Scratch Management and Scalable Flushing

Robert C. Bell | CSIRO IMT Scientific Computing

10 October 2018

Outline

- Resource sharing
- Scratch (temporary) shared filesystems
  - use and abuse, management
- Flushing – removal of “old’ files
  - Old CSIRO algorithm
  - New scalable algorithm

Resource Sharing

- HPC centres set up for ‘big’ problems
  - Long, large, many
  - Justification to funders
- In practice, resources are shared
  - Intermittent large jobs
  - Continual smaller jobs
  - Mixed workloads
  - Aim for maximum utilisation
  - Compute – “easy” – batch systems
  - Storage – harder
- Address temporary storage and its management

Temporary storage

- Scratch (temporary) shared filesystems
  - Often the biggest and highest performing FS on HPC systems
  - Provides temporary storage for the duration of jobs and sessions
  - Needs to hold some data for longer to support development, workflows, pre- and post-processing, analysis, etc.
- Shared – need policies for management
  - Many HPC sites have policies
  - Many sites do not have workable implementations
  - Need to deal with contention between users
  - Continuing from lessons supposedly learnt in kindergarten!

Goals

- Our approach is to manage scratch areas to try to maximize the use for the benefit of users, by allowing them to
  - gain access to large amounts of storage on demand (“campaign storage”)
  - store files in the scratch space for longer than individual jobs and sessions
  - reduce the copying to and from scratch
  - have large quotas (not Balkanized!)
  - have automatic clean-up of old files
- Need to prevent scratch from filling

Space management - flushing

- Old CSIRO SC
  - scripts and program to implement policy
  - triggered when usage reaches a threshold (typically 95%)
  - file audit, then sort, and delete oldest until second threshold reached (typically 90%), or 7 days (rare)
  - uses mod time and access time – problem for FSes that don't do atime
  - also removes empty directories

New implementation of flushing at CSIRO

- Scratch areas shared between clusters and SGI UV 3000 (NFS)
- Old script slurped entire FS metadata into memory
- Scalability question – how quickly could the flushing respond under pressure?
- Known problems of metatdata performance on distributed FSes (NFS, Lustre)
- Servers (best place to run flushing script) did not have enough memory
- Problem with access time (upon which flushing is partly based) not being updated

- New policy – warning about inaccurate access times

doi:10.1145/2814553

Article development led by queue.acm.org

Probabilistic algorithms are all around us. Not only are they acceptable, some programmers actually seek out chances to use them.

BY TYLER MCMULLEN

It Probably Works

PROBABILISTIC ALGORITHMS EXIST to solve problems that are either impossible or unrealistic (too expensive or too time consuming, and so on) to solve precisely. In an ideal world, you would never actually need to use probabilistic algorithms. To programmers who are not familiar with them, the idea can be positively nerve-wracking: "How do I know it will actually work? What if it is inexplicably wrong? How can I debug it? Maybe we should just punt on this problem, or buy a whole lot more servers ..."

However, to those who either deeply understand probability theory or at least have used and observed the behavior of probabilistic algorithms in large-scale production environments, these algorithms are not only acceptable, but it is also worth seeking out opportunities to use them. This is because they can help solve problems and create systems that are less expensive, more predictable, and can do things that could not be done otherwise.

The world is probabilistic—or at least it is way too complex for us to model 100% accurately. Networks, and especially the Internet, are also probabilistic. Whether or not a particular packet will get where it needs to go is not something we can have complete knowledge of. Knowing which paths it will take through the many networks it will traverse in getting from one side of the world to the other is even less certain. Systems that run across asynchronous/lossy networks are necessarily probabilistic. Like it or not, probability is all around us.

Uncertainty is the core of the problem that uninitiated developers have with probabilistic algorithms, leading them to believe the algorithms may work poorly or be impossible to debug. That idea, however, is not correct. Probabilistic algorithms incorporate an element of randomness (they are also referred to as *randomized algorithms*), which is quantifiable. It can be analyzed, allowing strong predictions about the algorithm's behavior to be made. There is, in fact, very little uncertainty about how the algorithms will behave.

This article addresses two types of probabilistic algorithms: those that explicitly introduce randomness through a `rand()` call, and those that convert input data into a uniform distribution to achieve a similar effect. I will discuss specific algorithms that fall into these classes, including LogLog (and its better-known successor HyperLogLog) and Bimodal Multicast, and the problems they attempt to solve.

In a third type of probabilistic algorithm, randomness is intrinsic to the data. This applies to problems such as GPS tracking, self-driving cars, sensor networks, and missile-guidance systems. For example, sensor networks often try to make a statement about the network as a whole despite having only partial data. In the case of self-driving cars and missile-guidance systems, the data is in continuous flux. By the time a program has processed the data, the current state has changed. So it is already wrong and that must be taken into account. This article does not discuss this type of algorithm, but for more information on the set of problems in this category and the algorithms to solve them, it is helpful to read up on Kalman filters[1] and estimation algorithms in general.

Count-Distinct

The count-distinct problem involves counting the number of unique items in a stream that may contain duplicates. More formally: find the cardinality of a multiset. Many problems fall into this category. For example: How many unique IPs have connected to a server over the past hour? How many different words are in a huge corpus of text? How many different users visited a popular Web site in a day? Or even, how many unique URLs have been requested through an HTTP proxy?

The naive solution to this problem is easy. You could easily have written the code in your head before getting to the end of the previous sentence. Here is some Python code that illustrates the solution:

```
...
def count_distinct(stream):
    seen = set()
    for item in stream:
        seen.add(item)
    return len(seen)
...
```

As with many problems, the difficulties with count-distinct come with scale. It may be easy and cheap to count a thousand or even a million unique users, IPs, URLs, or words, but how about 100 million? Still not too bad. How about 100 million per server across thousands of servers? Now it is starting to get interesting.

The problem now is: perform a count-distinct across 1,000 servers, which could be as many as 100 million unique items each, and find the cardinality of the union of their result. In other words: distributed count-distinct. Let's consider a naive solution.

Each server could keep a set of "seen" items, just as in the previous code example. When each finishes its individual count-distinct run, it sends the entire contents of that "seen" set to a central server, which can then take the union of all the sets and return the cardinality of the combined set.

New implementation of flushing at CSIRO

- Inspiration was gained from the article “It Probably Works”, McMullen, Tyler, CACM, vol 58, no 11, Nov 2015, pp 50-54.
- This article shows that we often do not need exact solutions, and can provide approximate solutions at far lower “cost”.
- Don’t need to flush files in exact age order – a batch of old files will do.

New implementation of flushing at CSIRO

- Part A: scanning
- Eliminate the sort (to save processing time). Assign files from the target filesystem as they are scanned to bins or buckets, based on the youngest of access and modify time.
- Define a starting time as the time of commencement of service, or (after the first flush) the age of the youngest file last flushed.
- Define 14 days (site policy) before present as our finishing time.
- Set up say 101 buckets – bucket 0 for files older than our starting time (some may have escaped!), and a linear mapping of times to buckets.
- As the scan proceeds, save just the path name into each bucket, with one bucket for files and one for directories.

Scanning for candidates for flushing

Oldest Newest

0 1 2 3 4 | 5 6 7 8 9 … 100 | cut-off

Keep these | May not need to create these …

0: <date0
1: date0–date1
2: date1–date2
3: date2–date3
4: date3–date4
…
100: date99–cut-off (e.g. 14 days)

New implementation of flushing at CSIRO

- We are never going to flush 100% of the files, or even many more than say 10%, so we need to save only perhaps 10 to 20 buckets: each will represent about 1% of the time period in question, and should hold around 1% of the files and 1% of the data
- The buckets can be saved on disc, thus obviating the need for large memory. The time boundaries need to be recorded.
- Do the scan in advance, not just when a flush is needed: overnight, or weekly or monthly, or only when a new scan is likely to be needed.

New implementation of flushing at CSIRO

- Part B: flushing
- When a flush is needed (we check every 5 minutes), the flush process looks at the bucket containing the names of the oldest files. It then reads this bucket, and for each file checks that the file still exists, and that its access and modify times are not later than the bucket's upper limit: if all is well, the file can be removed, and recorded. If not, just skip the file.
- "It probably works"
- Having done one bucket, it can be moved aside (for the record), and the file system checked against the desired threshold. If more needs to be done, the bucket containing the names of the next oldest files can be dealt with in the same manner, until enough buckets have been dealt with to reach the threshold.

New implementation of flushing at CSIRO

- Finally, if the threshold has not been reached, but all the buckets have been processed, it is time for another scan. Avoid this by always ensuring that there are plenty of buckets available.
- Flushing can start promptly when needed, since a list is ready to go. There is the extra expense of a lookup for the file's existence and times to be added, but this is small compared with having to scan the entire filesystem.
- There is no need to rescan while old buckets remain: once a bucket is done for a time interval, no new entries are likely (files should always go forward in time).
- Coding, testing and implementation done in 2016, after announcements to users.

Implementation

- Working!
- *The scan time no longer matters!*
- Flushing starts promptly when triggered – a few seconds
  - With our old code, could take hours to respond.
- Timings

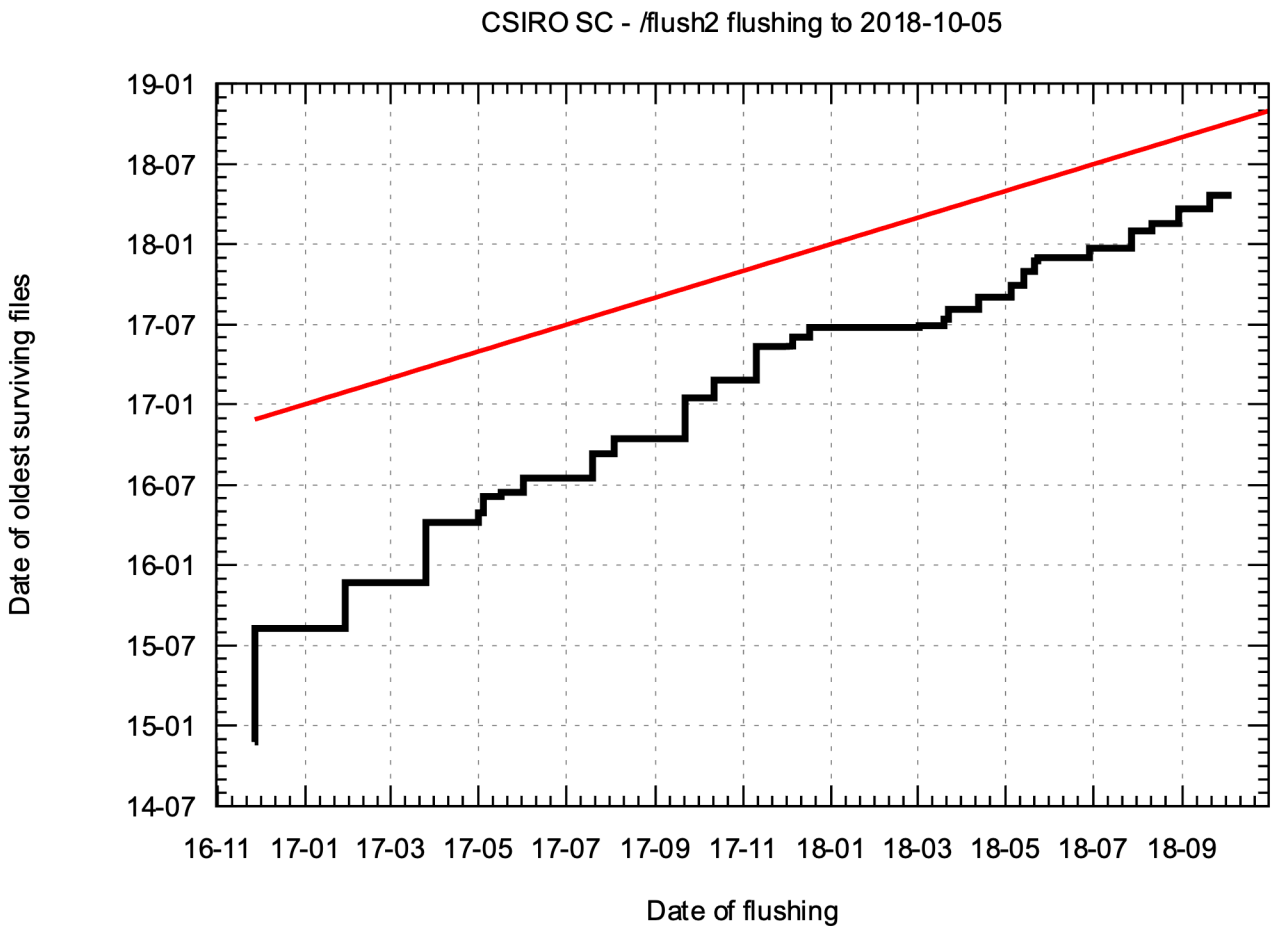
Filesystem	Files (M)	Scan time (minutes)	Flush time (minutes)	Removals (M)	Average wait (s)
/flush1	30	96	32	1.98	2
/flush2	33	26	9	4.29	7

- Buckets were nowhere near the uniformity I hoped for:
  - Saved 50 buckets instead.

Recent work

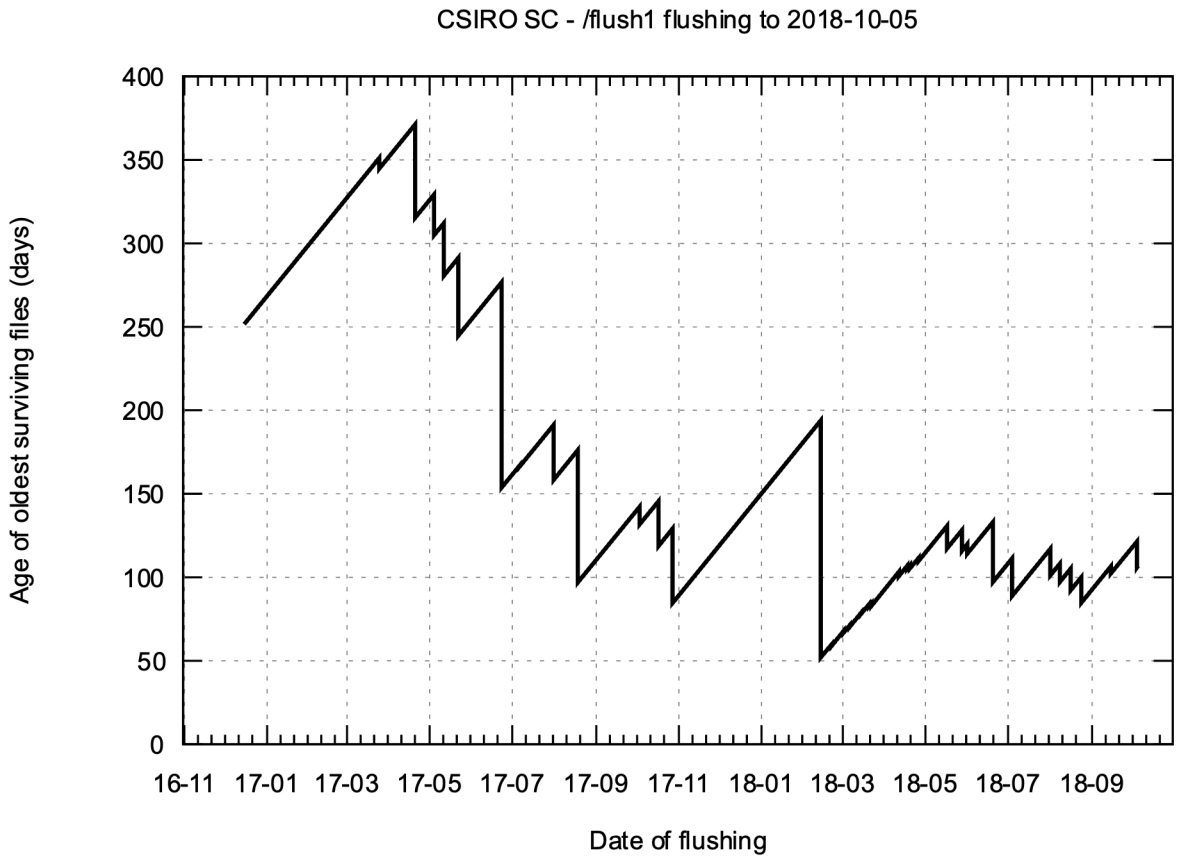
- Production hardening
  - Responded to incident where someone accessed all the files, leaving nothing to flush
  - Now save 100 buckets spanning whole date range – not much extra work
- Time and date consistency
- Keep lists of flushed files
- Can provide lists per user of files at risk
- Open source – on bitbucket
- Technical report available
- Proposal to increase default quotas by factor of 2.5
- Recent flush – 99.73% of files in the buckets that were processed were deleted.

Implementation: flush dates and surviving file ages

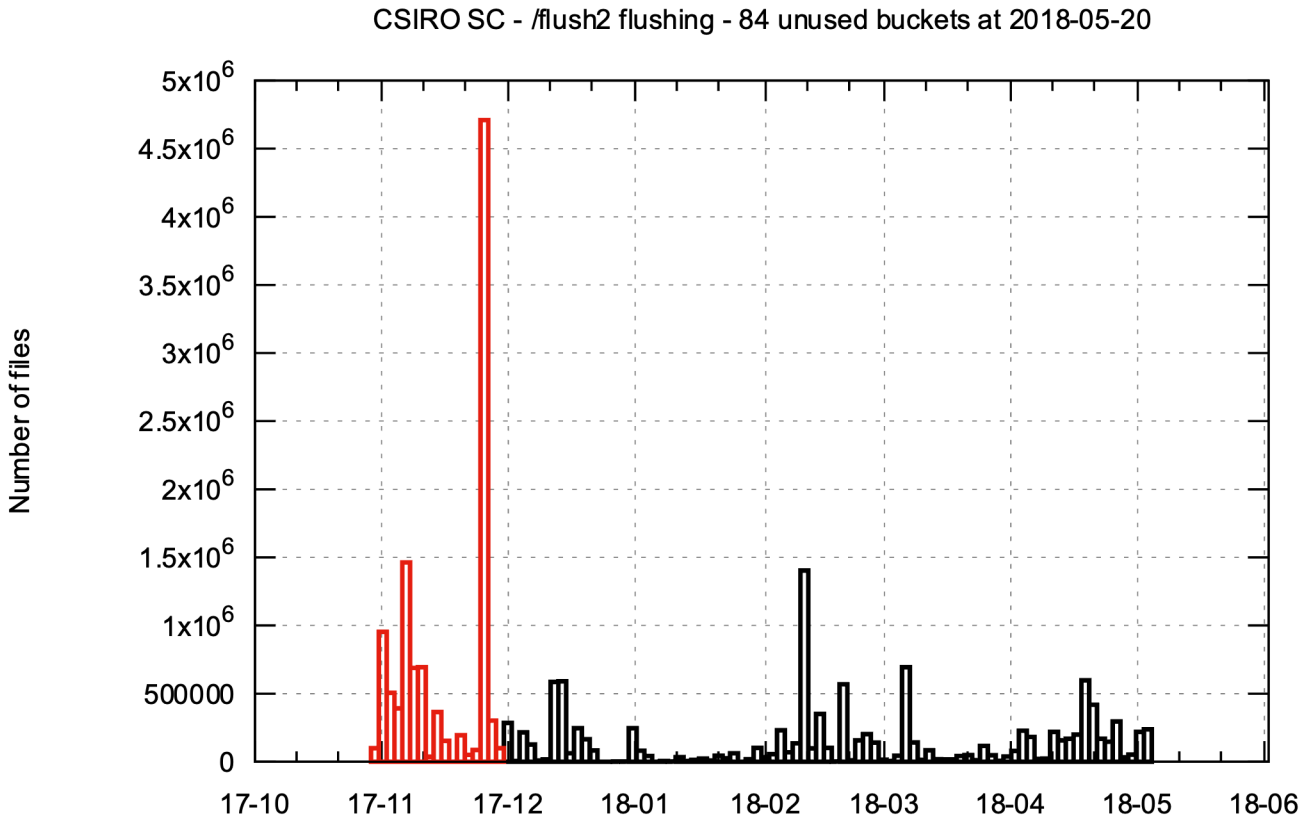


Last flush was at Thu Sep 20 06:55:02 2018
1741156 files older than Sun Apr 22 02:57:11 2018 were flushed

Implementation: age of surviving files

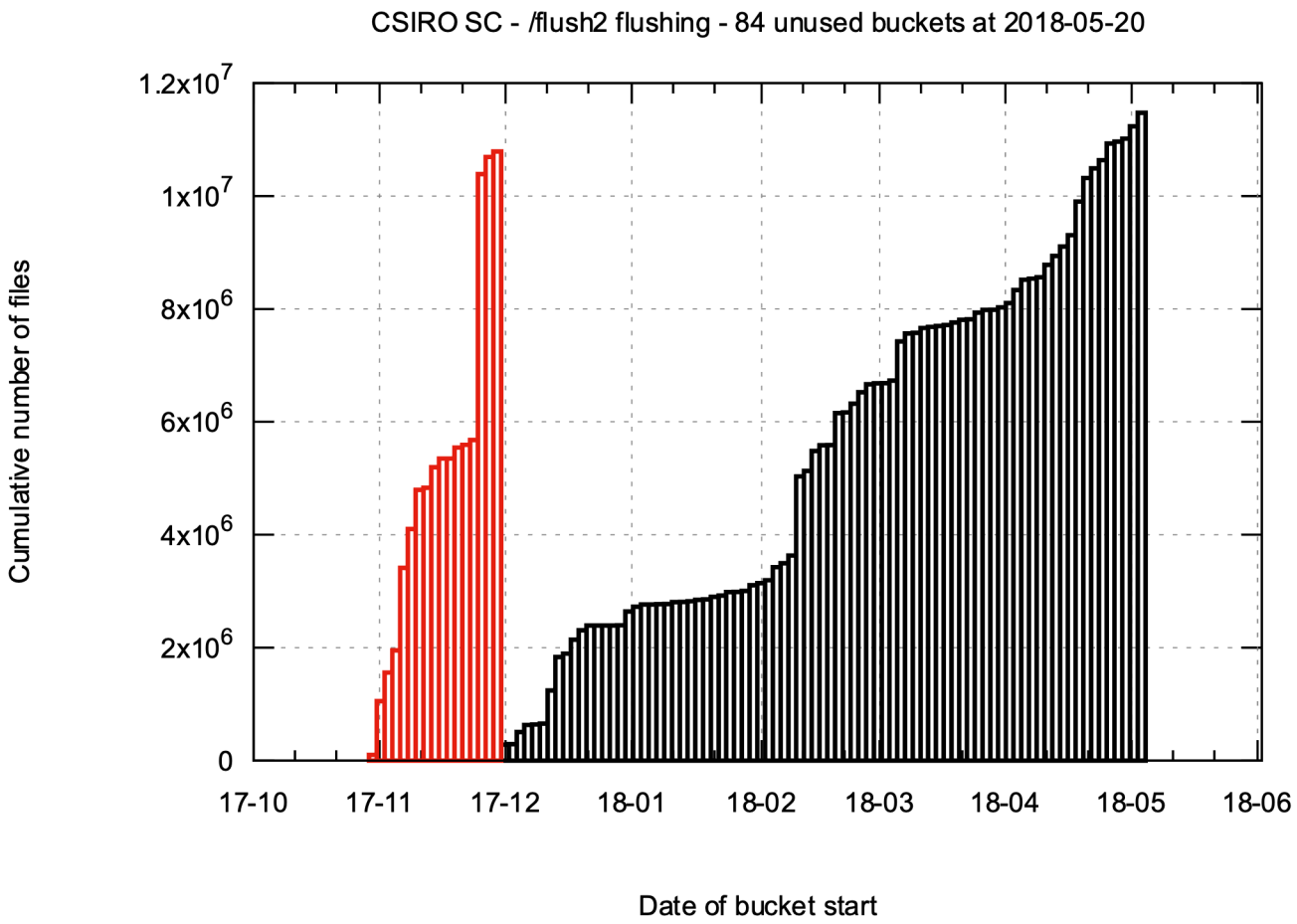


Implementation: numbers of files



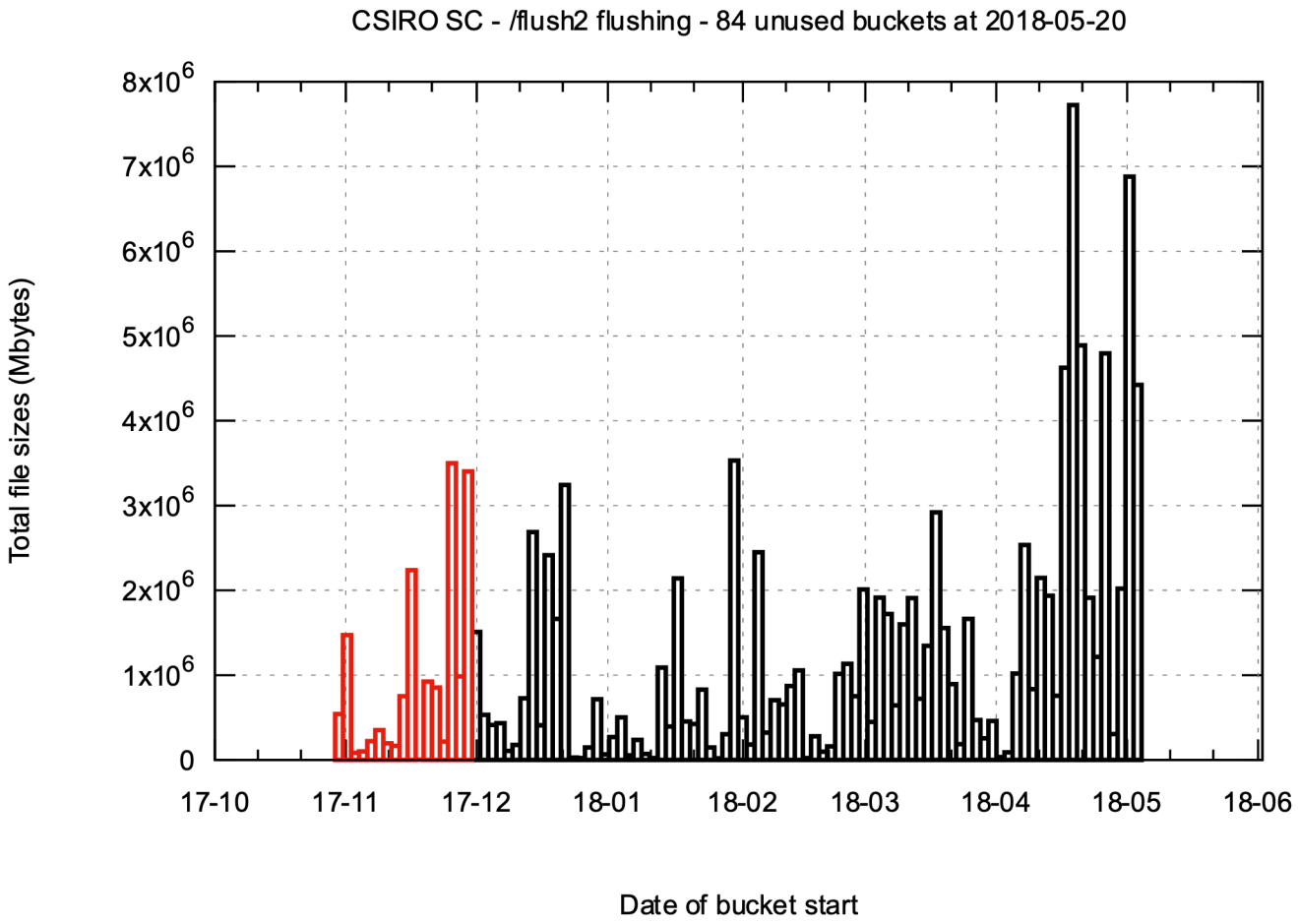
Available buckets contain a total of 11.473 million files
of 20.341 million inodes in use

Implementation: cumulative numbers of files



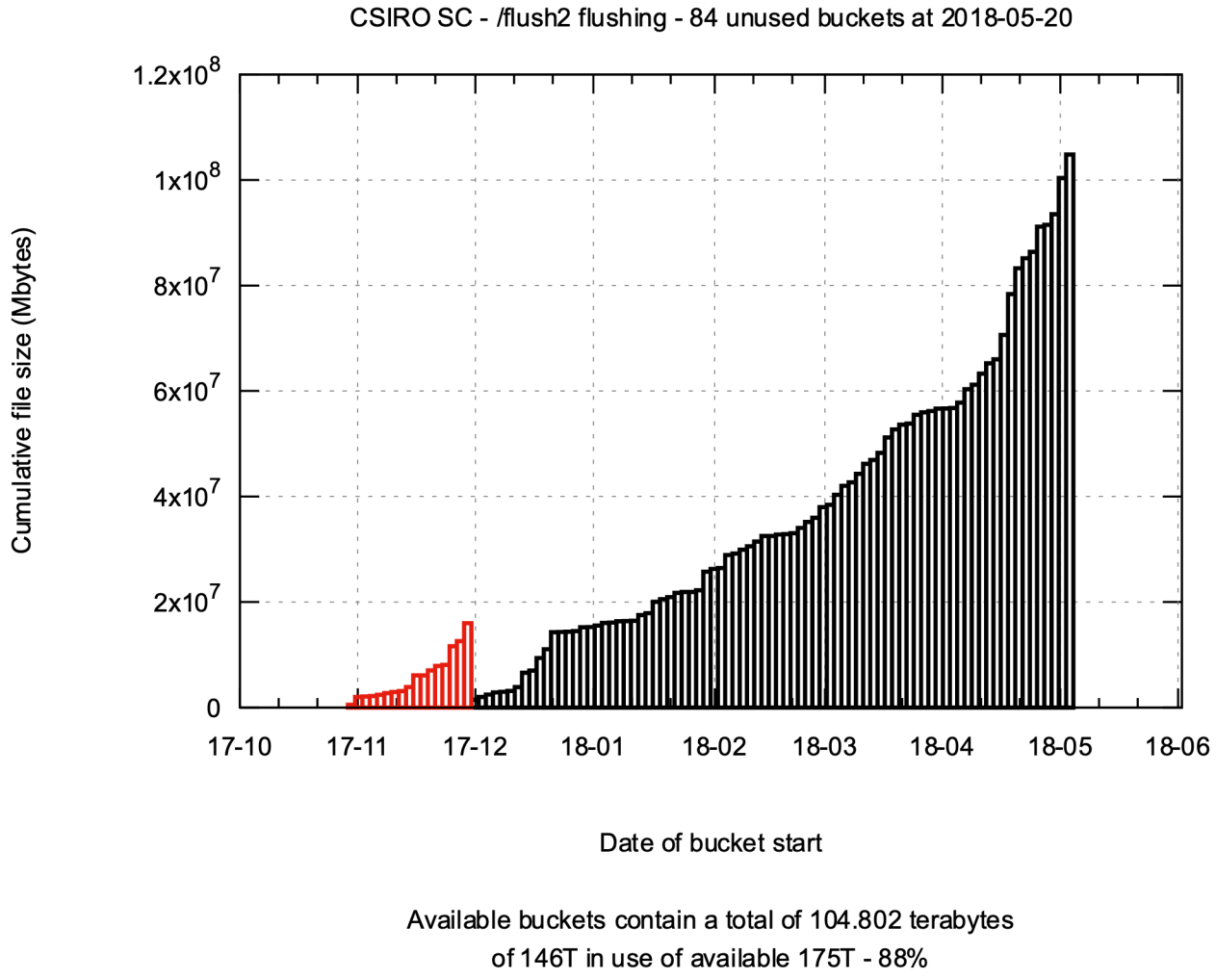
Available buckets contain a total of 11.473 million files
of 20.341 million inodes in use

Implementation: file sizes

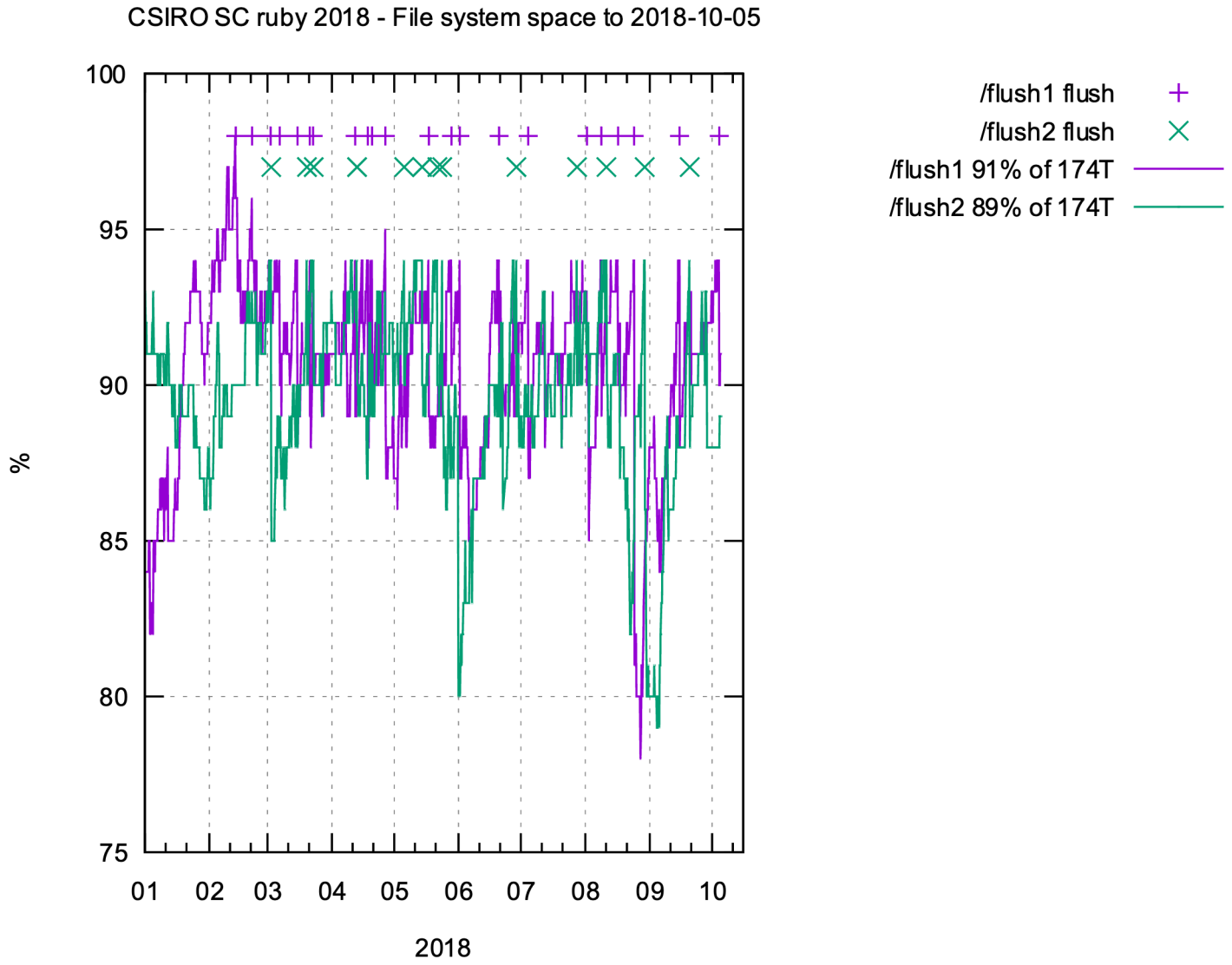


Available buckets contain a total of 104.802 terabytes
of 146T in use of available 175T - 88%

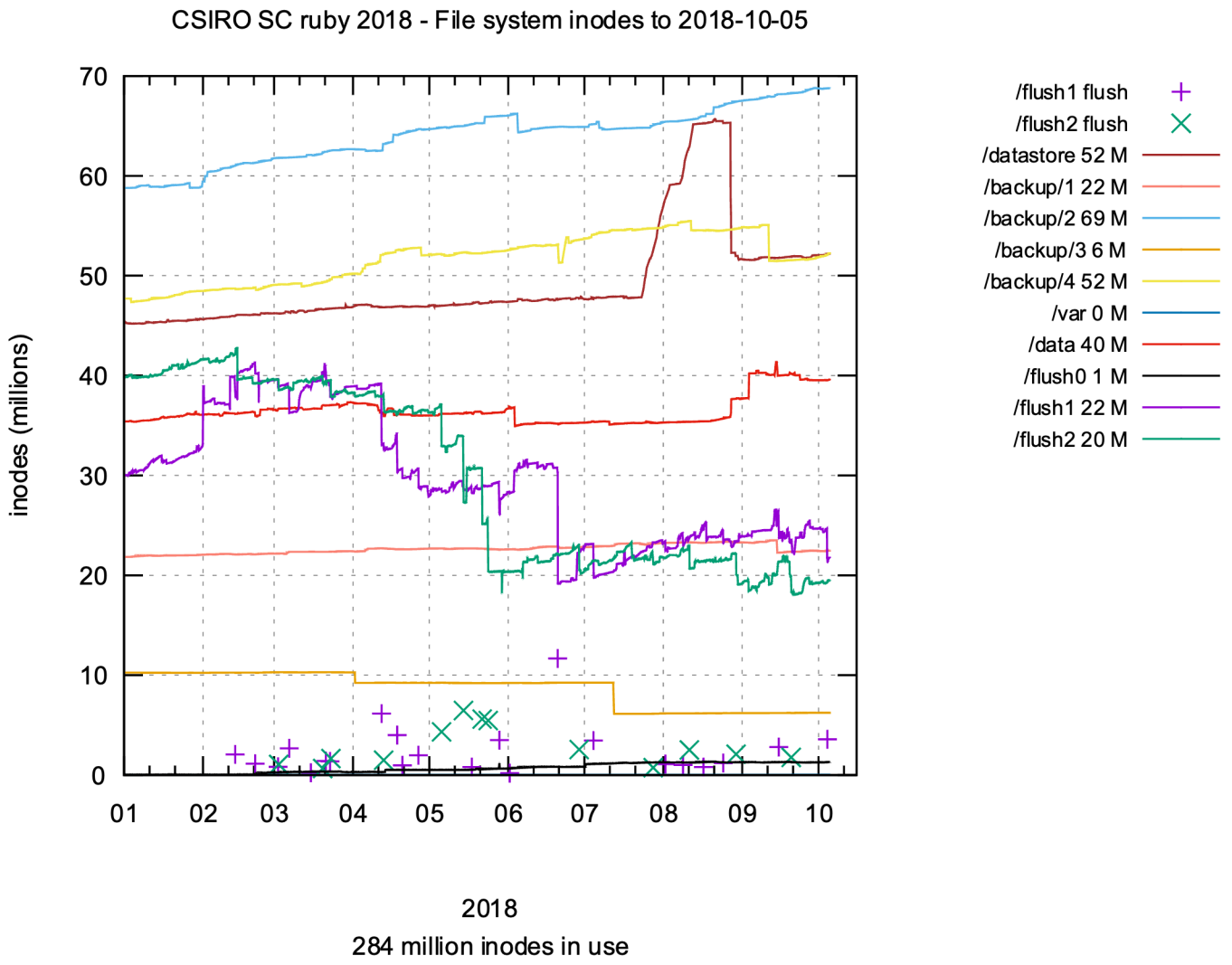
Implementation: cumulative file sizes



Implementation



Implementation



Further work

- Pawsey/NERSC interested in adopting this design – possible collaboration?
- NCI may adopt this for its next HPC system.
- Could save file sizes with pathnames, and sort buckets, so flush would start with biggest files – not needed
- Could do scanning in parallel – separate scans for each metadata server
- Could run flushing in parallel – separate flushing for each metadata server, and a separate thread for each bucket.
- Extension coded to allow flushing of part of a filesystem
- Death of /data, /projects: working on collection management in scratch – protection, syncing.

Conclusion

- Policies for temporary storage
  - necessary for users, systems staff and management and productivity of users
  - range of options: maximise the value of the resources
  - need to communicate the policies (beforehand!)
- Implementing policies
  - necessary, to avoid disasters and wastage
  - tends to be over-looked
  - disasters in waiting (users' ignorance and complacency), masked by reliable hardware (mostly)
  - mustn't add to the disasters!
- New scalable flushing methodology from CSIRO

Conclusion

- With scratch under control:

  Death of {/,/g/}data!

- CSIRO looking at abolition of /data area
- Hopelessly unmanageable storage!
- Users can use either scratch, or HSM-managed, or Bowen Research Cloud
- Support dataflows
- Prototype utility to mirror scratch areas into persistent storage, and have ability to re-create after flushing
- Another talk!

Acknowledgements

- Jeroen van den Muyzenberg
- Steve McMahon
- Ahmed Shamsul Arefin
- Peter Edwards

Thank you

CSIRO IMT Scientific Computing
Robert C. Bell
CSIRO HPC National Partnerships
t +61 428 108 333
e Robert.Bell@csiro.au
w www.csiro.au