Towards Mass Storage Systems with Object Granularity

Koen Holtman
CERN – EP division
CH - 1211 Geneva 23, Switzerland
Koen.Holtman@cern.ch

Peter van der Stok
Eindhoven University of Technology
Postbus 513, 5600 MB Eindhoven,
The Netherlands
wsstok@win.tue.nl

Ian Willers
CERN – EP division
CH - 1211 Geneva 23, Switzerland
Ian.Willers@cern.ch

Abstract

Many applications, that need mass storage, manipulate data sets with KB – MB size objects. In contrast, mass storage devices work most efficiently for the storage and transfer of large files in the MB – GB range. Reflecting these device characteristics, mass storage systems typically have a file level granularity. To overcome the impedance mismatch between small objects and large files, we propose a move towards mass storage systems with object granularity. With an object granularity system, the application programmer stores and retrieves objects rather than files. The system internally maps and re-maps these objects into files. The system can adapt to changing object access patterns by re-mapping objects. This allows the application to be more efficient than if it were built on top of a traditional file granularity mass storage system, employing a fixed mapping of objects to files.

In this paper we report on investigations on the potential benefits of object granularity systems. We present an architecture that incorporates solutions to the scalability and fragmentation problems associated with object granularity.

1 Introduction

For some applications, the application dataset is so large that data storage on tape is an economic necessity. Examples where datasets can be in the Terabyte scale are high energy physics data analysis and satellite image analysis. Such applications can be built on top of a mass storage system, which controls data movement between tape storage and a disk farm that serves both as a staging pool and as a cache, this disk farm is called the *disk cache* below.

Tape drives work efficiently if the data on them are accessed in terms of MB – GB size files. Reflecting these hardware characteristics, mass storage systems generally have a file granularity, with the expectation of managing large files. Conversely, in many mass storage applications, the application-level data consist of objects with sizes in the 1 KB – 1 MB range. The application designer must map the application-level objects to files on tape, with every file containing many objects. This mapping is usually done when the mass storage system is being filled, and no re-mapping is done over the lifetime of the dataset. Though such a fixed mapping to large files allows the mass storage system to function efficiently, it can cause application-level inefficiencies. The inefficiency will be especially high if the application often needs a small subset of the objects in a file.

To overcome the impedance mismatch between small application level objects and the large files desired on tape, we propose a move towards mass storage systems with object granularity, that hide the underlying files.

In a mass storage system with object granularity, the application programmer stores and retrieves objects rather than files. Caching and migration inside the system are also object-based. The system internally maps and re-maps objects to files. By re-mapping objects, the system can adapt to changing application-level object access patterns. This allows the application to be more efficient than if it were built on top of a mass storage system with file granularity, employing a fixed mapping of objects to files.

While the potential benefits of an object granularity system are clear, so are its potential problems. The size of the indexing and scheduling tasks associated with managing objects, rather than files, will be some orders of magnitude larger. Also, there is an obvious danger of data fragmentation on tape and in the disk cache.

In this paper we report on investigations on the potential benefits of object granularity systems. We present an architecture that incorporates solutions to the scalability and fragmentation problems associated with object granularity. This architecture incorporates a commercial object database, Objectivity/DB [1] and a traditional file granularity mass storage system (for example HPSS). By using these standard components, the implementation cost of our object granularity mass storage system is kept low.

We show that object granularity systems outperform file granularity systems for applications in which the following conditions are met.

1. **Sparse access condition**: The application data access patterns have to be so diverse or unpredictable that a fixed mapping of objects to files will lead to inefficiencies. We quantify this condition as follows. Take the initial, fixed mapping to files as created (and optimised) by the application designer. Any query will 'hit' a certain number of files in this initial set. Now consider the objects, in these hit files, that are actually needed by the query. These objects should make up 30% or less of all objects in the files, on average, for the sparse access condition to hold.
2. **Repetitive access condition**: The application data access patterns should also be such that object (sub)sets selected at the application level are read not once, but a few times over a period of time.

Our work was driven by the problem of Petabyte-scale data analysis in the next-generation high energy physics experiments at CERN (see for example [2]). This is one application area where the above two conditions hold.

2 Overview of the architecture

We developed an architecture for an object granularity mass storage system that contains solutions to the scalability and fragmentation problems mentioned above. This architecture should be seen as an existence proof for a system with object granularity. Systems which employ different solutions to the object granularity problems may also be feasible. We have investigated some alternatives, but do not claim to have surveyed all possible solutions.

2.1 Software components

Our work is part of a larger research project, aimed at exploring database technology options for the storage and analysis of massive high energy physics datasets [3]. Our architecture is based on software solutions being pursued in this project [4]. We use the Objectivity/DB object database product [1], which is interfaced [5] to a generic file granularity mass storage system, like HPSS. We develop new software components that 1) add an abstraction layer, which provides object granularity mass storage, on top of the object database, and 2) control file movement between disk and tape, and the management (remapping, deletion) of data in the disk cache. In line with the work in [5], HPSS only acts as a file stager, its disk pool management functions are not used. As such, the choice for HPSS as a software component is not critical, and it could be interchanged with another file granularity mass storage system.

2.2 Filling the system with objects

Our object granularity mass storage system provides an ‘append only’ storage model, in which new objects can be added at any time, but in which objects become read-only once added. The application programmer fills the mass storage system by supplying *chunks* to it. A chunk is a set of objects (typical size 10 MB - 10 GB), which is initially mapped, by the system, to a single file on tape. This chunk model gives the application programmer a degree of control over the initial mapping to files on tape that is similar to that found with a traditional file granularity mass storage system. We found that retaining such control is important. Object re-mapping can in principle compensate fully for a bad or random initial mapping of objects to files. But performing such a re-mapping will take significant system resources. It is better to save these resources in advance by allowing the application programmer to encode advance knowledge about access patterns into the chunks.

The mapping of an object *to a chunk* is retained throughout the lifetime of the object. During this lifetime, the object can be (re)mapped to many different files.

2.3 Object addressing

Once stored, an object is uniquely identified by its *chunk identifier* and its *sequence number* inside the chunk. Sequence numbers run from 1 to n for a chunk with n objects, and reflect the object storage order inside the original chunk file, which was determined by the application programmer. Figure 1 shows a visual example of object addressing.

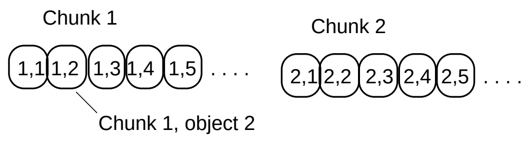

Figure 1: Example of logical object addressing

Our system maps the objects in chunks to physical files. Figure 2 shows an example of such a mapping. The system allows many files to be present for any chunk. Every file present for a chunk holds a subset of the objects in that chunk. In a running system, the number of files per chunk typically ranges from 1 to 20, depending on the access patterns to the objects in the chunk. The subsets of objects held by the different files may overlap, and generally do overlap, so that some objects in a chunk are physically present in multiple files.

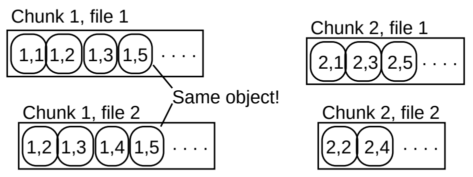

Figure 2: Example of physical mapping of objects to files

Files never mix objects from two or more chunks. This strict chunk-level separation makes indexing and scheduling problems much more manageable. The system does not maintain a single global index for looking up in what files an object is contained. Instead, there is a file-level index for each file, which can be used by sub-queries to read objects from the file, and by schedulers to quickly determine the exact set of objects in a file. All file-level indices are kept on secondary storage.

Our system does not require that the above 'files' are actual files managed by a filesystem. In our prototyping efforts using the software components described in section 2.1, the 'files' are actually (sets of) ODMG containers in the Objectivity/DB database.

2.4 Object access

The application programmer can access stored objects by executing a *query* against the store. A query specifies a set S of object identifiers, with the intention that all these objects must be visited, and a *query function*, which is a piece of executable C++ code (typically a loadable shared library). To execute the query, the system first computes the set C of all chunks that contain one or more objects identified in S. For every chunk, the system runs a *sub-query* over the objects in this chunk. This sub-query iterates through all objects in S which are in its chunk. For every object, the programmer-supplied query function is called. The query function is handed the object identifier and a reference to an in-memory copy of the object. The application programmer can optionally supply code that is to be executed at the start and the end of the query and of any sub-query.

Iteration by a sub-query always happens in the order of the sequence numbering of the objects in the chunk, this order was determined by the application programmer when the chunk was added to the system. The fixed iteration order allows the system to ensure fast data access and to prevent fragmentation.

The scheduling of sub-queries is outside the control of the application programmer: this is done by the system to ensure that the sub-query is synchronised with any necessary file staging operations preceding it. Many queries, and their sub-queries, can run in parallel. In applications with highly CPU-intensive application code, as found in high energy physics, tens to hundreds of sub-queries may be running in parallel on a CPU farm.

When a sub-query is scheduled to start its iteration, it first determines which files on disk should be accessed in order to read all objects that it needs to visit. To choose this set of files, the sub-query compares the set of objects it needs to the sets of objects present in the different files of its chunk. These set comparisons are implemented as comparisons between sets of object identifiers, the object identifiers of all objects in a file are obtained by accessing the file-level index of that file. Sometimes, because of an overlap in the object sets contained in the files, there are many options in choosing a set of files which together contain the needed objects. If there is a choice, the sub-query will always choose the set of files with the smallest sum of file sizes. This choice minimises any disk efficiency losses because of sparse reading when accessing the files, and, more importantly, it yields the best possible input for the cache replacement algorithm (section 4.3), in which file access statistics play an important role. When the sub-query comes to visiting a particular object that is contained in several of the chosen files, it will read that object from the smallest of these files. The choice for the smallest file is immaterial to the cache replacement algorithm, which works with file level access statistics and ignores object-level details. The smallest file is chosen based on the assumption that this usually minimises the overall sparseness of reading, so optimising the I/O performance.

2.5 Re-mapping of objects

Many mechanisms are possible for the re-mapping of objects to files. We chose a mechanism based on an object *copy*, rather than an object *move* operation. Using a copy operation has some advantages: in particular, a copy operation does not affect concurrently running sub-queries accessing some of the objects being copied. The use of move operations would require a strong synchronisation between these sub-queries: this makes the implementation more complex and might be a source of performance loss, caused by lower concurrency and more locking traffic. A disadvantage of copying is that the resulting duplication implies less efficient use of scarce storage space, in particular in the disk cache.

Object re-mapping is done during sub-query execution, while the sub-query iterates through its objects. Re-mapping is always from a file (staged) on disk to another file on disk. In the simplest case, shown in figure 3, some objects from a single existing file are copied into a new smaller file. The original file can then be deleted. The end result of such a re-mapping and deletion is that disk space previously occupied by the cold (non-queried, white) objects is freed, while all hot (queried, grey) objects are still present. Thus, we can effectively cache more hot objects on disk.

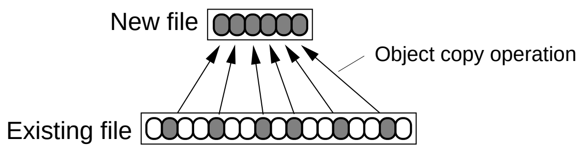

Figure 3: Simplest case of object re-mapping: the (grey) objects read by a sub-query are copied into a new file

Re-mapping decisions are based on the 'densities' of the hot objects in the existing files. For example, if an existing file contains 95% hot and 5% cold objects for a sub-query, then the hot objects in this file will never be copied to a new file, as the storage efficiency in the existing file is already near-optimal. The densities are determined at the start of the sub-query, using the file indices. From this information, a re-mapping function is computed, which can be used to quickly decide, for each object accessed by the sub-query, whether this object should be re-mapped.

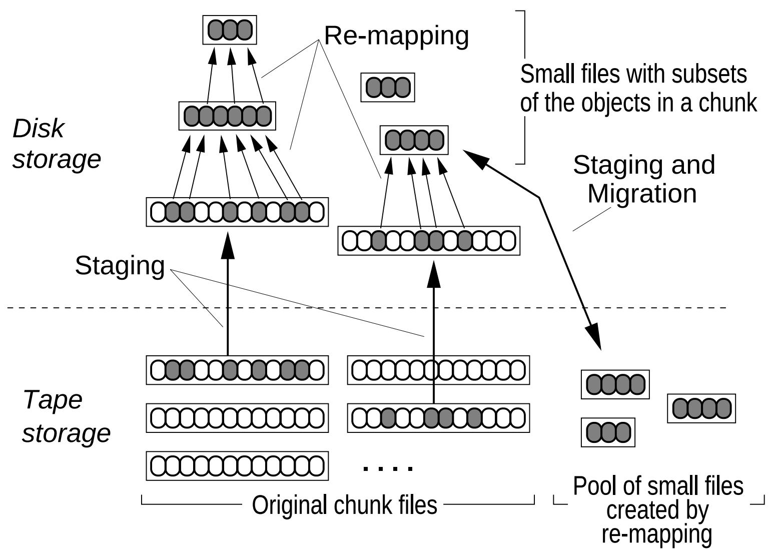
Re-mapping
Small files with subsets of the objects in a chunk
Disk storage
Staging and Migration
Staging
Tape storage
....
Original chunk files
Pool of small files created by re-mapping

Figure 4: Overview of the complete storage architecture

Figure 3 shows the simplest case of re-mapping. In more complicated cases, the copying of some objects into the new file can be suppressed because they are already present in another small file. More details on the optimisation problems involved can be found in [6].

2.6 Complete storage architecture

Figure 4 shows the complete storage architecture of our system, with both the tape and disk storage layers. The tape contains two different pools of files: 1) the original chunk files and 2) smaller files created by re-mapping. The original chunk files are created on tape when the application programmer adds chunks to the system (section 2.2). These files are never updated or deleted. As the original chunk files are always retained, all other files on disk and tape can be deleted, whenever more space is needed, without running the risk of loosing objects. The smaller files on tape are all first created on disk, by re-mapping operations, and later migrated down to tape.

As shown in figure 4, re-mapping operations can be done recursively: if a new query yields a smaller set of hot objects, a still-smaller file can be made, and the larger file can be deleted.

3 Bursty sequential reading

As noted in section 2.4, the high energy physics requirement of using highly CPU-intensive query function code implies that the system should allow for tens to hundreds of sub-queries running concurrently on a CPU farm, all accessing files containing objects. Re-mapping operations may scatter the objects needed by a sub-query over many files. In tests with our system, a sub-query generally reads objects only from a single file, but sometimes from a few files, and in extreme cases from up to 10 files.

Taking everything together, in a running system there could be concurrent access to a few hundreds of files on disk. We use an optimisation we call *bursty sequential reading* to guarantee a good parallel I/O performance. This optimisation works at the file level, and has two parts. First, the objects in a file are always read sequentially, possibly sparsely, in the order in which they are stored in the file. This sequential reading is easy to achieve: in the original chunk files the object ordering is by definition the same as the sub-query iteration order, and all re-mapping operations preserve the object ordering. The second part of the optimisation is that, in every file, the sub-query reads the needed objects in bursts of about 1 MB. The objects read in these bursts are buffered in memory until they are delivered to the query function. The bursty sequential reading optimisation, which we implemented on top of the Objectivity/DB database C++ binding, is sufficient to achieve good parallel I/O performance. Disk throughputs are near the maximum possible throughputs achievable with pure sequential reading, as specified by the disk manufacturers. It should be noted that the Objectivity/DB architecture, which employs no central database engine, also contributes to the good I/O scalability we found.

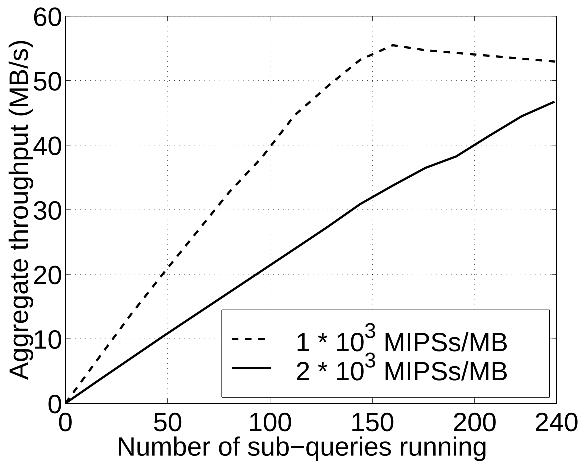

Figure 5: I/O scalability with bursty sequential reading

Figure 5 shows the results of some parallel I/O tests of our sub-queries with bursty sequential reading. The tests were performed on a 256-CPU HP Exemplar supercomputer. We ran up to 240 sub-queries concurrently. Each sub-query uses almost all of the power of a single CPU to execute application code. The two curves are for application code spending $1 * 10^3$ MIPSs per MB read, and $2 * 10^3$ MIPSs per MB read. Each sub-query reads its objects interleaved from 3 files, while also re-mapping (copying) every tenth object into a new file. Every sub-query is executed on its own in a private UNIX process. Multi-threading inside UNIX processes was not used, though it is in principle supported by the database and OS software used.

Both curves in figure 5 show very smooth I/O scaling, indicating efficient use of the available disk resources. In the lower curve, the system remains CPU bound. In the upper curve, with less computation per MB read, the system becomes I/O bound above 160 concurrently running sub-queries: at that point, the available I/O resources (16 disks in 4 striped file-systems) are saturated.

Note that these good scalability results were achieved with only a single optimisation layer on top of a commercial object database product. A special purpose parallel I/O library was not needed.

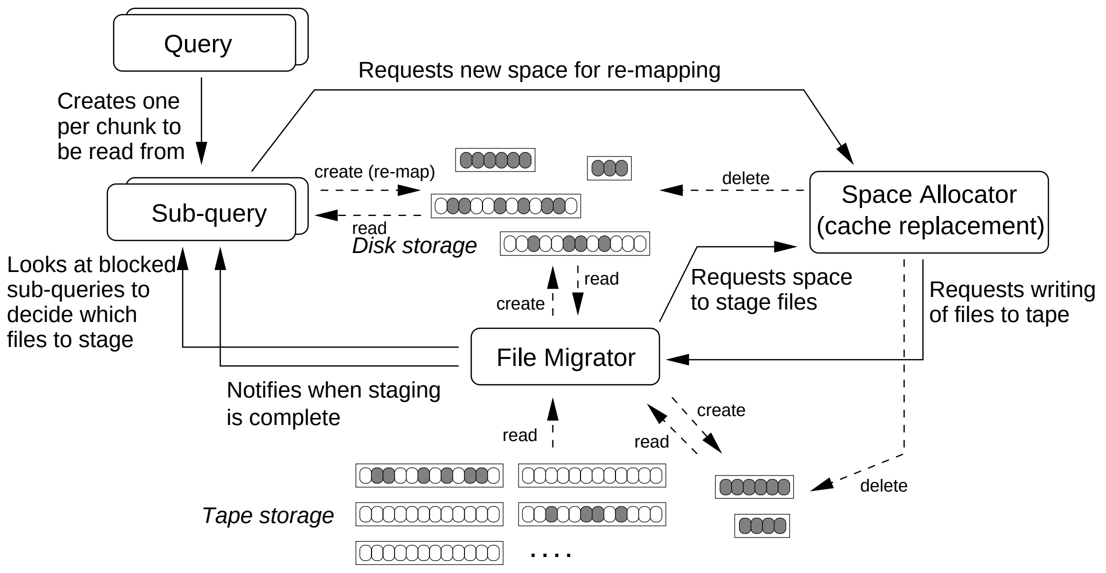
Query
Creates one per chunk to be read from
Requests new space for re-mapping
Sub-query
create (re-map)
read
Disk storage
delete
Space Allocator (cache replacement)
Looks at blocked sub-queries to decide which files to stage
create
read
Requests space to stage files
Requests writing of files to tape
File Migrator
Notifies when staging is complete
create
read
read
delete
Tape storage
....

Figure 6: System components and their interactions

4 Components and policies of the architecture

Figure 6 shows the different active components (rounded rectangles) in our architecture, and their responsibilities and interactions with each other. The components invoke scheduling algorithms (not shown) to make optimisation and scheduling decisions. The query component, and its production of sub-queries, have already been discussed in section 2.4. Below, the remaining components are discussed, along with some of the scheduling algorithms.

4.1 Sub-query

As already discussed in section 2.4, a sub-query performs the reading of selected objects in a single chunk, and executes the query function against them. The sub-query can also perform a re-mapping operation when it is running. When started by its parent query, a sub-query will immediately examine the (indices of) the files on disk to determine if all objects it needs are present on disk. If not, the sub-query 'blocks', it will suspend its execution to wait until all objects are present. It is the responsibility of the file migrator (see below) to ensure that blocked sub-queries eventually become un-blocked.

When a sub-query, possibly after having been blocked, finds all needed objects present on disk, it computes from which files to read these objects, and whether to do any re-mapping. After locking these files against deletion by cache replacement, the sub-query requests permission from a central scheduler (not shown in figure 6) to start reading objects. The central scheduler ensures that not too many sub-queries will do intensive disk I/O at the same time. For example, on the system configuration in section 3, a good limit would be to bound concurrency to some 400 sub-queries. When permission to read is obtained, the sub-query will first request some free disk space if re-mapping is to be done. Then, the sub-query iterates over the needed objects in its chunk. Objects are read from existing files, fed to the query function, and possibly written to a new file in re-mapping.

4.2 File migrator

The file migrator manages the tape drive(s) in the system, and migrates files disk and tape.

The file migrator examines all blocked sub-queries to decide which file to stage next. Many hundreds of sub-queries may be blocked at the same time. Sometimes, many sub-queries (of different queries) are all blocked, waiting for the staging of objects from the same chunk. The file migrator partitions the blocked sub-queries into clusters. Every cluster is a group of blocked sub-queries waiting for objects in the same chunk. For every cluster, the file migrator identifies a single file on tape, whose staging would allow all sub-queries in the cluster to un-block. This pooling of tape requests from different queries is known as *query batching*, and it can lead to dramatic savings [7], especially for workloads with many concurrent large queries.

In any tape-based system, it is important to minimise the randomness of tape I/O as much as possible, because tape mounts and seeks are expensive operations. After an investigation of alternatives, we chose the following policy that aggressively minimises mounts and seeks. The policy cycles over all tapes in the system in a fixed order. When the next tape is reached, and a tape drive becomes available for reading, the file migrator looks if any of the files needed by the current clusters are on this tape. If so, the tape is mounted. Then, any needed files are staged in the order of their position on tape. This results in a sequential pattern of seeking and reading on the tape. When the last file has been staged, the tape is rewound and unmounted. The fixed cycling order ensures that sub-queries are never blocked indefinitely.

4.3 Space allocator

The space allocator manages two pools of files: the files on disk and the small pool of files created by re-mapping on tape. Both these pools can be seen as caches, and so are managed by cache replacement policies.

For the pool of files on disk, the cache replacement policy has to achieve some conflicting aims. Firstly, a recently used file of course has to be retained as long as possible. But secondly, a file from which objects were recently re-mapped should be deleted as quickly as possible, so that the goal of the re-mapping operation, creating a tighter packing of hot objects in the cache, is actually achieved. Thirdly, if a query with a size many times that of the disk cache size is executed, no attempt at caching these files on disk should be made, but they should be deleted as quickly as possible, to maximise the available cache space for smaller queries. A special policy called ’usage based cooling’ was developed to reconcile these conflicting aims. Because of space limitations, we refer the reader to [6] for a detailed discussion of this policy.

For the pool of smaller files on tape, the following management policy is used. A set of tapes is reserved exclusively to hold these small files. One tape at a time is filled, files are written on the tape sequentially. When all tapes in the pool are full, the oldest tape is recycled: all files on it are deleted and writing starts again from the front of the tape. The above scheme amounts to a ‘least recently created’ cache replacement policy. Of course, a policy closer to ‘least recently used’ would potentially be more effective at maintaining a set of useful files on tape, if a way could be found to keep the associated free space fragmentation on tape in check. To investigate the potential benefits of other replacement policies on tape, we used simulation to determine the performance of ‘least recently used’ replacement under the (unrealistic) assumption that there is no performance loss due to fragmentation. We found that a ‘least recently used’ policy simulated under this assumption outperformed ‘least recently created’ with factors of 1.0 (no improvement) up to 1.2 over our range of test workload parameters. From this small improvement factor we conclude that our simple ‘least recently created’ tape replacement policy is an appropriate choice. A better policy may be possible, but it is unlikely to be better by more than a factor 1.2.

4.4 Writing of small files to tape

Chunk reclustering is done by copying some of the small files on disk, which were produced by re-mapping, to tape. When the space allocator determines that a file is soon to be replaced (deleted) from the disk cache, it invokes an algorithm to decide whether the file is to be copied to tape. If the file is to be copied, the space allocator includes this file in a batch of write requests to the file migrator, and will then hold off deleting the file from disk until it has been copied to tape.

There is no obvious method of deciding whether or not a file should be copied to tape before deletion on disk. Obviously, only the ’best’ files should be copied to tape, but when the space allocator offers a file up for consideration, it has already decided itself that this is one of the ’worst’ files it has on disk! We have tried to find a good selection method as follows: we simulated a system in which (unrealistically) all files would be copied, at zero cost, to a very large tape pool before deletion from disk. Then, we examined the files that were actually staged back onto disk in the simulation, to find some identifying characteristics. However, we failed to find a good predictive identifying characteristic that could be used in a selection heuristic. In the end, we used a simple heuristic based on the observation that very large files should obviously not be written back to tape, because the initial chunk files already present would allow for the staging of large object sets at similar costs. We introduced a size cut-off: all files smaller than 20% of the chunk size are selected for copying to tape. This value of 20% was determined in a tuning exercise. We found that 40%, for example, works almost as well. We tried some more refined heuristics but found no heuristic that was noticeably better.

5 Benefits of object granularity

To assess the benefits of object granularity, we used simulation over a large workload parameter space. In these simulations we compared the performance of our object granularity mass storage system with that of a file granularity system, over a range of application workloads and hardware parameters. The chunks of the object granularity system appeared as initial files on tape in the file granularity system. The workloads satisfied the sparse and repetitive access conditions formulated at the end of section 1. The workloads are multiuser workloads, with query sizes ranging from 0.03% to 6% of the complete dataset size, with an average of 0.34%. In our simulations, the disk cache size ranged from 2% to 20% of the

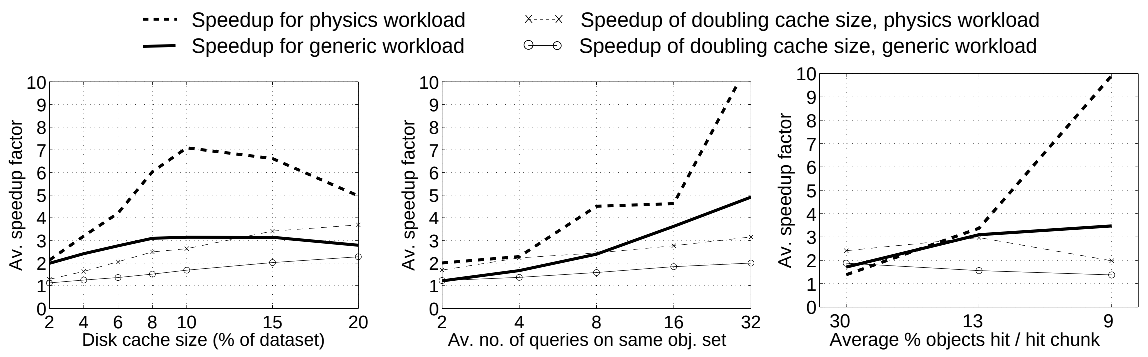

Figure 7: Dependence of different speedup factors on several parameters. Every speedup factor shown in a graph is the average over all combinations of the parameter values on the x-axes of the other two graphs. The rightmost graph shows the dependency on the sparseness of access, on the x-axis is the measure defined in section 1, the average percentage of objects which are needed by a query in every chunk that the query hits.

dataset size. For details on the simulation workloads used, we refer to [6].[1]

To assess the benefits, we determined the speedup factor of our object granularity system over the normal file granularity system. We found that the speedup factor is dependent on many workload and hardware parameters. We found speedup factors from 1 (no speedup) to 52. As expected, speedups are higher when workloads more often access the same sets of objects. For workloads with a high repetition factor, i.e. if on average at least 4 queries visit the same object set, speedups are typically a factor 2 – 4. Other forms of repetitiveness, for example if new queries access subsets of the object sets visited by older queries, also improve the speedup.

Again as expected, following the reasoning behind the sparse access condition in section 1, the speedup over file granularity systems is higher if access to the chunks is more sparse. Speedups in excess of 10 are found if, on average over all queries in the workload and all chunks in the system, a query ’hits’ less than 10% of the objects in every chunk that it hits.

5.1 Dependence on workload parameters

Figure 7 illustrates the dependence of the speedup factors on various system and workload parameters. All these graphs plot averages of speedup factors over parts of the parameter space, and so de-emphasise some of the more extreme cases. Curves for *physics* and *generic* workloads [6] are shown: in a physics workload, new queries access subsets of the object sets visited by older queries, this corresponds to what happens in high energy physics data analysis. In a generic workload the object sets selected by (sequences of) queries are completely independent. For comparison, figure 7 also shows speedup curves for doubling

[1]The workloads used for the tests discussed below differ from those in [6] only in that a new value, 9%, was added to the range of the parameter ’average percentage of objects needed in a chunk that is hit by a query’.

the disk cache size in the baseline system.

5.2 Benefits of keeping a pool of small files on tape

We found that the re-mapping of files on disk, followed by the deletion of original files on disk, contributed most to the speedup, by improving the storage efficiency in the disk cache.

The speedup contribution of having a pool of small files on tape was lower. The small files on tape do save tape resources, because staging in a smaller file can often substitute for staging in the much larger original chunk file. However, the resources needed to write the small files to tape in the first place are considerable. Typically half to all of the time saved by reading small files in stead of larger ones is spent in writing the small files. Figure 8 shows simulation results for our system with and without the optimisation of keeping a pool of small files on tape. The speedup contribution of having small files on tape was often only a factor 1.2 or lower. We found that this speedup depended strongly on the size of the disk cache: the smaller the disk cache, the larger the gains of maintaining a pool of small files on tape. For disk cache sizes of 4% or less than the application data set size, we sometimes found worthwhile speedup factors, from 1.5 to 2.1, depending on workload parameters.

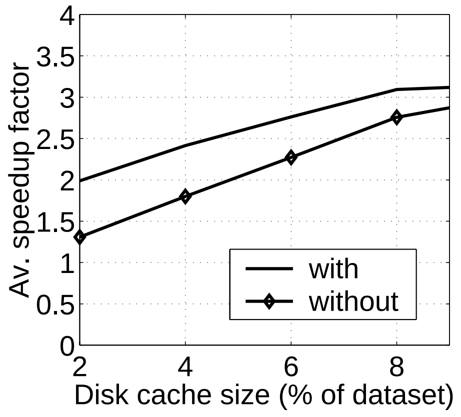

Figure 8: Average speedups with and without the optimisation of keeping a pool of small files on tape, for generic workloads

6 Towards object granularity

The architecture presented in this paper is a viable one, but not the only possible solution or even provably the best solution for an object granularity mass storage system. It therefore makes sense to review the design process that led towards the architecture in this paper, so as to distinguish between largely arbitrary design choices on the one hand and some forced moves on the other hand.

The challenge in moving towards a system with object granularity is to make the grain size smaller, so as to increase chances for optimisation, without making the grain size so small that the system design collapses under the increased complexity, or the I/O performance breaks down due to increased fragmentation. Our solution centres around introducing an intermediate level of granularity. Our system has objects, files, and chunks: three levels compared to the two, objects and files, discernible in applications that use file granularity systems. We perform tasks like migration and cache replacement at the intermediate file level, rather than the object level. This way, many of the drawbacks of true object granularity, like the risks of fragmentation and a too heavy load on the scheduling algorithms, can be avoided. Of course, it was not clear from the start of the design exercise whether the supposed benefits of object granularity, opportunities for successful optimisations, could be preserved when taking this route!

Working from the decision to have an intermediate file level, it is relatively easy to draw a first picture of data movement and layout, like the one in figure 4. Based on this picture one can make a list of all processes that have to be steered by the system implementation. Most these processes interact with each other, and this makes the creation of an optimised design very difficult. To make any progress at all, we decided to take the risky approach of completely disregarding these interactions at first, and decomposing the system into a few weakly interacting active components and scheduling algorithms. The creation of the individual components would then be followed by a 'big bang' integration step. Of course, this 'big bang' design method has a deservedly bad reputation. Before embarking on it, we spent considerable time in searching for more stepwise methods (without much success), analysing the associated risks, and developing techniques to mitigate these risks. We mitigated risks by making robustness of individual system elements an important design goal. We designed the active components and scheduling algorithms so that they would keep working, and ensure some degree of progress, no matter how bad the decisions of other scheduling algorithms would be.

For the 'big bang' integration phase, we used a simulation-driven approach. First, the tuning parameters in all scheduling algorithms were set to some plausible initial values, this way we obtained a first integrated, and more or less working, version of the whole system. Then, using simulations with likely workloads, we tuned the individual scheduling algorithms to globally deliver good performance. Beyond the initial 'big bang' point the integration phase was therefore an iterative one, with many test-adjust cycles. The integration phase was supported by a simulation framework that allowed parameters and algorithm details to be altered quickly. In exploring the workload and tuning parameter space to investigate design options, we used about 600 CPU hours running some 25000 simulations.

An analysis of the above design process shows some obvious opportunities for designing an object granularity system that outperforms our current one. The exact implications of many small decisions in the system design are unknown, so reversing these decisions may lead to a better system. Also, the grain size at the file level is fairly coarse, typically there are at most some 20 files per chunk, and ways could be sought to obtain a finer grain size, with possibly higher payoffs. Most importantly, given the knowledge gained in the creation and evaluation of the current architecture, the creation of a new design that more closely integrates the different scheduling tasks may now be a tractable problem.

7 Related work

To our knowledge, there is no other work which takes true object granularity in caching and staging as a goal, and develops and evaluates an architecture to deal with the associated scalability and fragmentation problems. Our previous work [8] uses techniques similar to re-mapping in a disk-only reclustering system. In fact, the system developed in [8] served as a proof of concept, which gave us confidence that re-mapping could feasibly be introduced in a mass storage system with both disk and tape. Our work [6] explores re-mapping in the disk cache, but not keeping a set of smaller files on tape to achieve some kind of object granularity in staging operations. Our architecture builds on experience from existing mass storage systems [7] [9] [10], especially with respect to cache replacement and staging policies.

Many systems cache data at a finer granularity when it moves upwards in the storage hierarchy, see for example [9]. At least one existing mass storage product [11] structures data into small units (atoms, like our objects), and allows the application programmer to request (sets of) such data units, rather than complete files. To our knowledge, this product uses a caching granularity below the staging granularity, but it does not go down to the 'atomic' level in its caching mechanisms.

Many tape based data analysis systems in use in science allow users or administrators to optimise performance through the creation of new, smaller datasets which contain some selected objects from the complete dataset. Queries can then be redirected to these new datasets, or are redirected automatically. Such strategies are known as 'view materialisation', or, in high energy physics, the creation of 'data summary tapes'. View materialisation strategies are similar in intent and effect to our two new optimisations. This can lead to similarities at the architectural level, see for example [12] for a view materialisation system that, though not targeted at tape based data, has some patterns in common with our architecture. We know of no existing tape based data analysis systems in which creation of such smaller datasets, the picking of objects for them, and their eventual deletion have been automated to a large extent.

8 Conclusions

For the foreseeable future, the use of tape storage remains the only cost-effective option for the massive datasets used in a number of scientific endeavours. CERN is actively pursuing research into data management options to address the needs of its future physics experiments.

We propose a move towards mass storage systems with object granularity, to overcome the impedance mismatch between small application level objects and the large files desired on tape. Such systems hide the mapping of objects to files from the application programmer, and dynamically re-map objects to files in order to improve application performance.

We have identified two conditions, the sparse access condition and the repetitive access condition, which an application must fulfil to make the use of an object granularity mass storage system underneath the application attractive.

We have investigated the potential benefits of object granularity mass storage systems by developing a viable architecture for such a system. The architecture resolves scalability and fragmentation problems by managing files containing (sub)sets of objects, rather than individual objects. The architecture incorporates a commercial object database and a normal file granularity mass storage system. We have evaluated the architecture through implementation and simulation studies. We found speedup factors from 1 to 52. The speedup gains of our object granularity system are mostly due to the increased cache efficiency on disk, which is achieved through object re-mapping. The storage of files with re-mapped objects on tape seems less attractive as an optimisation, except when disk space is very small compared to tape space or average query size.

The architecture is shown to be a viable one, but probably not an optimal one. We have identified some research opportunities that could lead to improvements over the current architecture. The results obtained here will serve as a basis for future R&D at CERN.

References

[1] Objectivity/DB. Vendor homepage: http://www.objy.com/

[2] CMS Computing Technical Proposal. CERN/LHCC 96-45, CMS collaboration, 19 December 1996.

[3] The RD45 project (A Persistent Storage Manager for HEP). http://wwwinfo.cern.ch/asd/rd45/

[4] J. Shiers. Massive-Scale Data Management using Standards-Based Solutions. 16th IEEE Symposium on Mass Storage Systems, San Diego, California, USA, 1999.

[5] A. Hanushevsky, M. Nowak. Pursuit of a Scalable High Performance Multi-Petabyte Database. 16th IEEE Symposium on Mass Storage Systems, San Diego, California, USA, 1999.

[6] K. Holtman, P. van der Stok, I. Willers. A Cache Filtering Optimisation for Queries to Massive Datasets on Tertiary Storage. Proc. of DOLAP'99, Kansas City, USA, November 6, 1999.

[7] J. Yu, D. DeWitt, Query pre-execution and batching in Paradise: A two-pronged approach to the efficient processing of queries in tape-resident data sets. Proc. of 9th Int. Conf. on Scientific and Statistical Database Management, Olympia, Washington (1997).

[8] K. Holtman, P. van der Stok, I. Willers. Automatic Reclustering of Objects in Very Large Databases for High Energy Physics, Proc. of IDEAS '98, Cardiff, UK, p. 132-140, IEEE 1998.

[9] R. Grossman, D. Hanley, X. Qin, Caching and Migration for Multilevel Persistent Object Stores. Proc. of 14th IEEE Symposium on Mass Storage Systems 127-135 (1995).

[10] S. Sarawagi, Query Processing in Tertiary Memory Databases, Proc. of 21st VLDB Conference, Zurich, Switzerland, 1995, p. 585–596.

[11] StorHouse, the Atomic Data Store. Vendor homepage: http://www.filetek.com/

[12] Y. Kotidis, N. Roussopoulos, DynaMat: A Dynamic View Management System for Data Warehouses. Proc. of ACM SIGMOD, May 31 - June 3, 1999, Philadephia, USA.