AN ABSTRACT OF THE THESIS OF

Raed H. Al-Rabeh for the degree of Master of Science in Electrical and Computer Engineering presented on September 27, 1990.

Title : Removing Inter-Minidisk Gaps on IBM's VM/CMS Operating System.

Redacted for Privacy

Abstract approved by : Prof. Roy Rathja

Disk space fragmentation is the proliferation of small and unusable gaps. This problem is considered within the context of IBM's VM operating system.

Facing this problem, researchers resorted to using algorithms based on memory management techniques, such as placement strategies (first-fit, best-fit, and worst-fit). Solutions based on these algorithms do not yield optimum results. Furthermore, in many cases, their efficiency and cost effectiveness are questionable.

This work proposes a new method to address the disk fragmentation problem using improved I/O techniques. Optimum results and efficiency are some of the qualities that contribute to the superiority of this approach. The algorithm was implemented and test figures were compared with calculated figures. The results clearly favor this new algorithm.

REMOVING INTER-MINIDISK GAPS ON IBM'S VM/CMS OPERATING SYSTEM

BY

RAED H. AL-RABEH

A THESIS
SUBMITTED TO

OREGON STATE UNIVERSITY

IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE
DEGREE OF

MASTER OF SCIENCE

COMPLETED SEPTEMBER 27, 1990

COMMENCEMENT JUNE 1991

APPROVED:

Redacted for Privacy

Associate Professor, Electrical and Computer Engineering in charge of major

Redacted for Privacy

Head of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented September 27, 1990.

Typed by Raed Al-Rabeh for Raed H. Al-Rabeh.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Removing Inter-Minidisk Gaps on IBM's VM/CMS Operating System

1. INTRODUCTION

THE CHALLENGE

During the past two decades, information technology has assumed an ever increasing role in the decision making process in government and private operations. Information processing today is a key component for increased productivity and competitiveness.

All computer installations today recognize their data as a valuable and irreplaceable asset. However, the volume and value of this data is growing exponentially. Therefore, the ability to store and manage this data effectively has become a complex and expensive challenge [1].

STORAGE MANAGEMENT TRENDS

To satisfy the goal of keeping data available while holding storage costs down, several strategies were recognized as essential.

1. Centralize and automate storage management functions.
2. Develop the role of a storage administrator.
3. Shift storage management responsibilities away from end users.

4. Frequently run utilities to extract and retrieve unused space.

However, the challenge of storage management is made even more complex with the development of real life problems, like the following [1,2].

1. The increasing complexity of computer systems.
2. The escalating cost of computer personnel.
3. A decline in service levels to users.
4. New application development is deferred due to lack of available disk space.
5. Current storage management techniques can't cope with the growth of data.
6. Continuous user demand for the system restricts the scheduled system maintenance time.

STORAGE MANAGEMENT IN VM

Essentially the VM operating system is geared toward the interactive end user. The user is given a contiguous space which the system treats like a disk. The user can format and store files in that 'virtual' disk. This architecture, although convenient from the user's point of view, presents non-trivial challenges to the system programmer. On one hand, disk storage is expensive if left to the appetite of end users; on the other hand, requesting disk drives at frequent intervals to satisfy demand is an expensive strategy.

Moreover, trained systems programmers are becoming more expensive as human costs increase while hardware costs decrease. Studies have shown that VM systems programmers spend approximately half their time managing disk space [2]. Therefore, any steps to reduce this burden will result in significant savings and increased productivity. Such tasks as moving user disks around to recover unused space are handled manually. To compound the problem, these tasks are handled in a crisis mode, such as when the system runs out of space, thereby increasing the chance for error and further distracting the systems programmer from vital development work [3].

GOALS AND GAINS

The aim of the work done in this thesis is to develop a method to systematically extract unused space between user disks in the VM operating system. The proposed solution will yield optimum results while keeping the cost of resource usage low. The ultimate goal is to increase system efficiency and improve disk space utilization. Several other gains also follow:

1. Improving system throughput by being able to process more users thus satisfying higher user service levels and demand.
2. Decreasing hardware operating costs by using less hardware. On mainframes this is significant since the maintenance bill for small

mainframes starts in the range of tens of thousands of dollars per year.

3. Deferring the need to purchase more hardware to satisfy the normal increase in user and system demand.
4. Reducing the human cost by decreasing the work load on systems programmers thus providing more time for vital development work.

OVERVIEW OF THE FOLLOWING CHAPTERS

The thesis is divided into five chapters. The following is a brief description of their contents.

In chapter 2, necessary background is presented. The IBM System/370 architecture is briefly overviewed. Basic CPU design, I/O subsystem mechanics, and disk storage subsystem layout is presented. Then, an overview of the VM operating system itself is included. In that overview major concepts, components, and tools are discussed.

In chapter 3, the fragmentation problem is formulated and the criteria for judging any solution are stated. Different placement strategies are considered and critiqued.

In chapter 4, the proposed solution is presented. Input/output enhancements are suggested and discussed. My proposed algorithm is examined and a working implementation of it is included.

Chapter 5 summarizes the problem and the different solution ideas. It emphasizes the superiority of the proposed solution as suggested by the performance criteria. It also indicates the potential of my solution to be used in several other applications that are I/O bound without fear of being rendered obsolete in a short period of time.

2. BACKGROUND

In this chapter, the basic IBM System/370 architecture and the main VM components are discussed. The CPU, I/O, and disk storage characteristics will be examined. Then the virtual machine concept, as implemented by the different components of VM, is introduced. Finally, a brief description of REXX, the language used for systems programming in VM, is presented.

IBM SYSTEM/370 MACHINE ARCHITECTURE

The basic architecture for IBM System/370 is a 32 bit design with 16 general purpose registers, 8 floating point registers, 16 control registers, and 16 access registers. All registers are 32 bits in size except the floating point ones which are 64 bits [4]. It is an interrupt driven design with 6 types of interrupts [4,6].

1. SVC (supervisor call) interrupts. These interrupts are initiated by a running process when the process generates a request for a particular system service such as performing I/O and obtaining storage. This mechanism secures the system from the users and allows the system to check privileges before granting a service to a user.
2. Input/output interrupts. These interrupts are initiated by the I/O subsystem to signal the completion of an I/O operation, the

occurrence of an I/O error, or that a device is made ready.

3. External interrupts. These interrupts can be caused by either the expiration of a time slice (used to implement time sharing), an interrupt from the operator, or from another processor in a multi-processor system.
4. Restart interrupts. These interrupts are forced by the operator or another processor on a multi-processor system.
5. Program check interrupts. These interrupts are caused by various types of errors experienced by a running process such as an attempt to divide by zero or an attempt by a user to use storage that does not belong to the user.
6. Machine check interrupts. These interrupts are caused by malfunctioning hardware.

The input/output subsystem has three components: channels (I/O processors), controllers that attach to channels (perform caching and I/O optimization), and I/O devices attached to the controllers [4]. Fig. 1 shows the general architecture for IBM/370 systems.

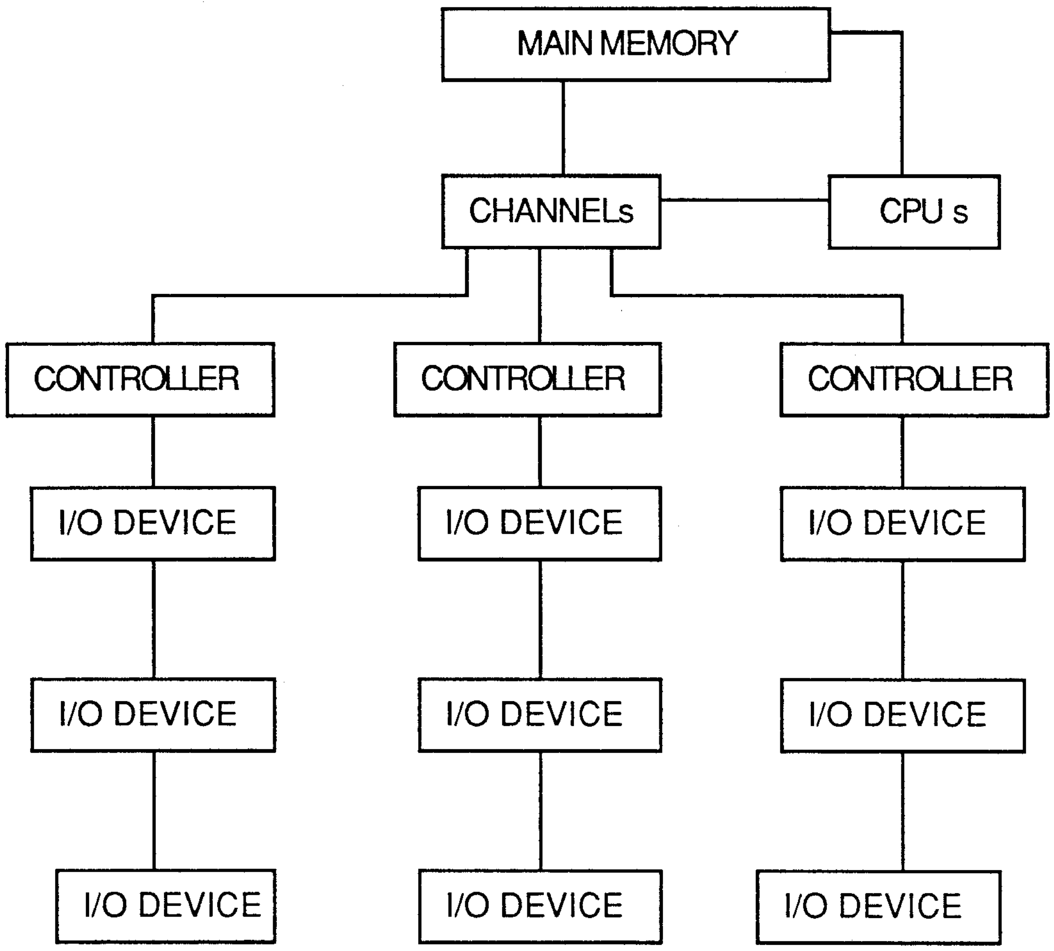


Fig. 1 The general architecture for IBM/370 systems.

The data transfer speed between the channels and controllers is 4.5 megabytes per second while the transfer rate between the controller and the I/O device depends on the I/O device. Relevant performance data for disks and cartridge tapes can be found in table 1 [4,5].

Table 1. The perfomace data for disks and cartridge tapes.

Device	Transfer Speed	Capacity	Avg. seek time	Rotational delay
Disk volume	3MB/sec	519MB	12 msec	8.3 msec
Cartridge tape	3MB/sec	199MB	—	—

To perform I/O the operating system has to build a channel program and tell the I/O subsystem to execute it through the SIO (start I/O) command. The I/O subsystem is responsible for scheduling the I/O by determining the path to be taken and queuing the request on the chosen channel. The I/O program is a chain of channel command words (CCWs) which are instructions to the channel, controller and device. The program describes the location of the data, its characteristics and the required operation to be performed [4,5,9]. There are many advantages to this approach.

1. The CPU only passes the appropriate instructions to the I/O subsystem. When the I/O operation ends it receives notification through an I/O interrupt.
2. A channel can pass the orders to the controller and then it can service another I/O request while the device is searching for the data or the controller is staging it.
3. The controller can attend to another device while the first device is searching for the data (during the seek and rotational

device is searching for the data (during the seek and rotational delay).

4. The I/O subsystem is responsible for selecting an appropriate path (channel-controller-device) lies entirely on the I/O subsystem. Furthermore, error checking and data transfer from device to memory (or vice versa) is also done by the I/O subsystem.

A disk pack, conceptually illustrated in fig. 2, is a stack of metal platters that are coated with a metal oxide material. Data is recorded on both sides of the platters. The current standard IBM disk packs have 15 recording surfaces each [5].

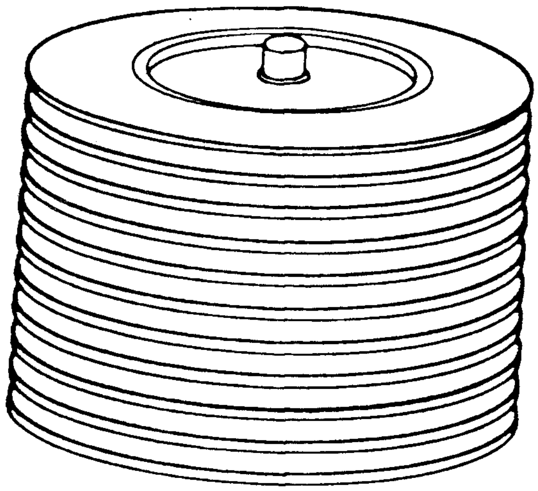


Fig. 2 Conceptual illustation of a disk pack.

Data is stored in concentric circles, called tracks, on each recording surface of a disk pack. This is illustrated in fig. 3. The current standard IBM disks have 885 tracks on each surface [5].

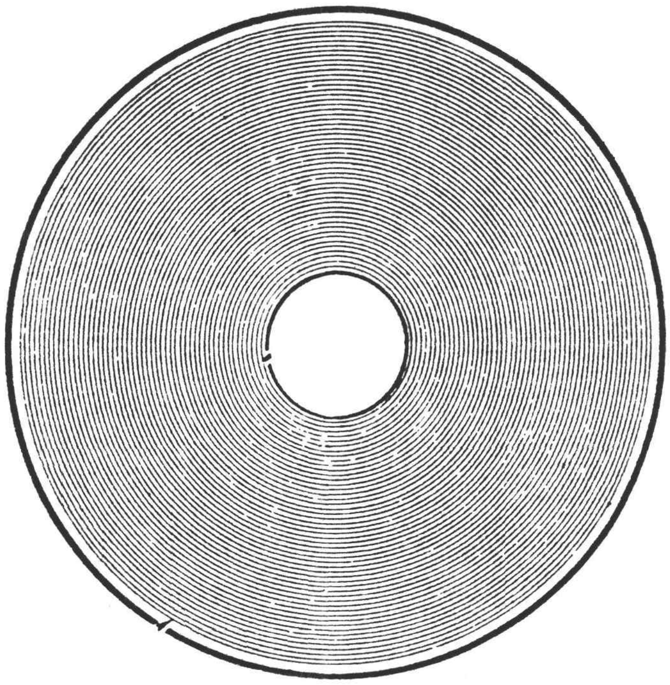


Fig. 3 Conceptual illustration of the tracks on a disk surface.

The data stored on a track is read by the disk's access mechanism, which is an assembly that has one read/write head for each recording surface. This is illustrated in fig. 4, which shows a side view of an access mechanism. The access mechanism moves all the heads at the same time the same distance. Hence, the access mechanism is positioned over the same track on all recording surfaces at the same time. As a result all these tracks can be operated on without moving the access mechanism. Although there are multiple read/write heads on each access arm, only one head transfers data at any given time. The tracks that can be accessed by a single positioning of the access mechanism make up a cylinder.

Consequently, there are as many cylinders on a disk pack as there are tracks on a single recording surface, namely 885. Therefore, it is faster to switch electronically from one head to another (to read/write tracks from one cylinder) than it is to move the heads mechanically to read/write tracks from different cylinders [5].

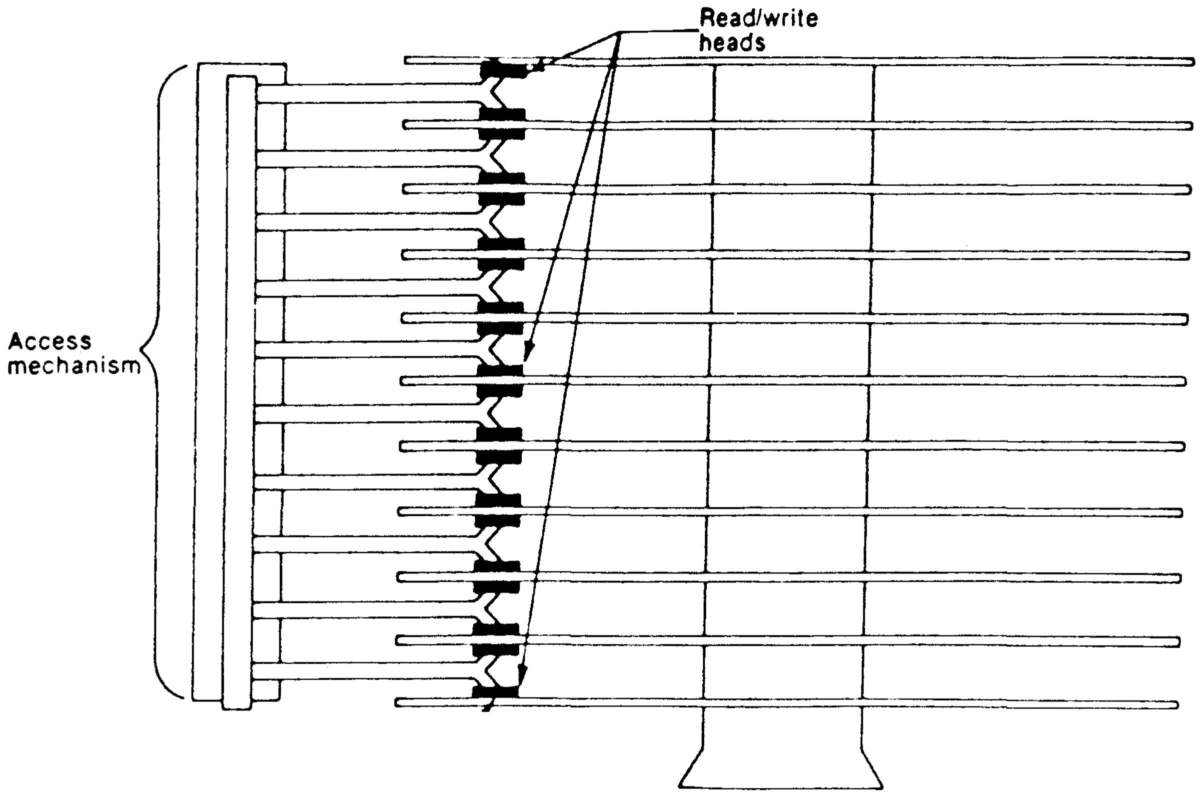


Fig. 4 Conceptual illustration of a side view of an access mechanism.

THE VIRTUAL CONCEPT

VM (Virtual Machine/System Product) is an interactive, multi-user operating system that was developed by IBM. The main idea behind this system is that although many people can share the use of a VM system, each person seems to have exclusive use of that system's resources. In fact, VM tries to make sure each user gets a fair share of system resources. This concept is applied to input and output devices as well as processor functions and storage [8].

The system simulates actual resources in such a way that it is like having a separate and complete system available to each of its users. Each user has a simulated (virtual hardware) machine exclusively for his own use, thus the name Virtual Machine/System Product. The user can add or remove resources to his virtual machine to customize it according to the needs. Virtual Machine resources will consist normally of a real terminal (virtual console), a processor (a time share on the real processor), direct access virtual storage devices (disks), virtual unit record I/O devices (reader, punch, and printer), and an operating system. The underlying goal is to give each virtual machine user the access to a share of each system resource [9].

VM has two major components, the control program (CP) and the Conversational Monitor System (CMS).

THE CONTROL PROGRAM

The Control Program (CP) manages system resources, i.e. it controls the real physical machine. CP is the one part of VM that is always required for system use. It is the component responsible for creating and simulating the virtual hardware required by a virtual machine work environment. CP controls the resources available to the user during a work session. However, to some extent, CP lets the user manage the portion of system resources it assigns to that user. In other words, CP manages one physical system so that it seems to give each of the users a separate and independent virtual system [8,10].

The functions of CP also include [10]:

1. Creating, deleting, and modifying virtual machines.
2. Monitor and record resource usage.
3. Maintain system integrity and security.
4. Process virtual machine requests to use input or output devices.
5. Maintain system queues and tables needed to manage system processors, storage, and input/output devices.

However, most of the user application work done on VM requires the use of an operating system in the working environment (the virtual machine) to help with data processing tasks and to manage work flow.

Some of the operating systems that can be run in a virtual machine are [8,9]:

1. The Conversational Monitor System (CMS)
2. Virtual Segments Extended/System Product (VSE/SP)
3. Multi Virtual Segments/System Product (MVS/SP)
4. UNIX/IX
5. Disk Operating System (DOS)
6. Virtual Machine/System Product (VM)
7. Any user created system that follows the standard interfaces for the IBM System/370 architecture.

CMS is the operating system most frequently run in a virtual machine and is supplied with CP in the VM package by IBM.

THE CONVERSATIONAL MONITOR SYSTEM

The Conversational Monitor System (CMS), although a component of the VM operating system, is itself an operating system. However, it was designed to run only in conjunction with CP. It is a disk oriented single user operating system that gives the user complete access to what appears to be a dedicated real machine but in reality is a virtual machine created by CP [8,9].

All CMS commands available to the user are stored in disk files. This makes the tailoring of the instruction set of the virtual machine an easy task. CMS allows the user to do a wide variety of tasks including [8,11] :

1. Writing, testing, and debuging application or system programs.
2. Creating, deleting, and editing files.
3. Executing application programs.
4. Processing jobs in batch mode.
5. Managing the virtual machine's working environment.
6. Communicating with other users of the VM system.

THE SYSTEM DIRECTORY

When a user logs on, CP uses the user's entry in the system directory to create the virtual machine and define its environment. The directory entry for a user includes information such as [9]:

1. The virtual machine's disks (called minidisks) locations and sizes.
2. Spool devices addresses and characteristics.
3. Classes of VM commands the user is authorized to issue.
4. Default and maximum virtual storage for each user.
5. Data to help account for system resources.
6. User priority for using system resources.
7. Passwords that allow the users to access the system.

The system directory is a file containing descriptions of all potential virtual machines allowed to run on that particular VM system. Therefore, the information in the system directory controls the access and the virtual machine's specification that each user can have [8].

MINIDISKS

A direct access storage device (DASD) volume may be dedicated to one virtual machine, or it can be shared among several virtual machines. A single DASD may be divided by CP into many minidisks, each of which may be allocated to a different virtual machine.

Minidisks are subsets of full disks. They each consist of a contiguous chunk of cylinders. Each is treated by a virtual machine as a complete physical disk device but one of smaller capacity than a full disk. Minidisks can vary in size to accommodate user or system needs. However, the smallest size a minidisk can be is one cylinder, the largest is a full DASD volume, and it is allocated in multiples of cylinders [9].

CP handles the mapping of minidisks to their respective spaces on the real disks, but the space within each minidisk is managed by the operating system running on the virtual machine to which the minidisk is assigned. CP prevents a virtual machine from referencing space on a disk volume outside the boundaries of that machine's minidisks. Minidisks can belong to one virtual machine or can be shared. Virtual machines can dynamically gain access to a minidisk through the CP LINK command or drop it through the DETACH command. CP will pretend that a real disk device is being connected to the virtual machine and will assign an address to it [8,9].

VIRTUAL UNIT RECORD DEVICES

Unit record devices refer to readers, punches, and printers. VM simulates the functions of these devices by spooling (storing) the data on direct access storage until the data is ready to be processed. Therefore, each virtual machine possesses 'virtual' card reader(s), card punch(es), and printer(s). The output or input of these devices will be a spool file that resides in the system spool area on disk.

The readers and punches are commonly used to receive or send files among virtual machines. The virtual printer is responsible for putting a file to be printed in the required print format so that it can be spooled to the required physical printer. Any of those virtual devices of a virtual machine can be connected to a real device in which case it becomes a dedicated device [8,9].

THE REXX LANGUAGE

The Restructured Extended Executor (REXX) language has been recently adopted as the language of choice for implementing system programming tasks in the IBM mainframe operating systems environments. These tasks include command procedures, application front ends, user defined macros and subcommands, prototyping, and other system or application related tasks of similar nature.
The REXX language, in essence, is a general purpose programming

language highly adapted to suit systems programming needs. It is similar to C. It has the usual complement of structured programming constructs such as IF, DO, WHILE, UNTIL, SELECT, and so on. However, it was designed with an emphasis on ease of use. This point is emphasized in two areas [7].

1. No restrictions are imposed by the language on program format, input/output, variable types.
2. The availability of an extensive library of useful built-in functions.

A language that is designed to be easy to use must be adept at manipulating the kinds of symbolic objects that programmers normally use: words, numbers, names, lists, multi dimensional arrays, and so on. Most of the features in REXX are included to make this kind of symbolic manipulation easy.

Although REXX is also designed to be highly system independent, it has the capability of issuing both commands and conventional interlanguage calls to its host environment [16]. To illustrate this point, consider the area of writing or tailoring user commands. Command program interpreters are increasing in importance in modern operating systems. Nearly all operating systems include some form of SHELL, BAT, or EXEC languages. In many cases such a language is so embedded in the operating system that it is unlikely to be of use outside its primary environment. There is, however, a clear trend toward providing command programming languages that are both powerful and capable of more general usage [7].

REXX carries this principle further by being a language designed primarily for generality but also for suitability as a command programming language [16].

3. ANALYSIS OF SOLUTIONS

PROBLEM STATEMENT

Initially, users' minidisks are carved in sequence on a disk volume. When a minidisk is deleted, as is the case when a user is removed from the system's directory or when a minidisk is returned to the available space pool, a gap will result. As time passes, more of these gaps with different sizes and locations will appear (fragmentation). To reduce fragmentation and to be able to satisfy larger requests, minidisks have to be moved on the disk volume to consolidate the gaps to as few as possible. How to move minidisks around to achieve the objective is the question to be addressed here.

As mentioned earlier, any solution to the disk compaction problem on the IBM VM operating system has to be judged against several criteria.

1. Optimum performance.
2 Speed, efficiency, and low usage cost.
3. Minimize systems programmer's intervention.

At first glance some of the criteria appear to be conflicting. Historically, it is believed that to gain speed either efficiency or cost have to be compromised. Moreover, in some instances, a less than optimal solution may greatly enhance the speed and efficiency. This is clearly demonstrated with the heuristic algorithms type of solutions.

However, in this case, an optimum solution is sought. Furthermore, the solution will have the merits of speed, efficiency, and low cost.

USING PLACEMENT STRATEGIES

The traditionally adopted approach to the disk fragmentation problem is to use memory management strategies. In my analysis the focus will be on placement strategies as they represent a good example of memory management methods. However, the drawbacks and limitations of solutions based on placement strategies are common to all solutions based on memory management strategies. Therefore, arguments developed here hold true for the whole spectrum of memory management strategies. There are three possible placement strategies.

1. The first-fit approach. This strategy states that to satisfy a request for a size m simply choose the first area encountered that is of size greater than or equal to m.
2. The best-fit approach. According to this strategy, for a request of size m all available slots will be looked at. The smallest slot that is of size greater than or equal to m will be chosen
3. The worst-fit approach. This is exactly the opposite of the best-fit approach. All slots will be looked at and the largest slot with size greater than or equal to m will be chosen.

The worst-fit approach is the least appropriate in this case for two reasons. The first is that it deliberately leads to the exhaustion of large slots at the beginning thereby limiting the ability to satisfy large requests early in the process. The second reason is that it does not lead to significant reduction in fragmentation and it is also slow.

Both the best-fit and the first-fit strategies have their merits depending on the circumstances. The best-fit method has a natural appeal. It appears to be a good policy since it saves the larger available areas for a later time when they might be needed. But several draw backs to using the best-fit method can be sighted. First, it is rather slow, since it involves a fairly long search; if best-fit is not substantially better that first-fit for other reasons, the extra search time can't be justified. More importantly, the best-fit method tends to increase the number of very small blocks, and proliferation of small blocks is undesirable and contributes more to the problem of fragmentation [4].

However, as good as the first-fit method may appear to be, it certainly almost never leads to an optimum solution. There will usually be gaps left. Furthermore, the first-fit method is not the optimum strategy if satisfying larger requests at a later stage is an important factor.

CRITIQUE OF PLACEMENT STRATEGIES

The key to the solution lies in the answer to the question: What is the best strategy to move minidisks on a disk volume to eliminate the gaps? Placement strategies, and memory management schemes in general, do suggest solutions; but, their solutions suffer from serious drawbacks.

1. They will not yield an optimum solution since placement methods will not lead to the total elimination of gaps.
2. They involve a trade off between speed and good results, which in any case requires intelligence in implemetation.
3. If all the gaps were small while the minidisks on the disk volume were large, no minidisk movement is possible thus producing total failure to recover anything in this case.
4. More importantly, using the standard IBM I/O routines, e.g. the CMS copy command, for moving any sizable amount of data will be a very slow process, as will be demonstrated later with an example. These utilities poorly handle the task at hand, namely moving potentially a large number of minidisks.
5. Since the task requires moving users' minidisks, it would have to be done in a scheduled maintenance time and this time is always limited.

MY APPROACH TO THE SOLUTION

Since scheduled maintenance time is always limited, I feel the key issue here is speed. A simple analysis will lead to the conclusion that to attain a good result a significant improvement will have to be made in the way the I/O operations take place. This is logical since the I/O operations constitute the bulk of the overhead and delay. Any significant improvement in this area will have a considerable effect.

To further illustrate the point consider the example where half a disk volume, 444 cylinders worth of minidisks [5], have to be moved to consolidate the gaps. The following describe the situation.

1. CMS works on a maximum size of 4K blocks, as in the COPY command. Therefore, a SIO command will be issued for each 4K to be transferred [12].
2. A cylinder is 600K in size [5]. Hence, the amount to be moved would be 444 * 600 = 266400K
3. The total number of SIO commands needed to transfer the data would be 266400 / 4 = 66600 SIOs
4. The equation to calculate the time incurred by a SIO command is [4,5]:

   SIO time = time to schedule I/O + I/O queue wait time +
   seek time + rotational delay time + transfer time

   But, I/O queue wait time can be ignored since this operation is being done during maintenance time. Thus, the SIO equation becomes:

SIO time = time to schedule I/O + seek time + transfer time +
rotational delay time

If actual values are substituted then the equation becomes [4,5]:

SIO time = 1 msec + 12 msec + 8.3 msec + 1.3 msec
= 22.6 msec

Total SIO time = 22.6 * 66600 = 1505.2 seconds (25.1 minutes)

On the other hand, my approach to the problem is to try to transfer as large a block as physically possible with each SIO command. The physical barrier was 600K, one cylinder. The reason is because while the access mechanism is positioned over one cylinder, all tracks belonging to that cylinder are accessible just by electronically switching the read/write head that is transferring. Thus, with each SIO command a total of one cylinder can be moved. Hence, the numbers of the above example become as follows [4,5].

SIO time = 1 msec + 12 msec + 8.3 msec + 195.3 msec
= 216.6 msec

Total SIO time = 216.6 * 444 = 96.2 seconds (1.6 minutes)

With such big improvement possible, 15.7:1 to be exact, new ideas for solving the fragmentation problem become feasible. The simplest and most effective is to dump the minidisks to be moved to a tape then load them back stacked one after the other, thereby eliminating gaps completely. The merits of this idea stem from the following points.

1. With the possibility of fast I/O it will not take much time to dump and restore the minidisks to their new positions on the disk volume. Again consider the 444 cylinders worth of minidisks used in the example above.

   Total time = 96.2 + 96.2 = 192.4 seconds (3.2 minutes)
2. Minimum processing overhead is incurred since no intelligence is required to figure out the placement strategy.
3. Simplicity in the logic of the placement strategy leads to robustness and ease of debuging.
4. Most important of all it leads to an optimum solution to the fragmentation problem, which no placement strategy discussed earlier was capable of achieving.

Dumping the minidisks to another disk volume and then reading them back is another way of implementing the solution. However, it is not practical nor cost effective for two reasons. First, the data transfer speeds for disk volumes and tape cartridges are the same. The second reason is that it would require a scratch disk volume to be used for the transfer. But in most computer installations disk space, in large quantities, is at a premium and it is unlikely to waste a disk volume as a scratch.

4. THE PROPOSED SOLUTION

INPUT/OUTPUT ENHANCEMENT

The key to the solution in my approach is to significantly cut down on the I/O overhead. To use the standard CMS copy software in conjunction with the CMS access methods obviously will not lead to the desired effect. Close coordination with CP and direct communication with the I/O subsystem, and eventually the I/O devices, proved very effective. Using the following I/O features and strategies it was possible to drive the I/O operation to the actual physical limit of the architecture.

1. In the I/O process, DIAGNOSE commands with code HEX'20' are issued. The DIAGNOSE command is a way to communicate with the I/O subsystem through CP. Through this command, the virtual machine can specify any valid CCW chain, i.e. a channel program, to be executed by the I/O subsystem. No I/O interrupts are reflected to the virtual machine thereby relieving it from interrupt processing. The DIAGNOSE instruction is complete only when all I/O commands in the specified CCW chain are finished. The CCWs will be processed first by CP, via modules DMKCCWTR and DMKGIOEX, in order to provide full virtual I/O in synchronous fashion. CP returns control to the virtual machine only after the operation is complete or a fatal

error condition is detected [13]. This technique of performing I/O has two advantages. The first is that it is fast and efficient because the I/O is done by CP and there is no interference from CMS since it has been taken out of the loop completely. The second is that good use will be made of the extensive built-in error recovery facilities available from CP.

2. Before the I/O starts, device type and features are checked through a DIAGNOSE command with HEX'24' issued to CP. DIAGNOSE code HEX'24' requests CP to provide the virtual machine with identifying and status information about the specified virtual device. CP will return information about the virtual device and the associated real device [13]. This device type and features verification allows the I/O operation to occur at the physical limit allowed by the architecture of the device.

3. The output of the I/O operation to the tape will have the format of variable unblocked records. The reason is that although the move is cylinder by cylinder, the output data will be in compact format achieved by compressing strings. The result is that it is not necessary to transfer a full cylinder in the I/O process making it even faster.

4. An EOF, end-of-file, marker is written to the tape following each dump of a minidisk. Furthermore, minidisks are loaded from tape in the order in which they were dumped. This strategy makes it a straight forward process to load the

minidisks in their new positions on the disk volume without having to rewind or fast forward through the tape for a required minidisk. This is important since data transfer speed for tape cartridges is the same as that of disk volumes, as indicated in table 1.

5. The I/O routines will produce a map of the moved minidisks showing the old and new locations. Moreover, there will be extensive informational, warning, and/or error messages concerning the steps performed. All this information will be output at address 00E, the default virtual printer. Having all the relevant information collected in one place makes it convenient for monitoring or debugging.

STEPS OF THE ALGORITHM

The following algorithm lists the logical steps needed to be taken to accomplish the defragmentation of a disk volume, or any part of it.

1. Obtain input parameters. The disk volume name, starting cylinder, and ending cylinder are obtained and verified.
2. Disable the system directory to prevent changes. Then, create a map of the area to be defragmented.
3. Adjust starting and ending values to a minidisk boundry and then create two lists, one for the minidisks to be moved and the

other for the minidisks that can't be moved. Unmovable minidisks represent system spool areas, paging areas, and other system related areas that require an IPL (initial program load) to complete their move.

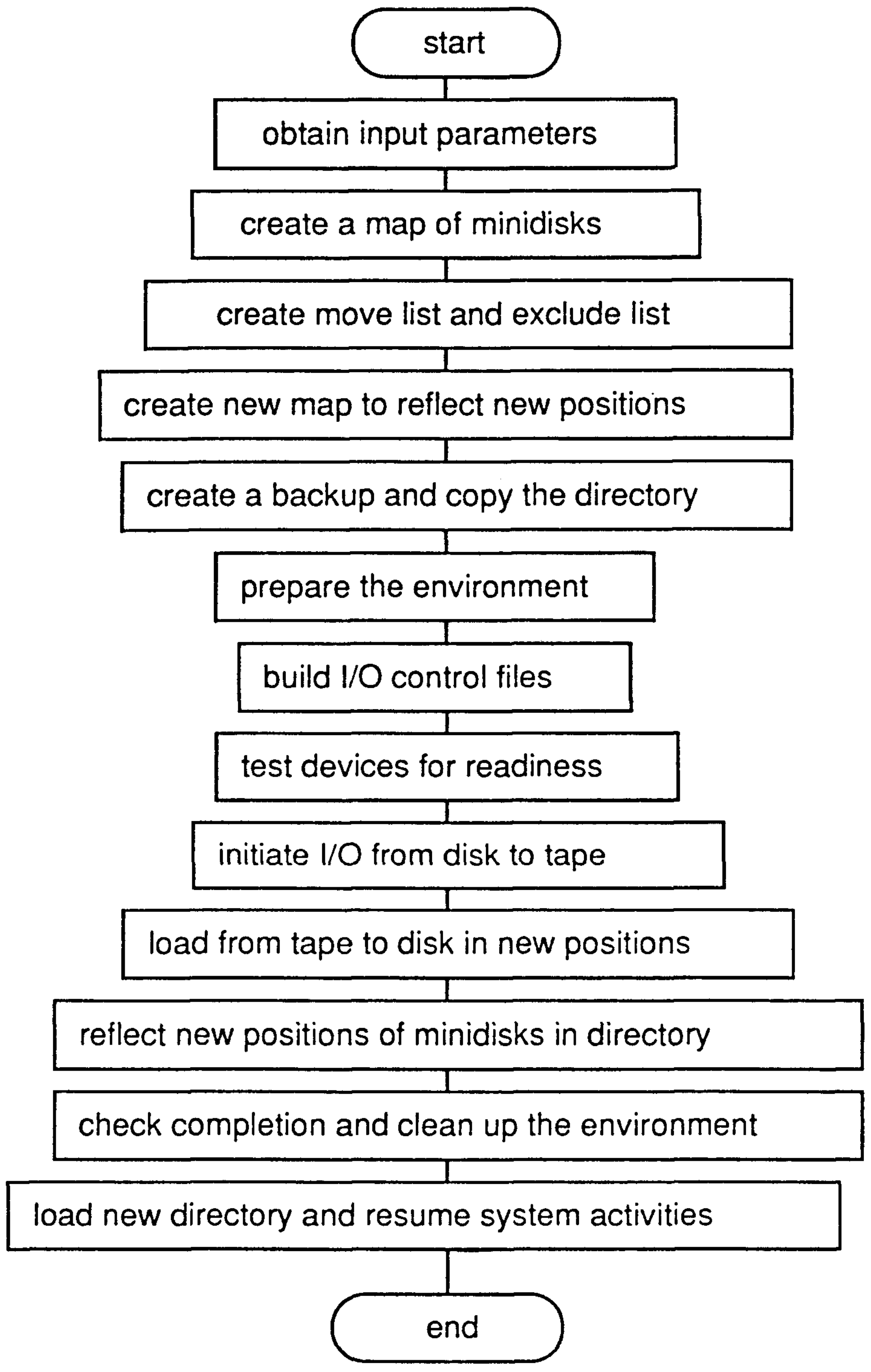
4. Create a new map which reflects the movement of minidisks to eliminate gaps.
5. Create a backup of the system directory then a copy of it.
6. Prepare the environment by establishing access to the disk volume to be operated on and quiescing the system to freeze any users' activity.
7. Build the necessary I/O control files that will direct the I/O operations. Three control files will be built, one for controlling the I/O from disk to tape, another will be used in the reverse process to load the minidisks in their new positions, and a third that may be used to roll back any changes made to the disk volume in case the whole operation is aborted after the load operation was done.
8. Set up the I/O paths and check the readiness of the physical devices, especially the attachment of a tape drive with a tape mounted.
9. Initiate the I/O process to dump minidisks from disk to tape.
10. Initiate the I/O process to restore minidisks to new positions according to the new map.
11. Update the copy of the system's directory, which was obtained

in step 5, to reflect the new positions of the minidisks which have moved.

12. Check completion codes, delete work files, and clean up the environment.
13. Load the system with the new directory and resume all suspended activities

FLOWCHART OF MY ALGORITHM

A flowchart of my algorithm is shown on the next page


start
obtain input parameters
create a map of minidisks
create move list and exclude list
create new map to reflect new positions
create a backup and copy the directory
prepare the environment
build I/O control files
test devices for readiness
initiate I/O from disk to tape
load from tape to disk in new positions
reflect new positions of minidisks in directory
check completion and clean up the environment
load new directory and resume system activities
end

Flowchart of my algorithm.

THE IMPLEMENTATION

The following is an implementaion of my algorithm using the standard IBM systems programming language, REXX.

```
/*****************************************************/
/* Module Name = SQUEEZE                             */
/*                                                   */
/* Module Type = REXX                                */
/*                                                   */
/* Function = Defragment a disk volume, or any part  */
/*            of it. Unused spaces between mdisks    */
/*            will be collected. Mdisks that can be  */
/*            moved will be moved to reduce or       */
/*            eliminate inter-mdisks gaps            */
/*                                                   */
/* Invocation = SQUEEZE volume start end (function   */
/*              Where function is either SIMULATE or */
/*            or SQUEEZE. For more details see       */
/*            HELP SQUEEZE.                          */
/*                                                   */
/* Output = MOVE    LIST   A  Old map of mdisks      */
/*          volname UPDATE A  New map of mdisks      */
/*          DUMP    CNTL   A  DDR Dump cntl          */
/*          RESTORE CNTL   A  DDR Restore cntl       */
/*          ABORT   CNTL   A  Restore volume cntl    */
/*          USER    INPUT     Updated Directory      */
/*          cons file on PRT  Simulation stats       */
/*                                                   */
/* Exit RC =   0  Successful completion              */
/*            40  Tape drive not ready on time       */
/*           100  QUIT entered by user               */
/*           200  DIRMAINT problem                   */
/*           300  File required not in reader        */
/*                                                   */
/* External Refrences =                                          */
/*    - EXCLUDE LIST A    List of mdisks that can                */
/*                        not be moved.                          */
/*                                                               */
/* Dependencies =                                                */
/*    - DIRMAINT must be up and enabled .                        */
/*     - Backup directory taken from DIRMAINT 193.               */
/*    - One free tape drive exists.                              */
/*    - Minimum 1 free cyliner on disk A.                        */
/*    - GETFMADR used to get free link address and               */
/*            mode before issuing CP LINK .                      */
/*                                                               */
/* Restrictions =                                                */
/*    - User ids must be upper case and mdisk                    */
/*       addresses must be 4 digits (leading zero)               */
/*     in EXCLUDE LIST A .                                       */
/*    - The machine running this exec must have a                */
/*       191 minidisk accessed as file mode A in                 */
/*       R/W mode. EXCLUDE LIST must be on A disk                */
/*                                                               */
/* Update History =                                              */
/*    - None.                                                    */
/*                                                               */
/*****************************************************************/

/*   T H E   I N P U T   S E C T I O N

Input: volume name, starting cylinder, ending cylinder,
function 'QUIT' is an optional answer to any prompt here.        */

do forever
  if arg() = 0  |  flag ^= 'FLAG'  then do
      say 'Enter function volume_name starting_cylinder',
        'ending_cylinder'
     say 'or enter "QUIT" to exit'
      parse upper pull function volname starting ending .
    end
  else do
      parse upper arg volname starting ending '(' function .
    end
    if function = 'QUIT' then exit 100
    say 'The function requested is ' function
    say 'The volume name is' volname 'starting at' starting
    say 'ending at' ending ', is this correct ? ',
      '(YES/NO/QUIT)'
   pull response
   if response = 'QUIT' then exit 100
   if response = 'YES' then leave
   flag = 1
end    /* end of input section */

/ *
   THE VOLUME MAP

If function is squeeze the directory will be disabled. Then
A map of the volume to be defragmented will be obtained.
All directory sevices are obtained through DIRMAINT.  */

address cms 'SET CMSTYPE HT'
address cms 'ERASE RESTORE CNTL A'
address cms 'ERASE ABORT CNTL A'
address cms 'ERASE DUMP CNTL A'
address cms 'ERASE' volname 'UPDATE A'
address cms 'ERASE USER INPUT A'
address cms 'ERASE MOVE LIST A'
address cms 'SET CMSTYPE RT'
'EXECIO 1 CP (VAR ANSR STRING QUERY FI *'
rdr_files.0  =  word(ansr,2)
address cms 'EXEC DIRMAINT USED'  volname
if rc ^= 0 then do
    say 'Error return code' rc 'from the command DIRMAINT',
      'USED' volname
    exit 200
end
if function = 'SQUEEZE' then address cms 'EXEC DIRMAINT',
   'DISABLE'
do I = 1 to 5
   sleep 4 sec
  'EXECIO 1 CP (VAR ANSR STRING QUERY FI *'
    rdr_files.I  =  word(ansr,2)
    if rdr_files.I ^= rdr_files.0 then leave
    if I = 5  &  rdr_files.I = rdr_files.0 then do
      say 'Error, DIRMAINT did not respond when expected ..'
      exit 200
  end
end
bufsize = 8192
Do Until cprc = 0
  'DESBUF'
  'EXECIO * CP (BUF' bufsize 'STEM RDR_LIST. STRING Q RDR',
      '* ALL'
   cprc = rc
   if cprc = 1 then bufsize = bufsize * 2
End
do J = 1 to rdr_list.0  while  file_no = 'FILE_NO'
    if word(rdr_list.j,10) = volname then ,
        file_no = word(rdr_list.J,2)
   if J = rdr_list.0  &  file_no = 'FILE_NO'  then do
     say 'Error, file' volname 'USEDEXT is not in RDR ...!'
     exit 300
  end
end
address cms 'SET CMSTYPE HT'
address cms 'RECEIVE' file_no '(REPLACE'
address cms 'SET CMSTYPE RT'
address cms 'DESBUF'

/* Create the exclude list and the move list. Adjust
     starting and ending to a mdisk boundry to prevent
     destroying a mdisk. The owner and address of volume
    will be found here.                                   */

'EXECIO 1 DISKW EXCLUDE LIST A (STR' userid() '0191'
total_used = 0
I = 0
'EXECIO * DISKR' volname 'USEDEXT'
do a = 1 to queued()
   pull line
   disk.1 = word(line,1)
   disk.2 = word(line,2)
```

```
    disk.5 = word(line,5)
    disk.6 = word(line,6)
    disk.7 = word(line,7)
    if disk.7 ^= volname then iterate
/*    next section is necessary to deal with full overlay
        mdisks, and will find the machine that owns
      the volume ...              */
   if  disk.5 = 0  &  disk.6 = 885  then do
      volume_owner = disk.1
      link_address = disk.2
      iterate
   end
   if disk.5 < starting & (disk.5+disk.6) > starting then
      starting = disk.5 + disk.6
    if disk.5 < ending & (disk.5+disk.6-1) > ending then
      ending = disk.5 - 1
   if disk.5 >= starting & disk.5+disk.6-1 <= ending then do
      total_used = total_used + disk.6
     notfound = 0
     realno = 0
      do until notfound
      'EXECIO * DISKR EXCLUDE LIST A' realno+1 ,
        '(LO /'disk.1'/'
        if rc = 0 then
          do
              pull . realno
              pull user mdisk
               if mdisk = '*' | disk.2 = mdisk then
               do
                  l = l + l
                   exclude.l.1 = disk.1
                   exclude.l.2 = disk.2
                   exclude.l.5 = disk.5
                   exclude.l.6 = disk.6
                   exclude.l.7 = disk.7
                  leave
              end
              else iterate
         end
        else
        do
          'EXECIO 1 DISKW MOVE LIST A 0 F 80 (STR' line
           notfound = 1
        end
    end
  end
end
queue 'BOT'
queue 'DELETE 1'
queue 'FILE'
address cms 'XEDIT EXCLUDE LIST A (NOPROF'

/*  The move list will be sorted in decending size
    sequance and then reread.                    */

'desbuf'
queue 'SORT * D 45 47'
queue 'FILE'
address cms 'XEDIT MOVE LIST A (NOPROFILE'
'EXECIO * DISKR MOVE LIST A'
j = queued()
mdisk_count = j
do b = 1 to queued()
  pull move.b.1 move.b.2 . . move.b.5 move.b.6 move.b.7
end

/*  Now a copy of the system directory will be obatained.
    If function is squeeze DIRM will be shutdown           */

address cms 'EXEC DIRMAINT USER BACKUP'
if function = 'SQUEEZE' then do
  address cms 'EXEC DIRMAINT SHUTDOWN'
  say ' *** <<    DIRMAINT IS NOW SHUTDOWN     >> ***'
end
'GETFMADR'
parse pull . fmode vaddr .
say '*** <<<  NEXT PROMPT IS FOR THE READ PASSWORD OF ',
   'DIRMAINT 193'
'LINK DIRMAINT 193 ' vaddr ' RR'
rc_link = rc
```

```
if rc_link ^= 0 then do
   say 'Link to DIRMAINT 193 mdisk failed, RC =' rc_link
    exit rc_link
end
address cms 'ACC ' vaddr fmode
address cms 'COPY USER BACKUP ' fmode ' = INPUT A (REPLACE'

/*    Establish a write link to volume by linking to its owner.    */

'GETFMADR'
parse pull . . voladdr .
'DESBUF'
say ' *** <<<  NEXT PROMPT IS FOR THE WRITE PASSWORD OF ',
   'VOLUME ' volname
address command 'CP LINK' volume_owner link_address ,
 voladder  ' W'
rc_link  =  rc
if rc_link ^= 0 then do
    say 'Link to 'volume_owner link_address 'failed, RC =',
        rc_link
   address cms 'REL' fmode '(DET'
    exit rc_link
end

/*   Now three DDR control files will be built.
     DUMP    CNTL A : to dump mdisks to tape
     RESTORE CNTL A : to restore mdisks to new positions
     ABORT   CNTL A : to return mdisks to old positions         */

xptr = 1
mptr = 1
flag = 1
new_start  =  starting
do while  mptr <= j
   if xptr <= l then
     do
         if move.mptr.6+new_start > exclude.xptr.5 then
        do
           do  smptr = mptr + 1  to  j
            flag = 1
```

```
                if move.smptr.done = 'YES' then iterate
                 if move.smptr.6+new_start <= ,
               exclude.xptr.5
              then do
                  ptr = smptr
                call BUILD_DDR
                 leave
             end
          end
            if  flag = 1  then do
                new_start = exclude.xptr.5 + ,
              exclude.xptr.6
              xptr = xptr + 1
          end
        end
      else
        do
            ptr = mptr
           flag = 1
          call BUILD_DDR
           do while move.mptr.done = 'YES'
              mptr = mptr + 1
          end
        end
    end
  else
    do
        ptr = mptr
       flag = 1
      call BUILD_DDR
       do while move.mptr.done = 'YES'
          mptr = mptr + 1
      end
    end
end

if function = 'SIMULATE' then signal DIRECTORY_UPDATE

/* Get A tape drive with scratch mounted and rewind */
```

```
say '*** <<<  QUIESCE SYSTEM NOW THEN HIT ENTER TO CONTINUE'
address command 'CP SLEEP'
raddr.1  =  '580'
raddr.2  =  '581'
found = 0
Do t = 1 to 2 Until found
  'EXECIO * CP (STRING Q SYS' raddr.t
   Pull . . status .
   If status = 'FREE' Then Do
      found = 1
     'EXECIO * CP (SKIP STRING ATT' raddr.t '*  181'
      attrc = rc
      If attrc = 0 Then Do
         Do r1 = 1 to 12 Until rewrc = 0
           'MSG OPERATOR Mount scratch tape on tape drive' raddr.t'.'
         say '*** <<<  PROGRAM IS WAITING FOR A SCRATCH ',
          TAPE ON' raddr.t
            Do r2 = 1 to 300 Until rewrc = 0
            'SLEEP 1 SEC'
            'EXECIO * CP (SKIP STRING REWIND 181'
               rewrc = rc
          End
       End
        If rewrc ^= 0 Then Do
          'MSG OPERATOR Mount timed out',
             'waiting for tape drive to be loaded.'
             call exitrc rewrc
       End
     End
     Else Do
         Do r1 = 1 to 12 Until rewrc = 0
           'MSG OPERATOR Attach and mount scratch on drive 'raddr.t'.'
          say '*** <<<   WAITING FOR ATTACH AND MOUNT ',
          'SCRATCH ON' raddr.t
            Do r2 = 1 to 300 Until rewrc = 0
            'SLEEP 1 SEC'
            'EXECIO * CP (SKIP STRING REWIND 181'
               rewrc = rc
          End
       End
```

```
         If rewrc ^= 0 Then Do
           'MSG OPERATOR Mount timed out',
              'waiting for tape drive to be attached and loaded.'
              call exitrc rewrc
        End
      End
    End
  End
  If ^found Then Do
    say '*** <<<  NO TAPE DRIVE AVAILABLE, CAN NOT CONTINUE ...'
     call EXITRC 600
  End

  /*    Start dumping mdisks     */

  address command 'CP SP PRINT * CLOSE CONT'
  address command 'DDR DUMP CNTL A'
  ddr_rc = rc
  if ddr_rc ^= 0 then do
     say '** << Non zero return code from DDR, RC = ' ddr_rc
     call EXITRC ddr_rc
  end

  /* Before loading mdisks in their new positions prompt
       the operator if he wishs to continue ...               */

  do until ansr = 'YES'
     say ' '
     say ' '
      say 'The next step will update the volume and system',
          'directory, '
     say 'do you wish to continue ?    (YES/QUIT)   '
      pull ansr
      if ansr = 'QUIT' then call EXITRC 100
  end

  /*  Reload mdisks in the new positions, and set up the
          update file for the system directory update
         operation        */
```

```
address command 'CP REW 181'
say '*** <<<  IF ONLY ONE TAPE WAS USED JUST HIT ENTER ,',
  'OTHERWISE '
say '** <<  MOUNT THE FIRST TAPE USED THEN HIT ENTER',
  'TO CONTINUE.'
address command 'CP SLEEP'
address command 'DDR RESTORE CNTL A'
ddr_rc = rc
if ddr_rc ^= 0 then do
   say '*** << Non zero return code from DDR, RC = ' ddr_rc
   call EXITRC ddr_rc
end
address command 'CP SP PRINT CLOSE'

/* Next step update the system directory from the stem MOVE. */

DIRECTORY_UPDATE:
do e = 1 to j
  'EXECIO * DISKR USER INPUT A' 1 '(FI /USER' move.e.1'/'
   pull . ru_no
   pull .
  'EXECIO * DISKR USER INPUT A ' ru_no ,
  '(LO /MDISK' move.e.2'/  MARGINS 1 72'
   rcode = rc
   if rcode = 0 then do
      pull . rm_no
      pull entry
       oldloc = word(entry,4)
      vname = word(entry,6)
  end
   if rcode ^= 0 | ^(vname=move.e.7 & old.e.5=oldloc) then
  do
      move.e.2 = substr(move.e.2,2,3)
    'EXECIO * DISKR USER INPUT A ' ru_no ,
     '(LO /MDISK' move.e.2'/  MARGINS 1 72'
      pull . rm_no
      pull entry
  end
    position = wordindex(entry,4)
    entry = delword(entry,4,1)
    entry = insert(move.e.5' ',entry,position-1)
   'EXECIO 1 DISKW USER INPUT A' rm_no '(STRING' entry
    update = move.e.1 || ' ' || subword(entry,1,6)
   'EXECIO 1 DISKW' volname 'UPDATE A 0 F 80 (STR' update
end
if function = 'SIMULATE' then signal SIMULATE
call EXITRC 0

/*   Clean up the invironment and exit with return code.     */

EXITRC:
arg  return_code
address cms 'SET CMSTYPE HT'
address cms 'ERASE' volname 'USEDEXT'
address cms 'ERASE' volname 'FREEXT'
address CMS 'RELEASE' fmode '(DET'
address command 'CP DET' voladdr
if function = 'SQUEEZE' then do
   address command 'CP DET 181'
    if return_code = 0 then do
     'SF USER INPUT A TO DIRMAINT'
      address cms 'VMFCLEAR'
      address cms 'SET CMSTYPE RT'
     say '***   THE NEW DIRECTORY IS IN DIRMAINTS READER'
      say '***   PLEASE INITLZ DIRMAINT AND ENABLE IT NOW'
      say '***          AND ISSUE A DIRM DIRECT          '
     say '***   PLEASE LOGOFF & LOGON DATAMOVE MACHINE'
   end
end
address cms 'ERASE USER INPUT A'
address cms 'SET CMSTYPE RT'
exit  return_code


BUILD_DDR:
queue 'INPUT' voladdr '3380' volname
queue 'OUTPUT 181 3420 (MODE 6250 LEAVE COMPACT'
queue 'DUMP' move.ptr.5 'TO' move.ptr.5 + move.ptr.6 - 1
queue ' '
queue ''
```

```
'EXECIO * DISKW DUMP CNTL A 0 F 80'

queue 'INPUT 181 3420 (MODE 6250 LEAVE COMPACT'
queue 'OUTPUT' voladdr '3380' volname
queue 'RESTORE' move.ptr.5 'TO' move.ptr.5 + move.ptr.6 -1
queue ' '
queue ''
'EXECIO * DISKW ABORT CNTL A 0 F 80'

queue 'INPUT 181 3420 (MODE 6250 LEAVE COMPACT'
queue 'OUTPUT' voladdr '3380' volname
queue 'RESTORE' move.ptr.5 'TO' move.ptr.5+move.ptr.6-1,
'REORDER' new_start
queue ' '
queue ''
'EXECIO * DISKW RESTORE CNTL A 0 F 80'
old.ptr.5 = move.ptr.5
move.ptr.5 = new_start
new_start = new_start + move.ptr.6
move.ptr.done = 'YES'
flag = 0
return


SIMULATE:
'EXECIO 1 CP (VAR ANSR STRING QUERY FI *'
rdrf = word(ansr,2)
address cms 'EXEC DIRMAINT FREE' volname
if rc ^= 0 then do
   say 'Error return code' rc 'from the command DIRMAINT',
     'FREE' volname
   exit 200
end
do I = 1 to 5
   sleep 4 sec
  'EXECIO 1 CP (VAR ANSR STRING QUERY FI *'
    rdr_files = word(ansr,2)
    if rdr_files ^= rdrf then leave
   if I = 5  &  rdr_files = rdrf then do
      say 'Error, DIRMAINT did not respond when expected .'
      exit 300
    end
  end
  bufsize = 8192
  Do Until cprc = 0
    'DESBUF'
    'EXECIO * CP (BUF' bufsize ,
    'STEM RDR_LIST. STRING Q RDR * ALL'
     cprc = rc
     if cprc = 1 then bufsize = bufsize * 2
  End
  file_no = 'FILE_NO'
  do J = 1 to rdr_list.0  while  file_no = 'FILE_NO'
      if word(rdr_list.j,10) = volname then ,
         file_no = word(rdr_list.J,2)
     if J = rdr_list.0  &  file_no = 'FILE_NO'  then do
        say 'Error, file' volname 'FREEXT is not in RDR ...!'
        exit 400
    end
  end
  address cms 'SET CMSTYPE HT'
  address cms 'RECEIVE' file_no '(REPLACE'
  address cms 'SET CMSTYPE RT'
  address cms 'DESBUF'
  'EXECIO * DISKR' volname 'FREEXT'
  pull .
  max_size = 0
  free_count = queued()
  do n = 1  to  free_count
     pull . . . . . size .
     if  size > max_size  then  max_size = size
  end
  total_cyls = ending - starting + 1
  free_cyls = total_cyls - total_used
  percent_free = free_cyls / total_cyls * 100
  percent_free = trunc(percent_free,1)
  address command 'CP SPOOL CON START'
  address cms 'VMFCLEAR'
  say 'Total number of cylinders          ===> ' total_cyls
  say 'Total number of cylinders used     ===> ' total_used
```

```
say 'Total number of free cylinders      ===> ' free_cyls
say 'Percentage of free cylinders        ===> %'percent_free
say 'Total number of free slots          ===> ' free_count
say 'The largest free slot found was    ===> ' max_size
say 'Total number of moved mdisks        ===> ' mdisk_count
address command 'CP SPOOL CON STOP CLOSE'
call EXITRC 0
```

IMPLEMENTATION ISSUES

Due to the nature of the problem, the solution handles many sensitive components of the system. The most sensitive component being the system directory. Due to the importance of the system directory, any updates made to it must be thoroughly tested and verified. Therefore, elaborate and extensive error checking is performed to insure the continued integrity of the system directory. Furthermore, a backup of the old directory is always kept in a protected file. If, for any reason, the updates to the directory were not successful, the system can always revert to the old directory with a simple command.

If, for any reason, after the completion of the movement of minidisks, the process was aborted and the old system directory was restored, then the minidisks also have to return to their old positions to match the description in the old directory. An I/O control file directs the I/O operations to achieve this goal and is always created as part of the process.

The implementation of my algorithm was tested and debuged for a period of two months on the IBM 4381 machine which is located in the university computer center. It is now being used in the university computer center on a production basis on an average of once a week. One disk volume has already been compacted. It had 40 gaps worth a total of 93.75 M bytes. After running the compaction routine, the result was that all the gaps were consolidated into one gap. A total of 51 minidisks, 180 cylinders or 105.5 M bytes worth, were moved. Total time consumed was 96.2 seconds. Theoretically, according to the formulas that were used in the calculation example the time should have been 78 seconds. The difference was due to the action of rewinding the tape after the end of the dump operation. However, this has a maximum limit of 21 seconds, which is the time it takes to completely rewind an empty tape.

5. CONCLUSION

Storing and managing data effectively is becoming increasingly complex and expensive. Faced with this challenge, several strategies were recognized as important; but, adhering to them is complicated by serious real life problems.

The IBM VM operating system is not isolated nor is it immune to these issues. The architecture of the system, with virtual machines and minidisks, presents unavoidable challenges. The challenge being targeted by the work done in this thesis is the disk fragmentation problem. A method is sought to systematically and efficiently extract unused spaces between minidisks. Two key apparently conflicting goals were targeted, optimum results and efficiency with low cost. Several gains, such as improved system throughput, also would follow.

To move minidisks to consolidate gaps, placement strategies (first-fit, best-fit, and worst-fit), present possible solutions. However, each one has serious drawbacks of its own together with shortcomings that are common to all of them. In any case, none of them answers the fragmentation problem with an optimum solution.

The approach taken by the solution presented in this thesis is to tackle the most fundamental obstacle, the I/O problem. The IBM/370 architecture itself has strong potential, but no software was found to be adequate if many or large minidisks need to be moved to consolidate the gaps. Improved I/O techniques and strategies achieved an improvement in the range of 15:1 in the I/O operations. This provided potential to ideas that were not feasible before. One such idea that was developed to its full potential is to dump the minidisks to tape and then load them back stacked one after the other. This solution not only ensured optimum results, but also exhibited the desired qualities of efficiency, low cost, and robustness.

The I/O techniques developed possess considerable potential. With little or no adaptation these techniques can be utilized in many I/O bound applications, such as databases and graphics applications. The basic IBM System/370 architecture has not changed for a considerable number of years. Even when changes were introduced, upward compatibility was always maintained. This means applications developed on older systems will run on new systems. As a result, The solution developed here will not become obsolete in a short period of time.

BIBLIOGRAPHY

[1]. W. G. Parker, "Corporate Data Storage Management for the 1990s," Technical Support, Vol. 3, No. 2, pp. 20-22, February 1989.

[2]. J. Kador, "The Challenge for VM," Technical Support, Vol. 2, No. 3, pp. 39-46, March 1988.

[3]. T. Margo, "DASD Management: A Look at the Limited Assistance Available in Native VM," VM Issues, Vol. 1, No. 3, pp. 1-4, June 1988.

[4]. IBM System/370 Principles of Operation, IBM Systems Library order number GA22-7000, IBM Corporation, September 1987.

[5]. IBM 3380 Direct Access Storage, IBM Storage Subsystem Library order number GC26-4491, IBM Corporation, September 1988.

[6]. H. M. Deitel, "An Introduction to Operating Systems," Addison Wesley, 1984.

[7]. M. F. Cowlishaw, "The Design of the REXX Language," IBM Systems Journal, Vol. 23, No. 4, pp. 326-335, 1984.

[8]. Introduction to VM/SP, IBM Systems Library order number GC19-6200, IBM Corporation, December 1986.

[9]. VM/SP Planning Guide and Reference, IBM Systems Library order number SC19-6201, IBM Corporation, December 1986.

[10]. CP General User Command Reference, IBM Systems Library order number SC19-6211, IBM Corporation, July 1988.

[11]. CMS User's Guide, IBM Systems Library order number SC19-6210, IBM Corporation, July 1988.

[12]. CMS Command Reference, IBM Systems Library order number SC19-6209, IBM Corporation, July 1988.

[13]. VM System's Facilities for Programming, IBM Systems Library order number SC24-5288-1, IBM Corporation, August 1988.

[14]. D. E. Knuth, "The Art of Computer Programming," Addison Wesley, 1973.

[15]. Application Development Reference for CMS, IBM Systems Library order number SC24-5284, IBM Corporation, July 1988.

[16]. VM/SP Interpreter User's Guide, IBM Systems Library order number SC24-5238, IBM Corporation, July 1988.

[17]. VM/SP Interpreter Reference, IBM Systems Library order number SC24-5239, IBM Corporation, July 1988.

[18]. VM/SP Editor User's Guide, IBM Systems Library order number SC24-5220, IBM Corporation, July 1988.

[19]. VM/SP Editor Command and Macro Reference, IBM Systems Library order number SC24-5221, IBM Corporation, July 1988.