

A Linux Filesystem Performance Analysis Tool

Hyunsu Ha¹ and Sang-Young Cho¹,

¹ Computer Engineering Department, Hankuk University of Foreign Studies, Wangsan,
Mohyeon, Cheoin, Yongin, Gyeonggi, 449-791, Korea
csehsha@gmail.com, sycho@hufs.ac.kr

Abstract. A storage performance analysis tool is very crucial to find performance bottlenecks of I/O storage systems and to develop efficient storage system architectures or algorithms. This paper presents an integrated performance analysis tool for the Linux filesystem. The tool provides the actual time information of functions of the Linux filesystem. Contrast to other existing tools, the tool provides with filtering capability, graphical interface, and system-level analysis information without any heavy load of measurement. Linux developers or end-users can utilize it for analyzing filesystem layers and measuring software performance to find out the bottleneck within the Linux filesystem.

Keywords: Linux filesystem, performance analysis, EXT4, kernel trace.

1 Introduction

Android is an open-source Linux-based operating platform mainly for mobile devices. Recently, the platform is adopted to not only smart phones but also various smart appliances such as tablets, smart TV's, set-top boxes, and even interactive consumer electronics. Though the application performance of an Android-based device can be affected by many factors, the most influential factor is the underlying I/O storage system performance [1]. The storage system is a collection of software layers such as SQLite, EXT4, block device driver, and storage media driver. Each software module of the I/O storage system has been elaborated for hard-disk storage media and now reveals several performance issues when the integrated I/O storage system is applied to battery-powered mobile devices that are using NAND-based media. Therefore many researches have been devoted to improve the performance of I/O storage system.

The performance of storage system can be measured in various ways and there are existing benchmarking tools. The most well-known tool is IOzone [2] that provides numerous testing environments and data views with read/write speed and I/O turnaround time. IOzone shows performance results in statistical values but has difficulties to use for Android-based systems. AndroBench [3] was developed to benchmark the storage performance of Android-based mobile devices at the application layer. These tools can give the overall performance behavior in the system level but have lack of ability to report the detailed inter or intra function behaviors that are useful to analyze the exact timing behaviors.

There are many function-level performance analysis tools, also known as function tracers [4], such as ftrace, perf, strace, dtrace, and blktrace. The tools allow users to trace almost every functional-level events within the Linux kernel including filesystem domain functions. However, the tools do not provide any convenient interfaces to interpret or to filter the trace results, thus users have difficulties in translating the trace results into useful analyzing information. The tools cannot focus on the trace events that users are concerning and usually gather all the kernel events including even *dmesg* events due to the kernel logging system. Furthermore, each tool is specialized in a sub-kernel domain. To get a system-level performance information, several tools should be used at the same time and integrate each tool's trace information to generate a whole system-level performance data.

A storage performance analysis tool is very crucial to find performance bottlenecks of I/O storage systems and to develop efficient storage system architectures or algorithms. In this paper, we describe the design and implementation of a powerful Linux filesystem analysis tool that provides with filtering capability, graphical interface, and system-level analysis information without any heavy load of measurement.

2 Related Studies

2.1 Linux file system event tracing

Linux filesystem can be analyzed through investigating the function call sequence of the kernel when read or write system call has invoked by a user process. The whole performance analysis information is acquired by collecting the timing data of every invoked function's call and return and processing them to draw the entire picture of read/write process. Though read and write system calls are all frequently invoked file-related functions in the Linux filesystem, write is the major influential system call of the entire storage system performance when the filesystem is constructed on NAND-based storage media. This is because the write operation of NAND-based media is more time-consuming than the read operation and journaling operation is performed during the write operations. Therefore, we focus on the analysis of write operation of the Linux filesystem.

Figure 1 shows the Linux filesystem layers and the detailed function-level calling behavior of the write system call. The Linux filesystem has a layered structure that is divided into layers of VFS(Virtual File System), EXT4(or others), generic block layer, I/O scheduler layer and block device driver, and physical storage media. A write system call invokes a series of function calls through layers and finally makes the storage media be accessed. To trace the concerned functions of layers, actual function names are required and it is burdensome to remember the names of functions. Each layer has its own jobs to perform for file operations and consists of related functions. For instance, `ext4_writepages()` function of the generic block layer passes the divided page information of a write system call to the I/O scheduler layer. Based on the layers,

it is desirable to provide users with function lists and for them to choose functions to trace.

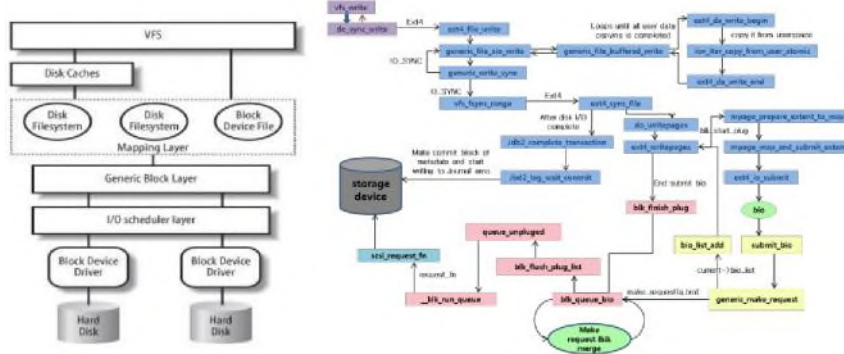


Fig. 1. Linux filesystem layers and the internal functions in case of write system call.

2.2 kretprobe

A static or dynamic Linux kernel tracepoint is a kind of breakpoint that provides a hook to call a function (probe) to extract kernel behavior information [5]. Static tracepoints cannot be inserted at run-time but they can be located any places in a program. Dynamic tracepoints can be easily inserted or enabled during program-run but they have some overhead and usually place at function entry and exit. The performance analysis tool of this paper was aiming at dynamic focusing capability and the probing overhead may not be significant because we concerns only timing information but local variable investigation. Therefore we used “kretprobe” that is a kind of dynamic probing tracepoint [6].

Kretprobe can be applied within a kernel module using the following function forms:

```
int register_kretprobes(struct kretprobe *p)
void unregister_kretprobes(struct kretprobe *p).
```

Parameter “struct kretprobe *p” refers to information of a tracepoint given by the user and its structure is showed in Figure 2.

```
struct kretprobe { struct kprobe
  kp; kretprobe_handler_t
  handler; kretprobe_handler_t
  entry_handler; int maxactive;
  int nmissed;
  size_t data_size;
  struct hlist_head free_instances;
  raw_spinlock_t lock;
};
```

Fig. 2. Definition of struct kretprobe.

“handler” refers to the custom tracepoint handler at a registered function return point and “entry_handler” refers to that of the function entry point. Because processor has to wait until each handler returns and this incurs probe effect of program execution, there are some variables to control system resources. To manage the excessive occupation of resources, “maxactive” is set to be the maximum number of kretprobe instances processed at the same time. If the number of simultaneous kretprobe instances exceeds the value of “maxactive”, the value “nmissed” increases by one for every missed handler. “data_size” is used for indicating the size of data sent from an entry handler to a corresponding return handler and “free_instance” holds the pointer to the list of free kretprobe instances.

When a function registered by the kretprobe interface is called, a breakpoint exception occurs and the “pre_handler_kretprobe” function is executed. This function allocates a kretprobe instance from “free_instance”, and calls an entry handler if it is defined and finally calls the “arch_prepare_kretprobe” function to prepare kretprobe process including registering the instance onto “kretprobe_inst_table”. The “arch_prepare_kretprobe” function is an architecture-dependent function that switches the address that the stack pointer is indicating with the address of “kretprobe_trampoline”.

After returning from the entry handler, the original function executes its own code until it reaches the return point. At the return point, the function jumps to the address of “kretprobe_trampoline” set as above. “kretprobe_trampoline” is an inline assembly routine that saves and loads the values of registers to call “trampoline_handler” that looks up a kretprobe instance from “kretprobe_inst_table” and executes the custom return handler.

3 Software architecture and system operating scenario

Figure 3 shows the software structure and the operating scenario of the storage performance analysis tool.

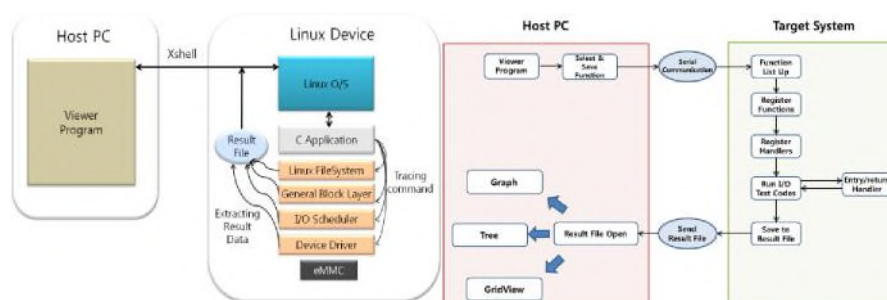


Fig. 3. The structure and operating scenario of the storage performance analysis tool.

A kernel module has been implemented under Fedora 3.13 using the kretprobe interface to store the tracing data of the Linux filesystem on main memory. The collected data is written on a text file after executing a test application that has file

access operations. To do this, user sets up the testing environment including writing a test program and choosing functions to trace. The testing environment setting is written as a text file and transmitted through Xshell via serial communication. The target Linux system receives the text file and configures a corresponding test environment based on the information of the file. The kernel module that controls the whole trace operation starts gathering tracing data with kretprobe interface. The result file is transferred to a Windows host PC through Xshell and analyzed by a viewer program that is developed under .NET framework 4 in C# language. The viewer program provides the storage performance analysis information using visualized interface such as graph, grid view and trees.

4 Implementation

There two main programs implemented: a trace module in the Linux kernel that exploits the mechanism of recording tracing data of file-related functions and a viewer program which is responsible for configuring test environment, choosing functions to trace and providing graphical result views.

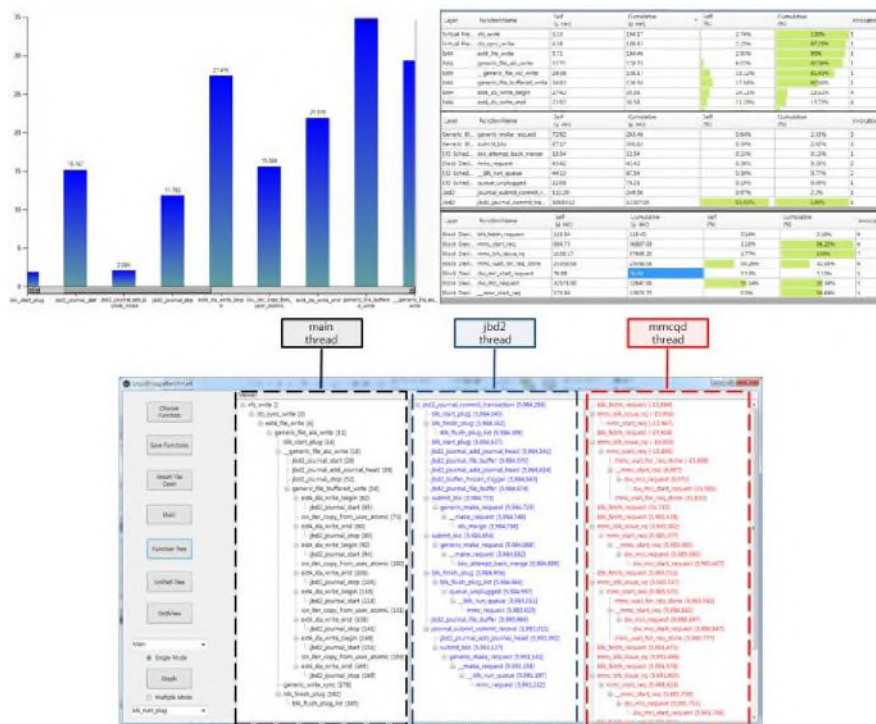


Fig. 4. The graphical three views of the viewer program: graph, grid, and tree.

The trace kernel module consists of four major parts: initializing a global list for storing tracing data, registering all the functions given in the input text file, handler routine that measures time at a function entry and return points, prepares the logging messages, and links the message data to the global linked-list, finally, unregistering functions, freeing allocated memories and record data in global linked-list on a result text file.

The viewer program uses two data structures. Tree is used to calculate the time values of each kernel function call and map is used to collect cumulative data of functions. Applying these structures, graph, grid view, and tree are provided for the three performance views and user can analyze performance via these interfaces.

5 Conclusions

A storage performance analysis tool is very crucial to find performance bottlenecks of I/O storage systems and to develop efficient storage system architectures or algorithms. The performance of storage system can be measured in various ways and there are many existing benchmarking tools. However some tools provide statistical results at the top level but lacks in detailed reporting. Other tools enable us to investigate the detailed behavior of inter and intra file-related kernel functions but within restricted portions of the kernel in inconvenient interfaces. In this paper, we describe the design and implementation of a powerful Linux filesystem analysis tool that provides with filtering capability, graphical interface, and system-level analysis information without any heavy load of measurement. The tool mainly consists of an in-kernel tracing module and a performance viewer. The kernel module has very low probing overhead compared to other existing tools and the viewer has three types of performance viewers.

References

1. Jeong, S., Lee, K., Lee, S., Son, S., Won, Y.: I/O Stack Optimization for Smartphones. In: 2013 USENIX Annual Technical Conference, pp. 309—320. San Jose (2013)
2. IOzone, <http://www.iozone.org/>
3. Kim, J.-M., Kim, J.-S.: AndroBench: Benchmarking the Storage Performance of Android-Based Mobile Devices. In: Frontiers in Computer Education, vol. 133, pp. 667—674, Springer, Heidelberg (2012)
4. Kernel Trace System, http://elinux.org/Kernel_Trace_Systems
5. LTTng Documentation, <http://lttng.org/docs/>
6. kretprobe, <https://www.kernel.org/doc/Documentation/kprobes.txt>