ExtOS: Data-centric Extensible OS

Antonio Barbalace[*]
Stevens Institute of Technology
antonio.barbalace@stevens.edu

Javier Picorel
Huawei Technologies
javier.picorel@huawei.com

Pramod Bhatotia[†]
The University of Edinburgh
pramod.bhatotia@ed.ac.uk

ABSTRACT

Today's computer architectures are fundamentally different than a decade ago: IO devices and interfaces can sustain much higher data rates than the compute capacity of a single threaded CPU. To meet the computational requirements of modern applications, the operating system (OS) requires lean and optimized software running on CPUs for applications to fully exploit the IO resources. Despite the changes in hardware, today's traditional system software unfortunately uses the same assumptions of a decade ago—the IO is slow, and the CPU is fast.

This paper makes a case for the data-centric extensible OS, which enables full exploitation of emerging high-performance IO hardware. Based on the idea of minimizing data movements in software, a top-to-bottom lean and optimized architecture is proposed, which allows applications to customize the OS kernel's IO subsystems with application-provided code. This enables sharing and high-performance IO among applications—initial microbenchmarks on a Linux prototype where we used eBPF to specialize the Linux kernel show performance improvements of up to 1.8× for database primitives and 4.8× for UNIX utility tools.

ACM Reference Format:
Antonio Barbalace, Javier Picorel, and Pramod Bhatotia. 2019. ExtOS: Data-centric Extensible OS. In *10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '19), August 19–20, 2019, Hangzhou, China.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3343737.3343742

1 INTRODUCTION

Computer architectures faced a fundamental shift in the last decade due to the introduction of blazing fast IO devices, buses, and interconnects, and the stagnation of the single thread speed of central processing units (CPUs) [68]. With today's technology it is difficult, if not impossible, for a single thread CPU to keep up with the data generated by a network interface card (NIC), up to 200Gb/s [48], or by a modern solid-state storage device (SSD), up to 7GB/s in read [60], and microsecond access latencies [8, 33].

With low-latency high bandwidth IO any unnecessary software operation executed by the CPU may slow down application's IO processing. Therefore, a top to bottom lean and optimized software is necessary to fully exploit new IO hardware. However, despite these hardware changes, the main software interface with the hardware, the system software, that is OS, runtime, and compiler, unfortunately uses the same design and interfaces of a decade ago.

As a consequence, in order to (partially) benefit from faster IO devices, programmers circumvent the OS—that is, moving device drivers and eventual OS services from kernel to user-space, e.g., kernel bypass [53, 65], DPDK/SPDK [36, 56]. Circumventing the OS works well to build appliances, i.e., a computer, or part of it, specialized for a single application. However, in such user-space-based IO approaches, the IO data used by an application cannot be easily and efficiently shared with potential other applications running on the same computer [37, 68]. Thus, all data-center, scientific, office, and mobile applications that rely on classic OSes and systems software interfaces, as well as on sharing devices, cannot fully benefit from the new IO hardware.

This paper attempts to re-design system software to wholly exploit new low-latency and high-bandwidth IO devices. The proposed design is inspired by near data processing (NDP) [5], and operator pushdown in databases [24, 32, 66]. The work extends such principles to generic software suggesting that *data should be moved to upper software layers only if it will be used to do any computation; if the computation is trivial, or the data is going to be used as-is, it shouldn't be moved to upper software layers —eventual computations should be run in place.*

In virtue of that, *the key idea beyond the proposed design is to enable applications to extend the OS's file IO operations (e.g., read and write, send and receive) with application defined functions*, such that frequent *simple applications' operations on IO data*, e.g., discarding or copying data, happen within the OS with minimal application involvement. Differently from circumventing solutions, data from IO devices can still be easily

[*]Part of this work was performed at Huawei German Research Center.
[†]Part of this work was performed at Huawei Dresden Research Center.

APSys '19, August 19–20, 2019, Hangzhou, China
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6893-3/19/08…$15.00
https://doi.org/10.1145/3343737.3343742

and efficiently shared among all applications running on the OS. This idea is embraced in the *data-centric extensible OS* (ExtOS) that comprises of a new OS kernel, as well as supporting compiler and runtime libraries, which let applications to push down parts of their executable code into the kernel and run them as part of the kernel IO subsystem itself. This enables a set of performance improvements by reducing user/kernel data copies and context switches, merging of the IO operations, and transparently map the push down code to the available processors, possibly including accelerators and emerging near data processing (NDP) devices [4, 5, 31, 49, 58, 70].

Extending the OS kernel at runtime with application code, which is untrusted because provided by a 3rd party, may impact the security of the entire system. Thus, ExtOS identifies methods for executing application's code in the kernel without impacting its stability and security. Note that this utterly different from OS extensibility by trusted code—kernel modules (Linux) or extensions (Darwin), which is common within classic OSes to selectively load at runtime for example device drivers, which are provided with the kernel.

The data-centric extensible OS aims to be practical and usable, not just a research prototype. Hence, ExtOS targets UNIX-like OSes. A Linux-based prototype that capitalizes on the Linux's BPF/eBPF machinery [26, 30, 39] has been implemented for an initial validation. Preliminary evaluation shows promising results—up to 1.8× performance improvements in database primitives and 4.8× speedups of UNIX utility tools. An important challenge is to ensure safety properties of the untrusted application code in the kernel. Specifically, we aim to ensure memory safety [51] and termination [19] properties to enable application-specific specialization of the kernel. We are investigating runtime verification approaches to prove these safety properties of the untrusted application code in ExtOS.

2 BACKGROUND AND MOTIVATION

The gap in OS research. Classic OS kernels, such as Linux and BSD are based on the monolithic OS design ("classic monolithic" in Figure 1). Despite many OS designs were proposed by the academic and industry research [2, 6, 7, 9, 13, 25, 34, 43, 44, 57, 61, 63, 64], the monolithic design is still the OS design of choice because it is a point of compromise between performance, security, sharing, genericity, and manageability. Additionally, it has been proven to be flexible enough to be re-purposed in other OS designs, design space in Figure 1.

Latency critical applications in order to mitigate overheads due to context switches and costly paths in kernel-space (for genericity), introduced the "bypass kernel approach" where a device driver is implemented in the application itself [68]. This model has been extended to the "libOS approach" implemented by DPDK/SPDK [36, 56], which moves entire kernel subsystems within the application. Note that this approach is inspired by libOS/exokernels [25] and has been improved within monolithic OSes by Dune [11] and IX [12], while within microkernels by Arrakis [52]. The right side of Figure 1 shows the extreme scenario when continuing in this direction, i.e., a single application integrating all the OS components that are strictly required for its execution.

Instead, the left side of Figure 1 targets solutions that improve performance and enable sharing of the hardware (and software) resources among different applications. In this space, the most notable contribution is the BPF/eBPF work [46], which implements the "filtering and policy approach". Between that and the extreme scenario *the research space is empty*, if not for the pioneering work in VINO [63] and Cosy [57], which have been proposed and implemented decades ago on radically different computer architectures, thus on different core assumptions, and their source code is not available. Finally, the extreme scenario is represented by a single address space OS [35] that offers a generic kernel to multiple applications without memory protection for isolation, but compiler methods.

The data-centric extensible OS aims at filling the gap in OS research by exploring the "Extensible Monolithic Kernel" design, via revisiting the pioneering work in the area on modern hardware with low-latency high-throughput IO, and adapting the fundamental principles of OS extension via application code among all IO resources, securely.

Why now? Today's IO devices are dramatically different from decades ago [8, 68],when the concept of extensible OS was originally introduced. Specifically, as discussed in the introduction, IO has higher bandwidths and lower latencies than before, and those are going to grow further based on the recent projections in networking [3], the debut of new media technologies in storage [33], the novel buses and interconnect [16, 18, 27], etc.

At the same time CPU speeds are stagnating, and although parallelism and heterogeneity enhance CPU compute rates, many software cannot be easily parallelized, may suffer from synchronization bottlenecks, or exhibit ample overheads if moved to accelerators—thus, demanding efficient single CPU execution, only achievable with lean software. Experiments demonstrated that on the hardware introduced in Section 6 reading a file from the page cache achieves a peak throughput of 13.13GB/s in kernel, and 3.93GB/s in user-space. Using a RAID storage device, 1.85GB/s in kernel and 1.61GB/s in user-space—the difference increases for faster IO devices. These motivate the timely rethinking of OS we propose [8, 17, 54, 68].

Why the proposed solution? As mentioned in the introduction, approaches like kernel bypass and DPDK/SPDK work well when the goal is to implement an appliance on a single machine or virtualized hardware. Hence, these approaches are not suitable when applications need to share the same IO resources [21, 57], which maybe the case when using dev-ops

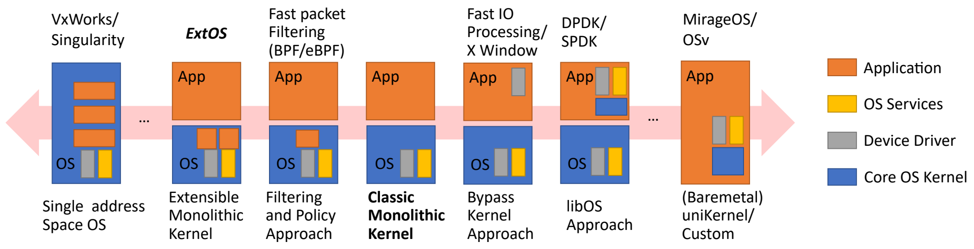
VxWorks/ Singularity
ExtOS
Fast packet Filtering (BPF/eBPF)
Fast IO Processing/ X Window
DPDK/ SPDK
MirageOS/ OSv
App
OS
...
Single address Space OS
Extensible Monolithic Kernel
Filtering and Policy Approach
Classic Monolithic Kernel
Bypass Kernel Approach
libOS Approach
(Baremetal) uniKernel/ Custom
Application
OS Services
Device Driver
Core OS Kernel

Figure 1: Monolithic kernel OS design space.

tools, bio-chemistry frameworks (e.g., freebyes, SAM, bamtools LAMMPS), LAMP/LEMP, etc. Beside that, shared IO has been shown superior to isolated IO [41], this is because uncoordinated accesses to the same storage media, or other IO devices, may cause interference, e.g., long seeks in HDD, or write amplification on SSDs. Moreover, virtualization does not have zero overheads [23] nor is supported by every IO device.

On the other hand, we acknowledge that kernel bypass and DPDK/SPDK are just user-space programs, therefore easy to deploy, debug, etc. Therefore, our future work will investigate tools to ease manageability of the code to be pushed down into the kernel. We envision such tools to be similar to what developers use when programming accelerators [50] to simplify adoption. Moreover, circumventing the kernel methods, and ExtOS, require modifying applications' source-code, hence tools to ameliorate programmability are sought.

Finally, since kernel bypass and DPDK/SPDK are user-space programs, they are oblivious of other programs running in the system and to the hardware topology. Therefore, they cannot capitalize on such knowledge to further improve performance, while this is possible when operating in the OS kernel—what we are proposing.

3 RELATED WORK

Extensibility in operating systems. Although today there is no commercial, nor open-source, fully extensible monolithic OS, extensible OS research was blooming in the 90's [13, 25, 42, 43, 63]. Microkernels [2, 10, 43, 62, 64] are extensible OSes by design, untrusted code that customizes the OS is separated by the microkernel via hardware memory protection. SPIN [13] is a research microkernel that enables OS extension by pushing down code into the microkernel, the code have to be written in a safe language, Modula-3. Exokernels [25] move all OS functionalities into the application that is compiled and run with its tailored OS (cf. libOS). Additionally, exokernels may be extended by "downloading" untrusted application code (network filtering, see below) into the kernel itself. To the best of authors' knowledge, VINO [63] is the only attempt to build a truly extensible *monolithic OS*. VINO is based on NetBSD, and like SPIN it provides different forms of application's defined OS extensions. Extensions are written in C/C++ and compiled with SFI for isolation outside the kernel.

Despite no fully extensible monolithic OS is available today, both Linux's and BSD's network stacks operations can be customized with application provided code, filtering functions [26, 46]. For security, the part of the application that is pushed down to the kernel must be developed in a reduced assembly language, BPF/eBFP [26]. Finally, these filtering functions affect all applications running on a system, hence only the sysadmin has full privileges to push down application code. This is fundamentally different from this work that aims to provide to any application the possibility to push down code in the OS kernel, without influencing other applications. Additionally, we identified BPF/eBPF to be too restrictive in functionalities and not security-hardened [28], hence we are looking at the possibility to extend it. It is worth noting that in the latest years BPF/eBPF subsystem in Linux has been successfully adopted in other subsystems than networking [15, 30, 39] However, none of these focused on any other IO subsystem than network. Only very recent work on File System in User Space [1] uses BPF/eBPF to exend the storage subsystem, but not in the kernel itself.

Finally, work on OKE [14] aims to enable unprivileged users to extend the OS kernel via kernel modules, securely. ExtOS pursues a similar goal but leaves kernel modules to privileged users.

OS mechanisms to mitigate data movement. Larry McVoy proposed the `splice()` system call [47], which copies the content of a pipe to a file without application involvement to mitigate OS-application data movements. The splice family of functions is now part of the Linux kernel. This work extends these syscalls to let the user attach application's defined operations to them.

For storage IO, the `mmap()` system call, which exploits the page cache as its core, provides the most performant form of zero-copy. However, our evaluation shows that code pushing down is consistently faster, especially on a kernel patched with KPTI (cf. Section 6). Another option for zero-copy storage IO in UNIX systems is to entirely skip the OS page cache. This can be achieved by using the `O_DIRECT` flag when opening a file, which is another way to circumvent the OS and has been previously criticized [38]. Our evaluations show that `O_DIRECT` has a variable latency based on the requested buffer size, the latency is at minimum double than the page cache read for small buffers, and it slowly improves with larger buffer sizes, but it is never faster than code push down.

Cosy [57] is an experimental OS that merges the ideas of mitigating IO data movements between kernel and user spaces and extending the OS kernel with application defined code. However, Cosy was built targeting storage operations only, and the source code is not available. The main difference with the proposed work is that Cosy doesn't provide a stream and filter interface and allows execution of system calls in the kernel itself (today completely disabled in Linux).

PipesFS [21] and Streamline [22] also aim to reduce the amount of copies between kernel and user spaces, as well as the number of context switch. Differently from this work, their focus is on network IO mostly. Moreover, PipesFS proposes a new file system while we aim to reuse most of the current Linux's interface—with minimum OS API changes, which will ease adoption. Similarly to PipesFS, our work extends to heterogeneous hardware components, but it additionally proposes to handle replication and load balancing.

4 DESIGN PRINCIPLES

The data-centric extensible OS targets UNIX-like monolithic OSes for which files are the main abstraction, giving access to IO device data—such as storage file system and network socket, but also applications' generated data such as UNIX pipes. The pushed down executable application's code extends the OS functionality for just about any IO subsystem. This enables the customization of OS's file access operations per application or system-wide.

The fundamental design principle beyond the ExtOS is performance, and reduced energy consumption is a side effect. However, since executing untrusted application's code in the OS kernel is a security concern, security is another design principle. Additionally, ease of programmability and manageability are also of fundamental importance, but we will fully address them in future work.

Improved performance is achieved by pushing down application's code into the kernel, which 1) avoids data copies to higher software layers, including user-space, of unused data; 2) avoids data movements to higher software layers, including user-space, of buffered data that is only accessed once (such as UNIX's `cp`, `grep`, etc.); 3) reduces context switches (increasingly expensive due to bugs in CPUs [29]); 4) enables single application optimizations based on the kernel's knowledge of the hardware; 5) enables cross application optimizations based on the kernel's knowledge of all applications running on the platform. Note that pushing down application's code provides knowledge of the application's behavior to the OS kernel, removing the semantic gap between the two, leading to better resource scheduling.

Safety and security are achieved by enforcing the code to be pushed down to be compiled within a restricted assembly language that is verified for formal properties as proposed by [59, 69], and come with a form of attestation. The code is then just in time compiled with SFI to execute in kernel. We are also investigating runtime verification approaches to prove safety properties, such as memory safety [40] and termination [19], of the untrusted application code in ExtOS. Moreover, such codes can 1) be hooked only to predefined kernel functions; 2) invoke only a subset of kernel functions/variable based on their privilege levels. Obviously, this requires compiler tools to be supplied to the developer, as well as application rewriting. For further isolation, we plan to leverage ideas from the nested kernel approach within a monolithic OS to run the application-specific code [20].

5 ARCHITECTURE AND IMPLEMENTATION

In this section we sketch the ExtOS architecture along with a prototype implementation. To achieve widespread adoption of the proposed OS design in practical deployments, we base our architecture and initial implementation on Linux and its BPF/eBPF infrastructure for network filtering. This has the additional advantage that toolchains, based on gcc or LLVM, already exist to compile subset of C language, Lua, Java, and Go into the eBPF ISA [55]. Our system implements mechanisms to attach application's provided code to kernel's operations on IO, including files, sockets, pipes, to execute application's provided code in the kernel, and to merge subsequent operations on IO. It is worth noting that although BPF/eBPF has been chosen for the prototype, such ISA is too limited to express all control flows we envision to implement with ExtOS—still, we believe BPF/eBPF is the best starting point.

Extensible operations on IO paths. Learning from Linux's BSD network filtering we added function hooks at all layers of the Linux IO subsystems, including network stack, block storage, page cache, character devices, and VFS. For each of these, hooks have been inserted to register application's defined functions on the most common IO related syscalls (such as `open()`, `close()`, `read()`, `write()`) callgraph's functions; including the splice family of functions.

Despite Linux's BSD network filtering infrastructure already introduces methods and mechanisms to push down, remove, register and deregister an application's defined function in kernel at runtime (cf., `bpf()` syscall) an extended API to associate such functions with file descriptors at the granularity of processes is missing. Therefore, we extended the standard Linux's IO API to carry over additional parameters to specify what function, or group of functions, to be executed on a IO operation, and additional per-call parameters. This enables to invoke BPF/eBPF filtering per syscall, on a specific file descriptor, thus narrowing the scope of BPF/eBPF to the application level—that means that other applications may operate on the same file with completely diverse functions.

Finally, all about original versus "filtered" file content access will be addressed in future work. Note that in networking, the packets are eventually discarded when filtered, while storage introduces ephemeral files that may require further handling.

Application's code loading and execution. Linux's BPF/eBPF subsystem includes a static checker as well as a just in time compiler (JIT) [26]. When a BPF/eBPF is pushed in the kernel it is statically checked for isolation properties first, and then just in time compiled for safe execution. Properties include: what function can be called, if there are loops, if the code terminates, and if branches are within code boundaries. However, recent work [28, 69] and CVE reports [45, 67] shown that such static checking may not be sufficient. Indeed formal verification solves the problem [69], still it put the burden on the programmer and involve longer execution times. Therefore, we are extendeding Linux's eBPF JIT to use Software Fault Isolation (SFI), by exploiting hardware SFI support (e.g., Intel MPX [51]) when available.

Application's code should be written within a "stream and filter" programming paradigm that resembles data-flow programming, and extends BSD network filtering. The code takes as input one or more blocks of data that in the case of networking is a packet, while in the case of storage a block is a disk block (but it can also be a line of text, or any other user-defined data entry/record). Similarly, the output consists of a return value and one or more output streams; but output streams may not have fixed block size. Because data may span multiple disk blocks, each filter can carry over state data among multiple calls—that can be used to store a partial entry, line, record, etc.

In the data-centric extensible OS application's defined code may also access and invoke a stable subset of kernel's functions and global variables. Application's defined code is connected to kernel's functions at JIT time; therefore the code has to be re-JITted anytime privileges change; if the code doesn't have privileges to be connected to a function (or the function doesn't exist) the code is JITted anyway but the function always return an error code. Variables are checked at runtime and don't require re-JITting.

Merging IOs and application-defined codes. A single application may use multiple IO streams, and data flowing out from one stream may be eventually transformed and piped into another stream. The simplest example is the UNIX `cp`, which reads a file and writes out its content to another file. It is convenient for such example to compose the read and write operations into one operation, e.g., Linux's `sendfile()`, which executes in kernel space, thus avoiding kernel/user data copies and context-switches, and gaining in performance. However, UNIX `cp` implementations do not transform the data in input before writing it to the output, rendering the usage of Linux's provided splice functions limited to such trivial applications. For this reason, ExtOS enables the attachment of application's defined IO operations to those syscalls. Also, it supports in-kernel runtime composition of IO operations, and associated "filtering" functions.

Many application are built as orchestrated sequences (or graphs) of simple tools that communicates via pipes, sockets or files, such as dev-ops tools, bio-chemistry frameworks, LAMP/LEMP, etc. Each simple tool can be rewritten to push down application's code in kernel. However, it is interesting to observe that when the output of an application is the result of a "filtering" function as well as the input of the subsequent application, these two can be composed into a single function for performance. Thus, ExtOS also supports in-kernel composition of IO operations among multiple applications—without applications' source-code modifications.

6 INITIAL RESULTS

A prototype of the data-centric extensible OS has been built based on Linux 4.15.9 in order to assess its benefits. The prototype has been evaluated on a Huawei multi-socket server with four Intel Xeon CPU E7-4820 v2 at 2GHz for a total of 64 cores and 512GB of DDR3 RAM, a RAID storage LSI MegaRAID SAS 2208 with 2.7TB, one Intel SSD 750 400GB, and one Intel Optane SSD DC P4800X 375 GB. Drives are formatted with the ext4 file system.

Push down. Previous work already demonstrated that on modern hardware higher performance can be achieved pushing down code in the network stack [26, 49]. BPF/eBPF and DPDK show that reducing kernel/user transitions and data copies largely reduce network latencies. Thus, in the following we show that similar results hold with storage IO, and how results differ changing IO device latency and bandwidth on three storage devices.

A rudimentary extensible OS has been built around the Linux kernel by modifying the `read()` system call to execute a "filtering" function registered via the help of a kernel module. A first set of experiments have been performed with the goal of evaluate the potential speedups when pushing down basic database operations, specifically filtering and aggregation. One microbenchmark per operation have been built. The filtering microbenchmark selects what amount of data from a file will be used by the application, hence when the filtering function run in kernel space only the selected data is moved to the application, while the other is discarded. The results of the evaluation for a file of 32GB are presented in Figure 2 varying the selectivity, and the buffering size (different lines).

Experiments have been repeated for different file sizes, from 256MB to 100GB and trends are similar. The results show that higher the buffer size higher the speedup that can be achieved by pushing down, up to 80% improvement. Despite the page cache results provide a hard limit on the achievable speedups some experiments reveal that storage results can

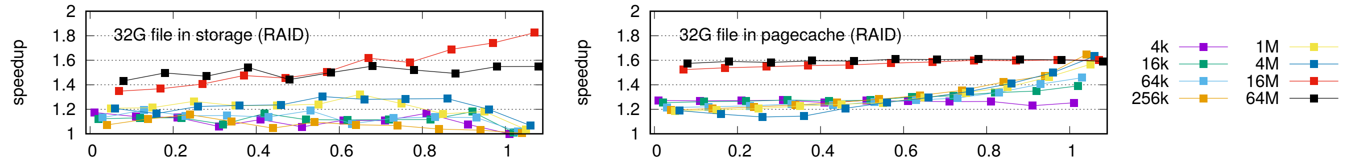

Figure 2: Filtering microbenchmark results for file on RAID storage and in page cache, varying selectivity (x-axis).

achieve higher speedups. The aggregation test, in which the push down code computes statistics on the data to be aggregated, such as min, max, and average, shows up to an order of magnitude speedups. We compared code push down with `mmap()`, and for the same set of experiments code push down is always faster, up to 49%, based on selectivity. This is due to fault-management overhead. Moreover, we compared with `O_DIRECT`, which is much worse than normal read for small buffer sizes (2 to 3 times), but it provides similar results to code push down for 4MB to 16MB buffer sizes, and deteriorates again for bigger sizes. This is due to the fact that is not always possible to copy data from storage to RAM, but it must be distilled from metadata or re-composed. For read/write, `mmap()`, and `O_DIRECT` KPTI reduce the performance up to ~ 30%.

UNIX tools. A second set of experiments have been performed on a real-world application to ensure microbenchmark results hold. The UNIX `grep` has been modified to comply with a stream and filter programming model. UNIX `grep` works in two stages: first it buffers a line of text, and then it searches for a text pattern. We modified UNIX `grep` to search for a text pattern while buffering. The new grep search code can be compiled as an independent unit and pushed down into the kernel. The grep application still issues read, but only matching lines are copied to user-space.

The original and the new UNIX `grep`s have been compared when searching a string in two files of 25GB with a match rate of 48% and 0.00002%, respectively. Pushing down part of UNIX `grep` in the Linux kernel has advantages. The low-selectivity case shows up to 2.52× speedup in the case of the page cache, while lower-latency high-bandwidth devices gave a higher speedup (up to 2.15× with the Optane). The high-selectivity case gives even better speedups (up to 4.83×) because input and output overlaps (see below).

Merging IO operations. A performance breakdown of UNIX `grep` shows that when the selectivity is high, it spends a lot of time in IO for `printf()`. The original version prints one line at the time, while the modified one a huge block of lines, hence the speedups. However, even greater speedups may be achieved by enabling the push down operation to "print itself", by "merging" IO operations in the kernel—in this case, the disk `read()` with the disk `write()` operations on standard output. To evaluate the gains in merging IO operations in kernel, an additional set of experiments have been run with UNIX cp. Such application copies one file to another, without any modifications, by reading the data from kernel to user space, and writing it back to kernel. The read and write calls have been substituted with Linux's `sendfile()` syscall that actually does the copy in kernel space without user space involvement (no kernel modifications have been made). Table 1 shows the execution speedups when using sendfile with a fixed buffer size of 4*kB* (FIX) and a variable one (SYS). Merging IO operations in kernel space provides up to 40% speedups when running on page cache, with variable buffer sizes.

	RAID FIX	RAID SYS	SSD FIX	SSD SYS	Optane FIX	Optane SYS
storage	3%	36%	2%	34%	1%	30%
page cache	7%	40%	7%	40%	7%	40%

Table 1: UNIX cp speedups using `sendfile()` with fixed and variable buffer sizes (FIX, SYS).

7 CONCLUSION

The data-centric extensible OS enables applications with strict low-latency IO requirements sharing the same IO resource between them—without one monopolizing it, and with other non IO-bounded applications, securely. Thus, democratizing the usage of emerging high-speed IO devices beyond the enterprise server market.

We shown that extensibility of UNIX-like monolithic OSes with application's defined code beyond the network stack is feasible in Linux, and can be achieved with modest rewriting by capitalizing on BPF/eBPF. The same "stream and filter" compute model can in fact be applied to any IO device, specifically to storage devices. However, BPF/eBPF is too limited, hence we sketch our plan to extend it. Additionally, we demonstrated that data filtering within the OS storage subsystem is largely beneficial, especially for modern storage hardware that is faster, reporting up to 4.8× faster execution of UNIX `grep`. Finally, merging IO operations and application's code in kernel also improve performance, up to 40%.

ACKNOWLEDGMENTS

The authors thank Anthony Iliopoulos for the very initial prototype that lead to this work, and the anonymous reviewers for their insightful comments.

REFERENCES

[1] Extension framework for file systems in user-space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, 2019. USENIX Association.

[2] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. pages 93–112, 1986.

[3] Ethernet Alliance. Ethernet's terabit future seen in new ethernet alliance roadmap, 2018. ttps://ethernetalliance.org/wp-content/uploads/2018/03/EA_Roadmap18_FINAL_12Mar18.pdf.

[4] Antonio Barbalace, Martin Decky, and Javier Picorel. Smart software caches. In *The 8th Workshop on Systems for Multi-core and Heterogeneous Architectures*, 2018.

[5] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. It's time to think about an operating system for near data processing architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 56–61, New York, NY, USA, 2017. ACM.

[6] Antonio Barbalace, Rob Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-ren Chuang, and Binoy Ravindran. Breaking the boundaries in heterogeneous-isa datacenters. In *Proceedings of the 22th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[7] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: Bridging the programmability gap in heterogeneous-isa platforms. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 29:1–29:16, New York, NY, USA, 2015. ACM.

[8] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, March 2017.

[9] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[10] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. Composing os extensions safely and efficiently with bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 239–252, New York, NY, USA, 2013. ACM.

[11] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazieres, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. page 14.

[12] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The ix operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Trans. Comput. Syst.*, 34(4):11:1–11:39, December 2016.

[13] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 267–283, New York, NY, USA, 1995. ACM.

[14] H. Bos and B. Samwel. Safe kernel programming in the oke. In *2002 IEEE Open Architectures and Network Programming Proceedings. OPENARCH 2002 (Cat. No.02EX571)*, pages 141–152, June 2002.

[15] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[16] CCIX Consortium. Cache Coherent Interconnect for Accelerators (CCIX), 2017. http://www.ccixconsortium.com/.

[17] Shenghsun Cho, Amoghavarsha Suresh, Tapti Palit, Michael Ferdman, and Nima Honarmand. Taming the killer microsecond. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[18] OpenCAPI Consortium. Welcome to OpenCAPI consortium, 2017. http://opencapi.org/.

[19] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 415–426, New York, NY, USA, 2006. ACM.

[20] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 191–206, New York, NY, USA, 2015. ACM.

[21] Willem de Bruijn and Herbert Bos. Pipesfs: Fast linux i/o in the unix tradition. *SIGOPS Oper. Syst. Rev.*, 42(5):55–63, July 2008.

[22] Willem de Bruijn, Herbert Bos, and Henri Bal. Application-tailored i/o with streamline. *ACM Trans. Comput. Syst.*, 29(2):6:1–6:33, May 2011.

[23] Ulrich Drepper. The cost of virtualization. *Acm Queue*, 6(1):28–35, 2008.

[24] Weimin Du, Ravi Krishnamurthy, and Ming-Chien Shan. Query optimization in a heterogeneous dbms. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 277–291, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[25] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. page 16.

[26] Matt Fleming. A thorough introduction to eBPF, 2017. https://lwn.net/Articles/740157/.

[27] Gen-Z Consortium. Gen-Z A New Approach to Data Access, 2017. http://genzconsortium.org/.

[28] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 1069–1084, New York, NY, USA, 2019. ACM.

[29] Brendan Gregg. Kpti/kaiser meltdown initial performance regressions, 2018. http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html.

[30] Brendan Gregg. Linux Extended BPF (eBPF) Tracing Tools, 2018. http://www.brendangregg.com/ebpf.html.

[31] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon,

Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 153–165, Piscataway, NJ, USA, 2016. IEEE Press.

[32] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *In Proc. of VLDB*, pages 276–285, 1997.

[33] F. T. Hady, A. Foong, B. Veal, and D. Williams. Platform storage performance with 3d xpoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, Sept 2017.

[34] Matthias Hille, Nils Asmussen, Pramod Bhatotia, and Hermann Härtig. Semperos: A distributed capability system. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, 2019. USENIX Association.

[35] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007.

[36] Intel. Storage Performance Development Kit (SPDK), 2018. http://www.spdk.org.

[37] Intel. BlobFS (Blobstore Filesystem), 2019. https://spdk.io/doc/blobfs.html.

[38] Jonathan Corbet. Page-based direct i/o, 2009. https://lwn.net/Articles/348719/, Online, accessed 01/05/2019.

[39] The Linux Kernel. Seccomp BPF (SECure COMPuting with filters), 2018. https://www.kernel.org/doc/html/v4.13/userspace-api/seccomp_filter.html.

[40] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 205–221, New York, NY, USA, 2017. ACM.

[41] C. A. Lai, Q. Wang, J. Kimball, J. Li, J. Park, and C. Pu. Io performance interference among consolidated n-tier applications: Sharing is better than isolation for disks. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 24–31, June 2014.

[42] W. S. Liao, See-Mong Tan, and R. H. Campbell. Fine-grained, dynamic user customization of operating systems. In *Proceedings of the Fifth International Workshop on Object-Orientation in Operation Systems*, pages 62–66, Oct 1996.

[43] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[44] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: a mobile operating system for heterogeneous coherence domains. *ACM SIGARCH Computer Architecture News*, 42(1):285–300, 2014.

[45] LWN. Linux >=4.9: eBPF memory corruption bugs, 2017. https://lwn.net/Articles/742169/.

[46] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association.

[47] Larry McVoy. The splice i/o model, 1998.

[48] Mellanox. ConnectX-6 200Gb/s Ethernet Adapter IC, 2018. http://www.mellanox.com/related-docs/prod_silicon/PB_ConnectX-6_EN_IC.pdf.

[49] Netronome. About agilio smartnics, 2019. https://www.netronome.com/products/smartnic/overview/, Online, accessed 01/05/2019.

[50] NVIDIA. Nsight Eclipse Edition, 2018. https://developer.nvidia.com/nsight-eclipse-edition.

[51] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018.

[52] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.*, 33(4):11:1–11:30, November 2015.

[53] Ian Pratt and Keir Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *IN PROCEEDINGS OF IEEE INFOCOM*, pages 67–76, 2001.

[54] Mia Primorac, Edouard Bugnion, and Katerina Argyraki. How to measure the killer microsecond. *SIGCOMM Comput. Commun. Rev.*, 47(5):61–66, October 2017.

[55] IO Visor Project. Bcc bpf compiler collection, 2018. https://www.iovisor.org/technology/bcc.

[56] The Linux Foundation Projects. Data Plane Development Kit (DPDK), 2018. http://www.dpdk.org.

[57] Amit Purohit, Charles P Wright, Joseph Spadavecchia, Erez Zadok, et al. Cosy: Develop in user-land, run in kernel-mode. In *HotOS*, pages 109–114, 2003.

[58] Andrew Putnam. Large-scale reconfigurable computing in a microsoft datacenter. In *Hot Chips 26 Symposium (HCS), 2014 IEEE*, pages 1–38. IEEE, 2014.

[59] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. Symdrive: Testing drivers without devices. In *Osdi*, volume 1, page 6, 2012.

[60] Samsung. Samsung pm1725a nvme ssd, 2018. https://www.samsung.com/semiconductor/global.semi.static/Samsung_PM1725a_NVMe_SSD-0.pdf.

[61] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 213–227, New York, NY, USA, 1996. ACM.

[62] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 69–87, 2018.

[63] Christopher Small and Margo Seltzer. A comparison of os extension technologies. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 4–4, Berkeley, CA, USA, 1996. USENIX Association.

[64] Andrew S Tanenbaum. A unix clone with source code for operating systems courses. *SIGOPS Oper. Syst. Rev.*, 21(1):20–29, January 1987.

[65] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Trans. Netw.*, 1(5):554–565, October 1993.

[66] Shivakumar Venkataraman and Tian Zhang. Heterogeneous database query optimization in db2 universal datajoiner. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 685–689, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[67] Common Vulnerabilities and Exposures. CVE-2017-16995, 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-16995.

[68] Daniel Waddington and Jim Harris. Software challenges for the changing storage landscape. *Commun. ACM*, 61(11):136–145, October 2018.

[69] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *OSDI*, pages 33–47, 2014.

[70] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. Netfpga sume: Toward 100 gbps as research commodity. *IEEE Micro*, 34(5):32–41, Sept 2014.