807). An increased usage of the `Dict` could lead to a considerably more compact on-disk representation and potential performance improvements. However, we note that the in-memory `Dict` seems to need considerable performance optimizations before we see any overall performance improvments.

Note that existing `Dict` entries can not be removed and it is not clear how to improve usage of the `Dict` without rewriting existing data.

One suggestion would be to make the `Dict` local to individual chunks. Since version 3.5.0 `irmin-pack` supports splitting on-disk data into multiple smaller chunks (see

mirage/irmin#2115). Instead of having one single global `Dict` we could implement a `Dict` per chunk. This might allow the chunk-local ~Dict~s to perform much better as they hold entries to data that has been accessed recently. This design seems to also fit well with how the garbage collector works.

It might also be worth to reconsider the first-come-first-serve strategy. An improvement might be to only add a `Dict` entry for path segments that appear at least `n` times. This would prevent the addition of `Dict` entries for single-shot accessed path-segments.

# 3 Audits

## 3.1 IO Activity

In order to measure real disk IO accesses we add some instrumentation to irmin-pack that allows us to measure how many bytes are read/written to individual files in how many system calls. In addition we also trace calls to IO reads and writes using the landmarks library.

| | Read | Write |
|---|---|---|
| Bytes per block (mean) | 591394.118 (56%) | 463528.962 (44%) |
| Number of system calls per block (mean) | 9645.864 | 4.061 |
| Bytes per system call (mean) | 61.310642 | 114141.58 |
| Relative time | 13.6% | 0.4% |
| Relative time in I/O | 96.5% | 3.5% |

We observe:

- Total I/O activity in bytes consists of about 56% reads and 44% writes. However, reads take 96.5% of the time [1].
- Reads are performed in many very small chunks (in mean 61 bytes per system call).
- Total time spend doing I/O is only 14%. This is an upper-bound for the effect on overall performance by improving I/O.

As read performance dominates the overall performance we concentrate on it in the following.

### 3.1.1 Baseline `fio` job

Based on the observations we can create a baseline `fio` job description that simulates the read access pattern of `irmin-pack`:

[1] We measure this by tracing the `Irmin_pack_unix.Io.Util.really_read` and `Irmin_pack_unix.Io.Util.really_write` functions

```
[global]
rw=randread
filename=/home/adatario/dev/tclpa/.git/annex/objects/gx/17/SHA256E-s3691765475--13300581

[job1]
ioengine=psync
rw=randread
blocksize_range=50-100
size=591394B
loops=100

fio baseline-reads.ini

job1: (g=0): rw=randread, bs=(R) 50B-100B, (W) 50B-100B, (T) 50B-100B, ioengine=psync, i
fio-3.35
Starting 1 process

job1: (groupid=0, jobs=1): err= 0: pid=153129: Thu Jun 15 10:14:00 2023
  read: IOPS=328k, BW=21.0MiB/s (22.1MB/s)(56.4MiB/2681msec)
    clat (nsec): min=210, max=3742.4k, avg=2833.00, stdev=20598.76
     lat (nsec): min=230, max=3743.1k, avg=2858.85, stdev=20605.58
    clat percentiles (nsec):
     |  1.00th=[   211],  5.00th=[   221], 10.00th=[   221], 20.00th=[   221],
     | 30.00th=[   231], 40.00th=[   231], 50.00th=[   231], 60.00th=[   231],
     | 70.00th=[   241], 80.00th=[   402], 90.00th=[   410], 95.00th=[   676],
     | 99.00th=[150528], 99.50th=[158720], 99.90th=[191488], 99.95th=[230400],
     | 99.99th=[329728]
   bw (  KiB/s): min=20846, max=21922, per=99.73%, avg=21483.80, stdev=449.08, samples=5
   iops        : min=317386, max=333862, avg=327161.20, stdev=6851.62, samples=5
  lat (nsec)   : 250=70.87%, 500=23.40%, 750=1.39%, 1000=1.64%
  lat (usec)   : 2=0.93%, 4=0.08%, 10=0.06%, 20=0.01%, 50=0.01%
  lat (usec)   : 250=1.59%, 500=0.03%, 750=0.01%, 1000=0.01%
  lat (msec)   : 4=0.01%
  cpu          : usr=12.13%, sys=9.74%, ctx=14307, majf=0, minf=10
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwts: total=879400,0,0,0 short=0,0,0,0 dropped=0,0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
   READ: bw=21.0MiB/s (22.1MB/s), 21.0MiB/s-21.0MiB/s (22.1MB/s-22.1MB/s), io=56.4MiB (5

Disk stats (read/write):
```

```
    dm-1: ios=14145/0, merge=0/0, ticks=2116/0, in_queue=2116, util=96.36%, aggrios=1430
    dm-0: ios=14301/0, merge=0/0, ticks=2132/0, in_queue=2132, util=95.57%, aggrios=1430
  nvme0n1: ios=14301/0, merge=0/0, ticks=1539/0, in_queue=1539, util=95.57%
```

We observe a read band-width of 21.0MiB/s which seems to match our observations.

## 3.2 Compactness of on-disk representation

Having a compact on-disk representation is not only good for disk usage but can also improve overall performance as data can be loaded into memory closer to the processor faster if it is smaller.

We analyze the compactness of the on-disk representation of `irmin-pack` by compressing with the Zstandard compression algorithm. Compact representation have a low compression ratio, whereas less compact representations may admit a more compact representation (for example by compressing with Zstandard).

| Store | Uncompressed | Compressed | Compression Ratio |
|---|---|---|---|
| Level 2981990 | 5108531200 | 3265930512 | 1.5641886 |
| Level 3081990 | 32111902720 | 10332988078 | 3.1077073 |
| Single suffix from level 2981990 | 5101987840 | 3261700639 | 1.5642109 |
| Single suffix from level 3081990 | 3633868800 | 940228651 | 3.8648778 |

There seems to be room to improve the compactness of the on-disk representation.

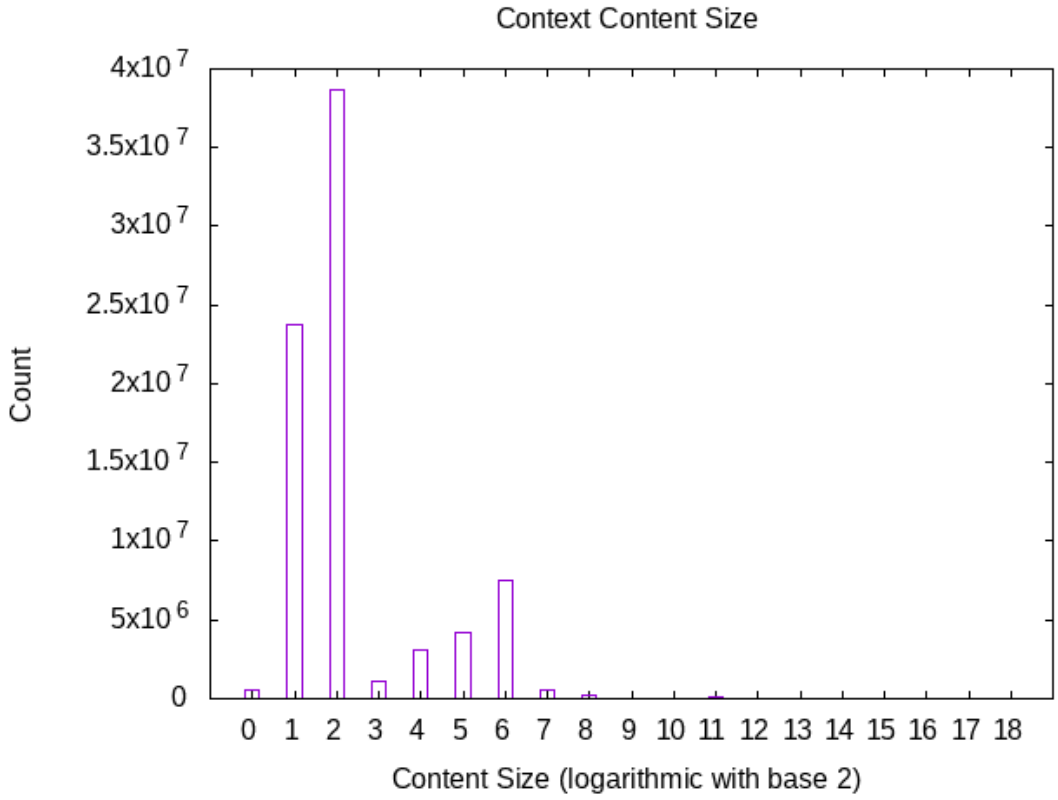
We can not explain the difference in compression rates between the stores of level 2981990 and 3081990.

## 3.3 Context structure and access patterns

### 3.3.1 Content Size distribution

| Content Size (logarithmic with base 2) | Count | Percentage of Content |
|---|---|---|
| 0 | 596661 | 0.74748116 |
| 1 | 23722650 | 29.719110 |
| 2 | 38698770 | 48.480798 |
| 3 | 1125580 | 1.4100969 |
| 4 | 3131048 | 3.9224944 |
| 5 | 4194650 | 5.2549469 |
| 6 | 7477058 | 9.3670611 |
| 7 | 532486 | 0.66708442 |

| Content Size (logarithmic with base 2) | Count | Percentage of Content |
|---|---|---|
| 8 | 194262 | 0.24336631 |
| 9 | 35410 | 0.044360714 |
| 10 | 35600 | 0.044598741 |
| 11 | 71535 | 0.089617161 |
| 12 | 4289 | 5.3731461e-3 |
| 13 | 2583 | 3.2359143e-3 |
| 14 | 260 | 3.2572114e-4 |
| 15 | 9 | 1.1274963e-5 |
| 16 | 1 | 1.2527736e-6 |
| 17 | 3 | 3.7583209e-6 |
| 18 | 7 | 8.7694154e-6 |




We observe that the size of content in the Tezos store is very small.

### 3.3.2 Access Patterns

We perform some audits in order to understand what paths in the Tezos Context are accessed how frequently.

1. Read Operations over 100000 blocks

   We analyze the 10000 blocks starting at block level 2981990 and record what paths are accessed by read operations such as `Context.find_tree`, `Context.mem` or `Context.find`:

| Path | Read operations * |
|---|---|
| `contracts/global_counter` | 2121152 |
| `big_maps/index/347453/total_bytes` | 614589 |
| `big_maps/index/347455/total_bytes` | 614589 |
| `big_maps/index/347456/total_bytes` | 614589 |
| `big_maps/index/103258/total_bytes` | 79037 |
| `big_maps/index/149768/total_bytes` | 46549 |
| `endorsement_branch` | 40004 |
| `grand_parent_branch` | 40004 |
| `v1/constants` | 40004 |
| `v1/cycle_eras` | 40004 |
| `version` | 40004 |
| `big_maps/index/149787/total_bytes` | 32835 |
| `big_maps/index/149771/total_bytes` | 29009 |
| `big_maps/index/149772/total_bytes` | 29009 |
| `first_level_of_protocol` | 20002 |
| `votes/current_period` | 20000 |
| `big_maps/index/103260/total_bytes` | 19991 |
| `cycle/558/random_seed` | 16183 |
| `cycle/558/selected_stake_distribution` | 16183 |

   We observe that certain paths are hit very often. The nodes on the commonly accessed paths should be kept in the `irmin-pack` LRU cache and should not incur any disk I/O.

2. Operations in a single block

   We also analyze a single block (block level 2981990), breaking down into categories of `mem`, `read`, and `write` operations. Here is the top 15, sorted by `read` and then `mem`.

| Path | `mem` | `read` | `write` |
|---|---|---|---|
| `contracts/index/0001f46d...63a0/balance` | 444 | 888 | 444 |

| Path | mem | read | write |
|---|---|---|---|
| contracts/index/01e12d94...da00/used_bytes | 222 | 443 | 222 |
| contracts/index/0001f46d...63a0/delegate | 0 | 443 | 0 |
| contracts/global_counter | 225 | 224 | 225 |
| contracts/index/01e12d94...da00/len/storage | 222 | 222 | 222 |
| contracts/index/0001f46d...63a0/counter | 222 | 222 | 222 |
| contracts/index/01e12d94...da00/paid_bytes | 222 | 221 | 222 |
| contracts/index/01e12d94...da00/balance | 0 | 221 | 0 |
| big_maps/index/347456/total_bytes | 111 | 110 | 111 |
| big_maps/index/347455/total_bytes | 111 | 110 | 111 |
| big_maps/index/347453/total_bytes | 111 | 110 | 111 |
| big_maps/index/149776/contents/af067ef3...5bad/data | 6 | 6 | 1 |
| contracts/index/00006027...b77a/balance | 3 | 6 | 3 |
| big_maps/index/149776/contents/af067ef3...5bad/len | 0 | 6 | 1 |
| contracts/index/0002358c...8636/delegate_desactivation | 0 | 5 | 2 |
| ... | | | |
| Total | 2363 | 3892 | 3180 |

We note a symmetry in number of `mem` and `write` operations to certain paths.

## 3.4 CPU boundness

We run a replay of 100000 Tezos blocks with three different CPU frequencies:

- 3.40GHz: Run on an Equinix Metal `c3.small.x86` machine with an Intel® Xeon® E-2278G CPU with 8 cores running at 3.40 GHz.
- 2.5GHz: Run on an Equinix Metal `m3.large.x86` machine with an AMD EPYC 7502P CPU with 32 cores running at 2.5 GHz.
- 1.5GHz: Run on an Equinix Metal `m3.large.x86` machine with an AMD EPYC 7502P CPU with 32 cores at 1.5 GHz. The CPU frequency was set using `cpupower frequency-set 1.5GHz`.

| CPU Frequency | 3.40GHz | 2.5GHz | 1.5GHz |
|---|---|---|---|
| CPU time elapsed | 73m27s | 103m26s 141% | 194m37s 265% |
| TZ-transactions per sec | 1043.919 | 741.288 71% | 393.969 38% |
| TZ-operations per sec | 6818.645 | 4841.928 71% | 2573.316 38% |

This seems to indicate that `irmin-pack` is CPU-bound. Improving CPU efficiency should directly improve overall performance.

# 4 Potential Improvements

## 4.1 Disable OS page cache

Operating systems implement their own cache mechanism for disk access. This can cause additional CPU cycles which are unnecessary as `irmin-pack.unix` implements it's own cache.

We can disable the operating system to cache blocks by using the `fadvise` system call with `FADV_NOREUSE`. This will prevent the operating system from maintaining blocks in cache, thus saving CPU cycles.

We can tell `fio` to do this with the `fadvise_hint` option:

```
[global]
rw=randread
filename=/home/adatario/dev/tclpa/.git/annex/objects/gx/17/SHA256E-s3691765475--13300581

[job1]
ioengine=psync
rw=randread
blocksize_range=50-100
size=591394B
loops=100
fadvise_hint=noreuse
```

```
fio fadvise-noreuse.ini
```

```
job1: (g=0): rw=randread, bs=(R) 50B-100B, (W) 50B-100B, (T) 50B-100B, ioengine=psync, i
fio-3.35
Starting 1 process

job1: (groupid=0, jobs=1): err= 0: pid=156889: Thu Jun 15 10:55:44 2023
  read: IOPS=432k, BW=27.7MiB/s (29.1MB/s)(56.4MiB/2035msec)
    clat (nsec): min=210, max=3652.8k, avg=2107.02, stdev=18068.27
     lat (nsec): min=230, max=3653.4k, avg=2131.96, stdev=18076.02
    clat percentiles (nsec):
     |  1.00th=[   211],  5.00th=[   221], 10.00th=[   221], 20.00th=[   221],
     | 30.00th=[   231], 40.00th=[   231], 50.00th=[   231], 60.00th=[   231],
     | 70.00th=[   241], 80.00th=[   402], 90.00th=[   410], 95.00th=[   442],
     | 99.00th=[121344], 99.50th=[152576], 99.90th=[224256], 99.95th=[254976],
     | 99.99th=[346112]
   bw (  KiB/s): min=27723, max=28868, per=99.72%, avg=28298.25, stdev=476.32, samples=4
   iops        : min=422174, max=439618, avg=430933.00, stdev=7326.27, samples=4
  lat (nsec)   : 250=71.97%, 500=24.01%, 750=1.26%, 1000=0.95%
```

```
  lat (usec)   : 2=0.51%, 4=0.07%, 10=0.06%, 20=0.01%, 50=0.02%
  lat (usec)   : 100=0.02%, 250=1.06%, 500=0.05%, 750=0.01%, 1000=0.01%
  lat (msec)   : 4=0.01%
  cpu          : usr=12.00%, sys=15.34%, ctx=10101, majf=0, minf=11
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwts: total=879400,0,0,0 short=0,0,0,0 dropped=0,0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
   READ: bw=27.7MiB/s (29.1MB/s), 27.7MiB/s-27.7MiB/s (29.1MB/s-29.1MB/s), io=56.4MiB (59

Disk stats (read/write):
    dm-1: ios=11489/0, merge=0/0, ticks=1868/0, in_queue=1868, util=94.89%, aggrios=12303
    dm-0: ios=12303/0, merge=0/0, ticks=2016/0, in_queue=2016, util=94.15%, aggrios=12303
  nvme0n1: ios=12303/0, merge=0/0, ticks=1629/0, in_queue=1628, util=94.15%
```

We observe a read bandwidth of 27.7MiB/s. This seems to be a considerable performance improvement (around 30%) that can be implemented fairly easily.

Unfortunately we cannot observe any performance improvement when testing these changes on the Tezos Context replay benchmarks:

| | 3.7.1 | `POSIX_FADV_NOREUSE` | `POSIX_FADV_RANDOM` |
|---|---|---|---|
| CPU time elapsed | 104m07s | 104m50s (101%) | 103m51s (100%) |
| TZ-transactions per sec | 736.349 | 731.387 (99%) | 738.298 (100%) |
| TZ-operations per sec | 4809.664 | 4777.256 (99%) | 4822.395 (100%) |

## 4.2 Inline small objects

We note that the size of most content in the Tezos Context Store is very small (see Content Size distribution). These small pieces of content are stored in individual leaf nodes of the tree with their own hash, incurring a large overhead. An optimization would be to inline the small content into the parent node. Preliminary work towards this has already been done (see mirage/irmin#884).

Concretely we can imagine a scheme where all content that is smaller than the size of the hash as well as small `Inodes` are inlined.

In order to approximate potential performance gains we use a modified `irmin-pack` and `bench/irmin-pack/tree.ml` bench to approximate time we could save if we inlined "small" contents objects into their parent inode.

The benchmark is run against ~10k blocks. Total time for the benchmark is 38s.

Counts loaded in `Pack_store.find_in_pack_file`.

| Type | Count |
|---|---|
| Commit | 12 |
| Inode | 1478220 |
| Contents | 78700 |

Counts and percentages when split at a 64 byte boundary. Objects less than 64 bytes might be good candidates for inlining since this is the point where the size of the hash (32 bytes) is equal to the size of the actual content.

| Type | < 64B (%) | 64B+ | Total |
|---|---|---|---|
| Inode | 1292769 (87.5%) | 185451 | 1478220 |
| Contents | 9068 (11.5%) | 69632 | 78700 |

Duration of key sections in that function, reading from disk and decoding into in-memory objects.

| Type | Read (us) | Decode (us) | Total (us) |
|---|---|---|---|
| Commit | 73.478 | 77.477 | 150.955 |
| Inode | 1516744.706 | 1721268.547 | 3238013.253 |
| Contents | 115519.722 | 16060.434 | 131580.156 |

Lets assume we can inline inodes or contents objects that are less than 64 bytes and eliminate the read time but maintain the decode time.

| Type | Read, < 64B (us) | Read, 64B+ (us) | Total (us) |
|---|---|---|---|
| Inode | 1305857.425 | 210887.281 | 1516744.706 |
| Contents | 8836.786 | 106682.936 | 115519.722 |
| | 1314694.211 | 317570.217 | 1632264.428 |

We could theoretically avoid 1314694.211us out of 1632264.428us read time, which is 80.5% savings of time spent reading. The entire bench takes 38s, so this would represent a 3.5% savings in total time.

For completeness, here is the breakdown of decode times on the 64 byte boundary.

| Type | Decode, < 64B (us) | Decode, 64B+ (us) | Total (us) |
|---|---|---|---|
| Inode | 866570.158 | 854698.389 | 1721268.547 |
| Contents | 2179.074 | 13881.36 | 16060.434 |

## 4.3 Optimize tree descent

Access to content in Irmin requires descending a path. Every node on the path needs to be de-referenced, accessed from disk and unpacked individually. The cost for accessing a piece of content is a function of the path size. It might make sense to optimize the tree descent.

### 4.3.1 Path Cache

We note that read accesses to the Tezos Context Store are focused on few paths (see Access Patterns).

The existing LRU cache in `irmin-pack` should prevent re-fetching these paths from the disk, increasing performance considerably. This is done by maintaining a mapping from position in pack file to tree object. When fetching a path, we need to sequentially iterate over the cache until we reach the object. This should be mostly in-memory operations as tree objects are cached. Still this incurs a lot of CPU cycles.

A potential optimization would be use a cache that maintains a mapping from path to tree object. When fetching a path that is cached, only a single lookup would be required to get the final tree object.

## 4.4 Optimize Tezos Storage Functors

Tezos usage of `irmin-pack` goes through the `tezos_context` library but also various storage functors implemented in the `tezos-protocol-015-PtLimaPt.environment`. This is where commonly appearing keywords like `index` or `total_bytes` are defined.

It may be possible that the way these storage functors use `irmin-pack` (indirectly via the `tezos_context` library) could be improved. In particular, it seems like many small files are created with long, nested paths. It may be more efficient to create larger files with shorter paths.

A concrete proposal would be to add additional Irmin types for the various storage abstractions offered by the storage functors. Irmin allows variable content types. Currently the only type used is `bytes`. Instead one could define more complex types such as:

```
module Contents = struct
  module StringMap = Map.Make(String)

  type t =
    | Bytes of bytes
    | KVMap of string StringMap.t

end
```

The type `KVMap` would be serialized to a single larger content object in the Irmin store. This might improve locality of data as well as increase the size of content objects that would improve I/O bandwidth (see Batching Reads). This proposal can be seen as a form of explicity inlining (see Inline small objects) that uses the existing Irmin API.

It would also be interesting to investigate the observed symmetry between `mem` and `write` access as observed in section Operations in a single block.

## 4.5 Decrease system call overhead

Read performance seems to be bad because of the large amount of small reads that are performed with individual system calls. The system call overhead seems to be significant and reducing it should increase performance.

Proposal such as Inline small objects would also help in increasing the size of reads and quite possibly will improve read performance.

### 4.5.1 Batching Reads

Performance could be improved by batching read operations into larger blocks. For example if we use 4KiB blocks:

```
[global]
rw=randread
filename=/home/adatario/dev/tclpa/.git/annex/objects/gx/17/SHA256E-s3691765475--13300581

[job1]
ioengine=psync
rw=randread
blocksize=4KiB
size=591394B
loops=100
```

```
fio batching-reads.ini
```

```
job1: (g=0): rw=randread, bs=(R) 4000B-4000B, (W) 4000B-4000B, (T) 4000B-4000B, ioengine=
fio-3.35
Starting 1 process

job1: (groupid=0, jobs=1): err= 0: pid=10702: Fri Jun 16 10:32:29 2023
  read: IOPS=8941, BW=34.1MiB/s (35.8MB/s)(56.1MiB/1644msec)
    clat (nsec): min=310, max=3684.8k, avg=110624.46, stdev=91545.42
     lat (nsec): min=330, max=3685.4k, avg=110715.69, stdev=91573.99
    clat percentiles (nsec):
     |  1.00th=[   422],  5.00th=[   628], 10.00th=[   820], 20.00th=[  1256],
     | 30.00th=[  3472], 40.00th=[122368], 50.00th=[134144], 60.00th=[142336],
     | 70.00th=[152576], 80.00th=[164864], 90.00th=[199680], 95.00th=[236544],
     | 99.00th=[317440], 99.50th=[415744], 99.90th=[667648], 99.95th=[700416],
     | 99.99th=[905216]
   bw (  KiB/s): min=32390, max=37875, per=100.00%, avg=34973.67, stdev=2756.26, samples=
   iops        : min= 8292, max= 9696, avg=8953.33, stdev=705.52, samples=3
  lat (nsec)   : 500=2.02%, 750=6.08%, 1000=6.39%
  lat (usec)   : 2=11.84%, 4=4.03%, 10=1.27%, 20=0.33%, 50=0.01%
  lat (usec)   : 100=0.06%, 250=64.49%, 500=3.18%, 750=0.27%, 1000=0.02%
  lat (msec)   : 4=0.01%
  cpu          : usr=1.58%, sys=7.00%, ctx=10010, majf=0, minf=11
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwts: total=14700,0,0,0 short=0,0,0,0 dropped=0,0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
   READ: bw=34.1MiB/s (35.8MB/s), 34.1MiB/s-34.1MiB/s (35.8MB/s-35.8MB/s), io=56.1MiB (58

Disk stats (read/write):
    dm-1: ios=8537/0, merge=0/0, ticks=1280/0, in_queue=1280, util=93.46%, aggrios=10000/
    dm-0: ios=10000/0, merge=0/0, ticks=1496/0, in_queue=1496, util=93.96%, aggrios=10000
  nvme0n1: ios=10000/0, merge=0/0, ticks=1149/0, in_queue=1149, util=93.90%
```

We observe a read bandwidth of 34.1MiB/s (a 60% performance improvement to the baseline). However, it is not clear how reads could be batched without major reorganization of the on-disk representation.

## 4.6 CPU efficiency

As noted in section CPU boundness improving CPU efficiency of `irmin-pack` may have a great effect on overall performance of the Tezos Context Store.

Some potential improvements are described in the following.

### 4.6.1 Cache datastructure

The currently implemented LRU cache in `irmin-pack` is implemented using doubly-linked-lists which requires non-negligible datastructure manipulations even while only accessing objects in the cache.

Other cache replacement policies and cache datastructures exist that might decrease the number of required CPU cycles. Futhermore they may provide better cache hit rates that will also improve overall performance.

## 4.7 Multicore

In order to simulate the potential performance improvement of using multiple cores we use a `fio` job similar to the baseline but instead use `N` cores that in total read the same amount of bytes:

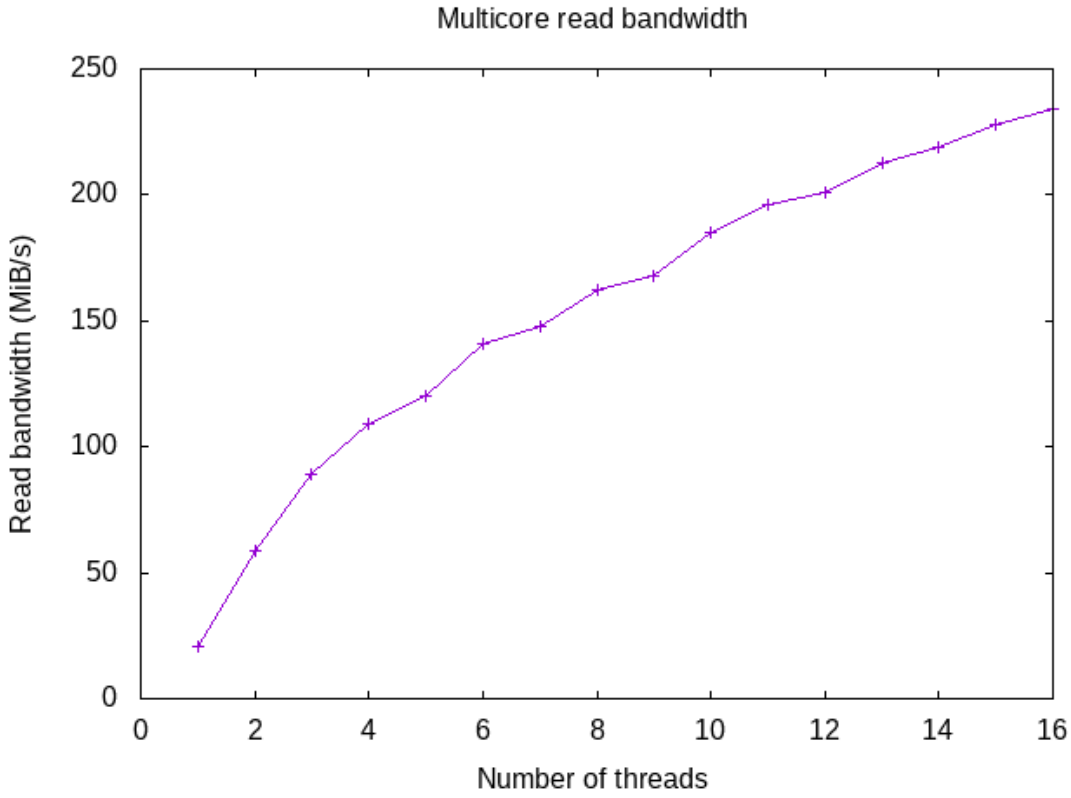
```
[global]
rw=randread
filename=/home/adatario/dev/tclpa/.git/annex/objects/gx/17/SHA256E-s3691765475--13300581
loops=100
group_reporting
thread

[job1]
ioengine=psync
rw=randread
blocksize_range=50-100
size=591394 / N
numjobs=N
```

Using the options `thread` and `numjobs` we create `N` threads that randomly read from the file.

We observe following read bandwidths:

| Number of Threads | Read bandwidth (MiB/s) |
|---|---|
| 1 | 21 |
| 2 | 58.8 |
| 3 | 89.4 |
| 4 | 109 |
| 5 | 120 |

| Number of Threads | Read bandwidth (MiB/s) |
|---|---|
| 6 | 141 |
| 7 | 148 |
| 8 | 162 |
| 9 | 168 |
| 10 | 185 |
| 11 | 196 |
| 12 | 201 |
| 13 | 213 |
| 14 | 219 |
| 15 | 228 |
| 16 | 234 |




Increasing the number of threads/CPU cores utilized seems to lead to considerable performance improvements.

Note however, that the we simulate `N` independent threads. In `irmin-pack`, and Irmin in general, reads may need to be sequential as they descend a tree by de-referencing nodes individually.

## 4.8 Modern hardware and asynchronous I/O

Modern PCIe-attached solid-state drives (SSDs) offer high throughput and large capacity at low cost. They are exposed to the system using the same block-based APIs as traditional disks or SSDs attached via serial buses. However, in order to utilize the full performance of modern hardware, the way I/O operations are performed needs to be changed.

In order to illustrate the capabilities of modern hardware we run `fio` using the Linux $\text{io}_{\text{uring}}$ API. This allows asynchronous I/O operations. We also increase size of the blocks read from a few bytes to 64KiB. This increases latency for individual reads, but allows much higher bandwidth.

```
[global]
rw=randread
filename=/home/adatario/dev/tclpa/.git/annex/objects/gx/17/SHA256E-s3691765475--13300581
loops=100
group_reporting
thread

[job1]
ioengine=io_uring
iodepth=16
rw=randread
blocksize=64KiB
size=100MiB
numjobs=8
```

```
fio read-io_uring.ini
```

```
job1: (g=0): rw=randread, bs=(R) 62.5KiB-62.5KiB, (W) 62.5KiB-62.5KiB, (T) 62.5KiB-62.5K
...
fio-3.33
Starting 8 threads

job1: (groupid=0, jobs=8): err= 0: pid=44365: Fri May 19 14:25:25 2023
  read: IOPS=147k, BW=8955MiB/s (9390MB/s)(74.5GiB/8517msec)
    slat (nsec): min=290, max=9700.4k, avg=9392.25, stdev=73806.67
    clat (nsec): min=130, max=23784k, avg=767189.99, stdev=1005905.48
     lat (usec): min=4, max=23786, avg=776.58, stdev=1007.13
    clat percentiles (usec):
     |  1.00th=[   17],  5.00th=[   30], 10.00th=[   43], 20.00th=[   62],
     | 30.00th=[   83], 40.00th=[  125], 50.00th=[  215], 60.00th=[  400],
     | 70.00th=[  914], 80.00th=[ 1795], 90.00th=[ 2278], 95.00th=[ 2606],
     | 99.00th=[ 3884], 99.50th=[ 4490], 99.90th=[ 6390], 99.95th=[ 7439],
```

```
     | 99.99th=[10028]
   bw (  MiB/s): min= 7688, max=10172, per=100.00%, avg=9047.06, stdev=83.12, samples=129
   iops        : min=125968, max=166674, avg=148226.72, stdev=1361.83, samples=129
  lat (nsec)   : 250=0.01%, 500=0.01%, 750=0.01%, 1000=0.01%
  lat (usec)   : 2=0.01%, 4=0.02%, 10=0.10%, 20=1.72%, 50=11.80%
  lat (usec)   : 100=21.71%, 250=17.36%, 500=10.21%, 750=4.71%, 1000=3.36%
  lat (msec)   : 2=12.41%, 4=15.67%, 10=0.88%, 20=0.01%, 50=0.01%
  cpu          : usr=2.68%, sys=14.62%, ctx=559563, majf=0, minf=0
  IO depths    : 1=0.1%, 2=0.1%, 4=0.3%, 8=0.5%, 16=99.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=99.9%, 8=0.0%, 16=0.1%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwts: total=1249600,0,0,0 short=0,0,0,0 dropped=0,0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=16

Run status group 0 (all jobs):
   READ: bw=8955MiB/s (9390MB/s), 8955MiB/s-8955MiB/s (9390MB/s-9390MB/s), io=74.5GiB (80

Disk stats (read/write):
    dm-1: ios=345735/95, merge=0/0, ticks=597292/4, in_queue=597296, util=98.72%, aggrios
    dm-0: ios=349972/95, merge=0/0, ticks=599656/4, in_queue=599660, util=98.56%, aggrios
  nvme0n1: ios=349972/87, merge=0/8, ticks=542294/6, in_queue=542303, util=97.77%
```

We observe a read bandwidth of about 8955MiB/s this is an improvement of more than 40000% compared to the baseline. Note that this is pure I/O read bandwidth without any data management.

On-going research is investigating how the performance of such hardware can be used in database systems (see LeanStore: In-Memory Data Management Beyond Main Memory). We note that the main ingredients for doing this in the OCaml ecosystem are present or are being actively worked on: OCaml 5 with Multicore support and Algebraic Effects as well as the eio library. The question seems to be: What Are We Waiting For? Let's Use Coroutines for Asynchronous I/O to Hide I/O Latencies and Maximize the Read Bandwidth!

# 5 Conclusion

In this report we have identified storage performance bottlenecks of Irmin and `irmin-pack` as used by Octez and propose potential improvements.

A selection of the different scenarios considered, with expected impact on performances is provided in the following table:

| Description | I/O | Overall | Effort (FTE) | On-disk format |
|---|---|---|---|---|
| Improve `Index` usage | - | 0% | - | No change |
| Improve `Dict` usage | 10-20% | -20% | 8-16w | Potential change |
| Disable OS page cache | 30% | 0-3% | 2-4w | No change |
| Inline small objects | 19.5% | 0-10% | 16-32w | Change |
| Path Cache | - | 5% | 8-16w | No change |
| More compact on-disk representation | 300% | 10% | 16-32w | Change |
| More efficient cache datastructure | - | | 8-16w | No change |
| Optimize Tezos Storage Functors | - | | 16-32w | No change |
| Batching reads | 60% | 5% | 16-32w | Change |
| Multicore I/O [2] | 700% | 12% | 32-64w | No change |
| Asynchronous I/O with modern APIs | 40000% | | 64w+ | Change |

Some details to the columns:

**I/O** Potential improvement in I/O performance measured and estimated with `fio`.

**Overall** Potential improvement in overall performance of the Tezos Context store based on performance of current implementation and structure of Context Store. The performance improvements are indicative and purely theoretical. Experiments on dict and OS page cache have shown that some results may be counter-intuitive.

**Effort** Estimated effort to implement performance.

**On-disk format** Indication if an on-disk format change is needed.

Note that performance improvements may not be additive and individual improvements may have unexpected results on the effect of other improvements.

We conclude with following recommendations:

1. Implement Inlining of small objects: This seems to be an obvious optimization. Even if the estimated performance gains from decreased I/O might be low we can expect more performance gains by doing larger reads (see Batching Reads), improved CPU efficiency while decoding content (see CPU efficiency) as well a more shallow tree (see Optimize tree descent).
2. Investigate CPU usage: We observe that the Tezos Context Storage seems to be CPU bound (see CPU boundness). Currently we have little insight into what are the bottlenecks and what areas should be optimized for best overall performance improvement. We propose using the OCaml 5.1 `Runtime_events` library for observing running applications (see ocaml/ocaml#11474). This approach can be used for also observing overall Octez code and identifying performance bottlenecks beyond the Tezos Context Store.

[2] Only considering I/O. We expect even larger performance improvements when CPU intensive operations are parallelized over multiple cores.

3. Explore Multicore and Asynchronous APIs: Modern hardware, APIs such as `io_uring` and multicore capabilities in OCaml 5 allow massively improved performance for storage intensive applications. However, they require fundamental redesign of the storage backends (see Asynchronous I/O). To remain competitive with regards to performance in the medium to longer-term, it is imperative to explore such ideas now.