Revisiting Operating System Mass Storage Presumptions Enables Higher Performance and Efficiency

Robert Gezelter, Senior Member, IEEE
Flushing, New York USA
gezelter@rlgsc.com

***Abstract*—Random access mass storage has been a pillar of computing since the 1956 delivery of the first IBM 350. The interface paradigm of position then transfer is pervasive, despite exponential advances in access speeds, transfer performance, density, and reliability.**

Consistency in interface paradigm does not mean that implementations have remained static, there have been numerous improvements: multiple heads, incremental positioning, recording technologies, and storage caches. While multiple levels of storage caches have become ubiquitous, caches are not a panacea. Caches benefit physically sequential usage and repetitive use by reducing physical media accesses. However, systems with hundreds or thousands of actively accessed files can clog caches with data that is only used a single time, yielding cache pollution.

Advances in semiconductors have vastly increased processing power and memory capacity, particularly since 1990. Solid state mass storage eliminates rotational delay, but can still have other implementation-related delays. Hierarchical mass storage provides the illusion of transparency, but as with virtual memory, delays attributable to data migration between levels cannot be concealed.

Through the 1980s, resource limitations obliged I/O infrastructure implementors to choose minimum resource implementations; other choices were infeasible. Feasibility is mandatory; efficiency and performance are desirable. Better resourced environments allow higher efficiency and performance with some increase in processing and memory consumption.

We will examine an approach that preserves the pre-existing I/O API, with minimal culturally-compatible extensions that enable higher performance and increased efficiency.

***Index Terms*—operating systems, mass storage, disk, input/output, performance**

I. Introduction

There is a fundamental disconnect between mass storage efficiency and the inherent behavior of multiprogrammed computing environments. Even simple mobile computers commonly have over a thousand active, independent threads, independently accessing hundreds or thousands of files. The different levels of abstraction, including files, volumes, and logical drives, among others, obscure the picture of actual physical device activity, making it difficult to optimize physical device activity.

Mass storage efficiency has been an ongoing research area since random-access mass storage debuted. One of the earliest answers to device contention was "use more physical disks." Much research has focused upon systems with a limited number of very active files. This focus has yielded some useful insights, but these insights have often proven less than ideal solutions, as was the case with *deceptive idleness* studied by Iyer. [1], [2]

Storage device throughput improves with increased locality. Most modern storage systems include one or more levels of caching; locality of reference improves cache hit rate. Classic rotating storage additionally benefits from reduced seek and rotational latency. Solid-state storage benefits from reduced operations. Hierarchical storage benefits from better retrieval planning. Processor efficiency benefits from reduced I/O delays.

Present I/O workloads are far more heterogeneous and dramatically more intense than in the past. Previous research has focused on microcosms, exemplar systems with one or a few highly active files and a single storage device. Event-driven simulation is a challenge with complex workloads, as there are numerous feedback loops in an I/O trace, small changes in timing can have dramatic effects on the downstream timeline, e.g., miss a sector on a classic rotating device, the timeline for the device defers an entire rotation. Benchmarking past a certain point requires extensive tests to understand the performance curve.

By contrast, this paper will show an approach that significantly improves throughput on systems with hundreds or thousands of threads accessing large collections of files. The approach does not require explicit synchronization between any of the processes other than classic read/write locking and arbitrary file fragmentation.The detailed derivation [3] is too lengthy for this paper, but small examples illustrate the benefits in common use cases, e.g., sequential processing of highly fragmented files. The foundational concept is that to achieve high performance, data transfers must be processed in a "device friendly" sequence, rather than the *block within file* sequence commonly implemented.

The techniques and approaches described in this paper are part of a pending US patent application.

Between the file and the actual physical device, there may also be multiple levels of virtualization.

Systematically eliminating unneeded serialization at all intermediate levels widens the optimization window at the lowest levels, yielding higher performance.

II. History

The generic I/O architecture is the accumulation of design choices, many made decades ago. Understanding the context of those decisions allows us to revist previous choices in the present context.

The 1956 introduction of the IBM 350 Disk Storage Unit revolutionized computing. Built as a peripheral for the IBM 305 RAMAC, the IBM 350 shipped with a capacity of 5 MB (later increased to 10 MB), with an average seek time of 600ms, and an average rotational latency of 25 ms (1,200 RPM). [4] IBM applied for patents, which were granted in 1964 and 1970. [5], [6]

Since that time, the speeds, capacities, and densities of random-access storage have increased dramatically, Table I.

TABLE I
Comparison: IBM 305 (1956) [4] and Seagate Exos X20 (2021) [7]

	IBM 350 Model 1	Seagate Exos X20
Year	1956	2021
Media Rotation Speed (RPM)	1.2K (50ms)	15K (4 ms)
Recording Density (BPI)	100	1.065M
Total Capacity (MB)	5MB	20,000,000MB
Physical Volume (m^3)	1.934656	0.0004
Weight (kg)	1000 kg	0.670 kg

The essential elements of operating system I/O mechanisms were well-established by the mid-1960s, as exemplified by IBM's OS/360 and its descendants. [8], [9] Later minicomputer operating systems, e.g., Digital Equipment Corporation's RSX–11 family and Bell Labs' UNIX followed the same pattern. [10], [11] Later operating systems including Microsoft Windows, Sun's Solaris, and Linux also have similar overall I/O APIs, which remain unchanged to the present day. [12]–[16] All have I/O APIs that convert a user-level request into a system internal data structure. The generated data structure describes the actual data transfer operation. The specified processing is managed by a *device driver*, a kernel module generally containing multiple entry points that implements I/O-related operations under the supervision of a *file system*, the software component that supervises volume usage and space allocation.

Many find it difficult to comprehend the computing context of that era, fifty years ago; mainframes with 1 MIPS processing power and 1MB of memory were the apex of the computing pyramid, operated by the largest corporations, research establishments, and universities. Smaller systems had but a fraction of these resources, e.g., the IBM System/360 Model 40 had 256KB of memory and clocked in at 0.034 MIPS. 1970s minicomputers were similar in capacity and performance to the previous decade's mainframes. As a consequence, there was often less than 10KB memory available for kernel structures.

The computing context changed dramatically by the 1990s. Desktop systems with more than 100 MIPS processing power and more than 100MB of main memory were common. The increase in main memory made far larger kernel data structures feasible.

Despite vastly expanded resource, core I/O architecture has remained as originally conceived. While the I/O APIs are pervasive throughout the pre-existing vast software base, the internal plumbing underlying those APIs remains hidden and not user-visible. While the syntax and semantics of the existing I/O API must be preserved, the underlying infrastructure can be re-examined and re-considered in the present-day resource context, provided the user-visible I/O interface semantics are preserved.

III. Previous Research

Large scale I/O tasks, e.g., sorting, tabulating, and searching, have been the raison d'être for data processing for more than a century. Large-scale sorting and tabulating has been a core major data processing tasks since the dawn of digital computing. I/O performance is fundamental to that task. [17] Fuller analyzed the low-level performance of rotating magnetic storage devices. [18]

Iyer observed that related operations often follow shortly after a previous operation. [1], [2] Iyer labeled this *deceptive idleness*. He proposed that devices be reserved for a short period of time after a request to see if a related request was forthcoming. Deceptive idleness was sufficiently promising that it was included in and made the default in Linux 2.6.0. Apparently, deceptive idleness did not yield the hoped for benefits, as Linux 2.6.33 deprecated the approach in favor of Completely Fair Queueing.

One challenge in increasing mass storage performance is the nature of the workload. An approach may seem to work for one class of workloads while having serious shortcomings with others. Experiments are often done in environments with one, or a few, high demand processes.

In effect, we are searching for ways to increase storage access efficiency akin to the use of multiprogramming to improve CPU efficiency. Process contexts focus on program counters, stacks, and register sets. Mass storage contexts center on the fundamental nature of the mass storage device, e.g., rotation and seek positions for rotating storage; banking in solid-state storage; and details of hierarchical storage. Target device awareness of upcoming requests enables optimization. Serializing requests at any level above that which is necessary impedes optimization and performance.

Consider what happens when a user-program requests a single n-block data transfer between a file $\mathcal{F}$ stored on mass storage and a buffer located in main memory. The virtual request, v may directly translate to a single request for a series of sequential blocks on mass storage or it may transform into as many as n separate operations on the underlying volume, each for a discrete set of sequential mass storage blocks.

There is no guarantee that the sequence of blocks within a file correlates in any way with how corresponding physical blocks are stored on the underlying storage media. Issuing one request at a time presumes just such a correlation. Issuing all of the component requests simultaneously enables optimization by storage device(s).

Consider a user-level file-based request v. Map v into an unordered set $\mathcal{L} = \{l_0, l_1, \ldots, l_n\}$ by completely mapping v at one time. The resulting $\mathcal{L}$ is unordered, there are no dependencies between elements of $\mathcal{L}$. Each l_n represents the transfer of a particular contiguous segment of the request v. All the $n!$ permutations of $\mathcal{L}$ are semantically equivalent. However, it is clear that semantic equivalence does not carry over into equivalent physical performance. Each ordered permtuation of $\mathcal{L}$ has a different duration depending on the characteristics and potentially varying state of the underlying mass storage device(s).

Deceptive idleness is file or device centric. While consecutive blocks within a file have a significant correlation to physical locations on an underlying physical volume, the correlation is imperfect, particularly when one considers non-contiguous files. [19] Even contiguous files may not be transferred in sequence, depending upon many physical variables including: caches; access arm position and rotational position in rotating storage; banking and other factors for solid-state storage. Physical volumes are increasingly rare, having been supplanted by logical volumes, virtualizations provided by intelligent storage controllers. This change creates a more complex reality, as there are several independent levels of abstraction underlying logical volumes, with each level having its own independent mapping. In turn, each of these mapping levels may mutate over time. Multiple, non-static mappings make the notion of "contiguous on a volume" a quaint anachronism.

Host operating systems cannot track the detailed internal state of storage devices accessed. Storage devices serve multiple hosts; they and their controllers may have multiple levels of internal caches; and other factors. Any attempt by a host operating system to model internal device/controller state is fated to fail.

Lack of accurate knowledge relating to the storage environment fatally impairs optimization at host level. There are many illustrative examples. Zoned Bit Recording (ZBR) is one. [20]

The anachronistic simple disk model is deeply enshrined in operating system I/O implementations. Present storage devices contradict the classic model in many ways. Rotating media devices and/or the associated controllers almost alwasys have caches. Solid-state storage, regardless of interface protocol, e.g., SAS, SATA, NVMe, has neither rotational nor seek delay, but may have caches and other non-uniformities. Multi-level hierarchical storage subsystems are composed from a collection of various storage technologies, each of which may have its own sequence-dependent timing issues.

Examples abound.

Many operating systems attempt to model seek ordering to optimize request sequencing using the classic cylinder/head/sector (CHS) model of a disk. The CHS model presumes that each disk track contains the same number of sectors. By contrast, ZBR media has a variable number of blocks/track. A fundamentally flawed model in concert with an indeterminate mapping to the physical sphere leads to incorrect or counterproductive optimizations.

Host-based ordering also does not take into account other possibilities including: the presence of caches between the host device driver and the physical media; and requests from other hosts. Inaccurate models, based upon an incomplete or incorrect presumptions lead to suboptimal choices.

Refocusing host operating system efforts from futile attempts at optimization to feeding storage controllers and devices the largest possible request pool, is far more productive. The larger queue sizes offered by modern SAS and NVMe device controllers is a step in the right direction, however full utilization requires changes above the block-storage layer. Increasing queue length enables more processes to have simultaneously active requests to a mass storage device, but the very increase in outstanding request sources, together with unneccessary serialization higher in the I/O stack, increases the time interval between successive, correlated requests from a single source and increases the diversity of queued requests.

Eliminating serialization and queueing at hosts and intermediate levels provides storage controllers and devices a more complete picture of the pending workload. Controllers and storage devices are free to subdivide pending requests and individually schedule each of the resulting sub-requests, so long as the sub-requests are completed before the parent request is considered completed, as required by the completion semantics of the parent request, Fig 1. Physical configuration and device knowledge is limited to the devices, which are the only components that actually have accurate, actionable knowledge of the physical device state.

IV. Standard Solutions

Through the first half of the 1980s, disk drives were physically large and expensive. The emergence of personal computers as a mass market item created incentives to produce far cheaper, modest performance mass storage devices. In 1987, Patterson, Gibson and Katz described how aggregations of inexpensive drives could implement higher degrees of performance and/or reliability. [22] The report included a taxonomy of approaches, termed RAID, an acronym for Redundant Arrays of Inexpensive Disks, including:

- RAID-0 Striping
- RAID-1 Mirroring

RAID-0 and RAID-1 often are combined, with the combination referred to as RAID-10

V. The Reality of File Systems

Raw random access mass storage is simply block addressable memory. Raw block addressable memory is not directly useful to an application. It was not even a question in the single user environments of the late 1950s, or later single user, single-process operating systems of minicomputers and personal computers, e.g., Digital Equipment Corporation DOS,

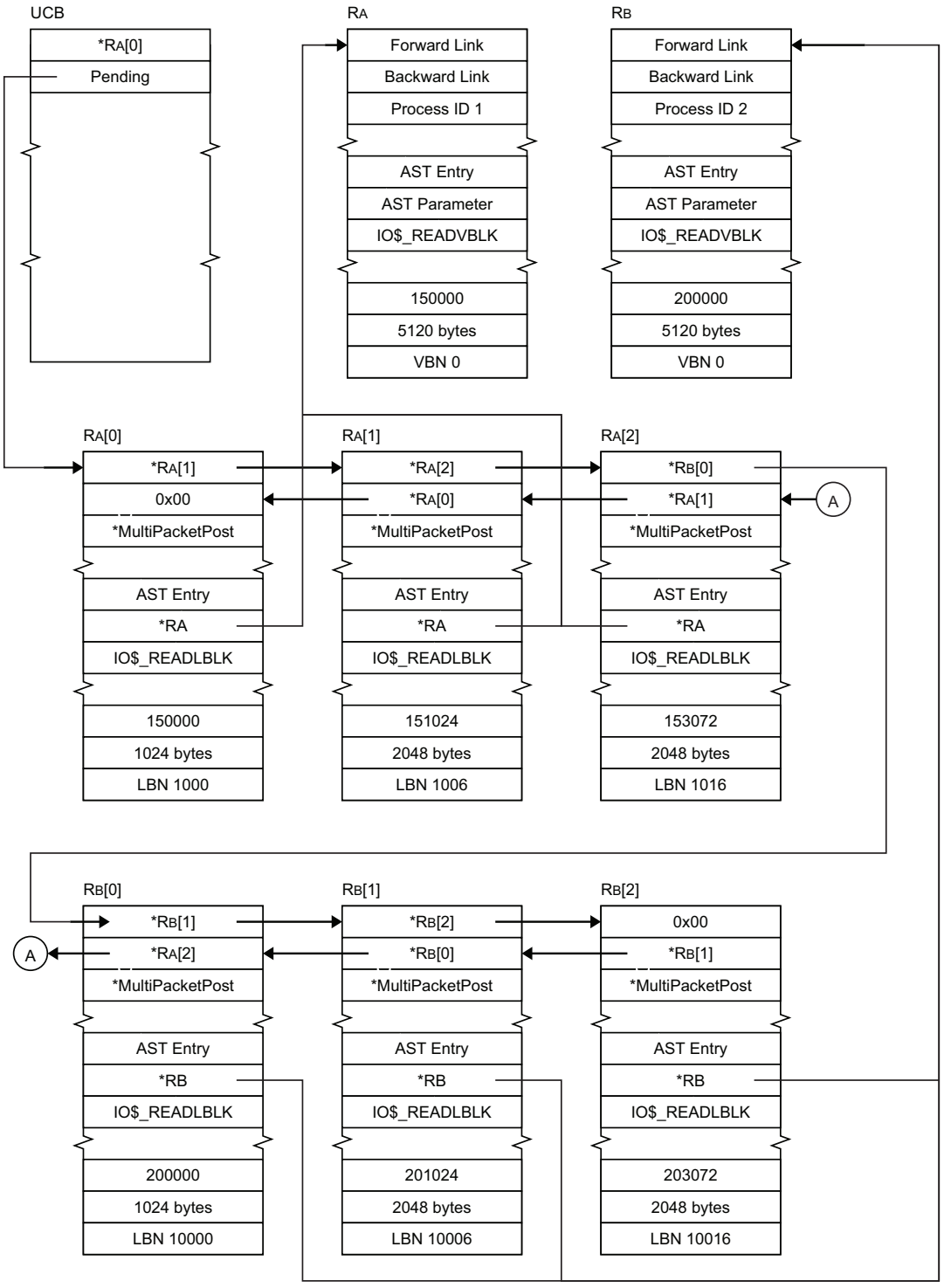

Fig. 1. User-level requests R_A and R_B are expanded into independent sub-requests using mapping information into $R_{A[0]} \cdots R_{A[2]}, R_{B[0]}, \cdots R_{B[2]}$, each referring to a contiguous set of blocks within a volume. The device controller may in turn further subdivide those requests to reflect how the volume is provisioned on actual media, e.g., RAID-0/1, logical volumes. Each of the resulting sub-requests is scheduled individually by the storage device according to the physical context at the media level. The *AST Entry*, if present, is invoked to signal the completion of each request. *Reprinted with permission from [21]*

Digital Research DOS, or Microsoft MS/DOS. Applications in general need to implement greater degrees of granularity, structure, and protection. Abstraction is even more crucial in multiple process and/or multi-user systems.

The first abstraction above a raw device is the concept of *files*, ordered collections of mass storage blocks within a storage *volume*. Operating systems use one or more *file systems* to administer files within each volume of otherwise indistinguishable blocks. File systems maintain data structures and associated definitions collectively referred to as a *file structure*. File systems also manage space allocation when files are created, extended, and deleted, maintaining the mapping between blocks within a file and the actual locations of those blocks on the volume administered by the file system. [23]

Increasing the size of a file at a subsequent time, referred to as extension, all but guarantees that the extended file is non-contiguous, containing at least two sequences of blocks that are discontiguous on the underlying storage volume.

Non-contiguous files are one of the most challenging factors when improving effective mass storage performance. Space is allocated as needed from the space available at the particular instant in time the extend is processed.

However, the importance of contiguous files is not all that it seems. A contiguous file is a useful simplification when using a primitive file system to bootstrap a system. If a single file dominates the I/O workload on a particular device performance may benefit from the simpler mapping. In constrained memory environments, data structure churn resulting from files with hundreds or thousands of discontiguous storage segments is a major problem. However, those problems are not always true in recent decades with the increased multiprogramming workloads and increased memory resources. On a multiprogrammed system, even a contiguous file accessed sequentially may not eliminate inefficiency, as other simultaneously executing programs may make other requests to the same physical storage device.

A single program request for n blocks within a file may or may not be a contiguous sequence within a volume. One cannot presume any relationship, e.g., increasing physical block number, between any two successive file blocks.

An initial allocation of 100 blocks may yield 100 blocks starting at block 200000; or a series of up to 100 segments, with block 200000 being only the first of the series. [19]

When file extension occurs at a later time, the extension is allocated space in the context of the space available at that instant in time. When the file is extended, the available space could be in the 500000-block range or in the 10000-block range. As at initial allocation, the space allocated could be a contiguous block range, or some number of discontiguous sub-ranges not less than the extension requested.

To make matters more complex, all requests are made against a constantly changing background of thousands of other executing threads, each of which may allocate or deallocate mass storage space.

It is difficult to transform the unpredictable potpourri of I/O requests issued at the convenience of uncoordinated executing programs into an efficient sequence of requests processed by storage devices.

VI. Rethinking Presumptions

The general architecture of I/O processing evolved in an era of memory scarcity, relatively primitive mass storage devices, limited processing power, and lower degrees of multiprogramming. The present environment is dominated by far larger memory capacity and far greater processing power at all levels, from hosts to physical devices.

Into the 1970s and 1980s, many systems lacked the memory resources needed to support more than a handful of simultaneously outstanding I/O requests. [10] Larger main memories eliminate that restriction. Present systems have sufficient memory to represent thousands of simultaneously pending I/O requests.

The conceptual issues remain unchanged, but the engineering tradeoffs made to ensure correct operation in the face of memory/processor scarcity can now be reexamined in the current resource context.

VII. Basic OS I/O Interface

User-mode programs are prohibited from directly accessing actual hardware. User programs request I/O services by invoking an operating system function that creates a summary of the request in system memory, e.g., buffer address, file number, starting block, and length, and then places that request summary on a queue for processing, Fig. 2.

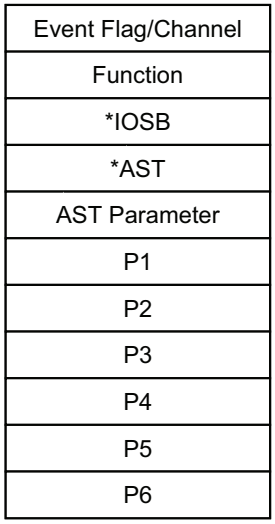

Fig. 2. The basic I/O API call transfers a byte range within a file to memory buffers, notifying the user-mode program upon completion. *Reprinted with permission from [21]*

When a range of blocks within a file is accessed, the memory conserving approach processes one contiguous segment at a time. While correct, such an approach sacrificies efficiency for low-footprint. Even though the higher levels of the I/O infrastructure are fully aware that additional segments will be

processed, serializing the requests denies this information to the device and controller. [24], [25]

Device drivers dequeue each request in turn and perform the processing needed to read/write the specified data from the mass storage device. Drivers may be cascaded to implement various aspects of the processing, e.g., a disk-class driver may do certain processing common to all block-addressable mass storage devices; followed by a device driver for a particular device model; which may in turn use a SCSI or other bus-focused device driver to actually access the communications channel connecting the host processor to the peripheral bus.

Some requests may require multiple sub-requests to accomplish an individual request.

Some systems classify the file system as a device driver, others use different terminology and structure. Naming is purely nomenclature; the responsibilities of the role remain.

When all of the relevant sub-operations complete the initiating request is completed and the next higher-level driver is notified. Eventually, when all of the required sub-processing is completed, the original program-issued request is completed with appropriate notification to the requestor.

VIII. Buffer Management

I/O requests are initiated by applications according to their internal logic. When standard language or system-supplied libraries are used to access files, it is common for underlying libraries to read-ahead or write-behind to optimize sequential reads and writes. Often, the buffer management strategy is to keep as many buffers full as possible.

Classic recommendations for double buffering are processor-centric. The focus of double buffering is prevent a pause in processor activity. However, it does not speak to transfer efficiency.

IX. Improvements

Exposing a single contiguous block range per file thread obscures the already-known future from lower I/O processing levels, e.g., the storage controller and device. Obscuring already inevitable requests cripples lower-level attempts to time-optimize operations. On the contrary, optimization is most feasible when lower-levels are aware of all pending sub-tasks of a particular operation. Optimization is enabled when related requests are delivered to the device closely spaced in time, within the same optimization window; rather than serialized as in the classic approach.

I/O specifications never specify the order in which requests transfer data between main memory buffers and peripheral devices; merely that the buffer contents are indeterminate from lower-level request issuance till the lower-level device signals request completion, successfully or not. [26]

Concentrating related requests in shorter time intervals ensures that:

- There is a higher probability that track and other caches will not experience churn from other streams contending for cache entries; and
- The lowest-level controller(s) will have a better view of the total request load, and be able to optimize for higher performance.

It is common for controllers and devices to optimize sequencing of outstanding requests, with the proviso that the requests are all in their respective queues simultaneously. If higher levels throttle the issuance of requests, the requests are not within the same optimization window, and cannot be reordered..

X. Operating System Actions

Optimizing mass storage performance and efficiency requires that the storage devices at the lowest levels of the hierarchy have the information needed to sequence operations in the most advantageous possible sequence. Put another way, higher levels from the I/O API to the actual physical device controller, must not take actions which obfuscate already known information about inevitable I/O requests from lower levels of the I/O infrastructure, Fig. 3. [27]

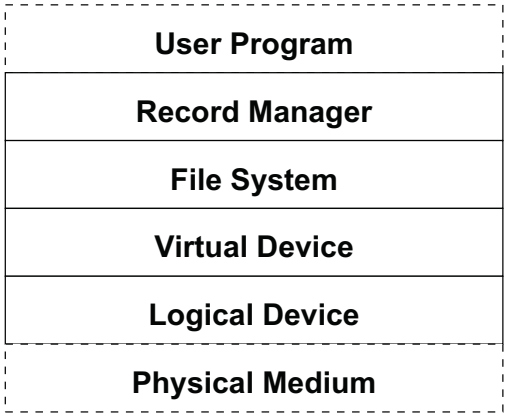

Fig. 3. I/O Infrastructure has nested software/hardware levels between a user-level request and the physical medium. *Reprinted with permission from [21]*

The operative principle is that intermediate level of the I/O libraries or infrastructure may not queue requests unless absolutely necessary. On the contrary, it is beneficial that all higher-level actions concentrate requests for a particular file into short, unordered, high-intensity bursts. Queueing at higher levels obscures information from lower levels and distances related requests in time, both of which foreclose lower-level optimization opportunities.

The classic approach can be altered in implementation without changing the user-visible API syntax or semantics, preserving both the vast pre-existing codebase and the skillset of the technical community.

The largest obstacle to optimizing low-level device workload scheduling is the presently truncated optimization horizon. The most correlated requests are requests to the same file coming from the same access stream, yet the upper levels of the I/O infrastructure presently serialize requests at multiple levels by enumerating $\mathcal{L}$ one element at a time.

while some condition **do**
  ...
  some processing
  ...
  System request for I/O
end while

Fig. 4. Write I/O System Call within inner loop. Limit checking omitted for clarity.

Removing unneeded serialization reduces or eliminates many structurally imposed limitations. [28]

When mapping within file requests to collections of logical device requests, it is advantageous to decompose a file relative request, v, file blocks 100-199, into l_n at once, creating an unordered set $\mathcal{L}$ of all the needed logical segments. Rather than insert each request individually into the pending queue, it is advantageous to first chain together the $l \in \mathcal{L}$, then atomically splice the resulting chain of sub-requests into the device queue in an indivisible operation.

Indivisibly splicing $\mathcal{L}$ into the queue solves the *first mover* problem. If the device queue is empty the first sub-request inserted potentially begins processing immediately. At that instant, the only sub-request in the queue is, by definition, the best choice for next operation. That choice may not be advantageous, depending upon the precise state of the hardware. Indivisibly splicing $\mathcal{L}$ into the queue ensures that all requests in that set are eligible for consideration before any are executed, eliminating the first mover problem.

Higher in the I/O stack, there are other highly localized changes that can increase the time density of related requests.

It not uncommon for programs, utilities, and libraries to go into a loop issuing I/O requests to a particular file, Fig. 4. The salient aspect of that loop is that it involves one system call per iteration. Preserving the API interface is paramount. However, there are many cases where the majority of paths using the system I/O interface API are within a system or third-party supplied library component. Replacing the pre-existing I/O API is infeasible; however culturally-appropriate enhancements are always possible.

Hoisting, the migration of iteration invariant computations from within loops, has been a feature of code optimization for decades. [29] *Hoisting* removes common sub-expressions from within inner loops. Transitions to/from kernel mode are relatively slow operations. Used properly, issuing tens or hundreds of I/O requests in the same system call, reduces overhead processing, seamlessly reducing multiple system calls to a single system call, Fig. 5. This eliminates context switches between I/O request system calls. As at the device driver level, this enhancement serves to increase the time density of related requests.

In this context, the existing I/O API can be extended with a new entry which allows the simultaneous issue of multiple I/O requests on the same file, with completion signaling for both the individual sub-requests and the request as a whole, Fig. 6.

Pending buffer list $\alpha \leftarrow \lambda$
while some condition **do**
  ...
  $\beta \leftarrow$ Buffer address, length, file offset
  ...
  $\alpha \leftarrow \alpha \cup \beta$
end while
Write all buffers $\beta \in \alpha$
$\alpha \leftarrow \lambda$

Fig. 5. Write I/O System Call hoisted outside of inner loop. Limit checking omitted for clarity

Simultaneous request queueing for discontiguous file ranges and buffer ranges, with individual sub-request completion signaling, allows user processes to proceed, as sub-requests complete.

Consider the case of sequential input. In the worst case, where the first buffers are delivered last, the resulting performance is no worse than the conventional case, e.g., buffers processed sequentially. If the first buffers become available earlier, then the user process can continue processing while later buffers are being received.

By definition, individual sequential output buffers are fungible. Once a buffer has been transferred to mass storage, the buffer can be released for other use. If the user process is stalled waiting for an available buffer, the process is able to continue processing. As with sequential reads, the worst case is the same as existing performance. The analysis for input buffers is similar, empty buffers are fungible.

Multiple discontiguous memory buffers combined with discontiguous file ranges are characteristic of databases and other programs. In that case, the originating thread is often building a list of buffers to be transferred between a file and memory. Each of the individual read/write operations are mutually independent. Individual read/write completion is potentially completely disconnected from the completion of other sub-requests. The overall completion offers the same overall completion as Linux `readv`/`writev` is used to signal the requestor that all sub-requests have been completed and any related aggretate data structures can be released, Fig. 6.

Singly issued I/O requests are either completed or uncompleted. Buffer contents are undefined for the duration of the I/O. Compound requests, e.g., System/360 channel programs, Linux `readv`/`writev`, and the multiply-issued requests described by this author are a completely different question.

The IBM System/360 I/O System architecture contained Program Controlled Interrupts (PCI), which reported execution progress of an I/O channel command sequence to the CPU. However, System/360 Channel Programs were in effect an SISD architecture for a particular device, serial command chains with simple conditional loops. The order of sub-requests was essentially fixed. There was no potential for reordering device operations to take advantage of physical-level opportunities.

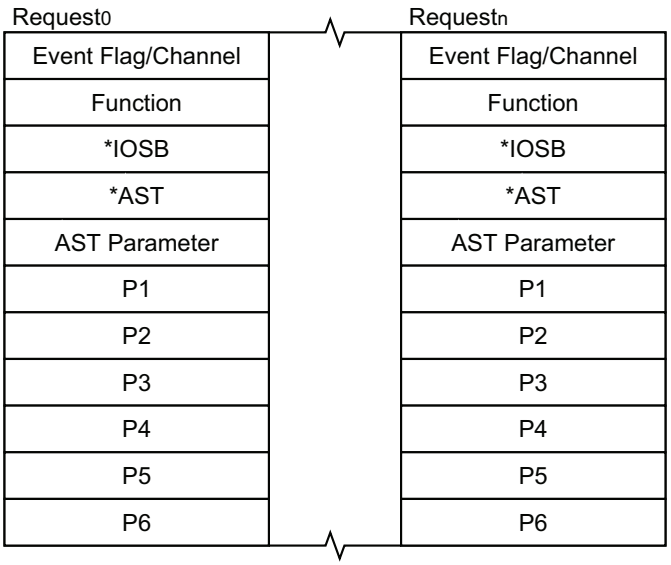

Fig. 6. A single system call which queues multiple, independent requests reduces aggregate overhead. *Reproduced with permission from [21]*

The Linux library has `readv` and `writev` calls, which transfer discontiguous memory areas to a contiguous area of a file. Completion is signaled when all transfers requested have completed. `readv`/`writev` does not have a mechanism to indicate completion of an individual buffer, nor does it have the capacity to transfer discontiguous buffers to discontiguous file regions.

The precise semantics of the multiple simultaneous issue I/O API extension is critical to improving performance and efficiency. The SISD limitation of System/360 channel programs limits the opportunity for re-ordering device requests to optimize storage device throughput. The complete/incomplete dichotomy provided by Linux `readv`/`writev` delays further processing until all buffers have been transferred, delaying potential CPU progess.

Simultaneous multi-request issue, combined with individual request completion, allows user programs to proceed with processing at an earlier point than would otherwise be possible, Fig. 7. Requesting transfers in multiple request blocks concentrates related requests in short bursts.

The differences between the approach described in this paper differ from the earlier approaches of IBM System/360 channel programs and Linux `readv`/`writev` by providing far more parallelism. This difference is summarized in Table II.

In this context, buffers are no longer about maintaining processor efficiency, but about maximizing both processor and peripheral performance. The goal is to ensure that there is always a buffer available for the processor and transfer requests are initiated to maximize device performance.

Previously, it was mentioned that how buffers are filled and processed affects performance.

When memory was scarce, it was common to transfer a single block at a time. More plentiful memory makes more extensive buffering physically possible, but leaves the question of when to request data transfers between mass storage devices and main memory to optimize performance.

TABLE II
COMPARISON OF SYSTEM/360 CHANNEL PROGRAMS, LINUX READV/WRITEV, AND MULTIREQUESTS [21]

	Parallel Execution	Individual Buffer Completion	Discontiguous Block Range within file
IBM System/360 Channel	No	Yes	Yes
Linux readv/writev	undefined	No	No
Multi-issue	Yes	Yes	Yes

The conventional approach used by systems supporting multiple buffers is to request transfers between mass storage and buffers at the earliest possible opportunity. If this strategy is employed over time, there is a steady stream of buffer requests as buffers are processed by the application. For an application with a dedicated volume, this can work well. If the application uses multiple large sequential files on the same volume, multiple outstanding requests will build up, depending on the delays in accessing the multiple files.

However, the most efficient sequence for information transfer is for requests to related physical storage device areas be simultaneously outstanding. This can be achieved by compressing the time interval between related requests. Rather than filling buffers as they become available, multiple transfers to the same file can be requested simultaneously, increasing the probability that multiple requests to closely related mass storage addresses will be processed in close time proximity. Simultaneity increases locality for both mass storage caches and media access operations.

Transferring buffers as they become available for transfer appears on the surface to optimize processor utilization, but does not create corresponding opportunities for transfer optimization by the storage subsystem and its components.

Transferring a single buffer at a time almost ensures that no two buffers from the same access stream are pending at the same time.

Deferring transfers until a number of buffers awaiting transfer can be enqueued as a group yields multiple outstanding requests for the same file, enabling storage system optimization.

XI. STORAGE CONTROLLER ACTIONS

At the storage controller and device levels, many of the issues that occur at the operating system level recur. It does little good to eliminate the first mover problem on the host, merely to re-encounter the issue at lower levels.

The grouping effect referred to at the user API-level applies throughout the layers down to the storage device. Grouping is preserved by creating a supra-individual request indicator. When a related series of requests are transmitted by a host, identifying the group as a whole serves this purpose. The relevant rule is that one cannot include members of a group

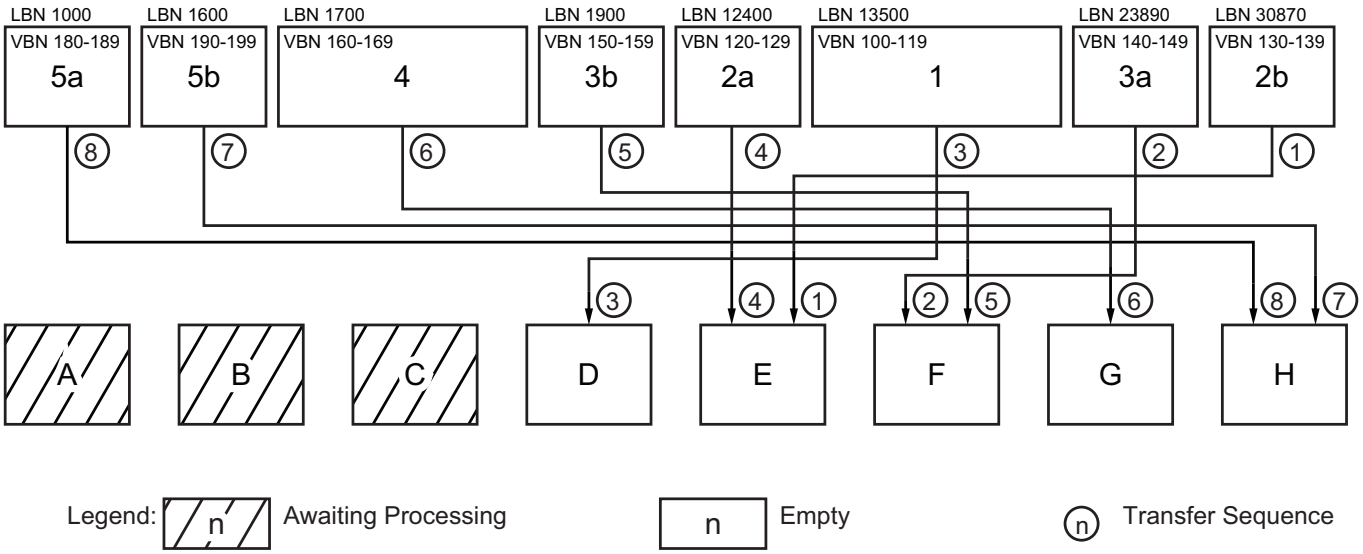

Fig. 7. Simultaneously initiated I/O requests individually complete, allowing processing to resume at the earliest possible time. Buffers A, B, and C are awaiting processing. Buffers D – H are awaiting input transfers. In this example, transfers are from high LBN to low LBN. The minimum time transfer sequence (indicated by the circled numbers) is in physical device space, which may be neither LBN order nor VBN order. *Reproduced from [30] with permission.*

in device scheduling until such time as the entire group is available for consideration. [21]

XII. Summary

The general architecture of operating system I/O infrastructure supporting random access mass storage was conceived in the 1960s and 1970s, an era of constrained:

- mass storage performance
- controller logic
- system memory

During that era, potential performance enhancements were eschewed in favor of minimum resource footprint. Correctness is obligatory; performance and efficiency are beneficial but far less important. The user-level APIs to access mass storage also incorporate the influence of these long-ago choices. Resource availability has dramatically increased since the tradeoff decisions. Memory and processing capacity in host systems, storage interconnects, storage controllers, and storage modules have undergone massive growth. However, a vast universe of software has been built to use existing I/O APIs. It is possible to modify the I/O infrastructure for vastly higher performance and efficiency while maintaining the existing user-level API and existing storage devices.

Acknowledgment

The author thanks Gideon Eisenstadter, Yoram Eisenstadter, and Paula Whitlock for reviewing drafts of this paper. Their comments and suggestions have been invaluable. I must also thank my Ph.D. committee and others who have assisted in my dissertation and the resulting monograph for their comments and suggestions.

References

[1] S. Iyer, "The effect of deceptive idleness on disk schedulers," Master's thesis, Rice University, Sep. 2001.
[2] ——, "Advanced memory management and disk scheduling techniques for general-purpose operating systems," Ph.D. dissertation, Rice University, Nov. 2005.
[3] R. Gezelter, *Re-Architecting Mass Storage Input/Output for Performance and Efficiency*, May 2018, ch. 3.
[4] IBM, "IBM 350 Disk Storage Unit."
[5] L. Stevens, "Data Storage Machine," US Patent 3,134,097, filed December 1954, patented May 1964.
[6] W. Goddard and J. Lynott, "Direct Access Magnetic Storage Device," US Patent 3,503,060, Dec. 1954, patented March 1970.
[7] Seagate, "Exos X20 Data Sheet," Nov. 2021.
[8] IBM, *System/360 Operating System, Field Engineering Handbook*. IBM Corporation, July 1971, S229-3169-3.
[9] ——, *z/OS Concepts*. IBM Corporation, 2010.
[10] Digital, "RSX-11 System Logic Manual, Volume 1," Technical Report, November 1978, order No. AA-5579A-TC.
[11] U. Valhalla, *Unix Internals: The New Frontier*. Prentice Hall, 1996.
[12] D. Solomon and M. Russinovich, *Inside Windows 2000, Third Edition*. Microsoft Press, 2000.
[13] P. Yosifovich, A. Ionescu, M. Russinovich, and D. Solomon, *Windows Internals, Seventh Edition, Part 1*. Microsoft Press, 2017.
[14] ——, *Windows Internals, Seventh Edition, Part 2*. Microsoft Press, 2017.
[15] J. Mauro and R. McDougall, *Solaris Internals Core Kernel Architecture*. Prentice Hall, 2001, ISBN: 0-13-022496-0.
[16] W. Mauerer, *Linux Kernel Architecture*. Wiley Publishing Inc., 2008.
[17] D. Knuth", *"The Art of Computer Programming: Sorting and Searching"*. Addison-Wesley Publishing Company, 1973, ch. 5, p. 1.
[18] S. Fuller, *Analysis of Drum and Disk Storage Units*. Springer-Verlag, 1975.
[19] R. Gezelter, *Re-Architecting Mass Storage Input/Output for Performance and Efficiency*, May 2018, ch. 13.
[20] ——, *Re-Architecting Mass Storage Input/Output for Performance and Efficiency*, May 2018, ch. 14.
[21] ——, "Re-Architecting Mass Storage Input/Output for Performance and Efficiency," Ph.D. dissertation, CUNY Graduate Center, May 2018.
[22] D. Patterson, G. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," EECS Department, University of California,

Berkeley, Tech. Rep., December 1987, Technical Report UCB/CSD–87–391.
[23] A. Goldstein, "Files-11 On-Disk Structure Specification (v.1)," Digital Equipment Corporation, Tech. Rep., 1975.
[24] R. Goldenberg and L. Kenah, *VAX/VMS Internals and Data Structures, Version 5.2*. Digital Press, 1991.
[25] R. Goldenberg and S. Saravanan, *OpenVMS AXP Internals and Data Structures, Version 1.5*. Digital Press, 1994.
[26] IBM, *IBM System/360 Principles of Operation*. IBM Corporation, 1968, pp. 103–104, GA22-6821-8.
[27] R. Gezelter, *Re-Architecting Mass Storage Input/Output for Performance and Efficiency*, May 2018.
[28] ——, *Re-Architecting Mass Storage Input/Output for Performance and Efficiency*, May 2018, ch. 3, 8.
[29] F. Allen and J. Cocke, *A Catalogue of Optimizing Transformations*. IBM, 1971, p. 17.
[30] R. Gezelter, *Re-Architecting Mass Storage Input/Output for Performance and Efficiency*. Erudite Aardvark, 2022.

Robert Gezelter (M '81 – SM '01) has over 45 years of experience in computing. His work focuses on operating systems, networks, input/output devices, algorithms, protocols, performance, security, and related areas. He has been in private practice since 1978, consulting to clients ranging from the Fortune 10 to small businesses, domestically and internationally.

He has spoken on topics relating to his work at many conferences throughout North America and overseas.

His articles have appeared in a variety of trade publications. His book chapters on computer security, systems architecture, and network architecture have appeared in several editions of the *Computer Security Handbook* as well as the *Handbook of Information Security*.

He received his Ph.D. (2018) in Computer Science from the CUNY Graduate Center. He received his M.S. (1983) and B.A. (1981) in Computer Science from New York University

The IEEE Computer Society appointed Dr. Gezelter to its Distinguished Visitor Program in 2004. He was named a Computer Society Charter Distinguished Contributor in 2021.

His offices are in Flushing, New York. He can be contacted via his firm's web site at http://www.rlgsc.com