KOVA : A tool for Kernel Visualization and Analysis

Manu Awasthi, Krishna T. Malladi
{manu.awasthi, k.tej}@samsung.com
Samsung Semiconductor, Inc

Abstract—

The time spent by an application can broadly be classified into two main categories - user mode and kernel mode. In order to optimize applications from a performance perspective, it is critical to know the code regions where they spend the bulk of their time. With datacenter applications becoming more I/O intensive and storage devices attaining higher performance with each generation, the contribution of the Linux kernel stack to overall performance is increasing to an all time high. These trends make it imperative to observe and visualize kernel behavior and performance in order to effectively optimize it for specific use cases. To that end, in this paper, we present KOVA: Kernel Overhead Visualization and Analysis framework that builds on top of existing kernel tracers to provide comprehensive insights into Linux kernel behavior.

I. INTRODUCTION

The age of datacenters has arrived. Large amounts of data is being processed by an even larger number of “Big Data” applications and frameworks. So far, hardware has kept pace with Moore’s law driving the industry by providing better and faster compute systems. However, the pace of these advancements is beginning to slow down. A number of hardware manufacturers have announced reductions in fast cycles of architecture and process scaling [5] that have fed technological innovations over the past two decades. Nevertheless, the rate of data growth is continuing to increase as we move into exabyte era. This data is being sent to back-end data centers to be analyzed for meaningful insights. Furthermore, with the increase in the number of connected devices, the amount of data that needs to be stored and processed will grow at an unprecedented rate. In light of these trends, system designers face daunting challenges to provide architectures that can consume massive data within short latencies – a primary requirement for some, real-time interactive applications. Relying on hardware innovations alone is challenging as a result of technological slowdowns.

Even as compute systems face scaling challenges, data storage systems have continued to evolve with Solid State Devices (SSDs), moving from hard-disks, providing orders of magnitude faster IOPS and bandwidth. Such uneven advancements between compute and storage have begun to expose the large inefficiencies in software stack computations, increasing their contributions to the overall access latency relative to data fetch time [13], [4]. Furthermore, new trends in computing are taking hold. These include mechanisms like faster access to networked storage devices using remote DMA (RDMA over ethernet or other media) iSCSI and remote procedure calls (RPC). Similarly, computing tasks are increasingly distributed between the host CPU and an accelerator [6] or an FPGA [8]. All these developments increase kernel involvement in the computing process, unlike the traditional model where most computation is done in userspace. Context switches to and from the kernel space will be frequent, exacerbating the software overhead of the overall computational process.

Given these trends, to extract maximum performance out of any system, the entire software stack will have to be optimized. Multiple recent studies [4] have shown that the bulk of latency overhead in high performance SSDs is caused due to the time spent in the kernel. This component is only going to increase, as storage devices get faster and equipped with better memory technologies [13]. While kernel optimizations are essential, it is also a complex piece of software with a number of new and legacy components interwoven together for backward compatibility. This makes modifying and optimizing specific portions of the kernel rather difficult. Furthermore, many components have interdependencies, adding to the overall complexity. In order to effectively optimize or modify specific components, we need effective visualization aids to understand the different code paths being executed, as well as determine the temporal overheads of the different sub-components.

A number of existing tools already provide profiling and visualization for programs executing in userspace [2], [1]. The field for such tools in the kernel space has also been increasing, but (a) the number of tools is still small, and (b) tools aren’t user friendly. More importantly, methodologies to obtain specific insights for performance optimizations and reducing kernel overheads are sorely missing. In this paper, we present KOVA: Kernel Overhead Visualization and Analysis tool. KOVA builds on information provided by ftrace and provides a comprehensive framework to analyze the Linux kernel for performance overheads. In the rest of the paper, we present brief background of kernel tracing and then describe KOVA functionality in detail. Finally, we demonstrate insights derived from detailed quantifications enabled by KOVA.

II. BASICS OF KERNEL TRACING WITH ftrace

The Linux kernel comes prepackaged with an inbuilt tool for tracing the kernel function calls, called ftrace. Over time, ftrace has evolved into a much larger framework which can not only trace functions, but also provide latency details as well as trace specific kernel events [9], [10]. ftrace can be dynamically configured using debugfs. Furthermore,
depending on the required information, a number of tracers can be utilized. For example, the \textbf{function} tracer provides a time-stamped list of all kernel functions, their respective tasks/PIDs and the CPUs that they were executed on. Similarly, the \textbf{function graph} tracer provides a nested function graph of kernel functions along with timestamps and CPUs on which these functions were executed [11]. An example snippet of one read call graph as produced by the \textbf{function graph} tracer in ftrace is shown below:

\begin{verbatim}
gm-1 | | Sys_read() { 
gm-1 | | __fget_pos() { 
gm-1 | | __fget() 
gm-1 | | __fget_light() { 
gm-1 | 0.038 us | | __fget(); 
gm-1 | 0.306 us | } 
gm-1 | 0.575 us | } 
gm-1 | 0.846 us | }
\end{verbatim}

In this annotation, each line includes: CPU number, task ID, time for function completion and the function name and presents that in a function graph format. Other tracers provide different annotations for tracing information. Although, there are many more available options, we build KOVA mainly on top of these two tracers.

III. KOVA FRAMEWORK

KOVA's design identifies the most relevant aspects of kernel tracing and presents different visualization techniques to effectively analyze performance breakdowns. Many modern applications including No-SQL databases, front-end caches, graph analytics are often read-intensive. Moreover, I/O access are frequently random. While writes could be buffered at different levels in the system (DRAM, SSD buffers etc.) reads are difficult to cache, making their latency critical to overall system performance. Due to this consideration, we choose I/O read path system call for simple 4KB random read \textbf{fio} based experiments as the example throughout the rest of this paper.

Function calls could follow multiple code paths within the kernel, depending on the system state. For example, in the case of the read system call, the data could be served by the page cache in DRAM, shared pages, local disk or network attached storage. Moreover, depending on the path being taken, these operations could go through multiple layers of the stack including the filesystem, the block I/O layer, or the upper and lower SCSI layers, often invoking various kernel functions such as tasklets, synchronization and RPC primitives. Such complex interactions result in different call graphs. KOVA handles all such call paths to aid visual analysis and quantify performance breakdowns.

KOVA uses ftrace’s trace annotations to provide its functionality. The first step is to collect the relevant trace from ftrace. Currently, KOVA supports parsing and visualizing information from \textbf{function} and \textbf{function graph} tracers. For example, the user can instruct KOVA to collect detailed information of the specified function or system call (\textbf{Sys_read} in the rest of this paper) using the right tracer (\textbf{function graph}). In the second step, KOVA post-processes these traces and can analyze massive profiling information from ftrace and enables comprehensive visualizations based on the function call paths, latency breakdowns and by CPU basis.

Breakdown across all function calls. KOVA can parse the \textbf{function} trace to generate a breakdown of the time spent in all functions, across all code paths and CPUs. Below is the parsed information from a sample, I/O intensive \textbf{fio} run depicting the top 5 functions, in the order that they were called. KOVA breaks down the time spent in each one of the constituent functions called as a result of a read system call.

\begin{verbatim}
function | overhead (us) 
Sys_read | 384.687 
__fget_pos | 184.043 
__fget | 159.995 
__fget_light | 61.484 
vfs_read | 512.642
\end{verbatim}

Breakdown by time per path. KOVA parses the \textbf{function_graph} trace and finds the number and type of distinct call paths/ stacks that the function being traced has taken and produces a data set for all the called functions, across all CPUs along with the average time spent in each function, and presents this information as a table. The first column lists the constituent functions and every successive column (which represents an individual call path for the function being traced) depicts the average time (in us) that was spent on each one of those functions. If no time was spent in a particular function (i.e., the function was not called as a part of the particular call stack), the corresponding column is marked “0”. Using this information, KOVA can track any outlier functions that might have accounted for larger than usual time spent in any constituent functions.

Figure 1 shows the the average percentage of the time spent in the top 10 member functions for all of the different paths taken by \textbf{Sys_read}. Each individual part of a single stack shows the percent time that was spent in that particular function, ignoring the time spent in any of the functions that this function might have called.

Figure 1 presents many insights to assist kernel stack optimization. First, for this particular experiment, \textbf{Sys_read} follows four code paths, as represented by the four stacked bar graphs in the same figure. More importantly, we can observe that a significant fraction of the time in a particular functions across all different call paths for \textbf{Sys_read} is different.
Breakdown by cumulative time. Next, we can use KOVA to analyze each call path, individually. Figure 2 shows the distribution of time spent in the top 10 most executed functions for Path 0 while tracing the `Sys_read` command. This plot is the cumulative data across all the different call paths taken by the function being traced, which in this case is `Sys_read`. An examination of the above plot shows that the top 10 functions account for 50.4% of all the time spent in the kernel for the `Sys_read` system call.

Breakdown by code path. KOVA also enables deeper analysis by plotting the total time (in us) spent in a particular function for each code path. In this experiment, we have a total of 4 paths that were taken by the `Sys_read` system call. Visualization in Figure 3 considers one of the four, namely Path2. In this visualization, we plot the total time spent in the top 20 functions executed by Path2. We can see that one of the most “expensive” functions for Path2 is `do_generic_file_read()`, which accounted for 2.5us of total execution time. This guides us to prioritize optimizing the callee functions of this routine in the filesystem.

Breakdown by CPU. Yet another analysis that can be generated from the tool is visualizing the time spent in different functions on a per-CPU basis. For example, Table I presents the absolute time spent in the top 10 most active functions on CPU 10. As can be seen, some function calls add some overhead even though all they perform is calling platform specific functions. KOVA is able to visualize this information as a pie-chart, which we are unable to present due to space considerations.

IV. RELATED WORK

Tracing as a post-hoc mechanism to understand the inner workings of a computer system has been widely studied. The desire to understand and optimize the Linux kernel for specific purposes is a more recent phenomenon. So far, tracing kernel processes and events has been the only way to probe the workings of the kernel. A number of tracing tools with varying degrees of overhead and functionalities are available today. These include ftrace, LTTng [7] and SystemTap [3]. However, the field of frameworks to summarize and visualize traces is relatively small. KernelShark [12] is probably the only publicly available tool for visualizing ftrace output.

V. CONCLUSIONS

In this paper, we present KOVA: Kernel Overhead Visualization and Analysis tool that provides a comprehensive platform to visualize and analyze information provided by multiple ftrace tracers. Through multiple types of visualizations, KOVA allows system designers to gather useful insights into the interactions of OS with applications. Finally, we present a number of examples for utilizing the framework to gain insight into application behavior.

REFERENCES