Author: zenground0
Date: April 17
Status: IN REVIEW

Determining Source of “badger” Disk Use

Step 1 -- Observing pattern of bloat

After init:

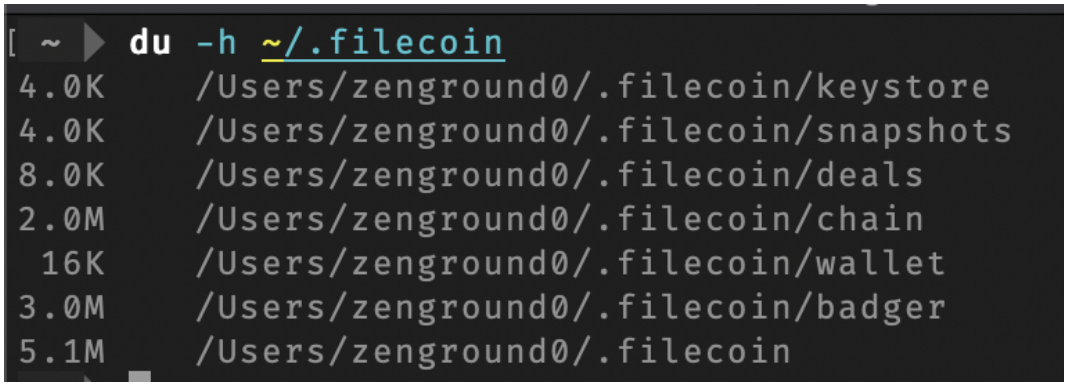
```
~ ▶ du -h ~/.filecoin
4.0K    /Users/zenground0/.filecoin/keystore
4.0K    /Users/zenground0/.filecoin/snapshots
4.0K    /Users/zenground0/.filecoin/deals
 12K    /Users/zenground0/.filecoin/chain
 12K    /Users/zenground0/.filecoin/wallet
 12K    /Users/zenground0/.filecoin/badger
 56K    /Users/zenground0/.filecoin
```

After blocks collected ~600 blocks on nightly chain):

```
~ ▶ du -h ~/.filecoin
4.0K    /Users/zenground0/.filecoin/keystore
4.0K    /Users/zenground0/.filecoin/snapshots
8.0K    /Users/zenground0/.filecoin/deals
 16K    /Users/zenground0/.filecoin/chain
 16K    /Users/zenground0/.filecoin/wallet
912K    /Users/zenground0/.filecoin/badger
972K    /Users/zenground0/.filecoin
```

Almost all data added to badger. This turns out to be DHT and block data.

After chain processed:



Badger grows bigger than chain (todo demonstrate with bigger chain to see actual bloat, at this size it's actually acceptable)

Step 2 -- Fine grained binning of storage

I set out to investigate was where all of this badger data was coming from. One hypothesis was that this data was written out as part of badger's internals (the directory is called badger after all) and was not directly related to go-filecoin writes. It turns out that this is just the name given to the directory in charge of the main datastore.

The main datastore is used by a few different parts of the system including the cborstore, blockstore, networking subsystem, and bitswap service. I split these out into different datastores (tree, vm-storage, dht, and bitswap respectively) and took care of some bugs this introduced (mostly around assumptions the genesis init function makes about everything being stored in the same place). After this I performed the same measurements with du.

After init:

After blocks collected:

All of the added data can be tracked to the network subsystem (which primarily uses the datastore for dht AFAIK) and the bitswap system which is storing block data while collecting the chain from the network.

After chain processed:

```
~ ▶ du -h ~/.filecoin
3.0M    /Users/zenground0/.filecoin/tree
4.0K    /Users/zenground0/.filecoin/keystore
4.0K    /Users/zenground0/.filecoin/snapshots
780K    /Users/zenground0/.filecoin/bitswap
268K    /Users/zenground0/.filecoin/dht
8.0K    /Users/zenground0/.filecoin/deals
2.0M    /Users/zenground0/.filecoin/chain
 16K    /Users/zenground0/.filecoin/wallet
 16K    /Users/zenground0/.filecoin/vm-storage
 16K    /Users/zenground0/.filecoin/badger
6.1M    /Users/zenground0/.filecoin
```

Almost all non-chain data added during processing can be traced back to the cborstore (tree) which is primarily used to store the bytes of the top level state of the filecoin state machine. Specifically these bytes define a map from all addresses in the system to the actor corresponding to that address. An actor is a simple 1 layer deep 4 field struct defined here.

VM state data (i.e. the actor specific storage that can be reached by following the actor head cid pointer) is also added to disk, but it has a footprint about 1000x less than the chain data.

Step 3 -- Tallying tree additions per invocation of state.Flush(ctx)

It didn't make immediate sense to me that so much state data was being written during chain processing. To be sure that these cborstore writes were in fact coming from state flushing I added some per-session bookkeeping to the ipld store, defined sessions here, here and here in the major state flushing calls, and added an output loop in the node. I actually saw more data than expected passing through these flush points (see step 4 for explanation of the double counting). The takeaway was that state data added through the cborstore is responsible for the disk buildup during chain processing.

```
Sync flush-point bytes written: 2222109
EC-A-0 flush-point bytes written: 2221817
EC-A-1 flush-point bytes written: 0
EC-A-2 flush-point bytes written: 0
EC-A-3 flush-point bytes written: 0
EC-A-4 flush-point bytes written: 0
EC-A-5 flush-point bytes written: 0
EC-A-6 flush-point bytes written: 0
EC-B-0 flush-point bytes written: 2222109
EC-B-1 flush-point bytes written: 0
EC-B-2 flush-point bytes written: 0
EC-B-3 flush-point bytes written: 0
EC-B-4 flush-point bytes written: 0
EC-B-5 flush-point bytes written: 0
EC-B-6 flush-point bytes written: 0
```

Step 4 -- Printing objects stored in HAMT

After getting the above output I had a few questions

(1) Why is there so much state data being written?

(2) As there is more data being written to the cborstore recorded here than seen by du could this analysis still be missing other cborstore writes?

To address (1) I simply started printing out the ipld nodes bing persisted through the cborstore. Outputs were all the same form, a single ipld hamt node mapping actor addresses to a simple actor data structure:

```Begin Example (with metadata)```

ipld store node:
cid:zdpuAteExXdjsCYsSbckNi7nWHeqqhSQHrHaWd55Vgt2pJyv6
size:1785
{"bf":"QAAAACAAAAEAQAAAAAAggAUIAAABAIAAABAAAABAAA==","p":[{"v":[[
"t13zauynes2qylof76m5ec57lulfl4dmf2t7yj56q",{"balance":"gICAgKS9u7rGoPPk8
pMD","code":{"/":"zb2rhmciMRS4PBXYBBHssphYKQrbFujVjj1SMyUyN2bU2SKF
x"},"nonce":"AA=="}]]},{"v":[["t2i4llai5x72clnz643iydyplvjmni74x4vyme7ny",{"balan
ce":"gICA2teU/LGBrSo=","code":{"/":"zb2rhmciMRS4PBXYBBHssphYKQrbFujVjj
1SMyUyN2bU2SKFx"},"nonce":"AA=="}]]},{"v":[["t2lq7rtgxxnpplcau35fvs63xqsxa4
m67gfw2hs2a",{"balance":"gICAtK+p+OOC2tQA","code":{"/":"zb2rhnjc9z7KtSPv3
9Pf7nyu3tAw14mFDNNxYGPgr91229rVJ"},"head":{"/":"zDPWYqFCzvVuGXGWq
X6e1CsoQ4n3NsLP5GxakuAWwKX4ETwSXbEM"},"nonce":"AA=="}]]},{"v":[["t2e
dzdcw3ibj3xst4o5shhz7zoosn52sca7bsqyby",{"balance":"gICA1arkzb7ly/b0hAQ=
","code":{"/":"zb2rhmciMRS4PBXYBBHssphYKQrbFujVjj1SMyUyN2bU2SKFx"},"
nonce":"AQ=="}]]},{"v":[["t2b2p2ajwuio3a25ykviv3xmrydhccj26m6i5baoa",{"balan
```

ce":"AA==","code":{"/":"zb2rhf5SNWYHN9Ytganp111W1soHZAyLhYf78EidRJZga
UQ57"},"nonce":"AA=="}]]},{"v":[["t1nr6qe66aipc5krn5ngpqutbssxigbgahwursrtq",{
"balance":"gICAgKS9u7rGoPPk8pMD","code":{"/":"zb2rhmciMRS4PBXYBBHssp
hYKQrbFujVjj1SMyUyN2bU2SKFx"},"nonce":"AA=="}]]},{"v":[["t1rmd5yncr7cecpj
my2jmztiyh22nqbqvieaqw2ai",{"balance":"gKD35e273sawrOHt8pMD","code":{"/":
"zb2rhmciMRS4PBXYBBHssphYKQrbFujVjj1SMyUyN2bU2SKFx"},"nonce":"BA=
="}]]},{"v":[["t2aej33sftxlutexxi2ks27npxdjn5csssukubtgi",{"balance":"gICgz8jgyO
OKAQ==","code":{"/":"zb2rhe71ud5XjA2RaFk2esBaQcD51EjKRLFP2yrB7Yq2Wo
iZ3"},"head":{"/":"zDPWYqFD4zm5iHKyZs4mVDHeDaANdXPCbKvprytX19ah9mk
LCXUS"},"nonce":"AA=="}]]},{"v":[["t13y435si3ag7732bjexbvnxo5akacskbbd3tvm
by",{"balance":"gICAgKS9u7rGoPPk8pMD","code":{"/":"zb2rhmciMRS4PBXYBB
HssphYKQrbFujVjj1SMyUyN2bU2SKFx"},"nonce":"AA=="}]]},{"v":[["t1d26zohp2xs
3n7uiqv6loca67hijveq47izablii",{"balance":"gODowaTfjoDhCQ==","code":{"/":"zb2
rhmciMRS4PBXYBBHssphYKQrbFujVjj1SMyUyN2bU2SKFx"},"nonce":"Ag=="}]]}
,{"v":[["t25j3aimcwcbmy2osgzy36iqoz45osmhl2eesybcy",{"balance":"AA==","code
":{"/":"zb2rhj9ChPyFeKYgsstgQohCp76B7xVe8sPDGXxBut5vBWdnE"},"head":{"/
":"zDPWYqFD5Pp2WLF2wQufknZHUwKbbXUBZPCWvoXG7s8oPLnihieq"},"non
ce":"AA=="}]]},{"v":[["t1jqcdvrldpwbgt3qoii5shw5jo4xgjj4332edigy",{"balance":"gIC
AgKS9u7rGoPPk8pMD","code":{"/":"zb2rhmciMRS4PBXYBBHssphYKQrbFujVjj1
SMyUyN2bU2SKFx"},"nonce":"AA=="}]]},{"v":[["t2ydigmboo6yjmbyqxyy3eyxifnrea
mnhdjzybfuy",{"balance":"gICA7ILMrqKb6TI=","code":{"/":"zb2rhmciMRS4PBXYB
BHssphYKQrbFujVjj1SMyUyN2bU2SKFx"},"nonce":"AA=="}]]}]}

```End Example```

Printing out the size of these hamt nodes and comparing with the block size (~500 bytes) reveals that state size is about 4x bigger (and only grows as more actors join the state machine). Watching many writes of very similar state data structures during chain processing further reveals that deduplication does not happen at the level of the hamt. Together this explains the larger amount of disk use by the cbor store. Furthermore the fact that the system writes the state root for *every tipset subset combination* (O(n) for a tipset with n blocks) explains the observed super linear growth in the relative space used by the state datastore when compared with the chain datastore.

Printing out the data being written to disk also helped address question (2). Almost every ipld node is recorded as being written multiple times by the modified hamt's bookkeeping. I chalk this up to datastore.Has being best effort in order to prevent it from blocking on ongoing writes. Furthermore, I hypothesize that although the datastore.Put logic is exercised more than once for the same cid, this does not lead to the datastore writing duplicate data internally (this is backed up by the du output). However confirming this would be good future work (discussed below).

Finally I edited the hamt to include a count of total additions and max size. As expected the max size was always capped at the size of the most recent state datastructure (state only increases in the system). Multiplying out max size and total additions got us a few kilobytes
```

over the total bytes persisted by the three invocations of the state.Flush() method according to the hamt's bookkeeping. This left me confident that it is indeed state flushing causing all of the disk writes through the cborstore.

Resources

To reproduce any of these observations the best place to start is with the code I wrote to run steps 2 - 4. It is important to note that this code relies on replacing the go-hamt-ipld dependency with a local version checking out the research/counting-hamt branch linked here. If you do run this code note that it is very noisy and you may need to comment out some print statements to make use of it.

Be warned that the go-filecoin code will need to be periodically rebased against master in order for the binary it builds to be able to connect to the nightly devnet as go-filecoin continues evolving. It will be up to you the user to rebase if you want to use this code.

Other Disk Inefficiencies

- 2x copying of chain data inefficiency: Our design should not be storing chain data once in a bitswap datastore and then copying it all over to a chain datastore. Originally I enforced this separation to reason about security. Specifically I wanted to prevent code from ever assuming a block had been verified without verifying it first. Right now this is utter overkill. The datastore separation does not help with this because (1) during node operation all tipsets added to the chainstore are kept in memory and (2) during load of a previously synced chain the node reads the head cids from the protected chain datastore. Furthermore when we (3) stop keeping all chain store tipsets in memory we can store metadata in the protected chain store without putting all chain data in the protected store.
- Chain datastore size increases by ~1 MB on every daemon restart: This is a seemingly unrelated issue I noticed when running du after restarting daemons. We need to analyze more to see if this is just badger eagerly allocated more because of repeated access during load, or if we are actually writing lots of unnecessary data during Load.
- DHT looks like it could be optimized: This is a secondary concern.
- datastore.Has is not filtering out cborstore.Put calls to datastore.Put: Because the analysis code in the counting hamt commonly saw what appeared to be multiple datastore writes of the same cid after performing a datastore.Has check it may be worth verifying that the datastore correctly never stores the same k,v twice. This is definitely of secondary concern however. My intuition is that ipfs folks would have found this long ago and that it is fine writing off the inaccuracy of datastore.Has as WAI and likely not causing extra storage.