Critique of

“

‘I/O-Lite: A Unified I/O Buffering and Caching System’

by

Vivek S. Pai, Peter Druschel and Willy Zwaenepoel

”

by anonymous

12/11/1999

This paper describes an integrated I/O system intended to replace that used by current-day general purpose operating systems. It focuses sharply on the practicality of real-world performance of the system, particularly when used for web serving.

It is a well-written paper about a well-designed system, which is not afraid to make sensible and informed trade-offs between simplicity and performance where necessary. The paper tends towards performance in these decisions.

The I/O-Lite system involves two main data structures. The first of these is an IO-Lite buffer, which is an integer number of contiguous virtual pages. These have the important property that they are immutable until discarded, and are only discarded when they are no longer referenced.

The second data structure is an IO-Lite aggregate. These are a sequential list of ranges or slices of IO-Lite buffers. They are high-level representations of a logical flow of data, formed from slices of existing buffers. Concatenating the slices together would form the expanded (or flattened) data flow. For example, a file with a header appended would consist of two slices — one for the header, and one for the file itself. Instead of copying data through pipes or IPC, only aggregates are transferred from one protection domain to another, which only reference the actual data.

The basis of IO-Lite is a caching mechanism which tracks and controls buffers. There are also operations such as appending, prepending, inserting which can be performed on aggregates without unnecessary copying.

The remainder of this critique is split into two sections: one covering the generally good design and implementation decisions which are presented in the paper, and another suggesting possible improvements or clarifications. Unlike the other two papers however, I found very little to disagree with in this paper.

Decisions

This design was arrived at by investigating the typical causes of I/O costs. The paper identifies three of these:

• Redundant data copying. Data is unnecessarily copied both within system code (e.g. from file system to network system) and between the system and the user code (e.g. fread). The lesson here is ‘share’.

• Multiple cached copies of data. With different parts of the operating system maintaining their own caches, much data is cached more than once, thus leading to a

reduction in the total amount of memory available for cache use, and accordingly decreasing cache hit rates. The lesson here is ‘simplify’.

• Lack of single-system optimisations. With a single caching system unexpected optimisations can be made, such as caching IP checksums on previously seen data in outgoing network packets. The lesson here is ‘unify’.

Laziness is rife throughout the design of IO-Lite. Even though every reference to a buffer is counted, it is only decommissioned when a new one is required. Buffers are grouped into pools, with each pool belonging to a fixed list of protection domains. This list is fixed so that address space mappings can be recycled (thus eliminating flushing problems) when a buffer is recycled.

There are two main operations used by clients to deal with I/O under IO-Lite — these are IOL_read and IOL_write. They both work with traditional unix file descriptors opened in the normal way, but instead of working with application buffers, they work with IO-Lite aggregates. These operations are atomic for each file/stream, and once the data is read it cannot be revoked without explicit permission — the buffers are persistent (of course it may be written to a backing store if physical memory runs low)

In section 2.7, the authors admit that cache management decisions are a complex area, especially given the latitude afforded by IO-Lite’s design — thus only a simple system is employed. This recognition that the area is beyond the scope of the paper is rare, and the absence of unfounded assertions is appreciated.

In section 2.8, an excellent explanation for the decision and the results of making buffers immutable is given. Three general cases (complete overwrite, block reorganisation, fine-grained reorganisation) of buffer changes are mentioned, and suggestions for action (IOL replaces buffer, IOL uses aggregates, use own buffer) and analysis of cost (minimal, better than usual, minimal w/out aggregates) are given for each.

The implementation of a web server using IO-Lite and the efficiencies which can be gained is discussed in section 2.10 — the exciting point here is the use of aggregates in a web-server CGI module. These modules often replace certain key parts in a HTML template with specialised data (maybe a web counter), and the fact that the static slices of the template are reused even to the level of not recalculating an IP checksum on repeated slices is quite impressive.

The results section is well-formatted and impressive for the tests conducted. After the intense focus on one particular application — web serving, in which the system using IO-Lite performs very well indeed — the authors go on to perform more general tests with other software, a move which greatly improves the credibility of their work.

The tests are well-thought-out and very real-world oriented (essential for a real-world system such as IO-Lite), and there is a good description of the methodology used, as well as an explanation of the results and explanations. The different web serving tests show that the authors expectations are realised.

This brings up the applicability of the system, which is unquestionable. The results section shows that it was relatively easy to modify all sorts of existing software to use IO-Lite — and not only the web server and CGI module, but the operating system itself (OpenBSD) and a number of other programs. And it shows that there is usually a

significant performance gain if this is done.

Indecisions

This section discusses the few things that are not pursued in the paper as well as I would have liked. It is noticeably shorter than sections of this kind in the critiques of the other papers.

The first few paragraphs in this section were going to be dedicated to absence of a section on related work, but an excellent section is found at the end. There is nothing particularly wrong with putting it at this end of the document, as it allows easier comparison to the work described in it. However, certain terms such as fbufs would have benefited from an earlier description, and placing the section at the start would better set the context for less experienced readers such as myself.

The reading of this section brings up the benefits that loss of transparency (user must explicitly invoke IO-Lite for full effect) and concurrent sharing (immutable buffers allow easier multiple access) have. In line with the decisions throughout the rest of this paper, the benefits are large in performance terms, and the inconveniences are small in complexity terms.

One of the issues worthy of mention here is the problem of granularity of access — if a process has access to any part of a buffer, then it has access to the whole buffer (not just the references pages), even if the aggregate includes only a small slice of it. I can imagine a number of situations where this would be undesirable, if the buffer stores data on a number of users, and a server wants to disclose the record for only one user.

Another issue arises from the decision that the entire ACL for a buffer must be decided before it is allocated. The paper indicates that this was required to allow recycling of page-table (and TLB) mappings and lazy remapping of blocks. This means that the final destination and all the waypoints of data must be known before it is brought into a buffer. This seems to remove some of the sharing possibilities implied by IO-Lite aggregates.

It is claimed that the incoming network packet handlers do not suffer from the early demultiplexing requirements this restriction entails — but if the demultiplexing information is kept in the packet when it is passed on, there will some unnecessary temporary copying of data before it is put into an IO-Lite buffer, just in order to read that demuxing info.

Also, more general uses of the system are not explained, such as what occurs if more than one protection domain opens the same file — surely the buffers read by one PD can be used by the other? This is almost certainly in fact possible, just not explained in this paper, because there are many instances when the destination of data will change (possibly quite different to these examples), and some reasonable memory remapping cost could be endured.

A possibility which is overlooked is the automatic flattening of aggregates — that is, if an aggregate becomes too fragmented, the IO-Lite system could ‘flatten’ it or part of it into a new buffer. This could help the automatic use of IO-Lite by IO-intensive applications which perform fine-grained operations on data, such as list reordering (sort, reverse) and graphic rearrangement algorithms.

Conclusion

Overall this is an excellent paper, which describes a high-performing and immediately useful I/O system. There remains however one feature that is claimed but not explored far enough — that is of the general use of this system for I/O. It clearly performs will in the examples of file systems and network systems, but there is a large class of systems which have been overlooked — databases!

Fortunately, the performance of this system with database applications looks to be quite good, especially since users can tailor their own cache control. Since databases generally access data in fixed-length records (even relational databases), and often combine them, the aggregate structure looks perfect to implement intelligent and fast database servers, without great programming difficulty.

Not being an expert in database engine design however, I suspect that there are a number of instances where this would not be appropriate (such as a sort). However, the IO-Lite implementation here provides the flexibility that applications are **not** required to use its aggregates, so database authors can in fact get greater control over OS memory management with this system, which is of benefit to all.

In conclusion, a very worthy system based on natural foundations and producing outstanding performance.