Improving go-git performance

Javi Fontán

source{d}

source{d}

What’s go-git

What’s go-git

source{d}

A highly extensible Git implementation in pure Go

- 100% Go library to interact with git repositories, like git command but in a library
- Supports most of git functionality
- Uses abstracted filesystem, possibility to implement new storage systems, for example S3

```
// Clone the given repository to the given directory
Info("git clone https://github.com/src-d/go-git")

_, err := git.PlainClone("/tmp/foo", false, &git.CloneOptions{
    URL:      "https://github.com/src-d/go-git",
    Progress: os.Stdout,
})
```

What’s go-git

source{d}

Why other implementations are so fast

- They have lots of people supporting those implementations
- Use lots (I mean **LOTS**) of tricks
- Some cases have specific code paths
- They use more than one thread

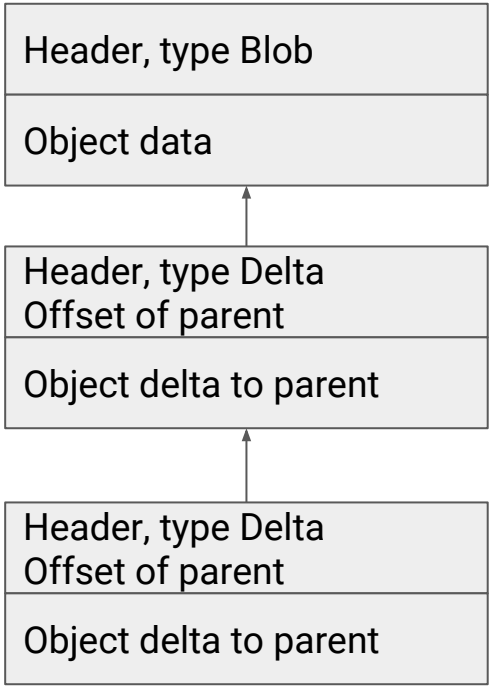
Now that go-git version 4 is almost feature complete compared with other implementations. We are working on make it faster. Most operations on small and medium sizes spend less than twice the time git command takes.

What's go-git

source{d}

How git stores data

- Packfiles hold objects in a very efficient manner.
- Each object has a header and its compressed data.
- An object can be a delta, in this case it has its parents type, points to the parent offset in the packfile and the data it holds is a delta from the parent.
- Compressed size or object hash is not written in the packfile. This is generated uncompressing, undeltifying and hashing each object.
- There's a companion index to the packfile that relates object hashes and packfile offset but it is not sent when cloning. It has to be generated.
- A CRC is calculated for each object's data in the packfile to check that it's correct.



What’s go-git

source{d}

Recent performance improvements

- Modify an LRU cache to be able to evict more than one object in case there’s no space in it. This change decreased disk access and decompression quite heavily. Index creation speed was increased 2x.
- Freed memory more aggressively when packing objects. This change decreased memory consumption between 3 and 5 times less. It also made some speed improvements as also decreased GC pressure.
- Modified delta packing algorithm to make better use of delta reuse. It doubled the speed of this process and decreased packfile size.

source{d}

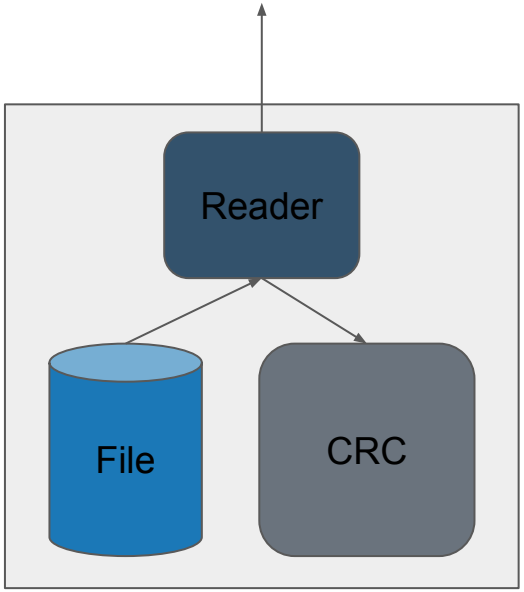
Improving index generation speed

Generating the index

source{d}

What's a teeReader

It's a reader wrapper that writes to a CRC hasher all the bytes read.



Generating the index

source{d}

Generating CRC - Original Code

```
func (r *teeReader) ReadByte() (b byte, err error) {
  b, err = r.reader.ReadByte()
  if err == nil {
    slice := []byte{b}
    _, err := r.w.Write(slice)
    if err != nil {
      return 0, err
    }
  }

  return
}
```

Generating the index

source{d}

Generating CRC - Original Profile

```
 70ms       70ms    416:func (r *teeReader) ReadByte() (b byte, err error) {
170ms      550ms    417: b, err = r.reader.ReadByte()
    .          .    418: if err == nil {
150ms      870ms    419:   slice := []byte{b}
 90ms      1.01s    420:   _, err := r.w.Write(slice)
 40ms       40ms    421:   if err != nil {
    .          .    422:     return 0, err
    .          .    423:   }
    .          .    424: }
    .          .    425:
 20ms       20ms    426: return
    .          .    427:}
```

Generating the index

source{d}

Generating CRC - Precreate Slice Code

```
func (r *teeReader) ReadByte() (b byte, err error) {
  if r.byteBuf == nil {
    r.byteBuf = make([]byte, 1)
  }
  b, err = r.reader.ReadByte()
  if err == nil {
    r.byteBuf[0] = b
    _, err := r.w.Write(r.byteBuf)
    if err != nil {
      return 0, err
    }
  }

  return
}
```

Generating the index

source{d}

Generating CRC - Precreate Slice Profile

```
40ms       40ms    418:func (r *teeReader) ReadByte() (b byte, err error) {
40ms       40ms    419: if r.byteBuf == nil {
   .          .    420:   r.byteBuf = make([]byte, 1)
   .          .    421: }
60ms      470ms    422: b, err = r.reader.ReadByte()
   .          .    423: if err == nil {
20ms       20ms    424:   r.byteBuf[0] = b
70ms      880ms    425:   _, err := r.w.Write(r.byteBuf)
20ms       20ms    426:   if err != nil {
   .          .    427:     return 0, err
   .          .    428:   }
   .          .    429: }
   .          .    430:
40ms       40ms    431: return
   .          .    432:}
```

Generating the index

source{d}

Generating CRC - Library Code

```
func slicingUpdate(crc uint32, tab *slicing8Table, p []byte) uint32 {
    if len(p) >= slicing8Cutoff {
        crc = ^crc
        for len(p) > 8 {
            // here calculates CRC
            [...]
            p = p[8:]
        }
        crc = ^crc
    }
    if len(p) == 0 {
        return crc
    }
    return simpleUpdate(crc, &tab[0], p)
}
```

Generating the index

source{d}

Generating CRC - Add a Buffer to CRC Writer

```
func newTeeReader(r reader, h hash.Hash32) *teeReader {
  return &teeReader{
    reader:    r,
    w:         h,
    bufWriter: bufio.NewWriter(h),
  }
}
func (r *teeReader) ReadByte() (b byte, err error) {
  b, err = r.reader.ReadByte()
  if err == nil {
    return b, r.bufWriter.WriteByte(b)
  }

  return
}
```

Generating the index

source{d}

Generating CRC - Add a Buffer to CRC Writer

```
 60ms       60ms    437:func (r *teeReader) ReadByte() (b byte, err error) {
130ms      540ms    438: b, err = r.reader.ReadByte()
    .          .    439: if err == nil {
 80ms      270ms    440:   return b, r.bufWriter.WriteByte(b)
    .          .    441: }
    .          .    442:
 30ms       30ms    443: return
    .          .    444:}
```

Preallocate Slices

source{d}

Slice grow is expensive

- Allocates double the capacity of the old slice
- Copies the old slice data to the new one
- Adds pressure to the GC as it has to free the old slice data
- This is especially problematic for a big amount of slices or big ones
- Of you know the size or can estimate beforehand use it to set the capacity

Preallocate Slices

source{d}

- Code example

```
targetSz, delta := decodeLEB128(delta)
var dest []byte

for {
  [...]
  dest = append(dest, src[o:o+sz]...)
```

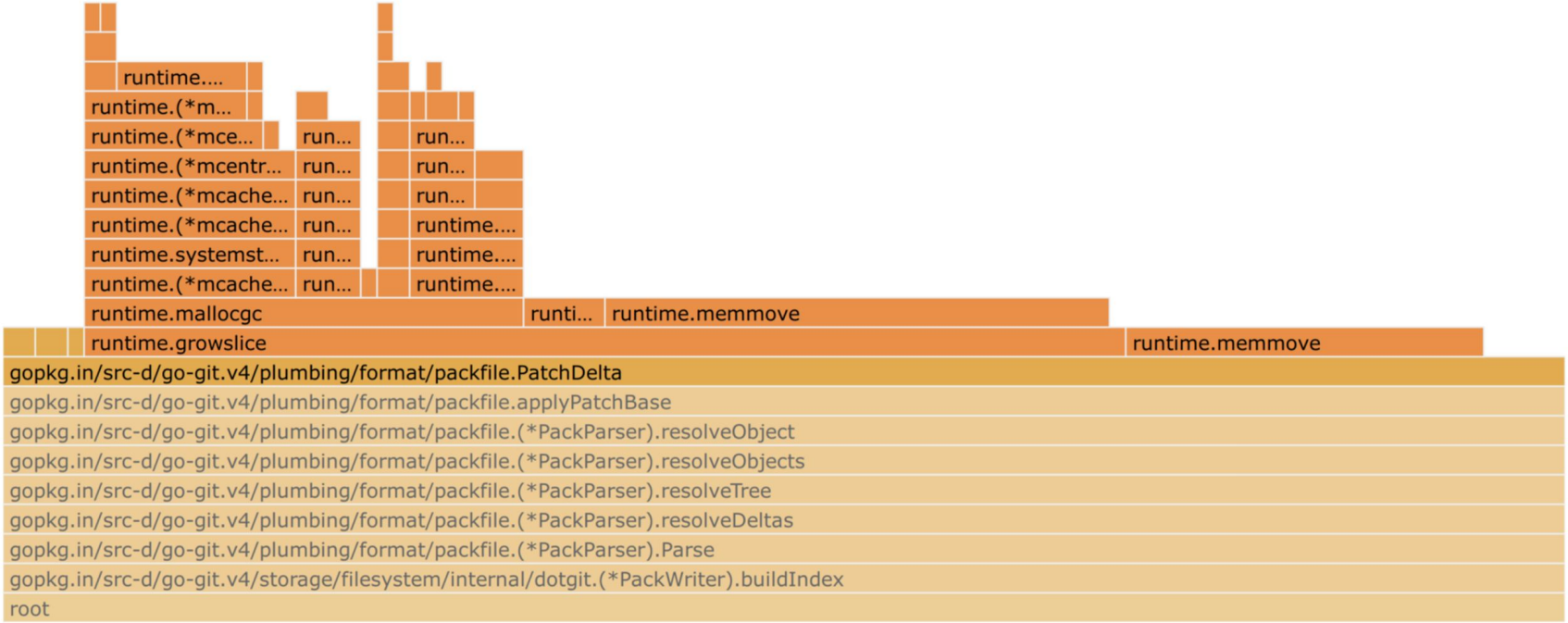
```
targetSz, delta := decodeLEB128(delta)
dest := make([]byte, 0, targetSz)

for {
  [...]
  dest = append(dest, src[o:o+sz]...)
```

Preallocate Slices

source{d}

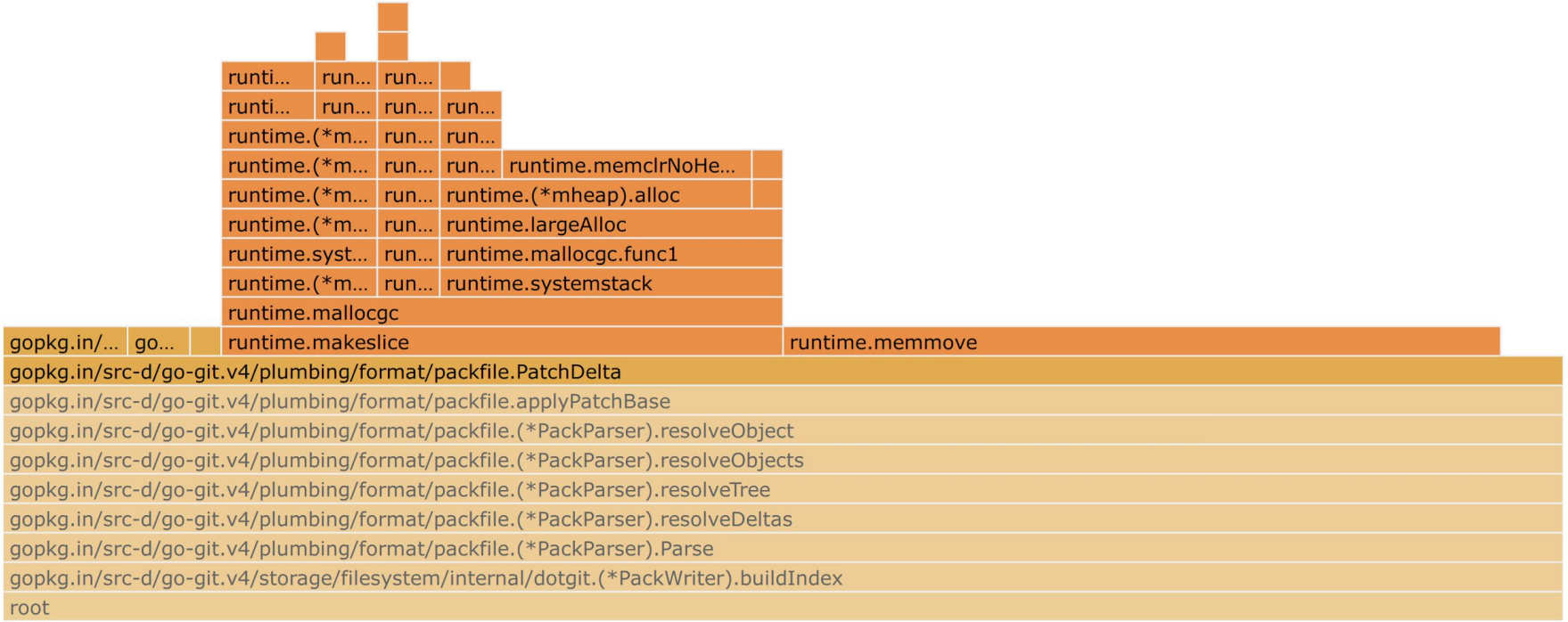
- Performance first version (0.96 s)



Preallocate Slices

source{d}

- Performance second version (0.50 s)



source{d}

Inspecting Syscalls

Inspecting Syscalls

source{d}

Understand what is doing with the system

- Your programs interact with the rest of the system
- Sometimes the bottleneck is not the software but the systems it's trying to use
- Other times is a bad use of these systems
- Calling the system has a price, it needs to switch context to kernel mode and back
- strace or dtrace can be useful to show what's it doing

```
$ strace -o strace.log -f my_software
```

- **-o**: sends the output to a file
- **-f**: follow threads

Inspecting Syscalls

source{d}

- Example of output

```
9220  openat(AT_FDCWD,
"test_repo/.git/modules/doc/scipy-sphinx-theme/objects/pack/pack-4a8c3dfb6e1
8cfab78c8602e0d60cb4734486416.pack", O_RDONLY|O_CLOEXEC) = 5
9220  lseek(5, 0, SEEK_SET)                = 0
9220  lseek(5, 0, SEEK_CUR)                = 0
9220  lseek(5, 2127, SEEK_SET)             = 2127
9220  lseek(5, 0, SEEK_CUR)                = 2127
9220  read(5,
"\227\32x\234\225P\273N\4!\24\355\347+ng\341j\200\5\2261\306\250\235&\32\243
\306\376\2\27"..., 4096) = 4096
9220  lseek(5, 0, SEEK_CUR)                = 6223
9220  lseek(5, 0, SEEK_SET)                = 0
9220  close(5)
```

Inspecting Syscalls

source{d}

- Find bad uses of SEEK_CUR

```
$ grep -E '(open|read|seek)' strace.log
[...]
11209 lseek(6, 0, SEEK_CUR)             = 224543
11209 lseek(6, 0, SEEK_CUR)             = 224543
11209 lseek(6, 0, SEEK_CUR)             = 224543
11209 lseek(6, 0, SEEK_CUR)             = 224543
11209 lseek(6, 0, SEEK_CUR)             = 224543
11209 lseek(6, 0, SEEK_CUR)             = 224543
11209 lseek(6, 0, SEEK_CUR)             = 224543
11209 lseek(6, 0, SEEK_CUR)             = 224543
11209 read(6, "\327\261\360"..., 4096) = 4096
11209 lseek(6, 0, SEEK_CUR)             = 228639
11209 lseek(6, 0, SEEK_CUR)             = 228639
11209 lseek(6, 0, SEEK_CUR)             = 228639
11209 lseek(6, 0, SEEK_CUR)             = 228639
```

Inspecting Syscalls

source{d}

- Find seek direction and jump size

```
$ grep SEEK_SET master.strace | grep -v -E '(= 0$|unfinished)' | awk 'BEGIN
{pos=0} {new=$6;print pos-new;pos=new}'
[...]
-19608
19843
-19843
19843
-423
188
-319
-19289
19843
-19843
19843
```

Conclusions

source{d}

- Biggest improvements are done changing the algorithm
- The profiler is your friend, learn to use it
- Check if you’re doing the same thing multiple times, this happens very often
- Try not to do one byte writes, use buffers is you really need to do it
- Preallocate slices when possible
- Your software does not work alone, check the use of the rest of the system
- When in doubt check library implementations

source{d}

Thanks!