[PostgreSQL] pgagroal: Replace the 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 worked on many different things, from algorithms and data structures, to systems engineering, to machine learning theory and applications, computer vision, computer graphics and application development and benchmarking. However, the things that are closest to 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 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 goal with this project is to learn from the community and assist pgagroal in achieving unparalleled performance among connection poolers.
Contributions: #408, #411, #427, #431

1. Synopsis

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

The motivation behind this project is the fact that libev is no longer being maintained. 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 incoming 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 the setup of io_uring. A successful Linux implementation of an efficient event loop for pgagroal necessarily utilizes io_uring -- as well as other Linux I/O interfaces (e.g. stdio) -- for efficient I/O.

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) Testing and Continuous Refactoring (Phase 2).

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

The existing code will definitely have to be changed to comply with the new interface. However, the interface will aim to limit the need for heavily redesigning functions and changing behavior of the existing code, modifying behavior only when absolutely necessary.

The result of this first phase would be a maintainable and small footprint event loop that suffices for pgagroal specific uses.

For *Phase 2*, I propose the creation of functional tests, for testing functional requirements of the connection pool, and performance tests, for measuring resource utilization for pgagroal under different workloads.

Functional tests would measure correctness of the new event loop implementation, while the performance tests would guarantee the responsiveness and stability of the new implementation.

Since the intention of this phase is to improve an experimental implementation, the tests would help achieve the desired functionality and would ensure that refactoring efforts would not introduce performance regressions.

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. There is a lot of room to further optimize the code by using io_uring code and its new features. The right usage of these features can be evaluated with `strace` to measure the number of system calls that are being spared. [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 other 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.

2.1.3. Implementation Details

As stated above, the replacement of pgagroal's I/O layer will *primarily* involve creating a new event loop interface for pgagroal, which will require changes in structs and functions to work with the newly created interface.

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.

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 of the interface and potentially the main code, thus, would have to deal with the difference in systems, following the recipe:

```
#ifdef HAVE_LINUX
...
#elif HAVE_FREEBSD
...
#endif
```

Details in the different implementations are 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: create `io_uring`, set up events for monitoring and wait on these events (`io_uring_wait_cqe`) inside an infinite loop. When the execution is returned, determine the type of event and access the event user data (`cqe->user_data`) that can hold a struct containing a callback.

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, determine the type of event and invoke the callback function that is held by user data (`udata`) inside `kevent`.

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 `cqe`s, timeouts can be abstracted to work as periodic tasks.

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, I intend to establish rigorous methods for measuring whether the implementation of the new I/O layer is correct and how it is evolving during the refactoring phase.

I believe that the state of the experimental implementation in Phase 1 should clarify how we should test for correctness. I also believe, however, the performance tests are agnostic to the first implementation and could be designed even before Phase 1.

For consistency in the approach, I propose defining tests in a posterior phase, after the Phase 1 implementation.

Also, I propose integrating these tests into the CI pipeline.

2.2.1. Functional Tests

I propose the creation of tests using testing frameworks in C. 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.

Testing the application should be followed by a coding period to fix any identified issues or rollback to the previous application state.

The functional tests should be implemented right after the implementation of Phase 1 in order to ensure correctness and should be run after every refactoring after the first implementation to ensure that no regression in functionality appears.

2.2.2. Performance Tests

Measuring resource utilization will enable the identification of bottlenecks in the code and areas where the new implementation can benefit from optimizations.

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

Performance tests can be conducted using `pgbench` and other Linux tools such as `gprof` and `perf`.

`pgbench` will be useful for comparing full implementations performance in the real world. Using this tool will allow us to understand in what scenarios (different settings) our implementation is failing, and, afterwards, use other tools to target what is making the code slow.

`gprof` may be the most useful after having a working version of the code, but before fully optimizing it. This tool is valuable for identifying how specific parts of the code are affecting performance and will help improve the internal logic.

`perf` is good for system level analysis, so it could be most useful at later stages of development. This tool will help fine-tuning, uncovering issues such as excessive system calls, or cache inefficiencies.

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., investigating other features of the queues interfaces, reducing system calls, improving cache performance, vectorization).

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 by allowing some "intense" weeks to be followed by "mild" weeks.

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. Refine the io_uring integration for I/O other than event loop, try to identify potential room for more advanced io_uring techniques.
5 & 6	May 29 - June 11	**Phase 1 for Linux:** Finish the I/O Foundation by integrating advanced networking io_uring features.
7 & 8	June 12 - June 25	**Phase 2 for Linux:** Design and implement functional tests to ensure functionality and stability. Fix potential issues with the code found during tests. Integrate tests into CI pipeline.
	June 26 - July 09	Midterm evaluations and planning personal time to visit family during mid-year holidays. I will have limited availability but I will be 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:** Finish the I/O Foundation. Search and experiment with advanced networking patterns and usage for kqueue that can benefit the implementation.
13 & 14	August 07 - August 20	**Phase 2 for FreeBSD:** Design and implement functional tests to ensure functionality and stability. Fix potential issues with the code found during tests. Integrate tests into CI pipeline.
15 & 16	August 21 - September 03	**Phase 2 for Linux & FreeBSD:** Design and implement a strategy for performance tests. Integrate tests into CI pipeline.
17 & 18	September 04 - September 17	**Phase 2 for Linux:** Identify code bottlenecks and define a strategy to optimize code based on findings and implement them.

19 & 20	September 18 - October 01	**Phase 2 for FreeBSD:** Identify code bottlenecks and define a strategy to optimize code based on findings and implement them.
21, 22, & 23	October 02 - October 22	**Phase 2 for Linux & FreeBSD:** Evaluate if expanding tests are necessary and, if so, implement more tests. 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