RJ10507 (ALM1201-007) January 23, 2012
Computer Science

IBM Research Report

DedupT: Deduplication for Tape Systems

Abdullah Gharaibeh[1], Cornel Constantinescu[2], Maohua Lu[2], Anurag Sharma[2], Ramani R. Routray[2], Prasenjit Sarkar[2], David Pease[2], Matei Ripeanu[1]

[1]The University of British Columbia

[2]IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
USA

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

DedupT: Deduplication for Tape Systems

Abdullah Gharaibeh*, Cornel Constantinescu, Maohua Lu, Anurag Sharma, Ramani R Routray, Prasenjit Sarkar, David Pease, Matei Ripeanu*

IBM Research – Almaden, *The University of British Columbia

ABSTRACT

Deduplication is a commonly-used technique on disk-based storage pools. However, deduplication has not been used for tape-based pools: tape characteristics, such as high mount and seek times combined with data fragmentation resulting from deduplication create a toxic combination that leads to unacceptably high retrieval times.

This work proposes DedupT, a system that efficiently supports deduplication on tape pools. This paper (*i*) details the main challenges to enable efficient deduplication on tape libraries, (*ii*) presents a class of solutions based on graph-modeling of similarity between data items that enables efficient placement on tapes; and (*iii*) presents the design and evaluation of novel cross-tape and on-tape chunk placement algorithms that alleviate tape mount time overhead and reduce on-tape data fragmentation.

Using 4.5 TB of real-world workloads, we show that DedupT retains at least 95% of the deduplication efficiency. We show that DedupT mitigates major retrieval time overheads, and, due to reading less data, is able to offer better restore performance compared to the case of restoring non-deduplicated data.

1. INTRODUCTION

Tape has been the workhorse of large-scale, long-term data storage for decades. Several factors still argue strongly for tape-based archival in a modern enterprise where regulatory requirements increasingly demand long-term storage of an ever-growing amount of data. These factors include longevity, reliability, and power. Tape media has a confirmed shelf life measured in decades and a bit error rate that is at least two orders of magnitude lower than that of inexpensive disk [1]. Perhaps even more importantly in a world where energy is increasingly a concern, the relatively low power consumption of tape is a huge advantage [2]. Finally, in a recent report [3], IDC increased their tape drive unit sales growth forecast over their prior year's forecast due to continued strong growth in tape drive and library sales.

Since tape will continue to play a large part in the storage landscape for the foreseeable future, technologies that improve storage efficiency for tape are as important as they are for disk. One of these technologies is deduplication: a popular compression method in storage archiving and backup. It is based on dividing a large data object (file) into smaller parts (data chunks), and storing only the unique chunks and replacing the original object by a set of chunk identifiers (e.g., hashes) that refer to the unique chunks.

Storage tasks for which tapes are well suited are also ones where deduplication is a good fit. Tape is generally used for long-term cold storage of large data, including archival of desktop and server backups, virtual machine images, database snapshots, and email (e.g., Google used tape backups to restore lost Gmail mailboxes [4]). These types of data are highly deduplicable [5-7]. This is confirmed by our own analysis (§4.4) which shows a 32% reduction in storage requirements for workloads captured from a commercial enterprise backup system.

Deduplicating data on tape can result in effectively storing more data per tape, and, thus, fewer tapes needed to store a given amount of data. This, in turn, can lead to fewer tape mounts to retrieve large amounts of data, cutting retrieval times and saving power, and can also lead to fewer tapes that need to be shipped off-site for disaster recovery or archival.

Enabling data deduplication on tapes, however, has to overcome two major problems. First, *high tape mount overhead*: if the chunks of a file[1] end up on more than one tape, then the cost of multiple tape mounts (which can be as high as 1-2 minutes each) will significantly increase retrieval times. Second, *high tape seek time*: chunks of a file that are placed out of order or far away from each other on tape due to data fragmentation (a consequence of deduplication) will increase retrieval time due to tape's comparatively high seek times (an

[1] This work uses files as the granularity unit since in our experience this is the granularity at which users attempt to retrieve data from the archive. However, all the techniques we present would work similarly at coarser granularities e.g., user-defined data collections.

end-to-end seek takes about 90 seconds). To address these two challenges, efficient algorithms for cross-tape and on-tape chunk placement are needed.

The contributions of this work are as follows:

- To the best of our knowledge, this is the first work to explore solutions for data deduplication on tape libraries. We identify and address the main challenges for efficient data deduplication on tapes. Our chunk placement algorithms are able to (i) preserve up to 95% of the deduplication efficiency while (ii) completely eliminating major access time overheads, and (iii) improving the performance of migrating data to tape pools by a factor proportional to the efficiency of data deduplication, which can lead to backup time savings in the order of hours.
- We present an innovative graph model for representing deduplicated data. Our model has a number of important properties. First, it exposes the degree of similarity between files. Second, it enables computing essential characteristics of the modeled deduplicated data (such as computing the final deduplicated size of each group of files after partitioning the original dataset). Finally, the model produces sparse, low memory footprint graphs, a key enabler for efficient graph processing.
- We design and evaluate a low-footprint *cross-tape* chunk placement algorithm that places all chunks of a file on the same tape. The cost to achieve this goal is storage space: some of the chunks that could otherwise benefit from deduplication, i.e., they could be stored once, may get replicated across multiple tapes. To minimize this overhead, we use our graph model and apply clustering methods to identify clusters of files that share significant amount of data, and place them together on the same tape. Using this approach, DedupT retains at least 95% of the deduplication efficiency. Equally importantly, in addition to tape deduplication, the algorithm can be applied to other use-cases such as chunk placement for deduplicated disk-based archival pools like Virtual Tape Libraries (VTL), or to reduce failure propagation on disk pools (if a disk fails, data loss does not spread to other disks).
- We design and evaluate an *on-tape* chunk placement algorithm that aims to reduce seek time overhead due to chunk fragmentation. We demonstrate that even a simple on-tape placement algorithm for the deduplicated data enables better restore time compared to restoring an equivalent non-deduplicated data (up to 40% improvement for our traces). This improvement is a result of reading less data from the tape.

We note that our placement algorithms use chunk-based metadata, which makes them independent of the mode of backup to tape, that is, the algorithms are oblivious to whether the backup is full, incremental or differential. This comes at the expense of some loss in deduplication efficiency across tapes in the case of incremental backup. Our experiments however demonstrate that this overhead is minimal.

The rest of this paper presents background on tape characteristics and related work (§2), the DedupT's operating environment (§3), the cross-tape chunk placement algorithm (§4), and the on-tape chunk placement algorithm (§5). §6 summarizes the results.

2. BACKGROUND

2.1 Tape Characteristics

Tape technology has advanced significantly over the past decades. Tapes like LTO (Linear Tape Open) [8], have high-capacity, a high data streaming rate, built-in data compression, large I/O buffers, and are block-addressable. LTO Generation 5, the current version, has an uncompressed capacity of 1.5TB and a streaming data rate of 140MB/s; the native compression ratio for compressible data averages 2:1. Data on LTO is recorded in a serpentine fashion, meaning that a set of tracks is written moving forward across the length of the tape, following which the tape heads are moved slightly across the face of the tape and data is written back down the tape towards the beginning. This back and forth movement across the tape creates a "wrap", and current generation LTO tapes have 80 wraps.

Tape, however, is still an append-only medium; data cannot be overwritten on a tape without losing access to already-written data that follows it. In the past this has made it difficult to implement a sophisticated and high-performance file system on tape due to the need to store the tape directory at the end of the data. Starting with LTO Generation 5, this problem has been addressed by creating the ability to partition the tape into individually-writable sets of wraps. The Linear Tape File System (LTFS) [9] uses this capability to record an updatable index on a small partition on an LTO tape while using the bulk of the tape for append-only data storage.

The LTFS file system is an appropriate choice for implementing tape-based deduplication. LTFS stores files in block-based extents, just as most disk file systems do. Files can be discontinuous on tape, and files can share blocks. When data in an existing file is updated, the new data is written to the end of the data partition and the file extent list is updated to reflect the

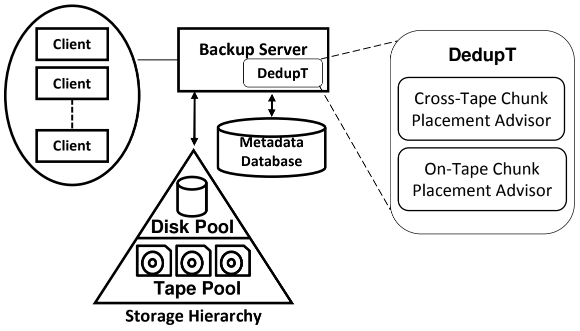

Figure 1: Design of a typical data protection solution

changed location of the data. This allows for deduplicated files to be efficiently represented on tape systems.

2.2 Related Work

We are not aware of any refereed publication presenting a solution for storing and restoring deduplicated data on tape pools. Data protection solutions from Symantec (Backup Exec [10]) and EMC (Avamar [11]), which include tape pools as a back-end disaster recovery storage have a deduplication option. Still, it is not clear whether the data written on tape is deduplicated or not (i.e., re-duplicated to original file format similar to what IBM Tivoli [12] does), and if deduplicated, how the chunks are placed on the tape pool.

MAID-based disk archival technologies that compete with tape have been examined in previous work [13]. However, these technologies acknowledge that they are not yet suitable for long-term cold storage, and the cost, low energy consumption, shelf-life, and portability of tape, continue to eclipse disk for peta- and exa-scale long-term enterprise class storage in real world deployments.

Burns et al. [14] present a solution for managing delta backups for file versions on tapes. Lillibridge et al. [15] present a technique for efficient chunk-lookup for large-scale deduplicated tape images, but does not address the problem of chunk placement.

Related literature on deduplication technology has mostly focused on the efficiency of duplicate detection [16-18], design [19] and scalability [20, 21] aspects on disk storage.

3. OPERATING ENVIRONMENT

The DedupT operating environment is based on the typical design of enterprise-class data protection and recovery solutions (Figure 1) [12]. Such storage solutions offer a storage-medium-agnostic interface to *client nodes* (such as enterprise application servers) in order to back up and restore data based on a defined schedule. Client nodes interface with the system via a component called the *backup server*, which performs data protection operations. The system allows users to define logical groups of files, referred hereafter as *filespaces*, to enable easy management and application-level grouping. Internally, the *backup server* includes a comprehensive policy framework that orchestrates data placement on multiple media types, such as disks and tapes, and migration between them based on specific criteria. Additionally, the backup server employs deduplication in disk pools in order to achieve storage efficiency.

The functional role of DedupT in this operating environment is to determine which tape and where within that tape a chunk will be placed when migrating already deduplicated data from disk pools to tape pools. We refer to these operations as *cross-* and *on-tape chunk placement*, respectively.

DedupT decides on the placement plan by examining the disk pool's deduplication metadata (typically stored in a scalable database) which offers information about chunk dependencies between files. To this end, DedupT is agnostic to the backup mode, whether it is full, differential or incremental, as the operation of identifying deduplicated chunks in a file is independent of the backup mode from which the file was obtained. Moreover, DedupT adheres to user-defined placement constraints (e.g., the files of a *filespace* must be placed on the same tape). The placement plan produced by DedupT is communicated to the *backup server*, which in its turn initiates the actual data placement.

4. *CROSS-TAPE* CHUNK PLACEMENT

The cross-tape chunk placement algorithm has two main inputs: (i) the deduplication metadata (the chunk-maps of files), and (ii) the tape size(s). The output is the list of chunks that will be placed on each tape.

The rest of this section discusses the requirements for cross-tape chunk placement (§4.1), the evaluation metrics (§4.2), our solution (§4.3), and its evaluation (§4.4).

4.1 Requirements

The following are the main requirements for cross-tape chunk placement:

- *All chunks of a file must be available on a single tape.* The idea is to restrict the retrieval of a file to accessing one tape only, hence reducing tape mounting overhead to the minimum (i.e., mounting at most one tape when restoring a file). An added benefit to this restriction is that tapes will be self-contained. This is useful for two reasons: first, self-

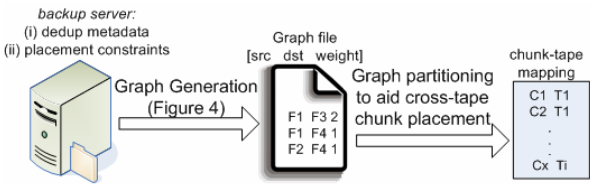

Figure 2: Cross-tape chunk placement high-level process

contained tapes reduce the effect of failures as a file will always depend on one tape only, and second, since tapes are usually moved offsite for archival, moving a file will require moving one tape only (no dependencies will need to be considered as it would be the case where deduplication is used and chunks have full placement freedom).

- *Scalable and fast solution.* The solution must be able to scale efficiently with respect to the total data size (e.g., be able to handle multi-petabyte systems). Also, depending on the frequency with which data is pushed from the disk pool to the tape pool, the solution must be able to provide a placement decision in a reasonable time window.
- *Cater to specific placement policies.* Backup solutions enable users to define policies that influence data placement. For example, a *filespace* must exist on a specific tape. Another example is having the files of a user placed together on the same tape. In this context, the solution must be flexible enough to take such policies into consideration when making chunk placement decisions.

4.2 Success Metrics

The competence of the solution is measured by the following metrics:

- *Deduplication loss, that is, the amount of data replicated across tapes.* To fulfill the first requirement above, the solution may need to store some chunks on more than one tape. For example, if two files share some chunks, and they end up on two different tapes, then the shared chunks must be placed on both tapes.
- *Transfer-to-tape performance.* This refers to the end-to-end performance of moving data from a disk pool to a tape pool. This is affected by the time and memory complexity of the placement algorithm, and the eventual volume of deduplicated data to be migrated.
- *Restore performance.* This refers to the time it takes to restore data. Restore scenarios include restoring a single file or a group of files (e.g., a database backup *filespace* which may include several log and data files).

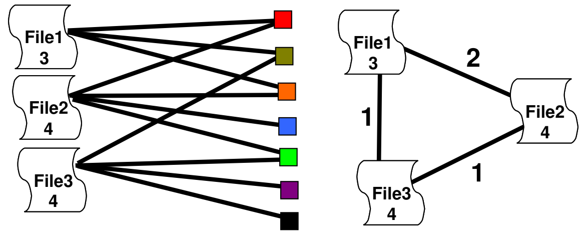

Figure 3: *Left*: Chunk-centric model. Chunks and files are modeled as vertices in a bipartite graph. An edge exists between a file and a chunk if the chunk exists in the file. *Right*: File-centric model. Files represent vertices; while edges represent sharing between files. Edge weights represent the amount of sharing, while vertex weights represent the file size (this example assumes fixed-sized chunks and weights represent number of chunks).

4.3 Solution: Graph-Based Placement

Our cross-tape chunk placement algorithm uses a graph representation to model similarity between files. The motivation behind this approach is to enable identifying clusters of files that share significant amount of data. The idea is to place these clusters together on the same tape to reduce the cost of replicating chunks across tapes.

Figure 2 illustrates the high-level process of our graph-based cross-tape chunk placement algorithm. The process includes two high-level phases: (i) graph generation, and (ii) graph partitioning. The rest of this section elaborates on how we model deduplicated data as a graph (§4.3.1), the graph generation (§4.3.2), and the graph partitioning (§4.3.3) processes.

4.3.1 Graph Models

We identify two main ways to expose data similarity through a graph representation: *chunk-centric* and *file-centric* (Figure 3). Note that either representation supports both fixed- and variable-sized chunks.

The ***chunk-centric model*** represents both chunks and files as vertices. Edges exist between files and chunks only: an edge exists between a file and a chunk if the chunk exists in the file; hence forming a bipartite graph (Figure 3, left).

The chunk-centric model includes detailed, chunk-level, sharing information. This information may better inform graph processing operations (such as graph partitioning to aid data placement). However, most practical graph processing algorithms assume that the graph data structure fits in memory. Since each unique chunk and file in the system is represented as a vertex, the resulting graphs are prohibitively large for realistic data repositories.

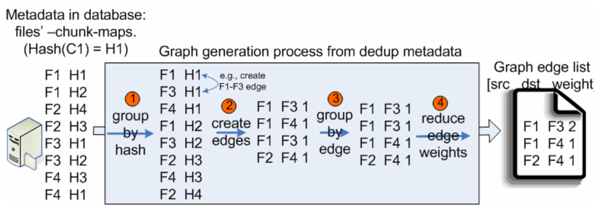

Figure 4: Generating file-centric graph from dedup metadata. A STAR linking strategy is used in this example. Note that the edge weights here refers to number of chunks; in practice, however, they represent number of bytes.

For example, assume a 10TB data repository with 1 million files, divided into 4KB chunks and with 70% unique chunks. A *chunk-centric* graph model for this system includes over 1.8 billion vertices and over 1.8 billion edges (as each chunk is connected to at least one file). Therefore, if both vertices and edges are represented by 4-byte integers, then the memory footprint of such a graph would be over 14GB.

The excessive size of the *chunk-centric* model makes it impractical. Therefore, we considered another model, dubbed *file-centric,* which offers a much lower memory footprint.

The ***file-centric model*** represents only files as vertices. Edges are placed between files that share chunks. Edge weights represent the amount of sharing (Figure 3, right). This model summarizes the detailed information offered by the *chunk-centric* model: it includes the amount of sharing rather than which chunks are shared.

Giving up the detailed information produces a relatively small graph that is proportional in size to the order of number of files, which typically is orders of magnitude lower than the number of chunks. For instance, in the example mentioned earlier, the memory footprint of a *file-centric* model will be in the order of a few tens of megabytes.

It is important to mention here that the vertices in our file-centric graph are connected with a *minimal set of edges* in the following sense. Let's say that *n* files share a chunk, then the number of edges between this set of files is *n – 1*. The idea is to not have duplicate edges that model the sharing of the same content between files.

This enables an important property of our file-centric graph model: the ability to approximate --as an upper bound-- the deduplicated size of a set of files (or its corresponding graph or sub-graph) through a simple graph traversal. In particular, size can be computed by a *breadth-first search* (BFS) traversal that sums the vertices' weights (which represent files sizes) and subtracts the edges' weights (which represent the duplicates size). We note that for a non-partitioned graph component[2] this is not an approximation but always produces the correct result.

4.3.2 Building the graph

One important practical aspect is efficiently building the graph, particularly identifying the edges, which represent sharing dependencies, between vertices (i.e., files).

The process takes as input deduplication metadata, particularly the file-chunk map which identifies the list of chunks each file is composed of. As output, the process produces a graph represented as a list of edges, called an edge list. The edge list represents the key input to the partitioning process discussed in the next section.

One important requirement for building the graph is scalability: it should scale to handle petabyte-scale systems. Therefore, scalability-limiting constraints (such as requiring large memory footprint) should be avoided. Figure 4 illustrates our process of building the graph. In the following, we detail the process:

1. *Group files by hash:* the goal of this step is to group together files that share a chunk. This can be done by sorting the file-chunk map using the chunk hash as the sorting index. Several efficient and scalable disk-based sorting algorithms which do not demand high memory footprint exist (e.g., ORDER BY SQL clause and the Linux tool `sort`).
2. *Create edges:* the goal of this step is to create edges between files that share a chunk. The grouping in the first step makes this step a simple scan over the ordered file-chunk map. For our sparse, *file-centric* graph representation, we have tested two heuristics that satisfy the 'minimal set' requirement outlined above: *STAR* – where one of the files is linked with every other file, and *CHAIN* – where the files are linked to each other in a linked list. For the STAR topology the master node is connected with *n-1* nodes, so it has degree *n-1*, while the rest of the nodes have degree 1. In the CHAIN topology, on the other hand, the two end nodes have degree 1 while the intermediary nodes have degree 2, so the distribution of node degrees is more balanced. Note that between these two heuristics there are many other ways to link the *n* files by a minimal set of edges. We have evaluated the impact of these two heuristics on the overall partitioning effectiveness.

[2] A component is a maximal subgraph in which there is at least one path between each two vertices in the subgraph.

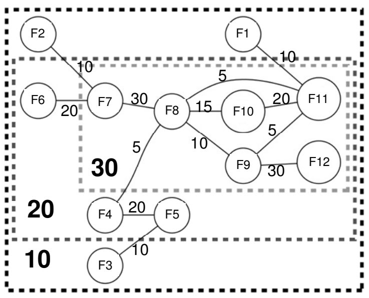

Figure 5: p-core algorithm in a hypothetical scenario. The files inside the inner clusters share more data with each other than the files outside. Numbers in bold indicate corness.

This step may produce more than one edge between any two vertices (consider two files sharing two different chunks). The next two steps aim to reduce edges between the same pair of nodes into one.

3. *Group edges:* this step groups together edges between similar vertices (files). As in the first step, this is done via sorting.
4. *Reduce edge weights:* this step performs a sum reduction over the weights of similar edges. This is done via a single scan over the ordered edge-list that resulted from the previous step.

It is important to stress here that this graph generation process does not impose excessive memory requirements. The performance of the two major operations, hash and edge grouping, can benefit from additional physical memory, but, as discussed above, can still be processed efficiently on disk.

Finally, our graph generation process caters to user-defined placement policies in the form of "*filespace* FS_x must be placed on tape T_y" (remember that a *filespace* represents a group of files). Such placement policies are handled by representing the group of files that represent the *filespace* as a single vertex in the graph. The size of this new *filespace*-based vertex is the sum of the sizes of all files in the *filespace* (after internal deduplication). Similarly, edges between files in the *filespace* and other files are grouped together.

4.3.3 Partitioning the graph

The goal for partitioning is to divide the graph into a number of partitions such that each partition can fit into a given tape size, while reducing the number of chunks replicated across the resulting partitions (i.e., reducing the edge cuts).

One solution to this problem is to recursively bisect the graph into two partitions of about equal size, while minimizing the number of edges that span the partitions. This is an NP-complete problem [22], but a number of heuristics exist to address it. The best available heuristic has O(*ve*) complexity [22], where *v* is the number of vertices and *e* is the number of edges. Such expensive algorithms can potentially be a bottleneck, especially when considering large petabyte-scale storage systems.

However, in our case, partitioning is not meant to identify balanced sized partitions; rather its goal is to identify clusters of files that share significant amounts of data. To this end, we propose a simple partitioning heuristic that is based on two linear-time graph processing algorithms: breadth-first search and k-core [23].

The algorithm first identifies the connected components of the graph. The rationale is that separate components do not share data; hence they can be placed on different tapes without cutting any edges (no chunks will be replicated). This step requires linear time as connected components can be determined through straightforward breadth-first search.

The size of some components, however, might be larger than what a tape can store, hence the need for further partitioning. To this end, the second phase of the algorithm partitions the large-sized components via a variation of the k-core clustering algorithm [24]. In particular, a k-core is a maximal sub-graph in which each vertex is connected to at least *k* other vertices. Maximal here means that no node can be added to the subgraph while preserving the aforementioned property. Determining the core decomposition of a graph has an appealing linear time complexity of O(*e*).

Since the focus is on the amount of data a file shares rather than the number of files it shares with, we adopt a variation of the k-core concept that takes into account the weight on the edges instead of the degree. This concept is called p-core [24]. Similar to k-core, p-core is a maximal subgraph in which the sum of edge weights of a node is at least *p*. Computing the p-core decomposition has a similar linear time complexity to k-core.

In particular, the p-cores of a graph form circles (Figure 5), where core (*p+1*) is always a subgraph of core *p*. The "coreness" of a vertex is defined as the maximum core it belongs to. Note that the inner circles contain vertices with higher coreness.

The core-based partitioning algorithm divides the large-sized components into smaller partitions such that each partition spans a range of circles. For example, the first partition includes files with coreness between [0,x), where x is chosen such that the partition size fills a tape, the second one between [x,y) and so on. This

Table 1: Workload summary

Workload	*WL1*	*WL2*
Total size	3,052GB	1,532GB
Duplicates (size)	978GB	460GB
Duplicates (%)	32%	30%
Num. of files	289,295	201,406
Avg. file size	10MB	7.79MB
Median file size	82KB	18KB
Num. of chunks	17,509,025	12,021,126
Avg. chunk size	182KB	102KB
Median chunk size	71KB	52KB

partitioning heuristic leads to grouping together files that share a specific amount of data, hence reducing the chance of splitting clusters of files that share significant amount of data.

4.4 Evaluation

4.4.1 Setup

We evaluate our algorithm using two real workload traces taken from Tivoli Storage Manager (TSM), a commercial backup solution from IBM. Deduplication in the traces is based on variable size chunking that used SHA1 to produce chunk fingerprints. Table 1 shows the characteristics of the workloads.

Building the graphs from the two workloads was sufficiently fast (Table 2). The graphs were generated from the workloads' metadata residing in a relational database on a commodity machine (quad-core Intel processor and 8GB of memory). The Linux sort command was used for the grouping steps in the graph generation process described in §4.3.2. These steps accounted for over 90% of the processing time.

Table 2 shows the characteristics of the generated graphs. The low graph density and edge degree of both workloads highlights the sparsity of our file-centric graph model.

Figure 6 shows the distribution of component sizes. The plot shows large number of small components and small number of large components. The size of the biggest component in each workload is 22% and 64% of the total size of workloads ***WL1*** and ***WL2*** respectively.

Unless otherwise noted, the experiments consider a scenario where the whole workload is placed on as many tapes as needed of a specific maximum tape size. In other words, the partition has a maximum size that the algorithm tries to utilize to produce the lowest number of partitions. All experiments present the results while varying the tape size. Although in practice tape size does not vary (current typical size is 1.5TB), the motivation behind varying the tape size is to evaluate the algorithm's behavior for different workload to tape size ratio.

Table 2: Graph characteristics

Workload	*WL1*	*WL2*
Time to generate	5.6min	3.8min
Number of nodes	289,295	201,406
Number of edges	327,472	246,244
Graph density	$8e^{-6}$	$12e^{-6}$
Num. of components	166,089	149,083
Size of the largest comp.	695GB	987GB

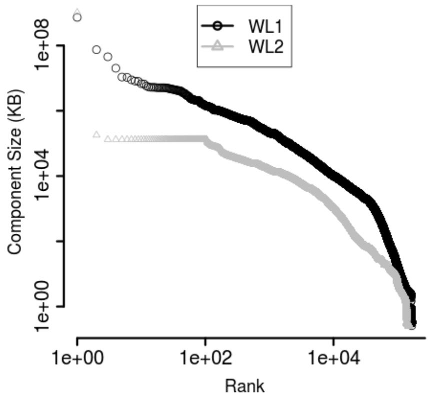

Figure 6: log-log plot of component sizes distribution

The majority of this section (§4.4.2 - §4.4.5) evaluates the storage overheads. To this end, we define the *deduplication loss* as the amount of replicated data across partitions divided by the total amount of duplicate data in the non-deduplicated system. The numbers on top of the bars represent the resulting number of partitions. The last section (§4.4.6) discusses transfer-to-tape and restore performance.

4.4.2 Comparing With Naive Placement

We compare DedupT with a naïve placement heuristic. The naive algorithm simply places the files one by one,

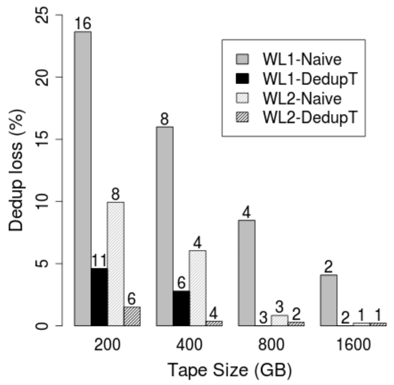

Figure 7: DedupT vs naive placement. Numbers on the bars are the number of resulting partitions. ***Missing bars represent zero dedup loss.***

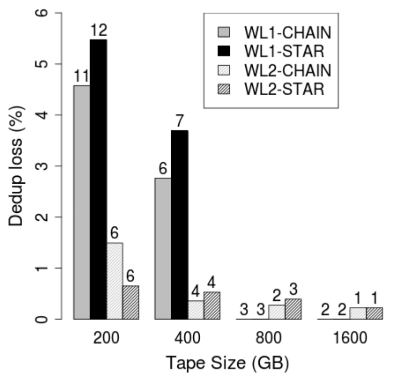

Figure 8: Comparing two linking strategies: CHAIN and STAR.

as long as there is enough space left, and then the files on the same tape are deduplicated.

Figure 7 shows the results. Small tape sizes force the algorithms to cut sharing dependencies between files to create corresponding small partitions. Still, DedupT maintains a lower than 5% dedup loss, and offers 5 times improvement over the naive algorithm for the smallest two tape sizes.

As the tape size increases, the deduplication loss decreases as larger clusters of files can fit in a tape. For the largest two tapes, DedupT is able to place the whole ***WL1*** without loss in deduplication as even the biggest component can fit in a tape without partitioning.

Finally, it is important to stress here that DedupT and naive algorithms represent examples from the two ends of the "complexity" spectrum of placement algorithms, where complexity here refers to all aspects of a solution such as development and time complexities. Other heuristics in the middle of this spectrum can be proposed (e.g., bin-packing), which will offer different complexity –storage overhead tradeoff.

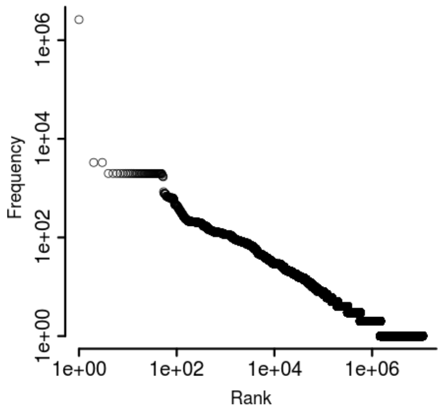

Figure 9: Log-log plot of chunk frequency distribution for WL1

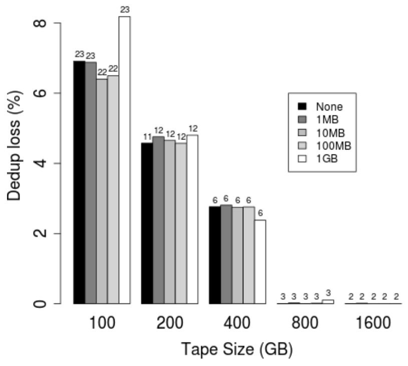

Figure 10: The effect of removing high frequency chunks from WL1. The "None" bar shows the result of partitioning without removing any chunks, while 1MB shows the result of partitioning after removing 1MB worth of most popular chunks.

4.4.3 Comparing Linking Strategies

As discussed in §4.3.2, the graph can have different representations. We evaluate the effect of the two extreme linking strategies: CHAIN and STAR. As Figure 8 shows, the difference between the two extreme topologies is insignificant (±1%). This illustrates the robustness of our sparse file-centric graph model. The reason behind this is the fact that both representations capture the same collective sharing information, which is the main director for graph partitioning.

4.4.4 Removing High Frequency Chunks

Ruling out the high frequency chunks from consideration when building the graph has the potential to reduce dependencies between files, hence improving the partitioning result.

Figure 9 shows the chunk frequency distribution for the bigger workload, ***WL1***. The plot shows close to a power-law distribution: few high frequency chunks, and many low frequency chunks.

Figure 10 shows the partitioning result for ***WL1*** after removing the most popular chunks and replicating them directly on all tapes. Removing 10MB or 100MB worth of high frequency chunks contributes to minor improvement. Furthermore for larger tape sizes, removing the most popular chunks contributes nothing. This is because the cost of replicating the most popular chunks on all tapes out weighs the gain in reducing dependencies between files.

4.4.5 Adding Data Incrementally

A typical scenario in archival systems is periodical data transfer from the disk pool to the tape pool. In this scenario, the tape pool will have some old data while

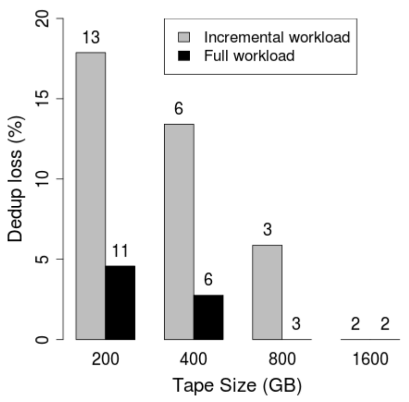

Figure 11: Incrementally adding data to the system versus placing the full workload at once.

new data is pushed to it. The assumption here is that old tapes are still part of the tape pool (e.g., has not been shipped offsite yet).

This experiment divides ***WL1*** into 10 batches, which represent 10 periods of time, and are pushed to the system one after another. The idea is to emulate incremental addition of data to the system. The assumption in this scenario is that data placed on a specific tape from an early batch cannot be moved to a different tape when placing new batches, and that for each new batch, a new graph is created for the files in the batch. To model files placed from previous batches, we adopt the following approach: each group of files placed on the same tape is represented as a single vertex in the new graph. The weight of these new tape-level vertices is the sum of all weights of the files in the tape (after internal deduplication). Similarly, edges between the files in a tape and the files from the new batch are grouped together.

This approach ensures that already placed files stay together in the same tape in the final partitioning result, while giving the chance for new files that share significant amount of data with some old files to be placed together. This is done by maintaining a specific percentage of free space in each tape after partitioning each batch.

Figure 11 compares the two scenarios: incrementally adding the workload versus placing the full workload at once. The incremental scenario suffers higher deduplication losses because it makes placement decisions that are based on the chunk sharing in a batch, which typically is less revealing compared to the chunk sharing in the whole workload. For example, two files in a batch that do not share chunks may end up on two different tapes; however, a later batch may contain a file that share significant amount of data with both of them. This new file will likely to be placed in one of the previous two tapes; hence the data shared with one of the previous two files will have to be replicated.

4.4.6 *Transfer-to-Tape Performance*

In the case of transfer performance (moving data from the disk pool to the tape pool), the major overhead is the tape write throughput, which is around 140MB/s [9], so writing terabytes of data takes hours (~2 hours for 1TB). Hence, the few minutes of preprocessing overhead, which includes generating and partitioning the graph, is negligible.

Thus, when comparing with a system that does not use deduplication, the critical factor for transfer performance for the deduplicated system is its ability to preserve the deduplication opportunities. For our workloads, depending on tape size, DedupT, with less than 6% deduplication loss, preserves the benefit of transfer time reduction that deduplication provides.

When comparing with a system that uses deduplication but does not use any intelligence in data placement and replication to make tapes self-contained, DedupT would be slightly slower for our workloads because of replicating some of the chunks. For example, in the ***WL1*** workload***,*** in the worst case, DedupT is only 1.3% slower: deduplication loss is 4.5% (Figure 4) of the 32% duplicate chunks in the data (Table 1), which is equivalent to 1.3% of the total amount of data.

5. *ON-TAPE* CHUNK PLACEMENT

Tape's high seek times combined with data fragmentation resulting from deduplication can lead to high restore times if chunks are not carefully placed; hence, there is a need for an on-tape chunk placement algorithm.

The placement algorithm has one input: the file-to-chunk map for the set of files to be placed on the tape. Its output is the placement order of the unique chunks on a tape.

There are two main restore scenarios that may influence on-tape chunk placement: (*i*) restoring an entire tape, and (*ii*) restoring a subset of all files. The first scenario is the traditional way to use tapes. However, the second scenario, restoring a subset of files from tape-based archival systems is gaining traction. The size of the restore subset can span a wide range. The subset size can be small. For example, a social network application may use tapes to store cold data to save energy and space. In this context, user requests for cold data could trigger restore requests for a small number of files (blog entries, emails, movies, photos, etc.) from tape. The subset size can also be large. For example, if the tape is used to hold multiple incremental backups, restoring a single incremental backup can easily occupy a large portion of the total tape size. We believe that restoring a subset of files from tape-based storage will be more common in the near future. New tape technology, which offer filesystem-like access interface [9], further encourages such usage scenario.

On-tape data placement for the first scenario is straightforward. To utilize the tape's high sequential read throughput, the chunks can be placed next to each other on the tape in any order. At restore time, the chunks are copied from the tape sequentially and placed on a disk-based scratch storage pool according to their physical offset on the tape (similar to the Unix `dd` command). Our cross-tape placement solution described in §4.3 ensures that all the chunks needed to reconstruct the files are on a single tape (so no other tapes need to be accessed).

In the scenario where a subset of all files are restored, fragmentation due to data deduplication can lead to high restore times. However, as shown in §5.3, restoring the same set of files can be faster from the deduplicated tape than from the non-deduplicated tape regardless of the size of the restored files.

The rest of this section focuses on on-tape chunk placement for the subset-of-files restore scenario. The section presents a performance characterization of modern tape systems (§5.1), our on-tape chunk placement algorithm (§5.2) and evaluation (§5.3).

5.1 Tape Performance Characterization

Tapes exhibit two important characteristics that affect the performance of retrieving a fragmented file: first, forward seek time offers roughly the same cost as reading data in the same direction within a wrap; and, second, short forward seeks are expensive and must be avoided. This section characterizes the performance of an LTO-5 tape from this perspective.

Figure 12 shows the seek latency on an LTO-5 tape, while varying the seek distance; the sequential-read latency for the same distances is shown for comparison. Our setup provided a sequential-read throughput of 90 MB/s. The measurements show that for strides less than 4 MB, seeks can be two times slower than continuous reads. This is because seeks have mechanical motion, and our measurements show that the tape-drive's data cache is invalidated when a seek request is performed. For strides greater than 4MB, seek performance is consistently better, with the exception of 32 MB seek strides. The exception of 32 MB seek strides is not surprising: LTO-5 drives employ multiple speeds to seek to different scales of distances [1], and 32 MB happens to be the distance where the tape drive switches the speed. Further

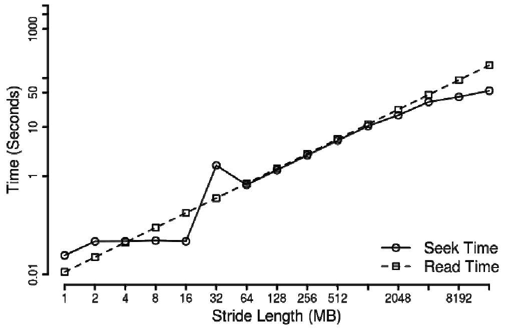

Figure 12: Tape seek and read time while varying the seek or read distance inside a LTO-5 wrap (18 GB).

experimentation showed that large seeks that span multiple tape tracks are much faster than reading that distance. In our setup a 274 GB seek takes an average of 47 seconds, at 90 MB/s this range would take 50 minutes to read. Therefore, an algorithm that tries to minimize both the number of seeks, and the distance between fragmented chunks, should improve restore throughput.

There are two approaches to reading two chunks that reside in the same tape track, but are separated by some unneeded data: either read all data and discard the unneeded data in the middle, or read the first chunk and perform a seek to the second one. We use the former approach when the distance between two reads is less than 4 MB; the latter approach is used otherwise.

5.2 Solution

The key to minimizing the restore time of a subset of files is to reduce the per-file inter-chunk distance. However, for a deduplicated set of files, the problem of minimizing inter-chunk distance is difficult because one does not know in advance the access pattern of files. For example, if we know a set of files are always accessed together, we can put chunks of these files close together on the tape to minimize the inter-chunk distance. However, in practice, it is not easy to identify the related set of files in advance. In the following discussion, we assume a file is the basic access unit and each file has the same access probability.

In this section, we present one simple algorithm to place chunks: denoted as Simple Placement (SP). The SP algorithm does not leverage the sharing structure of files, and simply places the chunks based on a specific file order. We pick the increasing file size order to guide the chunk placement, the same as the method proposed by Knuth in [25]. Particularly, for a chunk in a file, if it previously appeared in the same tape (by querying the per-tape membership metadata), it is not

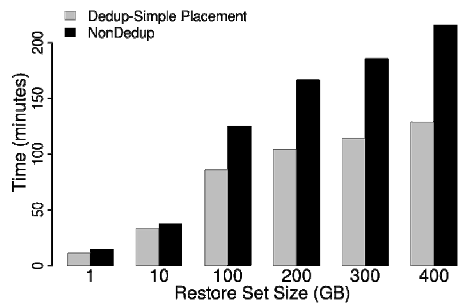

Figure 13: restore performance while varying the restore set size.

placed on the tape again; otherwise, the chunk is appended to the end of the tape. The membership metadata is implemented as a hash table keyed by the chunk identifier in the disk pool. As shown in the evaluation, the SP algorithm performs better than non-deduplicated data placement regardless of the restored subset size.

5.3 Evaluation

We use ***WL1*** (Table 1) to evaluate our on-tape chunk placement algorithm. We use only the largest component that resulted from the cross-tape placement phase. The non-deduplicated size of the largest component is 1500GB, while the duplicated size is 746GB.

Our evaluation setup consists of an LTO-5 tape-drive with a 1.5 TB cartridge, connected via Fibre Channel to a Windows 7 host machine running LTFS. The measured sequential read bandwidth of this tape-drive was 90 MB/s. To measure and analyze the restore performance, we carried out the following experiments.

We picked sets of files to restore from the tape, for restore-set sizes in the non-deduplicated form ranging from 1 GB to 400 GB. We refer to the data placement plan for the non-deduplicated files as the NonDedup placement plan, where the files are also ordered in the increasing file size order. The files within a restore-set were picked randomly from the NonDedup placement plan, but the same restore-set was evaluated against the SP algorithm.

When reading a restore-set with respect to a placement plan, a read plan was computed that read all the necessary chunks in the order of increasing tape-offset, thus all seeks were always in the direction of the data layout on the tape tracks, and never against. The read plan also used a read-vs-seek threshold of 4 MB, as mentioned in §5.1.

Figure 13 restores from the deduplicated data set (Dedup) always outperform restores from the non-deduplicated one (NonDedup), even though deduplication fragments the file data. This is mainly because the amount of deduplicated data for Dedup is smaller than that for NonDedup across all restore set sizes. Although there are more seeks for Dedup than for NonDedup, the seeks for Dedup were shorter. The figure also shows that as the restore set increases in size, the performance difference between Dedup and NonDedup increases. This is because of improved deduplication savings with larger restore set, which results in reading less data from tape. For example, for the restore set size of 1GB, the space savings due to deduplication was only 2.5%, while for the restore set size of 300 GB, the space savings due to deduplication is 30%. In particular, when the restore data size is larger than 100 GB, the restore time of the SP algorithm is 31% (for 100 GB) to 40% (for 400 GB) shorter than that for the NonDedup case.

5.4 Discussion

We tried multiple alternative data placement plans which mainly focused on different ways of clustering the shared chunks, but none of them were consistently better than the SP algorithm. For this reason, we focused only on the SP algorithm for the placement of deduplicated data. Due to space limitations, we do not describe the details of these algorithms.

The straightforward SP algorithm is effective for two reasons. First, duplicate data has spatial locality, which mitigates the impact of the fragmentation due to data deduplication. Second, the increasing file order helps DedupT to reduce global inter-chunk distance. The increasing file order effectively groups the smaller per-file singleton chunks closer to the first occurrence of the duplicate chunk; hence forming smaller overall gaps at the global level for those duplicate chunks, and reducing the average inter-chunk distance.

6. SUMMARY

The amount of archived data is predicted to grow at massive rates: doubling every two years and up to 50 times by 2020 [3].

Tape-based storage is well positioned to support the archival load based on longevity, reliability and power. To this end, we explore the feasibility of combining tape-based archiving with deduplication. This combination merges the major advantages of the storage media and the data reduction technique. Tape's low cost per gigabyte, low bit error rate, performance, ability to support filesystem-like access, long shelf-life time and small physical footprint strongly suggests that it will continue to be a major contender for archival load. At the same time, deduplication's ability to dramatically reduce storage footprint further improves storage utilization.

This paper proposes DedupT, a deduplication-based tape system. To the best of our knowledge, this is the first work to demonstrate that tape-based systems can fully benefit from the gains offered by deduplication without major penalties in terms of data retrieval time. DedupT addresses the main challenges for efficient data deduplication on tapes: high tape mount overhead and seek times. Its chunk placement algorithms are able to (*i*) preserve up to 95% of the deduplication efficiency while, at the same time, (*ii*) completely eliminating major recovery time overheads, (*iii*) improving the performance of migrating data to tape pools by a factor proportional to the efficiency of data deduplication, (*iv*), reducing tape wear, and (v) offering the restore performance that is 31% to 40% better than that of the non-deduplicated tape .

Disclaimers and Notices

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States or other countries or both: IBM, Tivoli Storage Manager (TSM).

7. REFERENCES

[1] A. Osuna, R. Sharma, M. Silvestri and S. Wiedemann, "IBM System Storage Tape Library Guide for Open Systems," IBM Redbooks, pp 59, 84, 2011.

[2] D. Reine and M. Kahn, "In Search of the Long-Term Archiving Solution — Tape Delivers Significant TCO Advantage over Disk," Clipper Notes, 2010.

[3] R. Amatruda, "Worldwide Tape Drive 2011–2015 Forecast Update and 2010 Vendor Shares," IDC Report, 2011.

[4] Google, "http://gmailblog.blogspot.com/2011/02/gmail-back-soon-for-everyone.html," Official Gamil Blog, 2010.

[5] D. Bhagwat, K. Eshghi, D. D. E. Long and M. Lillibridge, "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," in IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS'09), 2009, pp. 1-9.

[6] K. Jin and E. L. Miller, "The effectiveness of deduplication on virtual machine disk images," in Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference, 2009, pp. 7.

[7] D. Meister and A. Brinkmann, "Multi-level comparison of data deduplication in a backup scenario," in Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference, 2009, pp. 8.

[8] Ultrium – LTO Technology, "http://www.lto-technology.com," 2011.

[9] D. Pease, A. Amir, L. Villa Real, B. Biskeborn, M. Richmond and A. Abe, "The linear tape file system," in IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10), 2010. pp. 1-8.

[10] CA ARCserve Backup, "http://www.arcserve.com/us/products/ca-arcserve-backup.aspx," 2011.

[11] EMC Avamar, "http://www.emc.com/collateral/software/data-sheet/h2568-emc-avamar-ds.pdf," 2011.

[12] IBM Tivoli, "http://www.ibm.com/software/tivoli/," 2011.

[13] D. Colarelli and D. Grunwald, "Massive arrays of idle disks for storage archives," in Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, 2002.

[14] R. C. Burns and D. D. E. Long, "Efficient distributed backup with delta compression," in Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems, 1997.

[15] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality," in Proccedings of the 7th Conference on File and Storage Technologies, 2009.

[16] C. Policroniades and I. Pratt, "Alternatives for detecting redundancy in storage systems data," in Proceedings of the Annual Conference on USENIX Annual Technical Conference, 2004.

[17] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage," in Proceedings of the Conference on File and Storage Technologies, 2002.

[18] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," in FAST'11: Proceedings of the 9th Conference on File and Storage Technologies, 2011.

[19] F. Guo and P. Efstathopoulos, "Building a high-performance deduplication system," in Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference, 2011.

[20] B. Zhu, K. Li and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in Proceedings of the 6th USENIX Conference on File and Storage Technologies, 2008.

[21] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu and M. Welnicki, "Hydrastor: A scalable secondary storage," in Proccedings of the 7th Conference on File and Storage Technologies, 2009.

[22] M. Newman, Networks: An Introduction. Oxford University Press, 2010.

[23] V. Batagelj and M. Zaversnik, "An O (m) algorithm for cores decomposition of networks," ArXiv:Cs.DS/0310049, 2003.

[24] V. Batagelj and M. ZaverĹĄnik, "Generalized cores," ArXiv:Cs.DS/0202039, 2002.

[25] Knuth, Donald, "The Art of Computer Programming, Second Edition", pp 403-404,1998.