Optimizing File Accesses via Ordering and Caching

Jan Wassenberg

7 April 2006

Abstract

Slow I/O is widespread, as attested to by splash screens and progress bars; however, it can be done better. A reusable and highly efficient I/O solution is presented; design decisions and key algorithms are discussed, and the resulting performance analyzed.

Contents

1 Introduction

1.1 Motivation / Importance of Fast I/O

Since I/O is much slower than CPU or memory, it can easily become a bottleneck within the system as a whole. An estimate as of 2006 is 60 MB/s vs. 2600 MB/s. Many applications would therefore benefit from faster I/O; example scenarios include:

- slow startup time. The user is inconvenienced by waiting for required files to load; splash screens are one attempt to mitigate this by distracting the user. For a rather extreme illustration of the problem, see [6].
- on-demand loading. If the data set is too large to fit in memory, it must be loaded in increments as needed. This can cause 'freezes' in the application while waiting for the I/O to finish.
- heavy throughput requirements. Some applications, e.g. video players or editing tools, require high sustained I/O throughput.

1.2 Intended Application

The application for which our I/O library has been developed is a Real-Time Strategy computer game [1]. It utilizes the traditional method of loading files on startup as well as on-demand streaming of data, so both should be efficiently handled. While intending for the I/O solution to remain useful for a wide range of applications, several consequences arise from this and guide our design decisions.

First, much emphasis is placed on real-time behavior. Lag or 'freezing' in-game is not acceptable and must be minimized. This means that the caching algorithm must not have offline performance characteristics, reordering I/Os is probably not acceptable and any pre-fetching would have to be quite conservative (so as not to penalize time-critical on-demand loads).

Also, the working set is not static; depending on game mode and environment, different files may be needed. Provision must be made for varying access patterns.

Finally, and related to the real-time issue, is that of fragmentation. Games can run over several hours; during that time, performance must not degrade to unacceptable levels e.g. due to memory fragmentation. Given the real-time requirements, offline reorganization is not an option; the algorithms used must be designed accordingly.

Given these central design constraints, we now present the chief ideas behind our fast I/O method.

1.3 Techniques

Our approach is five-fold:

1. caching avoids repetitive slow I/Os;
2. ordering files according to access patterns minimizes hard-drive seeks;

3. compressing files reduces the amount of data to be read;

4. asynchronous I/O maximizes throughput and allows computation to proceed in parallel with I/O;

5. splitting I/Os into blocks simplifies caching and decompression while also avoiding copying buffers for alignment purposes.

We will discuss each of these in detail in Section 3, but first cover related theoretical work in this field.

2 Related Theoretical Work

2.1 Cache

For the cache, a central question is which files to keep in memory. This is known as the file- or web-caching problem. In short, given a set of file requests (each with size and retrieval cost), a cache is maintained such that total retrieval cost is minimized.

The special case where size and cost are uniform is called "paging", which has been studied extensively. Several algorithms that have an optimal competitive ratio are known. In particular LRU[1] is $k/(k-h+1)$ competitive, which is the best possible for a deterministic algorithm [17, page 207].

This model is appealing due to its simplicity, but is not sufficient for our needs. Files are not typically uniform size, and treating them as such would be monstrously inefficient (much cache space would be wasted by rounding up element size to that of the largest file).

Irani gives two $O(\log^2 k)$ competitive randomized algorithms that can deal with variable-sized files and uniform cost [13].

However, we would like to achieve full generality and provide for variable cost as well. This can be used as the name suggests to more accurately reflect load time (as will be seen later, this is not solely dependent on file size!), or as a hint from the application that certain files are not to be removed from the cache as early as they otherwise would.

Young develops such an algorithm and calls it Landlord. Briefly, each file receives 'credit' that is initially set to its cost. When determining which file is to be removed from cache (i.e. 'evicted'), each one is charged 'rent' proportional to its size and the minimum credit-per-size density currently in the cache. Items are evicted once their credit is exhausted. On every access, credit is increased in an arbitrary manner. This strategy is $k/(k-h+1)$-competitive, which again is optimal for a deterministic algorithm [19, 17].

We end up using an optimized variant of this Landlord cache management strategy.

2.2 Allocation

Another important part of caching is the memory allocation aspect. For reasons that will be discussed in Section 3.5.1, existing general-purposes allocators are not adequate;

[1] Least Recently Used; simply evicts the file whose access time is the least recent.

an alternative will have to be developed. We build on decades of work in this area.

Wilson et al. give a very thorough and helpful overview [18]. A simple but crucial point is made: fragmentation is caused by freeing regions whose neighbors are not free. Allocators are online algorithms whose only tool against this is placement, i.e. deciding where to allocate regions. The authors advocate benchmarking by means of traces (a record of allocations) from real-world programs, because randomized tests do not necessarily reflect reality. It is emphasized that allocation policy and mechanism must be considered separately. Results of tests show certain policies, namely address-ordered first (segregated) fit, to perform quite well, wasting only about 14 % memory. Finally, further discussion of implementation details such as boundary tags was helpful.

Johnstone and Wilson go on to refine their measure of fragmentation and conclude that the previously mentioned AO-first-fit policy actually only suffers from ca. 1 % fragmentation, the best of all techniques considered [14, page 10]. This promising result leads us to focus on that policy.

Masmano et al. present a "Two Level Segregated Fit" algorithm with $O(1)$ time complexity [15].

We end up implementing a simpler variant based on this idea that also avoids the need for block headers, which was the abovementioned problem preventing use of a general allocator.

2.3 Ordering - Traveling Salesman Problem

The problem of ordering files according to access patterns can be seen as an instance of the Traveling Salesman Problem. It is defined as: given a graph of nodes (cities) and the cost of traveling from one to another (travel distance), compute a path that will take the salesman to each city while incurring minimal cost. In our case, files correspond to cities and the hard-disk seek distance to cost.

TSP has perhaps been studied most among all optimization problems; numerous algorithms and heuristics have been developed, each with their strengths and weaknesses. The DIMACS Challenge [4] gives an extensive listing of algorithms, relative performance and techniques and was a valuable reference.

For our application, less than optimal orderings are acceptable due to non-static access patterns. Since variational file accesses (e.g. due to differing modes of play) would invalidate any ordering we establish, it does not make sense to insist on an optimal solution. The DIMACS Challenge shows several heuristics to perform quite well, coming to within 11 % of the Held-Karp bound (a good approximation of the optimal solution to an instance of TSP [16]). We therefore settle on a greedy heuristic for simplicity.

3 Detailed Discussion of Techniques

We now cover the individual techniques used to speed up I/O in detail.

3.1 Efficient Asynchronous I/O

For an understanding of how to achieve maximum I/O read throughput, we briefly explain how the disk is accessed on PC systems.

Early drives were addressed via Programmed I/O, where the CPU instructs it to transfer 2 bytes at a time. Due to significant per-transfer overhead (accessing I/O registers and interrupting CPU when complete), throughput only reached a maximum of 16.7 MB/s (PIO Mode 4) [5].

Once rising disk platter densities — and the resulting increased transfer speeds — caused this to become a bottleneck, bus-mastering DMA (Direct Memory Access) over the PCI bus became the norm. Here, the disk controller writes directly to memory, bypassing the CPU. It is free to perform other work during this time, so long as the bus is not needed - an important point that will affect our choice of I/O block size (see Section 3.4).

Given this information, we now examine the I/O interfaces provided by the Operating System. POSIX supports synchronous blocking I/O, blocking I/O in another thread, and asynchronous I/O ("aio"). We remove the first option from consideration because it does not allow work to proceed in parallel with the I/O. Several implementation details cause us to choose aio over the threaded approach:

- on Windows, aio bypasses the OS file cache. This allows bulk DMA transfers, which achieve higher throughput than the single page-in operations that would be issued by the memory-mapping-based OS file cache.
- aio places pending read requests in a queue so that the disk controller can proceed immediately with the next I/O; the disk is always busy. With threaded blocking I/O, the OS would have to return from and then re-enter kernel mode before relaying the application's next I/O request to the disk. This overhead reduces throughput.
- parallelism between computation and I/O is achieved without having to worry about the OS correctly scheduling all participating threads. Additionally, behavior is predictable and thread-switch overhead is avoided.

In short, a decent aio implementation[2] should not fare worse than the threaded blocking I/O approach. On Windows, it is in fact faster due to the abovementioned issues.

As a final detail, the POSIX aio functionality is emulated on Windows in terms of the "overlapped" `ReadFile` API. By using the POSIX interface, we ensure portability to virtually all systems.

To summarize, we use asynchronous I/O to achieve best possible throughput and allow computation to proceed in parallel. This is made possible by the hard drive's DMA capability. The validity of this approach is shown by a small test program that reaches maximum rated drive throughput and by [11].

[2] Linux used to emulate aio by spawning threads, which made for less than stellar performance. However, this is no longer the case.

3.2 Compression

The next cornerstone of our I/O library is compressing source files. This can dramatically reduce the amount of data to be read. Indeed the current 0 A.D. dataset has been compressed down to 46% of the original, a savings of 75 MB. (NB: the dataset includes 13 MB of uncompressible audio; 3d mesh files with compression ratios of around 3x are chiefly responsible for the reduction)

The compression algorithm used is Deflate, a combination of LZ77 and Huffman encoding as defined in [8] and used in the common Zip file format [10]. Other formats may achieve better compression ratios or feature faster compression/decompression speed, but these are not critical to success. We prefer the advantage of interoperability — tools to work with Zip archives are universally available.

In addition to the abovementioned significant reduction in file size, a further compelling argument to compress all data files is that it is effectively free! To show this, we must first discuss how exactly I/O and decompression will be parallelized.

Presuppose that I/Os are split into fixed-size blocks, the rationale of which will be explained in Section 3.4. These blocks are issued asynchronously up to a safe queue depth (currently 4). A block whose I/O has finished is then decompressed while the next ones are pending. This gives perfect parallelization if decompression requires less time than I/O.

Indeed a benchmark shows that a typical Pentium IV system (as of 2002) manages 40 MB/s I/O throughput and 100 MB/s decompression [7]. Note: The balance is not expected to change in the future for single-disk systems; even if it does, a compression method more suited to real-time decompression can be substituted.

Therefore, any reduction in file size due to compression lessens I/O time at no cost.

3.3 Ordering Files

The techniques so far are not yet sufficient. They achieve good sequential read performance, but overall throughput is quite poor because files will tend to be scattered throughout the disk. This incurs expensive seeks (moving the hard-disk read head); a rough estimation of their cost is the time taken to read 400 KB (assuming typical 7200 RPM drive with 10 ms seek and 40 MB/s throughput [9]). Given that files are often much smaller on average (25 KB for 0 A.D.), seek time dwarfs pure I/O read time.

Throughput can be much improved by arranging files on disk in order of access, thus avoiding seeks. Since we wish to use a standard filesystem for simplicity, but cannot control its placement strategy, files will have to be combined into one large OS-visible archive. As mentioned above, we prefer the Zip format for easy interoperability.

Incidentally, storing files in archives has an additional advantage. The FS needs to store metadata and typically sector-aligns files; since sectors are 512 bytes or more, this is very costly for tiny files[3]. In contrast, archives can contain files packed end-to-end with only minimal metadata/header information, thus wasting less space and by extension reducing read time.

It remains to determine the optimal file ordering that minimizes seeks. This will be done once (offline); performance is therefore not of paramount importance.

[3] ReiserFS4 is the only known exception, able to pack several files into one sector.

Before, though, we decide whether files may be repeated in the archive. To see the problem, consider the following sequence where file ‘C’ is loaded after ‘A’ 50% of the time and otherwise after ‘B’: ACBCACBC. It would seem that 50% of ‘C’ accesses must incur a seek, but placing two copies of this file in the archive — after ‘A’ and ‘B’ — could avoid them entirely. However, practical considerations lead us to disallow this: the act of finding a file within the archive would be a good deal more complicated.

Now back to the issue of finding an ordering for files. Our strategy is as follows:

1. view all files to be added as nodes in a DAG (Directed Acyclic Graph); edges indicate that two files are immediate neighbors in the archive.
2. record a “trace” of all file accesses over one or more program runs (recall that access patterns may differ between runs).
3. construct from this a list of possible edges sorted by their frequency (i.e. how often they occurred in the trace).
4. generate a set of ‘chains’ by committing the above edges as long as no cycle results. These chains are connected portions of the DAG that are known to have been accessed in that order.
5. output the final file ordering by stitching together all chains and then adding any remaining files that were not included in the trace.

Details on these steps follow.

3.3.1 Prepare DAG of Files

Each node holds all required information about the file. This includes its filename and the nodes that have been chosen to come before and after it in the final layout. All of these are stored as 16-bit IDs to reduce size and therefore improve locality; mapping from filename to ID is accomplished in logarithmic time via tree.

3.3.2 Record Trace

The acts of loading a file and releasing the resulting memory are logged (the latter is required by the file cache). Records consist of timestamp, filename, file size and any flags that affect I/O mode. For simplicity, we do not record file offset or transfer size: that would not yield any information because seeks are incurred by accessing any part of the file. Also, we assume that loading entire files at a time is the dominant behavior.

Besides the obvious application of determining optimal archive ordering, the resulting plain text file can be used to benchmark the I/O implementation under repeatable conditions. Even when lacking the actual data files, the trace can still be useful to benchmark performance of the file cache and ordering. For this, simply map filenames to an integral ID and simulate the cache and I/O parts.

Notes:

- we are careful to ensure that recording a trace does not incur any I/Os, which would skew performance measurements. Records are stored in binary format within an expandable array (no copying or memory waste due to pre-reserved virtual address space).
- trace files may log accesses over several program runs. This will be useful in the following steps because several mutual-exclusive but equally probably access patterns may exist, each of which should be equally considered. Program runs are differentiated by examining the timestamp, which starts at 0 on each run.

3.3.3 Construct Edge List

This step constructs a list of edges from the trace file. First, the trace is split into program runs, which are processed most recent first. In each of these, all adjacent pairs of files are examined; those not already in the list are added, otherwise the existing edge's frequency is incremented.

Important note: presuppose the existence of a file cache, which will be presented in the next section. Since frequent accesses to files will be absorbed by this cache, we do not want this inflated frequency to 'pollute' the edge list. That would displace other edges that might actually turn out to be more important because they actually would incur seeks, as opposed to the edge whose file I/Os would be satisfied by the cache. Our solution to this problem is to simulate the file cache whilst processing trace entries (only in the same program run!); if the file would not result in an I/O due to the cache, the current edge is ignored. Under the assumption that access patterns are similar to the trace, this scheme improves the quality of the ordering by making it reflect the trace more strongly (rather than being fooled by frequent I/Os). If not, correctness is not impacted; we merely risk incurring a few more seeks.

Checking if an edge already exists is accomplished by translating the two filenames into 16-bit IDs ($O(\log N)$ time), appending these into a 32-bit number and searching for that in a tree ($O(\log N)$ time).

Finally, this list is sorted by decreasing frequency[4]. The result is a list of unique edges (i.e. "file A should be stored after file B" relationships).

3.3.4 Generate Chains

This step is the heart of our file ordering strategy. The above edges are now 'committed' into the DAG in order. That means the files are marked to come after one another, i.e. their nodes in the DAG will be connected by an edge (unless a cycle were to result). For simplicity, committed edges are never removed, this being a greedy heuristic.

We check for cycles via "DFS", which actually simplifies to a list walk here since nodes have only one previous and next link. These are typically quite short and overall run time of this entire step is not a problem in practice (7 ms for 5000 files), so we do not

[4] The sort must be stable, i.e. preserving ordering of edges with identical frequency! This will become clear in the next step.

attempt more efficient and sophisticated cycle detection schemes. One such approach would be to store a pointer to the current end of list for each node and perform list jumping.

The result of this step is a set of disjoint chains, which are each a series of files that are to be stored immediately after one another. Due to the nature of the edge list, the files that are most frequently accessed after one another are grouped together. As such, we have attained a good approximation of an optimal tour.

Note: now the reason for the most-recent-first program run ordering becomes apparent. All but the most frequent edges are placed into the list in the order that they occurred in the trace (due to stable sort). Since they are also committed in the DAG in this order, they end up mostly as observed from the trace. Since the most recent trace is assumed to be the most accurate and reflective of current behavior, it is given the most weight (by allowing all edges that ensued from it to be committed first).

3.3.5 Stitch Chains together

The final step is to stitch together the disjoint chains and output them into the final ordered list. File nodes will be marked once they have been output. We iterate over all nodes and output the entire chain of which it is a part; this is done by following the node's previous link until at beginning of the chain. Incidentally, this iteration ensures all files appear in the output list, even if they were not included in the trace.

We have thus generated an ordering of files that minimize seeks assuming application behavior is similar to that which was recorded in the trace(s).

This is an approximation to a variant of the Traveling Salesman Problem; the question as to its quality (i.e. how many seeks are avoided) is interesting and will be examined in Section 4.3.

Rough complexity analysis: except for the cycle determination, none of these steps require more than $O(\log N)$ work per file. Expected case is therefore $O(N \log N)$, with $O(N^2)$ work in the worst case (if DFS always scans through very long chains). However, as mentioned above, this is an offline process; performance is entirely adequate, so we do not delve into a complete analysis or optimize the cycle determination step.

3.4 Splitting Into Blocks

Splitting I/Os into fixed-sized blocks is desirable for two reasons:

It would allow decompression of large files to proceed immediately and in parallel with the I/O. This is especially important when loading an alternating sequence of large and small files: all decompression can be 'hidden' behind I/O. The alternative, namely only decompressing after having finished loading the entire file, clearly breaks down in this case and does not parallelize well.

One further advantage is that of sector alignment. Due to the end-to-end packing in archives, files often start at unaligned offsets on disk. A limitation in the Windows `ReadFile` API would require copying such files to/from an align buffer. This can be avoided by splitting I/Os into blocks and rounding their offset/size down/up to sector boundaries.

We now decide on the block size. Many considerations come in to play:

+ theoretically, larger sizes are good due to economy of scale (less overhead per transfer).

+ block length should be a multiple of the sector size (required for sector alignment mentioned above).

− blocks should not be too large, or else decompression cannot be done in-cache. That would result in bus accesses, which interfere with the DMA I/O operation. Typical L2 cache sizes are 256 to 512KiB, which must cover the compressed source and decompressed destination buffers.

− I/Os for large blocks may end up being split into several I/O requests; beyond that point, there would be no advantage to increasing the block size. Background: PC DMA requires *physically* contiguous memory, which cannot be guaranteed from user programs because they only see virtual addresses. As a workaround, the OS typically analyzes the buffer and makes a "scatter-gather list" out of it. This is a list of contiguous regions (typically only one memory page due to fragmentation) that constitute the buffer; the driver can DMA into them individually without having to copy from a central DMA buffer. For concreteness, the Windows ASPI layer has a limit of 64 KiB per transfer because its scatter-gather lists are stored in non-paged pool, a memory region of limited size [3, page 6].

− in practice, there is no difference between aio read throughput for transfer sizes between 4 and 192 KiB [12].

+ However, the aio queue depth (maximum number of concurrent I/Os that can be queued by the OS) is system-dependent and should not be relied upon. Therefore, it is better to avoid all too small blocks, because it may not be possible to queue enough buffers to keep the disk continuously busy.

The result of these ruminations was a block size of 16 KiB. However, our measurements have shown 32 KiB to be most efficient (see Section 2).

This concludes discussion of our I/O techniques. To review, I/Os are automatically split into blocks (of aligned start position and length) and issued asynchronously. Once a block finishes, it is decompressed while the next block I/O is in progress. Finally, seeks are avoided by having arranged the files within an archive in order of access.

3.5 Caching

> It's not true that life is one damn thing after another; it is one damn thing over and over. *Edna St. Vincent Millay*

The final step we take in optimizing I/O is caching. By keeping commonly used files in memory, some repeated I/Os can be avoided outright.

There are two 'levels' of cache: entire files and blocks.

The small block cache serves to avoid overhead due to sector-aligning I/Os in transfers. Since files usually start at unaligned offsets within archives, data lying at the

beginning of a sector would be read twice (once for the real I/O and then again during the next file's I/O). The block cache absorbs this cost by keeping in memory the last few blocks read; it is organized as LRU.

The per-file caching strategy is due to the assumption that files will usually be loaded in one burst; it simplifies bookkeeping and avoids having to copy pieces of the file into a final buffer. Our file cache is a system consisting of the following components:

- an allocator doles out variable-sized chunks of a fixed-size memory region.
- the 'extant list' keeps track of which buffers are currently in use by the application.
- a cache manager provides efficient lookup of the file contents given filename and decides which files to keep in memory.

We now explain these in detail.

3.5.1 Allocator

A general-purpose allocator (e.g. malloc) is not acceptable for this application because file buffer addresses are required by Windows `ReadFile` to be aligned to a sector boundary. Rounding up returned addresses would waste unacceptable amounts of memory, so a special allocation scheme is needed that always returns aligned regions.

This entails not prefixing the allocated regions with a header. Our idea is to transfer ownership of an allocated region from the allocator to cache and/or extant list; these have to record region address and size anyway for their bookkeeping. When the region is to be freed, the extant list informs the allocator of its size and address, which is typically what a header would have stored.

Having now established the requirement for alignment and how to ensure it, we discuss the main problem of an allocator: fragmentation. There are basically two ways to deal with this: perform periodic reorganization, or prevent it from happening in the first place.

The former is not feasible due to our real-time requirements, and — more importantly — because users receive direct pointers to the cache memory. This allows zero-copy I/O and reduces memory footprint because multiple users of a file can share its (read-only) contents. However, it is believed that currently in-use and therefore unmovable regions would severely hamper defragmentation. We therefore focus on the latter approach.

With all pieces in places, we now discuss the allocation policy. As mentioned in Section 2.2, Address-Ordered good-fit performs well. When freeing, we coalesce regions immediately. This may perform unnecessary work, but is acceptable in light of its simplicity. Allocation first exhausts all available memory before reusing freelist entries. This is fine because the cache size is chosen such that it can and should be used in its entirety. The benefit is reducing freelist splitting, which tends to produce larger coalesced regions.

Note: in addition to policy, there is another approach to mitigating fragmentation. Its root cause is freeing objects whose neighbors are not free. We attack this by allowing for the application to pass hints as to buffer lifetimes, so that long-lived objects can be placed differently and not cause 'holes' around freed short-lived objects.

Implementation Details A 'good' fit is achieved by searching in segregated freelists. They are divided into size classes, where class $i >= 0$ holds regions of size $(2^{i-1}, 2^i]$. Determining size class can be done by taking the base-2 logarithm of the size. If a freelist is empty, the allocation can be satisfied by finding the next highest non-empty class ($O(1)$ due to bit scan) and splitting its first block.

Total allocation performance can be made $O(1)$ by further splitting size classes into fixed-size subclasses; this is the approach taken by [15]. However, we find that freelists are typically empty anyway (because the cache is always as full as possible) and therefore omit this for simplicity.

Coalescing works by storing boundary tags within the freed (!) memory. When freeing a block, we check if the regions that come before and after it have such tags (identified via distinctive bit patterns very likely to occur in normal data); if so, they are merged. Note that this is somewhat risky but the 'magic' bit pattern is long enough to make any mix-up extremely unlikely. This trouble is necessary because the tags cannot be added to the beginning/end of a region due to alignment requirements.

For convenience, memory is doled out from a fixed-size chunk of virtual address space, rather than separate on-demand allocations from the OS. This allows easily checking whether a given pointer is valid and was taken from the chunk. Due to on-demand committing of the virtual memory, only as much physical memory as necessary is used.

3.5.2 Extant List

This list tracks all buffers that have been handed out to the application but not yet freed. Since they are expected to be freed immediately (before allocating the next, which is enforced by a warning), this list only contains a few entries and therefore need not be organized as a tree.

It stores address and size of the allocated regions, which are passed to the allocator when freeing a buffer. This avoids the need for per-region headers, as explained above. An alternative would be providing a separate data structure associating allocated address with its size, but this is redundant since many of these regions are also stored in the cache. Therefore, our approach uses less memory.

3.5.3 Cache Manager

The cache manager is the heart of this system; it maps filenames to the file's cached contents and decides which ones to keep in memory. As mentioned in Section 2.1, we use the Landlord algorithm for this purpose. See Figure 1 for pseudocode describing its operation.

The quoted steps can be adapted as desired to yield various other strategies that Landlord generalizes (e.g. FWF, LRU).

We see that the naïve version of this algorithm has a high memory access cost: `makeRoomFor` involves two complete loops over all cached items (lines 9 and 11).

The first step towards mitigating this is to optimize the manager's item container (used to implement the filename-to-cached-file mapping) for good locality. An array-

Figure 1: Landlord Pseudocode

```
 1 def access(item):
 2   if cache.contains(item):
 3     "increase item.credit"
 4   else:
 5     makeRoomFor(item)
 6     cache.add(item)
 7
 8 def makeRoomFor(item):
 9   minCreditDensity = min{cached item i} (i.credit / i.size)
10   while cache.remainingSize < item.size:
11     for_each i in cache: i.credit -= minCreditDensity * i.size
12     "remove any subset of items that have credit 0"
```

based hash table will perform much better than a tree whose elements are scattered throughout memory.

We have developed several further improvements:

1. The costly divisions required to calculate credit density can be replaced with multiplying by the reciprocal. This trades less latency (4 vs. 20 cycles on Athlon XP CPUs [2]) for increased memory use.

2. the two loops can be fused by calculating the next MCD (Minimum Credit Density) value on the side. We therefore avoid iterating over all items twice, which is especially important for large sets of items that do not fit in cache.

3. a priority queue can return and remove the MCD item in $O(\log N)$ time; the rent that should be charged from all items can be accumulated and applied in batches.

   The validity of this approach is not immediately clear. Landlord specifies decreasing all credit by $MCD \cdot item.size$ and removing any subset of items with no credit remaining. By definition of MCD, at least one item will be removed, and this is exactly the one returned by the priority queue.

   Note that any pending charges must be committed before adding any items; otherwise, they too would be charged during the next commit cycle, which would be incorrect.

   Implementation note: to avoid duplicating code, the priority queue is separate from the filename-to-cached-file mapping. Since it is ordered by the item credit, the queue must be re-sorted after an item is accessed, which increases its credit. Due to limitations in the STL `priority_queue`, this takes $O(N)$ time on every access. Since cache hits are fairly rare, time is still saved overall; however, this bottleneck should be removed by substituting a heap implementation that allows a $O(\log N)$ "sift" operation.

These improvements are made available as template policy classes and can therefore easily be enabled for applications where they provide a benefit.

We examine results of these optimizations in Section 4.5.

This concludes discussion of the cache. To recap, the small block cache absorbs the cost of rounding I/Os up to block size boundaries. A file cache managed by the Landlord algorithm caches the contents of entire files.

4 Experimental Results

4.1 System Information

The test system has the following specifications:

CPU	Athlon XP 2400+ (2000 MHz)
Memory	768 MB DDR 2100 CL2.5
Chipset	NForce2
HD	Deskstar 7K250 (160 GB, PATA, 8 MB cache)[5]
OS	Windows XP SP2
Compiler	MS Visual C++ 7.1 (optimization flags "/Oxgb1y /G6")

We now describe methodology and show results of several tests measuring performance of our I/O library.

4.2 IO Throughput

4.2.1 Methodology

The basis for our I/O throughput measurement is a trace file recorded from the startup of 0 A.D. encompassing ca. 500 file loads. Using the trace simulation feature described above, we issue these I/Os as quickly as possible; this removes the influence of other system-specific conditions such as graphics card performance etc.

What is actually measured is the total amount of time elapsed between start and end of I/Os; this together with the amount of user data transferred yields effective throughput ("effective" because it differs from the actual disk throughput due to compression).

This was chosen as the benchmark measure because it reflects real-world performance of the entire system.

Note: if a cache is involved, we ensure it is empty so as not to skew results; in the case of the OS file cache, testing takes place after a clean reboot.

4.2.2 Results and Discussion

We are interested in the total improvement yielded by our I/O library, as compared to throughput reached by the bare OS-provided `read` API. According to the above measure, we see 29.3 MB/s vs. 2.96 MB/s, a staggering speedup of 990 %! We now examine which I/O techniques are chiefly responsible for these gains:

Leaving everything else the same but no longer compressing files stored in archives, performance falls from 27.2 MB/s to 22.2 MB/s[6]. We are led to conclude that disk throughput is a limiting factor; a good sign indicating seeks are not the bottleneck. Section 4.3 will examine this in more detail.

As an aside, decompression performance indeed mirrors the previously quoted 100 MB/s figure; we observe 94.5 MB/s.

When archives are disabled entirely and I/O is from loose files (stored in the normal filesystem), performance drops to 2.62 MB/s. The immediate conclusion is that reduced locality (due to poor FS ordering and extra headers) induces many costly seeks. We also notice that performance is worse than that measured for the synchronous API; this could be explained by increased overhead of the aio APIs. Indeed, they do not support the Windows FastIO driver entry points that avoid needing to build an I/O request packet.

Finally, we revisit the question of file block size. The initial choice of 16 KiB was not optimal; based on the results given in Section 2, we choose a size of 32 KiB.

It is interesting that performance begins to falls off starting with 64 KiB blocks. An explanation might be that transfers are split up due to the previously mentioned scatter-gather list limit, but this is speculation.

In summary, we have found that bundling files into archives is the most worthwhile improvement, due to reducing seeks. Once these are eliminated, the increased throughput afforded by the (free) data compression step contributes an additional 23 % speedup.

4.3 Ordering Quality

The above result indirectly shows that storing files in archives manages to avoid numerous seeks; else throughput would not be so high. We now examine exactly how many are incurred, thus evaluating the quality of the archive ordering and its TSP heuristic.

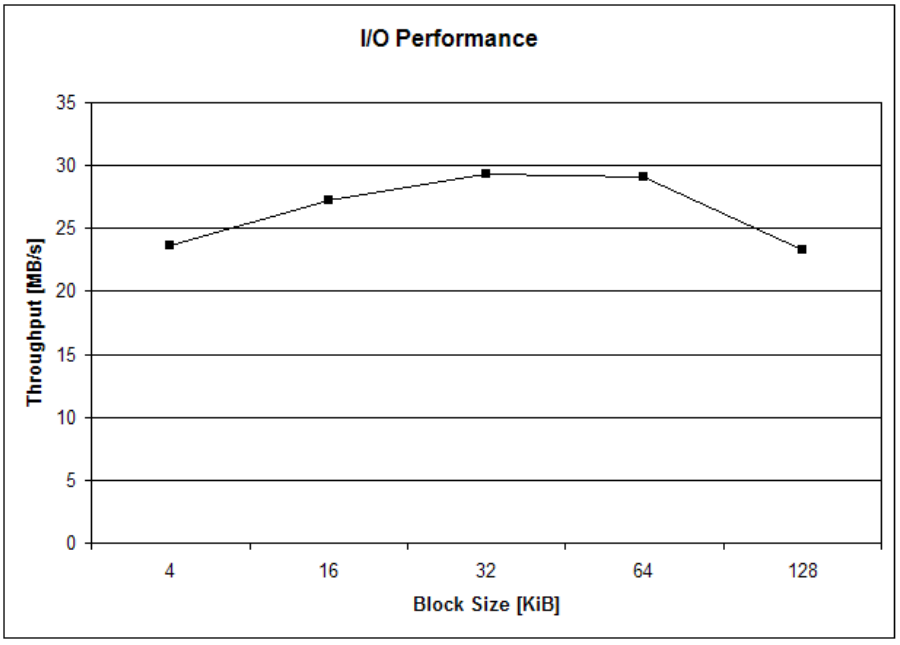
To measure total seek impact, we must first define their cost. Short seeks may actually be free because the HD controller has already read the target data into its cache. Also, long seeks may be more expensive due to physical limitations of the disk head (it must accelerate/decelerate to/from maximum velocity and then settle on the target track).

A good model for this would be a constant overhead plus cost proportional to the seek distance, plus rotational latency. However, this is quite disk-dependent and difficult to determine. For simplicity, we currently assume uniform cost and try to avoid all seeks.

Our first step in measuring them is to record a trace of three different 0 A.D. startup sequences, each loading a separate map (which share some files but differ in others, e.g. environment textures). This large trace consisting of some 2300 loads is used to guide creation of an archive. We then count the seeks incurred by each of the individual sequences - this is easily done in our code by comparing current I/O file and offset with the last known values.

[6] This measure differs from the peak performance listed above in that file block size was not yet the optimal value.

Figure 2: Effect of Block Size on I/O Throughput



Block Size [KiB]	Throughput [MB/s]
4	23.7
16	27.2
32	29.3
64	29.1
128	23.3

For the combined trace, no seeks are observed[7]. This is as expected because the archive was specifically ordered for that sequence.

The individual "Cantabrian Highlands" and "Neareastern" map sequences incur only 49 (9.4 % of total I/O requests) and 60 (10.6 %) seeks, respectively. It may come as a surprise that subsequences of the trace incur seeks, while the whole does not. The explanation is that our file cache also serves to avoid seeks[8].

These positive results justify our decision to use a heuristic to approximate TSP. Because the access patterns induced by separate maps differ widely, insisting on an optimal ordering for one particular pattern does not make sense. Instead, this heuristic produces good results for a variety of maps.

4.4 Caching Efficacy

We now appraise the effectiveness of the cache replacement policy, i.e. its tendency to keep files in memory that will be needed later. To measure this, we simulate cache operation over the combined trace mentioned above. It comprises 57 MB of data, of which 14 are repeated and therefore potentially cacheable.

Since the 0 A.D. dataset is as yet relatively small (real-world cache sizes may well be larger), we have artificially limited the cache size to ensure that items will have to be evicted from the cache. Without this action, the cache replacement policy would be irrelevant. A size of 10 MB has been chosen arbitrarily; its impact on cache performance will be studied below.

We first evaluate the well-known LRU algorithm under these conditions. The cache hit rate is determined to be 19 % (473 hits totaling 6.18 MB vs. 1915 misses totaling 51.22 MB). Our Landlord implementation more than doubles this to 39 % (945 hits totaling 8.88 MB vs. 1443 misses totaling 48.52 MB). A more intuitive view of these numbers is that the percentage of non-compulsory misses (i.e. files that were evicted but needed later) drops from 26 % to 2 %.

We are pleasantly surprised by this favorable result. Since our implementation does not yet take advantage of file cost hints from the application, the difference in performance is due solely to the Landlord algorithm's awareness of item size. This apparently leads to more efficient handling of the cache memory: fewer files need be evicted to make enough room for the next item. Another factor is that the repeated files in this trace are spaced widely apart (e.g. at the start of each of the three map loads constituting the trace); LRU would tend to remove exactly these items.

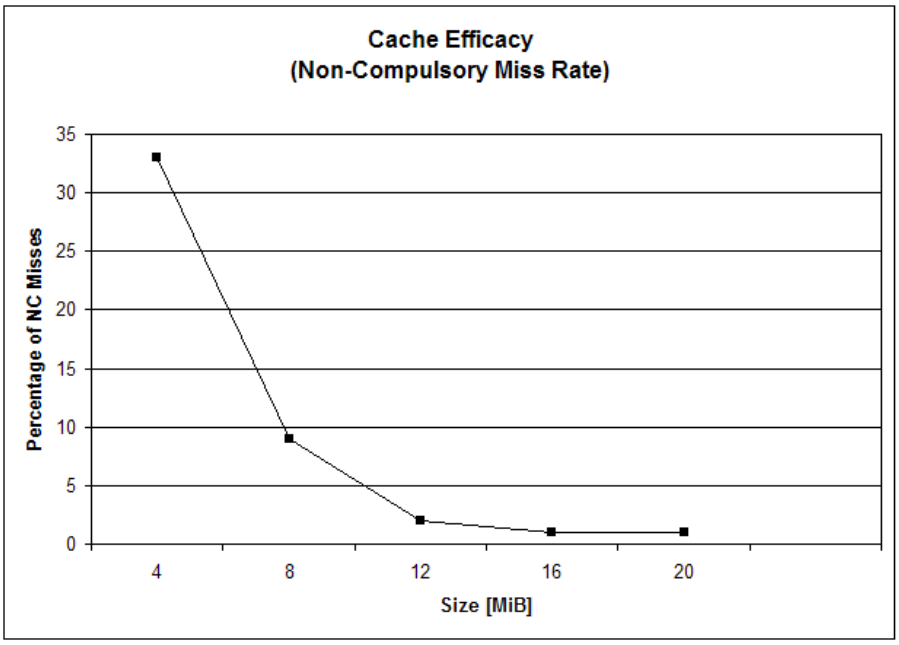
4.4.1 Effect of Cache Size

One question that remains is whether our arbitrarily chosen cache limit affects the results in an untoward fashion. To gauge this, we repeat the above measurements for

[7] We have excluded 4 unavoidable seeks that are unrelated to the archive ordering, namely 1 during the course of opening the Zip file (reading ECDR and then Central Directory) and 3 due to files that cannot be added to an archive.

[8] To see this, consider the following simple example: trace = ABACAD, optimal ordering = ABCD. For the entire trace, the file cache will absorb the latter two A accesses, whereas the subsequences AC and AD each incur a seek.

Figure 3: Effect of Cache Size Restriction



various cache sizes; the percentage of non-compulsory misses reflects the influence of the cache size (see Figure 3).

The result is as expected: non-compulsory misses decrease as the cache size limit approaches the size of the potentially cacheable data. The arbitrarily chosen 10 MiB is a good compromise: it forces some data to be evicted from the cache, but is not such a severe constraint that the cache is of no benefit.

One further result of this cache limit is a change in the number of seeks measured. Due to the interdependency mentioned in Section 4.3, seeks increase to 14 % of all files when the 10 MiB limit is in place.

4.5 Cache Manager Optimizations

Of further theoretical and practical interest is how much improvement the various Landlord algorithm optimizations yield.

Accounting CPU cost is done as follows. First, external influences are minimized by running at highest scheduler priority. Several thousand iterations of the target code are run while measuring elapsed time via high-resolution timer (precise to 1 CPU clock!). Each of these iterations performs an operation (e.g. allocate or free) chosen randomly; this avoids measuring characteristics that are specific to a given trace. Note, however, that we control the random distribution (in the example, ratio of “allocate” to “free” operations); these are weighted towards the most frequent and important operations.

The first result is that with the naïve Landlord implementation, dividing via multiplying

by reciprocal is actually 1.4 % slower! This is likely because the additional storage required for the reciprocal breaks the nice cache-friendly 16 byte element size. Since this algorithm iterates over all items twice, the memory access cost weighs more heavily than a few extra CPU cycles spent dividing.

Next, we find that the `Landlord_Cached` strategy (recall that it calculates minimum credit density while updating and therefore often avoids needing to iterate over all items) performs 21 % faster. However, its divide-via-reciprocal variant is again slower - this time by 0.6 %. We see that iterating less often increases the benefit from the reciprocal divider.

The final variant is `Landlord_Lazy` (which uses a priority queue to find the least valuable item in $O(\log N)$ and thus avoids iterating over all items when wanting to remove one from the cache). It performs 19 % better than baseline — slightly slower than the previous variant. Note that this result is heavily dependent on the relative frequency of cache add and remove operations: since the former require iteration over all items (to 'commit' a previous pending charge), decreasing their number from the current (and quite arbitrary) 70 % will cause this implementation to come out far ahead. Applying the reciprocal divider results in further gains of 0.8 %. Since we rarely iterate over all items here, the increase in size is outweighed by the faster division.

To conclude this section, we find that `Landlord_Cached` performs best in the current benchmark. Since it is less complex and requires less memory than the possibly faster `Landlord_Lazy` strategy, it is chosen as the default. However, the implementation via template policy classes allows easily switching strategies in applications where results differ.

4.6 Allocator Fragmentation

The important question of allocator fragmentation is next. We gauge it in the course of simulating the previous 500-file trace. A simple and adequate measure is to compare the total requested size with how much of the total file cache is actually occupied. The result is a total memory waste of 12 %, which is in line with the findings of [14]. While not great, this is acceptable.

5 Conclusion

Waiting for slow I/O is the bane of many a computer user; we have shown that this need not be and can be mitigated to a large degree.

A method for fast I/O has been presented and analyzed. The main contribution is a combination of techniques that greatly improves effective I/O throughput. By caching file contents, we can avoid repetitive I/Os; placing files in archives arranged in order of access reduces costly seeks. Asynchronous access maximizes read throughput and (together with block-splitting) allows the data to be compressed, which reduces the amount that must be read. The end result is a measured speedup of nearly 1000 % in the target application, which is expected to apply widely due to inefficient filesystems.

Of further interest are optimizations made to the memory allocation and cache management algorithms. They respectively allow returning aligned file buffers (required

by the aio implementation) without serious fragmentation and reduce CPU cost of the cache manager by 20 %.

Other applications can build on our work and easily speed up their load times and file accesses.

5.1 Implementation

Our I/O code has been developed in C++ and also contains a few time-critical assembly language subroutines. It encompasses ca. 12 KLOC[9], about 60 % of which is new; the rest was built upon previous work. Unfortunately there are dependencies on another 30 KLOC, so releasing and integrating into other applications is not as easy as it could be; this is being worked upon. Eventually releasing the code under the GNU General Public License (Free Software) is planned.

5.2 Lessons Learned

> Experience is what you get when you don't get what you want.
> *Dan Stanford*

Despite application of Software Engineering best-practices such as careful modularization, built-in self tests and pre/postcondition guards, the defect rate was unpleasantly high. This may be the norm for low-level C++ codebases of the above size, but was a problem given that file loading is a critical part of the application. The takeaway is that more self-tests and condition checking can be recommended unreservedly - they exposed many bugs.

The trace functionality (recording all I/Os) was found to be quite valuable. Besides its immediate application of ordering files, it allows testing I/O performance under repeatable conditions and reproducing bugs.

On a final positive note, much room for improvement was to be had with I/O! The gains achieved were surprising.

5.3 Future Directions

We have further ideas for improvement that could not yet be implemented due to time constraints.

Prefetching, i.e. reading data before it is needed (during idle time), shows promise. While requiring more work and tighter integration with the application, this can improve performance by always keeping the hard disk busy. The downsides that must be mitigated are increased power usage and potentially interfering with time-critical I/Os.

Currently, the main bit of 'intelligence' is offline and consists of finding a good ordering for files within an archive. We would like to bring more of this into real time and e.g. make decisions in the file cache based on predicted future behavior. In particular, small files known to be accessed after one another could be removed from the file cache

[9] Kilo Lines Of Code.

together, thus freeing up more space (meaning less fragmentation) without hurting performance (because one file not in cache will force reading the block in which it is stored, anyway).

Two approaches are envisaged that could realize these wishes. A Markov chain could be constructed and used to decide the probability of certain I/Os coming after one another. Also, previous traces could be examined at runtime to determine where in the load sequence we are, thus predicting further I/Os.

Stay tuned!

References

[1] 0 A.D. http://www.wildfiregames.com/0ad.

[2] *AMD Athlon Processor x86 Code Optimization Guide*. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22007.pdf.

[3] *ASPI Frequently Asked Questions*. http://www.eetkorea.com/ARTICLES/2000APR/2000APR05_CT_ID_AN.PDF.

[4] DIMACS TSP challenge. http://public.research.att.com/~dsj/chtsp/.

[5] *IDE/ATA Transfer Modes and Protocols*. http://www.pcguide.com/ref/hdd/if/ide/modes_PIO.htm.

[6] Least Patient Kid Ever. http://www.break.com/index/patiencechild.html.

[7] Ludicrous Speed Ahead. http://archive.gamespy.com/hardware/june02/p45331/index2.shtm.

[8] *RFC 1951*. http://rfc.net/rfc1951.html.

[9] Storagereview.com's Drive Performance Resource Center. http://www.storagereview.com.

[10] *Zip Application Note*. http://www.pkware.com/business_and_developers/developer/appnote/.

[11] J. Gray, E. Riedel, and C. van Ingen. A Study of Windows NT Sequential IO Performance. Technical Report P117, Microsoft Research (MSR), September 1997. http://research.microsoft.com/barc/Sequential_IO/default.htm.

[12] J. Gray, B. Worthington, and R. Horst. Windows 2000 Disk IO Performance, June 2000. http://research.microsoft.com/~gray/papers/Win2K_IO_MSTR_2000_55.pdf.

[13] S. Irani. Page Replacement with Multi-Size Pages and Applications to Web Caching. *Algorithmica*, 33, 2002.

[14] M. Johnstone and P. Wilson. The Memory Fragmentation Problem: Solved? *ACM SIGPLAN Notices*, 34, 1999.

[15] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A New Dynamic Memory Allocator for Real-Time Systems. In *Euromicro Conference on Real-Time Systems*, pages 79–86, 2004.

[16] D. Shmoys and D. Williamson. Analyzing the Held-Karp TSP Bound: a Monotonicity Property with Application. *Information Processing Letters*, 37:281–285, 1991.

[17] D. Sleator and R. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28, 1985.

[18] P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of the International Symposium/Workshop on Memory Management*, pages 1–116, 1995.

[19] N. Young. On-Line File Caching. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 82–86, 1998.