HPC workload characterization using eBPF

Shubh Pachchigar
NERSC, Berkeley Lab
snpachchigar@lbl.gov

Brian Friesen
NERSC, Berkeley Lab
bfriesen@lbl.gov

Brandon Cook
NERSC, Berkeley Lab
bgcook@lbl.gov

Abstract

Efficient interactions with filesystems are essential for scientific workflows operating at scale on HPC systems. In order to design new filesystems, tune system configurations or identify applications for optimization efforts, effective I/O characterization is needed. Darshan is a widely used tool for I/O characterization that relies on injecting code into application binaries but has some limitations in providing low-level insights. In this work, we propose leveraging eBPF, a relatively new kernel technology that enables the execution of user-defined programs within the kernel, to develop a new I/O characterization tool. Our approach aims to complement the capabilities of Darshan by using eBPF to gain deeper insight into application interactions with the underlying filesystems. This is achieved by deploying dynamic instrumentation techniques such as tracepoints, kernel probes, and user probes below the application layer to extract detailed I/O metrics. In this work, we demonstrate the collection of many HPC I/O metrics of various filesystems at NERSC. The metrics are periodically collected and exposed via a pinned eBPF map with a custom LDMS sampler. The use of LDMS allows for the collection of data at scale for further processing and analysis. Finally, to demonstrate its feasibility in production HPC environments, we establish that the overhead of the tool is low. This work demonstrates the potential of eBPF to enhance I/O characterization in HPC environments, providing valuable insights that can lead to improved performance and resource utilization.

Keywords

eBPF, Darshan, LDMS, HPC, Filesystem, Linux

1 Introduction

High performance scientific workflows have a wide spectrum of I/O patterns[2]. At NERSC, there are several network filesystems that a workflow running at scale might interact with: Global common for read-only software, Global homes, Community (CFS) and platform Scratch. An efficient interaction with these filesystems, especially in large-scale environments using MPI-IO, POSIX or other interfaces, is important for maximizing workflow performance. In order to understand and improve the I/O of bare metal or containerized workflows and inform or tune the design of future filesystems, effective characterization of the I/O patterns is essential.

Darshan[10] is a mature HPC I/O characterization tool that provides insights into the I/O behavior of applications running on HPC systems. It generates summaries of I/O activity, including counters, histograms, timers, statistics, and full I/O traces. It is widely adopted across HPC facilities and its modular design allows easy extension to new I/O interfaces and storage components, making it adaptable to evolving storage technologies. However, there are concerns with Darshan modifying binary executables and injecting custom code for instrumentation. There are also potential inaccuracies. For example, Darshan does not understand the notion of virtual and physical memory pages, or caching mechanisms in filesystems or device drivers.

Extended Berkeley Packet Filter (eBPF) [6] is a new technology originating in the Linux kernel that allows sandboxed programs to run in privileged contexts, such as the kernel, without modifying kernel source code or loading kernel modules. It enables developers to extend the kernel's capabilities safely and efficiently at runtime. eBPF programs are primarily written using three tools: libbpf [7], bcc [5], and bpftrace [3]. Each operates on a different level of abstraction, bpftrace being the highest and libbpf the lowest and more powerful.

To explore the power of eBPF in workload characterization, we developed several programs using BCC which leveraged different kernel instrumentation techniques, such as user probes on shared objects, tracepoints on the syscall interface, and kernel probes on the VFS to trace application calls. The goal was to understand how the application interacts with different filesystems over time. However, this association proved to be complex, requiring multi-level pointer chasing within internal kernel structures. We discuss this challenge and other findings in more detail later.

We then use Lightweight Distributed Metric Service (LDMS) [1], which is a low-overhead, low-latency framework for collecting, transferring, and storing metric data and is widely used in many HPC centers. In this work, we also explore how to sample eBPF maps directly with a custom LDMS sampler and transport them to an LDMS aggregator for further processing in the pipeline. This approach was chosen to minimize the impact on the system while reusing existing infrastructure. The eBPF programs dump metrics into a special file, which is regularly sampled by LDMS, which then pushes the data to the aggregator.

The rest of the paper is structured as follows: we begin with the implementation of both the eBPF programs and the LDMS samplers, followed by the methodology and setup. Then, we discuss the overhead and present the results.

2 Implementation

In this section, we present the implementation details of two important components and explain how they work together. The first subsection provides pseudocode and necessary plots for generating the eBPF programs, while the second subsection describes the technical details of the LDMS sampler and its integration with

CUG25, May 4 – May 8, 2025, Stony Brook, NY

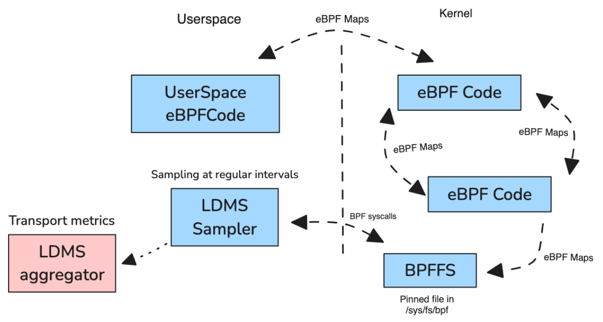


Figure 1: Software Architecture

eBPF. Figure 1 illustrates the overall architecture, and subsequent subsections will cover each component in detail.

2.1 eBPF programs

Three eBPF programs [9] were developed using BPF Compiler Collection (BCC), each targeting different layers of the HPC I/O stack using different kernel instrumentation techniques.

In the first eBPF program, we use uprobes — software breakpoints that enable dynamic instrumentation of arbitrary instructions or symbols within the userspace. Specifically, we target the MPI-IO routines provided by the OpenMPI library. Since the application under analysis links to this library as a shared dependency, these probes allow us to monitor the invocation of MPI-IO symbols in real time. This approach provides valuable insights into the usage patterns and argument values associated with MPI-IO function calls.

In the second program, we utilize kernel tracepoints—lightweight instrumentation hooks placed at strategic locations within the kernel code. Here, we attach tracepoints to the syscall interface to monitor the number of I/O-related calls made to the kernel. An eBPF map is used to store a `struct`, where the key corresponds to the process identifier (PID), and the value tracks the count of I/O calls issued by that process that reaches the kernel. For each `sys_enter` event corresponding to either a `read` or `write` operation—whether invoked through MPI-IO or any POSIX-compliant call, we wait for the corresponding `sys_exit` event. A successful exit indicates that the I/O operation completed correctly. This mechanism allows us to infer the presence of caching mechanisms in userspace above the kernel layer. Consequently, we can estimate the latency as seen from this level, effectively capturing the delay from one layer to the next in the I/O stack.

In the third program, we use kernel probes (kprobes), which are special breakpoints inserted into the kernel code. We attach kprobes at the Virtual File System (VFS) layer, specifically targeting the `vfs_read` and `vfs_write` functions. These probes provide insights into which specific filesystem is being accessed and I/O throughput for that application. The information we store are the filesystem type and the corresponding mount point. The mechanism for extracting this begins with the kprobe on the VFS function, which provides access to a `struct file`. This structure is the kernel's primary representation for each an open file, with one instance allocated per open file for MPIIO and POSIX I/O. To determine the filesystem type, we perform a series of pointer dereferences starting from `struct file`, which points to a `struct inode`, which in turn points to a `struct super_block`, and finally to a `struct file_system_type`. For mount point information, we again begin with `struct file`, which contains a `struct path`. `struct vfsmount` is a member of the `struct path`, which is also embedded within `struct mount`. Since we cannot directly get the base address of `struct mount`, we use the `container_of` macro to retrieve its base address. To enable this, we define a minimal version of `struct mount` within our script. This structure contains the necessary fields to extract mount point information.

Listing 1: BPF code: take pointer to struct file and move around kernel structs to get desired information.

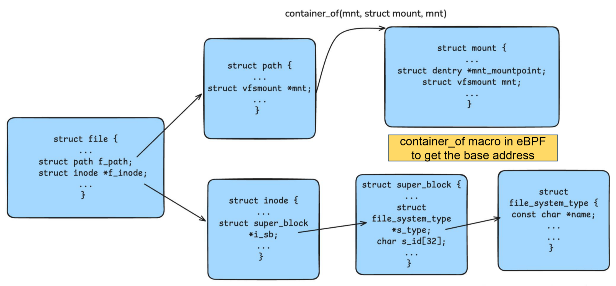
```
struct fs_stat_t {
    char fstype[16];
    char mountpoint[DNAME_INLINE_LEN];
};

static int trace_rw_entry(struct pt_regs *
    ctx, struct file *file, char __user *buf
    , size_t count)
{
    struct path p = file->f_path;
    struct vfsmount *vmnt;
    bpf_probe_read_kernel(&vmnt, sizeof(vmnt
        ), &p.mnt);
    struct mount *real_mnt = containerof(
        vmnt, struct mount, mnt);
    bpf_probe_read_kernel(&fs_info.mnt_id,
        sizeof(fs_info.mnt_id), &real_mnt->
        mnt_id);
    struct dentry *m_mp;
    bpf_probe_read_kernel(&m_mp, sizeof(m_mp
        ), &real_mnt->mnt_mountpoint);
    struct qstr m_dname;
    bpf_probe_read_kernel(&m_dname, sizeof(
        m_dname), &m_mp->d_name);
    bpf_probe_read_kernel(&fs_info.
        mountpoint, sizeof(fs_info.
        mountpoint), m_dname.name);
    return 0;
}
```

All the metrics gathered by the different programs are dumped to a special location in the BPF filesystem. Here, the eBPF maps, which are the primary communication method, are pinned and can only be extracted through the BPF syscall.

2.2 LDMS sampler and aggregator

We have created a new LDMS sampler [8] that collects metrics generated by eBPF programs in the previous section. These metrics are stored in eBPF maps, which are exported to the BPF filesystem.


container_of(mnt, struct mount, mnt)
container_of macro in eBPF to get the base address

```
struct file {
...
struct path f_path;
struct inode *f_inode;
...
}
```

```
struct path {
...
struct vfsmount *mnt;
...
}
```

```
struct mount {
...
struct dentry *mnt_mountpoint;
struct vfsmount mnt;
...
}
```

```
struct inode {
...
struct super_block
*i_sb;
...
}
```

```
struct super_block {
...
struct
file_system_type
*s_type;
char s_id[32];
...
}
```

```
struct
file_system_type {
const char *name;
...
...
}
```

Figure 2: Deep pointer chasing to find filesystem information in Linux

Since the BPF filesystem resides in kernel space, it can only be accessed via a BPF syscall. As a result, every time the LDMS sampler attempts to collect data, it first makes a BPF syscall to fetch the metrics, then updates its values accordingly.

Typically in LDMS, metrics are created first and then updated during the sampling process. However, in this case, the metric set is dynamic as new buckets or filesystem can be added with each read or write operation. To handle this, we use an LDMS record to initially populate the metric set, storing it in an LDMS list. Each record corresponds to a single metric, with the metric name formed by appending the filesystem name and the corresponding bucket. The value of the metric is either a count or a latency. In addition to the initial record list, a linked list is created to store all the LDMS metrics along with their corresponding LDMS metric IDs. This linked list plays a crucial role in the sampling process.

During sampling, the first step is to make a BPF syscall atomically in an LDMS transaction to read each entry from the pinned eBPF map. The sampler then checks the linked list to see if the corresponding metric ID already exists. If it does, the value is updated; if not, a new record is created, the value is set, and the record is appended to the existing LDMS list. For this process to work, the final static library for the sampler must be attached to libbpf, which allows the sampler to make the necessary BPF syscalls. Once the sampler is collecting metrics, we can push them to the LDMS aggregator for further post processing.

3 Evaluation

In this section, we first describe the computation platform and the various software used to implement this work. The initial part will focus on the essential components required to get everything up and running, along with the installation of side dependencies using the default package manager. The second part will then analyze the overhead of the three eBPF programs previously mentioned, within the environment detailed in the next section.

3.1 Computational platform and Software Environment

We ran developed our programs on a AWS EC2 instance with OpenSUSE Leap 15.6 and Linux kernel 6.4. For compiling to BPF bytecode, we used Clang 17. Additionally, we installed libbpf 1.5 from source to access all the necessary BPF helpers. We also utilized BCC 0.32 with Python 3.10 and bpftool for inspecting loaded eBPF code and map. We switched from Ubuntu because of some binary stripping causing removal of all symbols and to match production HPC system as closely as possible. For testing, we used Perlmutter TDS (Alvarez) with SUSE Linux Enterprise Server 15 SP5 having a HPE fork of Kernel 5.14. The presence of libclang in `ld.so.cache` can lead to build errors with eBPF programs. Rebuilding the cache without cray-pe.conf should solve this issue.

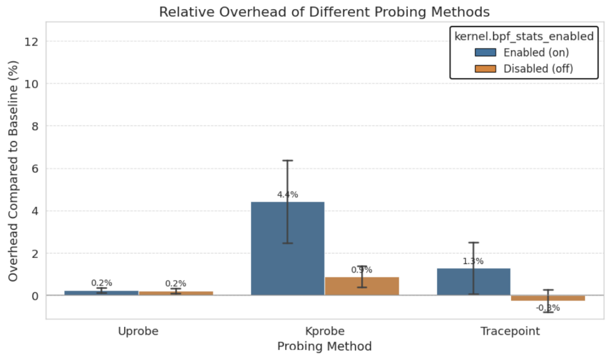


Figure 3: Overhead analysis on EC2 instance

3.2 Overhead anaylsis of eBPF

There are several ways to measure the overhead of an eBPF script, one of which is by setting `kernel.bpf_stats_enabled = 1`, enabling the tracking of cumulative execution time and the number of invocations of the eBPF program. Overhead can then be calculated as the ratio of total time spent to the number of calls. A comparison of three different kernel dynamic types—Uprobes, Kprobes, and Tracepoints—reveals distinct performance characteristics as shown in the plot above.

We conducted our experiments on an EC2 instance. We used IOR, configured to operate with MPI-IO. The IOR configuration were kept minimal: sequential I/O operations were performed using two MPI ranks on two physical cores, executing both read and write operations to the xfs filesystem. OpenMPI was used to make the MPIIO calls. To ensure statistical reliability, IOR configuration was executed three times, and the results were averaged to produce a single data point per instrumentation technique. This process was repeated five times to account for variability.

We also evaluated the overhead introduced by enabling the `sysctl` flag for BPF statistics collection. Specifically, we measure the impact of additional overhead from these runtime statistics for eBPF programs. Our results show that `uprobes` introduce the least overhead relative to the baseline. This is expected, as MPI-IO calls and their associated subroutines are not frequently invoked by the application or any other process. In contrast, `kprobes` and `tracepoints` exhibit slightly higher overheads. This is attributed to the fact that many processes, not just the target application, interact regularly with the kernel and trigger these probes. Interestingly, we observed that `kprobes` with BPF statistics enabled incurred higher overhead than when BPF statistics were disabled. The exact cause of this difference remains unclear and warrants further investigation.

Tracepoints are generally considered lighter-weight than Kprobes [4] and which is what we observe as well. Overall we see that overhead of these eBPF programs is mostly in the noise (<5%) and we are expecting similar results when tested with a real workload on Perlmutter TDS.

3.3 Results

The eBPF programs gave expected results when tested on Perlmutter TDS for a simple test application interacting with Lustre and GPFS. The programs are first loaded and corresponding eBPF Maps are pinned to the BPFFS, manually using bpftool. The LDMS sampler then samples the eBPF map for the relevant key and updates the corresponding value. In this case, the number of bytes read-/write at the VFS interface, captured by a kprobe, and its associated count are displayed. The key of the map consists of the filesystem type, mountpoint and a log2 bucket, while the value represents the count of I/O operations in that bucket. This demonstrates that the technique is feasible and can be used to gather a much more detailed analysis at a level below the application layer and in the kernel. The LDMS sampler fetches the pinned map and creates a custom key and pushes that to the aggregator nodes.

4 Conclusion

In conclusion, eBPF provides a unique capability to expose the lower layers of systems, allowing us to understand how and through which layers a request passes for I/O operations regardless of if the application runs in containers or bare metal. This capability enables developers and users to perform faster and more detailed analysis. While tools like Darshan work well at the application layer, eBPF can further enhance this analysis, complementing existing methods. We have demonstrated that eBPF programs integrated with LDMS can be deployed in a straightforward manner with minimal changes required on larger HPC systems. The overhead analysis confirms that eBPF introduces a very small proportion of overhead relative to actual application time. Future work will focus on deploying this solution on Perlmutter at NERSC, to assess its impact and overhead in production environments.

Acknowledgments

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DEAC02-05CH11231.

References

[1] Anthony Agelastos, Benjamin Allan, Jim Brandt, Paul Cassella, Jeremy Enos, Joshi Fullop, Ann Gentile, Steve Monk, Nichamon Naksinehaboon, Jeff Ogden, Mahesh Rajan, Michael Showerman, Joel Stevenson, Narate Taerat, and Tom Tucker. 2014. The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 154–165. https://doi.org/10.1109/SC.2014.18

[2] Jean Luca Bez, Suren Byna, and Shadi Ibrahim. 2023. I/O Access Patterns in HPC Applications: A 360-Degree Survey. *Comput. Surveys* (2023). https://doi.org/10.1145/3611007

[3] bpftrace contributors. 2018. bpftrace: High-level tracing language for Linux. https://github.com/bpftrace/bpftrace Accessed: 2024-12-20.

[4] Julia Evans. 2017. *Linux tracing systems & how they fit together.* https://jvns.ca/blog/2017/07/05/linux-tracing-systems/

[5] Matt Fleming. 2017. An introduction to the BPF Compiler Collection. *LWN.net* (December 2017). https://lwn.net/Articles/742082/

[6] Matt Fleming. 2017. A thorough introduction to eBPF. *LWN.net* (December 2017). https://lwn.net/Articles/740157/ Accessed: 2024-12-20.

[7] Andrii Nakryiko. 2022. Journey to libbpf 1.0. https://nakryiko.com/posts/libbpf-v1/ Accessed: 2024-12-20.

[8] Shubh Pachchigar. 2025. Custom LDMS sampler. https://github.com/shubhe25p/ldms-with-ebpf-sampler/tree/OVIS-4.4.2/ldms/src/sampler/ebpf_sampler Accessed: 2025-05-14.

[9] Shubh Pachchigar, Brian Friesen, Brandon Cook. 2025. HPC I/O collection with eBPF. https://github.com/shubhe25p/minimal-bcc Accessed: 2025-05-14.

[10] Shane Snyder, Philip Carns, Kevin Harms, Robert Ross, Glenn K Lockwood, and Nicholas J Wright. 2016. Modular HPC I/O characterization with Darshan. In *Proceedings of the 5th Workshop on Extreme-Scale Programming Tools*. IEEE Press, 9–17.