Announcing Continuous Memory Profiling for Rust

Debugging memory leaks and usage systematically in Rust

Frederic Branczyk & Brennan Vincent

December 20, 2023

Rust Pprof Profiling Memory Heap OOM Out-Of-Memory Memory-Leak Leak OOMKill Jemalloc

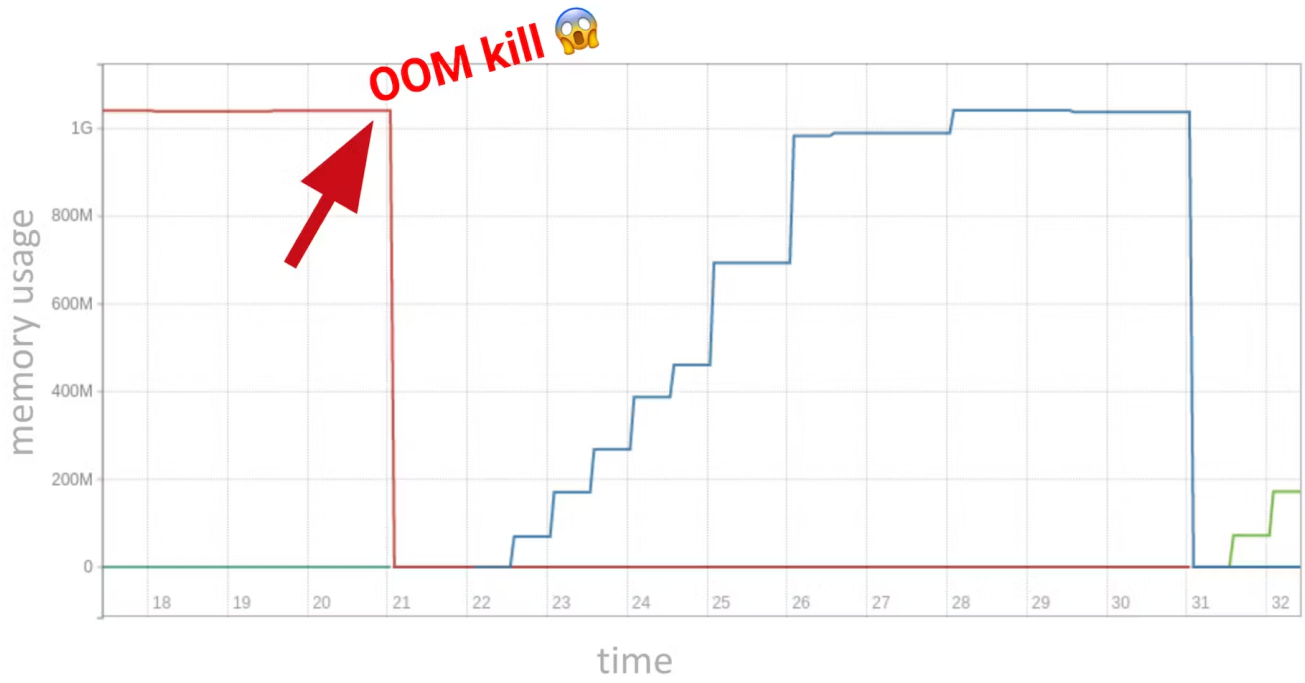
Along with CPU profiling, memory leaks in high-stress situations remains a difficult task for software engineers and SRE teams.

Today, in collaboration with Materialize, we're excited to announce rust-jemalloc-pprof, a library to export jemalloc's heap profiling capabilities in the pprof format. This library allows either one-off troubleshooting of heap memory usage using the regular pprof toolchain, or since Polar Signals Cloud offers compatibility with any pprof formatted data, it can also be used to continuously record heap profiling data and query it throughout time. There is nothing that ties this library to Polar Signals, it only happens to expose the format Polar Signals Cloud supports.

Why memory/heap profiling?

In comparison to CPU, memory is a much less elastic resource. When memory runs out, a process is killed by the Linux kernel, as opposed to running slower. That's not to say that CPU contention isn't also serious, but the failure modes are very different.

A situation many have experienced is the above-mentioned OOMKill scenario.



The bottom line? When leaking memory, it's crucial to have memory profiling available quickly, so you can understand where memory is allocated that's never released.

Of course, it's just as useful to understand and improve baseline usage or spikes.

How to use it

Much of the hard work has already been done by jemalloc, as it has built-in heap memory profiling. What rust-jemalloc-pprof does is give you a high-level API that under the hood interacts with jemalloc to produce a heap memory profile in the jemalloc format, and then translate it to pprof.

The library requires using tikv-jemallocator, with the profiling feature enabled in your `Cargo.toml`.

```
tikv-jemallocator = { version = "0.5.4", features = ["profiling" "unprefix
```

> *Note: While not strictly necessary, we recommend also enabling the "unprefixed_malloc_on_supported_platforms" feature.*

Enable the allocator and configure it to enable profiling.

```
#[cfg(not(target_env = "msvc"))]
#[global_allocator]
static ALLOC: tikv_jemallocator::Jemalloc = tikv_jemallocator::Jemalloc;

#[allow(non_upper_case_globals)]
#[export_name = "malloc_conf"]
pub static malloc_conf: &[u8] = b"prof:true,prof_active:true,lg_prof_sampl
```

> *If you do not use the "unprefixed_malloc_on_supported_platforms" feature, you have to name it "_rjem_malloc_conf" it instead of "malloc_conf".*

2^19 bytes (512KiB) is the default configuration for the sampling period, but we recommend being explicit. To understand more about jemalloc sampling check out the detailed docs on it.

The `dump_pprof` function is how you interact with this library. It creates a temporary file, instructs jemalloc to dump a profile into it, read it and convert it to pprof. Then it gzips the pprof protobuf representation, and returns the result.

We recommend borrowing the idea from the Go ecosystem, to expose this as an HTTP handler, so either automated mechanisms can continuously scrape them, or they're easily available in production any time you need to perform heap profiling.

It could look something like this:

```
#[tokio::main]
```

```
async fn main() {
    let mut v = vec![];
    for i in 0..1000000 {
        v.push(i);
    }

    let app = axum::Router::new()
        .route("/debug/pprof/heap", axum::routing::get(handle_get_heap));

    // run our app with hyper, listening globally on port 3000
    let listener = tokio::net::TcpListener::bind("0.0.0.0:3000").await.unw
    axum::serve(listener, app).await.unwrap();
}

use axum::http::StatusCode;
use axum::response::IntoResponse;

pub async fn handle_get_heap() -> Result<impl IntoResponse, (StatusCode, S
    let mut prof_ctl = jemalloc_pprof::PROF_CTL.as_ref().unwrap().lock().a
    require_profiling_activated(&prof_ctl)?;
    let pprof = prof_ctl
        .dump_pprof()
        .map_err(|err| (StatusCode::INTERNAL_SERVER_ERROR, err.to_string()
    Ok(pprof)
}

/// Checks whether jemalloc profiling is activated an returns an error res
fn require_profiling_activated(prof_ctl: &jemalloc_pprof::JemallocProfCtl)
    if prof_ctl.activated() {
        Ok(())
    } else {
        Err((axum::http::StatusCode::FORBIDDEN, "heap profiling not activa
    }
}
```

Then running the application, you can curl the endpoint and explore it with any pprof compatible tooling.

```
curl localhost:3000/debug/pprof/heap > heap.pb.gz
pprof -http=:8080 heap.pb.gz
```

rust-jemalloc-pprof under the hood

The library provides two main features: a Rust interface to jemalloc's profiling control functions, and a function to export jemalloc profiling data to `pprof` format.

The interface to jemalloc control functions is straightforward: just a simple Rust wrapper over the corresponding C functions. When profiling is activated, jemalloc begins recording the stack trace on allocations approximately every `2^n` bytes, where `n` is the value for `lg_prof_sample` configured in `malloc_conf` as described above. Since jemalloc does not actually symbolize the resulting stacks, but only collects addresses, this is very fast. Materialize has found that setting `lg_prof_sample` to 19 is an ideal tradeoff between barely measurable performance impact and faithful enough profiles, and recommends keeping this as the default value unless specific evidence suggests otherwise. Whenever memory is deallocated, jemalloc checks whether the stack trace corresponding to its allocation was sampled, and if so, removes that stack trace from its profile. Thus, each profile represents only the memory that is still allocated without having yet been freed.

The `JemallocProfCtl::dump_pprof` function is the second main piece of `rust_jemalloc_pprof`. It instructs `jemalloc` to dump its profile in its internal format, then uses the same formula as in jemalloc's `jeprof` tool to convert from this format to the number of bytes in each stack. (These formulas are a bit less trivial than they sound, which is why we don't recommend trying to interpret jemalloc profile dumps directly; see the comments in `parse_jeheap` if you're interested in the fine details.) Finally, it converts that "weighted stack" format to Polar Signals' native `pprof` format, which is basically a list of stack traces as well as the executable memory mappings needed to interpret them, all in a particular Protobuf schema. The `pprof` files generated contain only raw memory addresses, not human-readable symbols, as it is intended that symbolization happen on the backend, rather than in-process. Thus, users need to upload the debug symbols for their programs to Polar Signals. Materialize does this during their CI process as part of building release binaries.

Memory Profiling at Materialize

This library was originally developed at Materialize, and we at Polar Signals are maintaining it together with Materialize going forward.

> *"Continuous memory profiling with Polar Signals Cloud has allowed us to diagnose memory issues in production that were all but impossible to debug before—whether they're OOMs that happen in seconds or slow memory leaks that accrete over several days. I can't imagine working without it anymore." - Nikhil Benesch, CTO at Materialize.*

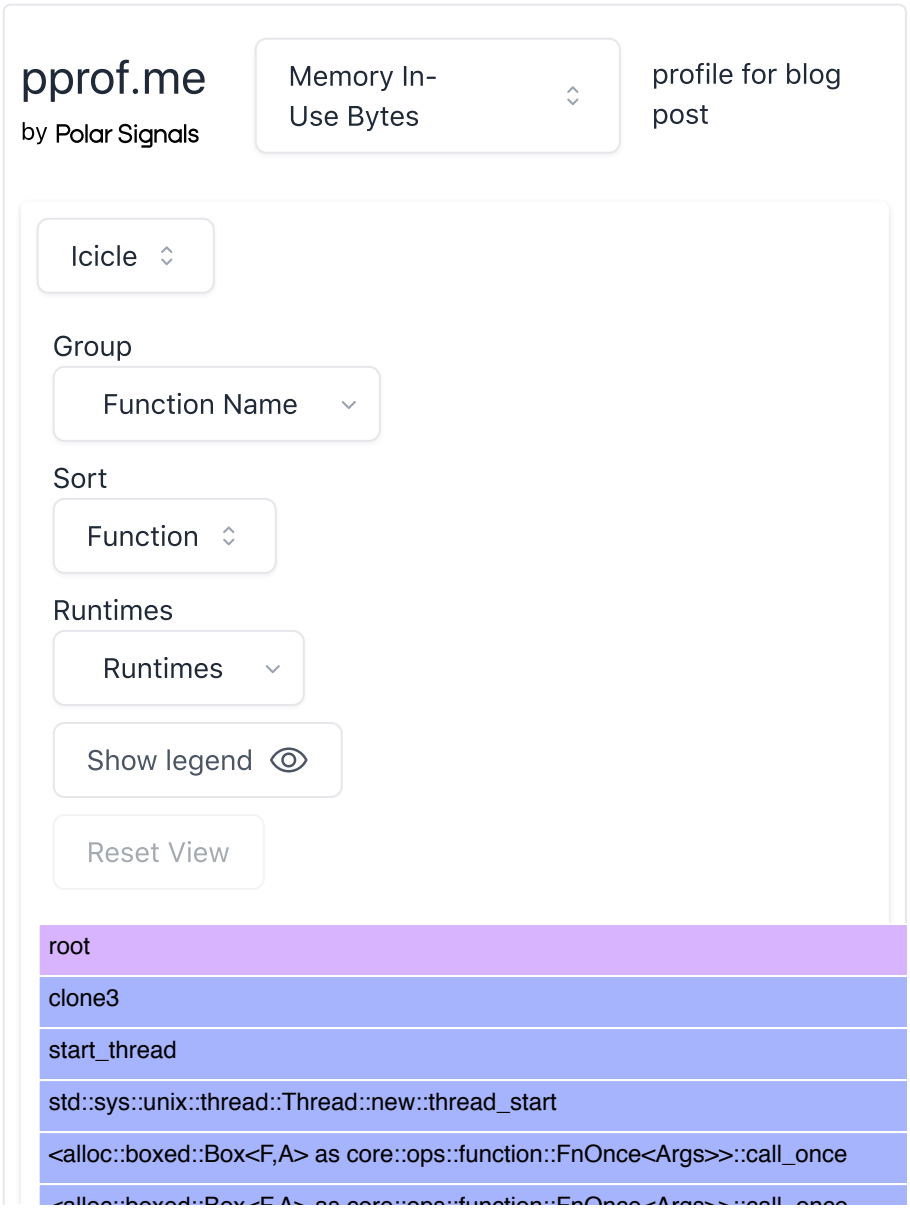
Materialize aims to build an operational data warehouse "providing interactive access to up-to-date data", by incrementally maintaining the resuls of analytical queries as new data arrives. Since the product's value depends on results being updated quickly, as much data as possible

must be kept in memory before spilling to (much slower) disk. Thus, memory has always been the primary constrained resource for most of Materialize's customers, and any optimization to decrease memory use is of particularly high value.

Before switching to Polar Signals, Materialize used a custom-built tool to render allocation profiles. This tool had several drawbacks. First, it required direct HTTP access to a running Materialize environment, meaning it could only be used by the small set of on-call engineers with privileged production access. Second, it required profiling to be turned on manually, meaning data about past events before the engineer began investigating was not available. Third, it performed significant work in the database process itself, possibly causing slowdown and disruption for the end user. Fourth, its UI was counterintuitive and difficult to use.

Since Materialize was already a happy user of Polar Signals for CPU time profiling, they knew the situation could be much better, and so developed the `rust_jemalloc_pprof` library to get memory profiling data into Polar Signals' native `pprof` format. By doing so and integrating with Polar Signals, all the issues mentioned above have been solved: allocation profiles are available to all engineers, collected continuously and cheaply, and easy to access and understand. This has greatly broadened the use of allocation profiling among Materialize engineers and enabled faster investigation of real issues.

As an example of a real-world allocation profile, take a look at this icicle graph:



Here most memory was allocated in batch creation and merging in differential-dataflow (Materialize's core computation engine). This is a typical usage pattern.

Start Profiling Memory!

Memory profiling is an important tool, and we're excited to have brought it to Rust together with Materialize. Check out more details, as well as usage examples, on the GitHub repository.

If you want to try our continuous memory profiling, we invite you to a free 14-day trial of Polar Signals Cloud.

Happy profiling!

Discuss:

Related Posts

Embracing performance profiling: Canonical enables frame pointers in Ubuntu 24.04 LTS

December 13, 2023 | Frederic Branczyk

Frame Pointers DWARF DWARF-Based Unwinding Ubuntu

New pprof.me Features: Profiling Data Comparison and Folded Stacks format support

December 07, 2023 | Frederic Branczyk, Manoj Vivek

Pprof Profiling Sharing

The Cost of Go's Interfaces and How to Fix It

November 24, 2023 | Frederic Branczyk

Go Golang PGO Observability Continuous Profiling EBPF