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:

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
and efficiently shared among all applications running on the OS. This idea is embraced in the data-centric extensible OS (Ex-
tOS) 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 opera-
tions, and transparently map the push down code to the avail-
able 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 im-

An important challenge is to ensure safety properties of the un-
trusted application code in the kernel. Specifically, we aim to ensure memory safety [51] and termination [19] properties to

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 hard-
ware 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
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/eBPF [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 extend 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 extending 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 I/Os and application-defined codes.** A single application may use multiple I/O 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 I/O operations to those syscalls. Also, it supports in-kernel runtime composition of I/O 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 I/O 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
achieve higher speeds. 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` has 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 `4kB` (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.

### 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


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


Antonio Barbalace, Javier Picorel, and Pramod Bhatotia


