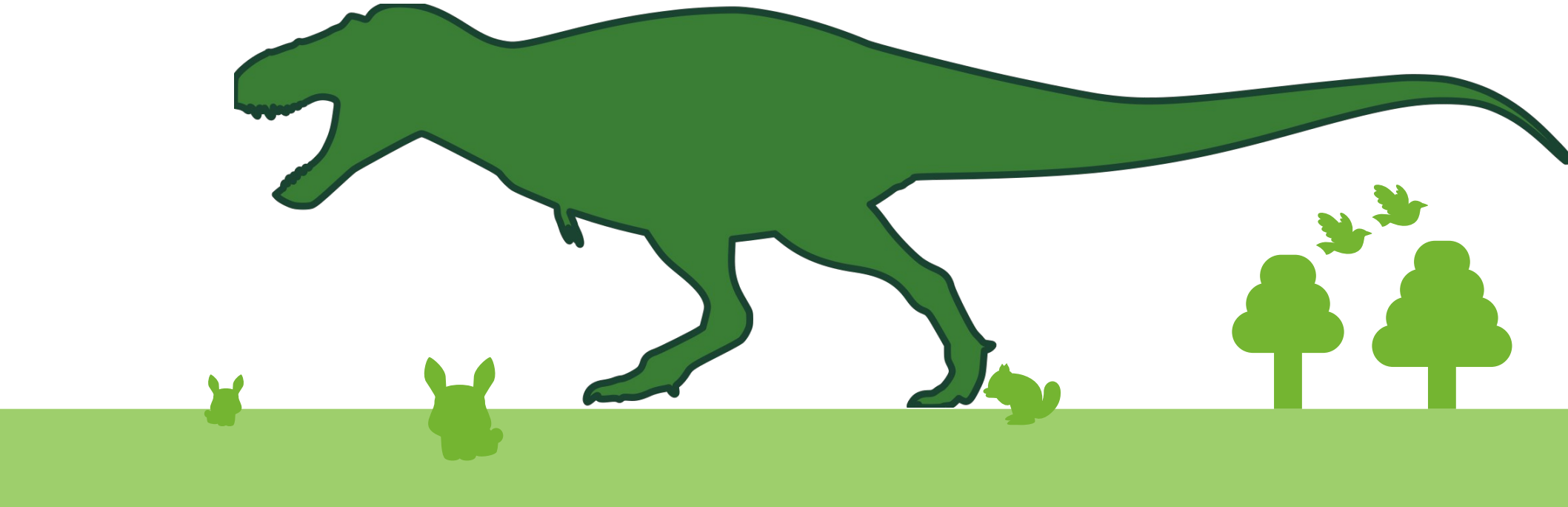
bcachefs – engineering for reliability at scale

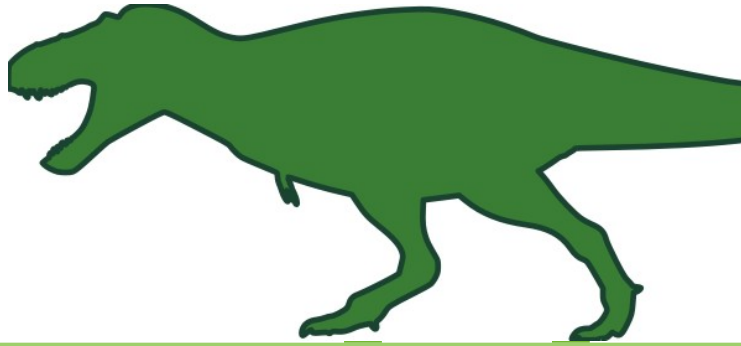


What’s this project about… (goals)

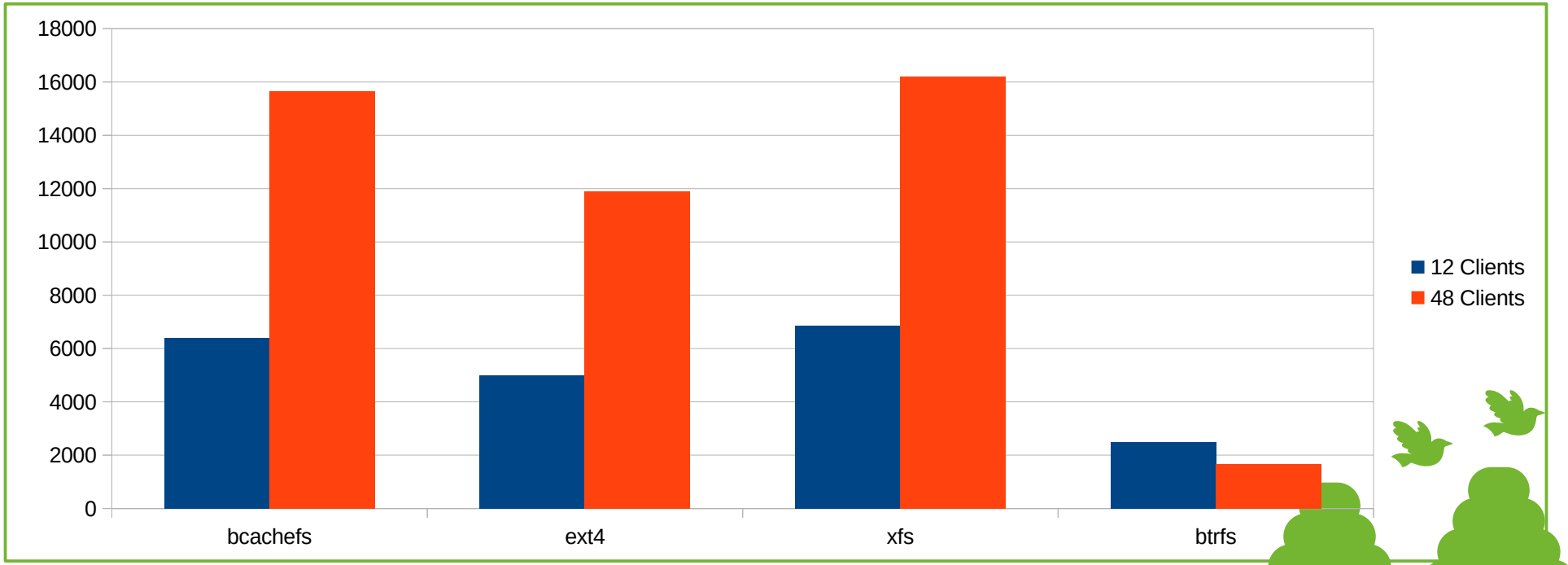
- Never, ever lose data: we don’t care whose fault it was or how damaged it was – the filesystem should never lose data
- Repair from anything, safely and automatically (self healing is a whole topic...)
- Easy for end users to understand, especially when something goes wrong
- Full modern featureset, as pioneered by ZFS
- Real database underpinnings – clean up the traditional filesystem architecture
- Performance and scalability to be truly general purpose
- End game: address the split between local and distributed, make the local filesystem a good foundation: so we’re not having to reinvent

Project status, recent highlights

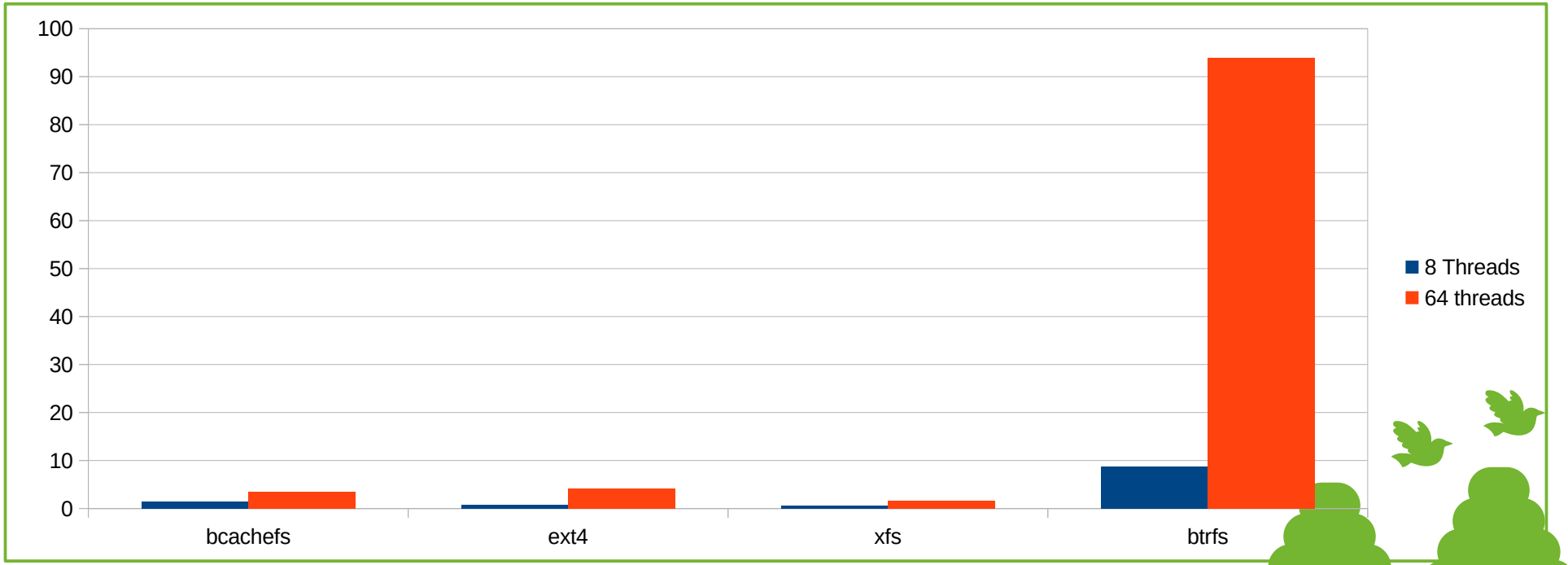
- No longer experimental
- Efficient, scalable snapshots (uses key versioning, not metadata COW, and snapshots are writeable and branchable)
- Data/drive management – reconcile, declarative and state driven, better than ZFS rewrite
- High performance, no write hole RAID/EC – stabilized, now seeing production usage
- Performance work: parity or better than XFS on dbench (high concurrency mixed data/metadata workload)
- Heavily focused on: working with users, documentation, debug tooling, usability – making everything simple and smooth
- https://bcachefs.org/bcachefs-principles-of-operation.pdf
  130 pages of reference manual



dbench mb/sec



SQLite runtime (Phoronix test suite)



The limits of software engineering are not what we can imagine, but what we can debug

- Engineering at scale ends up being less about the code itself, and more about all the things that seem at first glance like overhead
- Automated testing, introspection and runtime in-the-wild debugging, error messages, keeping code organized and well factored
- And now: memory safety, formal verification
- Or more simply - always use the best tools you have available

things we emphasize for reliability

- Automated testing – an entire cluster testing every commit, as it is pushed, with a massive and growing test suite
- Runtime debugging: all runtime state should be inspectable (even high rate in flight ops)
  (yes, all, also timestats; latency/jitter has long been opaque and it's only gotten somewhat better with bpf offcputime)
- Error messages should describe the entire operation that failed, and explain what we were doing and why

Automated testing: ktest

test name (regex) Search

Commit	Description	Passed	Failed	Failed to run	Not run	In progress	Unknown	Total	Duration
b87c2a19b87b5f	ioctl: get the opcodes from the C compiler	10584	66	1196	3117	0	0	14963	270034s
0178f9e09cb9d3	bindgen: delete the Fix753 workaround	10551	73	1224	3115	0	0	14963	275915s
7a3fc90be7ad8c	codegen: force bch_extent_crc to align 8 on 32-bit	10562	76	1208	3117	0	0	14963	279741s
aba5212210ff55	ec: lift stripe creation out of bch2_ec_stripe_head_get()	10566	83	1197	3117	0	0	14963	275858s
c9f39407fb77bd	ec: operations on a stripe should take the stripe	10572	81	1192	3118	0	0	14963	278828s
b4dfb54046e9cf	btree: set_node_max() has to un-root and write-lock too	6035	37	209	2072	0	0	8353	127432s
e69ef85d15923b	btree: topology repair must un-root a node before dropping it	1901	20	81	626	0	0	2628	41904s
4c6561ed9b3009	ec: one copy of "are there enough devices for a stripe"	2355	38	76	900	0	0	3369	64197s
eef2cf4f80954d	alloc: report why erasure coding wasn't available	1910	22	69	627	0	0	2628	46085s
22d83a6f2b5019	extents: union bch_extent_crc is __aligned(8)	4908	39	179	1877	0	0	7003	105122s

bcachefs idioms to Rust

darray	Vec
*_to_text()	Display
printbuf	Formatter
closures	async/await, Arc
lots and lots of x-macros	Derive
errcode.h	ThisError (partly)
try()	?

Error hierarchies

- Flattening errors (EINTR, EAGAIN) is a nightmare for debugability
- bcachefs philosophy: error codes should, wherever possible, be unique: printing the error code then identifies the line of code that threw the error
- This necessitates error heirarchies – any error code must be subtypable, and matchable with bch2_err_matches()
- bch_err_fn(fs, ret) – prints __func__ and the error name, which is frequently all we need

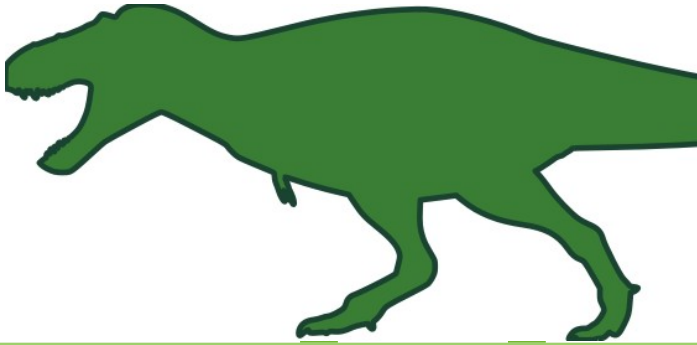
Error message examples

```
btree node read error at btree alloc level 0/0
  u64s 11 type btree_ptr_v2 SPOS_MAX len 0 ver 0  :
    seq 49f9e9d0fea8d6c9 written 15 min_key POS_MIN durability: 1
    ptr: demo1.img 0:25:0 gen 0
  demo1.img node offset 0/15: bad magic: want be57c3d2ece5d2ac, got b4aa67744ec56c8e
    demo1.img io: btree_node_validate_err
  flagging btree alloc lost data
  scheduling recovery pass check_topology (2)
  scheduling recovery pass check_allocations (8)
  scheduling recovery pass check_alloc_info (15)
  error btree_node_validate_err
error reading btree root btree=alloc level=0: btree_node_read_error, fixing
```

Individual error messages include

- How we retried
- Repair actions

The goal is to make the underlying logic of the system clear and accessible



Tracing in bcachefs

The bcachefs approach:

- Make tracing as simple as possible, for the end user and the developer
- Don’t bother with TRACE_EVENT(); all tracepoints are strings, so we can use our to_text() functions or Display trait
- Get as much mileage as possible out of a tracepoint definition: every defined tracepoint also has a corresponding always on counter

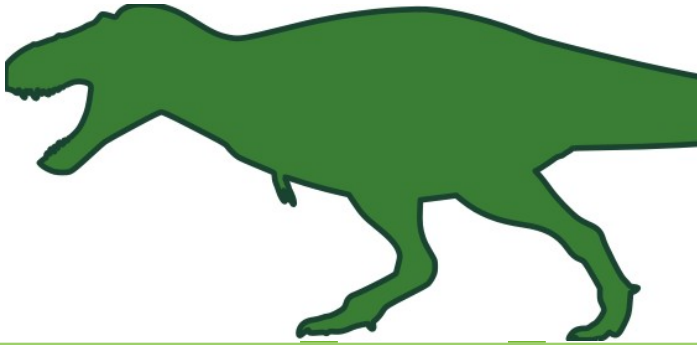
  Then we can include tools that let users go from system counters to individual tracepoints with a keystroke

bcachefs fs top

Every counter has a corresponding tracepoint; Enter drills into a live trace of the selected event, scoped to this mount.

q:quit Enter:live-trace h:human-readable Tab:page ↑↓:scroll PgUp/PgDn 1-9:interval
[counters] [devices]

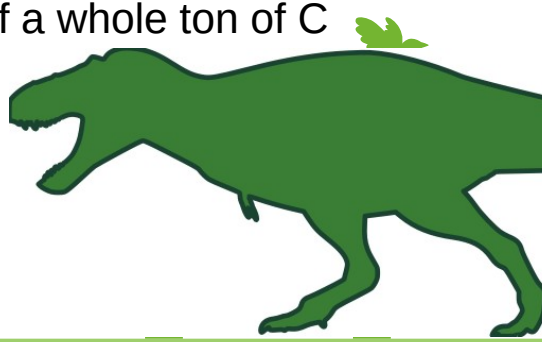
	1/s	total	mount
► sync_fs	1/s	7	207
data_read_hole	0B/s	0B	2.00M
data_read_reuse_race	0/s	0	39
data_write	125M/s	1.02G	15.5G
data_update_pred	135M/s	1.04G	14.0G
data_update	75.0M/s	584M	9.16G
data_update_in_flight	6/s	48	1144
data_update_fail	0B/s	0B	452K
data_update_read	75.0M/s	583M	9.15G
data_update_write	75.1M/s	583M	9.15G
data_update_key	74.9M/s	582M	9.13G
data_update_key_fail	44.0K/s	368K	13.1M
evacuate_bucket	4286/s	34.1K	445K
copygc	268/s	2131	27.9K
bucket_discard_worker	287/s	2151	19.5K
bucket_discard_fast_worker	1/s	3	280
bucket_discard	10.4K/s	76.7K	1.09M
bucket_alloc	9715/s	79.4K	1.22M
bucket_alloc_fail	75/s	466	32.4K
sectors_alloc	304M/s	2.42G	37.2G
btree_cache_scan	0/s	0	17
btree_cache_reap	0/s	0	1130
btree_cache_cannibalize_lock	3222/s	26.8K	403K
btree_cache_cannibalize_lock_fail	201/s	1990	28.1K
btree_cache_cannibalize_unlock	3222/s	26.8K	403K



The Rust conversion

- “Do as much ahead of time as C refactoring, so the conversion can be as mechanical as possible”
- Codebase organization, with good module boundaries, so we can convert file by file
- goto → CLASS()

  Prep work removed 2k gotos (currently at 584, vs. 2742 for XFS, ~75% of files goto free)
- Automated testing with good coverage, good asserts are key
- Pragmatism: we accept somewhat more unsafe than ideal –if it gets rid of a whole ton of C
- Refactor, refactor, refactor, to make life easier later



Rust conversion status

- Userspace: done (30k loc)
- Core APIs (btree transaction, others) – “good enough” to done
- Rust interfaces to APIs moved into kernel – April/May
- Most users now on Rust-enabled builds, with actual in-kernel bcachefs Rust code since bcachefs is DKMS, this means build servers – lots of build servers
- Next up: build pipeline work on remaining distros (Proxmox)
  would be lovely if we can get packaging of .rlibs and .rmetas standardized

  Then: poll the userbase to make sure no one’s missing Rust builds before it becomes a hard dependency, and the bulk conversion starts
- First target - fsck

What Rust opens up (what we like in bcachefs)

- Error handling! Result<>, ?
- Metaprogramming
- The build and module system
- We finally have a path to practical formal verification - the borrow checker makes the rest tractable

Verification in bcachefs – the beginnings

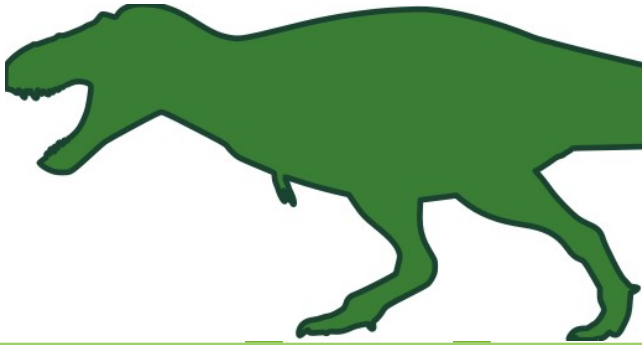
This is not your normal everyday btree – this is the sort of code we care about:

```
static struct bkey_packed *bset_search_tree(const struct btree *b,
                                const struct bset_tree *t,
                                const struct bpos *search,
                                const struct bkey_packed *packed_search)
{
        struct ro_aux_tree *base = ro_aux_tree_base(b, t);
        struct bkey_float *f;
        unsigned n = 1;

        do {
                f = &base->f[n];
                if (likely(f->exponent < BFLOAT_FAILED)) {
                        unsigned l = f->mantissa;
                        unsigned r = bkey_mantissa(packed_search, f);

                        if (likely(l != r) || !bkey_mantissa_bits_dropped(b, f)) {
                                n = n * 2 + (l < r);
                                continue;
                        }
                }

                struct bkey_packed *k = tree_to_bkey(b, t, n);
                int cmp = bkey_cmp_p_or_unp(b, k, packed_search, search);
                if (!cmp)
                        return k;

                n = n * 2 + (cmp < 0);
        } while (n < t->size);

        unsigned inorder = __eytzinger1_to_inorder(n >> 1, t->size - 1, t->extra);
        if (likely(!(n & 1))) {
                --inorder;
                if (unlikely(!inorder))
                        return btree_bkey_first(b, t);

                f = &base->f[eytzinger1_prev(n >> 1, t->size - 1)];
        }

        return cacheline_to_bkey(b, t, inorder, f->key_offset);
}
```

Verification in bcachefs today

100% runtime asserts in debug mode, formalized(-ish) and standardized; wherever possible we write assertions *on data types*
the lesson we've learned is that writing assertions in a "formal verification" mode of thinking leads to much better assertions that can catch classes of bugs with somewhat less exhaustive testing

- Btree transactions, iterators, locking – heavy usage
  `bch2_btree_path_verify()` and many others
- Input validation – if we can reject invalid input, the rest of the code becomes much easier to reason about
  not really formal verification, but has to be formalized in much the same way; sets the boundary of formal verification
- The big gap – filesystem level invariants
  invariants that btree transactions have to uphold – no practical way to address these without real formal verification



Formal verification intro

- Formal verification in conventional systems languages: technically possible, effectively infeasible - SeL4
- More successful in: functional languages - CoQ, CompCert
- And languages with a good type system - Rice’s Theorem, Idris
- Rust has a great type system
- And the borrow checker makes it a functional language in disguise...

Formal verification in Rust

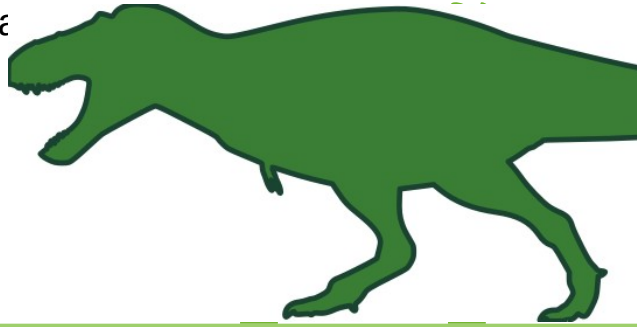
- Rust solves the most immediate problems for formal verification of systems code: it eliminates unrestricted side effects, and it has a real type system
- And now, formal verification in a mainstream systems language is finally becoming an actively explored topic, just starting to be used in production
- Verus
- Aeneas

Formal verification in bcachefs

- Primary approach: treat existing assertions as proof obligations, and fill out the proofs
  Note that a great many of our asssertions are already assertions on types
- Useful analogy: gradual typing
  When adding type annotations to a dynamically typed language, you don't do it all at once – you do it where it's useful. We can do the same thing.
- This should become another tool for responding to bug reports
  Today: "Where are we missing test coverage? Where are we missing introspection?"
  Tomorrow: "Where is our behaviour under specified? What proofs are we missing"
- The long term goal in software engineering: better abstractions, so we can make our code better specified and better behaved
  Better type systems, elimination of undefined behaviour, and now methods to formally and rigorously prove the behaviour of new abstractions

The expanding world of Rust – some predictions

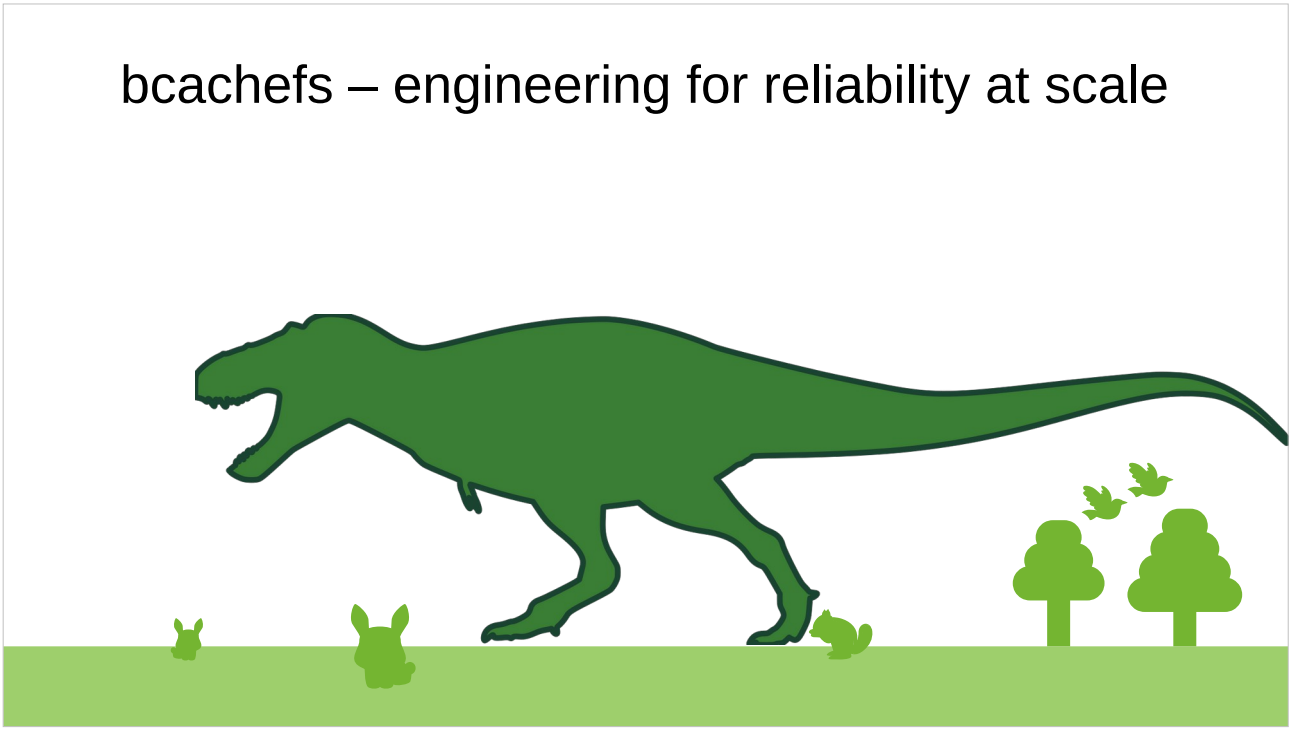
- CPU performance has been held back by C: CPU designers would like to go wider (VLIW), but unrestricted pointer aliasing is a problem
- A lot of code could be running entirely out of registers, if only enough code was written in a language with a smarter memory model…
- Open source, high performance CPU designs are starting to appear (XiangShan)
- GPU programmers still have to write to these things called “threads” and “warps” (which are different from CPU threads)
- But we have these things called “optimizing compilers” – if only people were ta
  them
- https://goals.rust-lang.org/2026/high-level-ml.html



Systems programming is changing

- New tools and new methods are going to reshape the way we work; less time debugging, more time to think about the structure of what we’re trying to do
- Systems of the future will be cleaner, easier to understand, easier to work on, and far more capable – if we build it

bcachefs – engineering for reliability at scale



This talk is about a couple things; status update on bcachefs, overview of the process of converting a large, stable and actively in use codebase to Rust. bcachefs is ~200k loc, not counting vendored code. And, I'm going to try to convey something from my own experiences of how to ship reliiable code at scale.

Rust has had a massive impact on the way we write code and talk about engineering. Thanks to Rust we're starting to bring real mathematical rigor to day to day systems engineeriing – the borrow checker, the safe/unsafe distiinction, and that's been changing how people think about and design APIs and codebases.

I first came to Rust over 10 years ago, exploring the compiiler codebase, contributing patches to the Rust bignum library, and I can stiill remember how it

What's this project about… (goals)

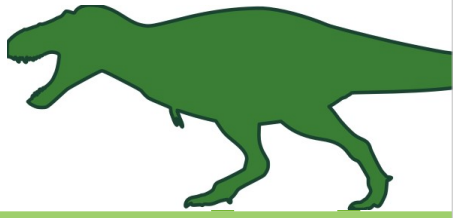
- Never, ever lose data: we don't care whose fault it was or how damaged it was – the filesystem should never lose data
- Repair from anything, safely and automatically (self healing is a whole topic...)
- Easy for end users to understand, especially when something goes wrong
- Full modern featureset, as pioneered by ZFS
- Real database underpinnings – clean up the traditional filesystem architecture
- Performance and scalability to be truly general purpose
- End game: address the split between local and distributed, make the local filesystem a good foundation: so we're not having to reinvent

I don't care if you metaphorically lit your hard drive/SSD on fire: it's the filesystem's job to repair anything that can be repaired and get back all of your data that's still there. If it doesn't, those reports go straight to the top of the pile, and I actively remind people that if something is broken – we want those reports so they can get fixed.

One early torture test was when a physics student in India, right after bcachefs was merged, put bcachefs on her family computer – a three drive filesystem, with no replication, and then accidentally wiped one of the drives. This was early, so handling this meant resurrecting the btree node scan patch series and two weeks of painstaking remote debugging – but I'm told in the end we repaired with a shockingly small amount of data missing – and now that disaster recovery code is something everyone can count on.

Project status, recent highlights

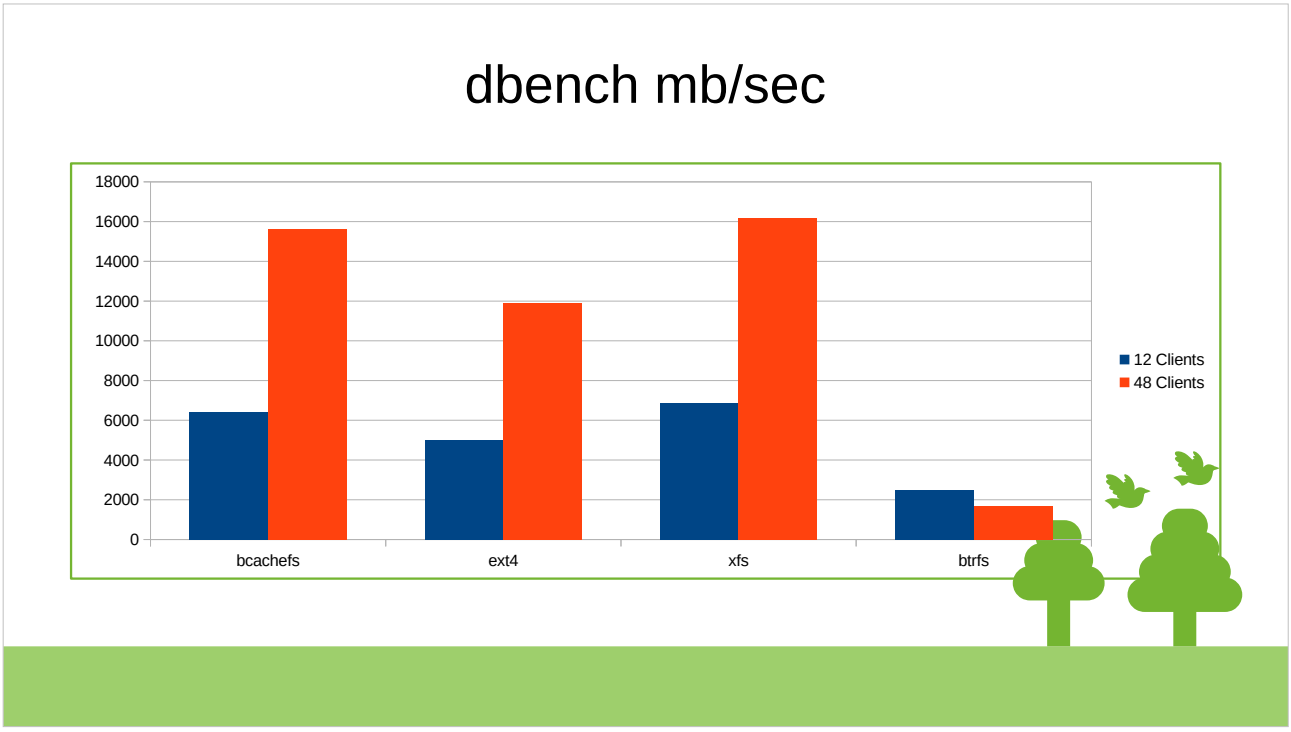
- No longer experimental
- Efficient, scalable snapshots (uses key versioning, not metadata COW, and snapshots are writeable and branchable)
- Data/drive management – reconcile, declarative and state driven, better than ZFS rewrite
- High performance, no write hole RAID/EC – stabilized, now seeing production usage
- Performance work: parity or better than XFS on dbench (high concurrency mixed data/metadata workload)
- Heavily focused on: working with users, documentation, debug tooling, usability – making everything simple and smooth
- https://bcachefs.org/bcachefs-principles-of-operation.pdf
  130 pages of reference manual



Lifting the experimental label has gone exceptionally smoothly; we've still got bug reports coming in and there's always more to do, but the bug reports have been "boring", not cause for panic. As adoption grows, I'm always heavily focused on the rate of incoming bug reports and making sure that we don't grow too fast and have bug reports getting lost. To grow the community we need to be able to work with people that report bugs; when we can do that, that's what gets people hooked and interested in learning the system well.

Building that community of people who know the system, know how to debug it and can help out other new users, that's been the big focus; deployment has been slow and cautious but it's growing.

Reconcile: fully state driven data and device

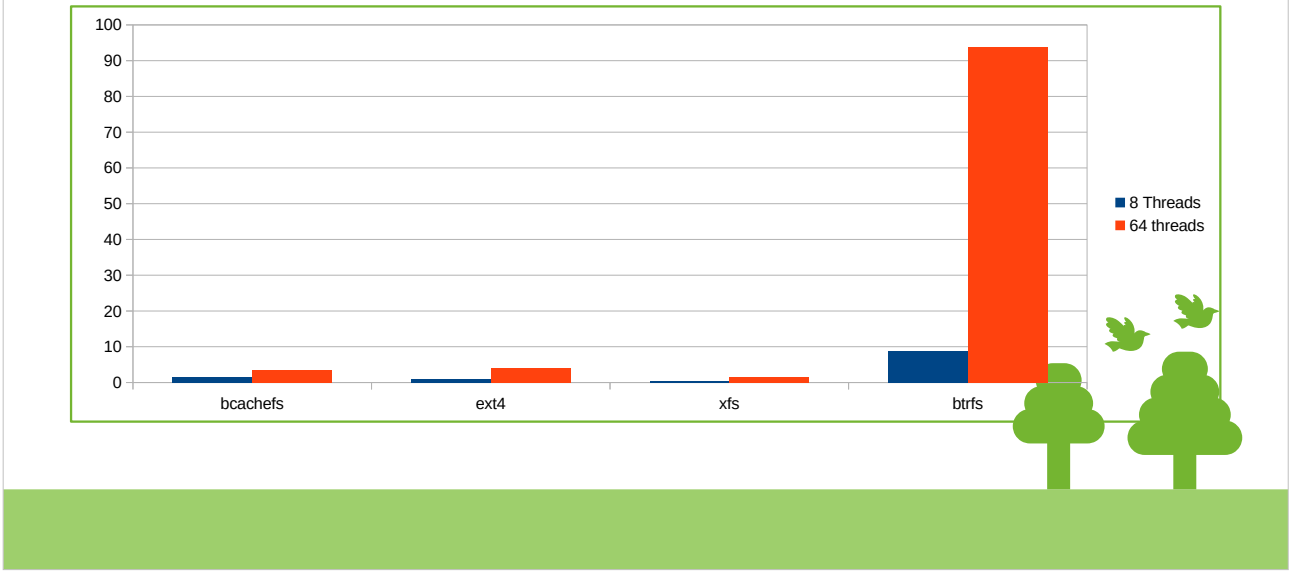


Dbench is a benchmark that simulates a fileserver, it does a lot of mixed reads and writes, with high concurrency, and mixing in a lot of metadata operations. XFS used to be king here; getting this result on a copy on write filesystem felt like a real achievement.

We've actually regressed a bit here since I last checked a few months ago – on the same hardware we were hitting 19.5 GB/sec, now about 16. But – matching XFS is still pretty good.

dbench at 48 clients is a particularly difficult test of multithreaded scalability; it hits data and metadata paths pretty equally, and throws in fsyncs as well.

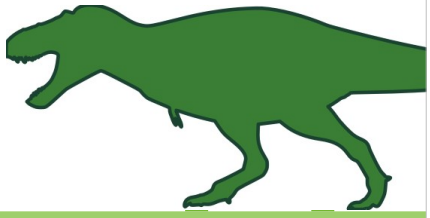
SQLite runtime (Phoronix test suite)



This SQLite benchmark is commit heavy – it stresses fsyncs very hard. I think last I checked, maybe six months ago, we were actually ahead of XFS at high client counts; the journal commit path that fsync hits is now fully lockless in bcachefs. I couldn’t resist mentioning the numbers to Dave Chinner when those optimizations landed; I’m curious if he went and landed some new optimizations for XFS – I haven’t checked the git history yet, but he’s pretty proud of the XFS CIL code :)

The limits of software engineering are not what we can imagine, but what we can debug

- Engineering at scale ends up being less about the code itself, and more about all the things that seem at first glance like overhead
- Automated testing, introspection and runtime in-the-wild debugging, error messages, keeping code organized and well factored
- And now: memory safety, formal verification
- Or more simply - always use the best tools you have available



I frequently tell people: I don’t care if code is “perfect” – but I do care if it can be debugged.

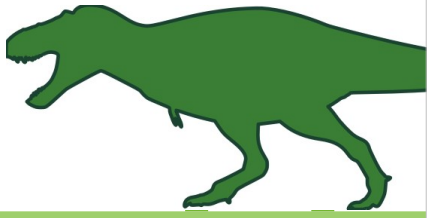
Can we find the relevant code in a hurry? Do we have the introspection to see what the system is doing? You can’t debug what you can’t see. If the system fails, will it give you an error message that tells you what you need to know to debug the issue – or will it fail by going off into undefined behaviour lala land?

All these things may seem like minor details in the moment, and they can take a lot of work to do well (especially good error messages) – but taken together, they mean the difference between bugs that are two weeks of panic, versus 15 minutes of “check this… ok check that… ok, I see the issue, here’s the patch”

This also might seem like a rather odd way of thinking

things we emphasize for reliability

- Automated testing – an entire cluster testing every commit, as it is pushed, with a massive and growing test suite
- Runtime debugging: all runtime state should be inspectable (even high rate in flight ops)
  (yes, all, also timestats; latency/jitter has long been opaque and it's only gotten somewhat better with bpf offcputime)
- Error messages should describe the entire operation that failed, and explain what we were doing and why

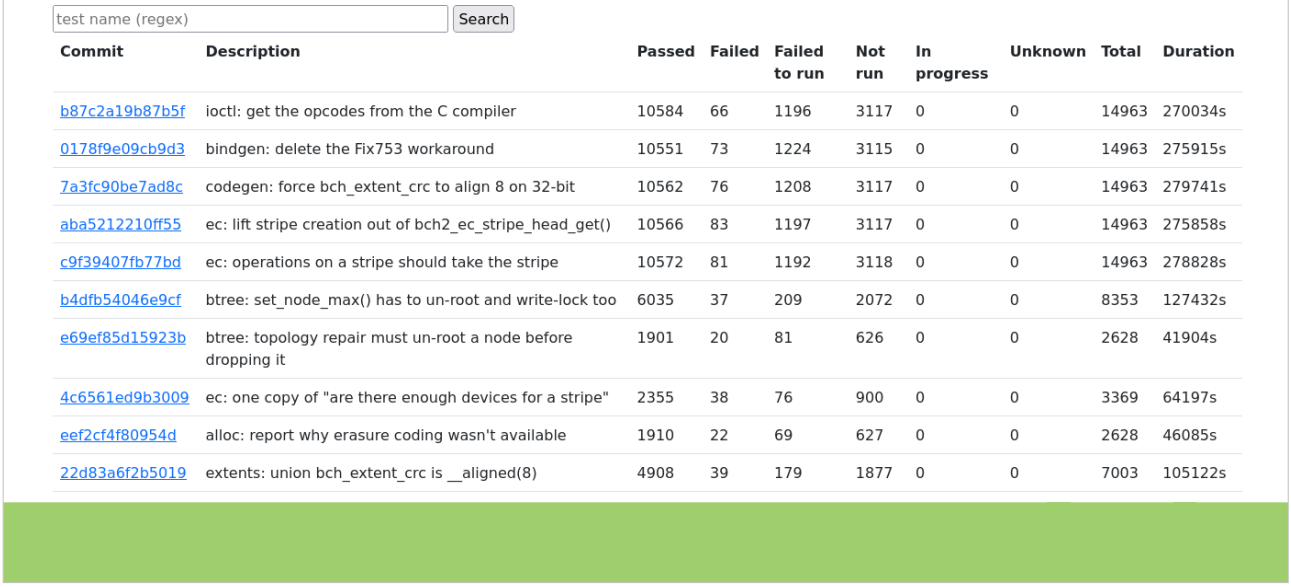


Testing: at scale, we have to think in terms of triage and data analysis – many difficult and critical bugs will only reproduce rarely, and the automated testing should gather as much data as possible and present it in a way that is as useful as possible

For much of the kernel, runtime introspection equates to "tracing" – but there's a whole lot more. For a modern filesystem there is a whole lot going on under the hood, and the entire state of the system really does need to be inspectable, at runtime, in the wild, easily and without a debugger. I have never regretted spending time on introspection code – even when I think I'm being paranoid or excessive, that code always ends up being used, and it only has to be used once or twice before the time investment pays for itself.

The talk isn't a tour of bcachefs introspection facilities

Automated testing: ktest



test name (regex) Search

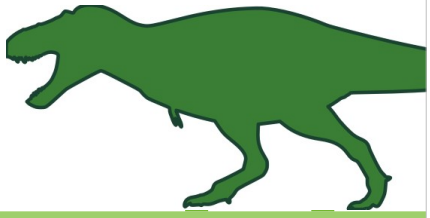
Commit	Description	Passed	Failed	Failed to run	Not run	In progress	Unknown	Total	Duration
b87c2a19b87b5f	ioctl: get the opcodes from the C compiler	10584	66	1196	3117	0	0	14963	270034s
0178f9e09cb9d3	bindgen: delete the Fix753 workaround	10551	73	1224	3115	0	0	14963	275915s
7a3fc90be7ad8c	codegen: force bch_extent_crc to align 8 on 32-bit	10562	76	1208	3117	0	0	14963	279741s
aba5212210ff55	ec: lift stripe creation out of bch2_ec_stripe_head_get()	10566	83	1197	3117	0	0	14963	275858s
c9f39407fb77bd	ec: operations on a stripe should take the stripe	10572	81	1192	3118	0	0	14963	278828s
b4dfb54046e9cf	btree: set_node_max() has to un-root and write-lock too	6035	37	209	2072	0	0	8353	127432s
e69ef85d15923b	btree: topology repair must un-root a node before dropping it	1901	20	81	626	0	0	2628	41904s
4c6561ed9b3009	ec: one copy of "are there enough devices for a stripe"	2355	38	76	900	0	0	3369	64197s
eef2cf4f80954d	alloc: report why erasure coding wasn't available	1910	22	69	627	0	0	2628	46085s
22d83a6f2b5019	extents: union bch_extent_crc is __aligned(8)	4908	39	179	1877	0	0	7003	105122s

Automated testing is worth thinking about; this is one of the things I work with and am looking at every single day. I find a lot of programmers don’t spend enough time on thinking about their tools, but the basic interfaces to the tools we use every day matter – this has a real effect on the quality of the code we’re able to produce.

And like I’ll be mentioning with other tools for building reliability, good automated testing is something that really feeds in on itself and changes the way you work. The bcachefs test suite is massive and constantly expanding; frequently I get contributions from people who may not be confident enough to fix the bug themselves, but are more than capable of writing a regression test. I can and do freely dump in regression tests for things that aren’t critical and so won’t be getting fixed right away (we always have to triage!) – and if the dashboard is good enough, they

bcachefs idioms to Rust

darray	Vec
*_to_text()	Display
printbuf	Formatter
closures	async/await, Arc
lots and lots of x-macros	Derive
errcode.h	ThisError (partly)
try()	?



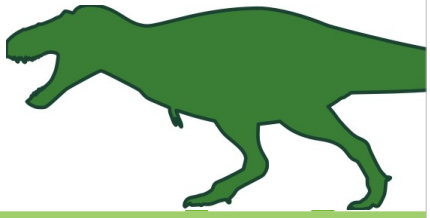
I’m showing this table early before we really get into the Rust stuff because a lot of what I’ll be showing are things that get dramatically easier in Rust. Rust metaprogramming in particular, coupled with standardized traits for formatting and display makes a lot of introspection dead simple. Later on we’ll start talking about the actual Rust conversion, and this starts to show how that conversion works in bcachefs and has been substantially mapped out.

The kernel has a pretty decent collection of data structures, but Vec is oddly missing – even in C, a vector is generally far preferable to a linked list. bcachefs does not use linked lists heavily, which avoids a lot of pin-type issues.

Good helpers for formatting text are essential, and bcachefs goes beyond what Rust natively has – printbufs have tabstops and indentation, both of

Error hierarchies

- Flattening errors (EINTR, EAGAIN) is a nightmare for debugability
- bcachefs philosophy: error codes should, wherever possible, be unique: printing the error code then identifies the line of code that threw the error
- This necessitates error heirarchies – any error code must be subtypable, and matchable with bch2_err_matches()
- bch_err_fn(fs, ret) – prints __func__ and the error name, which is frequently all we need



Rust hasn't caught on here, but introducing modern errcode.h errors to bcachefs was undoubtably the biggest single improvement to bcachefs's debugability. Unique error codes mean a simple one liner immediately tells you where the error was thrown and where the error was caught; frequently this is all we need to know to fix a bug.

Subtyping standard errno.h errcodes gives us easy compatibility with the outside world; bch2_err_class() returns the topmost ancestor and is called at module boundaries.

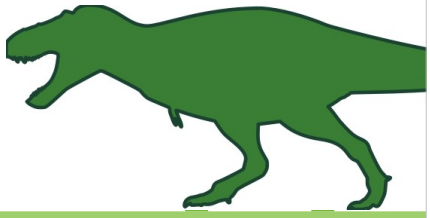
Error message examples

```
btree node read error at btree alloc level 0/0
  u64s 11 type btree_ptr_v2 SPOS_MAX len 0 ver 0  :
    seq 49f9e9d0fea8d6c9 written 15 min_key POS_MIN durability: 1
    ptr: demo1.img 0:25:0 gen 0
  demo1.img node offset 0/15: bad magic: want be57c3d2ece5d2ac, got b4aa67744ec56c8e
    demo1.img io: btree_node_validate_err
  flagging btree alloc lost data
  scheduling recovery pass check_topology (2)
  scheduling recovery pass check_allocations (8)
  scheduling recovery pass check_alloc_info (15)
  error btree_node_validate_err
error reading btree root btree=alloc level=0: btree_node_read_error, fixing
```

Individual error messages include

- How we retried
- Repair actions

The goal is to make the underlying logic of the system clear and accessible



This is harder than it looks, and producing error messages like this often required fundamentally restructing code, because logic that normally would be implicit in the control flow now has to be remembered and described within the type system.

But I've also found that the restructuring for error messages often improves and clarifies the code in the process; what was implicit is now explicit, making it more readable and easier to follow, often leading to improvements in actual error handling behaviour in the process.

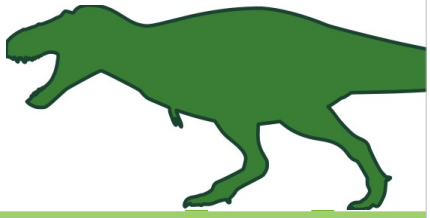
There's a deep principle underlying a lot of this work, which is that more transparent code is almost always better code.

Tracing in bcachefs

The bcachefs approach:

- Make tracing as simple as possible, for the end user and the developer
- Don’t bother with TRACE_EVENT(); all tracepoints are strings, so we can use our to_text() functions or Display trait
- Get as much mileage as possible out of a tracepoint definition: every defined tracepoint also has a corresponding always on counter

Then we can include tools that let users go from system counters to individual tracepoints with a keystroke



Conventional kernel style tracing suffers from having poor UX, both for the end user and the developer. And this is a problem, because there’s incredible value in having a well curated set of tracepoints that capture the logic of the system at runtime at key decision points – but poor UX creates friction, and that friction makes it hard to iterate the way you need to, to produce that set of curated, refined tracepoints.

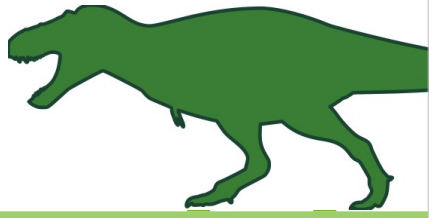
As with anything else – if it’s worth having, it’s worth doing well, and to do it well you need people using it and iterating on it.

bcachefs fs top

Every counter has a corresponding tracepoint; Enter drills into a live trace of the selected event, scoped to this mount.

g:quit Enter:live-trace h:human-readable Tab:page ↑↓:scroll PgUp/PgDn 1-9:interval [counters] [devices]

	1/s	total	mount
► sync_fs	1/s	7	207
data_read_hole	0B/s	0B	2.00M
data_read_reuse_race	0/s	0	39
data_write	125M/s	1.02G	15.5G
data_update_pred	135M/s	1.04G	14.0G
data_update	75.0M/s	584M	9.16G
data_update_in_flight	6/s	48	1144
data_update_fail	0B/s	0B	452K
data_update_read	75.0M/s	583M	9.15G
data_update_write	75.1M/s	583M	9.15G
data_update_key	74.9M/s	582M	9.13G
data_update_key_fail	44.0K/s	368K	13.1M
evacuate_bucket	4286/s	34.1K	445K
copygc	268/s	2131	27.9K
bucket_discard_worker	287/s	2151	19.5K
bucket_discard_fast_worker	1/s	3	280
bucket_discard	10.4K/s	76.7K	1.09M
bucket_alloc	9715/s	79.4K	1.22M
bucket_alloc_fail	75/s	466	32.4K
sectors_alloc	304M/s	2.42G	37.2G
btree_cache_scan	0/s	0	17
btree_cache_reap	0/s	0	1130
btree_cache_cannibalize_lock	3222/s	26.8K	403K
btree_cache_cannibalize_lock_fail	201/s	1990	28.1K
btree_cache_cannibalize_unlock	3222/s	26.8K	403K



As I mentioned, this is a live view where users can scroll through the list of events, hit enter – and see the live trace for that event. Users love this sort of thing.

‘perf top’ can sort of do this, but the problem is with standard kernel tracing ‘perf top -e bcachefs:*’ will OOM if you just leave it running, even thouugh all it wants is counters.

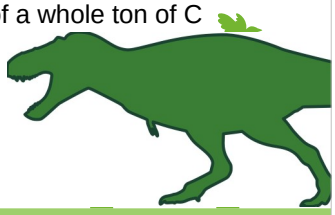
Always on counters are cheap and immensely useful. You don’t want introspection that’s off by default and requires users to plan ahead, you want users to be able to notice that something is wonky and just have a look.

There’s a lot of stuff like this in bcachefs; ‘bcachefs fs timestats’ is another. Good place to dig around and explore.

The Rust conversion

- “Do as much ahead of time as C refactoring, so the conversion can be as mechanical as possible”
- Codebase organization, with good module boundaries, so we can convert file by file
- goto → CLASS()

  Prep work removed 2k gotos (currently at 584, vs. 2742 for XFS, ~75% of files goto free)
- Automated testing with good coverage, good asserts are key
- Pragmatism: we accept somewhat more unsafe than ideal –if it gets rid of a whole ton of C
- Refactor, refactor, refactor, to make life easier later



The eventual Rust conversion for bcachefs has been in the planning and prep stage for a long, long time.

Prep is key. If you get all the weird lifetime issues sorted out beforehand, life will be so much easier. People can be ever so creative with pointers in C – but there’s also nothing stopping you from writing C where the pointers and lifetime rules work basically the same as they would in Rust.

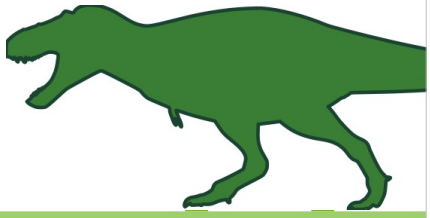
CLASS() helped greatly, and bcachefs uses that heavily. It’s not true RAII, but it can cover 80 or 90% of code and lets you write error handling that’s much closer to what you’d do in Rust.

Good test coverage is also, obviously, important.

Rust conversion status

- Userspace: done (30k loc)
- Core APIs (btree transaction, others) – “good enough” to done
- Rust interfaces to APIs moved into kernel – April/May
- Most users now on Rust-enabled builds, with actual in-kernel bcachefs Rust code since bcachefs is DKMS, this means build servers – lots of build servers
- Next up: build pipeline work on remaining distros (Proxmox)
  would be lovely if we can get packaging of .rlibs and .rmetas standardized

  Then: poll the userbase to make sure no one’s missing Rust builds before it becomes a hard dependency, and the bulk conversion starts
- First target - fsck

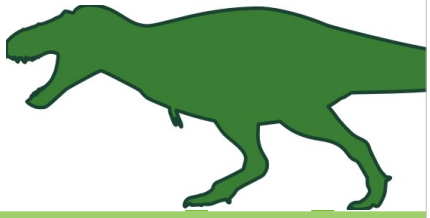


This is a large - 200k loc - codebase with an established userbase who expects things to stay stable, so we have to take a balanced approach: we want to minimize the risk of breakage, but we also have a lot of code to convert – we don’t want to spend more time than we have to with the codebase in flux.

We’ve gotten quite confident with automated testing and QA, and having already done the userspace conversion has already given us some practical experience. The userspace code was actually more prone to breakage than I expect the majority of the kernel code to be: a lot of weird glue code with mediocre test code, a lot of FFI issues. In comparison, the kernel code tends to be more regular, use fewer interfaces (a lot of code, especially fsck, uses essentially only the btree transaction interface), and has excellent test

What Rust opens up (what we like in bcachefs)

- Error handling! Result<>, ?
- Metaprogramming
- The build and module system
- We finally have a path to practical formal verification - the borrow checker makes the rest tractable



There's a lot more than just memory safety that really benefits filesystem work. Memory safety is actually probably one of the smaller things for bcachefs – a tiny fraction (10%?) of bug reports are memory safety bugs. Rust has a ton of other quality of life improvements, and those are going to have a bigger impact for bcachefs – especially error handling.

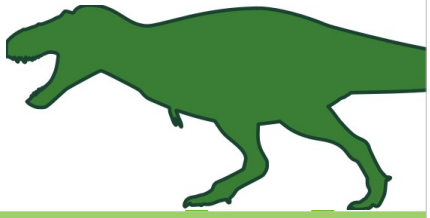
The depth of knowledge and experience the creators were drawing on when designing the language really shows. There are a lot of deep lessons that various programming language communities have learned from real world experience over the past 50 years, and to this day most programming languages haven't learned them – but I really can't think of anything Rust got seriously wrong. (Maybe making mem::forget safe).

Just having algebraic data types in a mainstream

Verification in bcachefs – the beginnings

This is not your normal everyday btree – this is the sort of code we care about:

```
static struct bkey_packed *bset_search_tree(const struct btree *b,
                                const struct bset_tree *t,
                                const struct bpos *search,
                                const struct bkey_packed *packed_search)
{
        struct ro_aux_tree *base = ro_aux_tree_base(b, t);
        struct bkey_float *f;
        unsigned n = 1;

        do {
                f = &base->f[n];
                if (likely(f->exponent < BFLOAT_FAILED)) {
                        unsigned l = f->mantissa;
                        unsigned r = bkey_mantissa(packed_search, f);

                        if (likely(l != r) || !bkey_mantissa_bits_dropped(b, f)) {
                                n = n * 2 + (l < r);
                                continue;
                        }
                }

                struct bkey_packed *k = tree_to_bkey(b, t, n);
                int cmp = bkey_cmp_p_or_unp(b, k, packed_search, search);
                if (!cmp)
                        return k;

                n = n * 2 + (cmp < 0);
        } while (n < t->size);

        unsigned inorder = __eytzinger1_to_inorder(n >> 1, t->size - 1, t->extra);
        if (likely(!(n & 1))) {
                --inorder;
                if (unlikely(!inorder))
                        return btree_bkey_first(b, t);

                f = &base->f[eytzinger1_prev(n >> 1, t->size - 1)];
        }

        return cacheline_to_bkey(b, t, inorder, f->key_offset);
}
```



I’ve always wanted to tell this story :)

So this slide is the actual core lookup code in bcachefs, essentially unchanged since bcache, over 15 years ago.

The story is, early on in the development of bcache – the predecessor to bcachefs – we ran into a problem: binary search is slow.

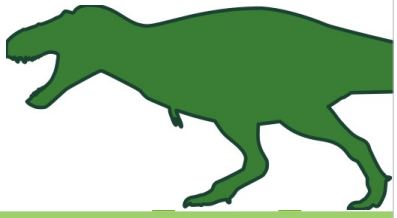
It’s the worst possible algorithm for the memory prefetcher.

We ran into this because at the time Google was building SSDs that were far faster than anything available on the market, and as a result bcache became the first production software to deploy eytzinger trees. Before eytziinger trees were published in the literature, bcache was using them

Verification in bcachefs today

100% runtime asserts in debug mode, formalized(-ish) and standardized; wherever possible we write assertions *on data types*
the lesson we've learned is that writing assertions in a "formal verification" mode of thinking leads to much better assertions that can catch classes of bugs with somewhat less exhaustive testing

- Btree transactions, iterators, locking – heavy usage
  `bch2_btree_path_verify()` and many others
- Input validation – if we can reject invalid input, the rest of the code becomes much easier to reason about
  not really formal verification, but has to be formalized in much the same way; sets the boundary of formal verification
- The big gap – filesystem level invariants
  invariants that btree transactions have to uphold – no practical way to address these without real formal verification



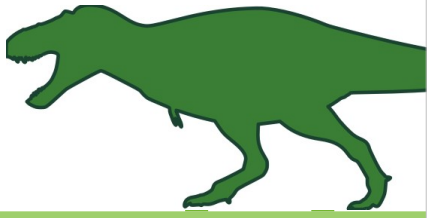
The approach of having a standard way of writing assertions on data types, that are always true, has proven to be particularly valuable over the course of development. Many btree bugs that might have otherwise been a nightmare turn into "we're not sure where the invariant was violated? just shotgun the verify function all over until we find it".

Doing this requires a lot of discipline and hygiene; you have to be clear on what the invariants are, sections of the code where they're violated have to be small and clearly delineated (just like unsafe!), and you have to be dilligent about writing new assertions in this standardized way and keeping the verify functions up to date and comprehensive.

Basically, this leads into proto-formal-verification – much like in Gary's talk he was talking lots and lots about invariants – in formal verification, the thing

Formal verification intro

- Formal verification in conventional systems languages: technically possible, effectively infeasible - SeL4
- More successful in: functional languages - CoQ, CompCert
- And languages with a good type system - Rice's Theorem, Idris
- Rust has a great type system
- And the borrow checker makes it a functional language in disguise...



Sel4 was an early pioneer in formal verification, a fully veriified version of the L4 microkernel. But the kernel itself was still implemented in C, and the verification was written as proofs about the behaviour of C code. Sel4 demonstrated that formal verification of the key component of the operating system was possible, but it was a massive undertaking – formal verification of real production systems was going to have to wait for better tools.

CompCert was another notable milestone, a fully verified C compiler, end to end, that could compile real programs. Here, the advance was showing what could be done in a language meant for formal verification – CoQ (now Rocq) was developed for proving mathematical proofs. But wasn't developed as an implementation language.

Rice's theorem deserves a real mention, because it's

Formal verification in Rust

- Rust solves the most immediate problems for formal verification of systems code: it eliminates unrestricted side effects, and it has a real type system
- And now, formal verification in a mainstream systems language is finally becoming an actively explored topic, just starting to be used in production
- Verus
- Aeneas

The borrow checker is about more than memory safety: W^R makes Rust a functional programming language in disguise, since mutable parameters are isomorphic to returning a new version of the parameter, and the escape hatch – UnsafeCell – is clearly specified in the type system.

I was just mentioning in the previous slide how a good type system is incredibly important for formal verification; the biggest thing that separates real type systems from the not so good is algebraic data types – which Rust also conveniently included. Like the borrow checker, algebraic data types help to make large classes of bugs unrepresentable, and affine data types are also critical for correctly modelling real world problems.

I should also point out that the borrow itself is a type system contruct; it's something that you reason

Formal verification in bcachefs

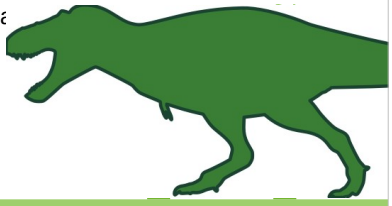
- Primary approach: treat existing assertions as proof obligations, and fill out the proofs
  Note that a great many of our asssertions are already assertions on types
- Useful analogy: gradual typing
  When adding type annotations to a dynamically typed language, you don't do it all at once – you do it where it's useful. We can do the same thing.
- This should become another tool for responding to bug reports
  Today: "Where are we missing test coverage? Where are we missing introspection?"
  Tomorrow: "Where is our behaviour under specified? What proofs are we missing"
- The long term goal in software engineering: better abstractions, so we can make our code better specified and better behaved
  Better type systems, elimination of undefined behaviour, and now methods to formally and rigorously prove the behaviour of new abstractions

We don't know what the end game is yet for formal verification and bcachefs. Is a 100% fully formal verified filesystem practically achievable? A modern filesystem is a massive beast – I think we'll get there eventually, but I couldn't tell you anythinig about what sort of timeline that will be on, or what the end result will look like, or what new tools and techniques we'll need beyond what we're looking at now.

What I can tell you is that there are a lot of components that will benefit greatly from formal verificatiion, where we've already got a lot of groundwork that makes it a natural step: the core btree code is an obvious example. Verus also has tools for proving the correctness of multithreaded code – the journal could use that; the journal is a whole pile of intricate lockless state machine. The allocator state machine – the way buckets cycle

The expanding world of Rust – some predictions

- CPU performance has been held back by C: CPU designers would like to go wider (VLIW), but unrestricted pointer aliasing is a problem
- A lot of code could be running entirely out of registers, if only enough code was written in a language with a smarter memory model…
- Open source, high performance CPU designs are starting to appear (XiangShan)
- GPU programmers still have to write to these things called “threads” and “warps” (which are different from CPU threads)
- But we have these things called “optimizing compilers” – if only people were ta
them
- https://goals.rust-lang.org/2026/high-level-ml.html



- People get stuck on how incredibly expensive it is to design CPUs, but you haven’t needed to own a fab to build a CPU in 20 years, and now even the core designs are starting to be open sourced. XiangShan – open source, RISC-V, 3/4s of the specint/ghz of AMD/Intel, and written in Scala. Open source really does rule all the core infrastructure we rely on, and that share is still growing, and that’s making new engineering projects possible on a scale that weren’t in years past. I expect the world of CPU design to become a lot more interesting in the coming years, enabled by Rust.

- The missing piece for making “Rust on the GPU” really powerful is MIR → MLIR, and this is now being worked on. The AI/ML community has an optimizing compiler for high level algoriithms, Triton, but it’s Python and LLVM. Doing this in Rust and taking advantage of the Rust type system will be far

Systems programming is changing

- New tools and new methods are going to reshape the way we work; less time debugging, more time to think about the structure of what we're trying to do
- Systems of the future will be cleaner, easier to understand, easier to work on, and far more capable – if we build it

I think everyone in this room is already aware of just how much Rust is taking over, and it's been pretty incredible to watch the whole thing unfold over the past 10 years. We still have a long ways to go to get our legacy codebases converted, it's a process that still be going for a long time in the future, but we're starting to feel concretely how much easier our jobs can be when we're building on top of tools that we can trust.

Remembering that can really help when at times we're deep in the midst of dealing with legacy codebases, looking for a path forwards. Seeing the work we're doing come together, comparing what we're doing today to what day to day engineering was like 5, 10, 15 years ago – I'm confident that the Rust community is building a foundation that future generations of engineers will appreciate.