Reprinted from the

Proceedings of the Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*

An AIO Implementation and its Behaviour

Benjamin C. R. LaHaise
Red Hat, Inc.
bcrl@redhat.com

Abstract

Many existing userland network daemons suffer from a performance curve that severely degrades under overload conditions to the point of collapse. The design of AIO was such that it should maintain a steady response rate when faced with a multitude of outstanding connections. With AIO for Linux becoming ready for more widespread use, the real world performance characteristics of the design and its shortcomings needs to be examined. What follows is an attempt to characterise the behaviour of async poll and read under various conditions, contrasting with poll(), and /dev/epoll.

1 Introduction

With the maturing of the Linux kernel, the scalability of various subsystems is becoming a greater concern for vendors in the quest for enterprise adoption. The advent of TUX [TUX] has shown that very high thruput content delivery systems can be built on top of the kernel's internal infrastructure, yet implementing these servers in userspace frequently exacts a large hit in performance.

Traditional IO models based on non blocking IO using select() or poll() have serious shortcomings when faced with loads that include many idle connections (HTTP, FTP, LDAP servers), or attempt to extract increased parallelism from slow subsystems (filesystem and disk IO). For example, the poll() syscall is at best O(n), where n is the number of file descriptors on which events are being monitored. Issuing parallel disk IO requests requires the use of threads, which have their own overhead, in addition to adding a significant amount of debugging effort to the application.

The Asynchronous IO implementation presented attempts to address these concerns twofold: by providing asynchronous operations that can proceed concurrently with the application, as well as utilising an event based completion mechanism to return the results of those operations in an efficient manner.

Previous work under Linux in this area includes the addition of SIGIO [SigIO] based readiness notification, reductions in the overhead of the poll() interface through the creation of /dev/poll [devpoll], and further optimizations with the event based /dev/epoll [EPoll]. The AIO implementation presented here should have similar performance characteristics to the event interface that /dev/epoll uses, as both models have a 1-1 correlation between events being generated and the potential for progress to be made.

One area where AIO poll differs significantly from /dev/epoll stems from readiness vs ready

state notification: an async poll is like poll in that the operation completes when the descriptor has one of the specified events pending. However, /dev/epoll only generates an event when the state of the monitored events changes. Further differences emerge once the async read and write operations are introduced.

2 AIO Design and API

The basic design of AIO for Linux is based on the POSIX AIO [PosixAIO] specification and NT's completion port mechanism [Russinovich]. Primary design goals included:

1. lightweight completion events
2. usable for libraries as well as applications
3. support for general disk and network IO
4. scalable for servers handling many connections
5. the desire to eventually suppport zero copy io (O_DIRECT disk io, and hardware checksum assist for TCP transmit)
6. a 64 bit kernel should be able to process structures from both 32 and 64 bit processes with minimal additional code

POSIX AIO fails to meet several of these criteria, in part due to its reliance on signals, as well as the nature of its io wait mechanism (aio_suspend is O(n) where n is the number of outstanding ios).

The core of this implementation centers around the io context which specifies a given completion queue. io contexts are created by io_queue_init and destroyed via io_queue_release. New ios are submitted via io_submit (which is similar to lio_listio), but can only be queued if there is sufficient space in the completion queue to receive any resulting io_event.

Events are read by means of io_getevents. One of the features of the design is that io_getevents can be implemented as a vsyscall, which reduces the overhead of receiving completion events under load.

3 Testing Methodoloy

The goals of testing are to highlight the strengths and shortcomings of the various IO models. To this end, the areas of interest examined include thruput under varying numbers of open file descriptors, thruput with differing message sizes, and the effects of increasing the parallelism in request processing.

To demonstrate the scaling issues involved when dealing with many file descriptors, a simple test application [PipeTest] was developed. Pipetest attempts to measure the number of token passes per second that a given io model can obtain under a set of conditions. The number of idle file descriptors, parallel tokens passes and message size are all parameters. In operation, pipetest opens a specified number of pipes, begins transmitting one or more seed tokens, then proceeds to receive and transmit the tokens for a number of repetitions.

Initial plans were to use TCP network connections between a set of clients and a server, but due to code maturity and other issues, the decision to use a pipe based test was made. Thankfully, use of the pipe mechanism eliminates several potential bottlenecks (including IO bandwidth and driver performance), and re-

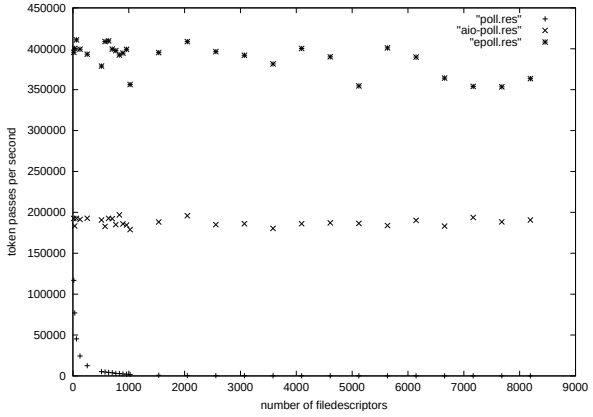

Figure 1: baseline performance vs number of file descriptors

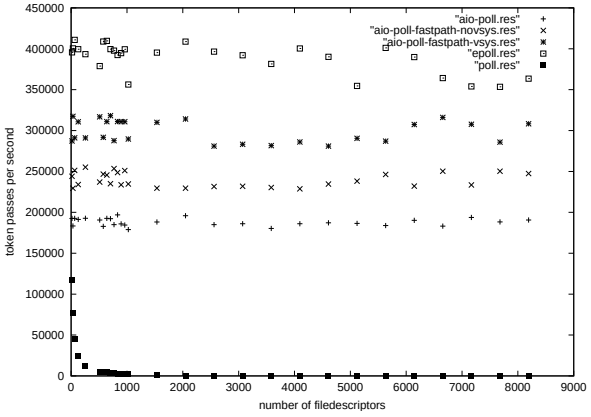

Figure 2: addition of poll fastpath and vsyscall mechanism to aio

stricts measurements to the actual overhead of the code under test. For all test runs, pipetest was able to run at 100% CPU usage.

For the sake of simplicity, and to avoid SMP scaling issues, all tests were run on the 2.4.19pre5 kernel with /dev/epoll and AIO patched in.

4 File Descriptors vs Thruput

It is well known that one of the crippling factors for heavily used server processes comes from the number of active client connections being serviced. To demonstrate this factor on scaling, tests were run where the number of open pipes increased while other factors were held constant. Using Pipetest, the number of token passes per second was measured as a function of the file descriptor count and IO model.

The baseline performance as shown in Figure 1 contains several striking features, most notably that the existing poll() model exhibits a rapid decay as the number of active file descriptors increases. This stems from poll()'s O(n) workload in searching for active file descriptors. Async poll remains flat as the number of fds increases, as does epoll. The overhead of epoll appears to be about half of async poll, which points to a few shortcomings in the allocation and initialization of the async poll structures.

5 A few optimizations

In an attempt to reduce the overhead present in async poll relative to epoll, a fastpath that does not allocate any control data structures leads was created. In Figure 2 we can see this leads to an approximately 20% improvement in thruput. The addition of the io_getevents vsyscall lead to 20% increase in performance, bringing async poll to roughly three quarters of epoll thruput.

6 Async read

Figure 3 compares async read to epoll and async poll thruput. It should be noted that

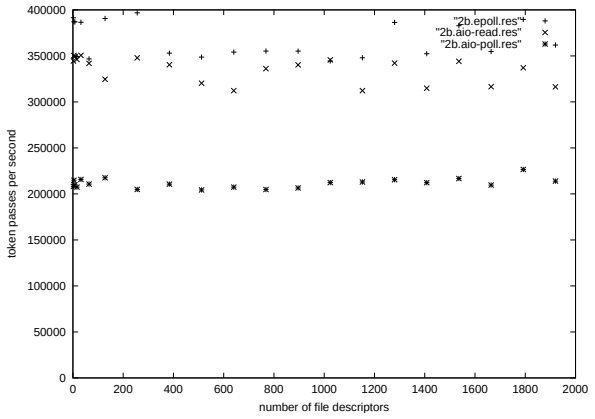

Figure 3: async read implemented

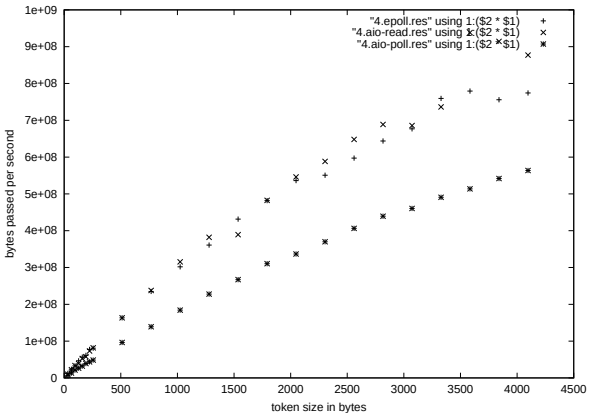

Figure 4: thruput as message size increases

async read overhead includes walking the page tables to find the underlying kernel pages for the user virtual address. Since the token write() is performed after the async read is posted, async read benefits from a single copy of the data. This brings async read throughput to within 15of epoll in the worst case. Cache effects are much more apparent for async read and epoll, probably owing to the differences in physical page colours between runs. As expected, async read also maintains a flat response when the number of open file descriptors increases.

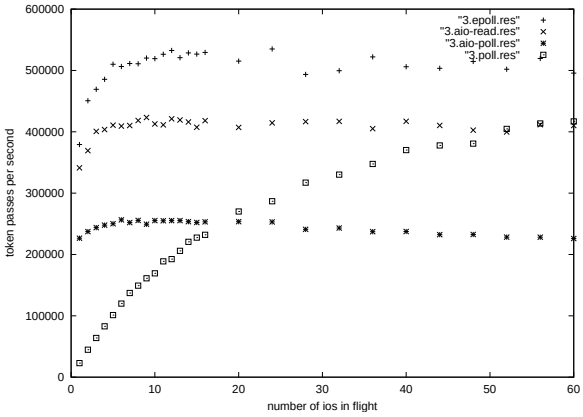

Figure 5: thruput while increasing in flight ios

7 Message Sizes

While epoll/read is faster than async reads for small message sizes, async read should become more efficient as the size increases and the benefits of the single copy begin to outweigh the static setup costs.

In Figure 4 it is apparent that for message sizes of 256 bytes or less, the async read overhead outweighs the effects of single copy. However, for message sizes of 512 bytes and greater, async read is able to exceed epoll thruput, but only by a small margin.

8 Multiple IOs in flight

The tests presented thus far only deal with one in flight IO through each iteration of the event loop. In real world servers, the number of IOs in flight will tend to increase with the number of active clients.

Figure 5 shows the results of a run with the number of IOs through to 60, 128 file descriptors and a 12 byte message size. The event

driven models show a small increase in thruput up to 122 IOs, then remain fairly flat. Poll is included to show that it takes at least 15% of the file descriptors to have IOs in flight to match async poll performance, and almost half to match async reads.

9 Conclusions

The first generation of AIO for Linux shows that the event driven model is able to provide significantly improved performance in cases where poll() degrades severely. Comparisons with /dev/epoll show that further work needs to be done to mitigate the cost of mapping userspace memory into kernel structures for small message sizes, but that the overhead is outweighed by the support of single and zero copy as the amount of data transferred increases.

10 Future work and directions

Much work remains to complete the implementation, as many existing parts of the kernel were not designed with asynchronous operation in mind. The first year of development has yielded significant insight into the benefits and drawbacks of various in kernel techniques for the AIO implementation.

For example: during the recent addition of a cancellation API for iocbs, the limitations of using tqueues as the basis for the worktodo helpers became apparent. It turns out that tqueues cannot be cancelled safely in an SMP environment. One option is to make use of tasklets, but that path is hindered by the fact that many internal APIs cannot be called from bottom half context.

References

[TUX] *TUX Web Server Manuals*, Red Hat, Inc. `http://www.redhat.com/docs/manuals/tux/`, (2002).

[RTSig] *Analyzing the Overload Behavior of a Simple Web Server* N. Provos, C. Lever, S. Tweedie, `http://www.usenix.org/publications/library/proceedings/als2000/full_papers/provos/provos_html/index.html`, (ALS 2000).

[EPoll] *Improving (network) I/O performance*, Davide Libenzi `http://www.xmailserver.org/linux-patches/nio-improve.html`, (2001).

[Russinovich] *Inside I/O Completion Ports*, Mark Russinovich `http://www.sysinternals.com/ntw2k/info/comport.shtml`, (1998).

[SigIO] To Be Added.

[devpoll] To Be Added.

[PosixAIO] To Be Added.

[PipeTest] To Be Added.