Summary
The paper proposes modifications to existing mainstream monolithic kernel operating systems such as Linux to allow user level application code to be pushed down to the kernel and be executed in privileged kernel space. This is done in order to speed up IO, particularly when interfacing with high speed network or storage devices. The aim is to reduce the amount of data copied from kernel to user space, and to reduce the number of context switches required during the execution of IO bound user applications. The implementation seems to be based on Berkeley Packet Filter (BPF) with the aim to extend it to IO devices other than network devices and provide extra functionality within the filter.

The proposed ExtOS would modify filesystem operations such as read and write to execute functions passed by the user as a filter within the kernel. This allows unneeded data to be discarded within the kernel rather than copied to user space before being discarded. To achieve this, ExtOS provides a modified Linux IO API in which system call functions take extra parameters to specify a function for the kernel to execute on an IO operation and any arguments it needs. This function is then called during the IO operation to filter the content before copying the residual content to userspace.

The proposed ExtOS would also allow for optimisations based on the kernel's knowledge of other running applications. The authors claim that by having access to the user application's code, the kernel will be able to streamline multiple applications' filters and make better scheduling decisions. However, the kernel in a monolithic OS already has access to the user address space if it needs, and this paper offers no insight into how this would be implemented, so a claim that it would allow better scheduling is dubious. It is also not specified how the kernel will analyse the filter functions of different user level applications in order to create a composite filter and thus improve performance.

In order to address the significant security risks of executing user level code within the kernel, the authors propose a series of unimplemented solutions. The primary proposed defence is that the code would be written and compiled with a reduced instruction set to prevent the application code from executing privileged instructions within the kernel. It is unclear here whether the authors intend to use a static checker and Just-In-Time (JIT) compilation to ensure the safety of arbitrary user code executing in the kernel, or to simply formally verify the code and solve the halting problem all at runtime. It appears that they will extend the JIT compiler to use Software Fault Isolation when there is hardware support for it, though it is unclear whether this has been implemented.

The authors also intend to create tools to simplify the creation and management of the code to be pushed into the kernel. However, the paper appears to leave this for future work.

Pros
- The authors provided a decent overview of some of the existing work

Cons
- The paper is riddled with buzz words that either add nothing to or actively detract from the paper
- Terms and abbreviations are introduced but either defined long after use or not at all: BPF/eBPF, DPDK/SPDK, SFI
- Use of unspecific power-of-two abbreviations: GB
- Continued incorrect reference to BSD instead of BPF
- There is a complete disregard for system security

- Statements that are unrelated to what is being discussed, or otherwise make no sense, making it difficult to understand what is actually being proposed
- Badly labelled graphs and tables

Critical Review

The paper attempts to address the issue of high speed IO devices being bottle-necked by either the CPU itself, or by the overhead of the OS. It particularly focuses on solving this issue within the constraints of mainstream monolithic OSs such as Linux. The paper seems to provide an effective overview of the related work in the field, mentioning both old and more recent approaches to the issues of super fast IO and their merits and shortcomings. But despite mentioning similar work and viable alternative solutions to the problem, the author discounts such solutions and proposes their own. It is hard to look past the fact that microkernels such as seL4 could provide most if not all of the (reasonable) functionality proposed in the article without the security vulnerabilities. Indeed the author even writes that “Microkernels are extensible OSes by design”. The author also notes PipesFS and Streamline which, by the writer’s own admission, aim to solve the same problems as ExtOS, but discards them as they focus primarily on network IO. Two potential solutions already exist, with one providing a principled and secure way to run user applications with low overhead, and the other only needing its principles applied to SSDs rather than network IO.

Despite the consistent use of meaningless buzz words, the author at least makes an effective argument for thinking about more efficient ways of handling super fast IO data. Low latency network cards already perform some of the processing of packets in order to take the load off the CPU, and it seems plausible that there could be a need to stream data to or from SSDs at speeds that would be impacted by OS overhead.

However, the solutions presented in the paper are not technically sound. The paper consistently proposes features and functionalities without providing details on how such features would be implemented. Examples include proposing improved scheduling via the ability to view user code from within the kernel, without addressing the fact that this already possible or how it would actually be done; proposing the ability to create composite filters comprised of different user level filter functions passed to the kernel without any indication of how that would be done in a way that would improve performance; and, perhaps most egregiously, proposing runtime formal verification of user code to ensure security within the kernel with no proposition as to how. The only concrete implementation described was a modification to the read system call which calls a user defined filter function on the output file stream.

The paper treats security within the kernel as an afterthought and provides far too little information on how incorrect or malicious user level code would be prevented from exploiting the kernel. It suggests using similar static code analysis and Just-In-Time (JIT) compiler as Berkeley Packet Filter, with a hand-wavy reference to formal verification and ensuring termination at run time – again without even considering how that would be done. It further mentions extending the JIT compiler to make use of software fault isolation by exploiting hardware support when available, which seems to suggest such security features are only sometimes present. The lack of an exhaustive, coherent and principled approach to security is a huge red flag and effectively prevents ExtOS from being a viable solution for anything but a system without a need for an OS, which of course defeats its purpose.

Further, the initial results detailed in the paper are not running on anything even close to the features of the proposed ExtOS and many of the results relied upon are not presented in the paper. The prototype ExtOS implementation used for the measuring the performance of the system consists of a single modified read function and what appears to be unchecked user code executed directly in the kernel. Upon this foundation, they claim that pushing user code into the kernel can result in an up to 80% improvement in performance based on trends observed but not provided in the paper. They also appear to claim that benchmarking the difference between using read and write system calls to copy a file compared to using the sendfile system call and observing a 40% speed improvement somehow assists in the case for ExtOS. Neither the system on which the benchmarks were run, nor the reported results are reliable representations of the performance of an ExtOS like system.

The graphs and tables within the paper are also difficult to interpret. Figure 2 omits the label on the x axis, and the data presented is overly noisy and its importance unclear. Similarly, it is unclear in Table 1 what the speedup is relative to and there is no information on what the variable buffer size is. That said, the diagram in Figure 1 is a good visualisation of the spectrum of approaches to implementations of monolithic operating systems.

Overall, this paper reads like a thought experiment rather than a research paper. There is a small amount of implementation, but nowhere near enough to assess ExtOS in any meaningful way. Security implications are not seriously considered despite the significant dangers of executing user code in the kernel. It also fails to recognise existing work in the field which could provide the results sought. For these reasons, I do not believe the paper is sufficiently new, significant, or developed to justify publication.