Achieving high performance in Mercurial

Bryan O'Sullivan

bos@serpentine.com

The origins of Mercurial

- In April 2005, Linux was suddenly without a revision control tool
- No viable replacements in sight
- Two people started writing their own
- Linus Torvalds: git
  - C core, random scripting languages on top
- Matt Mackall: Mercurial
  - Pure Python at first (now 5% C)

The problem domain

- Original target: Linux kernel
  - 20,000 files—one checkout is ~220MB
- Hundreds of contributors
  - Some use git, some Mercurial, some nothing
- Median rate of change:
  - 334 commits per week
  - ~3000 files modified per week
- Peak rate of change:
  - 342 commits in one hour, 2105 in one week

Where Mercurial is now

- High-profile adopters
  - OpenSolaris
  - One Laptop Per Child
  - Xen OS virualisation
- Growth in both free and commercial users
- Why are people drawn to Mercurial?
  - Easy to learn and understand
  - Very fast, scales from tiny to very large projects
  - Simple to deploy

Distributed revision control

- Everyone has full history locally
  - All common operations are local, even commits
- No single point of failure
- Resync with other developers when you want
  - Network only needed during resync
  - No loss of productivity on e.g. trains, planes
- Branching and merging are very frequent

The social aspect

- Centralised tools *enforce* a divided world
  - You have *committers* and *outsiders*
  - Outsiders have very limited (read-only) use of revision control
- Why not *choose* your development model?
- If you like close-knit centralised work, simply use a distributed tool that way
  - If you prefer a different model, distributed tools will *just work*

Why care about performance?

- Performance is not an end in itself
  - The point is to make the software more usable
- 30-minute waits cost more than just *time*
  - Easy to forget what you were doing
  - Less tool use, so fewer commits
  - Fewer commits means fewer fallbacks on error
- What does high performance *buy* you?
  - You don't switch to something else while you wait
  - Cheap branches encourage experiments
  - New ways to do revision control

Python—a surprising choice?

- People do not often choose Python for “intensive” workloads
  - (Plenty of exceptions, but generally true)
- ... So why did Mercurial choose it?
  - Revision control is often I/O-bound
    - If you're waiting for disk, no language will help!
  - Mercurial leaves much “heavy lifting” to C
    - Uses standard Python libraries, functions known to be written in C

Python—benefits

- The usual stuff
  - Expressiveness
  - Duck typing
  - I'm preaching to the converted, right?
- More interesting for long-term viability...
  - Plenty of useful, clean third-party code for reuse
    - urlgrabber, lsprof, coverage.py, ...
  - Lower obstacles to casual contribution
  - 75% of all contributors have sent in <= 5 patches
  - Many *never used Python* before fixing a bug or adding a feature in Mercurial and contributing it

Strategies for success

- When in doubt, do nothing
- Plan for performance on day one
- It's easier to make simple things fast
- If you're not measuring it, it's slow
- Find ways to avoid the disk

When in doubt, do nothing

- Every project has tar pits
  - 50% of the work, 0.5% of the benefit
- What does Mercurial do nothing about?
  - Content merges (*huge* tar pit!)
  - Line ending conversion
- Surely we can't have *no* answer!?
  - Content merges: external script
  - Line endings: contributed plugin
- Core stays simple, but users are happy

Plan for performance on day one

- Revision control is well understood
  - But people still write non-scalable tools
- Start with performance in mind
  - “I expect in most cases my tea will still be warm”
- Set yourself performance targets
  - You might have to refine them later
  - But you'll never meet them if they're not there
- Measure on large data sets from the outset
  - It's easier to start with good performance than to retrofit it

It's easier to make simple things fast

- Example: compare revision and working dir
- Strawman implementation
  - Compare one file in repo with one in working dir
  - If comparing entire tree, repeat for all files
- Doing this is slow on big trees
  - On small tree, hard to measure difference
- Less abstraction makes it easier to map from *user's intention* to *what the disk can do well*

If you're not measuring it, it's slow

- Two notionally linear-time algorithms
  - “A” knows about file layout on disk
  - “B” does not
- “A”p ermits more linear reads, fewer seeks
- The mechanical properties of the disk give “A” a huge advantage, *but ...*

- On *small* data sets, both look “fast enough”

- When you grow your data set, the fun begins!

Have *you* measured recently?

- It's easy to add performance regressions
  - They can go undiscovered for *months*
- *Two* regressions in scanning modified files
  - One changed file access ordering, so suddenly started seeking more
  - Another defeated an inner-loop optimisation
  - *Both changes looked completely innocuous to reviewers*
- You have to treat performance as a *process*
  - It's not just a “place you get to” and then ignore

Find ways to avoid the disk, 1

- How to walk the directory tree efficiently
- Don't use open().read() when os.stat() will do
  - Save stat data at checkout time, only do open&read if stat data has changed
- Don't use os.stat() when os.listdir() could do
  - On BSD & Linux, readdir(3) will tell the type of an entry, so no need to os.stat() directories
  - Requires a C extension, but saves up to 33% time
- Don't use os.listdir() when *nothing* will do
  - Prune directories and files to walk as early as possible

Metadata storage

- Some popular ways to store revision data
  - Weave: O(N) (SCCS)
  - Reverse delta: O(N) (RCS, SVN/bdb)
  - Skip delta: O(log N) (SVN/fsfs)
- Reads, writes, or both are expensive
  - Adding a rev to a weave can rewrite it all
  - Reading an old rev is costly with reverse deltas
  - Reading the newest rev with fsfs is expensive

Find ways to avoid the disk, 2

- Mercurial's answer: the *revlog*
  - Revision data stored as forward deltas
  - Files are only ever *read* or *appended*
  - Periodic fulltext storage makes retrieval O(1)
- Delta is computed against *head*, not *parent*
  - Trades off *delta size* against *seek probability*
  - Bigger deltas can be read linearly at small cost
  - Seeking to a parent revision is expensive
  - ~25% faster than delta vs parent in practice

Mercurial and revlogs

- Revlogs underly all historical metadata
  - *Filelog*—file modification history
  - *Manifest*—which rev of each file is present in a changeset
  - *Changelog*—rev of manifest to use, committer name, comment, other changeset metadata
- The only C code in Mercurial is here
  - One module for computing deltas
  - Another for composing them when reading a rev
  - Just 500 lines of code

File formats, flexibility, speed

- Two death knells of I/O performance
  - “Let's use sqlite/mysql/postgres!”
  - “Let's use XML!”
- SQL is great if you're in uncertain territory
  - *Bad* if you want to control I/O
- XML is OKish if you need to interoperate
  - *Expensive* to read and write
  - Unpleasant APIs make code hard to follow
- Each provides tempting extensibility
  - *“Low barrier to new features”* means *“high barrier to good performance”*

In praise of *inflexible* file formats

- **Note:** this only makes sense if you know *exactly* what you need
- Design as few formats as possible
  - Mercurial has two (*revlog* and *dirstate*)
  - Focus on the crucial abstractions you must have
- Make files *simple* to parse and generate
  - Mercurial uses string split/join, struct pack/unpack
- Think about I/O implications, and *measure!*

Simple performance tests

- Subversion 1.3.2 (ra_local) *vs.* Mercurial 0.9
  - Data: Linux 2.6.17 (20,041 files, 216MB data)
  - Host: Thinkpad X31, 1.3GHz Pentium M, 768M RAM

Operation	**Units**	**Svn**	**Hg**
Time to *add* all files to empty repo	secs	79.8	4.6
Time to *commit* all added files	mins	33.5	1.3
Modify 5,000 files, time *status*	secs	320.4	58.7
Modify 5,000 files, time *diff*	secs	329.2	70.6
Size of working copy (import of one rev)	MB	595	387
Size of working copy (full history)	MB	*~595*	567

- These are *best case* numbers for Subversion
  - ra_dav, ra_svn much more widely used, slower

The patch management problem

- A common situation:
  - You have third-party software
  - You need to modify it
  - You need to upgrade the package
- What do you do with your modifications?
  - Redo them by hand?
    - Painful!
  - Generate a patch, and apply it?
    - Not too bad, but conflicts are a pain
- What if you have *lots* of modifications?

The solution: Mercurial Queues

- Automates patch management
- Integrated into Mercurial
  - Patches look like changesets
  - Use tools like log, annotate, bisect on patches
- Need to upgrade underlying software?
  - No problem!
  - If patches have conflicts, use GUI tools to fix
- Used by Linux kernel and distro maintainers

Using Mercurial Queues

- When you create a new patch...
  (...or import an existing patch or series...)
- ...the patch looks like a changeset, *but...*
- You can edit files, then *refresh* the patch
- Not just for *managing* static patches
  - *Maintain* existing patches, follow upstream evolution
  - *Develop* new features as patch series
- Work with a *stack* of incremental patches
  - *Pop* down the stack to work on an older patch
  - Changesets for the popped patches vanish
  - *Push* patches back onto the stack again later

MQ and speed

- MQ was built for *lots* of patches
- Real-world test: Linux 2.6.17-mm1
  - Andrew Morton's Linux kernel patches
  - **1,738** patches, **687,500** lines of changes
- Push all patches in 233.5 seconds
  - 7.5 patches committed *per second!*
- Pop all patches in 30.3 seconds
- Refresh a big patch in 6.6 seconds
  - Patch touches 287 files; changes 22,779 lines

Wrapping up

- Mercurial hits a *very* sweet spot
  - Simple model; easy to learn; hackable code
  - Comprehensive features
  - Blazingly fast
- Performance is a *means*, not an *end*
  - Lets you focus on your *real* work
  - Enables new ways to get stuff done, such as MQ
- Performance takes dedication
  - If you don't *focus* on it, you don't *have* it
  - Take your eye off it, and it vanishes

And finally...

Thank you!