[PostgreSQL] pgagroal: I/O layer

0. Contributor Information

Name: Henrique A. de Carvalho
Email: decarv.henrique@gmail.com
GitHub: decarv
Languages: Portuguese, English
Country: Brazil (GMT -03:00)
Who Am I: I earned a Bachelor's degree in Computer Science from the University of São Paulo (2020 - 2023) and currently work as a Backend Software Engineer at a medium-scale Investment Fund. Throughout my bachelor's program, I did bunch of different stuff, from algorithms and data structures, to systems engineering, to machine learning theory and applications, computer vision, computer graphics and application development and benchmarking. Nevertheless, the things that live in my heart are systems engineering, performance-oriented development and computer graphics, which are things I rarely have the opportunity to do. Therefore, I see contributing to open-source as a great way to stay close to what makes me happy in software engineering and, of course, to be able to provide good software for free to other humans. Specifically, my interest in pgagroal stems from its system program nature and its promise of high-performance. My main objective with this project is to learn from the community and assist pgagroal in achieving unparalleled performance among connection pools.
Contributions: #408, #411, #427, #431

1. Synopsis

The project consists of replacing the I/O Layer of pgagroal, today highly dependant on libev library, for a pgagroal's own implementation of this I/O Layer. The so called I/O Layer is an event loop abstracted by libev.

The motivation behind this project is because libev is not being maintained any longer. Therefore pgagroal needs an efficient (i.e. maintainable, reliable, fast, lightweight, secure and scalable) implementation of an event loop that can be maintained by the pgagroal community.

Currently, pgagroal depends on libev to (a) watch for incomming read/write requests from its connections in a non-blocking fashion; (b) launch timer events; and (c) watch signals.

In Linux, these functionalities may be optimally achieved by using io_uring (introduced in Linux kernel 5.1). io_uring is a communication channel between a system's application and the kernel by providing an interface to receive notifications when I/O is possible on file descriptors. io_uring is accessible to system applications through liburing, which is a library that contains helpers for setup of io_uring. A successful Linux implementation of an efficient event loop for pgagroal necessarilly utilizes io_uring -- as well as other Linux I/O interfaces (e.g. stdio) -- for efficient I/O. For cases where io_uring may fall short, Linux has other options that may replace it, such as epoll.

In FreeBSD, these functionalities may be optimally achieved by using kqueue (introduced in FreeBSD 4.1). kqueue is an event notification interface that allows monitoring of multiple file descriptors. Like io_uring in Linux, kqueue enables non-blocking I/O operations, but it is specifically designed to fit into the FreeBSD kernel's event-driven architecture.

The objective of this proposal is to provide a plan to achieve such efficient implementations in both Linux and FreeBSD, which shall be, at the end of this program, at least as efficient as libev, but fully maintained and controlled by the pgagroal community.

2. Proposal

As mentioned before, my objectives with this proposal are to implement an efficient event loop for pgagroal. In order to accomplish this, I could benefit from dividing the implementation into two phases: (a) Experimentation (Phase 1); (b) Continuous Implementation and Profiling (Phase 2).

For *Phase 1*, I propose the implementation of an interface containing a simple abstraction for an event loop (with io_uring, for Linux, and with kqueue, for FreeBSD).

The result of this first phase would be a maintainable and small footprint event loop that suffices for pgagroal specific uses, potentially (but not necessarily) resulting in minimal changes in functions and behavior of the main code -- as the implementation would work as an interface for watching file descriptors, timers, and signals.

For *Phase 2*, I propose the definition of tests and profiling (for speed and memory) for pgagroal's new event loop, done in different settings, enabling comparison between previous and future versions.

The idea here is to accurately measure resource utilization for pgagroal in areas we believe are important, so that we can ensure that the new implementation of pgagroal is actually going in the right direction.

The specific benchmark criteria (performance metrics) should be discussed with the community prior to the development of the strategy, but this should include attempts to measure latency, throughput, CPU usage, and memory footprint.

Measuring resource utilization will enable the identification of bottlenecks and places where pgagroal can benefit from optimizations while allowing for the measurement of potential optimizations implementations.

The measurements made propose diving deeper into improvements that could be made to the simple event loop implementation of the previous phase. Here I intend to investigate the potential necessary changes in the structure of the main code, considering other optimizations (e.g., reducing system calls, investigating new features introduced in `io_uring_sqe` fields, kernel-side polling, cache performance, memory layout, vectorization).

2.1. Phase 1 Details

The Phase 1 implementation depends on me knowing how io_uring and kqueue work, and understanding how libev functions are used throughout the code, so I can determine how they will be replaced.

2.1.1. Linux implementation

io_uring is a communication channel between the application and the kernel, functioning as an asynchronous I/O interface. It works by placing I/O submission events and I/O completion events into two queues (ring buffers) that are shared between the application and kernel. io_uring has the potential to reduce system calls and implement asynchronous I/O. Much of the complexity of managing these data structures is abstracted by a library called liburing. [1] Documentation on this library can be found at [2].

All of pgagroal's I/O layer can benefit from io_uring operations, such as writing to and reading from sockets, but the networking event loop part of pgagroal can also benefit from io_uring.

A simple webserver example can be found at [3], and pgagroal's event loop can use the same structure.

io_uring can be used in the server as a replacement for epoll, but this is not as straightforward as it may seem, and there is a lot of room to further optimize the code by using io_uring code and its new features. [4]

In fact, io_uring developer Jens Axboe created a document [5] in 2023, presenting recipes for io_uring and networking. As Axboe points out in this document, simply replacing epoll with io_uring will work, but doing so "does not lead to an outcome that fully takes advantage of what io_uring offers". All of these features that could be used by pgagroal in different places, such as (a) batching, as networking can have dependent operations that can all be waited to be ready at the same time (such as accept followed by recv); (b) multi-shot, as networking can have recurring operations (again, accept and recv); or (c) socket state awareness. The usage of further features mentioned in this document should definitely be evaluated.

On another note, features continue to be implemented into io_uring, which makes it an exciting feature to add to a project. As an example, one feature particularly interesting for pgagroal's process model is `io_uring_spawn`, presented in 2022. This feature could potentially reduce the process spawning time, but it is still under active development. [6]

2.1.2. FreeBSD Implementation

For FreeBSD, the transition to a native I/O solution centers around kqueue. Similar to io_uring, kqueue enables non-blocking I/O handling by using queues shared by the kernel and userspace and by supporting kevent for communication. It supports a wide range of event types, specified as *filters*, including file descriptors readiness, timers, and process and signal monitoring.

This document [7] contains the API specification for kqueue, as well as an example of event handling. The specific documentation can be found at [8]. As the author of the document explains, and similar to io_uring, simply adapting the event loop to work with kqueue as it would with poll will not yield the best results, and applications should be rewritten to fully utilize kqueue's capabilities.

kqueue is targeted specifically at I/O multiplexing and, differently from io_uring, should be used solely for the event library. I could not find reference to any features designed to enhance networking usability, as with Axboe's document, but the aforementioned document provides enough examples to implement for pgagroal's needs.

Key to the FreeBSD implementation will be leveraging kqueue's strengths, such as efficient timer management and fine-grained control over event monitoring, to optimize pgagroal's event-driven architecture.

2.1.3. Implementation Details

As stated above, the replacement of pgagroal's I/O layer will primarily involve creating a new event loop for pgagroal and may require a redesign of the existing code due to the design of these event APIs. This should occur in main.c, worker.c, pipeline.c, prometheus.c, the respective headers, and potentially other files. They all use libev structs and functions for I/O and interacting with an event loop. Therefore, their code will be redesigned to work with the new format.

Since it is intended for pgagroal to support both Linux and FreeBSD, it is clear that pgagroal could benefit from wrapping the same interface for each implementation and using this interface throughout the code. Nevertheless, maintaining the same interface for both Linux and FreeBSD seems unattainable given the differences in both APIs, despite their similarities (both have notification event data structures shared between kernel and userspace). The implementation, thus, would have to follow the recipe:

```
#ifdef __linux__
...
#elif defined(__FreeBSD__)
...
#endif
```

Therefore, what may happen is the creation of different function abstractions for each implementation and the use of such abstractions throughout the code to avoid cluttering the main code. These abstractions would be defined in the same files and used throughout the code. This is an initial idea, and due to the complexity of the API usage, this may not be attainable either.

However, details in their implementations can be provided for some of the required features, as explored below.

Main Loop

The main event loop is currently (mainly) done with:

```
struct ev_io;
void ev_io_init();
void ev_loop();
```

Linux implementation. The implementation is straightforward, with `io_uring_wait_cqe` wrapped around a loop and a call to the `cqe->user_data` that can potentially hold a struct containing a callback. Every process should have its own loop, and the loop could be created as soon as the process is forked.

FreeBSD implementation. The implementation should also be straightforward: create a kqueue, set up a monitoring event for readability on the specified file descriptors, and then enter an infinite loop. When an event is detected, it checks for errors and, if none occur, determines the type of event (readable or writable) before invoking the callback function associated with the watcher.

Periodic Event Handling

Issuing periodic callbacks at regular intervals is currently done with:

```
struct ev_periodic;
void ev_periodic_init();
void ev_periodic_start();
```

Linux implementation. This could be accomplished with timeout commands, with `io_uring_prep_timeout`, but this approach should be further evaluated. The upside here is to keep the implementation simple with io_uring. Since everything is going to be wrapped around a while loop and I/O will wait on cqes, timeouts can be abstracted to work as periodic tasks (isn't this all they are, anyways?).

FreeBSD implementation. FreeBSD's implementation will leverage kqueue's timer events to achieve similar periodic functionality. Setting up timer events in kqueue involves specifying EVFILT_TIMER filters in kevent calls, allowing pgagroal to execute periodic tasks effectively within the event loop.

Signal Watchers

Monitoring signals are done with:

```
struct ev_signal;
void ev_signal_init();
```

Linux implementation. This can be accomplished by using `signalfd` along with io_uring. This way, we can handle signal events in the same asynchronous event loop used for I/O operations.

FreeBSD implementation. kqueue can monitor signal events directly, without needing a separate file descriptor, by leveraging `EVFILT_SIGNAL` as the filter.

Fork Handling

The forking is handled by libev with:

```
void ev_fork_loop();
```

This function is called whenever a `fork()` call is made so that the event loop is duplicated. In both operating systems, the implementation is based on reinitializing the event loop and taking care not to interfere with the parent's event loop. Since the processes do not interact, no concurrency control is needed.

2.2. Phase 2 Details

With testing and profiling, I intend to establish a method to measure how the implementation of the event loop is evolving in comparison to previous pgagroal versions and to previous versions.

Tests could be conducted through testing frameworks in C or simply by evaluating the behavior with simulated Postgres client connections using shell scripts. A unit test framework can be selected from [9], for example, and this decision should be aligned with the community in a discussion on GitHub. The state of the implementation in Phase 1 should clarify what we should be testing and how we should approach these tests.

Profiling could be conducted using Linux tools such as strace, gprof, and perf. The methodology for this will also need to be aligned with the community in a discussion on GitHub.

The profiling and testing should provide insights for further optimizations and improvements in the implementation.

3. Timeline

Below, I set a timeline for 22 weeks, considering this is a hard and large project. I try to leave room for flexibility in the timeline to accommodate unexpected challenges.

Week	Date	Description
1 & 2	May 01 - May 14	Community bonding period: Set up a call to know each other, discuss pgagroal's history and future, and talk about the specifics of this project. Begin Phase 1 with a discussion of the first design of the code.
3 & 4	May 15 - May 28	Phase 1 for Linux: Start on the I/O Foundation by designing and implementing a simple io_uring loop for core I/O operations, marking the initial replacement of libev functionalities.

5 & 6	May 29 - June 11	Phase 1 for Linux: Continue developing the I/O Foundation, refining the io_uring integration for I/O other than event loop, try to identify potential room for more advanced io_uring techniques.
7 & 8	June 12 - June 25	Phase 1 for Linux: Finish the I/O Foundation by integrating advanced networking io_uring features.
	June 26 - July 09	Midterm evaluations and personal time. Planning to visit family during mid-year holidays, with limited availability but reachable via email.
9 & 10	July 10 - July 23	Phase 1 for FreeBSD: Begin working on the I/O foundation for FreeBSD, focusing on integrating and adapting to FreeBSD's system specifics.
11 & 12	July 24 - August 06	Phase 1 for FreeBSD: Continue developing the I/O foundation.
13 & 14	August 07 - August 20	Phase 1 for FreeBSD: Finish the I/O Foundation, ensuring compatibility and efficiency.
15 & 16	August 21 - September 03	Phase 2: Design and implement behavior tests and stress tests for the event loop replaced code to ensure functionality and stability.
17 & 18	September 04 - September 17	Phase 2: Fix potential issues with the code found in previous weeks and expand tests if necessary.
19 & 20	September 18 - October 01	Phase 2: Design and implement a strategy to profile code. Identify code bottlenecks and define a strategy to optimize code based on findings.
21, 22, & 23	October 02 - October 22	Phase 2: Implement optimization strategies identified in the previous phase. Final testing of all implementations for both Linux and FreeBSD. Make sure everything is perfect.
24 & 25	October 23 - November 04	Finalize documentation, prepare a comprehensive report summarizing the development process, challenges faced, solutions implemented, and future work directions. Ensure all code is well-commented, and the repository is organized for easy navigation. Final preparations for project submission.
	November 04	Deadline to submit final work product and final evaluation.

4. References

[1] https://www.youtube.com/watch?v=-5T4Cjw46ys
[2] https://unixism.net/
[3] https://unixism.net/loti/tutorial/webserver_liburing.html
[4] https://developers.redhat.com/articles/2023/04/12/why-you-should-use-iouring-network-io
[7] https://people.freebsd.org/~jlemon/papers/kqueue.pdf
[5] https://github.com/axboe/liburing/wiki/io_uring-and-networking-in-2023
[6] https://lwn.net/Articles/908268/

[8] https://man.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2
[9] https://stackoverflow.com/questions/65820/unit-testing-c-code