UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* MASSACHUSETTS INSTITUTE OF TECHNOLOGY PROJECT MAC	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED
	2b. GROUP NONE

3. REPORT TITLE

STORAGE HIERARCHY SYSTEMS

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
INTERIM SCIENTIFIC REPORT

5. AUTHOR(S) *(First name, middle initial, last name)*

STUART E. MADNICK

6. REPORT DATE APRIL, 1973	7a. TOTAL NO. OF PAGES 155	7b. NO. OF REFS 90
8a. CONTRACT OR GRANT NO. N00014-70-A-0362-0006	9a. ORIGINATOR'S REPORT NUMBER(S) MAC TR-107	
b. PROJECT NO.		
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* NONE	
d.		

10. DISTRIBUTION STATEMENT

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

11. SUPPLEMENTARY NOTES PH.D. THESIS, DEPT. OF ELECTRICAL ENGINEERING, MAY 15, 1972	12. SPONSORING MILITARY ACTIVITY OFFICE OF NAVAL RESEARCH

13. ABSTRACT

The relationship between the page size, program behavior, and page fetch frequency in storage hierarchy systems is formalized and analyzed. It is proven that there exist cyclic program reference patterns that can cause page fetch frequency to increase significantly if the page size used is decreased (e.g., reduced by half). Furthermore, it is proven in Theorem 3 that the limit to this increase is a linear function of primary store size. Thus, for example, on a typical current-day paging system with a large primary store, the number of page fetches encountered during the execution of a program could increase 200-fold if the page size were reduced by half.

The concept of temporal locality versus spatial locality is postulated to explain the relationship between page size and program behavior in actual systems. This concept is used to develop a technique called the "tuple-coupling" approach.

Consistent with the results above and by generalizing conventional two-level storage systems, a design for a general multiple level storage hierarchy system is presented. Particular algorithms and implementation technqiues to be used are discussed.

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Storage Hierarchy						
Virtual Memory						
Dynamic Storage Allocation						
Operating Systems						
Paging						
Page Size						
Replacement Algorithms						
Computer Architecture						
Multi-level Memoires						
Spatial Locality						

STORAGE HIERARCHY SYSTEMS

Stuart E. Madnick

April 1973

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

STORAGE HIERARCHY SYSTEMS

by

STUART ELLIOT MADNICK

Submitted to the Department of Electrical Engineering on May 15, 1972, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ABSTRACT

The relationship between page size, program behavior, and page fetch frequency in storage hierarchy systems is formalized and analyzed. It is proven that there exist cyclic program reference patterns that can cause page fetch frequency to increase significantly if the page size used is decreased (e.g., reduced by half). Furthermore, it is proven in Theorem 3 that the limit to this increase is a linear function of primary store size. Thus, for example, on a typical current-day paging system with a large primary store, the number of page fetches encountered during the execution of a program could increase 200-fold if the page size were reduced by half.

The concept of temporal locality versus spatial locality is postulated to explain the relationship between page size and program behavior in actual systems. This concept is used to develop a technique called the "tuple-coupling" approach. It is proven in Theorem 5 that when used in conjunction with conventional hierarchical storage system replacement algorithms, tuple-coupling yields the benefits of smaller page sizes without the dangers of explosive page fetch activity.

Consistent with the results above and by generalizing conventional two-level storage systems, a design for a general multiple level storage hierarchy system is presented. Particular algorithms and implementation techniques to be used are discussed.

THESIS SUPERVISOR: John J. Donovan
TITLE: Associate Professor of Electrical Engineering

ACKNOWLEDGEMENT

This thesis describes research done at M.I.T. in conjunction with Project MAC and the Department of Electrical Engineering, both of whose support and encouragement are gratefully acknowledged. During the course of this research, I have been fortunate to have had the cooperation of the IBM Cambridge Scientific Center and the Honeywell Information Sciences Center.

I have also benefited greatly from conversations with numerous colleagues. In particular, special mention must be made of Professor J. H. Saltzer, Don Hatfield, Harold Schwenk, Paul Wood, Steve Zilles, Leonard Goodman, and Jerry Johnson, who provided receptive ears and were helpful in unraveling many of the important issues.

By agreeing to be Readers, presumably at a weak moment, Professor J. C. R. Licklider and Dr. Ugo Gagliardi were inflicted upon by this thesis and the associated chaos. I appreciate their cooperation and understanding and look forward to their assistance in the continuation of this research.

I am obligated to acknowledge Professor John Donovan as my supervisor for this thesis. But, during my studies as a graduate student student, he has been much more than just a thesis supervisor. He has been my teacher, advisor, colleague, and friend. His cooperation and assistance, not to mention his enthusiasm, have had a profound affect upon my research and I am truly grateful.

Most importantly, I thank my wife, Ethel, for struggling through and surviving the tortures of being married to a graduate student for countless years (I'm sure that she has counted them). The same commendations are extended to my sons, Howard and Michael. Unfortunately, I fear that the damage to them might be more permanent. I realized this recently when I learned that in response to a first-grade assignment to write about what you want to be when you grow up, Howard had written: "When I grow up I want to be a student like my daddy".

- - - - -

Work reported herein was supported, in part, by Project MAC, an M.I.T. research project sponsored by the Advance Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01).

CONTENTS

ILLUSTRATIONS

TABLES

MAJOR THEOREMS

CHAPTER 1.

INTRODUCTION AND PLAN OF THESIS

1.0 Introduction

The primary goal of this thesis is to provide insight into and shed additional light on several key problems in the design and analysis of general storage hierarchy systems.

1.1 Significance of Problem

The importance of research in storage hierarchy systems has been pointed out by Prof. F. J. Corbatò recently in the MIT Project MAC Progress Report VIII (July 1971):

> "By now, it has become accepted lore in the computer system field that use of automatic management algorithms for memory systems, constructed of several levels with different access times, can provide a significant simplification of programming effort. ... Unfortunately, behind the mask of acceptance hides a worrisome lack of knowledge behind how to engineer a multilevel memory system with appropriate algorithms which are matched to the load and hardware characteristics."

On multiple level storage hierarchies, Prof. J. H.

Saltzer was even more explicit (subject notes on "Information Systems", MIT, 1972, p. 4-58):

> "An interesting problem arises if one has three or more technologies to deal with. ... The problem of predicting the performance of a three level, automatically managed system is not at all well understood. ... Although the need for more than one level has already been argued, there is currently no known criterion for introducing three, four, or N levels for a given system. ... Although there are by now many implementations of two level memory systems, the dynamic management of a three or more level memory system is such an uncharted area that there do not yet exist examples of practical algorithms which one can examine."

1.2 Specific Goals and Accomplishments

The specific goals and acccmplishments of this thesis, which are further elaborated later, are:

- Analyze the affect of certain parameters, such as page size, upon the performance of a storage system.
- Develop a concept of locality based upon both spatial and temporal adjacency in address reference patterns that explains certain anomalies discovered in actual paging systems.
- Propose, formalize, and measure the performance of new "spatial-removal" storage management algorithms, in particular "tuple-coupling".
- Design a practical algorithm for effective

management of multiple level storage hierarchy systems and demonstrate its effectiveness under some simulated system loads.

1.3 General Structure of Thesis

The key plan of this thesis is to investigate several crucial problems and requirements of multiple level storage hierarchy systems. Particular areas are identified and corresponding theories developed and proven. A new and general design for storage hierarchy systems is also presented and evaluated. Finally, empirical measurements are presented to validate and calibrate the overall design and specific theoretical conjectures.

This thesis is organizationally divided into 8 chapters. The structure can be best introduced by outlining the content of the following chapters in the sections below.

1.3.1 Chapter 2: Motivation for Storage Hierarchy Systems

This chapter presents a perspective on the storage hierarchy problem and the motivation for such systems. It is primarily written for the benefit of people knowledgeable in the general computer field but who are not especially experienced in storage hierarchy systems. For the expert reader, this chapter exposes the biases and orientations of the author and thus sets the tone for the remainder of the thesis. This chapter also briefly reviews the history of research in storage systems and cites numerous references.

1.3.2 Chapter 3: Formalization of Storage Hierarchy Systems

A description and formalization of the basic characteristics of storage hierarchy systems is presented in this chapter. This is followed by a summary and critical analysis of research that directly relates to the specific goals of this thesis.

1.3.3 Chapter 4: A Storage Hierarchy System

In this chapter the key concepts of the proposed storage hierarchy system are presented and discussed. The principle and novel techniques are briefly described below:

1.3.3.1 Continuous Hierarchy

The ratio of performance between adjacent levels is kept moderate (e.g., a factor of 100 or less) to minimize discontinuities or awkward special-case algorithms. This is in contrast to many current systems with inter-level ratios of 1000 or more.

1.3.3.2 Shadow Storage and Page Splitting

Information is transferred in decreasing smaller size blocks as it is passed up from low performance levels of the hierarchy toward the "request generator" at the uppermost level. Thus, the information that is finally received by the request generator has left a "shadow" behind in the lower levels. The significance and rationale for this technique is further elaborated in Chapter 6.

1.3.3.3 Automatic Management

In order to reduce the load on the central processor and provide for more efficient and parallel operations, the storage management function will be distributed and incorporated into the storage levels (e.g., "intelligent" device controllers [1], etc.). This technique also reduces the complexity of the operating system software.

1.3.3.4 Direct Transfer

Storage transfers between two adjacent levels need not have any effect upon nor require the assistance of any other levels (e.g., there is no need to move information from level n to level 1 and then from level 1 to level n-1 if only level n to level n-1 was needed; this two step process is often required on contemporary systems). Direct transfer is accomplished by synchronizing non-mechanical storage devices or by using "rubber-band" buffers [33] between electro-mechanical storage devices.

1.3.3.5 Read Through

Storage transfers, as noted above, are only made between adjacent levels of the hierarchy, such as from level n to level n-1. But, each level, such as level n-1, can connect its input bus (from lower level n) to its output bus (to higher level n-2) so that the data can be <u>read through</u> (i.e., transferred to level n-2 while being stored in level n-1). A similar, though specialized, technique is already used in certain systems, such as the IBM System/370 Models 155 and 165 cache systems [52].

This results in performance similar to a direct connection from each level to the request generator but it provides much more control in the storage levels and a much simpler structure.

1.3.3.6 Store Behind

By using the excess capacity of the inter-level channels, there is a continual flow of altered data from the higher levels to the lowest level permanent storage. Thus, the actual updated information is <u>stored behind</u> (after) the store initiation from the request generator. The updated information is propagated down, level by level. Whenever information is altered at a particular level, it is tagged as altered and is scheduled for a "store behind" operation.

1.3.4 Chapter 5: Analysis of Page Size Considerations

One of the most important parameters of a storage hierarchy system is the page size chosen as the unit of transfer between two levels of the hierarchy. In this chapter, the factors influencing page size are examined from the device characteristics viewpoint and the program behavior viewpoint.

Of particular concern, it has been noticed by Hatfield [47] and Seligman [78] and formalized in Chapter 5 that:

> "There exists a page trace, P, and demand-fetch FIFO-removal or LRU-removal inter-level storage systems, S and S', with page sizes N and N'=N/2, respectively, such that the ratio, r, of fetch frequency f' to f exceeds 2."

This result runs counter to the hoped for behavior of decreased page sizes as noted by Denning [25]:

> " ... small pages permit a great deal of compression without loss of efficiency. Small page sizes will yield significant improvements in storage utilization ... "

In this chapter the significance of this problem is demonstrated by proving that even "well-behaved" removal algorithms, such as stack algorithms [63], are not immune to this adverse performance behavior. Furthermore, the nature of this phenomenon is analyzed and bounds on its behavior are developed.

1.3.5 Chapter 6: Spatial vs. Temporal Locality Model of Program Behavior

A primary rationale for hierarchical storage systems is based upon the "Principle of Locality". Unfortunately, this principle is still a poorly understood, or at least controversial, phenomenon. It is difficult to determine the original "discoverer" of this principle but it is interesting to note that its definition has changed in time. For example, Denning [29, p.3], in 1968 loosely described locality as:

> "the idea that a computation will, during an interval of time, favor a subset of the information available to it."

Later, in 1970, Denning [26, p.180] defined it more precisely based upon the concepts of "working set" and "reference density", which for a page i at time k:

$$a(i,k) = \Pr[\text{reference } r(k)=i],$$

such that R(k) is the ranking of all n pages based upon a(i,k); thus:

> "PRINCIPLE OF LOCALITY: The rankings R(k) are strict and the expected ranking lifetimes long."

This is a much more restrictive definition of locality than his earlier general concept.

In fact, many current storage management systems were devised first, a general model was then constructed to describe the system, and finally a "formal" definition of locality was developed to be consistent with the storage management model. This is a reasonable strategy as long as the underlying concepts of "the principle of locality" are not lost in the process. Unfortunately, this appears to have happened on several occasions. In particular, most popular definitions of locality tend to be useless for analyzing or explaining either the relationship of page size upon program behavior or the impact of generalizing from two-level storage systems to multiple level hierarchical storage systems.

In this chapter a new view of locality is presented (or an old-view resurrected since it most closely resembles some of the very early descriptions of locality). In particular, it is shown that the general concept of locality can be subdivided into two separate factors, _temporal locality_ and _spatial locality_. These concepts are defined and justified and then used to explain some peculiar phenomena ("anomalies") observed in actual two-level storage systems.

By means of address traces and storage system simplifications, the temporal and spatial locality behavior of actual programs is emperically measured. These results are used to reinforce and calibrate the storage hierarchy system design presented in Chapter 4.

1.3.6 Chapter 7: Spatial Removal Storage Management Algorithms

Various hierarchy storage management algorithms, such as fetch (e.g., demand-fetch) and temporal removal (e.g., first-in first-out (FIFO), least recently used (LRU), etc.) have been developed, primarily for two-level hierarchies. There appear to be no spatial removal algorithms described in the literature. Based upon Chapter 6, several spatial algorithms are proposed and analyzed.

It is also shown that some of the problems described in Chapter 5 can be solved by spatial removal algorithms. In particular, Hatfield [48] noted that:

"as yet we have been unable to prove that there is a replacement algorithm using only the past history of page requests which cannot generate more than twice the exceptions with half size pages."

In this chapter a new algorithm, named <u>tuple-coupling</u>, is presented. It is formally proven that it satisfies Hatfield's requirements above.

Furthermore, the operational behavior of tuple-coupling is analyzed by measuring the performance of actual programs.

1.3.7 Chapter 8: Discussion and Conclusions

In addition to a general summary of the significant aspects of the thesis, this chapter also outlines important areas for future research.

CHAPTER 2.
THE STORAGE HIERARCHY PROBLEM

2.0 Introduction

The evolution of computer systems has been marked by a continually increasing demand for faster, larger, and more economical storage facilities. In addition to the obvious concern for better performance, the organization of a computer system's storage plays a key role in program development and programmer efficiency. It has often been claimed that "any software design blunder can be overcome by adding more memory".

It has become generally recognized that the conflicting requirements of high-performance yet low-cost storage may be best satisfied by a mixture of technologies combing expensive high-performance devices with inexpensive lower-performance devices. This strategy has been given several names, such as "hierarchical storage system", "automatic multilevel storage management", "virtual memory", and the inevitable "virtual memory system for the automatic multilevel management of a hierarchy of storage devices". In this thesis the somewhat shorter term storage hierarchy system will be used.

Investigations into automated storage hierarchy techniques can be traced back more than a decade. If we were to include manual techniques, we would find storage hierarchies at the very dawn of the "computer age". Unfortunately, there are still many unsolved and poorly understood problems. This situation can be partly explained by the fact that these systems tend to be (1) extremely complex, (2) ill-suited to most conventional analytical techniques, and (3) deeply influenced by the rapidly evolving computer technology which keeps "changing the ground rules" at often frightening rates. In spite of these challenging stumbling blocks, a successful storage hierarchy system is so important to the future usefulness of computer systems that we cannot afford to abandon the search.

2.1 Storage Hierarchy Objectives

Before delving into details, it is worthwhile to briefly consider the needs and uses for an effective storage hierarchy.

2.1.1 System Performance and Economics

As logic technology and computer architecture techniques have advanced, we have found it possible to produce systems of incredible speed. Such systems are often rated, rather crudely, in terms of MIPS (millions of instructions per second). Experimental system of over 100 MIPS have been developed (e.g., ILLIAC IV and CDC STAR). Even "conventional" large-scale systems have passed the 5 or 10 MIPS mark (e.g., CDC 7600 and IBM 370/195). It has long been observed that the input/output (I/O) requirements, especially for "secondary storage", of a conventional system tend to be strongly related to the processor's speed. In fact, based upon several empirical measurements, it has been postulated that a computer system averages 1 bit of I/O for every instruction executed (this is often referred to as Amdahl's Constant [ref]). As a result, many of these high-performance systems have been confronted with massive bottleneck problems in the I/O area, especially since these I/O demands tend to occur in bursts. An effective storage hierarchy system could go a long way toward reducing this problem.

At the other end of the spectrum we find that medium- and low-cost processors, the latter are usually called <u>mini-computers</u>, have made substantial advances in recent years. The term "mini" can be quite misleading. These processors are typically hundreds of times faster than the early commercial computers at a fraction of the cost (e.g., the UNIVAC I, circa 1951, could perform about 2000 12-digit additions per second whereas contemporary mini-computers operate at around 1,000,000 5-digit additions per second). Although these mini-processors may be midgets compared to the computational problems attacked by their "big brothers" described above, they are more than adequate for the vast majority of infomation processing problems which have modest computational requirements. Due to technological advances and economies of scale resulting from large-scale production, some minicomputers are available for less than $2000 with slightly slower micro-computers being offered for as little as $66 [18]. In spite of these technological advances, these processors have not had much impact on most information system needs due to the continuing economic problem of producing large capacity inexpensive storage devices even at the modest performance required. A $66 processor is largely irrelevant if the storage costs are in the $100,000 or more range. By developing an effective storage hierarchy system, we can go a long way toward bringing the storage costs down to the level of these inexpensive processors. As a result, a tremendous number of currently known technical solutions to information processing problems will finally become economically feasible solutions.

2.1.2 Simplify and Automate Programming

As noted earlier, the organization of a computer's storage system has a considerable impact upon program development and programmer efficiency. To a large extent, this potential increase in productivity is obtained by reducing or eliminating constraints normally imposed by the storage system. These constraints often distract the programmer to the extent that he devotes a substantial amount of his time to overcoming the system's limitations rather than solving the intrinsic problems. Shooman [80] noted that:

> "The inherent error content of some programs is claimed to be related to the excess memory capacity available. The theory here is that if the memory is very cramped, the software writers will have to resort to overlays and other coding "tricks" to squeeze the desired functions into the allocated memory space. It is assumed that these tricks introduce great complexity and are the seat of many errors. This effect is cited by designers of airborne computers where the allocation of another block of 4k of memory is a major design decision."

For example, the programmer often has to worry about:

2.1.2.1 Programming language code efficiency.

If a higher-level language compiler tends to produce programs that are at all larger than those produced by a low-level language translator, it may be necessary to use the low-level language to conserve storage. This constraint is contrary to the generally accepted fact that high-level languages enhance programming productivity.

2.1.2.2 Program size.

For any specific storage size, there are programs that cannot be easily written to fit into that size constraint. Yet, programmers frequently try - with considerable effort.

2.1.2.3 Data structures.

The programmer is often faced with the need to choose between a data structure representation that is convenient to use and another representation that "saves storage". This saving may require the use of an awkward or unnecessarily complex data structure representation.

2.1.2.4 Specific equipment characteristics.

If the programmer must get the "most" out of his storage system in terms of capacity and performance, he may resort to techniques that are peculiar to his specific storage system equipment. If the equipment is changed, there may be a considerable impact upon his software.

We would like to develop storage hierarchy techniques that eliminate, automate, or at least minimize the programming problems described above.

2.1.3 Integrate New Technologies and Applications

Although there has been continual evolution, the basic storage device technologies in commercial use have not changed dramatically in the past decade. As a result, there has been a tendency, motivated by actual need, to relate applications to the specific available technologies. This has caused certain application areas to be abandoned as "infeasible" and many storage management strategies to be discredited as "irrelevant" or "inefficient". In the passage of time we remember the applications and techniques in use but frequently forget or ignore the alternatives possible and the reasons for bypassing these alternatives.

After this rather long "rest", it appears that we are on the verge of some major "awakenings" in applications and technology. It is hard to quantify the new application needs other than requiring more and faster storage for less money. Section 1.1.2 presents some of these motivations, the revitalized interests in time-sharing, artificial intelligence and automatic programming are also "fanning the fire".

Due to the uncertainty of advanced research in storage device technologies, it is difficult to forsee accurately which of the many active efforts will succeed (see for example, Ayling [7], Best [15], Bobeck [16], Camras [17], Dell [24], Fields [35], Gardner [39], Howard [50], Matick [&Matick.], Myers [69], Rector [74], Shahbender [79], Thompson [85]). Considering the technical advances clearly demonstrated in the laboratory and the driving "profit" motivation, it is reasonable to expect some dramatic changes in the next few years. Even if we don't know what or when, we would be foolhearty to totally ignore this situation.

Table 1 below indicates the performance and price characteristics of typical current-day storage technologies. The two entries marked by question marks (?), Bulk Store and Giant Store, indicate new technologies that have already

(ft1)

Device	Random Access Time (seconds)	Maximum Transfer Rate (byte/sec)	Price ($/byte)	Capacity (bytes)
1. Cache Store (IBM 3165)	1.6×10^{-7} (160 ns)	1×10^{8} (100M b/s)	8.8×10^{0} ($8.80)	1.6×10^{4} (16K)
2. Main Store (IBM 3360)	1.44×10^{-6} (1.44 us)	1.6×10^{7} (16M b/s)	5×10^{-1} (50¢)	5.12×10^{5} (512K)
3. Bulk Store? (AMS SSU[35])	1.3×10^{-4} (130 us)	8×10^{6} (8M b/s)	8.8×10^{-2} (8.8¢)	2×10^{6} (2M)
4. Large Store (IBM 2305-2)	5×10^{-3} (5 ms)	1.5×10^{6} (1.5M b/s)	2.2×10^{-2} (2.2¢)	1.1×10^{7} (11M)
5. Mass Store (IBM 3330)	3.8×10^{-2} (38 ms)	8×10^{5} (800K b/s)	4.5×10^{-4} (.045¢)	2×10^{8} (200M)
6. Giant Store? (Grumman MASSTAPE)	6×10^{0} (6 sec)	6×10^{5} (600K b/s)	2.2×10^{-5} (.0022¢)	1.6×10^{10} (16B)

Table 1.
Representative Storage Hierarchy

been placed in limited use. Since these two cost/performance positions were not part of our "traditional" technologies, we are faced with the problem of possible modifying our applications and developing new strategies to efficiently, effectively, and, hopefully optimally, integrate them into our overall hierarchical storage system.

As the entire spectrum of computer architectures, as well as storage device technologies, undergoes reshufflings, both evolution as well as revolutions, it is worthwhile to review and reconsider our current concepts on storage system design. Table 1, although a simplified summary of current storage technologies, illustrates the fact that there exists a spectrum of devices that span about 6 orders of magnitude of price/performance (100,000,000%). This is quite significant in the light of the excitement that normally accompanies an improvement of 10-20% in performance or a decrease of 10-20% in price in current-day systems. The participants in this "storage sweepstakes" may change in time, but with such large price/performance stakes, there will be continuing benefits to "playing the game" better.

2.1.4 Understanding of Program and System Behavior

As noted earlier, the detailed operational behavior of computer systems is often extremely complex. Thus, decisions on hardware, software, and system design must often be made in spite of insufficient knowledge. A better understanding of program and system behavior is essential to the intelligent and efficient development of future systems.

It is hoped that the research to be conducted as part of this thesis will shed considerable light on these matters.

2.2 Storage Hierarchy Approaches

"Storage hierarchy system" and similar terms have been used in many contexts. Consistent with the objectives outlined in the previous section, certain particular contexts are assumed in this research.

2.2.1 Spectrum of Approaches

The problems of storage hierarchy management have been attacked by a host of approaches. We can loosely characterize these efforts into three categories:

2.2.1.1 Manual Hierarchy Management

Given a specific ensemble of storage device technologies, after considerable thought the programmer can explicitly or implicitly specify how his information (i.e., programs and data) should be organized and distributed within the hierarchy and how and when his information should be re-arranged. Having determined the distribution, he must also specify his access to specify information accordingly.

When a programmer is directly operating upon his information at the lowest level (e.g., using machine language, direct I/O requests, etc.), he is explicitly controlling the storage hierarchy, this is _explicit manual hierarchy management_. In most conventional systems, the programmer communicates with the system via programming languages and control cards. Although this can relieve much of the tedious or intricate details of storage management, the overall control of the storage hierarchy is still primarily the responsibility of the programmer. This is _implicit manual hierarchy management_.

Manual storage management can be very economical since it usually requires no special hardware features nor special system software. Furthermore, it places the control of the storage hierarchy in the hands of the programmer who is presumably the one most familiar with his needs. Manual storage management, in its many manifestations, is the most common storage hierarchy approach in use today.

Manual storage management has many disadvantages, though. The amount of detail that the programmer must understand and use can add significant complexity to this task. This then introduces additional areas of error and decreased productivity. Furthermore, the assumption that the programmer is the best judge of optimal storage organization is often wrong. The complexities and dynamics common to modern systems are often beyond the understanding of most application programmers.

Multiprogramming, an almost universal technique in current systems, necessitates strategies for global optimization which usually differ substantially from the individual local optimizations of each program. For these reasons there has been continual search for "a better way".

2.2.1.2 Semi-Automatic Hierarchy Management

Many techniques have been developed to minimize the amount of effort required of the programmer and to provide feedback to him. The programmer still has the ultimate control in such a <u>semi-automatic hierarchy management</u> system.

Certain of these techniques are based upon the concept of the programmer providing "hints" to the system. These hints form the basis for a partially automated, partially manual storage management system. Although not especially widespread, this approach has been used in several systems (e.g., Jensen <u>et</u> <u>al</u> [53], O'Neill <u>et</u> <u>al</u> [70], etc.).

If there is a single application that is quite large and complex, techniques have been developed to analyze the actual performance and provide feedback to the programmer. This approach is primarily used in specialized, dedicated, predictable, high-performance systems, such as an airline reservations system. Numerous attempts have been reported, such as Arora <u>et</u> <u>al</u> [5], Ramamoorthy <u>et</u> <u>al</u> [72], etc.

The various semi-automatic hierarchy management approaches help to reduce the programmer's effort and to attain a better local optimization. Although useful for certain applications, these strategies do not remove the disadvantages already noted with manual hierarchy management systems.

2.2.1.3 Automatic Hierarchy Management

Certain aspects of logical information organization are inherent in a programmer's basic algorithm. In an _automatic hierarchy management_ system, all aspects of the physical information organization and distribution that are irrelevant to the underlying logical structure should be removed from the programmer's responsibility. The programmer may wish to, maybe even be encouraged to, use algorithms that are known to perform well in conjunction with the automated hierarchy management. But, the central responsibility of the storage hierarchy management is removed from the programmer.

Since this approach directly focuses on the storage hierarchy objectives presented earlier, it will be the primary approach to be pursued in this thesis.

2.2.2 Spectrum of Analysis Efforts

Each of the storage hierarchy approaches mentioned above, primarily semi-automatic and automatic, have been subjected to various forms of analysis. In this section we briefly outline the principal deficiencies of these efforts.

2.2.2.1 Generalized Models

One popular form of analysis is to assume a generalized model for hardware, software, and system behavior. If one is careful in choosing the characteristics of the model (e.g., Poisson arrival and service times, etc.), it is possible to develop precise analytical solutions. Unfortunately, it is usually difficult to validate these models except for rather simple solutions. Furthermore, since there are few truly automatic storage hierarchy systems in general use, it is extremely difficult to even determine realistic parameters for these generalized models even if the models were valid.

Generalized models have been reported in several papers, such as Aho _et al_ [2] and Denning [25] in the Bibliography.

2.2.2.2 Constrained Models

Another variation on the generalized model scheme is to analyze a particular program and then model its relationship to the rest of the system. There are at least two shortcomings in this approach. First, as in the generalized model case, it is difficult to realistically model the relationship between a program and the rest of the system. Second, the analysis and measurement of the particular program is normally converted into some form of probability matrix or probabalistic reference pattern. In either case, significant effort is required to accurately measure the program's behavior. Furthermore, the probabalistic characteristics are usually aggregated to reflect the overall behavior of the program and, as a result, the dynamic nature of the program and its impact on the storage hierarchy are often lost.

Example analyses of constrained models can be found in references: Arora and Gallo [5], Hatfield and Gerald [47], Lewis and Yue [60], and Ramamoorthy and Chandy [72].

2.2.2.3 Limited Environment

A common deficiency of most previous research is that only a limited environment was considered, in particular automatic hierarchy management over only two levels using a single page size. Of course, most current-day computers have only employed automatic hierarchy management in either Cache Systems (cache store - main store) or Paging Systems (main store - large store). Unfortunately, there is definite reasons to believe that many of the conclusions and techniques demonstrated for a two-level hierarchy do not necessarily generalize to handle the spectrum of program detail and device characteristics encountered in a truly multiple level storage hierarchy. Furthermore, many of the papers that attempted to investigate general storage hierarchies assumed techniques and approaches that are primarily based upon two-level hierarchy assumptions.

This limited environment has been studied by numerous people, such as Aho _et al_ [2], Belady _et al_ [10,11,12], Coffman and Varian [19,86], Conti _et al_ [21,22], Denning [25], Fotheringham [33], Guertin [45], Kilburn _et al_ [57], Mattson _et al_ [63], Seligman [78], Smith [81], and Wilkes [88].

2.2.2.4 General Hierarchy Environment

The studies of limited two-level storage hierarchies have been quite successful in many actual systems. A reasonable strategy would be to extend these techniques to a more general storage hierarchy environment. There have been a few attempts along these lines, but as mentioned in the previous section, most were hampered by:

(1) attempting to directly apply two-level hierarchy techniques without carefully considering their applicability,

(2) attempting to generalize techniques which were not even fully understood in a two-level environment.

The major thrust of this thesis is to provide insight into and shed additional light on these problems.

CHAPTER 3.
FORMALIZATION OF STORAGE HIERARCHIES AND RELATED RESEARCH

3.0 Introduction

In this chapter a formalization of the key characteristics of storage hierarchies is presented and performance measures are derived. The reported performance of actual systems is reviewed.

3.1 Major Parameters of a General Storage System

Table 2 and Figure 1 illustrate the major parameters of a storage hierarchy system. These parameters can be grouped into four categories: (1) basic technology, (2) configuration, (3) algorithm, and (4) program behavior.

3.1.1 Basic Technology

The basic technology parameters, cost/byte, C, and average access time, T, are primarily dependent upon the physical properties of the storage device technology. At any given time the state-of-the-art offers only a limited number of (C,T) alternatives that the system designer can select.

(ft2)

<u>Basic Technology</u>

- C cost/byte
- T average access time

<u>Configuration</u>

- L number of levels
- I interconnection of levels
- S size (capacity)
- B transfer rate (bandwidth)
- N number of bytes in page (page size)

<u>Program Behavior</u>

- A address trace

<u>Algorithm</u>

- F fetch
- P placement
- R replacement

Table 2.
Major Parameters of a Storage Hierarchy System

(f13)

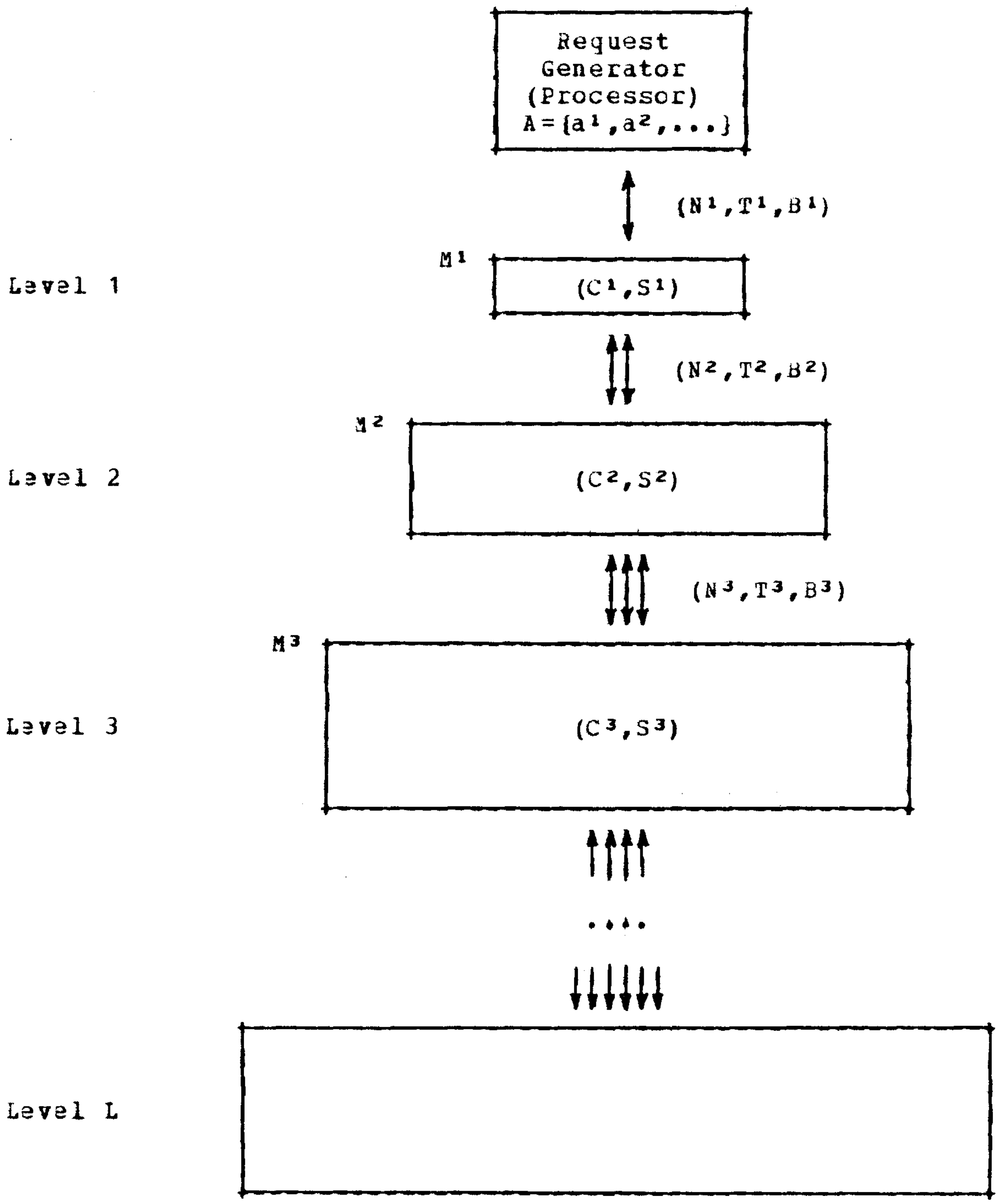


Figure 1.
Structure of a Storage Hierarchy System

3.1.2 Configuration

The system designer does have flexibility in organizing these storage devices. By serial and/or parallel structuring of the components of a given level of storage device technology, it is possible to specify, over a wide range of values, the size (storage capacity), S, and the maximum transfer rate (data bandwidth), B, of the system. For example, if a particular technology provides a basic device with S=s and B=b, connecting n of these devices in parallel produces a storage level with S=ns and B=nb. (To some extent the mechanism and cost of the organizational structure does influence the overall cost/byte and average access time of a level, this effect is usually minimal for small values of n).

On a more global basis, the designer must determine the number of levels, L, in the storage system, the interconnections of the levels, I, and the size, N, of a page (the unit of information moved between levels).

3.1.3 Program Behavior

The processor, under program control, produces a sequential series of references to the storage system. These processor references are in the form of logical address references which serve to uniquely identify each individual unit of stored informatation (e.g., an 8-bit byte) independent of its location (i.e., M^1, M^2, M^3, ...). The time sequence of logical address references, A, is called an address trace or address reference pattern. In general, each unique program and its input data will result in a different processor address trace. For purposes of analyzing the effectiveness of the storage hierarchy, the address trace is the primary characterization of a program that is needed (e.g., we don't care what the program's purpose is or what language it is written in, etc., we only care about its address trace). Thus, the address trace describes the program's behavior as observed by the storage hierarchy.

3.1.4 Algorithm

There are three basic decision algorithms that must be employed by an automatic storage management system. Fetch, F, decides when and which information should be moved up a level (e.g., from M^2 to M^1). Placement, P, decides where information should be placed in a level. Removal or replacement, R, decides when and which information should be transferred down a level (e.g., from M^1 to M^2).

3.2 The Storage Hierarchy Model

A completely general storage hierarchy algorithm, H, must consider all the parameters described above:

$$H = f(\langle Technology\rangle, \langle Configuration\rangle, \langle Program\rangle, \langle Algorithm\rangle)$$

$$H = f(\langle C, T\rangle, \langle L, I, S, B, N\rangle, \langle A\rangle, \langle F, P, R\rangle)$$

Clearly, attempting to optimize a system with so many parameters is difficult. Fortunately, it is possible to eliminate from concern or at least simplify certain parameters as explained below.

3.2.1 Configuration

Consistent with the title of this thesis, we shall consider only hierarchical interconnections of levels as illustrated in Figure 1, where $T^1 < T^2 < T^3 <$ etc. and $N^1 < N^2 < N^3 <$ etc. The rationale for this decision is elaborated in the thesis.

There are three basic strategies for information movement sizes: (1) select a single page size value, N, which is always used throughout the hierarchy - this approach is used on most contemporary automatic multilevel storage systems (e.g., Multics), (2) allow an arbitrary range of values for N to be used - this approach is primarily used on manually managed storage systems, and (3) select L values of N, a specific unit of transfer is used between any two levels of the hierarchy - this approach will be pursued and justified in this thesis.

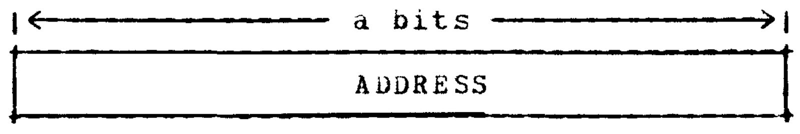
3.2.2 Program Behavior

Each logical address can be represented as a bits as shown in Figure 2(a). If the page sizes, N, are chosen to be powers of 2, the set of 2**a possible addresses can be partitioned into 2**p pages of N=2**n consecutive logical addresses each as shown in Figure 2(b). [Note: the notation "2**a" means 2 raised to the power a]. Since the information movement between storage levels is accomplished by transferring pages, we can analyze this interlevel movement by merely considering the time sequence of logical pages references, Ap, called a <u>page trace</u>.

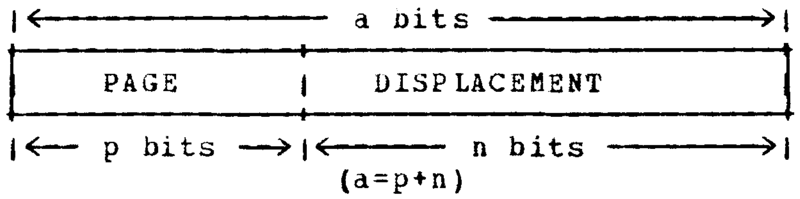
Since we allow the page size to be different between each level and requests are only passed down to a given level if they cannot be satisfied by any higher level, each level will usually experience a different page trace though all are algorithmically derivable from the same address trace. In fact, if all address references were broadcast to all storage levels, the page traces can be determined by a simple mapping from logical addresses into logical pages:

page address = **integer**(logical address/N)

(f2)



(a) Logical Address



(b) Logical Address
(Divided into Page Address and Displacement)

Figure 2.
Format of Logical Address

where N is the page size for that level.

3.2.3 Algorithm

The placement decision, P, is usually unconstrained or minimally constrained and, as a result, has relatively little impact upon performance.

A <u>demand fetch</u> policy will be used. Assume that at time t a request for logical address a (or, equivalently, p^1=**integer**(a/N^1)) arrives at level M^1. At that instant the information may currently reside in M^1, otherwise it must be found in a lower level. Under demand fetch, if p^1 is in M^1, the reference proceeds, the information is passed back to the processor, and no other page movement occurs in the hierarchy. If p^1 is not in M^1, a request for p^2=**integer**(a/N^2) is sent from M^1 to M^2. If p^2 is in M^2, the page is transferred to M^1 and processing continues as described above, otherwise a request for p^3=**integer**(a/N^3) is sent from M^2 to M^3, etc. Note that under the demand fetch policy, information is only moved up in the hierarchy when and if it is explicitly <u>demanded</u> (i.e., requested) by the processor.

Although demand fetch is only one possible fetch algorithm, it can be shown [63] that for hierarchically structured storage systems:

"... given any trace and replacement algorithm (not necessarily using demand paging) another replacement algorithm exist that uses demand paging and causes the same or fewer total number of pages to be transferred ..."

In other words, as you might intuitively suspect, moving pages only when necessary results in the minimal number of page movements. Of course, if page movement is required and the higher level that is to receive the page is already full, the removal algorithm must be employed to provide space for the new page.

3.2.4 Revised Storage Hierarchy Model

Based upon the discussion above, we can slightly simplify the parameters remaining for consideration in the storage hierarchy algorithm, H, so that it need consider only:

$$H = f(\langle Technology\rangle, \langle Configuration\rangle, \langle Program\rangle, \langle Algorithm\rangle)$$

$$H = f(\langle C,T\rangle,\ \langle L,S,B,N\rangle,\ \langle A\rangle,\ \langle R\rangle)$$

In this thesis all of these parameters will be considered and investigated. Special emphasis will be placed on analyzing and understanding the relationship between the pages sizes, N, and the removal algorithm, R, required for efficient operation of the storage hierarchy.

3.3 Performance Measures

There are various performance measures that we could consider. For an overall point of view, system measures, such as job throughput, job turn-around time, and processor utilization, are quite significant. Unfortunately, it is extremely difficult to directly relate these measures to the performance of the storage system, even an approximation would require consideration of many more parameters. Thus, we will only consider measures that relate to the effective performance of the storage hierarchy.

3.3.1 Performance Measurement Notation

Due to the strict hierarchical structure of our storage system and the demand fetch policy, we can analyze the performance of the system by separately considering the levels of the hierarchy starting with M^1. Since a given level only receives a page fetch request if the information has not been found at a higher level, each level usually sees a different page trace, Ap^1, Ap^2, Ap^3, etc.

There are several important properties of page traces. If P is a particular page trace (e.g., Ap^1) of a program, we define:

- $|P|$ length of the page trace sequence

- Q set of distinct pages referenced in P
- $|Q|$ number of pages in Q

For example, in the page trace

P = a, b, a, c, b, a

we observe that

$|P| = 6$

$Q = \{a, b, c\}$

$|Q| = 3$

(Lower case letters will be used to represent logical page addresses instead of page numbers).

For a specific storage hierarchy, we define $|M|$ to be the size of M in units of pages receivable from the next lower level. For example, $|M^1|=S^1/N^2$, $|M^2|=S^2/N^3$, etc.

For a specific page trace, P, storage level, M, and removal algorithm, R, we define the _result page trace_ or _fetch page trace_, P', as the time sequenced page references of P that were not found in M. We shall call page references that are found in M _successes_. The _success function_, Sf, is the number of references satisfied by M and can be computed as $|P|-|P'|$. By analogy to the success function, the number of references not satisfied by M, $|P'|$, is called the _failure function_, Ff. In general, we wish to maximize the success function or, equivalently, minimize the failure function. It is convenient to normalize the failure function by defining the _failure frequency function_, f,

$$f = |P'|/|P|$$

The _success frequency function_, s, can be easily computed as 1-f; it is often called the _hit rate_ on a two-level storage system. We also define the _system failure frequency function_, f^o, of a level to be:

$$f^o = |P'|/|A|$$

where A is the address trace generated by the processor and |A| is the length of the address trace (it is also true that |A| always equals $|P^1|$, thus they may be used interchangeably). The _system success frequency function_ is correspondingly defined as $s^o=1-f^o$.

If we apply the definitions above to the processor generated page trace, P^1, received by M^1, we note that the result page trace, P', is essentially the page trace, P^2, received by M^2. There is a minor relabeling required to adjust for the difference in page size used by M^2, $p^2=P'(N^1/N^2)$. By repeating this process recursively, we can develop the effective page traces, failure and success functions, and failure and success frequency functions for each level of the hierarchy. Since we assume that all referenced information exists in the storage hierarchy, the sum of the system success frequency functions must be 1.

One general measure of a storage hierarchy's performance is its effective access time, T', and effective cost, C', which are defined as follows:

$$T' = T^1 S^{01} + T^2 S^{02} + T^3 S^{03} + \ldots$$

$$C' = (C^1 S^1 + C^2 S^2 + C^3 S^3 + \ldots)/(S^1 + S^2 + S^3 + \ldots)$$

T' and C' can be viewed as characterizing the entire storage hierarchy according to a corresponding one-level system. From a cost/performance point of view, one should be indifferent between a single-level single-technology storage device with average access time, T', and average cost/byte, C', and a storage hierarchy system with performance parameters (T',C'). In particular, if the system designer needs a storage performance (T,C) and no such basic technology exists, he must attempt to develop a storage hierarchy such that (T',C') = (T,C).

3.3.2 Page Trace Simulation

One way to determine the success frequency function and the result page trace for a specific page trace is to simulate the storage management algorithms and note the contents of M at each step of the page trace. Clearly, these results depend upon numerous parameters (e.g., specific trace, removal algorithm, size of M, etc.). Figure 3 illustrates this step by step simulation assuming demand paging, FIFO (first-in first-out) removal, and $|M| = 2$ pages. For simplicity, the page trace, P, has been

(E3)

Parameters

- P = a, b, b, c, b, a, d, c, a, a
- |P| = 10
- Q = { a, b, c, d }
- |Q| = 4
- |M| = 2
- FIFO Removal

Simulation

Page Trace,P	a	b	b	c	b	a	d	c	a	a	
Fetch	*	*		*		*	*	*	*		
M Contents (after each reference)	a	b	b	c	c	a	d	c	a	a	<-"new"
		a	a	b	b	c	a	d	c	c	<-"old"
Page Trace,P'	a	b		c		a	d	c	a		

Results

- Pf = |P'| = 7
- Sf = |P|-|P'| = 3
- f = 70%
- s = 30%
- P' = a, b, c, a, d, c, a

Figure 3.
Example of Page Trace Simulation

normalized to be expressed in units of receivable pages. In particular, if M is M^1, then $|M|=S^1/N^2$ and p=**integer**(a/N^2) where a is a logical address reference and p is corresponding page reference. The pages in M are shown as ordered to indicate the FIFO ordering, the top page is the "last" ("latest") page fetched into M, whereas the bottom page is the "first" ("oldest") page in M and is the page selected for replacement when necessary. The asterisk (*) indicates that a fetch was required from a lower level of the hierarchy, the page reference is thus noted as part of the result page trace, P'.

It is normally assumed that all levels, except level L, are empty initially, thus there is a transient stage during which pages are loaded into M without any replacements needed. Since there are so few pages in M during this start-up stage, there are many fetches required. We will find it useful to separate out this transient phenomenon. This transient consists of the page trace up to the first $|M|$ unique page references, in the example of Figure 3 this is the first 2 page references (i.e., a, b). Consider the case if $|Q|\leq|M|$, there would be no further fetches into this level after the initial transient that loads the $|Q|$ pages into M. In this case, $|P'|=|Q|$ exactly, independent of $|P|$, and s tends toward 1 as $|P|$ increases.

In the particular example illustrated in Figure 3, we note that there were 3 'hits' and 7 'misses' out of 10 page references, so that s=30%. Thus, P' only consists of 7 page references to the lower levels.

3.4 Related Research

As noted above, we wish to develop a storage hierarchy with attractive cost/performance, (C',T'), characteristics. It is clear that we can arbitrarily decrease the cost/byte by making the size of each level, S, increasingly larger as we go from the high-performance high-cost to the low-performance low-cost levels (i.e., $C^1>C^2>C^3>\ldots$ and $S^1<S^2<S^3<\ldots$). In fact, this approach is the basic motivation for storage hierarchies.

Unfortunately, if the processor generated address references that were uniformly distributed in time and address, each byte would be equally likely to be referenced at any instant. This probability would be:

$$\Pr[\text{reference } a] = 1/(S^1+S^2+S^3+\ldots)$$

Thus, the expected system success function, s^0, for each level is proportional to the size of the level. For example,

$$s^{01} = S^1/(S^1+S^2+S^3+\ldots).$$

But, since we have assumed that $S^1<S^2<S^3<\ldots$, we find that

$s^{o}_{1} < s^{o}_{2} < s^{o}_{3} < \ldots$ Thus, the system success function for the Lth level dominates (i.e., is approximately 1) since we have assumed that it is the largest level. Referring back to our definition of effective access time, we find that T' would be approximately equal to the lowest performance level (level L) since all the other terms would be negligible. If this analysis were true, our storage hierarchy would result in a performance just slightly better than our lowest performance level at a moderate increase in price - not an especially exciting result. Fortunately, actual storage hierarchies do not behave this way. We will briefly review some related research on this subject.

3.4.1 Locality

It has been empirically observed that actual programs cluster their references so that, during any interval of time, only a subset of the information available is actually used. A detailed discussion of this phenomenon will be presented in the thesis.

It is important to note that due to our basic rankings of page sizes and access times in the storage hierarchy, each level "sees" a different view of the program. The high levels of the hierarchy must follow the microscopic instruction by instruction reference pattern whereas the middle levels follow a more gross subroutine by subroutine pattern. The very low levels are primarily concerned about the processor's references as it moves from subsystem to subsystem. We do not have any a priori guarantee that locality of reference holds equally true for all of these views, but we do have some reported evidence to encourage us. Most of these studies have been based upon two-level storage systems or restricted forms of three-level hierarchies.

3.4.2 Paging Systems

The earliest automatic storage systems were based upon two-level core-drum hierarchies (devices 2 and 4 of Table 1). This technique was introduced in the Atlas system [38,57] during the early 1960's. It has since been used on many contemporary systems.

The performance of paging systems has been studied by various researchers, such as Belady [12], Coffman and Varian [19,86], Hatfield [48], and Sayre [77]. In Coffman's results, for example, it was noted that even though $S^1/(S^1+S^2)=0.25$, s^1 often exceeded 95%. Hatfield studied the performance of system programs that had been carefully designed and found that for $S^1/(S^1+S^2)$ ratios as low as 0.25, it was possible for s^1 to often exceed 99.99%.

3.4.3 Cache Systems

Cache systems are based upon two-level cache-main hierarchies (devices 1 and 2 of Table 1). Although they have been proposed as early as 1965 (see Wilkes [88]), the major commercial use of cache systems did not occur until the introduction of the IBM System/360 Model 85 [21,61]. More recently, this technique has been used in several contemporary systems, such as the IBM System/370 Model 155 and Model 165 [52].

In these cache systems, IBM found that it was possible to drastically reduce $S^1/(S^1+S^2)$ to as low as 1% and still keep the hit ratio, s^1, above 90%. Similar findings were also reported by Bell and Casasent [13], Mattson [64], Meade [65], and Seligman [78].

3.4.4 Three-level Systems

There have been a few three-level systems reported in the literature, unfortunately they have all been somewhat _ad hoc_ in design and the results are far from conclusive. There have been at least three types of such hierarchies studied.

3.4.4.1 Main-Bulk-Mass Store Hierarchy

There have been several systems devised based upon devices 2, 3, and 5 of Table 1. The Bulk Store actually used, called Large Core Store (LCS), had a much lower access time (around 8 us) and a much higher price (about 25¢/byte). In order to compensate for peculiarties in the hardware structure and out of considerable concern for the extreme cost of LCS, these systems tended to become much more manually managed hierarchies than automatically managed. Although they were found to be effective, it is difficult to generalize the results. The most ambitious attempt reported was undertaken by Carnegie-Mellon University [36]. Results have also been reported by Durae [31], Williams [89], and others.

3.4.4.2 Main-Large-Mass Store Hierarchy

There does not appear to be any automatically managed systems of this type published in the general literature. The Multics system at MIT Project MAC has recently introduced a "page-multilevel" strategy based upon devices 2, 4, and 5 of Table 1. There has only been limited finding reported to date but it has been stated in the March 1972 issue of the MIT Information Processing Services <u>Bulletin</u>

(p. 11) that it

> "... does pay off since it meets fluctuating demands on the system, reduces the workload for the disks to an efficient level, is inexpensive, and keeps pages on the drum for an acceptable length of time."

As an indication of its effect, the new strategy is reputed to have increased the success frequency function, s^2, of the drum from 20% to more than 90% (i.e., "reduced from one page read from the disk for every four reads from the drum, to one page read from the disk for every ten to twenty pages from the drum").

3.4.4.3 Main-Large-Giant Store Hierarchy

The work of Considine and Weis [20] is difficult to categorize. It is based upon a three-level hierarchy where the first level corresponds to device 2 (main store) of Table 1, the second level corresponds to a combination of devices 4 (drums) and 5 (disks), and the third level consists of removable disks which can best be approximated by device 6 of Table 1. It is impossible to compute any success frequency functions from their data, but it appears that for $S^2/(S^2+S^3)=0.5$, s^2 is very high. They note (p.443), in particular, "most of the data moved to the archival storage (i.e., M^3) have stayed there."

3.4.5 Need for Additional Research

Although the results of research described above is encouraging, the design and performance of general multiple-level storage hierarchies are still inconclusive. This thesis is intendel to provide specific results in this area.

CHAPTER 4.
A STORAGE HIERARCHY SYSTEM

4.0 Introduction

In this chapter a design for a general multiple level storage hierarchy system, in particular with three or more levels, is presented. This design is based upon an orderly and uniform treatment of the logical structure of the storage levels and their interconnections. In addition to providing a solution to convenient storage management for the user, this design is intended to produce good performance for the storage hierarchy as measured by its effective access time, T', and effective cost, C'. The principle and novel techniques to be used are described separately in the sections below.

4.1 Continuous Hierarchy

As noted earlier, automatic storage hierarchy systems are still in the minority. Amongst those systems that do provide automatic storage hierarchy management, the majority limit their scope to two levels with a few rare three level systems. As a result of these limitations, the user is still forced to rely on manual or semi-automatic storage management techniques to deal with the storage levels that are not automatically managed. Thus, an automatic storage management system should consist of a <u>continuous hierarchy</u> that encompasses the full range of storage levels.

4.1.1 Cost/Performance of Adjacent Levels

A major obstacle to generalizing storage management algorithms, in particular in two-level paging systems, is the tremendous contrast, often over 3 orders of magnitude, in cost/performance between M^1 and M^2. As illustrated in Table 1 (page 28), a representative Main Store, M^1, has an access time of 1.44 us compared to a Large Store, M^2, with an access time of 5 ms. In such a two-level system, the effective access time, T', is

$$T' = T^1 s^{01} + T^2 s^{02}$$

$$T' = 1.44 s^{01} + 5000 s^{02}$$

and since $s^{01}+s^{02}=1$, we can substitute $s^{01}=1-s^{02}$ to get

$$T' = 1.44 - 1.44 s^{02} + 5000 s^{02}$$

$$T' = 1.44 + 4998.56 s^{02}$$

In order to attain an effective access time, T', that is comparable to the Main Store access time, T^1, we must keep the system success frequency function, s^{02}, very close to 0 or, correspondingly, keep s^{01} very close to 1. Even with s^{01} at 99.8%, an improvement to 99.9% would cut the effective access time, T', in half. With such pressure to attain very high s^{01} values, the systems designer is often forced to seek out very specialized techniques in contrast to our goals of orderly and uniform algorithms.

4.1.2 Moderate Cost/Performance Ratios

In order to make the storage hierarchy design robust and flexible, the cost/performance characteristics should differ by less than two orders of magnitude between adjacent levels. Thus, success frequency functions in the range 90% to 99% are adequate to insure reasonable performance. If the differences are much greater, it will be difficult to find sufficiently efficient general algorithms. Since minor changes in production techniques and technology evolution can result in a variation of a factor of two or three in the cost/performance for a given technology, it is not desirable to decrease much below one order of magnitude difference between adjacent storage levels.

4.2 Shadow Storage and Page Splitting

The time, Tm, required to move a page between two levels of the hierarchy usually consists of summing two components: (1) the average access time, T, and (2) the transfer time, BxN.

If all page sizes were set to provide exactly the amount of information, N^1, requested by the processor, the page movement time would be

$$Tm = T + BxN^1$$

where T and B would depend upon the particular storage levels. By examining the representative devices shown in Table 1 (page 28), we see that access time varies much more than transfer rate (i.e., access time spans 6 orders of magnitude whereas transfer rate varies by only 3 orders of magnitude).

4.2.1 Marginal Increase in Page Transfer Time and Reference Probability

Let us assume that N^1 is quite small, such as 8 bytes. We can ask the question: What is the marginal increase in Tm if we transfer the adjacent N^1 bytes in addition to the N^1 bytes requested by the processor? Table 3 on the next page answers this question. Notice that the marginal increase in Tm decreases from a high of 5.5% (level 2 to level 1) to a low of .002% (level 6 to level 5). This fact is only interesting if we also consider the concept of locality (see Chapters 5 and 6 for additional discussion) and the question: What is the probability, Pr, that the processor

(ft3)

Level Transfer	Tm (1 unit)	Tm (2 units)	Marginal Increase in Tm
2 to 1(*)	1.44 us	1.52 us	5.5%
3 to 2	131 us	132 us	.8%
4 to 3	5006 us	5011 us	.1%
5 to 4	38010 us	38020 us	.03%
6 to 5	600013 us	600027 us	.002%

Table 3.
Marginal Increase in Page Transfer Times

- - - - -

* The figures for access time and transfer rate for the Main Store listed in Table 1 are approximations that are only meaningful for very large page sizes. For the page sizes under consideration in this chapter, the figures used in the table above are more appropriate.

will reference the adjacent N^1 bytes with a short interval of time, such as Tm seconds? Due to locality of program reference, we would expect Pr to be much larger than merely the reciprocal of the logical address space size. Furthermore, Pr should increase as Tm increases. Thus, for a given level, if Pr is larger than the marginal increase in Tm, it is beneficial to transfer the additional N^1 bytes and thereby avoid the necessity of expending Tm seconds to transfer these N^1 bytes later separately.

These same arguments can be applied to the question of transferring the adjacent nxN^1 bytes, etc. Since the marginal increase in Tm decreases monotonically as a function of storage level, the number of N^1 byte packets to be transferred as a single page should increase monotonically. This confirms our earlier decision that $N^1<N^2<N^3<$ etc.

4.2.2 Choice of Page Size

In order to simplify the implementation of the system and to be consistent with the mapping from logical address to page address illustrated in Figure 2 (page 46), we will require that all page sizes be a power of two. Thus, each page size (e.g., N^3) is some power of two larger than the page size of the next higher level (e.g., $N^3=N^2**i$).

Clearly, the specific values of Pr and thus the choice for each page size depends upon the characteristics of the programs to be run and the effectiveness of the overall storage system. Preliminary measurements indicate that a ratio of 4:1 between levels is reasonable, Meade [65] has reported similar findings. Other important factors affecting page size are discussed in Chapters 5 and 6.

4.2.3 Page Splitting

Now let us consider the actual movement of information in the storage hierarchy. At time t, the processor generates a reference for logical address a. Assume that the corresponding information is not currently stored in M^1 or M^2 but is found in M^3. For simplicity, assume that page sizes are doubled as we go down the hierarchy (e.g., $N^2=2N^1$, $N^3=2N^2=4N^1$, etc.; see Figure 4). The page of size N^3 containing a is copied from M^3 to M^2. M^2 now contains the needed information, so we repeat the process. The page of size N^2 containing a is copied from M^2 to M^1. Now, finally, the page of size N^1 containing a is copied from M^1 and forwarded to the processor. In this process the page of information is split (i.e., <u>page splitting</u>) repeatedly as it moves up the hierarchy.

(f14)

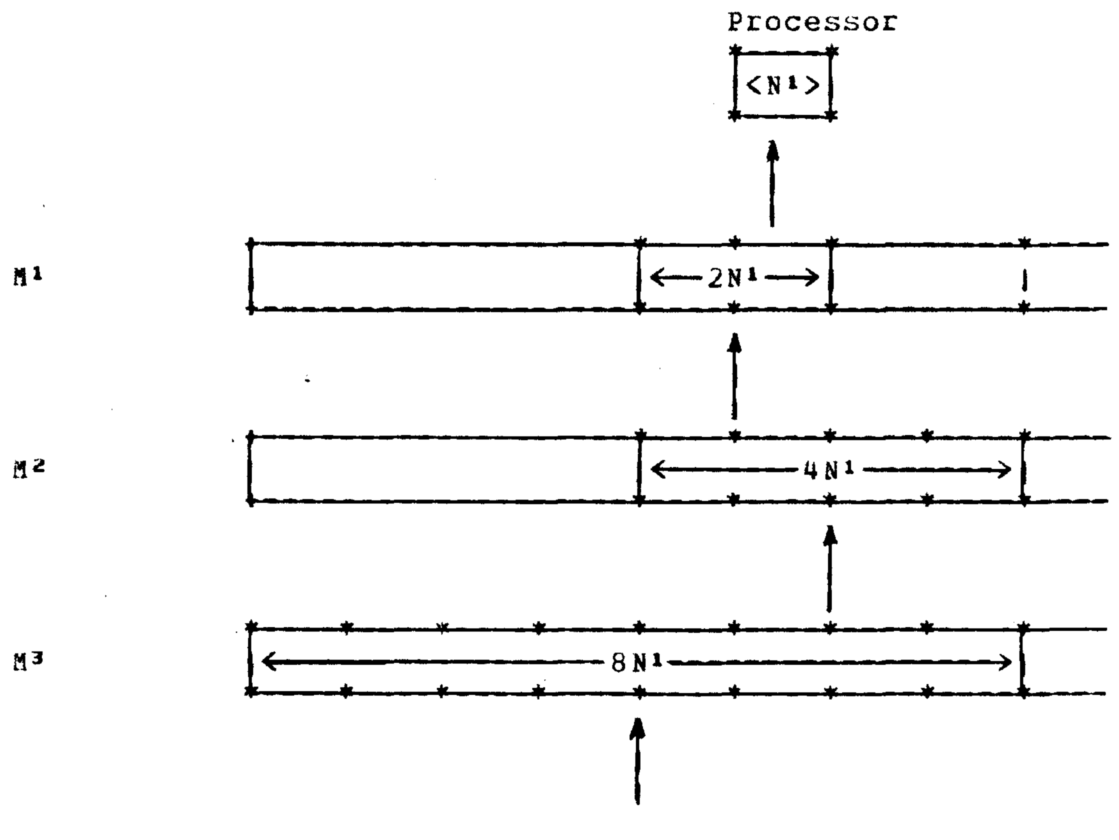


Figure 4.
Page Splitting and Shadow Storage

4.2.4 Shadow Storage

As a result of this splitting, the page of size N^1 that is received by the processor has left a "shadow" consisting of itself and its adjacent pages behind in all the lower levels (i.e., shadow storage). Presumably, if the program exhibits locality of reference, many of these shadow pages will be referenced shortly afterward and be moved further up in the hierarcy also.

4.2.5 Copying of Pages

In the strategy presented, pages are actually copied as they move up the hierarchy; a page at level n has one copy of itself in each of the lower levels. Since processor "fetch" requests substantially outnumber "store" requests (e.g., by more than 5:1 in some measured programs), the contents of pages are seldom changed. Thus, if a page has not been changed and is selected to be removed from one level to a lower level, it need not be actually transferred since a valid copy already exists in the lower level. The contents of any level of the hierarchy is always a subset of the information contained in the next lower level. Thus, the total information capacity of the system is equal to the size of the level L store rather than the sum of the capacities of all the levels. Since the capacity of level L is assumed to be much larger than the capacity of L-1, etc., the difference in total system capacity due to shadow storage is minimal.

4.3 Direct Transfer

In the description above it is implied that information actually moves between adjacent levels. This approach, called *direct transfer*, is indeed intended. By comparison, though, many proposed and experimental multiple level storage systems are based upon an *indirect transfer* (e.g., the Multics "page multilevel" system mentioned in Chapter 2). In these systems, all information is routed through level 1. For example, to move a page from level n to level n-1, the page is moved from level n to level 1 and then from level 1 to level n-1. Clearly, this indirect approach is undesirable since it requires extra page movement and consumes a portion of the limited M^1 capacity in the process.

There have been two major obstacles to direct transfer in previous systems: (1) interconnection structure and (2) synchronization.

4.3.1 Interconnection Structure

For many reasons, some technical and some historical, most contemporary systems are physically structured in a radial manner. That is, there is a central element to the system, either the processor itself or the primary store, and all other storage devices and/or processors are directly connected to this central element. Except for some possible control signals, there are no direct data transfer connections between the non-central elements. This structure is, of course, quite consistent with a non-hierarchical storage management system. A logical storage hierarchy system should be based upon a physically hierarchical interconnection structure.

4.3.2 Synchronization

As indicated in Table 1, storage devices often have different timing and transfer rate characteristics. In order to accomplish a direct data transfer between levels, synchronization is necessary. It may be obvious that a storage device can not transfer data faster than its rated performance, but for many storage devices, especially electromechanical devices, it is not possible to transfer data slower than its rated speed.

Based on current technology, this problem can be solvel. Many of the storage devices are now non-electromechanical (i.e., strictly electrical), such as the Cache, Main, and Bulk Stores of Table 1. It is quite feasible to provide direct transfer between any of these devices and any other storage device; this is one reason for the radial interconnections described above where the Main Store acted as the common means of providing synchronization. Using a similar approach, we can allow direct transfer between electromechanical devices if this transfer is routed through a small and reasonably inexpensive electrical storage buffer. Femling [33] discusses such a device, which he calls a rubber-band memory presumably because it "stretches" to match the characteristics of the source and destination devices.

4.4 Read Through

In the description above, it is implied that a transfer up the hierarchy from level 2 to the processor (level 0) consists of two sequential steps: (1) transfer page of size N^2 from level 2 to level 1, and then (2) extract the appropriate page subset of size N^1 and transfer it from level 1 to the processor (level 0). In general, a transfer from level n to the processor would consist of a series of n steps. Thus the system page transfer time would equal the sum of n inter-level page transfer times (e.g., $Tm^{13}+Tm^{23}+Tm^{32}+ \ldots$). Furthermore, for many electromechanical storage devices, the second access, required to forward the page subset, may experience the "maximum" access delay rather than the "average" (i.e., after storing the information into the level, a complete mechanical revolution may be required to reposition to read the same information and forward it to the next level).

This inefficiency can be avoided by allowing information to be stored into all upper levels simultaneously. Figure 5 illustrates this mechanism. If information is to be transferred from M^3 to the processor, M^3 turns on its output data gate, G^3out, when it is ready to start and transfers N^3 bytes and their corresponding logical addresses up the data bus. M^2 turns on its input data gate, G^2in, to receive these N^3 bytes; furthermore, when the appropriate N^2 bytes needed by M^1 are detected by M^2, it turns on its output data gate, G^2out, and these N^2 bytes are forwarded to M^1 while being stored in M^2, etc.

For example, assume a reference to logical address a is generated by the processor and the corresponding information is current stored at level n (and all lower levels, of course). At the instant that the N^1 bytes containing a are

(£15)

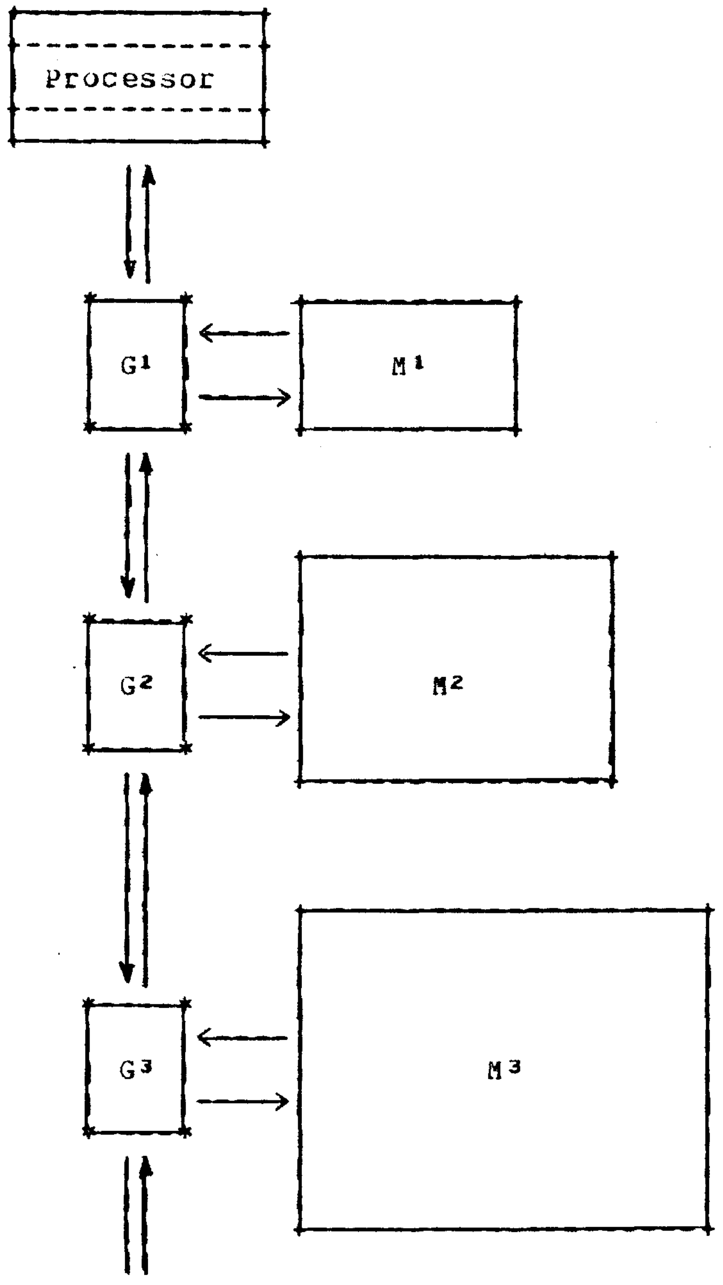


Figure 5.
Read Through Structure

placed on the data bus by level n, these N^1 bytes will be stored into all levels from level n-1 to level 0 (the processor) simultaneously. Likewise, the N^2 bytes containing a are simultaneously stored into all levels from level n-1 to level 1. This strategy thus makes it appear that the N^1 byte page requested by the processor is read through directly to the processor without any delays.

4.4.1 Page Transfer Time

Using the read through strategy, the page transfer time to the processor is actually less than the page transfer time to the adjacent storage level. For example, if the requested information is stored in M^3, the page transfer time to the processor, via read through, is

$$Tm^{30} = T^3 + N^1B^3$$

whereas, the page transfer time from M^3 to M^2 is

$$Tm^{32} = T^3 + N^3B^3.$$

Since $N^1<N^3$, then $Tm^{30}<Tm^{32}$.

4.4.2 Availability and Servicability

The read through mechanism described above offers some important advantages to the availability and serviceability of the storage system. Note that all storage levels are connected to the gated data bus not directly to each other.

If a storage level must be removed from the system for servicing, it is merely necessary to manually set both Gin and Gout "on". In this case, the information is really "read through" this level as if it didn't exist. No other changes are needed to any of the other storage levels or the storage management algorithms although we would expect the performance to decrease.

4.5 Store Behind

Under normal steady-state operation, all the levels of the storage hierarchy will be full (except possibly level L). Thus, whenever a page is to be moved into a level, it is necessary to remove a current page. If the page selected for removal has not been changed by means of a processor "store", the new page can be immediately stored into the level since a copy of the removed page already exists in the next lower level of the hierarchy. If the processor generates a "store" request, all levels that contain a copy of the information being modified must be updated. This can be accomplished in three basic ways: (1) store through, (2) store replacement, or (3) store behind.

4.5.1 Store Through

Under a _store through_ policy, all levels are simultaneously updated whenever the processor generates a "store" request. This is the obvious inverse of the read through policy. But, there is a crucial distinction. Under read through, only storage levels 1 through n are used, where n is the highest level containing the requested information. Store through must update the contents of levels n through L. Thus, read through speed is limited by its slowest level affected, level n; store through is always limited by the speed on level L, the slowest level of them all. If 20% of all processor requests are "stores", the system success frequency function of level L will be at least 20%. Due to its large average access time, level L will be the dominate portion of the system's effective access time, T'.

Store through can be used efficiently only if the access time of level L is comparable to the access time of level 1, such as in a two-level cache system. In fact, it is used in some cache systems, such as the IBM System/370 Models 155 and 165 [52].

4.5.2 Store Replacement

Under a <u>store replacement</u> policy, the processor only stores into M^1. If a changed page is later selected for removal, it is then moved to the next lower level, M^2, immediately prior to being replaced. This process occurs at every level and, eventually, level L will be updated but only after the page has been selected for removal from all the higher levels. Due to the extra delays caused by updating changed pages before replacement, the effective access time for fetches is increased. Various versions of store replacement are used in most two-level paging systems since it offers substantially better performance than store through for slow second level storage devices (e.g., drums and disks).

4.5.3 Store Behind

<u>Store Behind</u> is a compromise strategy that bridges the gap between store through and store replacement and offers substantially better performance. In both strategies above, the storage system was required to perform the update operation at some specific time (e.g., at the instant of the "store" request for store through or at the instant of removal for store replacement). Once the information to be stored has been accepted by the storage management system,

the processor doesn't really care how or when the copies in the storage hierarchy are updated. Store behind takes advantage of this degree of freedom. Due to the large disparity between average access time and transfer rate for most levels, the maximum data transfer capacity is rarely reached (i.e., at any instant of time, a storage level may not have any outstanding requests for service or it may be waiting for proper positioning to service a pending request). During these "idle" periods, data can be transferred down to the next level of the storage hierarchy without affecting or delaying any fetch operation. Since these "idle" periods are usually very frequent under most actual circumstances, there can be a continual flow of changed information down through the hierarchy towards level L.

4.6 Automatic Management

Although an effective storage management system should attempt to minimize page movement and its associated "housekeeping", there will still be a substantial amount of work required to manage the hierarchy. It is desirable to remove as much as possible of the storage management from the concern of the processor and the programs running on the processor, including the operating system. There are two primary motivations for this objective: (1) the storage hierarchy should function as an independent component of the system to eliminate any added complexity to the processor or programs, and (2) we want to conserve the processor's computational powers for solving the user's problems rather than for "system overhead". In actuality, of course, the storage hierarchy can not be divorced entirely from the rest of the system, but the remaining interdependencies should be minimal.

4.6.1 Distributed Control

In the hierarchical storage system described above, all storage management operations can be determined local to a single level or, at most, in consideration of information from neighboring levels. Thus, it is possible to distribute the control of the hierarchy into the levels, this also facilitates parallel and asynchronous operation in the hierarchy.

In a comprehensive multiple level storage hierarchy, as illustrated in Table 1, this automatic and distributed control can be accomplished by using two mechanisms: (1) processor functions, and (2) "intelligent" controllers.

4.6.1.1 Processor Functions

The management of the first storage level must operate at speeds comparable to the processor. As a result, it is usually necessary to incorporate the first level store and its associated management operations into the processor hardware itself. This approach is used in the IBM System/370 cache systems [52].

It is often desirable to incorporate the management of the second storage level also into the processor. This level requires substantial performance to handle the demands for service from the first storage level. Since its requirements are not quite as demanding as the first level, it is an ideal candidate for firmware control, assuming that the processor is microprogrammed. This approach has not been used in any current commercial systems, although the integrated (i.e., microprogrammed) channels of certain models of the IBM System/370 are based upon similar concepts. There have been a few experimental systems, such as the VENUS System at MITRE, which provides processor functions to essentually manage the paging system via microprogramming.

4.6.1.2 "Intelligent" Controllers

For the third storage level and beyond, the storage management performance requirements are much more modest since most of the storage activity should occur at the first and second levels. For these lower levels, it is possible to develop independent storage management control facilities for each level. This can be accomplished by extending the functionality of conventional device controllers. Some recent sophisticated device controllers are microprogrammed and are already capable of performing the storage management function [1].

4.6.2 Multiprogramming

Up to now we have tacitly assumed that the processor becomes idle whenever it is necessary to fetch information from the storage hierarchy. This may be a reasonable policy for two-level cache systems since the processor is never idle for more than one or two microseconds at a time. But, for paging systems and general multiple level storage hierarchies, the processor may be idled for periods of hundreds or thousands of microseconds at a time. It is worthwhile to try to find useful work for the processor while the storage hierarchy is retrieving the requested information.

In most conventional computer systems, processor idle time is utilized by multiprogramming. This requires that there be multiple programs available to be run. Whenever one program must be delayed due to a time-consuming storage request, the processor is switched to another program. Under reasonable circumstances (e.g., many programs ready for execution and moderate load on the storage system), it is possible to keep the processor continually busy. Thus, the effective system storage access time, T', will very closely approximate T^1.

Unfortunately, the process of switching execution from one program to another can result in a considerable amount of processor overhead. For example, an early version of the Multics operating system was reported to require 10 milliseconds to switch programs; typical operating systems require up to 1 millisecond. The time required to accomplish this multiprogram switch can be drastically reduced if the multiprogramming management is also incorporated into the processor along with the first and second storage level management. Although the particular purposes were different, hardware supported multiprogramming has been available on several computing systems, such as the Honeywell 800 series [46] and more recently in the Singer

System Ten [30]. The less frequently executed operating system functions, such as job scheduling and time-sharing management algorithms, can be supported by the software operating system as on conventioal systems without adversely affecting performance.

4.7 Comments on the Storage Hierarchy System Design

This chapter has presented the key concepts of a general multiple level storage hierarchy system. Many of the particular details of the system will require considerable investigation and experimentation to determine an optimal implementation. Three important factors are extensively studied in the following chapters: (1) other page size considerations, (2) removal algorithms, and (3) relevant models for program reference behavior.

CHAPTER 5.
ANALYSIS OF PAGE SIZE CONSIDERATIONS

5.0 Introduction

One of the most important parameters of a storage hierarchy system is the page size, the unit of information transfer between two levels of the hierarchy. In this chapter, the factors influencing page size are examined from the device characteristics viewpoint and the program behavior viewpoint.

5.1 The Page Size Issue

On contemporary two-level paging systems (based upon two devices similar to devices 2 and 4 of Table 1), the page size is usually quite large (typically 4096 bytes for paging systems) to take advantage of M^2's large transfer rate to compensate for its slow access time. Such a large page size is justified by reliance on the Principle of Locality. Considering the devices of Table 1 for example, a single byte can be accessed and transferred between M^1 and M^2 in about 5 milliseconds whereas 4096 contiguous bytes can be fetched in 7.8 milliseconds, only 56% more time.

5.1.1 Page Size Investigations

Although paging systems have been used successfully, the effect of page size has become the subject of increasing investigation. This interest has been aroused due to several considerations:

1. It has been noted by Denning [26] that the utilization of M^1 is maximized and "page breakage" minimized by using rather small pages, such as 200 bytes. In particular, he emphasizes:

> "These results are significant ... small pages permit a great deal of compression without loss of efficiency. Small page sizes will yield significant improvements in storage utilization ..."

2. The success of cache systems indicates that the Principle of Locality applies on the microscopic scale as well as the macroscopic scale of conventional paging systems.

3. The recent introduction of several new device technologies, such as the "semiconductor drum" [35] with an average access time of about 100 microseconds, drastically reduces the benefits of very large page sizes in a paging system.

4. Although most current multilevel systems employ only two levels, this thesis is concerned with multiple

level storage hierarchies (i.e., three or more levels). In fact, storage systems with six or more levels are quite plausible. A deep understanding of the effects of various page sizes is essential to the development of such systems.

Thus, although there are many reasons for considering new page sizes, there is not a complete understanding of the impact of such a change. Denning [26] sums up our current knowledge as follows:

> "Two factors primarily influence the choice of page size: fragmentation and efficiency of page-transport operation."

In this chapter some other factors of potentially crucial importance will be discussed.

5.2 Anomalies

One of the more intriguing and frustrating aspects of complex systems, such as paging systems, is the occurrence of anomalies (i.e., phenomena that are contrary to "common sense"). For example, Belady [10] has shown that certain storage management removal algorithms, in particular FIFO (first-in first-out), may actually cause performance to decrease as the capacity of M^1 is increased. This result is contrary to the general belief that "more main memory makes things work out better". Thus, one must exercise considerable care when considering "tinkering" with the parameters, such as page size, of a multilevel storage system.

The objective of this chapter is to present and analyze some anomalies encountered when the page size parameter is changed in a paging system.

5.3 The Page Size Anomaly

For simplicity, let us start by considering the effect of decreasing the page size used in a two-level system, S, from N to N' where N' = N/2 in this new system, S'. In particular, we wish to investigate the effects upon the failure frequencies which are f and f', respectively. We define the ratio f'/f to be r. The possible results can be partitioned into three interesting regions:

1. $r < 1$.
2. $1 \le r \le 2$.
3. $r > 2$.

5.3.1 Case 1: $r < 1$ ($f' < f$).

This would be a highly desirable result since the number of page fetches is actually decreased. Furthermore, the time required to access and transfer a page of size N' would be expected to be less than that required for the

<u>Parameters</u>

As seen by S:

- P = a, b, c, a, b, c
- $|P|$ = 6
- Q = { a, b, c }
- $|Q|$ = 3
- $|M^1|$ = 2
- FIFO Removal

As seen by S':

- P = a^+, b^+, c^+, a^+, b^+, c^+
- $|P|$ = 6
- Q = { a^+, b^+, c^+ }
- $|Q|$ = 3
- $|M^1|$ = 4
- FIFO Removal

<u>Simulation</u>

Page Trace:	a^+	b^+	c^+	a^+	b^+	c^+
<u>S</u>						
Fetch:	*	*	*	*	*	*
M^1 Contents:	a	b	c	a	b	c
		a	b	c	a	b
<u>S'</u>						
Fetch:	*	*	*			
M^1 Contents:	a^+	b^+	c^+	c^+	c^+	c^+
		a^+	b^+	b^+	b^+	b^+
			a^+	a^+	a^+	a^+

<u>Results</u>

- $F = 6$
- $F' = 3$
- $r = 3/6 = 0.5$

Figure 6.
Example of Case 1

larger page size N. Figure 6 illustrates an instance of this case. In converting an address trace to a page trace for N', the logical page addresses p^+ and p^- are used to represent the two halves of the page p of size N. Note that when using a page size of N/2 instead of N, M^1 actually holds twice as many pages though each page is only half as large.

In the example of Figure 6, r = 0.5, which means that the number of page fetches was cut in half by using the smaller page size N'. This type of result might be expected from a program that exhibited a rather sparse and non-localized reference behavior. Recall that in typical two-level paging systems, a page of size 4096 bytes is fetched even though a single reference uses only a few bytes. Unless the program immediately makes many more references to this page, much of it will have been fetched but not used. Under these circumstances, M^1 might be better utilized by holding a larger and more diversified collection of pages, even if each page were smaller.

5.3.2 Case 2: $1 \leq r \leq 2$ $(f \leq f' \leq 2f)$

This is a transitional region. For r = 1, S' will perform better than S since the number of page fetches is the same and the time required for each fetch is less. For r = 2, S' will require twice as many page fetches. This will usually swamp any page transfer benefit derived from the smaller page size, thus S would perform better. The specific point of transition, r', depends largely upon the time required to access and transfer a page, T and T' respectively in S and S', such That r' = T/T'.

Figure 7 illustrates an extreme example of Case 2 where r = 2.0. This means that the number of page fetches was doubled by using the smaller page size N'. This type of result might be expected from a program that exhibited a dense, localized, and sequential reference behavior.

Intuitively, the r = 2.0 result is the "worst" case since we are being forced to always load both the p^+ and p^- halves of each original page p, thereby losing all the benefits of the smaller N' page size and incurring twice as many actual page faults. This intuitive observation is false; r = 2.0 is not the "worst" case.

5.3.3 Case 3: r > 2 (f' > 2f)

This third region, besides being intuitively impossible, is clearly undesirable. Since the number of page fetches required would be more than doubled, the performance of S' would be undoubtedly worse than S. Depending upon the actual value of r, the performance could be much worse. Figure 8 illustrates a reference pattern that produces a result of r = 2.75. This region of operation will be the

(f5)

__Parameters__

As seen by S:

- $P = a, a, b, b, c, c$
- $|P| = 6$
- $Q = \{ a, b, c \}$
- $|Q| = 3$
- $|M^1| = 2$
- FIFO Removal

As seen by S':

- $P = a^+, a^-, b^+, b^-, c^+, c^-$
- $|P| = 6$
- $Q = \{a^+, a^-, b^+, b^-, c^+, c^- \}$
- $|Q| = 6$
- $|M^1| = 4$
- FIFO Removal

__Simulation__

```
Page Trace:   a+ a- b+ b- c+ c-
_S_
Fetch:        *     *     *
M1 Contents:  a  a  b  b  c  c
                    a  a  b  b
_S'_
Fetch:        *  *  *  *  *  *
M1 Contents:  a+ a- b+ b- c+ c-
                 a+ a- b+ b- c+
                    a+ a- b+ b-
                       a+ a- b+
```

__Results__

- $F = 3$
- $F' = 6$
- $r = 6/3 = 2.0$

Figure 7.
Example of Case 2

(f6)

<u>Parameters</u>

As seen by S:

- P = a, b, a, b, c, c, b, a, a, c, c
- $|P|$ = 11
- Q = { a, b, c }
- $|Q|$ = 3
- $|M^1|$ = 2
- FIFO Removal

As seen by S':

- P = a^+, b^+, a^-, b^-, c^+, c^-, b^+, a^+, a^-, c^+, c^-
- $|P|$ = 11
- Q = { a^+, a^-, b^+, b^-, c^+, c^- }
- $|Q|$ = 6
- $|M^1|$ = 4
- FIFO Removal

<u>Simulation</u>

Page Trace:	a^+	b^+	a^-	b^-	c^+	c^-	b^+	a^+	a^-	c^+	c^-
<u>S</u>											
Fetch:	*	*			*			*			
M^1 Contents:	a	b	b	b	c	c	c	a	a	a	a
		a	a	a	b	b	b	c	c	c	c
<u>S'</u>											
Fetch:	*	*	*	*	*	*	*	*	*	*	*
M^1 Contents:	a^+	b^+	a^-	b^-	c^+	c^-	b^+	a^+	a^-	c^+	c^-
		a^+	b^+	a^-	b^-	c^+	c^-	b^+	a^+	a^-	c^+
			a^+	b^+	a^-	b^-	c^+	c^-	b^+	a^+	a^-
				a^+	b^+	a^-	b^-	c^+	c^-	b^+	a^+

<u>Results</u>

- $F = 4$
- $F' = 11$
- $r = 11/4 = 2.75$

Figure 8.
Example of Case 3

subject of discussion for the remainder of this chapter. We formalize this situation by the following existence theorem.

(th1)

THEOREM 1:

There exists a page trace, P, and demand-fetch FIFO-removal two-level storage systems, S and S', with page sizes N and N'=N/2, respectively, such that the ratio, r, of fetch frequency f' to f exceeds 2.

Proof:

By example (Figure 8).

5.3.4 Other Removal Algorithms

Theorem 1 states the anomaly that decreasing page size by a factor of two can cause the page fetch frequency to increase by more than a factor of two. The two-level demand-fetch conditions of Theorem 1 are typical of most contemporary paging systems. But, to put this situation into perspective, other removal algorithms must be considered. Due to its simplicity, the FIFO removal algorithm was used in many of the early paging systems. In recent times it has been found that FIFO has certain disturbing pecularities (e.g., the system's success frequency, s, is not a monotonic function of primary store size, $|M^1|$ [10]). Furthermore, other removal algorithms have been found to be empirically closer approximations to the "optimal" removal algorithm, MIN [11]. MIN itself is not physically realizable since it requires future knowledge, but it can be used as a basis for performance comparison with practical algorithms.

Various forms of the "least recently used" (LRU) removal algorithm have become popular in contemporary systems. Under LRU, the page selected for removal from the primary store is the one that has not been referenced for the longest time (i.e., the least recently used page). Empirically, LRU has been found to closely approximate the performance of the "optimal" algorithm for many actual programs. Furthermore, Mattson et al [63] have studied LRU and found that it is a member of a general class of removal algorithms called "stack algorithms". The class of stack algorithms, as noted by Denning [25], "contains all the 'reasonable' algorithms". In particular, stack algorithms all satisfy an inclusion property that results in well behaved characteristics. For example, it has been proven that all stack algorithms, including LRU, have a success frequency that is a monotonic function of primary store size and immune to the FIFO pecularariarity observed by Belady. Thus, one might be tempted to assume that the page size anomaly is also a phenomenom unique to FIFO removal and would not occur if a "well behaved" removal algorithm, such as LRU, were used. This expectation can be rapidly destroyed by observing Figure 9, which is the same system as Figure 8 but with an LRU removal algorithm. In this example, the page fetch frequency ratio, r, is 2.2 which still exceeds 2. This result leads us to Theorem 2 and Corollary 2a.

(th2)

THEOREM 2:

There exists a page trace, P, and demand-fetch LRU-removal two-level storage systems, S and S', with page sizes N and N'=N/2, respectively, such that the ratio, r, of fetch frequency f' to f exceeds 2.

Proof:

By example (Figure 9).

COROLLARY 2a:

Given a page trace, P, and demand-fetch two-level storage systems, S and S', with page sizes N and N'=N/2, respectively, the use of a "stack" removal algorithm (i.e., an algorithm with the "inclusion property") is not sufficient to guarantee that the ratio, r, of fetch frequency f' to f will be bounded by 2.

(f7)

<u>Parameters</u>

As seen by S:

- P = a, b, a, b, c, c, b, a, a, c, c
- $|P|$ = 11
- Q = { a, b, c }
- $|Q|$ = 3
- $|M^1|$ = 2
- LRU Removal

As seen by S':

- P = a^+, b^+, a^-, b^-, c^+, c^-, b^+, a^+, a^-, c^+, c^-
- $|P|$ = 11
- Q = { a^+, a^-, b^+, b^-, c^+, c^- }
- $|Q|$ = 6
- $|M^1|$ = 4
- LRU Removal

<u>Simulation</u>

Page Trace:	a^+	b^+	a^-	b^-	c^+	c^-	b^+	a^+	a^-	c^+	c^-
<u>S</u>											
Fetch:	*	*			*			*		*	
M^1 Contents:	a	b	a	b	c	c	b	a	a	c	c
		a	b	a	b	b	c	b	b	a	a
<u>S'</u>											
Fetch:	*	*	*	*	*	*	*	*	*	*	*
M^1 Contents:	a^+	b^+	a^-	b^-	c^+	c^-	b^+	a^+	a^-	c^+	c^-
		a^+	b^+	a^-	b^-	c^+	c^-	b^+	a^+	a^-	c^+
			a^+	b^+	a^-	b^-	c^+	c^-	b^+	a^+	a^-
				a^+	b^+	a^-	b^-	c^+	c^-	b^+	a^+

<u>Results</u>

- $F = 5$
- $F' = 11$
- $r = 11/5 = 2.2$

Figure 9.
Example of Case 3
(for LRU Removal)

5.4 Significance of the Page Size Anomaly

The previous theorems prove that there exist page traces that result in significantly increased page fetch frequencies if the page size is decreased. It is necessary to consider the likelihood of encountering such page trace patterns in actual programs. For example, it can be proven that, as you are reading this sentence, all the molecules of air in the room may suddenly move towards the opposite corner and cause you to suffocate. If you survived the last sentence, you have probably deduced that the likelihood of that event is extremely small, fortunately.

5.4.1 Simulation Studies

Hatfield [48] and Seligman [78] have performed experiments that indicate that the page size anomaly is very common, if not inevitable, in actual programs. In both cases actual programs were monitored and their corresponding page trace reference strings were recorded, usually on magnetic tape. Then simulators were developed that mimicked the software and hardware of the two-level storage systems then in use or being considered. By supplying the monitored page traces as inputs to the simulators, the performance of such a system can be accurately measured. These simulators were scrupulously accurate, not just approximations. The validity of these results have been confirmed in some cases by running the real programs under a real two-level storage system.

5.4.2 Hatfield Studies

Hatfield [48] performed studies in the hardware environment of the IBM System/360 Model 67 with programs running under the CP-67/CMS Operating System. The simulated performance was measured for various page sizes, N, and various primary store sizes, $|M^1|$. In summary, it was confirmed that certain programs, which were viewed as examples of low-density storage use, resulted in decreased page fetch frequency when page size was decreased. But, it was observed that for programs with much greater localization of heavily used storage:

> "not only does the smaller page size often generate nearly twice as many page fetches as the large page size, it often resulted in more than twice the page fetches, contrary to our intuitions."

In particular, the substantially increased page fetch frequency appears to be:

> "a characteristic of programs which have a high locality and therefore perform well on systems using relocation hardware for address translation and is characteristic of those programs in the region of low paging rate."

In other words, the anomaly is most prevalent in programs "optimized" for performance in a two-level storage system when running under nearly "optimal" conditions!

5.4.3 Seligman Studies

Whereas Hatfield was concerned with a paging system with page sizes in the range from 2048 to 16384 bytes, Seligman [78] analyzed a proposed cache system with much smaller page sizes in the range of 8 to 256 bytes. He observed that:

> "interestingly, the missing page probability (for this data) is minimized for a page size which increases slowly with total memory size. Note that the associative memory organization, where page size equals one word, is not optimum; to borrow a phrase from economics, the marginal utility of the extra words fetched in a page is higher than that of those displaced".

Thus, continual decreasing of page size appears to have an inevitable adverse effect upon system performance.

5.4.4 Other Questions Raised

Now that it has been shown that the page size anomaly is theoretically possible and likely to occur in practice, there are several other questions of interest. Since it has been proven that the page fetch frequency ratio is not bounded by $r = 2$, what bounds, if any, do exist? Hatfield implicitly raised another question by the statement:

"as yet we have been unable to prove that there is a replacement algorithm using only the past history of page requests which cannot generate more than twice the exceptions with half size pages."

The answers to these questions are the subjects of the following sections and chapters.

5.5 Bounds on the Page Fetch Frequency Ratio

It has been shown that the page fetch frequency ratio can exceed $r = 2$, but just how bad can it get? Of equal importance, what factors influence this bound? These questions will be discussed in this section.

5.5.1 Cyclic Page Traces

Figures 10 and 11 represent page trace simulations for two sets of demand-fetch LRU-removal two-level storage systems with primary store sizes $|M^1|=2$ and $|M^1|=3$, respectively. In both cases, it can be observed that the page trace simulated is cyclic with a repeated pattern, Pc. In Figure 10, the page trace consists of the repeated pattern:

$$Pc = a^+ \quad b^+ \quad c^+ \quad c^- \quad b^- \quad a^-$$

(f8)

Parameters

As seen by S:

- P = a, b, c, c, b, a, a, b, c, c, b, a
- $|P|$ = 12
- Q = { a, b, c }
- $|Q|$ = 3
- $|M^1|$ = 2
- LRU Removal

As seen by S':

- P = a^+, b^+, c^+, c^-, b^-, a^-, a^+, b^+, c^+, c^-, b^-, a^-
- $|P|$ = 12
- Q = { a^+, a^-, b^+, b^-, c^+, c^- }
- $|Q|$ = 6
- $|M^1|$ = 4
- LRU Removal

Simulation

	transient cycle						steady-state cycle					
Page Trace:	a^+	b^+	c^+	c^-	b^-	a^-	a^+	b^+	c^+	c^-	b^-	a^-
S												
Fetch:	*	*	*			*			*			*
M^1 Contents:	a	b	c	c	b	a	a	b	c	c	b	a
		a	b	b	c	b	b	a	b	b	c	b
S'												
Fetch:	*	*	*	*	*	*	*	*	*	*	*	*
M^1 Contents:	a^+	b^+	c^+	c^-	b^-	a^-	a^+	b^+	c^+	c^-	b^-	a^-
		a^+	b^+	c^+	c^-	b^-	a^-	a^+	b^+	c^+	c^-	b^-
			a^+	b^+	c^+	c^-	b^-	a^-	a^+	b^+	c^+	c^-
				a^+	b^+	c^+	c^-	b^-	a^-	a^+	b^+	c^+
						↑		same				↑

Results

- F = 6
- F' = 12
- r = 12/6 = 2.0

For the steady-state cycle:

- F = 2
- F' = 6
- /r/ = 6/2 = 3.0

Figure 10.
Cyclic Page Trace with $|M^1|$ = 2

whereas Figure 11 repeats the similar pattern:

$$Pc = a^{+} \quad b^{+} \quad c^{+} \quad d^{+} \quad d^{-} \quad c^{-} \quad b^{-} \quad a^{-}$$

5.5.2 Steady State Cyclic Page Traces

Let us consider Figure 10 first. The page fetch ratio, r, is 2.0 in this case. As noted earlier, the page trace can be subdivided into an initial transient stage, Pt, with a high page fetch frequency followed by a steady-state stage, Ps, with usually a lower page fetch frequency. In Figure 10, the first Pc cycle contains the entire start-up transient stage and completely fills all the available space in M^1. Thus, the second Pc cycle represents the start of the steady-state stage. Furthermore, since the content and page ordering of M^1 is exactly the same at the end of the second cycle as they were at the beginning of that cycle for both S and S', the page trace cycle, Pc, can be repeated continuously with exactly the same results each time for page fetch requests and M^1 contents. If /r/ is defined to be the page fetch frequency ratio for the first steady-state period, Pc, of a cyclic page trace, (Pc)*, /r/ is also the page fetch frequency ratio for the entire steady-state portion of the page trace defined by the regular expression:

$$P = Pt \bullet Ps = Pt \bullet (Pc)^{*}$$

As the length of the page trace, |P|, becomes large in comparison with the length of the transient stage, |Pt|, the overall page fetch frequency ratio, r, asymptotically approaches the value of the steady-state cycle page fetch frequency ratio, /r/. In Figure 10, /r/ = 3.0, thus r will increase from 2.0 towards 3.0 as the page trace is lengthened by continually repeating the pattern Pc. Thus, the page fetch frequency ratio, r, for the page trace

$$P = (a^{+} \quad b^{+} \quad c^{+} \quad c^{-} \quad b^{-} \quad a^{-})^{*}$$

is bounded by 3.0 when $|M^1| = 2$.

A similar situation is illustrated in Figure 11. In this example, r = 2.28 and /r/ = 4.0. Thus, the page fetch frequency ratio, r, for the page trace

$$P = (a^{+} \quad b^{+} \quad c^{+} \quad d^{+} \quad d^{-} \quad c^{-} \quad b^{-} \quad a^{-})^{*}$$

is bounded by 4.0 when $|M^1| = 3$. By generalizing these examples, we arrive at Theorem 3 and Corollary 3a.

(th3)

THEOREM 3:

For any two demand-fetch LRU-removal two-level storage systems, S and S', with page sizes N and N'=N/2 and primary store sizes $|M^1|$ and $|M^1|' = 2|M^1|$, respectively, there exists a cyclic page trace, P = (Pc)*, where |Pc| = $2(|M^1|+1)$, such that the steady-state page fetch frequency ratio, /r/, equals $|M^1|+1$.

Proof:

(See below).

(f9)

Parameters

As seen by S:

- P = a,b,c,d,d,c,b,a,a,b,c,d,d,c,b,a
- $|P|$ = 16
- Q = { a, b, c, d }
- $|Q|$ = 4
- $|M^1|$ = 3
- LRU Removal

As seen by S':

- P = $a^+,b^+,c^+,d^+,d^-,c^-,b^-,a^-,a^+,b^+,c^+,d^+,d^-,c^-,b^-,a^-$
- $|P|$ = 16
- Q = { a^+, a^-, b^+, b^-, c^+, c^-, d^+, d^- }
- $|Q|$ = 8
- $|M^1|$ = 6
- LRU Removal

Simulation

	|←			transient cycle				→|	|←			steady-state cycle				→|
Page Trace:	a^+	b^+	c^+	d^+	d^-	c^-	b^-	a^-	a^+	b^+	c^+	d^+	d^-	c^-	b^-	a^-
_S_																
Fetch:	*	*	*	*				*				*				*
M^1 Contents:	a	b	c	d	d	c	b	a	a	b	c	d	d	c	b	a
		a	b	c	c	d	c	b	b	a	b	c	c	d	c	b
			a	b	b	b	d	c	c	c	a	b	b	b	d	c
_S'_																
Fetch:	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
M^1 Contents:	a^+	b^+	c^+	d^+	d^-	c^-	b^-	a^-	a^+	b^+	c^+	d^+	d^-	c^-	b^-	a^-
		a^+	b^+	c^+	d^+	d^-	c^-	b^-	a^-	a^+	b^+	c^+	d^+	d^-	c^-	b^-
			a^+	b^+	c^+	d^+	d^-	c^-	b^-	a^-	a^+	b^+	c^+	d^+	d^-	c^-
				a^+	b^+	c^+	d^+	d^-	c^-	b^-	a^-	a^+	b^+	c^+	d^+	d^-
					a^+	b^+	c^+	d^+	d^-	c^-	b^-	a^-	a^+	b^+	c^+	d^+
						a^+	b^+	c^+	d^+	d^-	c^-	b^-	a^-	a^+	b^+	c^+
								↑				same				↑

Results

- F = 7
- F' = 16
- r = 16/7 = 2.28

For the steady-state cycle:

- F = 2
- F' = 8
- $/r/$ = 8/2 = 4.0

Figure 11.
Cyclic Page Trace with $|M^1|$ = 3

COROLLARY 3a:

For any two demand-fetch LRU-removal two-level storage systems, S and S', with page sizes N and $N'=N/2$ and primary store sizes $|M^1|$ and $|M^1|'=2|M^1|$, respectively, there exists a cyclic page trace, $P = (Pc)^*$, where $|Pc| = 2(|M^1|+1)$, such that the overall page fetch frequency ratio, r, asymptotically approaches the bound $|M^1|+1$ as $|P|$ approaches infinity.

5.5.3 Proof of Theorem 3

5.5.3.1 Notation and Properties

Assume a fixed page size N and primary store of size S^1, let n = the number of pages in M^1 (i.e., $n = |M^1| = S^1/N$). It has been shown by Mattson et al [63] that a demand-fetch LRU-removal algorithm has the following properties:

P1. If M^1 is initially empty, it fills with the first n distinct pages referenced by the trace.

P2. At any time t, M^1 contains the n most recently referenced distinct pages.

P3. a) LRU satisfies the <u>inclusion property</u>

$$M^1(1) \subset M^1(2) \subset \ldots \subset M^1(m)$$

where $M^1(1)$ means the contents of M^1 if $n=1$, etc.

b) At any time t after M^1 has become filled, there is a strict removal ordering referred to as the <u>LRU stack</u>

$$S = \{ s(1), s(2), \ldots, s(n) \}$$

where

$$s(i) = M^1(i) - M^1(i-1) \quad \text{for } i = 1, 2, \ldots, n$$

and $s(n)$ is the page to be removed next.

5.5.3.2 Definition 3-a:

For any integer n, let us consider a page trace, P^o, consisting of the repeated pattern, Pc^o, of length $|Pc^o| = 2(n+1)$

$$P^o = Pc^o[n]^*$$

where

$$Pc^o[n] = \{ Pc^o(1), Pc^o(2), \ldots, Pc^o(2n+1), Pc^o(2n+2) \}.$$

The $Pc^o(i)$s are defined as follows:

$$Pc^o(i) = \begin{cases} 2(i-1) & \text{for } i = 1, \ldots, n+1 \\ 4n+5-2i & \text{for } i = n+2, \ldots, 2n+2 \end{cases}$$

Thus, for n = 2 --

$$Pc^o[2] = \{ 0, 2, 4, 5, 3, 1 \}$$

and

$$P^o[2] = \{ 0, 2, 4, 5, 3, 1, 0, 2, 4, 5, 3, 1, \ldots \}$$

The cyclic page trace pattern, $Pc^o[n]$, is used to define corresponding cyclic page trace patterns, Pc[n] and Pc'[n], for S and S', respectively. These are defined as follows --

For a given value of n and i = 1, 2, ..., 2n+2

$$Pc(i) = \textbf{integer}[Pc^o(i)/2]$$

$$Pc'(i) = \begin{cases} (\textbf{integer}[Pc^o(i)/2])^+ & \text{if } \textbf{rem}[Pc^o(i)/2]=0 \\ (\textbf{integer}[Pc^o(i)/2])^- & \text{if } \textbf{rem}[Pc^o(i)/2]=1 \end{cases}$$

Thus, for n = 2 --

$$P[2] = \{ 0, 1, 2, 2, 1, 0, 0, 1, 2, 2, 1, 0, \ldots \}$$

$$P'[2] = \{ 0^+, 1^+, 2^+, 2^-, 1^-, 0^-, 0^+, 1^+, 2^+, 2^-, 1^-, 0^-, \ldots \}$$

We can see that these page traces are identical to the page traces of Figure 8 with appropriate relabeling (i.e., a=0, b=1, c=2).

5.5.3.3 Lemma 3-b:

The page references of the set

$$\{ Pc(1), \ldots, Pc(n+1) \}$$

are distinct.

Proof:

Based upon the definitions of $Pc^o[n]$ and $Pc[n]$, we see that

For i = 1, ..., n+1

$$\begin{aligned} Pc(i) &= \textbf{integer}[Pc^o(i)/2] \\ &= \textbf{integer}[2(i-1)/2] \\ &= \textbf{integer}[i-1] \end{aligned}$$

$= i-1.$

Thus, each value of $Pc(i)$ for $i = 1, \ldots, n+1$ is distinct.

Q.E.D.

5.5.3.4 Lemma 3-c:

The page references of the set

$$\{ Pc(n+2), \ldots, Pc(2n+2) \}$$

are distinct.

Proof:

Based upon the definitions of $Pc^o[n]$ and $Pc[n]$, we see that

For $i = n+2, \ldots, 2n+2$

$$\begin{aligned} Pc(i) &= \textbf{integer}[Pc^o(i)/2] \\ &= \textbf{integer}[(4n+5-2i)/2] \\ &= \textbf{integer}[2n+2+(1/2)-i] \\ &= 2n+2-i \end{aligned}$$

Thus, each value of $Pc(i)$ for $i = n+2, \ldots, 2n+2$ is distinct.

Q.E.D.

5.5.3.5 Lemma 3-d:

At the end of each cycle, $Pc[n]$, of the page trace, $P[n]$, M^1 contains the pages, in LRU stack order,

$$S^o = \{ s^o(1), \ldots, s^o(n) \}$$

where

$$s^{o}(j) = j-1 \qquad \text{for } j = 1, \ldots, n$$

Proof:

Since each cycle, Pc[n], of P[n] is of length $2n+2$ which is greater that n, the S^{o} LRU stack consists of the last n page references of Pc[n] in reverse order by property P2, P3, and Lemma 3-c. Thus,

$$s^{o}(j) = Pc(2n+3-j)$$

such that

$$s^{o}(1) = Pc(2n+2),\ s^{o}(2) = Pc(2n+1),\ \ldots,\ s^{o}(n) = Pc(n+3).$$

When j takes on values { 1, ..., n }, $2n+3-j$ takes on values { $2n+2$, ..., $n+3$ }. Thus, for $j = 1, \ldots, n$ and based upon Lemma 3-c:

$$\begin{aligned} s^{o}(j) &= Pc(2n+3-j) \\ &= 2n+2-(2n+3-j) \\ &= j-1. \end{aligned}$$

Q.E.D.

5.5.3.6 Lemma 3-e:

Given a demand-fetch LRU-removal two-level storage system, S, with page size N, primary store size S^{1} containing $n=S^{1}/N$ pages, the page fetch function, F, resulting from each steady-state cycle, Pc[n], of the page trace P has the value 2 (i.e., F[Pc[n]]=2 during steady state).

Proof:

Let us subdivide the Pc[n] cycle, which is of length 2n+2, into four regions as follows:

Region 1: $Pc^1 = \{ Pc(1), \ldots, Pc(n) \}$

Region 2: $Pc^2 = \{ Pc(n+1) \}$

Region 3: $Pc^3 = \{ Pc(n+2), \ldots, Pc(2n+1) \}$

Region 4: $Pc^4 = \{ Pc(2n+2) \}$.

and compute the number of page fetches in each region, F^1, F^2, F^3, F^4, respectively. Since the page trace regions are concatenated, the page fetches are cumulative, so we know that

$$F = F^1 + F^2 + F^3 + F^4.$$

Region 1: $Pc^1 = \{ Pc(1), \ldots, Pc(n) \}$

From Lemma 3-b, we know that

$$Pc(i) = i-1 \qquad i = 1, \ldots, n+1$$

and from Lemma 3-d, we know that at the beginning of each cycle

$$S^0(j) = j-1 \qquad j = 1, \ldots, n.$$

The page references $\{ Pc(1), \ldots, Pc(n) \}$ are actually the sequence $\{ 0, \ldots, n-1 \}$ which is identical to the contents of M^1 at the start of the cycle, S^0. Therefore, no page transfers are required although LRU stack reordering may occur. (<u>$F^1=0$</u>).

Region 2: $Pc^2 = \{ Pc(n+1) \}$

Page reference $Pc(n+1)$ is page n which is not contained in S^0 nor loaded during region 1 (in fact, no pages were fetched during region 1); thus, a page transfer is required (<u>$F^2=1$</u>). Using similar techniques as in Lemma 3-d, since each reference of Pc^1 is distinct, the LRU removal stack at this point is

$$S = \{ s(1), \ldots, s(n) \}$$

where

$$s(j) = Pc(n+1-j) \qquad j = 1, \ldots, n.$$

Page $s(n)$ is selected for removal, this is actually page $Pc(n+1-n)=Pc(1)=0$. The new LRU stack ordering becomes

$$s(j) = Pc(n+2-j) \qquad j = 1, \ldots, n.$$

Region 3: $Pc^3 = \{ Pc(n+2), \ldots, Pc(2n+1) \}$

The page references $\{ Pc(n+2), \ldots, Pc(2n+1) \}$ are actually the sequence $\{ n, \ldots, 1 \}$ as shown in the proof of Lemma 3-b. The LRU stack ordering immediately prior to reference $Pc(n+2)$ is

$$S^{oo} = \{ s(1), \ldots, s(n) \}$$

which is actually

$$\{ n, \ldots, 1 \}$$

since it has been shown earlier that at reference $Pc(n+2)$

$$s(j) = Pc(n+2-j) \qquad j = 1, \ldots, n.$$

Thus, as in region 1, every page referenced is already contained in M^1 and there are no page transfers required (<u>$F^3=0$</u>).

Region 4: $Pc^4 = \{ Pc(2n+2) \}$

Page reference $Pc(2n+2)$ is actually page 0. This page was not contained in S^{oo}, thus a page transfer is required

($\underline{F^4=1}$).

Therefore, we can conclude

$$F[Pc[n]] = F^1[Pc^1] + F^2[Pc^2] + F^3[Pc^3] + F^4[Pc^4]$$
$$= 0 + 1 + 0 + 1$$
$$= 2.$$

Q.E.D.

5.5.3.7 Lemma 3-f:

Given a demand-fetch LRU-removal two-level storage system, S', with page size N'=N/2, primary store size [M^1] containing 2n=[M^1]/(N/2) pages, the page fetch function, F', resulting from each steady-state cycle, Pc'[n], of the page trace P' has the value 2n+2 (i.e., F'[Pc'[n]]=2n+2 during steady state).

Proof:

The proof follows directly from the definition of P', the LRU properties, and the previous Lemmas.

- Each page reference in the cyclic pattern Pc'[n] is distinct. (This can be easily seen from the definition or proven in a similar manner to Lemmas 3-b and 3-c).
- Each cycle is 2n+2 references long.
- At any time t, page reference P'(t) = P'(t-2n-2).
- The primary store, M^1, can hold 2n pages in S' since N'=N/2.

- Since the cyclic pattern only repeats after $2n+2$ steps and M^1 is only $2n$ pages large, M^1 always holds the last $2n$ page references (since they are distinct).
- Thus, at any time t, page reference $P'(t)$ will not correspond to any page currently in M^1 (i.e., M^1 holds references $\{ P'(t-1), \ldots, P'(t-2n) \}$ and $P'(t)=P'(t-2n-2)$ is not in that set). As a result, a page fetch is required for every page reference.
- Since there are $2n+2$ page references per cycle, there are $2n+2$ page fetches required per cycle. Thus, $F'=2n+2$.

Q.E.D.

5.5.3.8 Theorem 3:

For any two demand-fetch LRU-removal two-level storage systems, S and S', with page sizes N and $N'=N/2$ and primary store sizes $|M^1|'=2|M^1|$, respectively, there exists a cyclic page trace, $P=(Pc)^*$, where $|Pc|=2(|M^1|+1)$, such that the steady-state page fetch frequency ratio, /r/, equals $|M^1|+1$.

Proof:

This proof follows trivially from Lemmas 3-e and 3-f. We know that for each steady-state cycle of S, $F=2$ (Lemma 3-e). Also, for each steady-state cycle of S', $F=2n+2$ (Lemma 3-f). Since the page fetch frequency ratio, r, is defined as f'/f or $(F'/|P|)/(F/|P|)$ which equals F'/F, we find that in steady-state

$$/r/ = F'/F = (2n+2)/2 = n+1.$$

Q.E.D.

5.5.4 Comments on Theorem 3

The above results expose another facet of the page size anomaly. As the size of the primary store, M^1, is increased, the overall page fetch frequency ratio as stated in Corollary 3a also increases. This means that the larger the primary store that you have, the more "dangerous" the page size anomaly becomes. For example, in a two-level paging system based on devices 2 and 4 from Table 1, $|M^1| = 128$ pages and $N = 4096$ bytes, if the page size is decreased by half to 2048 bytes, it is possible that the page fetch frequency would increase 129-fold (a 12,800% increase in paging activity!). Of course, one would assume, or at least hope, that such pathological page trace patterns would be very rare, but we know that they can exist. It is interesting to note that the pathological pattern shown above (e.g., a^+ b^+ c^+ c^- b^- a^-) corresponds to the expected references of nested subroutine calls (i.e., subroutine a calls subroutine b which calls subroutine c, etc., and each subroutine, of course, returns to its caller). This is also true of other stack-like program constructs. Such highly modular program design is quite typical and, furthermore, is often explicitly encouraged. In view of Hatfield's finding where the overall r exceeded 2.0 in many programs, it is reasonable to assume that there were probably regions in which r was quite small, possibly below 1.0, which were counterbalanced by regions with very high values of r. At present we do not have this particular information available, but if it were true, performance could be greatly improved by eliminating the high r value regions. This problem will be discussed in the next section.

5.5.5 Bounds for FIFO Removal Algorithm

Theorem 3 applies to LRU removal algorithms and many other removal algorithms, although these other cases will not be explicitly proven in this thesis. It is interesting to consider whether the result of Theorem 3 applies to the FIFO removal algorithm. Unfortunately, due to the peculiarities of FIFO, a simple generalizable cyclic page trace pattern has not been found. But, isolated examples have been found, as illustrated in Figure 12, that show that it is possible for r to exceed $|M^1|+1$. This result is stated in Theorem 4. Based upon other examples, it is conjectured that the r, when FIFO removal is used, may be as high as

(r10)

Parameters

As seen by S:

- $P = a,c,a,b,b,c,c,a,a,b,b,c,c,a,a,b,b,c,c$
- $|P| = 19$
- $Q = \{ a, b, c \}$
- $|Q| = 3$
- $|M^1| = 2$
- FIFO Removal

As seen by S':

- $P = a^+,c^-,a^-,b^+,b^-,c^+,c^-,a^+,a^-,b^+,b^-,c^+,c^-,a^+,a^-,b^+,b^-,c^+,c^-$
- $|P| = 19$
- $Q = \{ a^+, a^-, b^+, b^-, c^+, c^- \}$
- $|Q| = 6$
- $|M^1| = 4$
- FIFO Removal

Simulation

```
                                                 steady-state
          |<------ transient ------>|<------------- cycle -------------------------->|
Trace:    a+  c-  a-  b+  b-  c+  c-  a+  a-  b+  b-  c+  c-  a+  a-  b+  b-  c+  c-
S
Fetch:    *   *       *               *               *               *
M1:       a   c   c   b   b   b   b   a   a   a   a   c   c   c   c   b   b   b   b
              a   a   c   c   c   c   b   b   b   b   a   a   a   a   c   c   c   c
S'
Fetch:    *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *
M1:       a+  c-  a-  b+  b-  c+  c-  a+  a-  b+  b-  c+  c-  a+  a-  b+  b-  c+  c-
              a+  c-  a-  b+  b-  c+  c-  a+  a-  b+  b-  c+  c-  a+  a-  b+  b-  c+
                  a+  c-  a-  b+  b-  c+  c-  a+  a-  b+  b-  c+  c-  a+  a-  b+  b-
                      a+  c-  a-  b+  b-  c+  c-  a+  a-  b+  b-  c+  c-  a+  a-  b+
                                  ^                                               ^
                                  |-------------------- same ---------------------|
```

Results

- $F = 6$
- $F' = 19$
- $r = 19/6 = 3.16$

For the steady-state cycle:

- $F = 3$
- $F' = 12$
- $/r/ = 12/3 = 4.0$

Figure 12.
Cyclic Page Trace with FIFO Removal

$2|M^1|$.

(thf)

THEOREM 4:

For any two demand-fetch FIFO-removal two-level storage systems, S and S', with page sizes N and N'=N/2 and certain primary store sizes $|M^1|$ and $|M^1|'=2|M^1|$, respectively, there exists a cyclic page trace, P = Pt•(Pc)* where $|Pc| = 2(|M^1|+1)(|M^1|)$, such that the page fetch frequency ratio, r, exceeds $|M^1|+1$.

Proof:

By example (Figure 12).

CHAPTER 6.
SPATIAL VS. TEMPORAL LOCALITY MODEL OF PROGRAM BEHAVIOR

6.0 Introduction

Early in this thesis it was explained that a major rationale for multilevel storage systems is based upon the Principle of Locality. Unfortunately, locality is still a poorly understood, or at least controversial, phenomenom. In this chapter some novel viewpoints and insights will be presented.

6.1 Types of Program Reference Locality

Let us consider two extreme forms of program reference locality which will be called temporal locality and spatial locality:

6.1.1 Temporal Locality

If the logical addresses { a^1, a^2, ... } are referenced during the time interval t-T to t, there is a high probability that these same logical addresses will be referenced during the time interval t to t+T.

This behavior can be rationalized by program constructs such as: loops, frequently used variables, and frequently used subroutines.

6.1.2 Spatial Locality

If the logical address a is referenced at time t, there is a high probability that a logical address in the range a-A to a+A will be referenced at time t+1.
This behavior can be rationalized by program constructs such as: sequential instruction sequencing, and linear data structures (e.g., arrays).

6.1.3 General Locality

The definitions of temporal and spatial locality above are quite extreme. Usually we consider only the general spatiotemporal properties and define locality as:

Locality

If the logical addresses { a^1, a^2, ... } are referenced during the time interval t-T to t, there is a high probability that the logical addresses in the ranges a^1-A to a^1+A, a^2-A to a^2+A, ..., will be referenced during the time interval t to t+T.

It is important to recognize that temporal locality and spatial locality are indeed the underlying phenomenon and that the "general locality" is merely a simplifying merging and blurring of these basic concepts.

6.2 Conventional Removal Algorithms

We can begin to understand the factors causing the page size anomaly by studying how the various conventional removal algorithms handle temporal and spatial locality. In particular, we see, that whereas temporal locality policies are given explicit attention, spatial locality policies are usually handled implicitly and subtlely. The "least recently used", LRU, removal algorithm, for example, is very much concerned about the temporal aspects of the program's reference pattern. The spatial aspects are handled as a by-product of the fact that the demand fetch algorithm must load an entire page (i.e., a spatial region) at a time and LRU removal decisions are based upon these pages. With these thoughts in mind, we can see that decreasing page size causes the conventional storage management algorithms to increase their sensitivity to temporal locality and decrease their sensitivity to spatial locality. Increasing page size, of course, results in the reverse effect.

6.3 Locality in Actual Programs

Many of the techniques for improving the locality behavior of programs, such as the method of automatic program restructuring by sector (subroutine) reordering described by Hatfield and Gerald [47], result in both increased temporal and spatial locality. But, it seems that the reordering technique does, in fact, significantly favor spatial locality since it was noted [47] that:

> "the better orderings not only concentrate appropriate sectors into pages, but these pages also naturally cluster into larger units that satisfy nearness requirements on the page level - and cluster better than do the pages of the other orderings ... clustering sectors into pages also clusters pages into larger units."

6.4 Locality Mixes

An effective multilevel storage management system must take both temporal and spatial locality into consideration. As we have seen from both Hatfield's and Seligman's results, neglecting spatial locality can have disasterous results. Any given program, or portion of a program's operation, can have its reference locality characterized by the two-by-two matrix:

$$\begin{array}{cc|c|c|} & & \multicolumn{2}{c}{\text{TEMPORAL}} \\ & & \multicolumn{1}{c}{\text{Low}} & \multicolumn{1}{c}{\text{High}} \\ \cline{3-4} \text{SPATIAL} & \text{Low} & 1 & 2 \\ \cline{3-4} & \text{High} & 3 & 4 \\ \cline{3-4} \end{array}$$

Quadrant 1, low-temporal and low-spatial locality, is definitely undesirable for operation in a multilevel storage system. There have been numerous algorithms and programmer training techniques developed, as mentioned above, to minimize the number of programs with these poor locality characteristics. Quadrant 4, high-temporal and high-spatial locality, has traditionaly been the region of best performance and is usually the objective of good program design. Unfortunately, it is not always possible or convenient to design programs which attain both high temporal and high spatial locality; thus, we find many programs operating in quadrants 2 or 3.

6.5 Spatial Locality Algorithms

Storage management techniques are needed which provide far more flexibility and robustness for balancing the system's sensitivity to temporal and spatial locality. These algorithms must explicitly consider the spatial locality of a program. The tuple-coupling approach, described in the next chapter, is one such technique. It takes advantage of the temporal locality and compactness possible with small pages characterized by quadrant 2 behavior, yet it adjusts to the spatial locality and clustering characterized by quadrant 3 behavior by simulating the removal policies associated with large pages.

6.6 Comment on the Page Size Anomaly

With this insight, we can now see that the page size anomaly is not really even a function strictly of page size! Instead, it is an issue of locality, temporal versus spatial.

CHAPTER 7.
SPATIAL REMOVAL STORAGE MANAGEMENT ALGORITHMS

7.0 Introduction

As stated earlier in this thesis and noted by Hatfield, a removal algorithm that would limit the page fetch frequency ratio, r, to 2 would be very desirable. In this section a technique, called the "tuple-coupling approach", is described which, when used in conjunction with conventional removal algorithms, such as LRU or FIFO, guarantees that r will not exceed 2.

7.1 Tuple-Coupling Approach

The basic concept behind the tuple-coupling approach is extremely simple. First, the two portions, p^+ and p^-, of each original larger page, p, must be identifiable (i.e., the set of pages of S' are viewed as a collection of 2-tuples). Second, the removal ordering policies must be applied to both elements of a tuple (i.e., the tuples are coupled in regard to ordering decisions) such that a page p^+ or p^- of S' is never removed unless the corresponding page p of S would also have been removed from M'. The particular implementation of this approach may vary slightly depending upon the removal algorithm, e.g., LRU, FIFO, etc., that is to be used. Any removal algoritam to which the tuple-coupling approach can be incorporated is said to be "tuple-couple-able".

7.1.1 An Example of LRU Tuple-Coupling

Figure 13 illustrates the application of the tuple-coupling approach to the LRU removal example previously shown in Figure 9. It should be noted that, in this case, r has indeed been limited to 2 although it had a value of 2.2 when normal LRU removal was used. The reader should carefully compare Figures 7 and 11 to understand how the tuple-coupling approach affects the removal algorithm. The M^1 contents are identical, of course, for S in both examples, but there are subtle differences in M^1 contents for S'. Each state of M^1 contents is marked, 1 to 11, in Figure 13 for reference purposes. Notice that in this implementation of tuple-coupling whenever both halves of a page, p^+ and p^-, are in M^1, they are always adjacent in the M^1 ordering; compare this with Figure 9.

At page trace step 3 we can see the first difference between Figures 7 and 11. Page a^- is referenced and must be fetched in Figure 9, it is then placed at the top of the M^1

(f11)

Parameters

As seen by S:

- P = a, b, a, b, c, c, b, a, a, c, c
- |P| = 11
- Q = { a, b, c }
- |Q| = 3
- $|M^1|$ = 2
- LRU Removal

As seen by S':

- P = a^+, b^+, a^-, b^-, c^+, c^-, b^+, a^+, a^-, c^+, c^-
- |P| = 11
- Q = { a^+, a^-, b^+, b^-, c^+, c^- }
- |Q| = 6
- $|M^1|$ = 4
- LRU Removal with Tuple-Coupling

Simulation

	1	2	3	4	5	6	7	8	9	10	11
Page Trace:	a^+	b^+	a^-	b^-	c^+	c^-	b^+	a^+	a^-	c^+	c^-
S											
Fetch:	*	*			*			*		*	
M^1 Contents:	a	b	a	b	c	c	b	a	a	c	c
		a	b	a	b	b	c	b	b	a	a
S'											
Fetch:	*	*	*	*	*	*		*	*	*	*
M^1 Contents:	a^+	b^+	a^-	b^-	c^+	c^-	b^+	a^+	a^-	c^+	c^-
		a^+	a^+	b^+	b^-	c^+	b^-	b^+	a^+	a^-	c^+
			b^+	a^-	b^+	b^-	c^-	b^-	b^+	a^+	a^-
				a^+	a^-	b^+	c^+	c^-	b^-	b^+	a^+

Results

- F = 5
- F' = 10
- r = 10/5 = 2.0

Figure 13.
Example of LRU Removal with Tuple-Coupling
(see Figure 9 for comparison)

ordering which becomes a^-, b^+, a^+. On the other hand, in Figure 13 at step 3, it is noticed that a^+ was already in M^1. Thus, when a^- is placed at the top of the M^1 ordering, a^+ is coupled to it resulting in the ordering a^-, a^+, b^+. At page trace step 7 of Figure 13 we see another interesting example of the tuple-coupling approach. At the previous step the ordering was

$$c^- \quad c^+ \quad b^- \quad b^+$$

when the reference to b^+ is made, there is no need to initiate a fetch since b^+ is already in M^1. The M^1 ordering then becomes

$$b^+ \quad b^- \quad c^- \quad c^+$$

since LRU requires that the most recent reference move to the top. Under this tuple-coupling scheme, b^- is also moved toward the top of the ordering to continue to be adjacent to b^+.

7.1.2 Implementation of the Tuple-Coupling Approach

It is important to note that there are often various ways to implement tuple-coupling. In particular, in the LRU tuple-coupling algorithm described above, the 2-tuples, whenever both portions were in M^1, were arranged to be adjacent in the M^1 removal ordering. The requirement that neither portion, p^+ or p^-, of a tuple in S' be removed unless the corresponding page of S would have been removed can be accomplished in other ways. For example, the LRU removal stack can be left in its normal ordering, as in Figure 9. In this case, when it is necessary to remove a page from S' the bottom page is not necessarily the correct choice to satisfy tuple-coupling. There is an algorithm which can scan the LRU stack and select the correct page for removal (in fact, it will select, of course, the same page selected by the algorithm illustrated in Figure 13).

7.1.3 An Example of FIFO Tuple-Coupling

It is interesting to consider the effect of tuple-coupling upon FIFO removal. Figure 14 illustrates the application of the tuple-coupling approach to the FIFO removal example previously shown in Figure 6. Once again, the page fetch frequency ratio, r, which originally was 2.75 has indeed been limited to 2. The example of Figure 14 does not fully illustrate all the interesting aspects of tuple-coupling upon FIFO removal. In particular, if page p^+, for example, is referenced in a page trace and it was not already in M^1, it must be fetched. The M^1 contents are reordered as follows:

1. If p^- is not currently M^1, p^+ is placed at the top of the FIFO ordering.
2. If p^- is currently in M^1, p^+ is placed immediately

(E12)

Parameters

As seen by S:

- P = a, b, a, b, c, c, b, a, a, c, c
- $|P|$ = 11
- Q = { a, b, c }
- $|Q|$ = 3
- $|M^1|$ = 2
- FIFO Removal

As seen by S':

- P = a^+, b^+, a^-, b^-, c^+, c^-, b^+, a^+, a^-, c^+, c^-
- $|P|$ = 11
- Q = { a^+, a^-, b^+, b^-, c^+, c^- }
- $|Q|$ = 6
- $|M^1|$ = 4
- FIFO Removal with Tuple-Coupling

Simulation

	1	2	3	4	5	6	7	8	9	10	11
Page Trace:	a^+	b^+	a^-	b^-	c^+	c^-	b^+	a^+	a^-	c^+	c^-
S											
Fetch:	*	*			*			*			
M^1 Contents:	a	b	b	b	c	c	c	a	a	a	a
		a	a	a	b	b	b	c	c	c	c
S'											
Fetch:	*	*	*	*	*	*		*	*		
M^1 Contents:	a^+	b^+	a^-	b^-	c^+	c^-	c^-	a^+	a^-	a^-	a^-
		a^+	a^+	b^+	b^-	c^+	c^+	c^-	a^+	a^+	a^+
			b^+	a^-	b^+	b^-	b^-	c^+	c^-	c^-	c^-
				a^+	a^-	b^+	b^+	b^-	c^+	c^+	c^+

Results

- $F = 4$
- $F' = 8$
- $r = 8/4 = 2.0$

Figure 14.
Example of FIFO Removal with Tuple-Coupling
(see Figure 8 for comparison)

before p^- in the logical FIFO ordering

p^-'s relative ordering remains unchanged.

The reason for the second part of this rule can be seen from the normal FIFO ordering rule which places a page p at the top only if it were not already in M^1. If it were in M^1, it remains at its previous ordering position. Under tuple-coupling, this rule applies jointly to the (p^+,p^-) tuple as stated above. The reader is encouraged to work through the example of Figure 10 using the tuple-coupling approach to illustrate this FIFO ordering phenomenom. The effect of the tuple-coupling approach is summarized in Theorem 5.

(th5)

THEOREM 5:

For any two demand-fetch two-level storage systems, S and S', with page sizes N and N'=N/2, respectively, the use of the "tuple-coupling" approach for S' in conjunction with a removal algorithm that is "tuple-couple-able" is sufficient to guarantee that the page fetch frequency ratio, r, cannot exceed the value 2 for all possible page traces, P.

Proof:

(See below).

7.1.4 Proof of Theorem 5

As described earlier, when an adress trace, A, is applied to storage systems S (with page size N) and S' (with page size N'=N/2), it can be represented as page traces P and P', respectively. At time t^1, let us consider a specific address reference, a, whose corresponding page references are p (in S) and p^+ (in S'). In processing this reference there are four possible fetch actions in S and S' depending upon the current content state of primary store, M^1:

State	page p (S)	page p+ (S')	F	F'	F'-F	effect
1	in M^1	in M^1	0	0	0	r => 1
2	in M^1	not in M^1	0	1	1	r => >1
3	not in M^1	in M^1	1	0	-1	r => <1
4	not in M^1	not in M^1	1	1	0	r => 1

Recall that the page fetch frequency ratio, r, equals F'/F. In states 1 and 4 the same action (i.e., no page fetch in 1 and a page fetch in 4) occurs in both S and S', the occurrence of these states cause r to tend towards 1. In state 3, a page fetch is required in S but not in S', this situation, if frequent, will cause r to decrease toward zero. This is usually the intended result of reducing page size. Only state 2, in which S' alone requires a page fetch, contributes to an increase in r. Thus, we will concentrate our analysis on this particular situation.

Since state 2 requires that page p be in M^1 at time t^1, if we scan the address trace backwards, there must be some previous reference time t^2 that caused page p (in S) to be fetched into M^1 (this may have been the only previous reference to p or the page p may have been fetched and removed many times). At time t^2, there must also be a corresponding reference to either p^- and p^+ of S'. These two cases will be considered separately:

Case 1:
$$\underline{p} = \ldots p \ \ldots p$$
$$\underline{p}' = \ldots p^- \ldots p^+$$
$$t = \ldots t^2 \ldots t^1$$

This case merely illustrates the fact that it can require two page fetches (for p^+ and p^-) in S' to fetch the same amount of storage as page p in S. If this were the only case for state 2, r would never exceed 2.

Case 2:
$$\underline{p} = \ldots p \ \ldots p$$
$$\underline{p}' = \ldots p^+ \ldots p^+$$
$$t = \ldots t^2 \ldots t^1$$

In this case we see that subsequent to reference t^2 page p of S and page p^+ on S' must be in M^1. Yet at time t^1 page p of S is still in M^1 but page p^+ of S' is not. Under these circumstances r can certainly exceed 2, merely making p^- the next reference will account for 3 fetches in S' compared to 1 fetch in S. Furthermore, it is possible that the references between t^2 and t^1 could be repeated to continually cause fetches for p^+ in S' without any corresponding fetches required in S. Thus, we see that this is precisely the situation that allows r to exceed 2.

Under closer analysis, we see that this situation requires that in S' p^+ be removed from M' between t^2 and t^1 whereas in S p remains in M^1. In other words, this general situation can only occur if at some time t, p^+ or p^- of S' is selected for removal from M^1 and the corresponding page p of S is not also removed from M^1. But, the tuple-coupling algorithm (see page 125) is "such that a page p^+ or p^- of S' is never removed unless the corresponding page p of S would also have been removed from M^1". Thus, the tuple-coupling eliminates the possibility of case 2 and therefore guarantees that r cannot exceed 2.

Q.E.D.

7.2 Effectivness of Tuple-Coupling

Clearly, the tuple-coupling approach has an influence upon the overall effectiveness of the basic removal algorithm being used and the benefits of the smaller page size. It is obvious that there are certain reference patterns (with r less than 2) for which tuple-coupling increases the value of r. On the other hand, it can be shown, as a simple exercise for the reader, that the example of Figure 6 retains its low page fetch frequency ratio of 0.5 even when tuple-coupling is used. In fact, tuple-coupling may often result in the "best of both worlds" by placing a bound on the page fetch frequency ratio, r, for high r regions without interfering with the performance of originally low r regions.

A program's reference behavior in S', during a short interval of its operation, may be characterized by three regions based upon the value of the page fetch frequency ratio, r, when tuple-coupling is not used:

1. Sparse reference - small r (e.g., less than 1).
2. Moderate reference - moderate r (e.g., between 1 and 2).
3. Dense reference - high r (e.g., greater than 2).

In the sparse reference region, it is unlikely that both portions, p^+ and p^-, of a page, p, will be in M^1 simultaneously; thus, the tuple-coupling will have minimal effect upon performance. In the dense reference region, we have already seen that tuple-coupling prevents extreme values of r. Based upon some recent, though limited, measurements, it appears that in the moderate reference region tuple-coupling performs about as well as the non-tuple-coupled algorithms.

CHAPTER 8.

DISCUSSION AND CONCLUSIONS

8.0 Introduction

Efficient and effective storage management is important to the development of future computer systems. It has been estimated that the storage subsystems account for over 70% of the cost of most contemporary installations and, based upon present trends, this percentage is expected to increase.

Much more research will be needed before all the problems of automatic storage management are understood and the obstacles to effective operation eliminated. This thesis has solved several open problems and has provided insight that should lead to the solution of many more problems.

8.1 Summary

A detailed discussion of the many facets of storage management is presented in Chapter 2. It also contains a general discussion of the requirements which a system must satisfy to be effective for the user.

In Chapters 3 and 4 a model for storage hierarchy systems is formalized and an implementation is proposed. The system's design is based upon an orderly and uniform treatment of the storage levels. Specific techniques to improve performance, such as continuous hierarchy, shadow storage, direct transfer, read through, store behind, and automatic management, are explained.

In Chapter 5 the "page size ancmaly" is presented (see also Hatfield [48]):

> "The assumption about virtual memory systems that as overhead (time for access and software page management) decreases page size should be reduced is not always a good one. Recent experiments indicate that larger sizes can provide better performance for programs that make highly localized use of memory space."

This phenomenom is formalized and a bound on the performance is proven.

In Chapters 6 and 7 the concept of spatial locality is introduced and serves as the basis for a new storage removal algorithm called "tuple-coupling". These concepts are used to explain the occurrence of the "page size anomaly" in actual systems. It is proven that the tuple-coupling approach is a sufficient strategy to avoid the occurrence of the "page size anomaly" and it offers potential performance improvements for the storage hierarchy system.

The techniques and theorems presented in this thesis provide a much more scientifically sound basis for examining and designing storage hierarchy systems than most current ad hoc approaches. Although there is still a long way to go, development of these formalisms is essential to the advancing of the "science" in Computer Science.

8.2 Further Work

There are many areas touched on by this work in which questions remain. One of the most significant is in the development and study of other possible "spatial locality" removal algorithms in addition to the tuple-coupling approach studied in this thesis. This is an entirely wide open area.

Although tuple-coupling is studied extensively in this thesis, there are still many unanswered questions. How does tuple-coupling compare with the class of "stack" algorithms studied by Mattson et al [63], in particular under what circumstances, if any, is tuple-coupling a stack algorithm? Likewise, how does tuple-coupling compare with the theoretically optimal replacement algorithm, called OPT [63] or MIN [12]? On a more practical side, how efficiently can a tuple-coupling algorithm, or other spatial removal algorithms, be implemented?

In order to ascertain specific proof of the utility and efficiency of general storage hierarchies, it will be necessary to actually construct and measure the performance of such a system or, at least, perform more extensive simulation analysis. Furthermore, we must develop overall programming techniques and execution environments that are even more amenable to efficient operation in a storage hierarchy system.

Many of these questions are currently under investigation, the results will be published later in a MIT Project MAC Technical Report.

REFERENCES AND BIBLIOGRAPHY

Abbreviations used in the references:

CACM	Communications of the ACM
FJCC	Fall Joint Computer Conference
IEEE-TC	IEEE Transactions on Computers
IEEE-TEC	IEEE Transactions on Electronic Computers
JACM	Journal of the ACM
SJCC	Spring Joint Computer Conference

[1] Ahearn, G. R., Y. Dishon, and R. N. Snively, "Design Innovations of the IBM 3830 and 2835 Storage Control Units", IBM Journal of Research and Development 16, 1 (January 1972), 11-18.

An interesting article illustrating the use of microprocessors to produce sophisticated and flexible mass storage control units.

[2] Aho, Alfred V., Peter J. Denning, and Jeffrey Ullman, "Principles of Optimal Page Replacement", JACM 18, 1 (January 1971), 80-93.

Presents a model of program behavior based upon 1-order non-stationary Markov processes where p(x,u,t) is the probability that a reference to page x is generated at time t given that P is currently in state u and |U|=l+1. They are only able to carry through the analysis for "almost" stationary 0-order Markov models. They do note that although "we are able to give only approximate extensions to the general 0-order case ... we believe that the simplest program model is a good starting point for the formal investigation of paging algorithm behavior." Unfortunately, they do not provide any justification or empirical evidence to even substantiate this choice of a model.

[3] Amdahl, G. M., and L. D. Amdahl, "Fourth-Generation Hardware", Datamation, (January 1967).

[4] Anacker, W. and C. P. Wong, "Performance Evaluation of Computer Systems With Memory Hierarchies", IEEE-TEC EC-16, 6 (December 1967), 765-773.

[5] Arora, S. R. and A. Gallo, "Optimal Sizing, Loading and Re-loading in a Multi-level Memory Hierarchy System", SJCC 38, (1971), 337-344.

[6] Austin, B. J., "A Dynamic Disc Allocation Algorithm Designed to Reduce Fragmentation During File Reloading", The British Computer Journal 14, 4 (1971), 378-381.

Presents an interesting file allocation technique that minimizes the need for "page maps" (file maps) by periodically dumping and reloading all files, in general daily. The statistics on file usage are quite relevant. Of the 5000 files on the system, almost 50% of the files were 1 page or less in length (1 page = 512 36-bit words = 2K bytes). On the other hand, these 1 page files consumed only 10% of the space used. In fact, file allocation was rather evenly distributed amongst files from 1 to 1000 pages in length (i.e., files of page sizes = 1, 2-3, 4-7, 8-15, ..., 511-1023, consumed about 10% in each range).

[7] Ayling, J. K., "Monolithic Main Memory is Taking Off", 1971 IEEE International Convention Digest, (march 1971), 70-71.

[8] Bard, Y., "Performance Criteria and Measurement for a Time-Sharing System", IBM Systems Journal 10, 3 (1971), 193-216.

[9] Batson, Alan, Shy-ming Ju, and David C. Wood,

"Measurements of Segment Size", CACM 13, 3 (March 1970), 155-159.

[10] Belady, L. A., R. A. Nelson, and G. S. Shedler, "An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine", CACM 12, 6 (June 1969), 349-353.

[11] Belady, L. A. and C. J. Kuehner, "Dynamic Space-Sharing in Computer Systems", CACM 12, 5 (May 1969), 282-288.

[12] Belady, L. A., "A Study of Replacement Algorithms for a Virtual Storage Computer", IBM Systems Journal 5, 2 (1966), 78-101.

[13] Bell, Gorden C. and David Casasent, "Implementation of a Buffer Memory in Minicomputers", Computer Design, (November 1971), 83-89.

[14] Bensoussan, A., C. T. Clingen, and R. C. Daley, "The Multics Virtual Memory", Proceedings of the ACM Second Symposium on Operating System Principles, Princeton University, (October 20-22, 1969), 30-42.

[15] Best, Donald T., "The Present and Future of Moving Media Memories", 1971 IEEE International Convention Digest, (March 1971), 270-271.

[16] Bobeck, Andrew H. and H. E. D. Scovil, "Magnetic Bubbles", Scientific American 224, 6 (June 1971), 78-90.

[17] Camras, Marvin, "Information Storage Density", IEEE Spectrum, (July 1965), 98-105.

[18] Cashman, M. W., "Technology: 1971" (editorial note), Datamation, (January 1972), 47.

Brief review of significant technical developments of 1971. Items mentioned include the Texas Instruments 960A minicomputer priced at $2850 ($1350 for processor plus $1500 for 4K memory) and the Intel MCS-4 "cpu on a chip" priced at $66 (in quantities of 100 - 999).

[19] Coffman, E. G., and L. C. Varian, "Further Experimental Data on the Behavior of Programs in a Paging Environment", CACM 11, 5 (July 1968), 471-474.

[20] Considine, James P. and Allan H. Weis, "Establishment and Maintenance of a Storage Hierarchy for an On-line Data Base Under TSS/360", FJCC 35, (1969), 433-440.

[21] Conti, C. J., D. H. Gibson, and S. H. Pitkowsky, "Structural Aspects of the System/360 Model 85: I. General Organization", IBM Systems Journal 7, 1 (1968), 2-14.

[22] Conti, C. J., "Concepts for Buffer Storage", IEEE Computer Group News, (March 1969), 6-13.

[23] Cook, Robert W. and Michael J. Flynn, "System Design of a Dynamic Microprocessor", IEEE-TC C-19, 3 (March 1970), 213-222.

[24] Dell, Harold R., "Design of a High Density Optical Mass Memory System", Computer Design, (August 1971), 49-53.

[25] Denning, Peter J., "The Working Set Model for Program Behavior", CACM 11, 5 (May 1968), 323-333.

[26] Denning, Peter J., "Virtual Memory", Computing Surveys 2, 3 (September 1970), 153-189.

[27] Denning, Peter J., "Third Generation Computer Systems", Computing Surveys 3, 4 (December 1971), 175-216.

[28] Denning, Peter J., "Thrashing: Its Causes and Prevention", FJCC 33, (1968), 915-922.

Discusses thrashing and points out interdependency between processor scheduling and memory management.

Of particular interest, he discusses the affect of page traverse time T on the efficiency (busyness) of the processor. He notes that "Reducing T by a factor of 10 could reduce the memory requirement by as much as 10, the number of busy processors being held constant ... or increase by 10 the number of busy processors, the amount of memory being held constant." He states that the 360/67 at Carnegie-Mellon University reported by Fikes et al [36] confirms these projections. [Unfortunately, the situations are not really analagous, in fact, for the simple case implied by Denning, Fikes comments that strictly replacing the drum by a smaller but faster LCS (large core storage) "yielded only a modest improvement". Major changes to the system were required to improve performance, these changes may even have improved the drum version; thus, a scientific comparison can not be established]. At the end, Denning briefly speculates on reducing T by using a three-level memory system and possibly using small page sizes (since access delay is assumed to be minimal for the 2nd level store). He does not pursue this point very far in this paper.

[29] Denning, Peter J., "Resource Allocation in Multiprocess Computer Systems", MIT Project MAC Report MAC-TR-50, Massachusetts Institute of Technology, Cambridge, Mass., (May 1968).

[30] Dickinson, R. V. and W. K. Orr, "System Ten - A New Approach to Multiprogramming", FJCC 37, (1970), 181-186.

[31] Durae, Melvin J., Jr., "Finding Happiness in Extended Core", Datamation, (August 15, 1971), 32-34.

[32] Farr, William W. and William E. Peisel, "An Optimum Disc Organization for a Virtual Memory System", Computer Design, (June 1971), 49-54.

[33] Femling, Don, "Rubber-band Memory", Electronic Design 13, (June 24, 1971), 64-68.

[34] Fetch, G. C., "Memory Organization and Hierarchies of

Storage Workshop", Computer Group News, (January 1969), 24-25.

[35] Fields, Stephen, "Silicon Disk Memories Beat Drums", Electronics, (May 24, 1971), 85-86.

[36] Fikes, Richard E., Hugh C. Lauer, and Albin L. Vareha, Jr., "Steps Toward a General-Purpose Time-Sharing System Using Large Capacity Core Storage and TSS/360", Proceedings of the 23rd ACM National Conference, (1968), 7-18.

[37] Finch, Tudor R., "Semiconductor Memory", 1971 IEEE International Convention Digest, (March 1971), 272-273.

[38] Fotheringham, John, "Dynamic Storage Allocation in the Atlas Computer Including an Automatic Use of a Backing Store", CACM 4, 10 (October 1961), 435-436.

An early paper that briefly describes the Atlas system. Though short, it presents the basic ideas rather clearly. See the paper by Kilburn et al [57] for a more extensive presentation.

[39] Gardner, W. David, "Debate Gives Peek at IBM's Direction", Datamation 18, 1 (January 1972), 58-60.

[40] Gentile, Richard B. and Joseph R. Lucas, Jr., "The TABLON Mass Storage Network", SJCC 38, (1971), 345-356.

[41] Gentile, Richard B. and Robert W. Grove, "Mass Storage Utility: Considerations for Shared Storage Applications", IEEE Transactions on Magnetics MAG-7, 4 (December 1971), 848-852.

[42] Gertz, Jeffrey Lee, "Hierarchical Associative Memories for Parallel Computation", MIT Project MAC Report MAC-TR-69, Massachusetts Institute of Technology, Cambridge, Mass., (June 1970).

[43] Goldberg, Robert P., "Virtual Machine Systems", MIT Lincoln Laboratory Technical Memorandum Number 28L-0036, (August 1969).

[44] Greenes, R. A., A. N. Pappalardo, C. W. Marble, G. O. Barnett, "A System for Clinical Data Management", _FJCC_ _35_, (1969), 297-305.

[45] Guertin, R. L., "Programming in a Paging Environment", _Datamation_ _18_, 2 (February 1972), 48-55.

Discusses programming techniques "which reduce the working set of a program or reduce the probability of requiring page swaps". Primarily discusses techniques related to array processing in FORTRAN.

[46] Hatch, Theodore F., Jr., and James B. Geyer, "Hardware/Software Interaction on the Honeywell Model 8200", _FJCC_ _33_, (1968), 891-901.

Describes the H8200 which incorporates hardware controlled multiprogramming of up 8 job processes plus an executive process. The hardware multiprogramming is accomplished by interleaved instruction execution of the (up to 9) active processes; there is a separate set of processor registers for each of the 9 processes. It is claimed that the "horizontal multiprogramming", besides eliminating conventional multiprogramming software overhead, also provides greater I/O throughput. Reference [30] should be examined for an implementation of hardware "vertical multiprogramming".

[47] Hatfield, D. J. and J. Gerald, "Program Restructuring for Virtual Memory", _IBM_ _Systems_ _Journal_ _10_, 3 (1971), 168-192.

[48] Hatfield, D. J., "Some Experiments on the Relationship Between Page Size and Program Access Pattern", _IBM_ _Journal_ _of_ _Research_ _and_ _Development_ _16_, 1 (January 1972), 58-66.

Presents the results of many experiments with page size and access patterns. Provides much of the empirical evidence behind the "page size" anomaly (i.e., decreasing page size by half can result in a drastically increased paging I/O rate - sometimes more than double). These results were based upon instruction traces of real IBM System/360 programs.

[49] Hobbs, L. C., "Present and Future State-of-the-Art in Computer Memories", IEEE-TEC EC-15, 4 (August 1966), 534-550.

[50] Howard, Harry, "Memories: Modern Day 'Musical Chairs'", EDN/EEE, (August 15, 1971), 23-31.

[51] IBM, "IBM System/360 Time Sharing System, System Logic Summary, Program Logic Manual, Form Number GY28-2009-2, (June 1970), 17-22.

[52] IBM, "A Guide to the IBM System/370 Model 165", Form Number GC20-1730, (June 1970), 19-25.

[53] Jensen, J., P. Mondrup, and P. Naur, "A Storage Allocation Scheme for Algol 60", CACM 4, 10 (October 1961), 441-445.

Presents a design whereby the compiler, with its run-time support, handle multilevel management of the backing store with the aid of "hints" from the programmer. This is an example of one of the many semi-automatic techniques for storage management that have been used.

[54] Johnson, R. R., "Needed: A Measure for Measure", Datamation, (December 15, 1970), 22-30.

[55] Katzan, Harry, Jr., "Operating Systems Architecture", SJCC 36, (1970), 109-118.

[56] Katzan, Harry Jr., "Storage Hierarchy Systems", SJCC 38, (1971), 325-336.

[57] Kilburn, T., D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, "One-level Storage Systems", IEEE-TEC EC-11, 2 (April 1962), 223-235.

[58] Lehman, Meir M. and Jack L. Rosenfeld, "Performance of a Simulated Multiprogramming System", FJCC 33, (1968), 1431-1442.

[59] Lew, Art, "On Optimal Pagination of Programs", University of Hawaii Information Sciences Report, Honolulu, Hawaii, (May 1970).

[60] Lewis, P. A. W. and P. C. Yue, "Statistical Analysis of Program Reference Patterns in a Paging Environment", Proceedings of the 1971 IEEE International Computer Society Conference, (September 1971), 133-134.

[61] Liptay, J. S., "Structural Aspects of the System/360 Model 85: II. The Cache", IBM Systems Journal 7, 1 (1968), 15-21.

[62] Martinson, J. R., "Utilization of Virtual Memory in Time Sharing System/360", IBM TR53.0001, IBM Systems Development Division, Yorktown Heights, N. Y., (October 28, 1968).

[63] Mattson, R. L., J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies", IBM Systems Journal 9, 2 (1970), 78-117.

[64] Mattson, Richard L., "Evaluation of Multilevel Memories", Memorandum, IBM Research Laboratory, San Jose, California, 1971.

Presents address trace analysis techniques for two-level memory hierarchies (which use "stack" replacement algorithms, e.g., LRU) that are up to 1000 times faster than conventional simulation (see also reference [63]). He considers multiple classes (i.e., set-associative constraint), various primary memory sizes and page sizes, and other factors, such as cost and speed of technologies. A sample analysis is illustrated. Although it is a powerful technique, Mattson's analysis described in this paper and reference [63] is not immediately applicable to multiple level memories (i.e., three-levels or more) if the page size varies and is subsetted between the levels or if an "non-stack" replacement algorithm is used, such as "tuple-coupling".

[65] Meade, Robert M., "On Memory System Design", FJCC 37, (1970), 33-43.

This is a very extensive survey of storage hierarchy systems. Meade presents various diagrams indicating trade-offs between block size, buffer store size, transfer rates, processor utilization, etc. His data also illustrates the "page size anomaly" although he doesn't explicitly comment upon it. He investigated three level storage hierarchies based upon extending cache systems and stated that: "By analysis like that above, the block size for a third-level should be from one to eight second-level blocks. Preliminary results indicate that a 4:1 ratio (256 bytes at the third level) is best."

[66] Meade, Robert M., "How a Cache Memory Enhances a Computer's Performance", Electronics, (January 17, 1972), 58-63.

[67] Meyer, R. A. and L. H. Seawright, "A Virtual Machine Time-Sharing System", IBM Systems Journal 9, 3 (1970), 199-218.

[68] Morenoff, Edward and John B. McLean, "Application of Level Changing to a Multilevel Storage Organization", CACM 10, 3 (March 1967), 149-154.

[69] Myers, Edith, "He Dreams the Impossible Dream ... or Does He?", Datamation, (July 1, 1971), 52-53.

[70] O'Neill, Robert W. and Burnett H. Sams, "Preplanned vs. Dynamic Storage Allocation Techniques", CACM 4, 10 (October 1961), 416-418.

An early discussion that compares and contrasts the techniques of preplanned and dynamic storage allocation. Preplanned techniques are based upon information either provided by the programmer or the compiler. Dynamic techniques assume that the storage allocation is handled primarily at run-time by the operating system. More recent debates on this subject can be found in Denning [26] and Sayre [77].

[71] Penny, Samuel J., Robert Fink, and Margaret Alston-Garnjost, "Design of a Very Large Storage System", _FJCC 37_, (1970), 45-51.

[72] Ramamoorthy, C. V. and K. M. Chandy, "Optimization of Memory Hierarchies in Multiprogrammed Systems", _JACM 17_, 3 (July 1970), 426-445.

[73] Randell, B. and C. J. Kuehner, "Dynamic Storage Allocation Systems", _CACM 11_, 5 (May 1968), 297-306.

[74] Rector, Robert W. and C. J. Walter, "The Fourth - Another Generation Gap?", _Modern Data_, (March 1969), 42-48.

Presents the problems of designing the next generation of machines. Briefly discusses the importance of a memory hierarchy and speculates on the technologies that will be available.

[75] Rice, Rex and William R. Smith, "SYMBOL - A major Departure From Classic Software Dominated von Neumann Computing Systems", _SJCC 38_, (1971), 575-587.

[76] Rosin, Robert F., "Contemporary Concepts of Microprogramming and Emulation", _Computing Surveys 1_, 4 (December 1969), 197-212.

[77] Sayre, D., "Is Automatic 'Folding' of Programs Efficient Enough to Displace Manual?", _CACM 12_, 12 (December 1969), 656-660.

[78] Seligman, Lawrence, "Experimental Data for the Working Set Model", MIT Project MAC Computation Structures Group Memo Number 39, Massachusetts Institute of Technology, Cambridge, Mass., (March 1968).

[79] Shahbender, R., "Magnetic Memories - Present Status and Future Trends", 1971 IEEE International Convention Digest, (March 1971), 274-275.

[80] Shooman, Martin L., "Notes on Computer Hardware, Software, and Systems Reliability", MIT IAP Seminar: Computer Architecture, Department of Electrical Engineering, Cambridge, Mass., (January 7, 1972).

These are notes on various aspects of reliability that were used as part of an MIT seminar on computer architecture. In particular, there is a section on software reliability and the "nature of bugs". A few storage system related problems are mentioned.

[81] Smith, John L., "Multiprogramming Under a Page on Demand Strategy", CACM 10, 10 (October 1967), 636-646.

[82] Smith, William R. et al, "SYMBOL - A Large Experimental System Exploring Major Hardware Replacement of Software", SJCC 38, (1971), 601-616.

[83] Solomon, Martin B., Jr., "Economics of Scale and the IBM System/360", CACM 9, 6 (June 1966), 435-440.

[84] Sumner, F. H., "Operand Accessing in the MU5 Computer", Proceedings of the 1971 International Computer Society Conference, (September 1971), 119-120.

[85] Thompson, Steve, Jack A. Morton, and Andrew Bobeck, "Memories: Future Storage Techniques", The Electronic Engineer, (August 1971), 33-39.

[86] Varian, L. C. and E. G. Coffman, "An Empirical Study of the Behavior of Programs in a Paging Environment",

[87] Walter, Cloy J., Arline Bohl Walter, and Marilyn Jean Bohl, "Impact of Fourth Generation Software on Hardware Design", IEEE Computer Group News, (July 1968), 1-10.

[38] Wilkes, M. V., "Slave Memories and Dynamic Storage Allocation", IEEE-TEC 14, 2 (April 1965), 270-271.

[39] Williams, John G., "Large-Core Storage in Perspective", Computer Design 11, 1 (January 1972), 45-49.

[90] Woolf, Ashby Morefield, "Analysis and Optimization of Multiprohrammed Computer Systems Using Storage Hierarchies", University of Michigan, Ann Arbor, Michigan, Systems Engineering Laboratory, SEL Technical Report Number 53, (April 1971).

BIOGRAPHICAL NOTE

Stuart Elliot Madnick was born in Worcester, Massachusetts, on July 10, 1944. He attended public schools there, graduating from Classical High School in May, 1962. He entered the Massachusetts Institute of Technology in September, 1962, where he studied Electrical Engineering, receiving the degree of S.B. (June, 1966). In December, 1964, he married the former Elthel J. Westerman of Malden, Mass. They have two children, Howard Jon and Michael Andrew.

In September, 1967, he entered the M.I.T. Alfred P. Sloan School of Management, where he studied Management Sciences and Computer Science in conjunction with the M.I.T. Department of Electrical Engineering, receiving the degrees of S.M., Management, and S.M., Electrical Engineering (June, 1969).

Mr. Madnick joined the staff of the M.I.T. Electrical Engineering department in September, 1966, as a teaching assistant; in July, 1971 he became an Instructor. He has taught several computer science courses in addition to M.I.T.'s principal systems programming course (6.251). In 1969 he was a recipient of the Carlton E. Tucker Award for Excellence in Teaching.

At M.I.T., Mr. Madnick has been engaged in various computer-related projects for the Student Aid Center, Civil Engineering Department, Mechanical Engineering Department, and the Computation Center. In June, 1968, he became associated with M.I.T. Project MAC, where his research in operating system design, computer architecture, programming languages, and software engineering formed the basis of his doctoral dissertation.

Mr. Madnick has been a consultant to the IBM Corporation since 1966. He assisted in the development of the IBM 360/40 and 360/67 CP/CMS Virtual Machine Time-Sharing projects. During the summer of 1967, he was an Associate Engineer at the Lockheed Palo Alto Research Laboratory, where he designed and implemented the control modules for the Lockheed/NASA DIALOG information retrieval system.

Mr. Madnick has consulted to the IBM Cambridge Scientific Center, the Lockheed Palo Alto Research Laboratory, the Honeywell Programming Systems Division and Honeywell Advanced Systems and Technology Organization, Martin-Marietta Co., Naval Underwater Systems Center, and INTERCOMP, Inc. on the design of multi-terminal multi-access computing systems.

Mr. Madnick is a member of Sigma Xi, the Institute of Electrical and Electronics Engineers (IEEE), and the Association for Computing Machinery (ACM). He has been a reviewer for the ACM Computing Reviews since 1969.

Publications

Madnick, S.E., "String Processing Techniques", Communications of the ACM, Vol. 10, No. 7, July 1967.

Madnick, S.E., and Moulton, G.A., "SCRIPT, An Online Manuscript Processing System", IEEE Transactions on Engineering Writing and Speech, Vol. EWS-11, No. 2, August 1968.

Madnick, S.E., "Multi-Processor Software Lockout", Proceedings of the 1968 ACM National Conference, August 1968.

Madnick, S.E., "Time-Sharing Systems: Virtual Machines Concept vs. Conventional Approach", Modern Data Systems, Vol. 2, No. 3, March 1969.

Madnick, S.E., and Alsop, J.W., "A Modular Approach to File System Design", Proceedings of the 1969 Spring Joint Computer Conference, Vol. 34, May 1969.

Madnick, S.E., "MIS - Problems Plus A Solution", Computer Forum Report, Vol. 1, No. 4, July 1969.

Madnick, S.E., "Design Strategies for File Systems: A Working Model", File Organization - Selected Papers From FILE-68, IFIP Administrative Data Processing Group (IAG), Publication No. 3, 1969.

Madnick, S.E., and Alsop, J.W., "A Modular Approach to File System Design", IAG Quarterly, IFIP Amsterdam, Vol. 2, No. 3, 1969.

Madnick, S.E., "What is Microprogramming?", Proceedings of the IEEE Computer Conference, September 1971.

CS-TR Scanning Project
Document Control Form

Date : 2 / 15 / 96

Report # LCS-TR-107

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:

Document Information

Number of pages: 156(162-images)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:

Check each if included with document:

☒ DOD Form (2) ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other:

Page Data:

Blank Pages(by page number):

Photographs/Tonal Material (by page number):

Other (note description/page number):

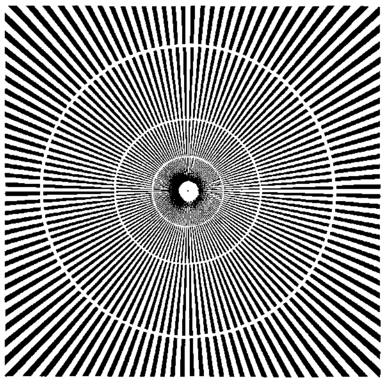
Description : Page Number:
IMAGE MAP: (1-156) UNH'ED TITLE PAGE, 2-155,
UNH'ED BLANK PAGE.
(157-162) SCANCONTROL, DOD (2), TRGT'S (3)

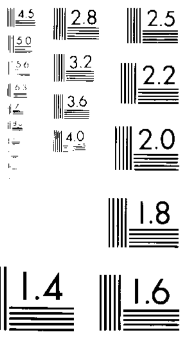
Scanning Agent Signoff:
Date Received: 2 / 15 / 96 Date Scanned: 3 / 11 / 96 Date Returned: 3 / 11 / 96

Scanning Agent Signature: [illegible]

Rev 9/94 DS/LCS Document Control Form cstrform.vsd

5 10



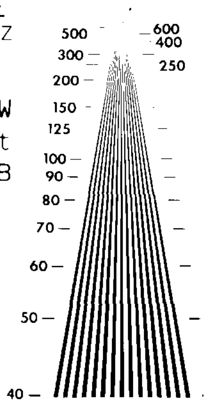


NMA MICROFONT QJKLPYZ
6BSI2GH5D4X7U3W8V9E
PQR45DE9UV670FG8STHIJNOWXABYZ
3KLMI2C [illegible]

ABCDEFGHIJKLMNOPQRSTUVW
XYZabcdefghijklmnopqrst
uvwxyz0123456789 OCR-B

ABCDEFGHIJKLMNOPQRSTUV
WXYZabcdefghijklmnopqr
stuvwxyz1234567890PICA

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
1234567890 Elite



ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
1234567890 Spartan Medium 6 pt

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
1234567890 Spartan Medium 4 pt

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
1234567890 Spartan Medium 8 pt

65 120

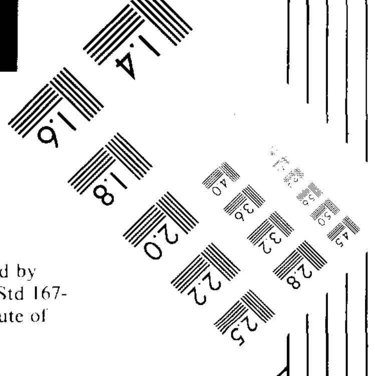
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
1234567890 Spartan Medium 10 pt

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
1234567890 Spartan Medium 12 pt

IEEE Std 167A-1987

FACSIMILE TEST CHART

Prepared by the IEEE Facsimile Subcommittee and printed by Eastman Kodak Company. Use in accordance with IEEE Std 167-1966, Test Procedure for Facsimile. Copyright 1987, Institute of Electrical and Electronics Engineers.



123456 0123456 0123456 6543210

A4 Page 6543210

AIIM SCANNER TEST CHART #2

Spectra

4 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

6 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

8 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

10 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

Times Roman

4 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

6 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

8 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

10 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

Century Schoolbook Bold

4 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

6 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

8 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

10 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

News Gothic Bold Reversed

4 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

6 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

8 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

10 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

Bodoni Italic

4 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

6 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

8 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

10 PT ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz;:",./?$0123456789

Greek and Math Symbols

4 PT ΑΒΓΔΕΞΘΗΙΚΑΜΝΟΠΦΡΣΤΥΩΧΨΖαβγδεξθηικλμνοπφρστυωχψζ≧∓",./≦±=≠°><⊁⊀⋟⋞≡

6 PT ΑΒΓΔΕΞΘΗΙΚΑΜΝΟΠΦΡΣΤΥΩΧΨΖαβγδεξθηικλμνοπφρστυωχψζ≧∓",./≦±=≠°><⊁⊀⋟⋞≡

8 PT ΑΒΓΔΕΞΘΗΙΚΑΜΝΟΠΦΡΣΤΥΩΧΨΖαβγδεξθηικλμνοπφρστυωχψζ≧∓",./≦±=≠°><⊁⊀⋟⋞≡

10 PT ΑΒΓΔΕΞΘΗΙΚΑΜΝΟΠΦΡΣΤΥΩΧΨΖαβγδεξθηικλμνοπφρστυωχψζ≧∓",./≦±=≠°><⊁⊀⋟⋞≡

White

Black

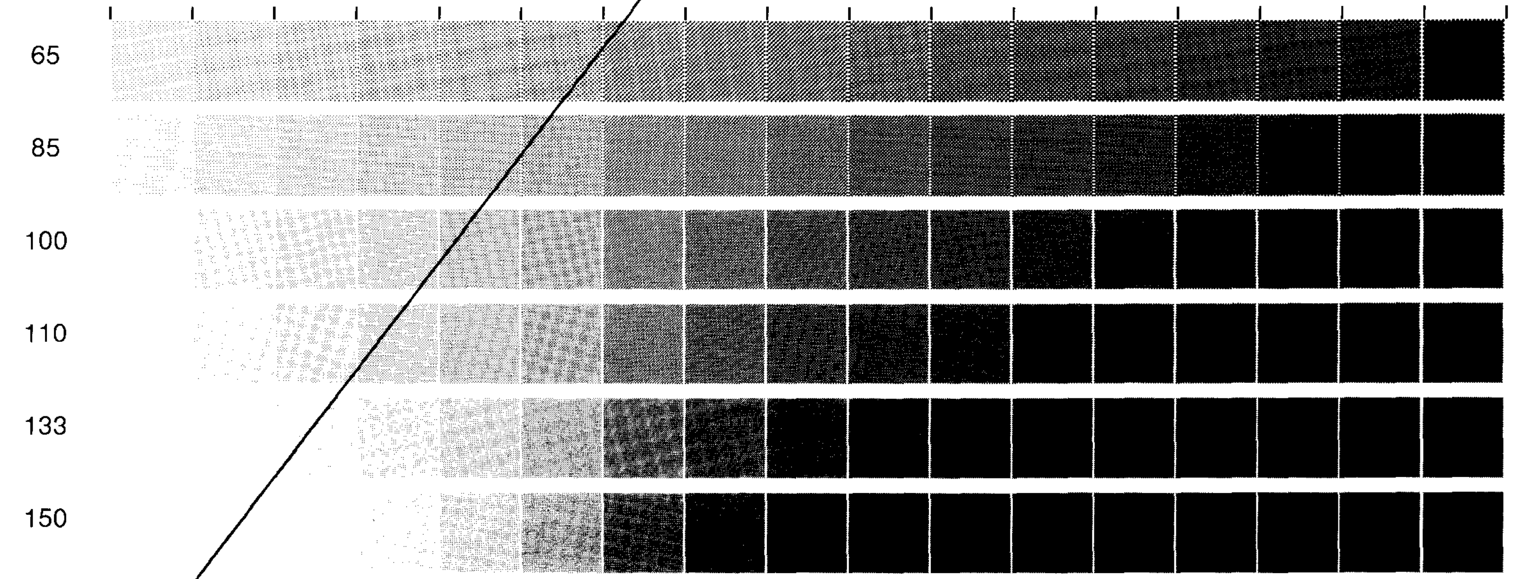
Isolated Characters

e	m	1	2	3	a
4	5	6	7	o	°
8	9	0	h	I	B

123456 6543210

A4 Page 6543210

MESH HALFTONE WEDGES



A4 Page 6543210

3456 6543210 6543210 6543210