Automatic I/O Scheduler Selection for Latency and Bandwidth Optimization

Seetharami Seelam, Jayaraman Suresh Babu, Patricia Teller
{seelam, jsuresh, pteller}@utep.edu
Department of Computer Science
The University of Texas at El Paso
El Paso, TX 79968

Abstract—Disk systems are increasingly accessed by multiple workloads with diverse characteristics and requirements. The Linux 2.6 kernel provides four disk I/O schedulers that cater to varied workloads but only one can be active at any time. As a consequence of this fixed scheduler scheme, potentially, some workloads obtain suboptimal performance. Currently, there is no automatic mechanism that can activate the appropriate scheduler. We present a novel approach called ADIO that provides latency guarantees while providing fair disk allocation to each I/O-generating processes. DASS, a feedback-based controller, sits on top of the deadline and CFQ schedulers and activates one of them based on observed system response to I/O requests while ensuring latency and bandwidth guarantees. Our prototype implementation in the Linux kernel shows improved I/O resource utilization, while enforcing strict latency bounds.

I. INTRODUCTION

The Linux 2.6 release provides four disk I/O schedulers: anticipatory, deadline, completely fair queuing (CFQ), and noop along with an option to select one of these four at boot time or runtime. The selection is based on a priori knowledge of the workload, file system, and I/O system hardware configuration, among other factors. In the absence of a selection, one of the schedulers is provided as default.

To provide best possible performance, I/O scheduler selection must exploit the workload characteristics, hardware configuration of the I/O system, file system etc. Although the Linux kernel (releases 2.6.11 and above) provides an interface to dynamically switch among the schedulers, diverse workload requirements and their I/O characteristics, and the idiosyncracies of the hardware and software make such selection a daunting task, at best.

Therefore, automating scheduler selection and doing so dynamically based on workload requirements, is an important objective. The question, however, is whether such an automatic and dynamic scheduler selection is feasible even for two schedulers.

This paper presents preliminary results of our work towards automatic and dynamic scheduler selection based on workload and system needs, and hence, attempts to answer the above-stated question. Note that such a scheduler selection methodology would be progress in the right direction, if it could achieve any performance improvement over the default scheduler. Providing a generalized and complete solution for the entire set of I/O schedulers is out of the scope of this paper, however, we present preliminary results w.r.t selection of one of two schedulers.

In mixed workload and multi-user systems it is critical to provide request response time with a bounded latency and maximize the utilization of the I/O subsystem. The deadline scheduler is designed to provide the bounded request latency, while CFQ is designed to share bandwidth equally among all active I/O applications, thus, increasing the utilization of the I/O system. However, scheduling optimized for bounded latency (as in the deadline scheduler) may result in reduced utilization of I/O resources. On the other hand, fair queuing scheduling (such as CFQ in the Linux kernel or SFQ, etc.) improves I/O resource utilization but may not provide bounded latency. We propose to select one of two schedulers, one that optimizes for bounded latency and one that provides fair queuing scheduling, according to the following two criteria.

1. Provide maximum request latency within a certain bound.
2. Provide best throughput, when the request latency falls below the latency bounds.

Thus, our Automatic and Dynamic I/O scheduler selection algorithm, called ADIO, is optimized for bounded per-request latency as well as disk utilization.

To evaluate this idea, we implemented ADIO in the Linux kernel to select between the deadline and CFQ schedulers. This selection is automatic and dynamic based on feedback from the I/O system and the workload. We tested our implementation on a RAID system as well a single drive system with synthetic and HPC workloads. We show that our system is able to switch between the schedulers and, thus, provide bounded latency and maximal throughput, which cannot be achieved with just any one of the schedulers.

The paper is organized as follows. Section 2 motivates the need for automatic and dynamic I/O scheduler selection, while Section 3 describes our methodology for scheduler selection. Section 4 presents an experimental evaluation of our methodology on a set of synthetic benchmarks and real workloads. Conclusions and future work are presented in Section 5.
II. THE NEED FOR AUTOMATIC AND DYNAMIC SCHEDULER SELECTION

The Linux 2.6 release provides four disk I/O schedulers to cater to diverse workload requirements in terms of throughput, per-request latency, average request latency, etc. The anticipatory scheduler (AS) is the default for kernels downloaded from kernel.org; it has been shown to provide best global throughput under several situations [10, 11, 1]. However, for kernels distributed by some of the major Linux operating system vendors, such as Novel, SuSE and RedHat, CFQ is the default scheduler as it offers the best global throughput and minimal per-request latency for a wide range of applications and I/O system designs [see, e.g., 6, 9]. The performance of a workload slightly different from the expected can be degraded [see, e.g., 12, p. 448] under the provided default scheduler. Hence, the Linux 2.6 release provides an option to select one of the four I/O schedulers at boot time or runtime. A scheduler selected at boot time is assigned to all the drives in the system; such a selection can be changed at runtime on a per drive basis.

From the cases described above and from the literature [see, e.g., 1, 2, 6, 9, 10, 11, 12], we could conclude that no one scheduler can provide the best possible performance (e.g., maximal throughput or minimal per-request) for all workloads. One possible solution to this conundrum is to develop one scheduler that can best serve different types of workloads; however, such a solution may not be possible due to diverse and orthogonal workload requirements in real world systems [3, 15]. Thus, one of the remaining options to achieve best possible performance is to provide multiple schedulers in the operating system and provide the capability to select one of them at runtime. Even though Linux provides four schedulers, the system administrator or a performance tuning expert is required to select one to satisfy workload needs based on observed workload system behavior and idiosyncrasies of system hardware and software.

Even when dealing with just four schedulers, in systems that service concurrent or non-concurrent workloads with different needs and I/O behaviors, such a selection, i.e., selection of the scheduler that fits well with the workload, is an intricate task even for an I/O performance expert. Systems that execute different kinds of workloads concurrently (e.g., a web server and a file server) – that require, individually, a different scheduler to obtain best possible performance – may not provide such performance with a single scheduler selected at boot time or runtime [9, 12]. Similarly, non-concurrent workloads with different phases, each phase with different I/O characteristics, is not best served by a priori scheduler selection. Of course, there are some “rules of thumb” for scheduler selection (see, e.g., 12, 9, 1), such as use CFQ to fairly share the available bandwidth among all processes, use deadline for database workloads that require real-time response, use AS for interactive desktop systems, etc. But scheduler selection based on these rules may not provide best possible performance [12, 9, 1]. Hence, a priori scheduler selection is a black art.

Accordingly, automating scheduler selection and doing so dynamically is an important objective. This paper presents our preliminary attempts at providing automatic and dynamic scheduler selection for a set of two schedulers, based on measured workload and system characteristics. We attempt to select one of the two schedulers to provide bounded latency and maximal throughput. Note that such a scheduler selection methodology represents progress in the right direction if it achieves any performance improvement over the default scheduler.

III. AUTOMATIC AND DYNAMIC SCHEDULER SELECTION METHODOLOGY

First, ADIO selects between the deadline and CFQ schedulers, ensuring that all requests have bounded maximum request latency. Once the maximum latency is below the desired bound, it optimizes for throughput.

Similar to the deadline scheduler, ADIO strives to impose 400 ms maximum latency for read requests (D_{r_{max}} = 400) and a four-second maximum latency for write requests (D_{w_{max}} = 4000). Both D_{r_{max}} and D_{w_{max}} are tunable parameters. At the same time, similar to CFQ, it tries to allocate system bandwidth as fairly as possible, but it imposes strict control on maximum latency. We assume that there is adequate bandwidth to meet the aggregate workload throughput requirements, as excessive workload inhibits any scheduling framework from providing bounded latency.

This latency and bandwidth optimization is done through feedback-based control and appropriate I/O scheduler selection component called DASS (similar to Façade). Figure 1 shows the structure of our ADIO scheduler framework for automatic selection. It consists of two I/O schedulers, deadline and CFQ, and a feedback-based controller, called Dynamic and Adaptive Scheduler Selector (DASS). Next, we outline the deadline and CFQ schedulers, and then describe the components of DASS.

A. I/O Schedulers: Deadline Scheduler

The deadline scheduler maintains two separate lists, one for read requests and one for write requests, which are ordered by logical block number—these are called the sort lists. During the enqueue phase, an incoming request is assigned an expiration time, also called deadline, and is inserted into one of the sort lists and one of two additional queues (one for reads and one for writes) ordered by expiration time—these are called the fifo lists. Scheduling a request to a disk drive involves inserting it into a dispatch list, which is ordered by block number, and deleting it from two corresponding lists. When the scheduler assigns deadlines, it gives a higher preference to reads; a read is satisfied within a specified period of time—500ms is the default—while a write has a five-second deadline. One should note that the assigned deadlines are not hard; sometimes there may be I/O requests in the pipeline that must be served before serving a request with an expired deadline. Thus, this scheduler strives to serve all requests within a specified period of time, providing an upper bound on latency of any particular request. This scheduler is useful for database, video compression, and other real-time workloads that require real-time response. Since all requests are placed in one global queue, this scheduler cannot maintain...
fairness in distributing the available I/O resources among all competing processes.

B. I/O Schedulers: Completely Fair Queuing (CFQ) Scheduler

The CFQ scheduler maintains one queue for every process class making I/O requests. A process class can be created based on process group id, thread group id, user id, or group id. During an enqueue operation, the request is inserted into a queue indexed by its process’s class id in FIFO order. During a dequeue operation, a set of requests from each of the non-empty queues is selected, sorted, and placed on a dispatch list, and are later sent to the disk controller. The number of requests fetched from each class is controlled by a tunable parameter, quantum. Thus, this scheduler aims to distribute the available I/O bandwidth equally among all process classes in the system. However, it does not impose any bound on per-request latency. This scheduler is used mainly in database applications that do not require real-time response [see RedHat05 and IBM05]. It also provides better I/O system utilization than does the deadline scheduler [RedHat05].

C. DASS Components

DASS consists of two components: a request statistics collection monitor and an I/O scheduler selection controller. These components determine when the current active scheduler should be swapped to satisfy the deadline requirements or to impose fair bandwidth when request latencies are within the bounds. These two components are described further below.

1. Request Statistics Collection Monitor: The monitor records a number of statistics reported by the block layer of each I/O scheduler. Every W seconds the block layer records these statistics based on I/O arrivals from workloads and I/O completions reported by the underlying storage system. We call $W$ the length of the adaptation time period. $W$ is a tunable parameter; we set $W = 1$. At the end of every adaptation time period $t$ (i.e., after every $W$ seconds), the monitor collects the following information:

- Maximum latency for any read request, $L_{r_{\text{max}}}(t)$,
- Maximum latency for any write request, $L_{w_{\text{max}}}(t)$,
- Number of read requests serviced, $N_r(t)$, and
- Number of write requests serviced, $N_w(t)$.

The above statistics are used by the controller (described below) to determine system status for the given time period as well as to decide if the active scheduler should be swapped out to accommodate latency and bandwidth guarantees.

2. Scheduler Selection Controller: After every adaptation time period, $W$, the scheduler selection controller makes a decision whether or not to switch from the currently active scheduler to the alternate scheduler. A decision point is needed since as requests are served by the current active scheduler, their maximum latency could vary during the adaptation time period. First, we present an intuitive explanation for appropriate scheduler selection and then give a precise algorithm.

If there are requests with expired deadlines, i.e., deadlines that exceed the deadline bounds $D_{r_{\text{max}}}$ or $D_{w_{\text{max}}}$, the controller selects the deadline scheduler. As soon as all request latencies are below their deadline bounds, the controller selects the CFQ scheduler. This scheme might suffer from few requests having high latency for a short duration due to noise in the system, resulting in frequent switching from CFQ to deadline and vice versa. To make switching schedulers immune to noise and other short term I/O bursts, the controller initiates a switch only if at least two requests exceed their deadline bounds within any given $W$. Once the deadline scheduler is swapped in, as soon as the request latencies fall below their deadline bounds, a switch to CFQ might happen immediately.

To avoid a frequent ping-pong effect, we initiate a switch from deadline to CFQ only when the maximum latency of any request is below 50% of the upper bound.

The switching involves writing the name of the scheduler into the sys file system. For example, for an sda drive, to switch to cfq the controller uses the following command:

```
echo cfq > /sys/drive/sda/scheduler
```

Scheduler swapping involves draining the current scheduler queues, initializing the new scheduler queues, and re-queuing requests in the selected scheduler queues. When there are a large number of requests, this operation might take a considerable amount of time, but it does not necessarily impact the effective utilization of the system. We discuss this further in the experimental evaluation section.

IV. EXPERIMENTAL EVALUATION

We evaluate our DASS controller using tiobench[14] benchmark. We conducted all of the experiments on a dual-processor (2.28GHz Pentium 4 Xeon with Hyper Threading) system with 1GB main memory and 1MB level 2 cache, running Linux 2.6.11. A total of four processors are used in the study. To eliminate the interference of I/O requests from the operating system (OS), all benchmarks are configured to access drives that are different from the disk hosting the OS.
The external device is a RAID-0 with four Maxtor IDE 7.2K RPM 20GB drives. It is configured with an ext3 file system; to remove cache effects, prior to each experiment, the file system is unmounted and remounted.

Figure 2: Request response with different schedulers

A. Enforcing Latency Bounds with the ADIO Scheduler Selection

This experiment focuses guaranteeing latency bounds with the ADIO methodology. Since latency bounds are imposed on a per-request basis by the DASS controller, to show ADIO’s effectiveness, we compare request response times of ADIO scheduler selection with those of the deadline and CFQ schedulers. For this purpose, we use tiobench, a file system benchmark that tests I/O performance using multiple threads that make simultaneous accesses in a sequential or random fashion. Given the number of blocks to read/write, read/write pattern (random or sequential), the block size, and the number of threads, this benchmark executes the operations and reports the average bandwidth provided by the system, average latency of request responses, and maximum latency of all accessed blocks. In order to determine whether or not ADIO provides a better latency, we augmented tiobench to report the latency of every request. We configure tiobench to read 2 GB data in 4KB blocks with 64 concurrent threads, which results in 8192 read requests. Request latency response time is shown in Figures 2 (a), (b), and (c) for the deadline scheduler, CFQ scheduler, and ADIO scheduler selection, respectively. We plot every other request response time to reduce the x-axis size by half. The 400ms deadline bound for each read request is represented by the dotted horizontal line in the figures. Observe that both deadline and CFQ schedulers have several requests that have latencies exceeding the deadline bound. However, ADIO scheduler selection strictly enforces the bound, hence, all requests have response times below the dotted line. We should reiterate that if the workload requirements exceed the I/O system capacity, ADIO scheduler selection will not be able to enforce the latency bounds. After all, these are enforced in ADIO scheduler selection by the underlying deadline scheduler. Now, the interesting question is: What is the impact of ADIO scheduler selection on the bandwidth of the application? This is explored in the next section.

B. Bandwidth Optimization with ADIO Scheduler Selection

This experiment compares the disk utilization of the three scheduling methodologies under four different workload profiles. For this experiment, we configure tiobench to access 2GB data in 4KB blocks using 2, 4, 8, 16, 32, and 64 concurrent threads. However, for this experiment, we configure it to execute sequential reads, random reads, sequential writes, and random writes. The bandwidth reported by each of the scheduling methodologies is shown in Figures 3 (a), (b), (c), and (d), respectively. The general observation is that in all the cases, ADIO, deadline, and CFQ have some disk utilization differences. Now to the specifics: for two of the four profiles, i.e., for sequential writes and random writes, which have a four-second deadline bound for each request, ADIO has better disk utilization than the default CFQ scheduler. For the other two profiles, the percentage differences among the schedulers corresponding to the reads are much smaller. CFQ and deadline have better utilization, but as shown in Figure (a) and (b), for streaming reads, although these schedulers achieve the higher disk utilization, several requests are not serviced in the latency bounds. In contrast, ADIO, by enforcing strict latency bounds, results in slightly lower disk utilization. ADIO clearly provides the bounded latency and optimizes for the disk utilization.

V. CONCLUSIONS AND FUTURE WORK

Providing latency bounds and fair utilization of I/O resources require an automatic agent to swap between the deadline and CFQ schedulers in the Linux 2.6 operating
system. Using feedback-based information, DASS is able to swap deadline and CFQ to enforce both latency and bandwidth constraints. Our experimental results, run on two different disk systems, show that DASS enforces the latency bounds, while providing better utilization than the default scheduler, CFQ. In order to cater to diversified workload requirements, we strongly assert that feedback-based control, such as that provided by DASS, should be an integral part of an operating system.

This work presents a novel approach to select between one of two disk I/O schedulers to enforce latency bounds and to optimize bandwidth. Currently, we are extending this methodology to select one of the four I/O schedulers in Linux to enforce several other requirements, such as maximizing bandwidth, providing best average latency, as well as bandwidth. We also are incorporating scheduler-draining time into the switching model to make the methodology more robust. Future analysis will quantify the memory and CPU requirements of I/O schedulers; such information may warrant using the noop scheduler, which has the least memory and CPU requirements, under some circumstances. Finally, there are a number of tunable parameters associated with each scheduler; with respect to these, we have started to investigate methodologies to select a set of parameters that optimize performance.

ACKNOWLEDGMENT

This work is supported by the Department of Energy under Grant No. DE-FG02-04ER25622 and the University of Texas at El Paso.

REFERENCES


Figure 3: ADIO, Deadline, CFQ Performance Comparison — Different Workloads