Input and Output Optimization in Linux for Appropriate Resource Allocation and Management

James Avery King

March 25, 2016

University of North Georgia Annual Research Conference

Abstract:

There is one evident area of operating systems that has enormous potential for growth and optimization. Only recently has focus been put on upgrading resources in the input/output (I/O) mechanisms of Linux operating systems. This focus has proven that there is no real optimal methodology for I/O scheduling devices in Linux. In order to allocate resources efficiently for time-intensive experiments on metadata and mobile devices, which both rely heavily on energy resources, Linux operating system developers must create new techniques for appropriately allocating these integral parts of computation. Advances must also be made to reduce the traffic in the file system alongside the optimization of energy resources in order to ensure that the system runs as efficiently as possible while aggregating different requests. Coupling the improvement of energy resources with that of request aggregation, as seen in the research presented in the collaboration of several national laboratories and universities, helps to maintain a higher throughput during run-time. With the advent of an ideal scheduler choice based on the I/O request, maximum energy efficiency methodologies, and the unification of I/O requests into a singular block, there are increases in the potential for throughput, execution time, state transition power consumption, and other expensive resources used by the Linux operating for their full capabilities. Even though these advancements are revolutionary and unique in many ways, they will only ultimately prove one thing: the process of diversification concerning research of I/O mechanisms in Linux plagues the majority of professionals in the field.

Introduction:

To understand the dynamics of the developments in input and output mechanisms in the Linux operating system, it is necessary to have a background in the basics of what I/O mechanisms are and their capabilities in any operating system. In general, input and output is considered anything that acts as an interface between the operating system and the computer user. This could include the keyboard, monitor, printer, mouse, microphone, or anything that is used by the user to input data into the operating system. These devices submit requests to be executed in a given order. The data submitted by the peripherals is then saved or manipulated through some sort of program present on the computer and outputted to the user in the intended format. According to the University of Illinois in Chicago (Silbershatz, n.d.), there are two primary conditions that every I/O has to overcome. The first is for developers to prefer existing types of I/O hardware and software when creating new I/O devices, which does not help in diversifying the approaches to input and output mechanisms in any operating system. Secondly, the conception of new types of I/O mechanisms are hard to apply to existing system due to the aforementioned condition.

Input and output hardware can be understood by three different primary classifications of hardware according to the University of Illinois in Chicago. These are storage, communications, and user-interface (Silbershatz, n.d.). The storage category has to do with accessing data in physical or virtual memory. An example of the hardware used for the communications category would be a system bus. User-interface is implemented through using encapsulated or layered architecture in order to manage respective device drivers for all of the different I/O mechanisms. These device drivers are categorized and accessed by the user through a “common interface for all devices” (Silbershatz, n.d.). An additional category to the previously mentioned three different main classifications is the Kernel I/O subsystem. This subsystem manages the execution of the different device drivers through a scheduling mechanism that includes some sort of prioritization. I/O schedulers in the Kernel help solve one of the largest modern problems in operating systems: the time it takes to move the read/write head from one part of a disk to another, commonly known as seek time. These four categories show just how important I/O is to any operating system. The handling of interrupts as well as the handling network traffic are integral parts of managing resources, with an emphasis on time, in every operating system.

There are two main concepts in understanding the operations of I/O in terms of scheduling. These are asynchronous or synchronous I/O. In asynchronous input and output scheduling, processes are permitted to begin execution at the same time that another process is running. This is a stark contrast to synchronous I/O process scheduling in which each process much completely finish execution before another process may be allowed to begin execution.

Input and output mechanisms are a completely essential part of the Linux operating system. As in every operating system, Linux includes I/O schedulers that vary in complexity. These schedulers perform two basic functions: merging and sorting. Merging is defined as the process of using two or more sequential I/O requests and forming them into a single request. Sorting is a term that is self-explanatory. When applied to I/O requests, sorting is the methodology of ordering the different I/O requests by their respective block order. The Linux operating systems allow users to configure their own I/O scheduler settings for optimum performance in the user space, commonly known as the command line. These initial concepts are all essential for understanding why most development in Linux I/O mechanisms are incremental. With such an array of inter-connected material it can be challenging for developers to create an enhancement that fulfills the needs of every single piece of the operating system.

Discussion:

The methods developed by researchers at the University of Erlangen show how merging can be used to more efficiently allocate resources for energy-aware applications (2002). Through their work, “Cooperative I/O – A Novel I/O Semantics for Energy-Aware Applications” (2002) Weissel and his colleagues “introduce a new operating system interface for cooperative I/O which can be exploited by energy-aware applications.” This interface was able to minimize the time was used by the operating system in an active state by nearly 60% in a variable-time read-operation experiment (Weissel, 2002).

To setup the experiment, Weissel (2002) and his colleagues identified a common problem concerning the limitations of an extremely valuable resource for computer systems: energy. The collaboration of researchers found that the main problem concerning embedded systems and mobile devices was the inefficient allocation of energy resources based on the state transitions of different components of the operating system (Weissel, 2002). They also found that there was a varying threshold, based off of the type of operating system and the programs being executed, for the transition into low-power modes; the threshold was only efficient, “if time for the next request is long enough” (Weissel, 2002). In order to solve this problem, the team implemented an interface that they designed specifically for batching, or merging, deferrable I/O requests with the intent of initializing more concentrated periods, during which the transition to a lower-power state would be beneficial (Weissel, 2002).

Through two experiments the researchers at the University of Erlangen were able to prove the effectiveness of their newly conceived interface. In the first experiment, the team used read operations and variable time constraints for the idle state of the computer to be implemented (Weissel, 2002). They were able to produce a lower frequency of mode switches with their new cooperative I/O interface as well as reduce the energy resources needed for an active state by 70% (Weissel, 2002). To conduct the second experiment the collaboration used varying lengths of state and idle time constraints for write operations (Weissel, 2002). This experiment was able to prove that even though Linux had implemented an update strategy to account for state transitions, the strategy did not “match power saving requirements” (Weissel, 2002). Through the second experiment’s parameters concerning the write operations, Weissel (2002) and his team were able to conclude that there was little to no effect on energy consumption concerning state transition.

Another team of researchers published in the ACM Transactions on Embedded Computing Systems journal were able to further improve the problem of state transition energy consumption in the I/O mechanisms of Linux operating systems. Their technique, I/O Burstiness for Energy Conservation (IBEC), was able to curtail the problem concerning “storage devices account for almost 27% of total energy consumption of computing system(s)” (Manzanaers, 2010). By having their experiment deal only with two transition states of the hard disk, sleep and active, the researchers were able to produce specified results concerning the aforementioned problem that was encountered (Manzanaers, 2010).

To optimize the energy efficiency of their computing system, Manzanaers (2010) and his colleagues set the goal of minimizing power transitions to make the parts of the operating system running in the background maintain a sleep state for as long as possible. Their methodology was similar to the researchers at the University of Erlangen in that focus was put on aggregating similar I/O requests into “larger contiguous blocks of requests when the disk is active” (Manzanaers, 2010). This technique allowed for a re-evaluation of the thresholds for power state transitions (Manzanaers, 2010). To prove the integrity of I/O burstiness, the team compared the results of IBEC with three prominent strategies at the time for lessening the burden on energy resources during state transitions.

Concrete results were produced through experimentation with the team finding that, “IBEC reduces the power consumption of real-time embedded disk systems by up to 60%” (Manzanaers, 2010). By being able to reduce the power consumption of the disk, the collaboration of researchers were able to increase the longevity of battery life in the Linux system implementing the IBEC as opposed to the Earliest Deadline First algorithm and its variations (Manzanaers, 2010). The most productive result of the research was that it minimized the sum of power state transitions in the hard disk while the device was executing a request-stream.

By additionally implementing aggregation methods on I/O requests, a team of from the IBM Linux Technology Center accompanied by academic professionals at the University of Texas at El Paso were able to further optimize the Linux operating system’s ability to efficiently allocate resources to I/O mechanisms. Not only did the researchers aggregate components of I/O, but they also designed a completely new form of node to handle I/O processing in conjunction with the compute node (Seelam, 2005). Their research served to solve the problem of, “data access rates of storage devices [not keeping] pace with the exponential growth in microprocessor performance” (Seelam, 2005).

To go about creating a solution to the given problem the team needed, “an increasingly large number of I/O devices…to provide corresponding I/O rates” (Seelam, 2005). The collaboration also sought to decrease the prominence of bottlenecks in I/O. By creating a variable ratio between the newly conceived I/O nodes and compute nodes the team was able to isolate a considerable amount of I/O traffic to the respective node instead of the normally used compute node. This alleviated the compute nodes of a majority of their burden of I/O interpretation, freeing up the entire computational capability of the Linux system in operation while at the same time, “circumventing the lack of direct connectivity to the file system” for the compute nodes (Seelam, 2005).

In a collaboration of researchers from Sandia and Argonne national laboratories as well as Ohio State University (2009) were able to revolutionize I/O scheduling through their publication, “Enhancements to Linux I/O Scheduling” in the second Linux Symposium (Ali, 2009). Through their research the team was able to further optimize the problem of I/O scheduling mechanisms in Linux. As of the release of Linux 2.6, there were four different I/O schedulers available on the operating system. These included deadline, noop, anticipatory (AS), and completely fair queueing (Ali 2009). During the system boot, the user was able to select the I/O scheduler that would accommodate whatever workload they intended to use during their time on the computer system through the Linux command line. The goal of the researchers was to create a new anticipatory scheduler by building on the original AS. By incorporating additional features into the AS, they produce the cooperative anticipatory scheduler to improve both throughput and execution time of different parts of I/O in Linux, although they concluded that there was no I/O scheduler in existence at that time that could optimize any given workload (Ali, 2009).

In the experiment, the researchers implemented two programs: Program A and Program B. Program A tested the ability of the I/O scheduler to execute “synchronous read requests by a single process” while Program B tested “a sequence of dependent chunk read requests, each of which was generated by a different process” (Ali, 2009). By running the programs independently, concurrently, and with web, mail, and file servers, the team was able to conclude that the CAS outperformed all pre-programmed I/O schedulers in terms of throughput and execution time. In some cases it was found that “CAS can run up to 62% faster in terms of run-time” (Ali, 2009).

Conclusion:

Aggregation is the common denominator of the first three sources used for evidence in the body of this paper. In order to efficiently allocate resources in I/O mechanisms most researchers look to couple together similar functions of I/O. Whether they are batching deferrable I/O requests to increase the productivity of active states, combining similar I/O requests into, “larger contiguous blocks of requests when the disk is active” (Manzanaers, 2010), or joining components of I/O to create the I/O node, each one sought to optimize the Linux operating system through aggregation. These techniques are building on one another in order to achieve similar goals. Additionally, the last source concerning I/O scheduling built on the idea of the Linux anticipatory scheduler. This demonstrates the original statement in introduction about the lack of diversity in the advancement of not just I/O mechanisms, but the entire Linux operating system. In order to truly advance there needs to be researchers thinking completely out of the box. Without unique ideas, the I/O mechanisms of Linux will only fester in the drawbacks associated with their older versions. The purpose of using these sources was to show that the majority of work being done is original in some ways but built on the same foundation of knowledge. Until researchers realize that they are only incrementally solving problems there will be no profound breakthroughs in I/O mechanisms of Linux operating systems.

Without the efforts of researchers in the fields of energy resource management, state transition, I/O request aggregation, system traffic, and I/O scheduling, the users of the Linux operating system would be waiting an exponentially longer amount of time with an exponentially lower amount of battery life for the execution of their I/O mechanisms and requests. Through the advent of cooperative I/O and I/O burstiness there have been major improvements in the amount of power that a computer running a Linux operating system needs in order to transition between the active, idle, standby, and sleep states, the amount of time spent in each state, and the efficiency of I/O request processing in the active state. Also as a result of the efforts of researchers, there have been breakthroughs in the Linux user’s ability to optimize the scheduling of I/O requests through the conception of scalable I/O and the cooperative anticipatory scheduler. Resources for I/O mechanisms that are in high demand from the computing and operating systems running Linux have only just begun to become allocated efficiently. Though perfection will never be attainable, optimization through methodologies that this paper has referenced will change the environment of I/O scheduling and resource management in the Linux operating system incrementally.

Bibliography:

Ali, N., Carns, P., Iskra, K., Kimpe, D., Lang, S., Latham, R., . . . Sadayappan, P. (2009). Scalable I/O Forwarding Framework for High-Performance Computing Systems. Retrieved March 25, 2016, from http://www.mcs.anl.gov/papers/P1594A.pdf

Silberschatz, A., Gagne, G., & Galvin, P. B. (n.d.). I/O Systems. Retrieved March 25, 2016, from https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/13_IOSystems.html

Manzanaers, A., Ruan, X., Yin, S., Qin, X., Roth, A., & Najim, M. (2010). Conserving Energy in Real-time Storage Systems with I/O Burstiness. *ACM Transactions on Embedded Computing Systems (TECS), 9*(3), February. Retrieved March 25, 2016

Weissel, A., Beutel, B., & Bellosa, F. (2002). Cooperative I/O—A Novel I/O Semantics for Energy-Aware Applications. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*. Retrieved March 25, 2016, doi:10.1145/1060289.1060301

Seelam, S., Romero, R., & Teller, P. (2005). Enhancements to Linux I/O Scheduling. In *Linux Symposium*. 2. 175-192. Retrieved March 25, 2016