CS636 Project: SlimFast

180301 - hrugved wath

https://github.com/Hrugved/CS636_project_slimfast

NOTE: please change `JAVA_HOME` in `msetup` as per your system.

1 Introduction

FastTrack is a state-of-the-art data race detection tool, which greatly outperforms former tools like lockset, eraser, etc in terms of accuracy, precision, and performance. Although FastTrack has good performance in terms of speed, it suffers from memory overheads sufficient enough to hinders its ability to scale to very large programs. One major reason for this is having a lot of metadata redundancy as can be seen from Fig:**1**.

One obvious way to solve this would be to have a global set of metadata objects. This approach will be the most space-optimized solution as it set will disallow any duplication, but it will affect time overhead drastically and hence is impractical.

SlimFast, on other hand, reduces the redundancy by

- exploting FastTrack's rules to eliminate metadata duplication on writes.
- using caches per thread to store recent metadata for possible reuse on reads.
- optimizations for quicker search for metadata like emptying of caches on releases of locks and links to get access to next update of reading vector clocks, etc.

2 Project Files

- tools:
  - `src/tools/fasttrack` : original fasttrack implementation: **FT2**
  - `src/tools/fasttrack_r` : fasttrack with support for Redundancy tracking: **FT2_r**
  - `src/tools/slimfast_w` : slimfast[write-only]: **SF_w**
  - `src/tools/slimfast_w_ep` : slimfast[write-only,EpochPair]: **SF_w_ep**
  - `src/tools/slimfast` : slimfast: **SF**
- scripts
  - `script_redundancy.sh` : outputs[`output_script_redundancy.txt`] redundancy ratio by running FT2_r.
  - `script_correctness.sh` : outputs[`output_script_correctness.txt`] errorTotal and distinctErrorTotal of FT and SF.
  - `script_space_optimisation.sh` : outputs[`output_script_space_optimisation.txt`] space optimisation ratio for SF_w,SF_w_ep and SF wrt to FT.
  - `script_cacheSize.sh` : outputs[`output_script_cacheSize.txt`] average space optimisation ratio for cacheSize 5,10,...50.
  - `script_time.sh` : outputs[`output_script_time.txt`] average time overhead of SF normalized to FT2.

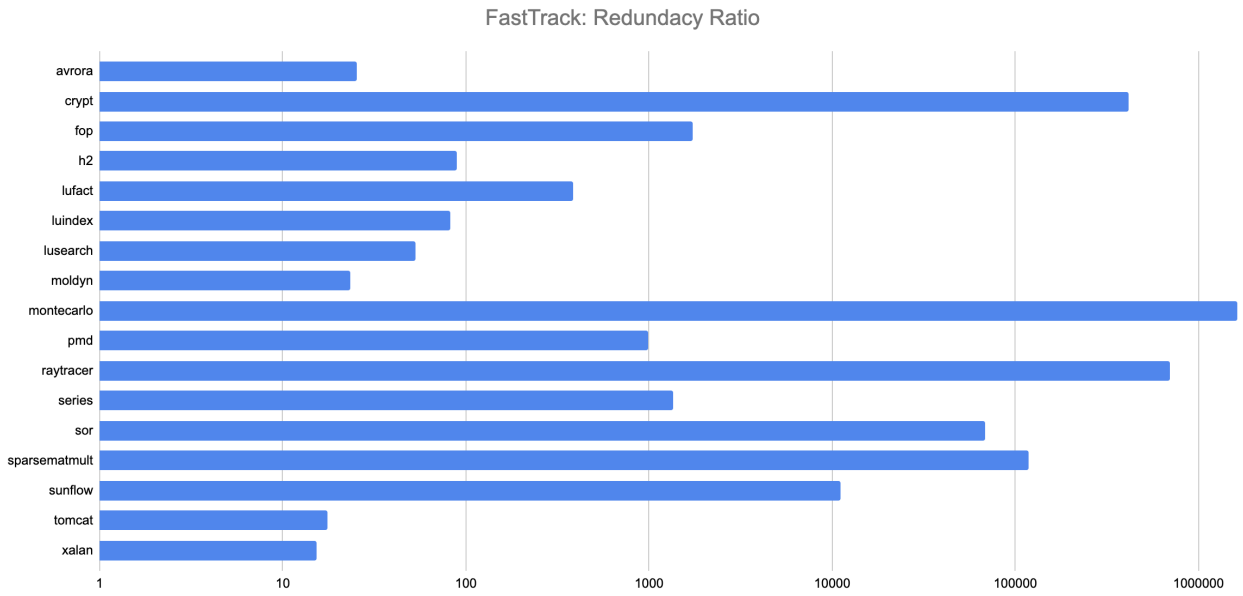

Fig: 1
The redundancy ratio is the normal FT metadata overhead compared to the best case(no duplication of metadata)

3 Implementation

- **FT2_r**:
  - Atomic integers are used as counters for tracking total fasttrack metadata objects like `writeEpochs`, `readEpochs`,etc.(`FastTrackTool_r.java`)
  - Concurrent sets are used for tracking fasttrack's unique metadata objects.(`FastTrackTool_r.java`)
  - On each update/creation of any metadata obj, the relevant counter is updated and a copy is also inserted into relevent set.(FastTrackTool_r.java)
  - The redundancy ratio is then calculated by dividing sum of all counters by sum of sizes of all sets.(`FastTrackTool_r.java`)
  - A hashcode method is added to `VectorClock`(`src/tools/util/VectorClock.java`) for usage by `Set<VectorClock>`.
  - To Run:
    - run: cd benchmarks/avrora && ./Test -tool=FT2_r
    - in output xml log, look for entry/tool/Redundancy
    - Alternatively, can run `script_redundancy.sh` which runs on all benchamarks
- **SF**:
  - To Run:
    - run: cd benchmarks/avrora && ./Test -tool=SF -cacheSize=15
    - cacheSize sets the size of fixed arrays `EpochPairCache` and `EpochPlusVCCache` in `SFThreadState.java`.
    - in output xml log, look for entry/system/memUsed for memory usage.
    - Alternatively, can run `script_space_optimisation.sh` which runs on all benchamarks
  - `EpochPair.java`:
    - `EpochPair` is ShadowVar.

    - contains two epochs, one for each read(R) and write(W).
  - EpochPlusVC.java:
    - EpochPlusVC is also ShadowVar and extends EpochPair.
    - Since it extends EpochPair, it has one epoch for write(W), one read(R) epoch which is set to Epoch.READ_SHARED.
    - Additionally has one read VectorClock(RVC).
    - Also contains Array(next) of EpochPlusVC and related methods used for Optimizing EpochPlusVC accesses(Slimfast:IV,C,2).
  - SFThreadState.java:
    - ShadowThread object for threads.
    - Contains thread-specific metadata vectorclock(VC) and epoch(E).
    - Contains other optimisation-related metadata and functions:
      - *Optimization for Writes* (Slimfast:IV,A):
        EpochPair(currentWriteEpoch) equivalent to W_t.
      - *Optimization for reads involving EpochPairs* (Slimfast:IV,B):
        Array(EpochPairCache) equivalent to S_t used as cache for storing recent EpochPairs.
        Whenever a new/updated EpochPair is needed, SlimFast calls getEpochPair(), which first calls getEpochPairFromCache() to check if required EpochPair is already contained in EpochPairCache and returns it, else if not present then calls generateAndInsertNewEpochPairIntoCache() to create new required EpochPair and inserts into EpochPairCache and returns it. (Slimfast:IV,B,2)
      - *Optimization for* EpochPair *to* EpochPlusVC \textit{Inflations} (Slimfast:IV,C,1):
        Array(EpochPlusVCCache) equivalent to Q_t used as cache for storing recent EpochPlusVCs.
        Whenever a new/updated EpochPlusVC is needed for inflating from EpochPair, SlimFast calls getEpochPlusVC(), which first calls getEpochPlusVCFromCache() to check if required EpochPlusVC is already contained in EpochPlusVCCache and returns it, else if not present then calls generateAndInsertNewEpochPlusVCIntoCache() to create new required EpochPlusVC and inserts into EpochPlusVCCache and returns it. (Slimfast:IV,B,2)
    - EpochPairCache and EpochPlusVCCache are both implemented as fixed-sized array for performance reasons.
    - Contains refresh() which resets all optimisation related metadata.(Slimfast:IV,B,1)
  - SlimFastTool.java:
    - Minimal changes are made to implement the Slimfast operational rules(Slimfast:Fig 4)
    - All metadata updates involving SFVarState are immutable. Hence, RR's putShadow() method is which Updates the shadow location, returning true if atomic test-and-set succeeds.
    - read() loops calling try_read() till it succeeds(returns true). try_read() process the read event and uses putShadow() to immutably update the ShadowVar. If it fails, read() uses RR's putOriginalShadow() method which updates the state field, right before event was dispatched and then call try_read() again. This continues till try_read() succeeds.
    - write() and try_write() works the same way as their read counterparts.
    - Whenerver SFThreadState VC is incremented (ex.in methods like maxEpochAndVC(),etc.), its refresh() method is called.(Slimfast:IV,B,1)

- **SF_w and SF_w_ep**:
  - both are impemented the same way as SF with unwanted optimisations removed.

4 Results

In terms of correctness, SF gives similar results to that of FT2 (validated using `script_correctness.sh`). All benchmarks were run on FT2, SF_w, Sf_w_ep, and SF with cacheSize of 15. The space usage was computed from the entry/system/**memUsed** field in output log XML (Note: Slimfast authors used a different method for this using GC every 100ms and averaging them). All runs were averaged across 5 iterations and memory usage was normalized to that of FT2.
`script_space_optimisation.sh` was run to automate all this. This was performed on mac machine consisting of 2.3 GHz 8-Core Intel Core i9 with 16GB of RAM.

The average space-optimization was around 1.4x with a max of 3.18x(*crypt* with cacheSize=20). The results for the same are summarized in Fig:**2**. The average time overhead is 0.87x with a max of 1.12x on *avrora*. Fig:**3** summarises the time overhead of SF.
Some Insights from results obtained:

- Slimfast generally performs better on benchmarks with high redundancy ratios like *crypt* and *Montecarlo*.
- Although *raytracer* has a high redundancy ratio, its performance is worse. One reason might be the less concurrency (smaller number of threads and less shared memory accesses).
- Slimfast performs worse than FastTrack when the workload is less like in the case of lufact. This is because Slimfast inherently uses more storage in terms of metadata per thread and variable. Hence for less workload, the costs outweigh its benefits.
- Which optimization configuration works better depends on the target program. Like in the case of *Jython*, EpochPair and EpochPlusVC optimizations do more harm than benefit. A possible reason is the high percentage of write redundancy than reads.
- I experimented with cacheSize parameter by running `script_cacheSize.sh` and results are summarized in Fig:**4**. The impact of the cacheSize parameter isn't much, but it performs slightly better for middle cacheSizes 15-35. Since cacheSize 15 gives the best performance, it is used as default.

5 Futher Ideas

Since the performance of SlimFast depends highly on the target program, we could run profiling tools first to get an idea about the program(no. of threads, level of shared-memory access, the share of reads and writes in total redundancy, etc.) and then fine-tune(selecting cache size and turning on-off some optimizations, etc.) SlimFast accordingly and run it. It will for sure incur more time-overhead, but for very large programs, it will help to solve the main goal of SlimFast of reducing memory-overhead and scaling to large programs.

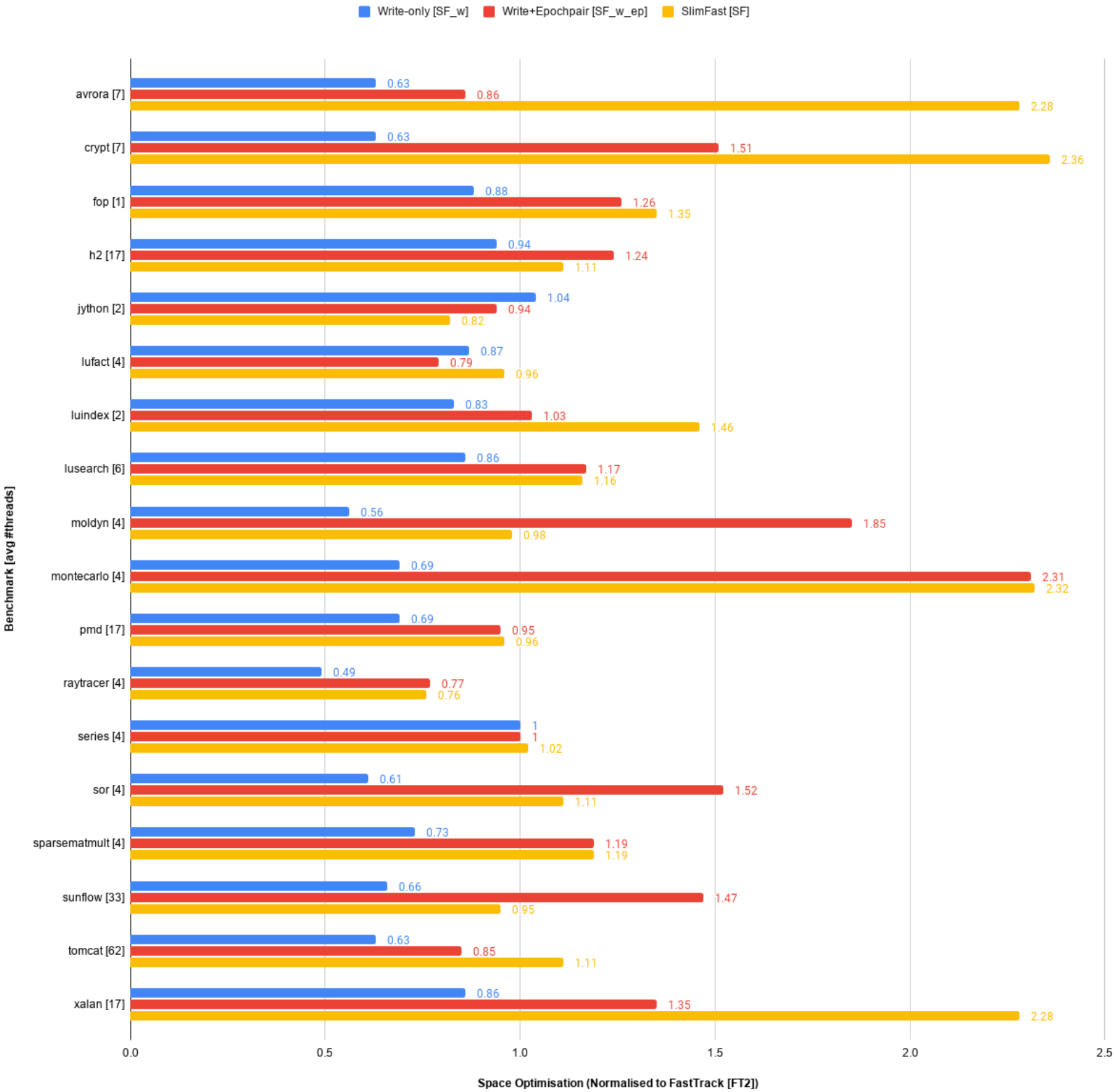

Fig: 2
Data is taken from output of `script\_space\_optimisation.sh`

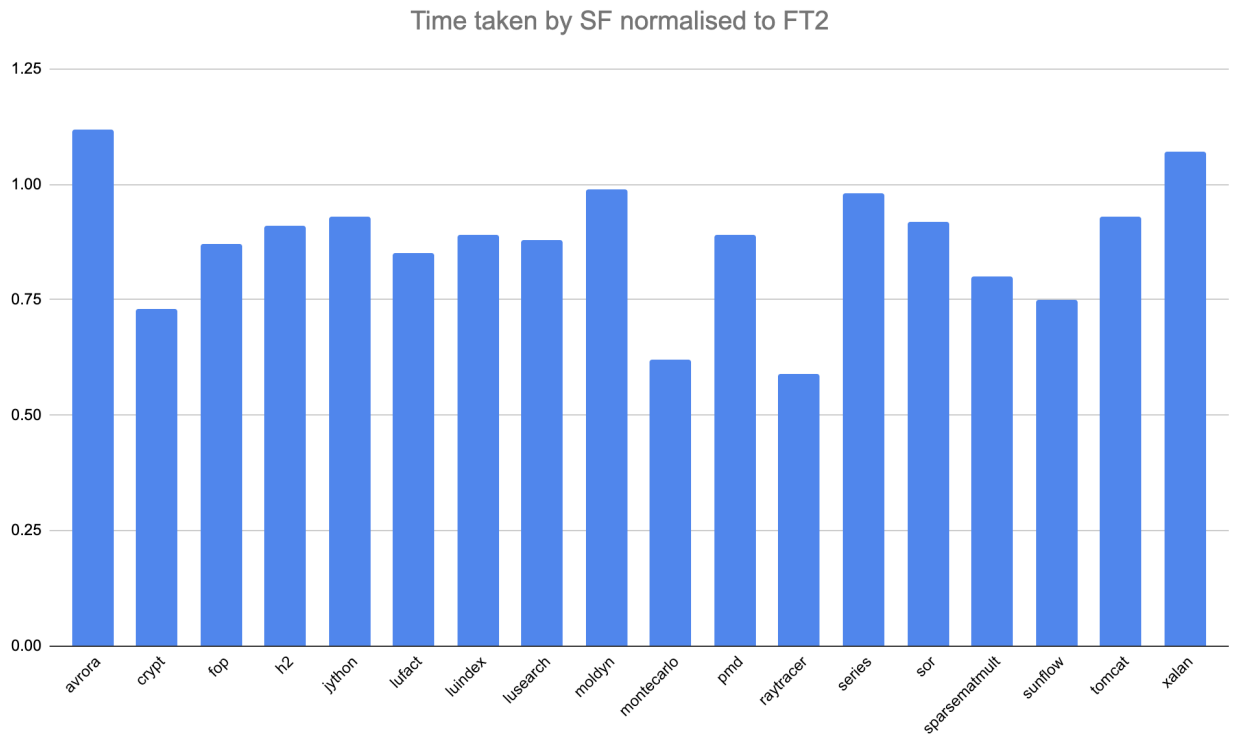

Fig: 3
Data is taken from output of `script_time.sh`

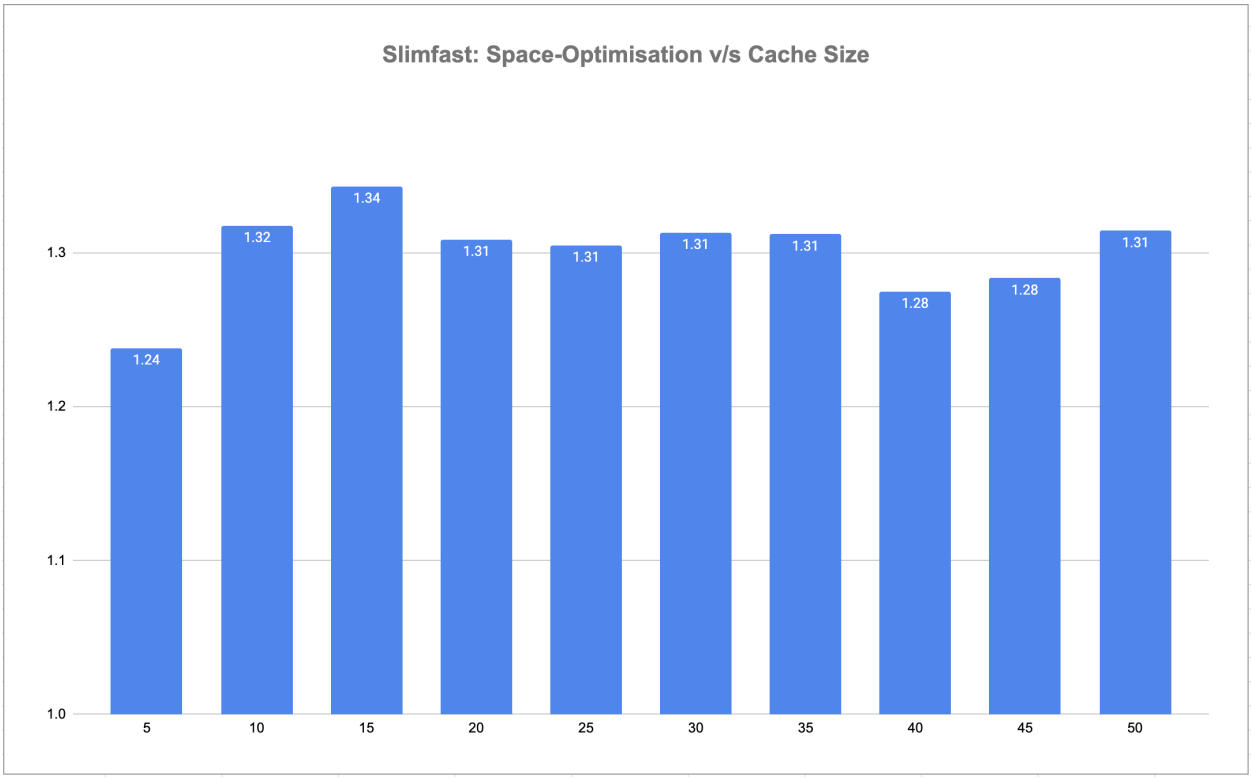

Fig: 4
Data is taken from output of `script_cacheSize.sh`