Allocation and Data Placement Using Virtual Contiguity

Randal C. Burns, Robert M. Rees
Storage Systems Software
IBM Almaden Research Center

Zachary N. J. Peterson[†], Darrell D. E. Long
Department of Computer Science
University of California, Santa Cruz

Abstract

We describe an allocation and data placement technique called *virtual contiguity* that keeps the blocks of a file "near" each other so that a file system can read many blocks of a file at the same time with a single disk head movement. This technique avoids the disk seek penalties incurred when reading discontiguous block allocations, without requiring allocations to be strictly contiguous. At the same time, virtual contiguity provides for the fine-grained allocation and writing of data required to support memory-mapped I/O and efficiently perform copy-on-write for file systems that snap-shot data. Preliminary experimental results show that virtual contiguity groups data effectively and reduces the number of disk seeks required to read a file.

1 Introduction

We present the concept of virtual contiguity for the allocation and placement of data in a file system and show how this concept can be used to improve read performance. In virtual contiguity, data from the same file are grouped together in regions of storage so that the file system reads a whole region in a single large I/O. By grouping many blocks of a file together and allocating these blocks densely in a region (many blocks from the same file in a small piece of storage), virtual contiguity reduces the number of seeks required to read a file even when data are not strictly contiguous. Blocks that are not part of the file are discarded.

Our original motivation for virtual contiguity came from the observation that a modern disk drive performs a single 256K read faster than two small reads separated by a seek [13]. For this reason, a file system should make every effort to perform few large I/O operations rather than many small ones. However, the need to organize and manipulate data at a fine granularity competes with our desire to perform large I/O.

Increasing the block size of a file system can be used to improve read performance for large files. The block size is the fundamental unit of I/O such that a file system does all data reads and writes in multiples of this size. The Tiger Shark file system uses blocks of 256K or larger [5] to achieve high performance for parallel applications and multimedia data. Large files are generally accessed sequentially, which reduces the effectiveness of caching. This makes read performance at the disk more important. By choosing large block sizes, Tiger Shark reduces the number of seeks required to read large files, allowing the file system to realize most of the potential bandwidth of a disk drive.

A drawback to this approach is that large blocks create internal fragmentation, the unused storage space contained in blocks allocated to a file, resulting in poor storage utilization. For small files, much of the storage in a large block is unused (consider a 1K file occupying a 256K block), and it is generally accepted that most files are small.

More importantly, large blocks can also result in poor caching performance. To support writes that come from virtual memory (through paging or memory-mapped I/O), a file system must either pin large amounts of data in the buffer pool (cache) or perform *read-modify-write*. The memory system manages its own cache and conducts operations on its boundaries – the page size – and cannot respect the file system block size. When the page size is smaller than the block size (Figure 1), the file system receives a write from the memory system at a page granularity, which it must write back into the file system block in its buffer pool before writing the block to disk. The file system either (1) retains (pins) blocks in its buffer pool when files are memory mapped or (2) upon receiving a write to a memory page the file system *reads* the corresponding block into its buffer pool, *modifies* the block with the page, and the *writes* the block back to disk. The first choice wastes space in the file system buffer pool and is not always possible because the buffer pool can be full of mapped data blocks. The second choice results in poor write performance analogous to the small-write problem of RAID [3].

[†]Supported in part by the National Science Foundation under Grant NSF CCR-0073509 and by the Institute for Scientific Computing Research at Lawrence Livermore National Laboratory.

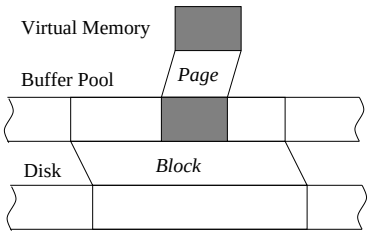

Figure 1: Writing pages with large blocks.

The natural choice for a file system designer is to choose the block size to be the same as the memory page size. Memory mapped files are cached only by the virtual memory system and reads and writes from the memory system are aligned for file I/O. In fact, a popular approach in modern file systems (NTFS [12] and IBM AIX's JFS) sets the file system block size equal to the page size and memory maps all files. In this way the virtual memory manager determines which blocks of a file are in the memory cache and the file system does not maintain an independent buffer pool. However, setting the file system block to the memory page size results in a small block and performance generally suffers.

FFS [10] and many similar file systems allocate the page-sized blocks of a file sequentially so that I/O may be performed on contiguous block ranges. While this approach has been very successful, it has two limitations. First, data must be allocated when they are first written and a file system can, in general, only keep data blocks contiguous when they are written at the same time. Also, many file systems write data out-of-place [6, 7, 8]; *i.e.*, when rewriting an existing block of data, the file system allocates a new disk location. Out-of-place writing is necessary for file system snap-shot and breaks up contiguous file ranges.

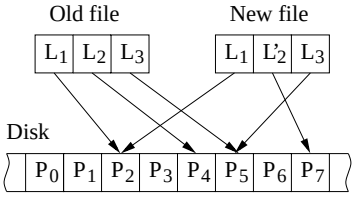

Figure 2: Copy-on-write example.

Snap-shot is an increasingly important feature in modern file systems, because it is necessary for point-in-time consistent backups [7, 8] and can be used to aid in fast recovery from failure [6]. Snap-shot requires a type of out-of-place writing called *copy-on-write* (COW) in which a new allocation is created when data are rewritten and the old allocation persists as part of a previous snapshot (Figure 2). Through COW, frequent snapshots destroy contiguity which decreases read performance. In our estimation, snap-shot is fundamental to data management and a requirement of a modern file system. Therefore, file systems must expect out-of-place writing and perform well under this workload.

Virtual contiguity provides a solution with some of the best properties of both large and small blocks. Page-sized data blocks are grouped so that reads can be performed on a large number of blocks at the same time. Also, the file system supports memory-mapped I/O without poor cache performance or read/modify write. In virtual contiguity, we allocate data sequentially when it is first written and then continue to place blocks near to the original allocation when they are re-written out-of-place. This keeps data close and maintains read performance even in the presence of COW.

We note that other file organizations [6, 15, 1] do not allocate data contiguously. Instead they choose to optimize writes and argue that caching takes care of read performance. These organizations are particularly efficient for writing both file meta-data and file data. In our target architecture [2], data and meta-data are stored on separate devices in order to separate the workloads. This approach improves meta-data write performance [11]. Thus, we are more concerned with the read performance of large sequential files than we are with meta-data. Because caching does not address read performance for large files, write-optimized allocation policies are not suitable for our file system.

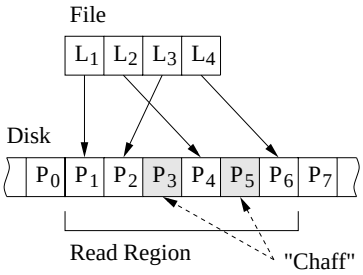

Figure 3: Reading virtually contiguous file blocks.

2 Virtually Contiguous Data Allocation

The key concept in virtual contiguity is that file blocks are placed near each other in storage, but not necessarily strictly contiguous, and can be read using a single head movement of a disk drive. In our example (Figure 3), we see a file that consists of four blocks placed non-contiguously in a region of storage. The blocks of the file are separated by several physical blocks (P_3 and P_5) that might belong to other files, might be empty, or might contain data from an old version (in the COW sense) of the same file. To read this file, the disk drive reads the physical range P_1 to P_6. Physical blocks P_3 and P_5 are discarded at either the disk drive or by the file system after being read. We read a range consisting of data and

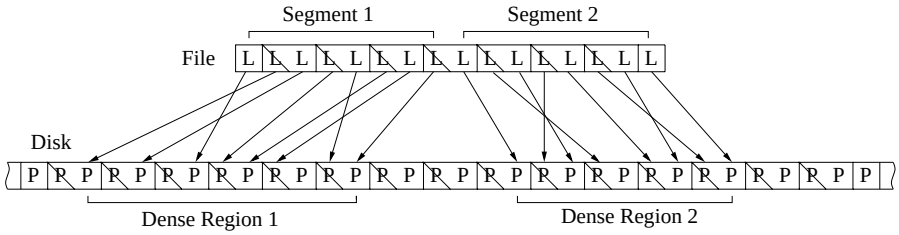

Figure 4: Large regions of virtual contiguity.

"chaff"[1] – blocks that are not part of the file – to perform a more efficient read.

The quality of a virtually contiguous allocation is best measured by the *density* of the useful blocks in a region; *i.e.*, the fraction of all blocks in the allocated region that contain data for the file to be read. The whole file need not be contained in a single region and may consist of multiple dense regions that will be read separately.

For our experiments, we have chosen to break logical file address space up into fixed size segments of 256K that are composed of 64, 4K blocks. We allocate the logical blocks of each segment into a dense region of physical storage (Figure 4) and read each dense region with a single I/O operation. Therefore, the number of disk seeks required to read a file is proportional only to the file size and is invariant to the manner in which the file is written. This policy for allocation is simple and deterministic and therefore easy to implement.

Many more sophisticated policies for allocating dense regions are possible (see Section 4). However, allocating fixed size dense regions also allows us to directly compare virtual contiguity to both small block allocation (4K) and large block allocation (256K), in which the segment is analogous to the large block. We have chosen a simple, static policy for allocation in order to achieve our goal of validating the concept of virtual contiguity.

2.1 Reallocation

Considered together, copy-on-write and virtual contiguity present a unique allocation problem. For traditional block allocation protocols, reallocating a block – creating new storage to COW an existing block – is no different than an original allocation. In contrast, when reallocating a block with virtual contiguity, we place the new allocation as close as possible to the original allocation.

The manner in which traditional dynamic storage allocation algorithms consume free space conflict with the goals of virtual contiguity and reallocation. For reallocation we desire free space near an original allocation in which to place the COW blocks. However, traditional algorithms like Next-Fit, First-Fit, Best-Fit, and Buddy system (for all algorithms refer to Knuth [9]) allocate free space on an empty disk from first block to last block sequentially, leaving no room for reallocations. Over time, as objects are deallocated, free space will appear. But, we would like to initially place objects in a manner that increases the likelihood that nearby free space is available for reallocation.

A randomized dynamic storage allocation improves the probability that a previous allocation has free space nearby. Rather than consuming space sequentially for initial allocations, we randomly (uniform over all space) select a start offset and begin searching forward from that start offset for a contiguous allocation. By randomizing the start offset, we attempt to keep the density of allocated blocks uniform across the disk. This algorithm does not systematically consume contiguous block offsets as do previous algorithms.

3 Experimental Results

In order to understand how virtual contiguity affects a file system, we conducted a trace-driven simulation of data placement and allocation. These preliminary experiments compare three different allocation policies: *small block* (SB) allocation, *large block* (LB) allocation, and virtually contiguous (VC) allocation. SB allocation uses a 4K file system block, a next-fit policy for finding free-space on the disk, and sequentially allocates data. SB emulates the allocation and data placement policies of FFS [10]. LB uses a 256K block, a next-fit policy for finding free-space, and does not allocate data sequentially. LB approximates allocation and data placement in the TigerShark file system [5]. Finally, VC uses a 4k file system block and forms dense regions of data on 256K segments. VC allocates the first write to a segment sequentially and finds disk space for this allocation by searching with randomized start offsets (Section 2.1). Subsequent allocations and out-of-place writing through COW to an existing segment are allocated as near as possible to existing data. We designed these experiments so that VC would be directly comparable to SB and LB. VC uses the same 4K block size as SB. The 256K segments are analogous to the blocks of LB, so that in keeping data dense VC approximates the read performance of LB allocation.

To drive our simulation, we used system call traces developed at the University of California [14]. These traces recorded every system call made by the HP-UX

[1] As in separate the wheat from the chaff.

operating system running on Hewlett-Packard 700 workstations over the period of about 3 months in 1997. From these traces we extract the logical block addresses of a file that are written over time. We assume that daily snapshots were taken (as needed to support daily backups). Therefore, out-of-place writing through COW occurs only if a trace contains writes to a file on multiple days. Data that are re-written on the same day are written back to the existing allocation.

Traces were taken over three distinct workloads: a group of four *instructional* machines used by undergraduates in a computer lab, a group of four *research* machines used by faculty and staff for research projects, and a single machine used a *web server* for an online library project. In terms of allocation, the instructional and research workloads are nearly identical, consisting of many small files with little COW. However, the web server trace contains an order of magnitude more actively written files than other workloads. These files are much larger (a handful of files larger than 1GB) and exhibit a high degree of re-written blocks requiring COW.

It is our assertion that virtual contiguity is beneficial in two distinct aspects of file system allocation. Like SB, it allocates data at a fine granularity, which results in minimal internal fragmentation and support of fine-grained I/O for memory-mapped files. At the same time it provides the read performance benefits afforded to us by LB allocation.

Workload	Small Block	VC	Large Block
Instructional	66.84%	66.84%	98.45%
Research	71.47%	71.47%	97.61%
Web Server	16.97%	16.97%	82.88%

Table 1: Internal fragmentation.

Fragmentation results, which measure the amount of unused space in partially filled disk blocks allocated to files, verify the benefits of fine-grained allocation. Table 1 looks at the percentage of internal fragmentation over the different allocation policies. Owing to the large number of samples, confidence intervals on all of our results are tight (smaller than the precision of the published number) and therefore not included. Because most files are small, almost all space allocated to files in LB is unused. Even a file of 1 byte consumes a 256K block.[2] Virtual contiguity shares the same block size as FFS, therefore the internal fragmentation incurred on the system is the same. The markedly different results from the web server trace reflect a larger average file size.

We use two metrics, number of seeks and file system density, to evaluate read performance. Experiments are preliminary because we do not attempt to measure or simulate [4] the performance characteristics of a disk drive. Rather, we rely on the following expression to describe the time to read a file:

$$\sum_{k \in K} \left(\sigma_k + \frac{|k|}{\mu \rho_k} \right). \quad (1)$$

A file consist of K regions each of size $|k|$, density ρ_k, and time to seek to the region σ_k. The read rate of the disk drive is μ. We contend that this equation is dominated by σ_k and that by reducing the number of seeks, the time to read a file decreases. We include density in Equation 1 to make it general enough to describe VC in addition to SB and LB, which have unit density.

Workload	All Files	COWed Files	COWed Segs:
Instructional	97.00%	91.10%	88.42%
Research	99.59%	91.82%	90.96%
Web Server	99.70%	96.73%	86.35%

Table 2: Density of a virtual contiguous region.

In terms of read performance, VC differs from LB only by having lower density and therefore taking slightly longer to read data. Table 2 contains density results over the three workloads. The column titled *All Files* describes the average density over all files in the trace. *COWed Files* are files that have been re-written out-of-place and *COWed Segs* is the density of only the segments of files that have been re-written. Results show that VC keeps data densely allocated on disk even in the presence of COW.

Workload	FFS	FFS (COW)	VC	VC (COW)
Instructional	1.51	7.85	1.07	1.21
Research	1.11	4.63	1.07	1.27
Web Server	1.94	68.26	1.88	1.86

Table 3: Average number of seeks.

The main advantage of VC lies in reducing the number of seeks needed to read a file as compared to SB. Table 3 presents the average number of seeks to read a file over various file system allocation policies. Results compare the average number of seeks over all files and those files that have been re-written using COW for both VC and SB. Note, by definition LB and VC have the same number of seeks. VC consistently outperforms SB in number of seeks and therefore in overall read performance. VC can be as much as 36 times better, in the case of the web server, when reading files on which copy-on-write has been performed. In SB, files become non-contiguous in two ways: (1) copy-on-write and (2) files are written through multiple write commands issued separately. VC improves performance in both cases through

[2] Tiger Shark reduces internal fragmentation by fragmenting blocks so that many files share a single disk block.

re-allocation of data near existing allocations. While SB performs reasonably in comparison to VC when averaged over all files, it is important to perform well on every file. In particular, files that have been COWed are files that have been accessed on more than one day, providing some indication of their importance.

4 Discussion and Directions

We present preliminary results on a rudimentary implementation of virtual contiguity that indicate the promise of this technique. Our findings can be furthered both in expanding our work on data placement and allocation and by improving our experimental techniques.

In our evaluation of performance, we disregard the physical properties of disk drives. Note that seek time (σ_k in Equation 1) is a complex variable that depends on placement of data, position of the head, and device properties. Integrating our experiments with a disk-drive simulator [4] will allows us to accurately model seek time.

Because our simple, static allocation policy has some shortcomings, we are actively exploring other policies that adapt to the current layout of data on disk and integrate disk drive parameters (like track size and seek profile). The fixed size segments that we have described can result in arbitrarily bad allocations when the disk nears capacity. For example, when re-writing a segment, the allocator will search as far as necessary to find the nearest free blocks, which can result in very low densities. We are currently working on an adaptive algorithm that allows for segments to grow very large as long as they remain dense and also allow for segments to be divided up when re-allocation would result in low density.

5 Conclusions

Experimental results verify our claim that virtual contiguity captures the benefits of both small and large block allocation, while avoiding the detrimental features of each. For all workloads examined, the advantages of virtual contiguity can be clearly seen. We reallocate blocks "near" to the original allocation, producing dense regions of related blocks that can be read using a single disk I/O. At the same time, internal fragmentation of data is low. Virtual contiguity supports page granularity copy-on-write, allowing a file system to take snap-shots without degrading read performance. It is our belief that the addition of virtually contiguous allocation and data placement to a file system will provide significant improvements for I/O performance, particularly for large files that are read sequentially and files that exhibit copy-on-write.

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM TOCS*, 14(1), February 1996.

[2] R. C. Burns. *Data management in a distributed file system for Storage Area networks*. PhD thesis, University of California at Santa Cruz, 2000.

[3] A. L. Drapeau, K. W. Shirriff, J. H. Hartman, E. L. Miller, S. Seshan, R. H. Katz, and D. A. Patterson. RAID-II: A high-bandwidth network file server. In *Proceedings of the 21st Int'l Symposium on Computer Architecture*, 1994.

[4] G. R. Ganger. *System-Oriented Evaluation of I/O Subsystem Performance*. PhD thesis, University of Michigan, Ann Arbor, 1995.

[5] R. L. Haskin. Tiger shark – A scalable file system for multimedia. *IBM Journal of Research and Development*, 42(2), 1998.

[6] D. Hitz, J. Lau, and M. Malcom. File system design for an NFS file server appliance. In *Proceedings of the USENIX San Francisco 1994 Winter Conference*, January 1994.

[7] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM TOCS*, 6(1), February 1988.

[8] M. L. Kazar, B. W. Leverett, O. T. Anderson, V. Apostolides, B. A. Bottos, S. Chutani, C. F. Everhart, W. A. Mason, S. Tu, and R. Zayas. DEcorum file system architectural overview. In *Proceedings of the Summer USENIX Conference*, June 1990.

[9] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison Wesley Longman, 3 edition, 1998.

[10] M.K. McKusick, W.N. Joy, J. Leffler, and R.S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3), August 1984.

[11] K. Muller and J. Pasquale. A high performance multi-structured file system design. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991.

[12] R. Nagar. *Windows–NT File System Internls: A Developer's Guide*. O'Reilly and Associates, September 1997.

[13] J. Palmer. Private correspondence: The performance properties of modern disk drives. IBM Almaden Research Center, 1999.

[14] D. Roselli and T. E. Anderson. Characteristics of file system workloads. Reserach report, University of California, Berkeley, June 1996.

[15] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1), February 1992.