A journey to performance
Using Rust in Mercurial

FOSDEM 2021
Raphaël Gomès @ Octobus
Mercurial refresher

- Version Control System started in 2005
- Written in Python (200k lines)
- Boosted by C extensions (45k lines), and now Rust (60k lines)
- Handles huge repos with millions of files and/or revisions
- Very powerful extension system
Why Rust?
Maintainability

Compared to C

• Better signal/noise ratio
• Better compile-time guarantees
• Standardized and modern tooling
• "Memory-safe" by default (unsafe blocks)
Performance

• It’s the main topic of this talk
• Compiled so no startup
• Little to no runtime
• “Zero-cost” abstractions
• Multithreading is a lot easier
Attacking the Python from both sides

- Build Rust extensions called from Python for hot paths
- Python ↔ Rust interop using the rust-cpython crate
- Wrap Python with a Rust executable for fast paths using PyOxydizer
The case of « hg status »
The « status » command

- Purpose: show the changed files in the working directory
- Possible values: modified, added...
- Has to compile the .hgignore patterns into one big regex
- Has to check every tracked file for changes and show untracked files (default)
hg status example

$ hg st
A new-file
M modified-file
? unknown-file
Last year

- Re-implemented parts of the dirstate in Rust
- Used multithreading in traversal for a considerable speedup
- Created and used shared Rust+Python iterators
Previous milestones

• Up to 4x faster with Rust extension
• Available in CentOS and Gentoo
• Used on foss.heptapod.net
• Google employees showing strong interest
Previously at FOSDEM 2020

Performance
“hg status” experiment
by Valentin Gatien-Baron

<table>
<thead>
<tr>
<th></th>
<th>hg</th>
<th>Rust hg</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>2.4s</td>
<td>50ms</td>
</tr>
<tr>
<td>status -u</td>
<td>2.4s</td>
<td>39ms</td>
</tr>
<tr>
<td>status -mard</td>
<td>400ms</td>
<td>14ms</td>
</tr>
</tbody>
</table>

"This is unrealistic" - me, not long ago
## Not so unrealistic

<table>
<thead>
<tr>
<th></th>
<th>hg</th>
<th>Experiment</th>
<th>New Rust</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mozilla clean</td>
<td>1.151s</td>
<td>73ms</td>
<td>44ms</td>
</tr>
<tr>
<td>Mozilla dirty</td>
<td>2.129s</td>
<td>203ms</td>
<td>64ms</td>
</tr>
<tr>
<td>Pathological</td>
<td>7.334s</td>
<td>973ms</td>
<td>53ms</td>
</tr>
</tbody>
</table>
How « status » works
The dirstate

• Very naive approach: open all tracked files and compare the content with the Mercurial store

• Better: store the size, exec bit, mtime and state (added, removed, etc.). This is called the dirstate

• Has not been changed since 2005, has a flat structure
Current dirstate structure

<table>
<thead>
<tr>
<th>Parent 1 hash</th>
<th>Parent 2 hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>some/otherfile</td>
<td>Metadata</td>
</tr>
<tr>
<td>some/file</td>
<td>Metadata</td>
</tr>
<tr>
<td>not-sorted.txt</td>
<td>Metadata</td>
</tr>
</tbody>
</table>
Old algorithm

• Walk and check the working directory respecting ignore rules, yielding the results of any file encountered.
• Check if any files in the dirstate that were not seen on disk: they were deleted, ignored, under a symlink.
Issues with this approach

- Expensive mapping to compare it to the dirstate, itself expensive to build (tens of ms)
- Expensive ignore and security check
- Iterating more than once over the dirstate
A better datastructure

Let’s build a tree instead to mirror the working directory:

- Allows for a unified iteration
- More efficient ignore rules
- Symlink substitution and nested .hg are very cheap to ignore
- Status in a directory comes for free
Implementing a new dirstate
Components of the Tree

Disk data containing full nodes
Components of the Tree

- Disk data containing full nodes

```rust
struct Vault<'a> {
    Reference
    Vec of full paths
}
```
Components of the Tree

- Disk data containing full nodes
- `struct Vault<'a> { Reference, Vec of full paths }
- `struct OnDiskNode { start: VaultKey, }
- `struct InMemoryNode { path: VaultedPath, children, state, etc. }
Append-only on-disk storage

- **Unreachable (dead) entries**
- **Reachable (alive) entries**
- **Mutated (new) part**
Append-only on-disk storage

- Used in Mercurial since the beginning (Revlog)
- Also works with dynamic sized entries
- Once the unreachable parts become too large, re-write everything (vacuum)
Transactionality

• There are two files per storage
• A small atomic « docket » file keeps track of metadata (generation, size, root node offset, etc.)
• A data file with a unique name per vacuum cycle
Memory mapping

- Memory usage stays the same with multiple processes
- Append-only: don’t have to worry about truncation
- Basically removes read time
Removing « build time »

- Mmap is basically free most of the time
- Keep the full paths, don’t compress the roots: no allocation needed
- Only new nodes are allocated
Other optimizations
Fast directory traversal

- Recursive implementation with 1 task per directory (naive)
- Iterative walk is more complicated and will probably not be useful
- Using rayon’s threadpool: easy
- crossbeam-channel for the results
Mtime caching

- Compare themtime of directories with the cached version
- If different: need to read the dir, otherwise just stat the children, then recurse
- Can even cache unknown/ignored files
Mtime caching

• Only works with certain OS/filesystem pairs

• Needs to be properly invalidated (parents changes, moved in FS, new ignore rule, etc.)
Let’s recap
Building the tree

- Open the docket and read it
- Mmap the corresponding data file
- The Vault already knows how to access the data, it does not need to be read first
- The root node is given by the docket
Checking the working copy

- Iterate over both the dirstate and the working copy
- Check for any relevant differences
- Directory → symlink checks are free when using mtime caching
- Send the results with a channel from all threads
## Performance (POC)

<table>
<thead>
<tr>
<th></th>
<th>hg</th>
<th>Experiment</th>
<th>New Rust</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mozilla clean</td>
<td>1.151s</td>
<td>73ms</td>
<td>44ms</td>
</tr>
<tr>
<td>Mozilla dirty</td>
<td>2.129s</td>
<td>203ms</td>
<td>64ms</td>
</tr>
<tr>
<td>Pathological</td>
<td>7.334s</td>
<td>973ms</td>
<td>53ms</td>
</tr>
</tbody>
</table>
Dirstate is used in many commands

- Commit
- Diff
- Update
- Purge
- Files
- etc.
Linux is the simpler platform
Paths, filesystems, IO

- Paths are just plain bytes in Linux
- Great filesystems like BTRFS
- On MacOS: unicode normalization issues, case sensitivity
- Windows, above issues + file IO is very slow and has more footguns with transactionality
Linux is faster, but could be faster still?

- Using getdents64 directly instead of the (most likely) slower opendir/readdir loop/closedir
- A lot of our time is spent in system instead of user
- Is fstatat/openat faster because it doesn’t have to re-walk the tree?
Mercurial
is always getting faster
Other Rust endeavors

• Ancestor iteration (up to 3x perf in discovery)
• Copytracing (up to 20x speedup)
• Nodemap (up to 2000x speedup in index lookups!)
• hg cat (more than 10x speedup)
• hg files (more than 10x speedup)
Non-Rust endeavors

- Cloning needs 40% less RSS memory
- Sparse-revlog (up to 99% reduction of manifest size)
- New revlog format (fewer files, better information)
- New branch cache, etc.
Keep an eye out
In the next few months
Thank you!

Raphaël Gomès @ OCTOBUS