

**INTEGRATED RESOURCE MANAGEMENT ALGORITHMS
FOR COMPUTER SYSTEMS**

John Busch

**1984
Report No. CSD-840035**

MASTER COPY

111

112

113

114

115

116

UNIVERSITY OF CALIFORNIA

Los Angeles

Integrated Resource Management Algorithms
For Computer Systems

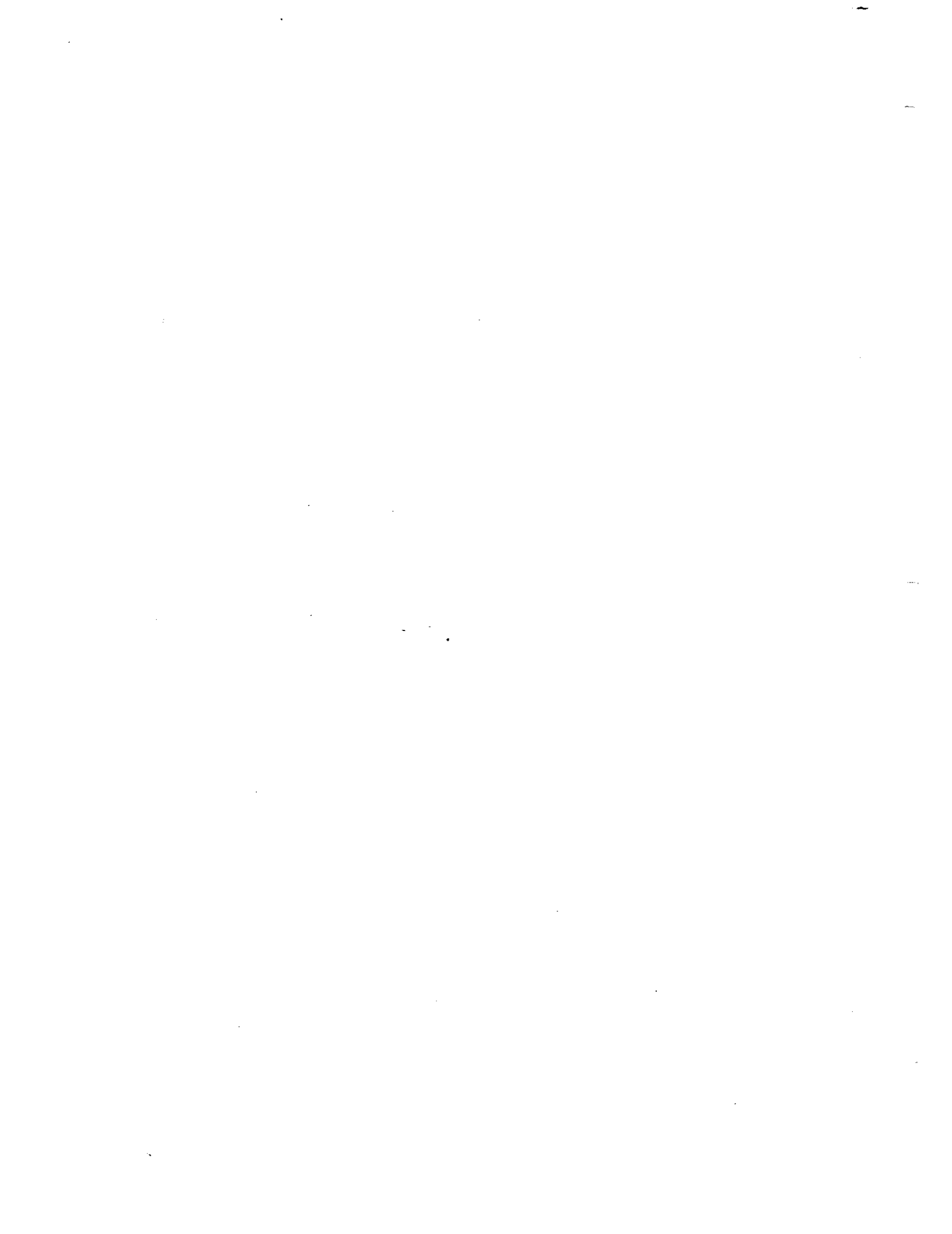
A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

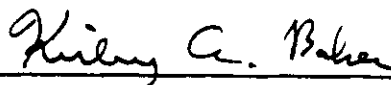
John Richard Busch

1984

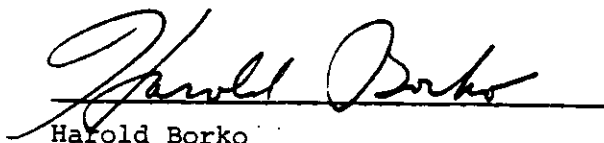
© Copyright by
John Richard Busch
1984



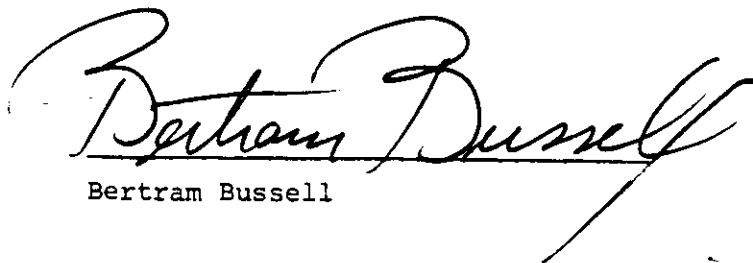
The dissertation of John Richard Busch is approved.



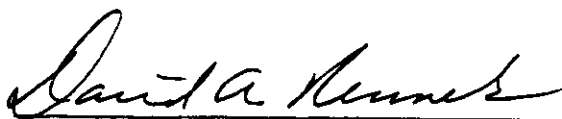
Kirby A. Baker



Harold Borko



Bertram Bussell



David A. Rennels



Wesley W. Chu, Committee Chair

University of California, Los Angeles

1984

To my wife and father

Contents

1. Introduction.....	1
1.1 Need For Integrated Strategies.....	1
1.2 Research Objectives.....	2
1.3 Research Approach.....	3
1.4 Research Contribution.....	6
1.4.1 Bread-Board Philosophy and Design.....	6
1.4.2 Workload Characterization of Case Study Family.....	7
1.4.3 Empirical Analysis of Popular Algorithms.....	7
1.4.4 New Algorithms.....	7
1.4.5 Principles and Structure of Integrated Algorithms.....	8
1.4.6 Applicability to Decentralized Systems.....	9
1.5 Layout of the Dissertation.....	9
2. The Case Study Computer Family.....	12
2.1 Architecture.....	12
2.2 Configurations.....	13
2.3 Application Mix.....	16
3. The Bread-Board.....	19
3.1 The Bread-Board Kernel.....	20
3.1.1 Kernel Structure and Implementation.....	20
3.2 Performance Measurement and Analysis Tools.....	23
3.2.1 Statistics Support and Display.....	25
3.2.2 Workload Generators.....	27
3.2.3 Disc Workload Characterizer.....	28
3.2.4 Disc Cache Simulator.....	29
3.2.5 Analytic System Model.....	32
4. Workload Characteristics.....	35
4.1 Sessions and Jobs.....	38
4.2 Memory Usage.....	38
4.2.1 Memory Usage By Objects.....	38
4.2.2 Object Size Distributions.....	39
4.2.3 User/System Object Partition.....	43
4.2.4 Memory Reference Locality.....	45
4.2.5 Object Sharing.....	46
4.3 Processor Usage.....	48
4.4 Semaphore Contention.....	50
4.5 Disc Usage.....	53
4.6 Summary of Salient Workload Characteristics.....	61
5. Basic Algorithms.....	64
5.1 Secondary Store Access and Space Management.....	65
5.1.1 Previous Research.....	66
5.1.2 Bread-Board Results.....	68
5.2 Priority Assignment.....	74
5.3 Semaphore Management.....	78
5.4 Processor Management.....	79
5.5 Main Memory Management.....	83
5.5.1 Placement and Garbage Collection.....	84
5.5.2 Main Memory Replacement and Load Control.....	92
5.6 Conclusions.....	114

6. Integrating Data Management.....	117
6.1 Introduction.....	117
6.2 Overview of Memory Hierarchies.....	122
6.3 Need For Main Memory/Disc Balancing.....	125
6.4 Alternative Balancing Approaches.....	131
6.4.1 External Caching Techniques.....	133
6.4.2 Using Large Main Memories.....	141
6.4.3 Improved Concurrency Control.....	153
6.4.4 Requirements of Transaction Recovery.....	155
6.5 Evaluation of Alternatives.....	158
6.6 The Bread-Board Disc Cache.....	168
6.6.1 Read Handling.....	171
6.6.2 Locating Cached Disc Regions.....	173
6.6.3 Write Handling Mechanisms and Policies.....	175
6.6.4 Cache Fetch, Placement and Replacement.....	180
6.6.5 External Caching Controls.....	183
6.6.6 Performance of the Bread-Board Caching.....	184
6.7 Conclusions.....	188
7. Further Research.....	191
7.1 Integrated Resource Scheduling in Decentralized Systems.....	192
7.2 Integrated Data Management in Decentralized Systems.....	194
7.3 Conclusions.....	208
8. Conclusions.....	209
Appendix A Measurement Subsystem Specification.....	215
Appendix B Instrumentation Formatting.....	217
Appendix C Disc Workload Characterizer.....	221
Appendix D Disc Cache Simulator.....	227
Appendix E Analytic System Model.....	235
Appendix F Kernel Tuning Commands.....	246
Bibliography.....	251

List of Figures

Figure 1: HP 3000 Family of Computers.....	14
Figure 2: HP 3000 Configurations.....	15
Figure 3: Breakdown of HP 3000 Installed Base.....	16
Figure 4: Language Usage With the HP 3000 Family.....	17
Figure 5: Bread-Board Performance Tools.....	24
Figure 6: Disc Cache Simulator Verification.....	31
Figure 7: Active Object Occupancy in Memory.....	39
Figure 8: Object Size Distributions.....	42
Figure 9: User / System Partition of Active Objects.....	44
Figure 10: Working Set Composition.....	45
Figure 11: Memory Reference Behavior.....	46
Figure 12: Effects of Sharing on Replacement Policies.....	48
Figure 13: Processor Burst Distribution.....	49
Figure 14: System Semaphore Usage Profile.....	52
Figure 15: Inter-Extent Reference Locality.....	55
Figure 16: Access Method Use of Secondary Store.....	57
Figure 17: Spatial Locality of Reference of Access Methods.....	58
Figure 18: Temporal Locality of Reference of Access Methods.....	59
Figure 19: Disc Utilization With Seriality and Clustering.....	70
Figure 20: Processor Pause Interval Distribution.....	80
Figure 21: Processor Allocation Flow Chart.....	82
Figure 22: Impact of Allocation Unit Size.....	87
Figure 23: Global Garbage Collection.....	89
Figure 24: Garbage Collection Impact on Hole Distribution.....	91
Figure 25: CLOCK Algorithm in Segmented System.....	99
Figure 26: Effects Of Working Set Window Size.....	101
Figure 27: PFF vs WS With Moderate Load.....	104
Figure 28: Comparison of CLOCK, WS, PFF in Action.....	107
Figure 29: Bread-Board CPU, Memory and Disc Management.....	110
Figure 30: Performance Impact Due to Integrated Algorithms.....	112
Figure 31: Standard Computer System Storage Hierarchy.....	123
Figure 32: Current Cost vs Capacity for Discs.....	126
Figure 33: Conventional Hierarchy Management Performance.....	130
Figure 34: Access Method Hit Rate vs Cache Size.....	133
Figure 35: External Caching : Buffer Per Disc.....	134
Figure 36: External Caching : Intermediate Level.....	135
Figure 37: External Caching Without Write Wait.....	137
Figure 38: External Caching With Write Wait.....	138
Figure 39: External Caching Sensitivity to RH and WW Prob.....	140
Figure 40: Local File/Application Caching.....	141
Figure 41: Global File/Application Caching.....	142
Figure 42: Explicit Global Disc Caching in Main Memory.....	143
Figure 43: Internal Caching Through File Mapping.....	145
Figure 44: Comparison of Explicit Internal/File Mapping.....	147
Figure 45: File Mapping Without Write Wait.....	149
Figure 46: File Mapping With Write Wait.....	150
Figure 47: File Mapping vs Read Hit/Write Wait Prob.....	152
Figure 48: Nowait Logging via Kernel Serial Posting.....	157
Figure 49: Serial Write Queue Management.....	158

Figure 50: Alternatives in High Utilization Range.....	161
Figure 51: Alternatives Across Broad Processor Range.....	164
Figure 52: Bread-Board Internal Disc Cache Interface.....	170
Figure 53: Disc Read Caching in the Bread-Board.....	172
Figure 54: Cached Disc Domain Location Mechanism.....	174
Figure 55: Disc Write Caching in the Bread-Board.....	176
Figure 56: Internal Caching Write Protocol.....	179
Figure 57: Dynamic Partitioning of Cache Across Discs.....	182
Figure 58: Kernel Caching Performance Impact.....	185
Figure 59: Effects of Multiple Discs and Disc Caching.....	186
Figure 60: Current Research Configuration.....	198
Figure 61: Software Architectural Components.....	199
Figure 62: Shared/Local Virtual Space Partitioning.....	200
Figure 63: Bread-Board Partition of Functionality.....	203

Acknowledgments

First I would like to thank my advisor, Wesley Chu, for his ongoing encouragement and support throughout this research effort. I would also like to thank my other committee members.

Next I would like to thank the many colleagues of various universities and small and large companies for their valuable discussions on many aspects of this research.

I would like to thank my parents and wife for their ongoing encouragement and confidence throughout the years of building, analyzing and documenting this research effort.

Finally I wish to thank the Hewlett-Packard Corporation for its backing of this research with computer resources and financial support.



Vita

July 16, 1952 - Born, Los Angeles, California

1973 - B.A. Math(cum laude), California State University, Northridge

1977 - M.A. Math., University of California, Los Angeles

1978-1984 - Computer Science Researcher, University of California, Los Angeles and the Hewlett-Packard Corporation

Publications

John R. Busch, *The MPE IV Kernel : History, Structure and Strategies*, Proceedings of the HP 3000 International User's Group Conference, Orlando, April 27, 1982, F-9; reprinted in Journal of the HP 3000 International User's Group Conference, Inc, Vol 5, No 3,4, July 1982

John R. Busch, *Disc Caching in the System Processing Units of the HP 3000 Family of Computers*, Proceedings of the HP 3000 International User's Group, Edinburgh, October 1983; reprinted in Journal of the HP International Users Group, Vol 7, No 1, January 1984, pp 21-24

John R. Busch and Wesley W. Chu, *Secondary Store Caching Alternatives - Analysis and Measurement*, in preparation

John R. Busch and Wesley W. Chu, *Integrated Computer Resource Management Algorithms For Computer Systems*, in preparation



ABSTRACT OF THE DISSERTATION

Integrated Resource Management Algorithms

For Computer Systems

by

John Richard Busch

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1984

Professor Wesley W. Chu, Chair

This research focused on the integration of computer resource management algorithms in order to realize significant system level price/performance improvements from current trends in component technologies. The research proceeded by building a bread-board kernel and performance measurement and analysis tools for a current computer family, characterizing the workload and system behavior, implementing popular algorithms and observing their interactions, developing and analyzing improved functional partitions and algorithms, examining applicability of the principles and algorithms to decentralized system configurations, and establishing the follow-on research into algorithm integration for decentralized systems.

Shifting in functional responsibilities and algorithm integration were found to be needed from the kernel through the data management subsystems. Integration was found to impact the structure and goals of

resource management, requiring ongoing interaction and unselfish local management decisions.

Major results in algorithm integration include : a simple method of expressing performance objectives through external tuning and to have them reflected in local resource management decisions; disc access scheduling based on a prioritization which integrates the fetching and posting requirements of transient and permanent objects based on current system urgency; an integration of file/database buffering, posting, and locality control with the memory management of transient objects to provide high performance secondary store caching in the primary memory of transactional systems; a hybrid clock/working set replacement algorithm which achieves the advantage of each of the two known algorithms, reflects process urgency, and adapts well across a broad range of configurations and workloads; and a background garbage collection algorithm for segmented memory systems which operates globally and is distributed in time.

Modelling results are presented on the alternatives for secondary store caching as a function of processor speed, disc count and speed, and read hit and write wait probabilities.

Measurements of the bread-board system with the integrated algorithms demonstrate the behavior and performance potential, resulting in an order of magnitude improvement in system performance relative to the pre-integrated system over a broad mix of workloads and configurations.

Applications of the algorithms and principles to other computer families and to problems in decentralized systems are examined.

Chapter 1

Introduction

1.1 Need For Integrated Strategies

In the research and building of computer systems, components are frequently analyzed, optimized and constructed in isolation. Requirements, functionality, and interfaces are specified for the components. Local merit of worth indexes are defined. Algorithms and implementations are evaluated and developed to optimize these indexes.

Components fit together into a system. System objectives are not necessarily optimized by collecting together locally optimized components. Indeed, the interactions between components frequently determine to a large extent the success of a system to achieve its global objectives.

An area of computer research and design in which this principle of global versus local optimization is difficult to apply in practice involves computer system resource management with the objective of optimizing system performance. The problems are big, and making analysis and implementation tractable requires partitioning. Moreover, the technologies of the components are evolving independently and

there's a delay until the new improvements are understood and exploited.

The partitioning of functionality between applications, databases, file systems and kernels has redundancies that not only cause duplicate development and maintenance effort, but also limit the performance potential of the system in both centralized and decentralized configurations.

This research effort takes a fresh look at computer resource management by observing the interactions of algorithms and system components in order to discover some principles and behavior of algorithm integration, some new improved algorithms, and improved functional partitioning between system components.

1.2 Research Objectives

The objectives of this research effort include the following :

- a. construction of a bread-board suitable for analysis of the behavior and performance of alternative algorithms for main memory, processor, disc and semaphore management;
- b. workload characterization;
- c. implementation and empirical analysis of the behavior and performance of popular algorithms in the bread-board environment;

- d. introduction and analysis of new algorithms which cooperate to optimize system performance objectives given current technology trends;
- e. observations on the behavioral and structural impact resulting from algorithm integration;
- f. selective modelling of new results to evaluate their range of applicability;
- g. examination of extensions of the principles and algorithms in decentralized systems.

1.3 Research Approach

In order to discover principles and behavior of integrated algorithms, it was decided to observe algorithms in action in realistic situations. For this purpose, a bread-board system for a general purpose computer family was constructed consisting of a custom operating system kernel and extensive performance measurement and analysis tools. The workload was characterized. Popular algorithms were implemented in the bread-board, and their behavior, interactions and performance analyzed. Improved integrated algorithms were developed. The behavioral and structural impact due to algorithm integration for the centralized system was observed. The integrated algorithms for caching secondary store were simulated and modelled to evaluate their range of applicability. Extensions to problems in

decentralized systems were considered, and the follow-on research into integrated distributed transaction management was established.

The Hewlett Packard 3000 family was used for the case study due to extensive availability of computer resources for the research. A new operating system kernel, a measurement subsystem, and a set of performance measurement and analysis tools were designed and implemented to establish the research base.

The kernel was structured and implemented so as to support easy incorporation of new algorithms. Extensive measurement support needed for workload and algorithm characterization was designed and implemented in the kernel. Measurement display tools were developed to sample, reduce, and display the measured statistics. A trace driven simulation for detailed analysis of disc caching strategies and a system analytic model incorporating disc caching for broad ranges of system configurations and workload characteristics were designed and built. Benchmarks and real user sites were selected for workload generation in algorithm comparison and workload characterization experiments.

Selected known and iteratively improved algorithms for main memory, disc, processor and synchronizing resource management were implemented and analyzed across a range of workloads and system configurations. External controls to express system performance objectives and mechanisms to reflect these objectives in local resource management were examined.

The behavioral and structural impact of algorithm integration in the case study were observed.

The simulation model was used to examine locality characteristics of secondary store referencing and the benefit of various fetch and flush policies for secondary store caching.

The system model was used to estimate the performance potential of alternatives for secondary store caching in light of evolving component technologies. The impact of configurations with a range of disc and processor speeds and varying read hit and write wait probabilities was investigated.

The applicability of the concepts and algorithms to decentralized systems of current interest was considered. Applications and extensions of the results to systems with shared storage subsystems, file servers, discless workstations and concurrent distributed transaction support were examined.

Based on the research experience, we set our follow-on research directions for exploring integrated system architectures and algorithms to provide efficient distributed transaction processing.

1.4 Research Contribution

1.4.1 Bread-Board Philosophy and Design

The research base was designed for change and measurability. Performance was achieved through a careful functional split and parallel, cooperating algorithms. Local optimizations were selectively performed only after global optimization had been achieved.

The case study proved to be successful in obtaining a system which fully exploits the capabilities of the components to maximize the system objectives and to allow technological advances to be easily incorporated. The incremental development cost and performance for designing for measurability and change were minimal.

Production systems must support a broad range of functions. They take a long time to build, and a longer time to get reliable and performing well. New systems can't be "rolled from scratch" to take advantage of technology improvements. Exploiting advances in system components, and structure, and integrating them to optimize system objectives, requires evolution and iteration.

Building and optimizing systems using the design principles and structure employed in this research could help satisfy these needs.

1.4.2 Workload Characterization of Case Study Family

The more we know about real systems and workloads the better.

1.4.3 Empirical Analysis of Popular Algorithms

Many algorithms in the research literature have never been implemented or have only been blindly implemented in real environments. Empirically analyzing the algorithms in action disclosed factors which models and intuition failed to capture. This research disclosed some interesting results on the behavior and interactions of replacement algorithms (working set, page fault frequency, and clock), disc head optimization scheduling, alternative placement algorithms (best fit, first fit), semaphore management, and file buffering.

1.4.4 New Algorithms

Observations of the characteristics and weaknesses of known algorithms in action with current and evolving technology suggested new, improved algorithms. The new algorithms forwarded in this research include : a simple method of expressing performance objectives through external tuning and to have them reflected in local resource management decisions; a modified clock main memory replacement algorithm which reflects process urgency and adapts well across a broad range of configurations and workloads; a background garbage collection algorithm for segmented memory systems which operates globally and is distributed in time; a disc access prioritization algorithm which integrates posting requirements of transient data, write-ahead logging,

and files data with the fetching of code, transient data, and file data based on current system urgency; and an integration of file and database posting and locality control with the memory management of transient objects to provide integrated, high performance, cost effective secondary store caching in transactional systems.

In addition to the empirical results, simulation and analytic modelling of the alternatives for secondary store caching was performed. These modelling results indicate the importance of secondary store caching to exploiting technology trends in processors and memories, and identify the significant performance advantages of integrated policies for caching of discs in large primary memories.

1.4.5 Principles and Structure of Integrated Algorithms

Integration was found to impact the structure and goals of algorithms. It was found that integrated algorithms must be structured for ongoing interaction, and must make unselfish local management decisions. In order to exploit evolving technology, the partition of functionality needs to change, and integrated management approaches applied.

1.4.6 Applicability to Decentralized Systems

The external tuning controls, priority assignments and adjustments, and tagging of actions with these priorities as hints for local resource management decisions appear to extend well to decentralized resource management problems with shared secondary storage subsystems and file servers. Our follow-on research into architectures, functionality splits, and integrated algorithms to extend our centralized system results to efficiently support concurrent transactions in decentralized systems may contribute to the performance of distributed systems.

1.5 Layout of the Dissertation

During the discussions in each major area, an attempt is made to provide a summary of research in the area along with literature references.

Chapter 2 summarizes the main features of the case study computer family. The family's architecture, configurations, and application mix are summarized.

Chapter 3 describes the objectives, functions and designs of the bread-board kernel, measurement support, and simulation and analytic models which comprise the research base.

Chapter 4 presents a summary of the workload characteristics of the case study family. The temporal and spatial reference locality of executing processes for code and data objects and for databases and

file systems for disc domains are presented. Processor, semaphore, disc and main memory requirements are summarized. Relationships between workload characteristics and resource management are highlighted.

Chapter 5 presents the investigation results into algorithm integration for disc, processor, main memory and semaphore management. Findings on the performance and behavior of some popular algorithms as well as improved, integrated algorithms are reported. Principles and interactions in algorithm integration which were observed during the research are described.

Chapter 6 extends the algorithms to integrate data management in orders to exploit processor and memory technology trends. The need for secondary store caching is identified, and alternatives are evaluated in light of current and evolving technology based on cost, performance, and reliability. The impact of processor speed, disc count and speed, read hit and write wait probabilities, and effective multiprogramming level on the alternatives is modelled. The design and measurements of our bread-board implementation of one alternative, explicit global disc caching in the primary memory, are presented.

Chapter 7 discusses extensions of the concepts and algorithms to decentralized systems. The focus is on examining the potential of the concepts for improving performance of shared file access in some distributed configurations of current interest. Shared storage subsystems, file servers, discless workstations, and homogeneous distributed transaction systems are considered. Our direction for



continuing research into functional partitioning and algorithm integration for such systems is set.

Chapter 8 discusses conclusions which can be drawn from this research.

The appendices describe in more detail the measurement subsystem, kernel tuning commands, instrumentation, disc cache simulator, and analytic system model built for this research.

Chapter 2

The Case Study Computer Family

In order to empirically investigate algorithms in action in realistic environments, computers are required. The Hewlett-Packard 3000 computer family was selected since extensive computer resources for this family were available for the research. This chapter summarizes the main features of this computer family. The family's architecture, configurations, and application mix are summarized. Following chapters describe the bread-board, workload characteristics, and algorithmic measurements and analysis.

2.1 Architecture

The HP 3000 architecture is that of a stack oriented, segmented machine. Stack computers are discussed in general in [Bul 77]. The stack implementation for the HP 3000 family is discussed in detail in [Bla 77]. Segmented architectures are discussed in [Ran 69, Bat 70, Han 73, Hab 76].

The HP 3000 architecture is object oriented to the extent that an executing process can only address a specific set of objects and not the entire address space. A process can address its stack, up to 255 code segments, and up to 1024 data segments. Stacks and data segments are limited in size to 64 kbytes, while code segments are limited to 32 kbytes in length.

There are base and bounds registers for the current stack, code segment, and an extra data segment. The segment base registers contain physical memory addresses. Address computations are performed relative to these registers without translation. As a consequence, an entire code or data segment must be present in main memory for a reference to it to go through (i.e. no paging beneath the segmentation). A reference bit is associated with each code and data segment. This bit is set by microcode during instructions which reference the segment.

The instruction set provides move instructions between two data segments, and transfer of control within and between code segments through the procedure call instruction. The procedure call instruction saves the return state in a stack marker, looks up the physical address of the called procedure in a segment description table, and sets the current instruction register to this value.

2.2 Configurations

The processing units comprising the HP 3000 family, along with the family's growth history and directions, are shown in Figure 1 [HP 83a]. The target directions have been towards higher performance at the same cost on the high end of the family, and lower cost at the same performance on the low end. The current processors range from .25 to 1 IBM MIPS in performance.

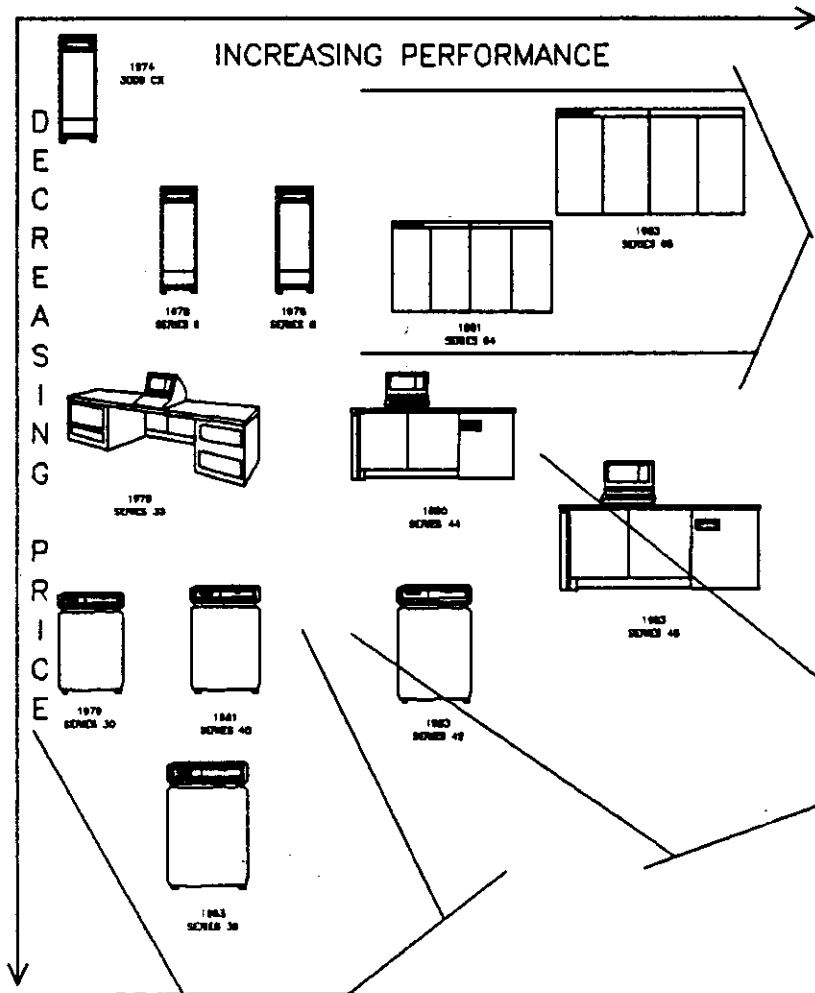


Figure 1: HP 3000 Family of Computers

There are approximately fifteen thousand HP 3000 installations. The installations have configurations ranging from 1 to 256 terminals, 1 to 8 tape drives, 1 to 8 line printers, 0 to 8 network communication lines, .25 to 8 Mbytes of main memory, and 1 to 32 disc drives. Figure 2 depicts the supported configuration range [HP 83a].

HP3000 Family Profile

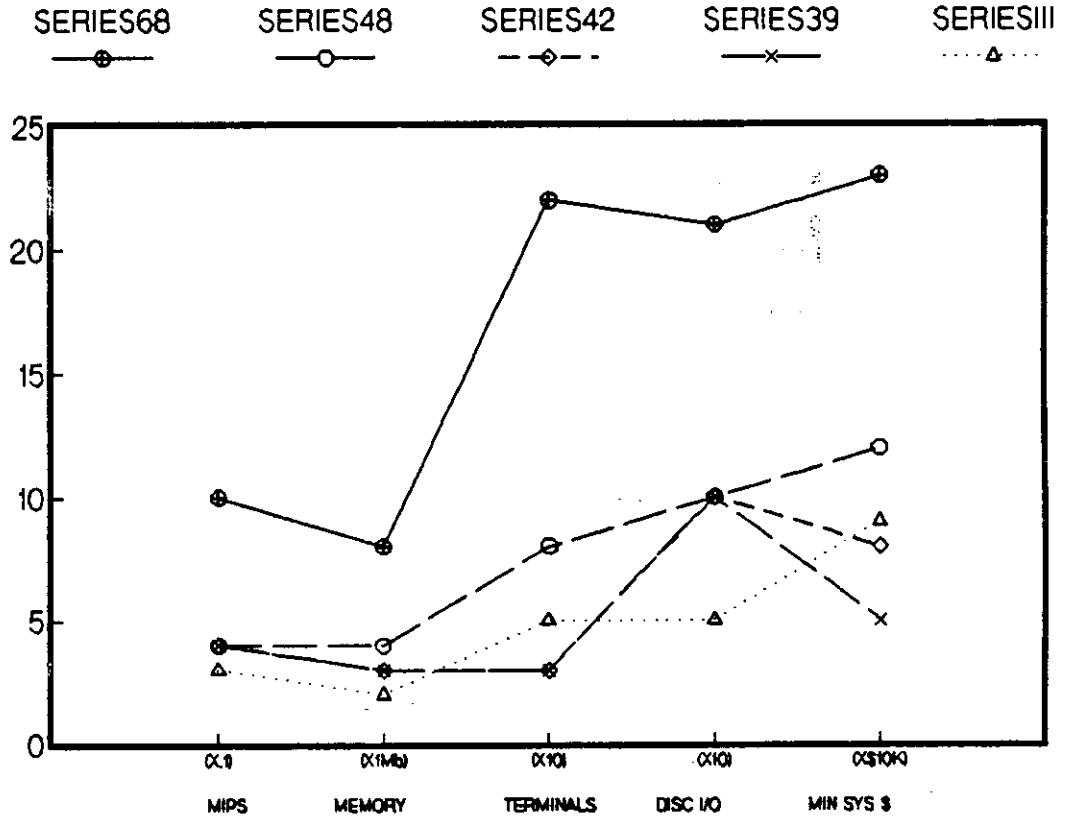


Figure 2: HP 3000 Configurations

2.3 Application Mix

Figure 3 provides a breakdown of the HP 3000 installed base in terms of business classification [Int 84]. HP 3000 computers are used primarily for office and manufacturing applications.

HP 3000 INSTALLATIONS BY BUSINESS CLASS

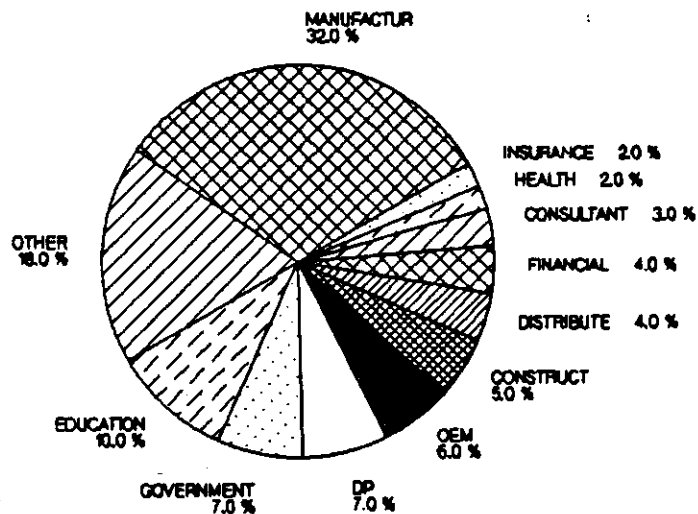


Figure 3: Breakdown of HP 3000 Installed Base

Figure 4 provides a breakdown of programming language usage in by the installed base. COBOL dominates, with FORTRAN, BASIC, and interpretive transaction processing languages following.

LANGUAGE USAGE IN HP 3000 INSTALLATIONS

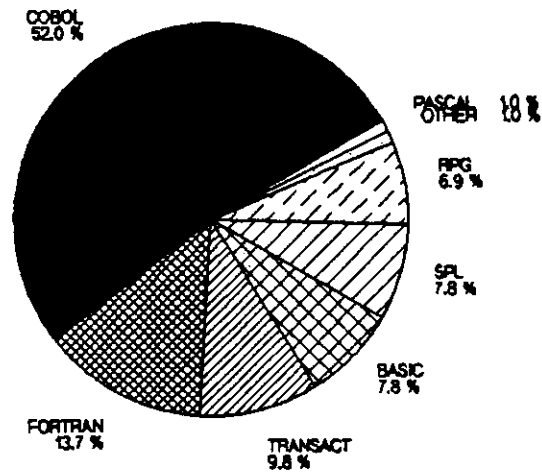


Figure 4: Language Usage With the HP 3000 Family

Extensive application software packages are available for the family. They are provided by the manufacturer, by software houses, and by installation programming staffs. Packages are available which provide database systems and utilities, manufacturing application packages, and office products including text editing, graphics and electronic mail.

Many installations are fairly dedicated, with one or two distinct application packages being used by most of the sessions (e.g. database or text editing). Many sites are dominated by batch processing during off-hours, and allow only a few batch jobs at a time to compete in the background during the normal work hours. There are however a

significant number of sites which are more diverse, such as program development sites.

Chapter 3

The Bread-Board

In order to use the HP 3000 family of computers for a case study in analyzing integrated algorithms, extensive operating system modification and a set of performance measurement and analysis tools was required. The existing operating system for the HP 3000 family would not accommodate the easy inclusion of new algorithms nor did there exist tools for local and global behavior and performance measurement and analysis. Thus, a new operating system kernel, a measurement subsystem, and a set of performance measurement and analysis tools were designed and implemented to form an adequate research base.

The kernel was structured and implemented so as to support easy incorporation of new algorithms. Extensive measurement support needed for workload and algorithm characterization was designed and implemented in the kernel. Measurement display tools were developed to sample, reduce, and display the measured statistics. A trace driven simulation for detailed analysis of disc caching strategies and a system analytic model incorporating disc caching for broad ranges of system configurations and workload characteristics were designed and built. Benchmarks and real user sites were selected for workload generation in algorithm comparison and workload characterization experiments.

This chapter describes the objectives, functions and designs of the kernel, the measurement support, and the simulation and analytic models providing the bread-board for algorithm analysis.

3.1 The Bread-Board Kernel

The kernel of an operating system [Cof 73, Han 73, Shaw 74, Hab 76] is the level of software, and sometimes hardware or firmware [Dur 79, Mye 78, Kah 79, Har 83], which provides the high level portions of the operating system with a virtual interface to the hardware and primitive system resources. The details of system configuration, of hardware interfacing, and of controlling access to system resources are all hidden by the kernel.

The kernel multiplexes the system resources among the competing processes through the application of service policies. The overall performance of a general purpose computer system is to a large extent determined by the effectiveness of these policies.

A new kernel was required which would support the research objectives. The major objective of the kernel design was to support multiple algorithms with minimal development impact. Ideally, the algorithms could be mixed and matched, and new algorithms could be plugged in easily.

3.1.1 Kernel Structure and Implementation

The concept of modularity in system design is discussed by Parnas in [Par 72]. Parnas recommends that the decomposition into modules be based on hiding the design decisions, not on steps in processing. A module should be a responsibility, not a subprogram. He points out the benefits of the modularization : reduced development time, product flexibility, and comprehensibility. To save procedure call overhead due to the decomposition, Parnas suggests that smart tools could be built to restructure the source or object for efficiency.

Kahn [Kah 81] discusses the extensible operating system design of iMAX for the Intel 432. The goal of iMAX was to provide the basic mechanisms in a manner which permitted the flexible implementation of policy by higher levels of the system. The structure of iMAX is that of a library set from which application specific operating systems can be constructed. The basic mechanisms are provided by iMAX, but the policies are flexibly implemented by higher levels of the system.

A hybrid approach to modularization was selected for the design and implementation of the bread-board kernel.

In order to make it easy to replace the resource managers for main memory, processor, disc, and semaphores, these functions were structured as independent modules. Access and control of their resources is limited to that provided through their explicit interfaces.

The design objective for the construction of a resource manager itself was leverage so that alternative resource managers could be

easily built. Thus, services within a resource manager were modularized more on a primitive function basis.

Throughout the design and implementation, efficiency at the instruction level was of secondary concern.

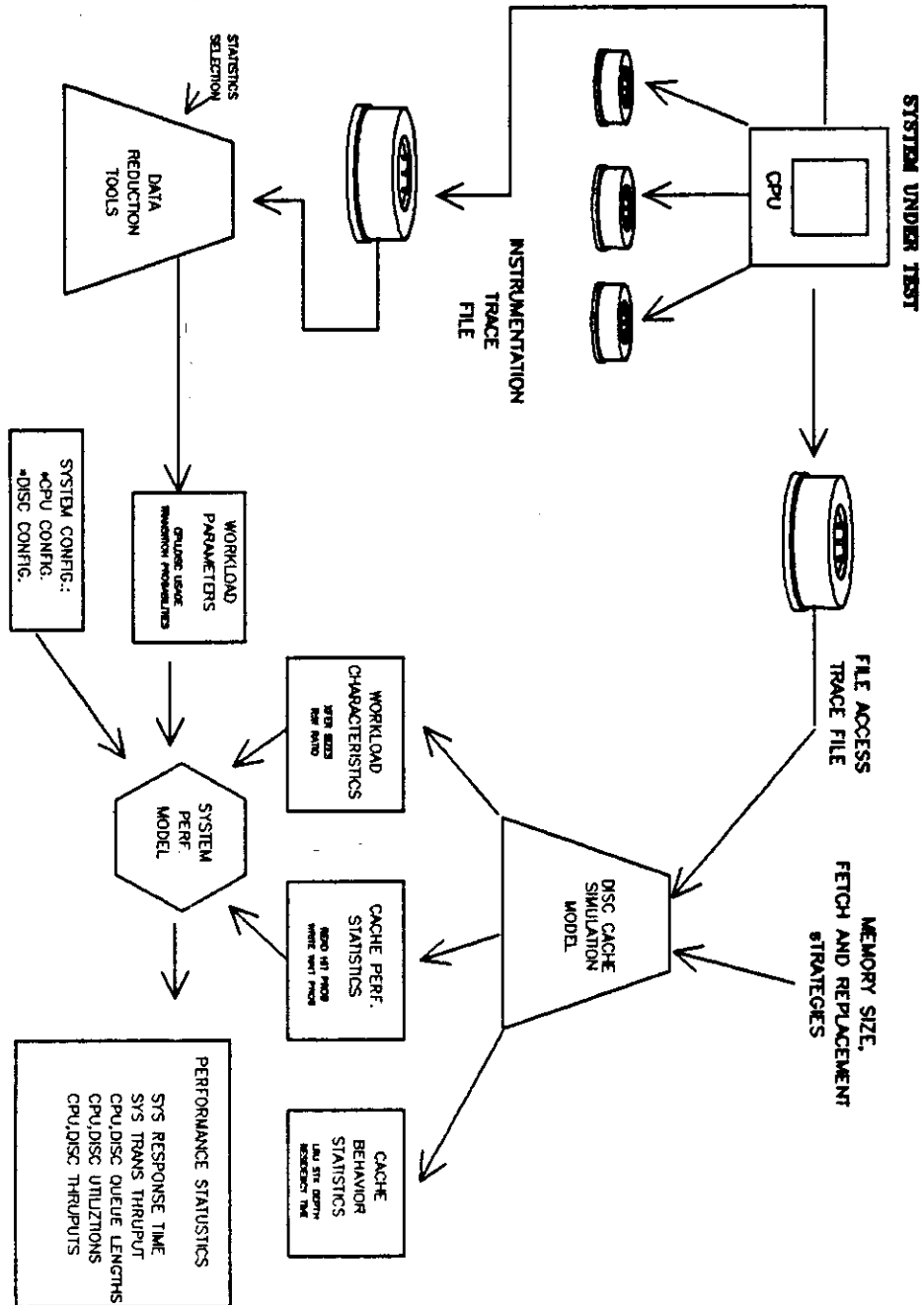
Thus, the kernel structure is based on decomposing the system into resource management modules so that they can be easily replaced, while enforcing within a module strong separation of policy from mechanism so that new algorithms could be implemented in a leveraged manner. Performance was sought through good algorithms, not instruction level optimization.

The bread-board kernel provides full support for all memory management, processor management, disc access and space management, ipc, and semaphore management. It interfaces below to the I/O drivers and above to file systems, databases, and applications. The board kernel consists of roughly 35000 lines of SPL (an Algol-like language), roughly 25000 instructions.

3.2 Performance Measurement and Analysis Tools

The literature was researched to aid in determining appropriate measurement and analysis techniques. Instrumentation was defined, a statistics updating and reporting subsystem was designed and constructed, and display tools were specified and built. Workload generators were collected, and workload characterizers were constructed. A simulation was built to analyze secondary store caching, and a system model was constructed for analysis of the impact of disc caching across a broad range of configurations and workloads. Figure 5 shows the relationship of the various performance measurement and analysis tools which were constructed for this research. Specifications and sample outputs of the tools are provided in the appendices. This section provides an overview of them.

BREADBOARD PERFORMANCE ANALYSIS TOOLS



DBRFXD

Figure 5: Bread-Board Performance Tools

3.2.1 Statistics Support and Display

A Datapro survey of commercially available performance tools is found in [Dat 77]. There are tools to provide information on processor, memory, disc, and other device utilization for load balancing and configuration improvement indications (CUE, Sara, Planr, QCM, Dream). There are tools for displaying the external performance indices directly (SMF, TSO MON, VS/Insight). There are tools which identify heavily used pieces of code to indicate where local optimization will pay off most (QCM, TSA). There are tools which observe the system and dynamically adjust the external tuning controls of the operating system (EQU LPRTY, OPTIMUS, Regulator). There are tools for resource consumption accounting and billing (Mister, Cims). There are tools for database statistics gathering and reduction (IMS/DC, IMS ASAP).

Calloway [Cal 75] discusses the use of IBM 370 VM tools to address the questions of the user, system operator, system analyst, design analyst and installation manager.

Techniques of data collection through sampling and event driven tracing are taught in the performance texts by Svobodova [Svo 76] and Ferrari [Fer 78].

The measurement requirements of this research called for thorough evaluation of the local and global behavior and performance of a variable set of resource management algorithms. User and installation

oriented measurement support was not required. Rather resource demand and utilization statistics with the various resource management algorithms were required.

Statistics including resource queue length and service time histograms, service class compositions, transition distributions, and resource specific event frequencies were supported by the various processor, main memory, disc, and semaphore managers.

The approach to kernel support of the statistics was chosen to be event driven rather than sampled. A sampling approach would have required that the clock interrupt handler be knowledgeable of the current strategies and their supporting data structures in order to find the information required for the enabled statistics. This would have required enhancing and maintaining the sampling clock handler whenever a strategy was implemented or changed. An event driven approach in which in-line macros update centrally located event and time counters was simpler to implement and maintain. The processor overhead of kernel instrumentation updating and displaying is less than 3%, so interference of the measurement with the system under test was not a problem. Interestingly, the current sampling tool for the HP 3000 has a processor overhead of 5%.

In order to avoid having to modify the measurement display tools when the kernel algorithms were changed, instrumentation updating and reading was provided independently of both the algorithm implementations and the measurement display tools. A measurement subsystem was designed and implemented. This subsystem provides

functions for the selective enabling/disabling, updating, and delivering of classes of statistics. A specification of this subsystem and its interfaces is provided in Appendix A.

A statistics display tool was specified and developed. This tool allows the measurer to specify the range of statistics to be measured and the desired duration of measurement. The tool starts statistics collection by invoking a measurement subsystem interface routine, specifying the statistics classes to be enabled. The measurement subsystem constructs measurement collection data structures and enables the corresponding kernel instrumentation through the setting of an enabled measurement class bit-mask. As events occur effecting a statistic, the instrumentation is updated through a measurement subsystem macro. The tool samples the current values of the enabled instrumentation (sampling interval user specified, normally minutes) by invoking the measurement interface's statistics delivery interface routine. The tool appends the current values of the statistics to a log file. When the required number of samples of the instrumentation have been collected, the tool formats the statistics on an interval and cumulative basis. A formatting of a subset of the supported instrumentation is provided in Appendix B.

3.2.2 Workload Generators

In order to evaluate different algorithms across ranges of workloads and configurations, a controllable, reproducible workload generator was required.

Workload generators are discussed in [Svo 76, Fer 78]. They can be constructed from sample programs, traces or synthetic benchmarks which place a sequence of resource demands on the system to simulate resource usage behavior.

A script-driven benchmark workload generator was available. It executes a set of scripts, one per simulated session or job, on a separate computer connected to the system under test through terminal cables. It allows the measurer to specify the number of interactive and batch sessions, distributions for think times between the session transactions, and the script that each session is to execute. Several sets of scripts and configurations which are representative of the user installations were available. The load generator logs the time of transaction initiation and completion. A reduction program produces throughput and response time statistics from this log for each session and overall. This benchmark system provides a good external load generation and performance measurement facility.

3.2.3 Disc Workload Characterizer

In order to analyze the disc subsystem in detail, a trace driven disc workload characterizer was specified and built. This tool consumes a disc access trace file as input, where a trace record contains the time, accessor class, access transfer count, access function and access location for each disc access initiated by the system being traced.

The disc workload characterizer program produces a profile of disc I/O workload characteristics. This includes a breakdown of the workload into its access type components (database, directory, sequential access, etc.), inter-reference times, distributions of transfer sizes, active extent sizes, and extent lru (least recently used) stack reference depth.

The use of the LRU stack depth as a measure of program locality is discussed in [Chu 76, Tur 77, Spi 77]. This tool applies the principle to the referencing characteristics of the discs.

Insight into the spatial and temporal locality of reference to secondary store is provided through the use of the the disc workload characterizer. The referencing characteristics reported by the disc access characterizer are used as input to the analytic system model.

A sample of the output from this tool is provided in Appendix C.

3.2.4 Disc Cache Simulator

The disc cache simulator provides insight into the impact on a workload's disc traffic which would be realized with various forms of disc caching.

The disc cache simulation program simulates disc caching on a disc I/O trace file. It allows refined control over cache management policies (rounding, extent adherence, write handling, fetch sizes for each access type, and flush control). Any subset of the accessors can

be cached, so that the localities of the access methods can be investigated in isolation.

The output of the disc cache simulator decomposes the disc references into mode and access function with cache hit information for each access type. In addition to the cache performance information for the access types, the disc cache simulator also gives cache behavior information, including distributions of cache entry counts, cache reference lru stack depths and cache inter-replacement times.

The hit rates obtained from the simulations are used as the processor to disc and processor to cache transition probabilities in the system model. By obtaining the hit rates through simulation, main memory size and contention do not need to be captured by the system model. This significantly reduces the complexity of the system model.

The disc cache simulation program uses the approach of building an LRU reference stack for its replacement decisions. The use of an LRU reference stack and program reference traces is a standard approach [Spi 77] for analyzing fault rates for global LRU type replacement algorithms in main memory management. This simulator applies the same techniques to disc reference traces in order to evaluate alternative fetch and flush policies in disc caching.

Figure 6 compares simulated and measured cache hit rates for various workloads. The simulation predictions are consistent with measurement within 5%.

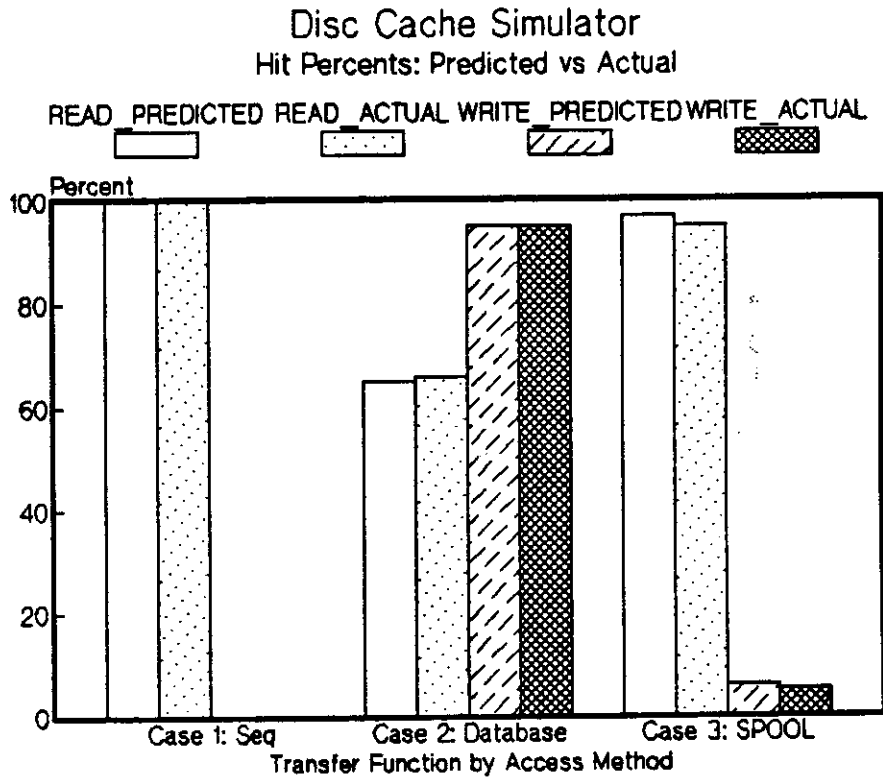


Figure 6: Disc Cache Simulator Verification

The disc cache simulation consists of roughly 3000 lines of Pascal. It requires roughly 1 CPU minute on a 1 MIP machine for each 10,000 trace records processed. Appendix D describes the implementation of the disc cache simulator, and gives a sample of its output.

3.2.5 Analytic System Model

A custom analytic system model was built for this research. A custom model was required in order to explicitly include a disc cache service station. It allows the user to specify the system configuration and workload characteristics, and produces global performance estimates as well as resource demand and utilization statistics. The system model explicitly captures the effects of alternative secondary store caching mechanisms, including external caching of discs through an intermediate storage level and internal caching of discs through file mapping or explicit caching in primary memory.

The system model queries for workload and configuration information. These inputs are obtained from knowledge of the installation, from the disc cache simulation, and from the statistics collection and reduction tools. Inputs are specified for processor speed, disc configuration and speed, channel speed, processor and disc service demand, disc cache overhead, and disc cache read hit rate, fetch size, and post handling method.

Given the input system configuration and workload characterization, the system model program constructs a closed queueing model. The total workload is aggregated, and the network is decomposed into a simple central server model. For internal caching, the disc cache service station is absorbed into the processor station. The resulting central

server model is solved through a standard iterative procedure [Lav 83] which iterates on the effective multiprogramming level.

Performance and behavioral characteristics are calculated for each server, and the system as a whole. For each iteration on the mean multiprogramming level, the utilization, mean queue length, throughput and response time of each server is calculated, as is the overall system response time and throughput.

Lavenberg and Sauer [Lav 83] provide a handbook on current modelling techniques. Techniques for handling open networks, general arrival or service distributions, multiple servers, state dependent queues and multiple service classes through exact and approximate methods are presented.

These refinements over the simple central server model are useful in obtaining accurate performance prediction in general situations. To obtain approximate insight into the impact of disc caching across a range of system configurations and workloads, the model constructed for this research was adequate.

The impact of serialization is not explicitly captured by the model. Models capturing workload serialization are explored in [Thom 83]. Serialization due to contention for access control semaphores can have a significant impact on system performance. Its effects can be approximated by reducing the effective multiprogramming level.

The system analytic model consists of roughly 1500 lines of Pascal, and requires less than 1 CPU second on a 1 MIP machine for calculation. A description of the system model with sample input and output is provided in Appendix E.

Chapter 4

Workload Characteristics

This chapter discusses the referencing patterns and the processor, semaphore, disc and main memory resource usage profiles of representative workloads for the case study family. Relationships observed during the course of the research between workload characteristics and resource management policies are noted. The resource management policies which are referenced in this chapter are discussed in detail in the following chapters.

We examined the workload characteristics of the case study family for a number of reasons. There is a basic need for research data on real workload behavior. We wished to identify the scope of applicability of the empirical results on algorithm analysis which we generated. Knowledge of the workload characteristics helped us to better understand the behavior of the algorithms we studied. Finally, an understanding of the workload behavior helped to focus our search for technological improvements in resource management algorithms and system components.

The bread-board kernel and measurement tools were used to examine the workload characteristics of the case study family over a number of years as the family evolved.

The memory and processor characterizations reported in this chapter and the algorithm behavior characterizations reported in the next chapter were gathered using a benchmark called EDP which consists of 48 sessions and two batch jobs. The system under test was running the bread-board kernel without internal caching of secondary store on an HP 3000 Series 44 with 2 Mbytes of main memory, and 2 HP 7925 disc drives. The benchmark was running on a separate computer which was generating the load on the system under test through 48 terminal cables, one for each session.

The scripts consisted of : a 2500 line Cobol program performing 48 file inquiries, 14 file updates, and 7 file appends through the VPLUS/3000 and IMAGE/3000 subsystems; a data inquiry script using the QUERY/3000 subsystem on a shared database with 10 FINDS and 9 REPORTs performed on data set containing 455 entries; and a program development script consisting of a compile and link of a 1200 line COBOL program. Each script was being executed by a third of the sessions.

The salient workload characteristics of processor, main memory and semaphore usage are similar in most workloads. The disc referencing patterns however vary considerably between workloads, depending primarily on the access methods which dominate in the workload. The general behavior of the access methods at different installations tends to be similar though.

The disc spatial and temporal locality characteristics reported in this chapter were obtained from an HP internal installation used by product support for manual and training material production and for

database tracking of operating system bugs. The installation was monitored for 5 consecutive days during a two hour window each morning. The disc data presented in this chapter comes a representative sample during which 9 users were performing text editing with the TDP utility, 2 users were performing graphics editing using the HPDRAW utility, and 7 users were performing database queries and updates against an Image database of pending bug reports.

Instrumentation samples of the research lab's timeshare machines taken over a number of years are also referenced in and the following chapters.

The chapter begins with a discussion of transaction behavior. The chapter then proceeds through a discussion of main memory, processor, semaphore and disc usage. Object types, their size distributions, and memory reference localities are described. Processor burst and semaphore request/block profiles are presented. Disc spatial and temporal referencing behavior is characterized as a function of accessor mode (database, directory, sequential). The chapter concludes with a summary of the salient workload characteristics.

4.1 Sessions and Jobs

A session consists of a user at a terminal carrying on one or more transactions with the system. A transaction begins with the user providing input through the terminal. In executing a transaction, the corresponding process locks semaphores protecting shared data, reads and writes data in one or more files, outputs to a window of the screen, and waits for the next input. A job behaves like a long transaction to the system, except that it uses disc or tape files for input and output.

The distributions of think times between transactions tend to be highly installation and time-of-day dependent.

4.2 Memory Usage

The segmented nature of the HP 3000 architecture and the operating system and subsystem structure influence the address space decomposition and referencing behavior of a workload. The memory objects in the system include stacks, code segments, data segments, and domains of the disc cached by the kernel. This section explores the relative usage of these objects, their size distributions, their system/user decomposition, and the extent of sharing found in the case study family.

4.2.1 Memory Usage By Objects

Figure 7 shows the relative partitioning of memory between active objects based on space occupancy and number of objects in a typical steady state. The relative space partitioning is more a function of memory size than application mix, with the portion allocated to cached disc domains falling as memory size decreases.

Memory Utilization by Objects

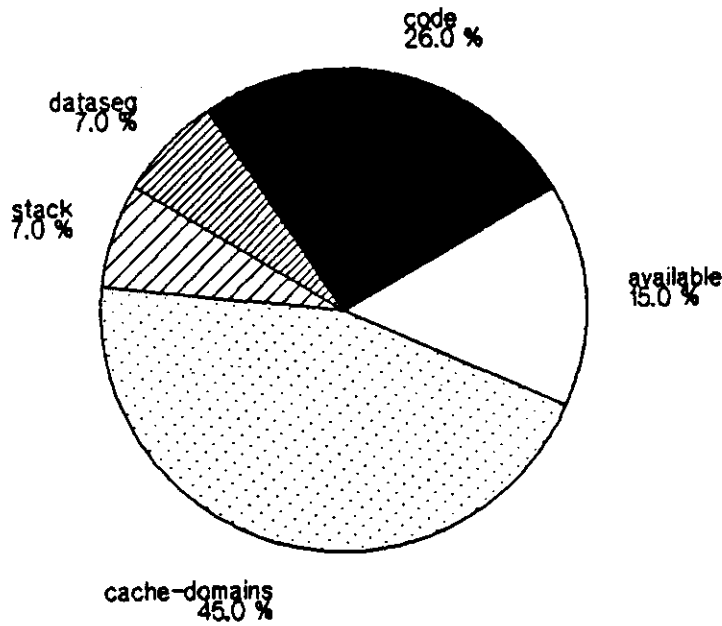


Figure 7: Active Object Occupancy in Memory

4.2.2 Object Size Distributions

Figure 8 presents normalized distributions of the sizes of active objects in a typical steady state.

The code distribution is very similar to those reported by Batson [Bat 70] for the Burrough's 5500, and Batson and Brundage [Bat 77] for Algol 60 programs, though the means are somewhat larger.

The distributions are skewed toward the lower range of potential object sizes.

For code and data segments, this is due to operating system and program developers taking advantage of the segmentation to group logical units together into segments that are referenced together. Infrequently referenced units would be grouped together into one segment, or placed in segments by themselves, so that they would be in main memory only on those occasional times when needed.

Such restructuring for improved locality has been recommended by Hatfield [Hat 81], Ferrari [Fer 74,76], and [Denning 76]. Opale, a tool for restructuring programs for paged virtual memory systems discussed in [Ach 78], claims a reduction of 40 to 70% in the page fault rate for the Sirius 8 operating system.

The data segments are used to keep track of process and job state and accounting information, for file and database buffering, and for application specific use.

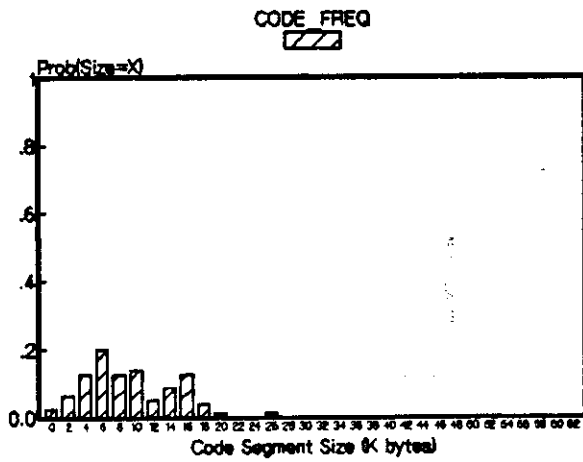
Cached disc domain distributions have two peaks, with their relative heights a function of the ratio of sequential to random access for files and databases. The peak of small cached disc domain sizes is due to caching of writes and cached reads of randomly referenced domains. The larger peak is due to prefetching of large disc domains on

sequential read access. This peak has low frequency since these domains are flushed immediately after use.

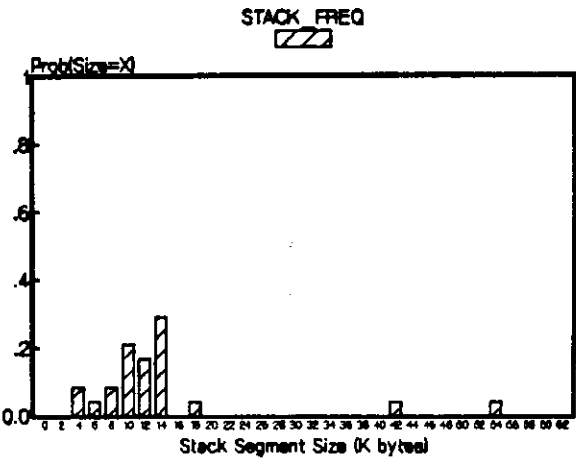
The characteristics of the object size distributions are consistent across workloads and over the years. However, the means of the code and data segment distributions gradually doubled after the system's sensitivity to segment size went away through improved memory management and larger memories, and the cached disc domain mean size will gradually become larger as subsystem and application designers feel out the new tradeoffs introduced by disc caching.

These shifts give an indication of the way application and subsystem designers allow the characteristics of the services they use to influence their decisions. This is not uncommon, indeed encouraged. For example, Paulhamus and Ward recommend specific blocking factors for programs executing on the IBM 370 under MVS based on their empirical study of the effect of varying blocking factors using a particular operating system version and configuration. The benefits achieved by specific data clustering in the IBM 370 MVS environment are also discussed in [Pau 77].

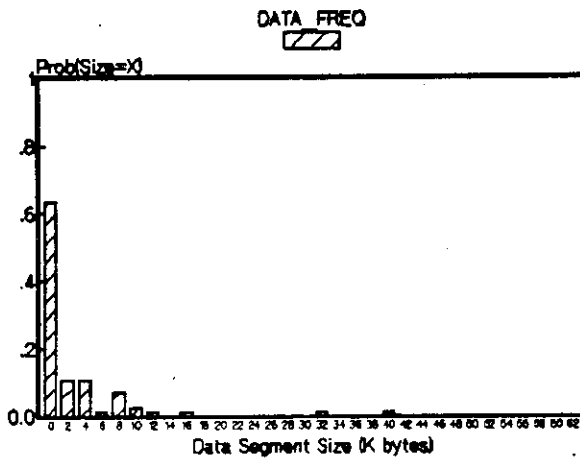
Code Segment Size Histogram



Stack Segment Size Histogram



Data Segment Size Histogram



Cache Segment Size Histogram

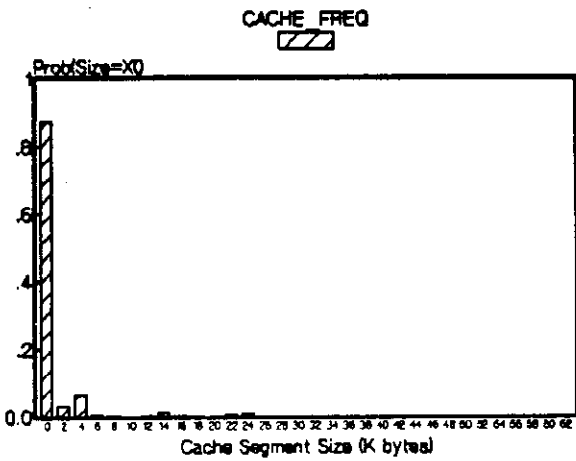
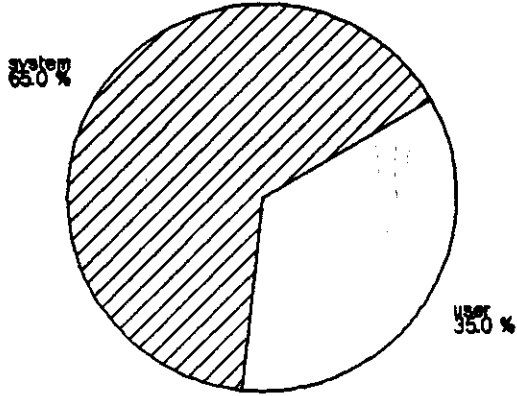


Figure 8: Object Size Distributions

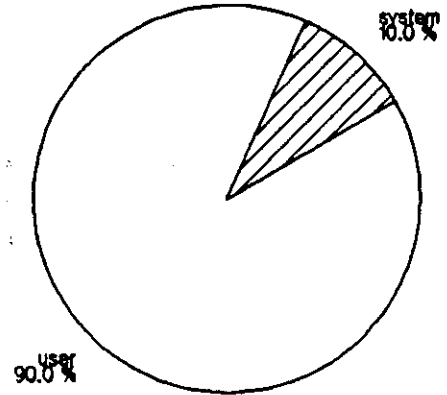
4.2.3 User/System Object Partition

Figure 8 shows the relative partitioning of active code and data between system and user domains. The high percentage of system code results from the operating system and subsystems doing most of the work, while user code just invokes services. In distinction to code, data object space in memory during steady state processing of a normal workload is dominated by user data.

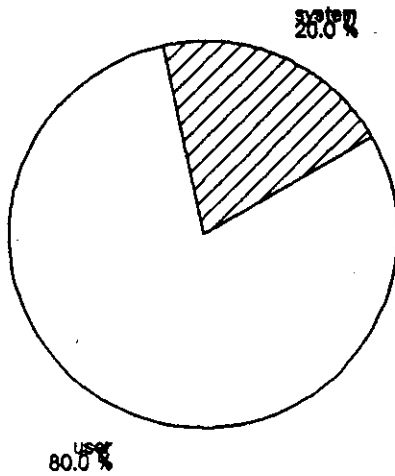
Object : Code Segments



Object : Stacks



Object : Data Segments



Object : Cache Domains

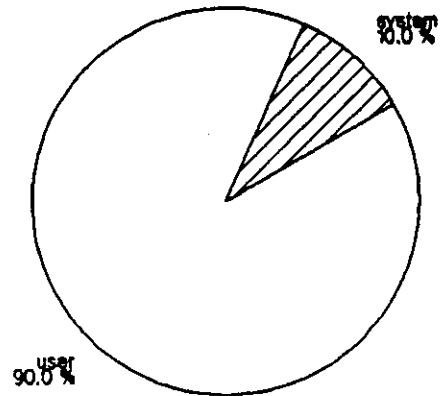


Figure 9: User / System Partition of Active Objects

4.2.4 Memory Reference Locality

Figure 10 shows a breakdown of a typical process' working set based on measurements made with the bread-board kernel using a per process working set replacement policy with the working set window at 150k instructions. This does not include cached disc domains. They are referenced indirectly through the buffers in the data segments.

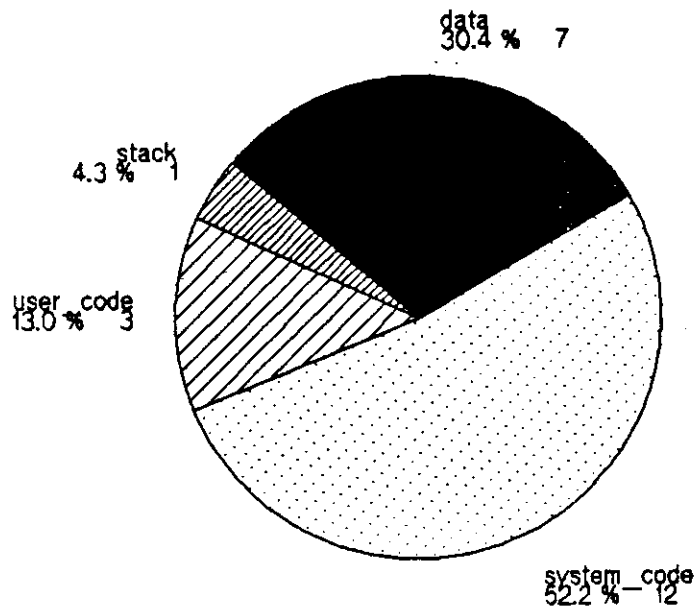


Figure 10: Working Set Composition

The memory referencing behavior is similar to that reported for other systems. The working set characteristics [Den 68, Spi 72] are evident, as shown in Figure 11. Observe the rapid drop in fault rate

for code and data objects as the working set window widens, and the fairly rapid plateauing. The drop in fault rate is much less for the user code since it accomplishes its work through system and subsystem services, and so is more compact and tightly referenced.

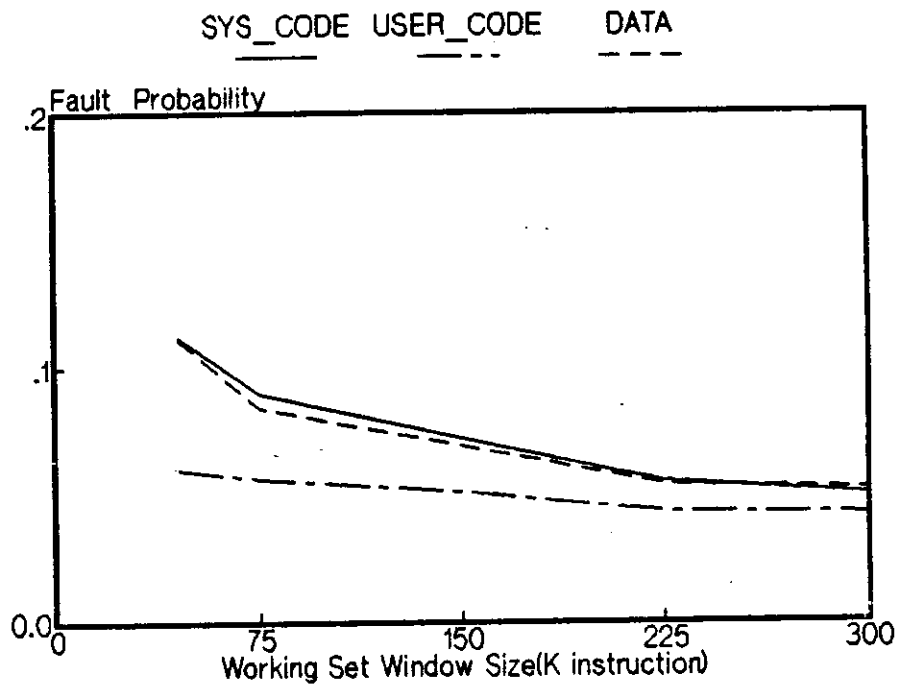


Figure 11: Memory Reference Behavior

4.2.5 Object Sharing

All code segments are read only and sharable. All processes share the code segments comprising the operating and subsystem services. All

processes executing the same program share the same segments of the program, and typically many processes are executing the same program.

File and database data segments are shared by all processes accessing the same file or database. Job related data segments are shared by all processes which are part of the same job or session. The stack and process related data segments are private to the process. All cached disc domains are globally shared.

This high degree of code and data sharing can be expected to have an impact on the performance of memory management algorithms. That sharing and not just process locality need to be considered was clearly demonstrated when comparing replacement algorithms.

Two versions of the working set algorithm were implemented, one version ignoring object sharing, and one version taking working set intersections into account. The version ignoring sharing releases the space occupied by an object immediately when it leaves the working set of a process in the current multiprogramming set or when it is in the working set of a process being swapped out. The version taking sharing into account releases the space occupied by the object only if the object is not in the working set of another process in the current multiprogramming set. Note that an object can be recovered even though its space has been released until it is actually overlaid in memory.

Figure 12 shows the impact. Observe the high fault and recovery rates for the data and system code segments when sharing is not accounted for.

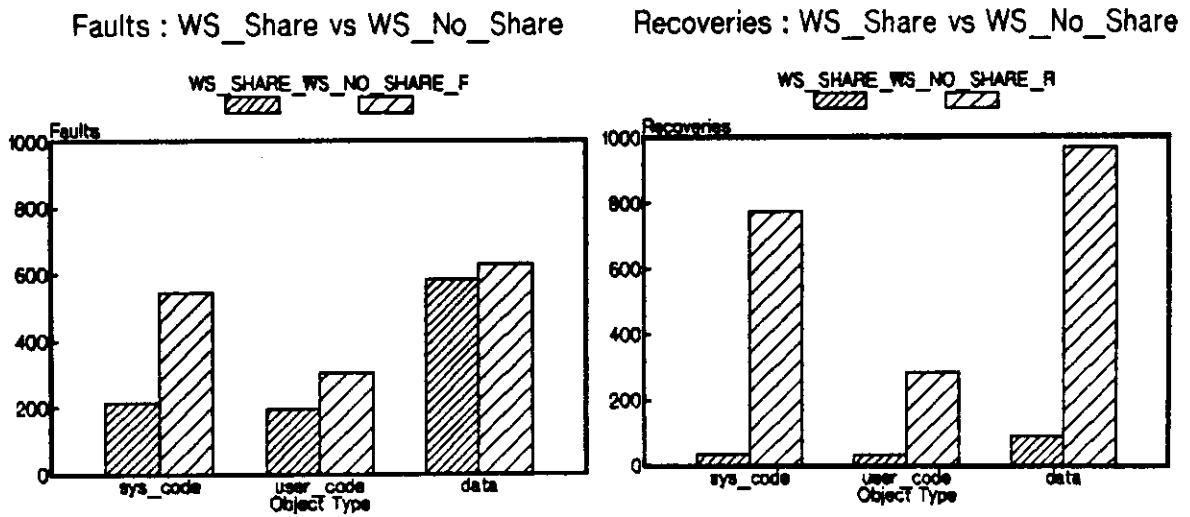


Figure 12: Effects of Sharing on Replacement Policies

4.3 Processor Usage

Figure 13 shows a distribution of processor service time per visit. The exponential character is classic [Kle 75].

CPU Burst Interval

No jobs vs. One job

NO_JOBS

ONE_JOB

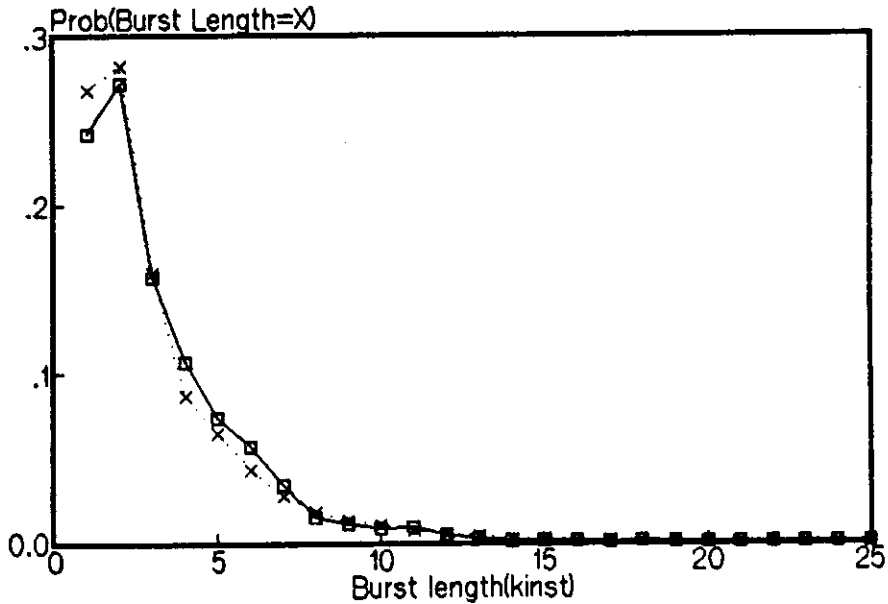


Figure 13: Processor Burst Distribution

Observe that the presence of jobs in the mix has minor impact on the characteristics of the distribution. Jobs use the same services that interactive transactions do. The implementation of these services dominates behavior, not service class.

The mean of the processor service time distributions shifts however, based on changes in the algorithms. When disc caching was introduced, the mean tripled since 90% of the transitions from the processor due to a disc service wait were eliminated, thus increasing the service time per visit and reducing the number of visits required to perform a

function. Some of the swap and replacement algorithms cause the means to become smaller due to high fault and preemption rates.

The total processor requirements to perform a functions depend on the function and how it is implemented. A relational database implementation for the HP 3000 family uses ten times the processor to perform the same function as a database employing a network implementation.

4.4 Semaphore Contention

A description of the use of semaphores for synchronization and mutual exclusion can be found in [Dij 65, Cof 73, Fra 73, Sha 74, Shr 75]. The impact of semaphore contention on performance is modelled in [Thom 83].

Semaphore facilities are provided at the system and user levels. The operating system uses over fifty distinct semaphores to synchronize access to shared resources. These semaphores protect memory and disc resident data structures, and serialize access to services which cannot be provided in parallel. The operating system provides an unlimited number of user defined semaphores, as well as facilities for their acquisition and release. The user level semaphores are defined and manipulated by subsystems and applications.

Contention for operating system semaphores is normally low, contributing less than 2% to the process stop distribution even under heavy loads. The probability of finding a system semaphore busy in a

normal workload is below 5%. A profile of system semaphore usage and conflict is shown in Figure 14. The top graphs show the system semaphore request/block profile and holding times without batch jobs in the mix, while the bottom graphs show the impact on the profiles and holding times due to batch jobs.

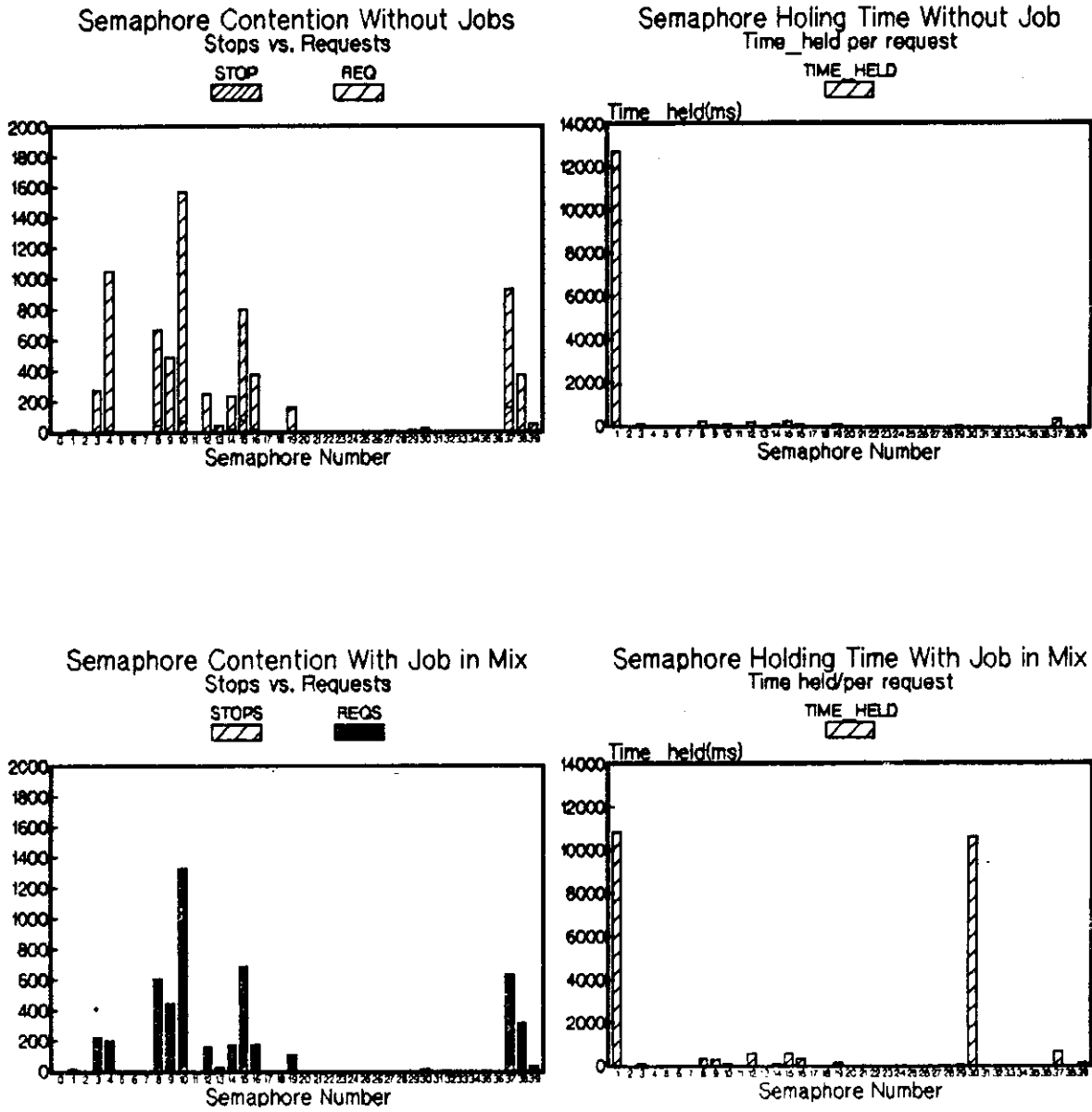


Figure 14: System Semaphore Usage Profile

The above figure shows the holding time profile of the system semaphores. It is not a coincidence that the semaphores with long holding times are infrequently used. The system designers refined lock granularity to insure that semaphore contention remains low. The semaphores held for long duration are relatively infrequently used so the development effort of providing refinement wouldn't pay off.

The presence of jobs in the mix increases the mean holding time of the semaphores. This is an indication that semaphore holding time is load and system policy dependent.

Contention for user semaphores is very high in a significant portion of the installations. This is due primarily to insufficient parallelism in the database. In installations in which many of the users are sharing the same database, semaphore waits contribute over 50% to the process stop distribution, and over 90% to the realized response time.

4.5 Disc Usage

The disc traffic patterns depend not only on memory size but also on the management policies employed by the kernel, file system and database.

In small memories (< 2 Mbytes), code and process control structure traffic can be significant, with the rates and relative mix of objects flying in and out highly dependent on the memory management algorithms. This behavior is explored in the next chapter.

We saw above that the code and transient data objects comprising a process' working set can fit in 100 kbytes. The high degree of code and data sharing makes the memory requirements for transient objects strongly sub-linear with respect to the number of active sessions. Thus, in large memories, disc traffic is dominated by file and database referencing patterns (assuming decent swapping algorithms).

Figure 15 provides a profile of disc reference locality on an aggregate basis. It shows a typical distribution of inter-extent reference times and extent LRU stack depths. Notice that although the mean inter-reference time is often minutes and more, the median is on the order of milliseconds. The small median and relatively small mean of the LRU stack depth distribution of inter-extent references indicate that references are localized to a relatively small subset of the disc space occupied by files and databases.

Inter Extent Reference Interval Distribution Stack Depth Distribution of Inter Extent Ref.

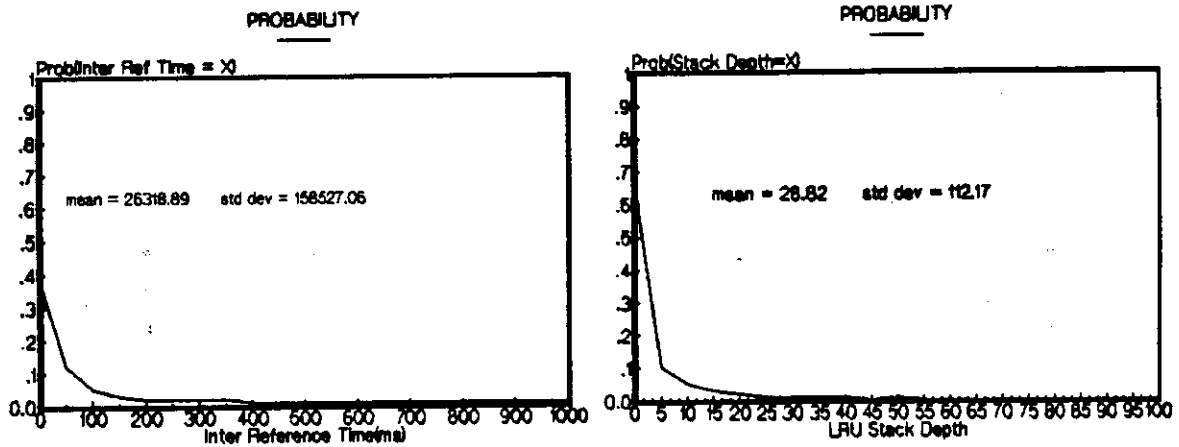


Figure 15: Inter-Extent Reference Locality

These locality distributions normally have very large standard deviations, consistent with the findings of Smith [Smi 78] on long term file referencing patterns.

The primary access methods are directory, sequential, and database. The directory manager maintains a hierarchical disc resident data structure which keeps track of the security and disc locations of all files. Sequential access is used primarily in file copies and compilations. The IMAGE database system presents a network model with hierachically structured data sets and predicate locking for concurrency control.

The disc workload analyzer and disc cache simulator were applied further to investigate the access methods' use of secondary store and their inherent spatial and temporal locality of reference in detail.

Figure 16 shows the cumulative probability distributions and characteristic means of the disc access transfer sizes made by the access methods. The read to write ratios are shown as well. Observe that all the access methods tend to move chunks of about 1 kbyte between their buffers and their total address space. Directory and database movement is highly skewed towards reads, while sequential movement is split evenly between reading and writing.

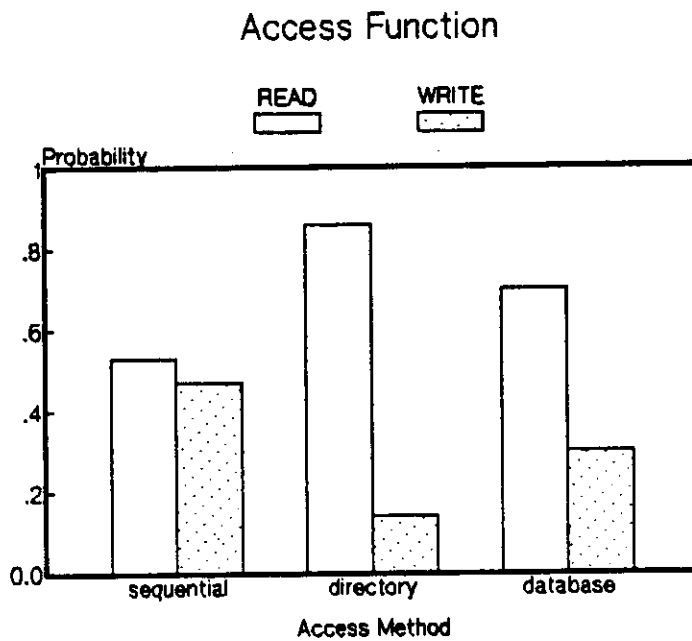
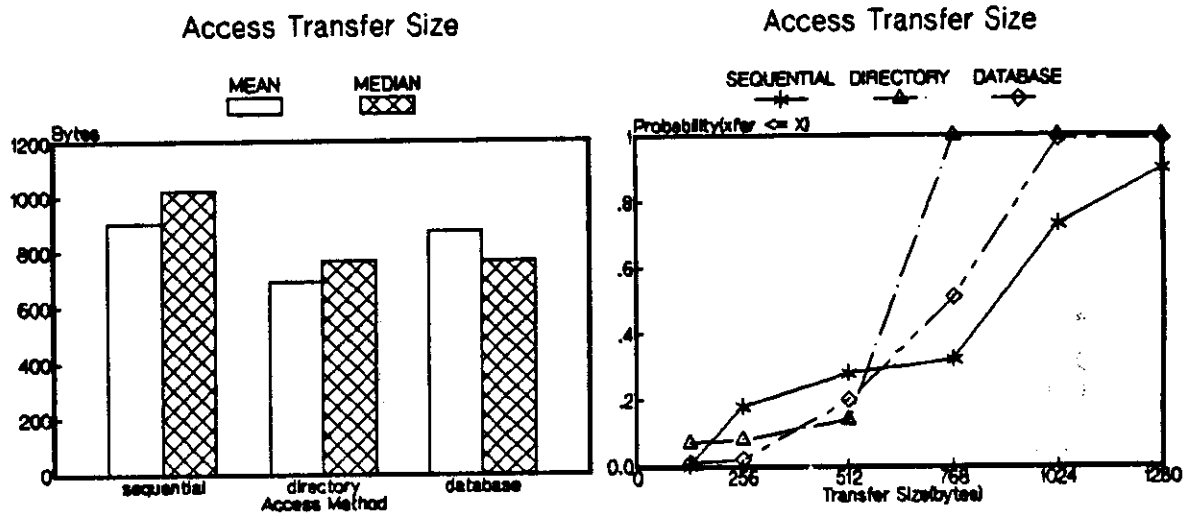


Figure 16: Access Method Use of Secondary Store

In order to examine the inherent locality of the access methods, disc traces from large installations were obtained and the disc cache simulator was run on them with an infinite cache size, caching only one access method at a time.

Figure 17 shows representative spatial locality characteristics of the access methods. The cumulative probability distribution function of the LRU stack depth references and the means and medians of the stack depth distributions are shown for directory, database and sequential access.

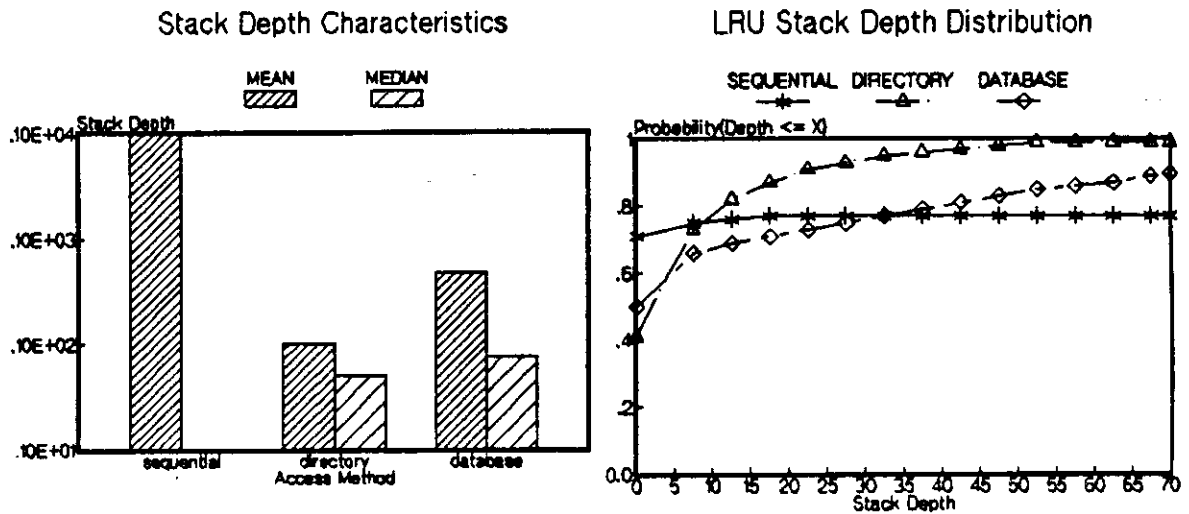


Figure 17: Spatial Locality of Reference of Access Methods

Figure 18 shows representative temporal locality characteristics of the access methods. The cumulative probability distribution function and the means and medians of the time between references to blocks of the address space of the access methods are shown for directory, database and sequential access.

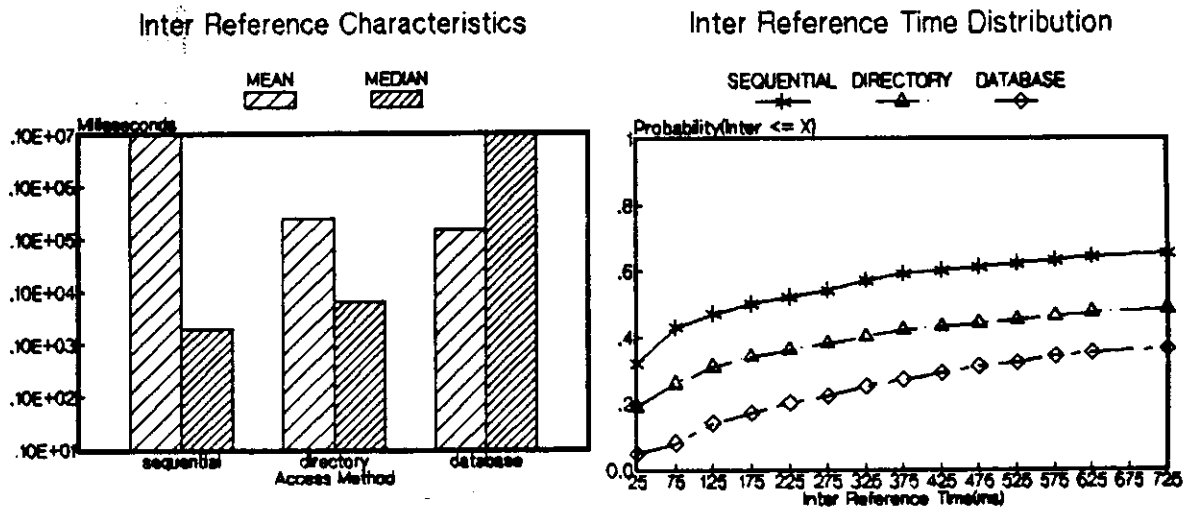


Figure 18: Temporal Locality of Reference of Access Methods

Sequential access shows a high degree of temporal and spatial locality within a small portion of its address space at a time. It's references are milliseconds apart to its top of stack elements, then those elements are never again referenced. The large mean in the LRU stack depth for sequential reference is due to misses on writes which

are related to data created by the sequential process and being posted for the first time to previously unused regions of the disc.

Directory access shows a fairly high degree of spatial locality of reference within its reference space, with 90% of its references occurring within the top ten LRU stack elements, but a relatively low temporal locality of reference, with over 50% of its repeat references to the same block being at least minutes apart.

Database access shows considerably less spatial and temporal locality than the other access methods, with one fourth of the references outside the top 50 LRU stack elements and over half of the referenced blocks are never used again for hours or days (thus the infinite median inter-reference time for database access).

Rosell [Ros 76] reports that in his study of IMS he found the working set size of the database system was linear with respect to the window size, indicating poor locality. Sequential access dominated the referencing patterns in his study.

The IMAGE database locality characteristics presented here do indicate relatively poor spatial and temporal reference patterns. However, we will see in Chapter 6 that the database locality patterns are poor only relatively speaking. Notice that the cumulative LRU stack depths and cumulative inter-reference time distributions continue to grow, although slowly, with increasing stack depth and inter-reference times. This indicates that caching database references in a high speed memory would continue to provide gradual increases in

hit rate as the cache size increases, although the inter-reference times of cached pieces could be minutes or more apart.

4.6 Summary of Salient Workload Characteristics

The workload characteristics of the case study family changed over time following the evolution of the system hardware and software components. The workload characteristics depend on the processor and system architecture, and on the algorithms in compilers, applications, databases, file systems and kernels. The functions and algorithms of each of these system components evolved over time, out of phase with one another, and the evolution of the workload characteristics followed the evolution of the system components.

The general characteristics of this case study workload are :

- a). Most of the work is performed by database and operating system software on behalf of the applications. This results in system code dominating the working set of transient objects.
- b). Program referencing patterns involving code and transient data follow the classical working set model of program behavior, with a working set window of a quarter million instructions providing adequate locality.
- c). There is a very high degree of sharing of code and data, resulting in a high degree of intersection between process working sets.

- d). Processor, memory and disc service requirements and characteristics vary significantly with the structure and algorithms of the operating system and subsystem. Processor burst intervals between disc accesses are relatively short, on the order of 6 kinst without internal caching of secondary store.
- e). There is light contention for the system semaphores, but there is heavy contention for application semaphores in database intensive workloads. The semaphores are often kept locked during disc accessing.
- f). The access methods move about 1 kbyte between their external and internal stores per transfer, with a 5:1 read to write ratio for random type access (directories, database) and a 1:1 ratio for sequential access. Transfer sizes of segments follow the segment size distributions.
- g). The locality of reference characteristics of the access methods differ considerably between the access methods, with sequential access having extremely high spatial and temporal locality of reference, directory having high spatial but moderate temporal locality of reference, and database showing relatively low temporal and spatial locality of reference. Sequential access tends to reference the top LRU stack elements at milliseconds apart, then never references them again. 90% of directory references occur within the top ten LRU stack elements, but 50% of repeat references are minutes or more apart. Over 25% of

network database references are outside the top 50 LRU stack elements, and over half of the referenced blocks are not referenced again for hours or days.

h). Batch jobs generate similar workload demands on processor, main memory, disc and semaphore resources as do interactive transactions.



Chapter 5

Basic Algorithms

This chapter presents the research experience into basic disc, memory, processor and semaphore management. Disc access scheduling, global priority assignment and reflection in local decisions, processor and semaphore allocation, and main memory allocation, replacement, and garbage collection are discussed. Integration of higher level data management is presented in the next chapter.

Computer literature searches and texts on performance and computer resource management provided a base for the selection and analysis of resource policies. The existing system and workload were investigated to gain an understanding of general requirements and behavioral characteristics. Algorithms for the various resource managers were implemented, and their local and global effects investigated. Principles and improved algorithms were proposed and investigated. The results of these investigations into resource management policies are related in this chapter.

The chapter begins with a description of the investigation into secondary store access and space management. This study gave the clue to integrating the basic algorithms : don't optimize locally, rather

perform the service request which contributes most towards achieving the system performance objectives.

Next, the global priority assignment scheme is presented. This assigns the priority of processes based on external policy specifications. These process priority assignments are then reflected in the service requests for the various system resources so that their management decisions contribute to the global objective.

Processor allocation is then discussed. The use of idle time for background garbage collection, and the interactions with the memory manager to provide protection of the urgent processes in the current multiprogramming set are presented.

The semaphore allocation policy and priority adjustment of the holder when a more urgent process requires the semaphore is presented.

Next the empirical results on alternative memory allocation, replacement, and garbage collection algorithms are presented. The importance of parallelism, of taking sharing into account, of proper interfacing with disc and multiprogramming set scheduling, and of adaptable algorithms is demonstrated.

Finally, the conclusions from this empirical investigation into processor, memory, disc and semaphore management are summarized.

5.1 Secondary Store Access and Space Management

The investigation into secondary store management indicated some principles and lead to some interesting algorithmic results. This section reviews the previous research into secondary store space and access management, and reports on the findings in these areas uncovered during this research.

5.1.1 Previous Research

Teorey and Pinkerton compare disc scheduling policies in [Toe 72]. The performance criteria used are expected seek time, expected wait time, and the variance of wait time. Their objective was to determine the best design of a basic disc subsystem. They do not consider the nature of the processes generating the requests. Based on simulation and analytic studies, they conclude that variations on selecting the next request for service to be the one with the shortest seek time in the current scan direction provide the best service. Further, different variations are optimal depending on whether the device is lightly or heavily loaded.

Fuller and Baskett analyze drum storage units in [Ful 75]. They analyze storage organizations on the drums and scheduling algorithms for drum access with multi-stage Markov models. The storage organizations considered are for file and paging support. The access scheduling algorithms considered are first-in-first-out and shortest latency time first. The drum is analyzed independently of the rest system. with expected waiting time as the performance measure. They

conclude that scheduling discipline is more important than drum organization.

Baudet, Boulenger, and Ferrie analyze drums allowing multiple page transfer requests in the arrival process. The generalization to include "bulk" transfer requests is made so that the effects of operating systems which swap multiple pages of a locality set can be captured. They feel that "in order to build an operating system, one must, first of all, solve the problem of secondary memory access." Through analytic and simulation methods, they determine the waiting time of bulk requests as a function of drum utilization and the requested transfer size. They conclude that it is important to distribute the single requests of the bulk requests on adjacent regions of the disc.

Smith [Smi 78] says that the remaining research problems having to do with discs involve intelligent discs, discs associated with gap filler technology, and access scheduling strategies.

Chin [Chi 79] discusses the management of free space within databases to minimize head movement.

Platz, Blackledge and Hughes describe DEC's HSC50 storage subsystem in [Pla 83]. This storage subsystem provides service for multiple hosts. It manages up to 24 discs or tapes, providing global sharing of mass storage resources. The control of the storage subsystem is managed by multiple, parallel microprocessors. The control RAM of the subsystem maintains the access request queue, which can be as large as

1000 requests. The cylinder and instantaneous rotational position of all heads on all drives are kept track of, and requests for a device are serviced based on minimal seek time.

5.1.2 Bread-Board Results

The workload characterization indicated that the processing requirements of a transaction were small compared to the disc access requirements (4000 instructions/file system disc access). Thus, the performance of the disc subsystem would be expected to be critical to the global performance of the system. Optimization of this subsystem would likely provide the largest payoff. Direct improvements in system performance would be expected from adding more discs, controllers and channels for increased parallelism (more servers).

Analysis of the existing system indicated that increasing disc access capacity provided minimal performance improvement. Further, it was reported that disc head optimization had been attempted, but had been demonstrated not to be worth the overhead in software.

After instrumenting and measuring the existing system, the causes were obvious. The disc queue lengths were skewed toward one device, and most of the time most of the disc queues were empty.

Further analysis indicated why. The disc space for all the stacks and data segments was restricted to reside on a single disc, so that any process swapping would have to queue up at this device. Thus the disc queue skewing. The problem causing the queues to be empty most of

the time was that the memory manager was a serial server. Until one swap-in completed, including all the disc accesses required to accomplish it, no further swap-in requests could be initiated. Thus, the service delay for the memory management system was on the order of hundreds of milliseconds, so that it was impossible to sustain a high multiprogramming level under memory pressure.

The method to achieve a disc queue balance was straight forward: allocate swap space across the discs and spread file extents among the disc volumes. Keeping disc queues long so that the access to secondary store is used to capacity requires a memory manager capable of sustaining a high multiprogramming level through parallel service.

Figure 19 compares the HP 3000 production operating system in use at the time (MPE III) with the final breadboard kernel. It shows the mean queue length distributions in a busy system. The bread-board kernel has transient object disc space spread across the volumes, provides parallel memory management service, and has background disc traffic for data posting. In MPE III, the disc space for system control structures is centralized on a single volume and the memory manager is a serial server. Disc queue lengths are long and balanced when the system is parallel and balanced, and short and skewed when the system is serial and skewed. This is obvious from queueing theory (see Appendix E for queue lengths as a function of transition probabilities), but it really can happen.

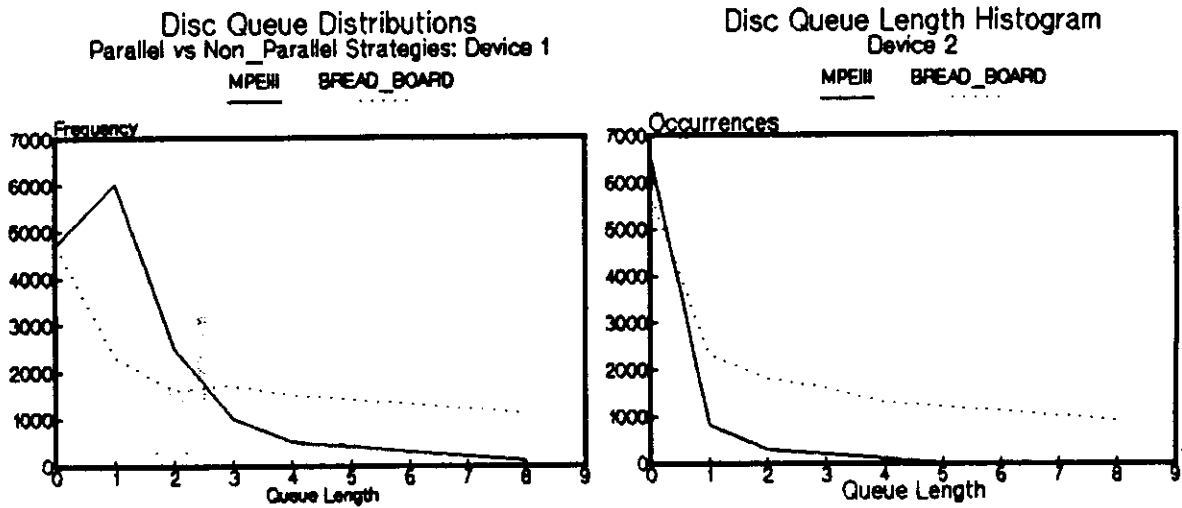


Figure 19: Disc Utilization With Seriality and Clustering

Users are normally given control over file placement among volumes so that they can balance references across the devices to maximally exploit parallel service between main and secondary store. Optimal file allocation on disc units is discussed in [Boi 78].

Once the disc queues have some depth, scheduling policies can play a role. A large number of disc access scheduling policies were investigated, and their impact on the local and global performance factors observed.

The shortest seek time first algorithm did maximize disc subsystem throughput relative to other policies, but the system performance effects were disastrous. Certain processes got excellent response time while the response time of others went to infinity.

The enhanced algorithm of selecting the next request as the one with shortest seek time in the same direction insured service to all requests, but it favored processes doing no-wait sequential access. Such processes are often background jobs. The system level result of this policy was to give excellent service to the jobs at the expense of degrading response time for the interactive sessions doing database queries which require more random disc referencing.

This provided a hint for improving the service policy : take into account relative system urgency among the service requests.

But, what is the relative urgency among the different memory management initiated transfer requests (code vs stack vs data segment fetch and data vs stack stores)? What is the relative urgency of memory management transfers with regard to file transfers. And what is the relative urgency among the file transfer requests? This called for experimentation.

The next policy investigated was to service memory management requests before the file access requests. Memory management is important, right? This policy favored batch jobs by allowing their swapping activity to hold off the file accesses for interactive

sessions, thereby increasing their response times and decreasing their throughput.

The correct policy was coalescing. The next request to be serviced should be the request which contributes most to system performance objectives directly, towards transaction throughput and response time.

The code for the disc access policy was amended to support combinations of service priorities. It was driven by a bitmask which selected the weightings for the various access types.

The results of the investigation gave the key to integrating the policies for all subsystems. The policy that gave the best performance was to select the request required by the most urgent process waiting for the transfer to take place.

The determination of urgency and mechanism for expressing it is simple. Process priorities are assigned and adjusted to optimize the global performance objectives. These objectives can be reflected in the disc subsystem by tagging the requests for disc service with the priority of the most urgent process waiting for the transfer to complete, and by performing pure priority servicing for each device based on these priorities. Thus, memory management transfers get integrated with file system transfers and the transfers for batch jobs get performed as background activity.

Investigation into further anomalies indicated additional requirements on disc scheduling. The disc scheduler had to provide the requestor with on-going status information, allow the issuer to

change the priority that a request was initially tagged with, and to allow the requestor to cancel a pending service request. These came about as follows.

The memory manager, in order to speed up fetch requests later on, pre-issues disc write requests for data segments which are good replacement candidates. The writes are issued at a background priority so that they will be serviced only if the disc would be otherwise idle since no process currently requires the region in which the segment resides. If the data segment is referenced before it is actually overlaid, the memory manager needs to cancel the pending write request if it has not already completed. On the other hand, the memory manager needs to increase the priority of the request to that of the process getting memory scheduling if the region occupied by the data segment is selected as the target for one of that process' segments.

Another situation that turned up had to do with processes requiring a segment for which a fetch had already been initiated on behalf of another process. The original fetch had been for a background batch job, and a process servicing an interactive session (mine) required the segment. The system was busy, and the batch job's fetch request wasn't serviced for an extended period. The priority of the fetch had to be increased to avoid delaying the interactive process for a long time.

Further refinements had minor second order effects on system performance. Optimization within an urgency class for minimizing head movement had a positive effect in some situations, but caused large standard deviations in others due to capture effects. Performing file

transfers before memory management transfers of the same priority and performing requests related to full swap-ins after those requests where only one segment is required were intuitively appealing variations, but the effects were found to be negligible. FIFO servicing within a priority class was found to provide good stability and high performance.

The impact of caching of discs is discussed in detail in the next chapter, and the sharing of discs between systems is discussed in the following chapter. These refinements are not second order. We study there the impact of integrating database requirements with disc scheduling by allowing the specification of posting order constraints, resulting in write-ahead logging without wait.

5.2 Priority Assignment

The principle uncovered from the disc subsystem analysis was that in order to effectively utilize the capacity of a subsystem, the other subsystems must cooperate by generating enough requests to keep the subsystem busy and informing the subsystem of the relative urgency of the service requests so that global performance and not local performance could be optimized.

The priority assignment scheme is critical. This is to be used as the guide for all subsystem scheduling policies. The goal of the priority assignment algorithm is then to assign and adjust priorities to activities so as to reflect their global urgency. Each subsystem

uses the priority assignments as indicators so that local scheduling decisions contribute to the global performance objectives.

Scheduling disciplines have been well researched over the years. Coffman and Denning provide a history scheduling algorithm analysis in [Coff 73]. Optimal and near optimal schedules for various arrival and service demand distributions are analyzed by Coffman and Kleinrock in [Coff 68]. Shaw [Sha 74] surveys schemes for moving priorities in multilevel scheduling algorithms employing dynamic priority assignment and preemption. Ellison [Ell 75] discusses the use of priority improvement as a process' resource demand declines and priority degrading as resources are consumed employed the Utah TENEX scheduler.

A flexible framework for investigating the impact of expressing priority assignment globally throughout the management of system resources was created. It consists of an external command which allows the specification and control of scheduling classes and priority assignment. The parameters consist of a base and a limit priority for each scheduling class and a minimum and a maximum on the filter rate which controls the drift from the base priority to the limit priority within a scheduling class.

When a process is created, it is specified as to which priority scheduling class the process belongs. The process' priority starts at the base of the class and drifts towards the limit of the class as resources are consumed.

This command allows the simple specification of a number of different priority scheduling policies. For example, round robin at a fixed rate can be enforced by setting all the base and limit priorities equal to one fixed priority, and setting the minimum and maximum filter equal to the desired fixed rate. Multilevel queueing disciplines are created within and between service classes through the setting of the class' base and limit priorities and filtering values.

This external control proved useful not only as a research vehicle but also as the system tuning control. It allows the system manager to directly express the system performance objectives. The ongoing priority assignment and adjustment within the system then uses these externally expressed guidelines. Process priority assignment and migration is based on the scheduling class and resource consumption, and all resource service requests are tagged with the priority of the process requiring the service so that the guidelines are reflected in local resource management decisions.

The standard objectives for general purpose, interactive timeshare systems are to maximize the transaction throughput and minimize transaction response time while providing adequate service to jobs.

A good tuning assignment to accomplish these objectives was found to be placing the processes relating to interactive sessions into one class and those related to batch jobs in a separate class. The base and limits of these classes can be tuned to reflect the desired performance objectives. For example, to give preferential treatment to interactive sessions, place the base and the limit priorities of the

interactive class below the base and limit of the batch class. To filter out the long transactions and jobs, make the range from base to limit priorities large, and the filter relatively small. To cause the batch to compete equally with the long transactions, overlap the batch class with the interactive class by placing the base of the batch class near the limit of the interactive class.

It was found to be necessary to reset an interactive process' priority to the class' scheduling base when the related session completes a transaction. Without this feedback, the very short transactions of a session get prolonged due to previous resource demands. This is undesirable from both a system throughput viewpoint and from the viewpoint of the individual session where short transactions are expected to go through fast whereas delays on long transactions are less noticed.

Various resource consumption factors were tried out to determine an optimal policy for filtering a process from the base to the limit priority. Processor time was found to produce the most stable results, and the best performance came when the system dynamically tunes the filter to the processor time required to complete an average short transaction. Other estimates on transaction length such as disc accesses were found to lack stability.

The danger of such a priority policy is that under a very high demand situation, the low priority activities could be held off forever. It was interesting to observe that additional feedback mechanisms to handle such situations were not required. The

interactive workload tends to be so bursty that there are plenty of short intervals for adequate background service of the background activities. To handle the isolated sites in which the workload is constantly heavy, the system can be tuned to overlap the scheduling classes.

A complete description of this tuning command is given in Appendix F. The command allows the tuning of three scheduling classes, C, D, E. The scheduling classes A and B are reserved for system process priority assignment.

5.3 Semaphore Management

Semaphores are non-preemptable resources, and the holding time of a semaphore is variable. The holder of the semaphore is subject to the resource management policies of the system. If a high priority process requires the non-preemptable resource, it must wait until the less urgent process holding the resource gets enough service to release the resource.

As was found in disc access scheduling, some priority adjustment is required so that semaphores are freed up at the priority of the most urgent process waiting for the semaphore. The required priority adjustment is to temporarily improve the priority of the resource holder to the priority of the most urgent process waiting on the resource, and to hold it there until the resource is released.

5.4 Processor Management

Possible activities for processor assignment consist of running ready processes, swapping in processes requiring main memory scheduling, performing various background activities, or pausing. The priority assignment algorithm distinguishes the relative urgency among the pending activities so that processor allocation among them is relatively straight forward.

If two processes are ready to execute, the process with the more urgent priority is given the processor. If a process is ready to execute, but a more urgent process requires memory scheduling, memory management service is initiated for the more urgent process. If a process becomes ready while another process of lower priority is executing, the current process is preempted. If a process becomes ready while memory scheduling is underway for a less urgent process, the memory scheduling is deferred.

Processor allocation becomes complicated when it comes to deciding on whether the processor can safely be applied to swapping in a process. The decision to apply the processor to attempt to increase the multiprogramming level must be integrated with the memory management subsystem. The research experience with alternative load control schemes is related in the next section which deals with main memory management algorithms.

When it is determined that it is not safe to swap in a process, the processor can be applied to background activities. A processor pause

interval distribution is shown in Figure 20. The frequency and duration of pauses depends on the workload characteristics, resource management algorithms and system component capacities. The pause duration and frequency decrease with increasing algorithm parallelism and resource capacity. The mean idle time on a loaded systems ranges from 3 to 20 ms with pauses occurring with a 10 to 50 percent frequency relative to launches in the case study environment.

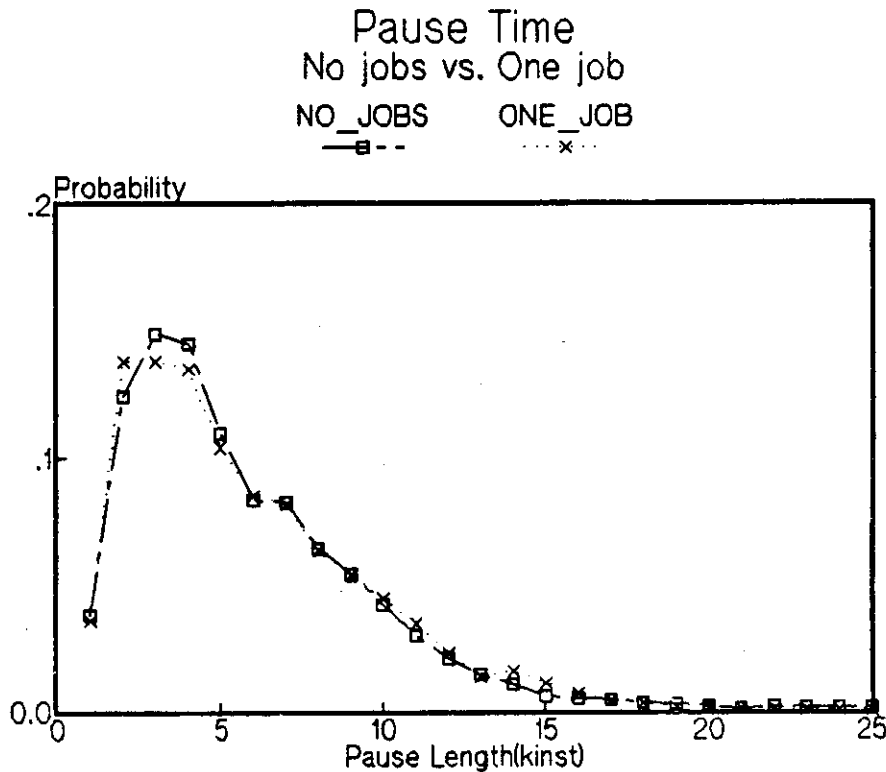
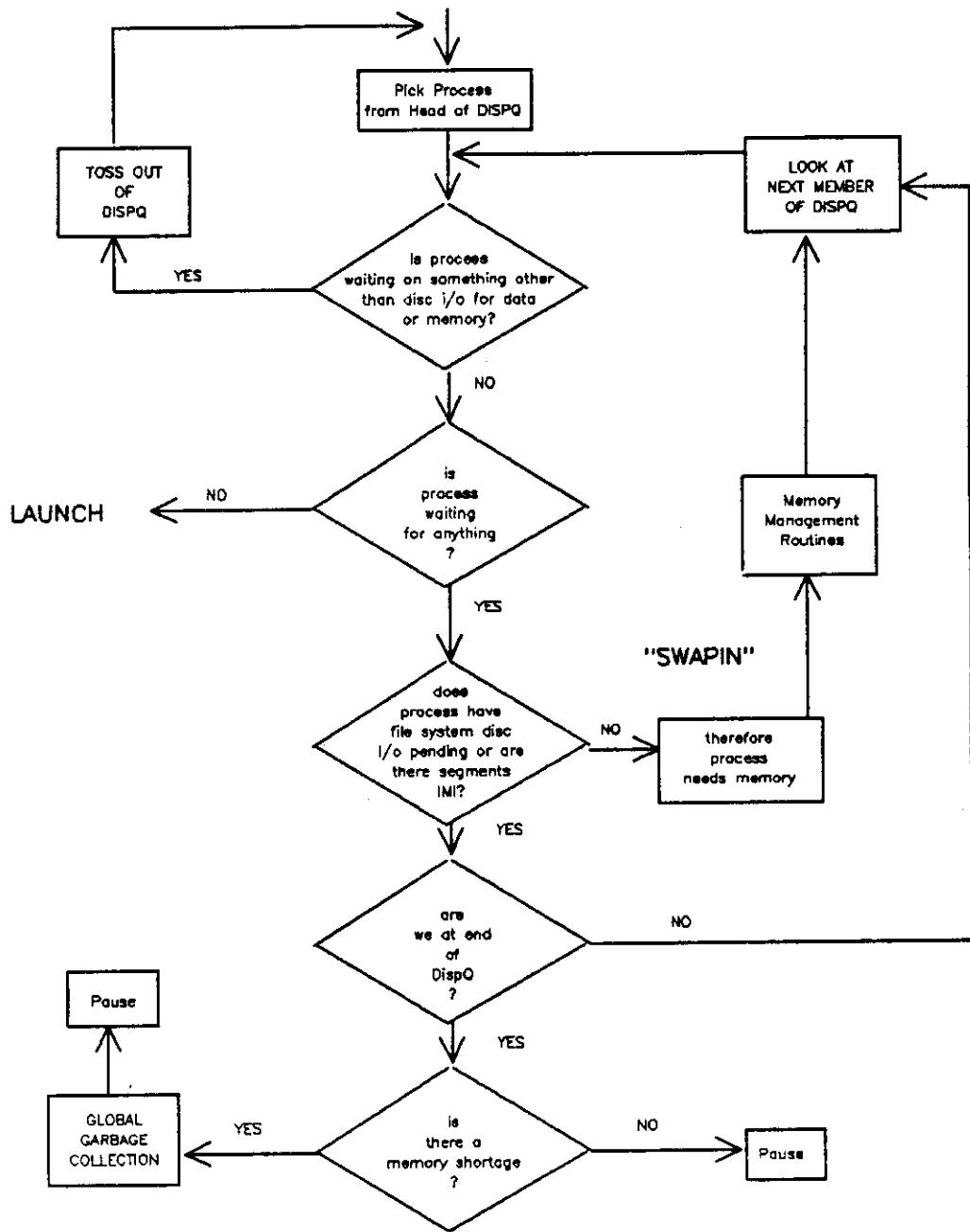


Figure 20: Processor Pause Interval Distribution

A special opportunity for background activity which can be applied in segmented systems to profitably consume these idle periods is main

memory garbage collection (combining holes in memory to overcome external fragmentation). The research experience with background garbage collection is related in the next section.

A flow chart of the processor allocation algorithm is provided in Figure 21.



DISPQ25

Figure 21: Processor Allocation Flow Chart

5.5 Main Memory Management

Main memory management requirements for the bread-board kernel consist of free space allocation, segment replacement, and garbage collection.

Free space allocation is required when a segment fetch is to be performed. The free space allocation algorithm selects the location into which the segment read should be directed.

Segment replacement must be performed when a hole of adequate size for a segment fetch request is not available.

Garbage collection may be required in a segmented system to combine holes into larger holes. A variable sized allocation strategy may produce small, unusable holes scattered about memory. Garbage collection attempts to minimize this fragmentation by combining the small holes into usable large holes.

This section reports on the research into alternative algorithms for these memory management functions. Special focus is placed on the interrelationships between the algorithms, and on the relationships between memory management and other resource management algorithms.

5.5.1 Placement and Garbage Collection

In a segmented system, the placement problem is to select from the available holes one which is at least as large as the size requested. When no holes of adequate size are available, the replacement algorithm is invoked.

Randell [Ran 68] analyzes internal and external fragmentation resulting from rounding up requests for storage to reduce the number of different size blocks in memory. He concludes that as the quantum of allocation is increased, there is more loss of memory utilization due to internal fragmentation than is saved through the reduction in external fragmentation.

Shore [Sho 75] analyzes the external storage fragmentation resulting from the first-fit and best-fit allocation algorithms. His simulation studies indicate that the relative performance of the two algorithms depends on the frequency of requests that are large compared to the frequency of requests that are small. He finds that first fit outperforms best fit as measured by storage utilization when the coefficient of variation of the request distribution is greater than one.

The buddy algorithm (hole sizes restricted to 2^k for $k=0,1,\dots$) was introduced by Knowlton in [Kno 65] and analyzed statistically by Purdom and Stigler in [Pur 70]. Variations on the basic algorithm are discussed by Shen and Peterson [She 74] for a weighted buddy scheme which decreases the internal fragmentation by allowing hole sizes of 3

* 2^k as well as 2^{k+1} , and for Fibonacci buddy systems (holes split into sizes which are Fibonacci numbers rather than powers of two) are analyzed by Hirschberg [Hir 73] and Cranston and Thomas [Cra 75].

Results of an extensive simulation study on alternative placement strategies are reported by Shore in [Sho 77]. Memory ordered free list, LIFO/FIFO free list, multiple free lists with a list for each possible hole size, buddy lists, and size ordered free list schemes are evaluated based on their data structure overhead, allocation and deallocation time, and resulting internal and external fragmentation.

Buddy schemes are shown to be very fast for allocation, and relatively fast for deallocation with some of the improved recombination schemes suggested in the literature. However, the internal fragmentation resulting from the buddy schemes is large, fully committing memory while containing only half the segments of some of the other schemes.

The data structure overhead of the multiple free list strategy is the largest, but it provides improved memory utilization and comparable allocation and deallocation times.

Size ordered free lists save on the data structure overhead but increase allocation time ten-fold.

Compaction while allocating or deallocating improves memory utilization but increases allocation/deallocation time.

The topic of placement algorithms still has current interest, with fairly sophisticated systems being built with microprocessors having segmented architectures (e.g. Apple's Lisa using the Motorola 68000). A paper describing a storage allocation algorithm using a Cartesian tree [Ste 83] which asserts space performance of the first fit and time performance close to buddy made it into the most recent SIGOPS.

First fit, best fit and size ordered free list strategies with memory compaction at allocation, deallocation and as a background activity were implemented in the bread-board kernel in order to discover the the significance of their tradeoffs and their interactions with the rest of the system.

An immediate implementation observation was that it helped to know the size of the current largest hole. This allows the replacement algorithm to be triggered without having to go through a costly scan with possible compactions along the way. The first fit algorithm is unfriendly in the cost of maintaining this information. Best fit based on size ordered free lists and multiple free list strategies allow this piece of information to be naturally kept around. But this is not a deciding factor.

The impact of the allocation unit size used for rounding is shown in Figure 22. In agreement with Randell's findings, the smaller allocation units reduce the amount of memory management traffic required to perform a workload significantly. As the allocation unit size increases, the number of memory allocations and segment transfers per transaction increases rapidly.

Effects of Allocation Unit

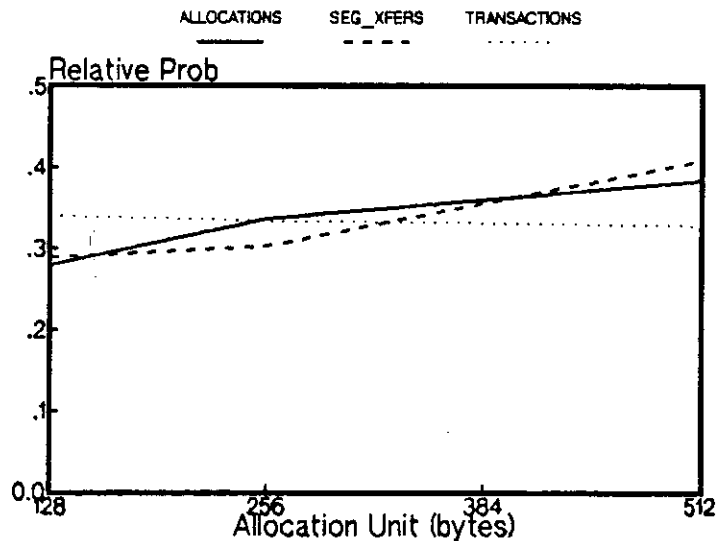


Figure 22: Impact of Allocation Unit Size

Allocation time scales with processor speed, and processor speeds are increasing more rapidly than disc access times are reducing. Thus, current technology tradeoffs favor the benefits of reduced internal fragmentation offered by refined hole sizes over the allocation time improvement of large allocation units. Allocation time increases with a size ordered free list as memory size increases since the population of holes increases. Thus, multiple free list strategies with a reasonably small allocation unit best support current technology tradeoffs.

A new algorithm for garbage collection was developed to exploit the pause interval distribution shown in the last section. The algorithm is designed to run distributed over time, so that it can give up when

more urgent activity becomes pending. In order to provide the most benefit in the shortest time, the algorithm operates globally with the objective of increasing the size of the larger holes.

The garbage collection algorithm is depicted in Figure 23. The algorithm begins at the largest hole and looks above and below it for the closest holes. The intervening assigned regions are moved to small holes unless the move length would be too long (either committing the system to garbage collection too long or consume an already sufficiently large hole).

If it is decided that collecting around the current hole would not pay off, the next largest hole is checked. Any time a combination is performed, the algorithm starts over. The algorithm periodically checks for a message from the scheduler that a more urgent activity has become pending. Polling is used rather than straight preemption since the garbage collector would leave memory in a bad state if suddenly preempted.

GARBAGE COLLECTION

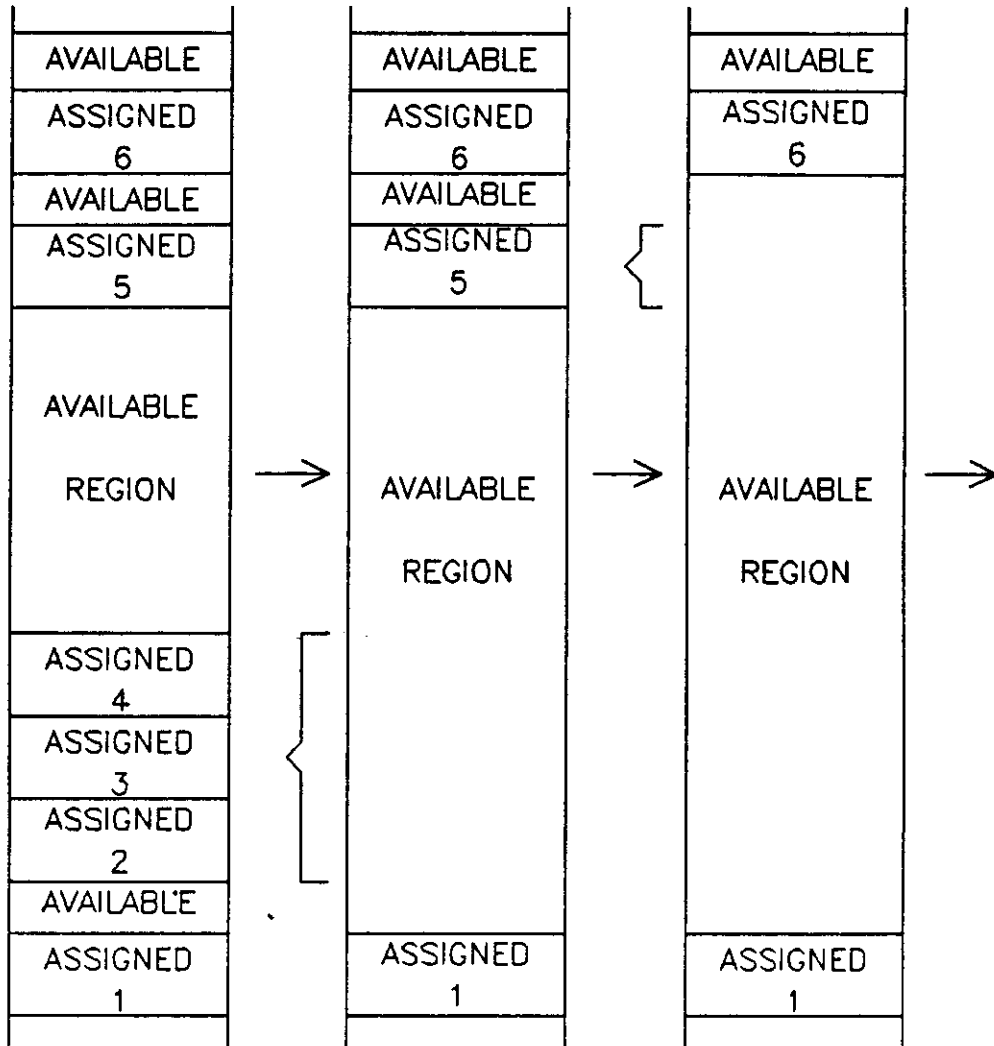


Figure 23: Global Garbage Collection

This algorithm has the effect of gradually making the large holes larger while covering the small holes, all during otherwise idle time. The algorithm is different from algorithms described in the literature in that it is global in nature and distributed in time. It takes advantage of the otherwise idle processor cycles rather than combining holes locally at allocation or deallocation time or all at once as is conventionally done.

The resulting equilibrium distribution of free hole sizes is shown in Figure 24. The distribution with background garbage collection keeps the distribution from being clustered at the very small hole sizes, which is what happens without garbage collection. The shift in the distribution of hole sizes due to background garbage collection reduces the probability that the replacement algorithm must be invoked, and does it at near-zero cost by using otherwise idle periods.

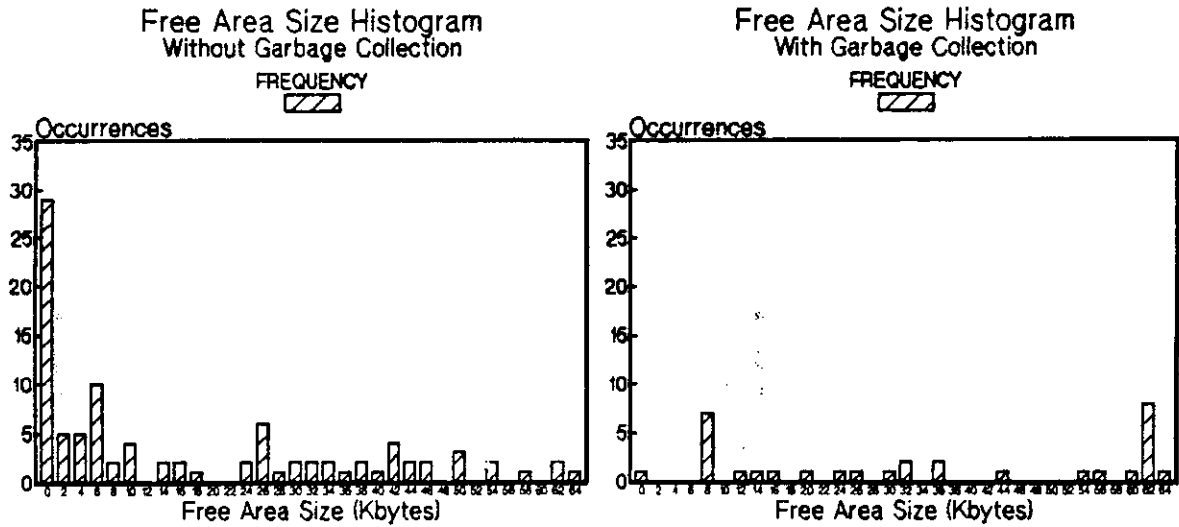


Figure 24: Garbage Collection Impact on Hole Distribution

The background garbage collection algorithm benefits from free space allocation algorithms which keep track of holes on a size ordered basis, since they make it easy for the algorithm to locate the largest holes. Fast allocation and deallocation are also useful. Thus the multiple free list strategy supports background garbage collection the best.

This background garbage collection algorithm was found to improve performance when good replacement algorithms were used, but to degrade

performance with poor replacement algorithms due to the consumption of mistakenly chosen overlay candidates.

5.5.2 Main Memory Replacement and Load Control

Memory replacement algorithms have been given a lot of attention in the literature.

Denning introduced the working set model for program behavior in [Den 80]. Ever since then, Denning has been trying to prove that the working set concept is not just a good model of programming behavior but that it can be used for memory management to obtain near optimal multiprogramming.

The working set of a process consists of those pages referenced by that process during its last T instructions, where T is the working set parameter. In working set memory management, a process' working set must be in memory for it to be launched. The current multiprogramming set consists of the processes which have their working sets in memory.

Working set memory management is based on explicit selection of the current multiprogramming set. The number of the processes to allow into the current multiprogramming set (the multiprogramming level) depends on the size of memory and the size of the working sets. Controls are needed to decide when to increase the multiprogramming level by swapping in a process requiring service and when to reduce the multiprogramming level by swapping out a process.

Badel, Gelenbe and Leroudier [Bad 75] examine the maximization of system throughput by the regulation of the degree of multiprogramming. They discuss the structure and implementation of an adaptive scheduler consisting of an estimator and optimizer. The estimator is invoked by a watch dog timer, and measures traffic patterns. The optimizer makes a decision on adjustments to the multiprogramming set at a certain frequency based on information provided by the estimator. The frequency of invocation of the optimizer determines the sensitivity of the control mechanism.

Leroudier and Potier [Ler 76] present an adaptive scheduler based on paging device utilization. They observe the principle that optimum performance is achieved the access capacity to secondary store is 50% utilized, and that at the optimum point the sensitivity of the degree of multiprogramming to the utilization of secondary store is at a minimum. Their adaptive scheduler uses the utilization of the paging drum as the estimate, and the optimizer adjusts the level of multiprogramming up or down if the current utilization is more than a delta off from 50%.

Denning [Den 76] discusses the use of the L (lifetime=inter-page-fault time) = S (paging device service time) as the criterion for the optimizer, with the estimator observing the last k program lifetimes and averaging them to get the system lifetime. Their simulations show that this produces a near optimal multiprogramming control as long the system is neither I/O nor processor bound.

Chu and Opderbeck introduced the PFF (page fault frequency) algorithm in [Chu 72]. They provide simulation results which compares the PFF algorithm with the LRU and working set algorithms in a uniprogramming environment, based on fault frequency and space-time product measures.

The page fault frequency algorithm releases pages from a process' working set only if the time since the last fault exceeds T instructions, with T being the control parameter of this policy. The algorithm is triggered at page fault time. If the process' virtual time since last fault exceeds T instructions, those pages are released which were not referenced since the last page fault.

Chu and Opderbeck found that the performance of the PFF algorithm is comparable to that of the working set algorithm, but that performance with PFF algorithm is less sensitive to its control value than is the working set.

They continue their analysis of the PFF algorithm in [Chu 76] using a Semi-Markov model to represent the replacement algorithm and lru stack depth distributions to represent process referencing. In [Chu 76b] they discuss multiprogramming level control policies with the PFF algorithm. This mechanism requires that to increase the multiprogramming level there must be a pool of free pages of at least a critical number and the most recently activated process must have received a minimum of service.

Ryan and Coffman [Rya 74] use a semi-Markov process to represent main storage utilization and a queueing network model to capture the

costs of dispatching, paging, and swapping. They conclude that the swapping rate is sensitive to the margin of free storage, but the average multiprogramming level is not.

Chow and Chiu [76] employ similar modelling techniques and conclude that the working set window should be adjusted to adapt to the changing need of storage demand. If the paging rate is low, shrink the window size so that the multiprogramming level can be increased. If the paging rate is high, increase the working set window size so that the multiprogramming level will be reduced.

Smith [Smi 76] proposes a modified working set algorithm which detects major locality phase transitions. The working set size is increased on a page fault only if the time since the last reference to the least recently used page in the working set is less than some fraction of T . Otherwise the new page replaces the least recently referenced page in the working set. This then keeps a process from requiring excessive memory during locality transitions.

Another class of variable partition algorithms is the global LRU type. The global LRU replacement algorithm releases the page from memory that hasn't been referenced for the longest time when a replacement is required.

Maintaining an exact LRU stack of referenced pages would require an update on each reference, so approximations are found in real implementations. Some approximations are made with an aging bit map associated each page. The bit map for each page is periodically

shifted, the current value of the page's reference bit is rolled in, and the reference bit is cleared. When a page replacement is required, the bitmaps are compared and a page with the reference bit off for the longest string in the bitmap is selected for release. The width of the bitmaps and the frequency of updating control the accuracy of this approximation to the pure global LRU.

Another approximate implementation of the global LRU replacement algorithm is the Multics CLOCK algorithm [Hab 75]. This algorithm maintains a pointer that it uses to cycle through the reference bits when a page replacement is required. The reference bit of the page with the pointer at it is turned off. If the bit was on, the page is skipped (indicating the page has been referenced since the last time checked). If the bit was off, the page is selected for replacement. In some variations, the page is skipped if it is a dirty data page even though the reference bit was off so that the fetch for the new page need not wait for the post of the dirty page to complete. The decision on whether to increase the multiprogramming level is based on the rotation rate of the "CLOCK arm" cycling among the reference bits. If the CLOCK arm is going too fast, the replacement is held off for an implementation dependent amount of time.

Denning writes in [Den 80] :

there is, unfortunately, little published performance data on the CLOCK and global LRU obtained from real systems. ... The evidence suggests that CLOCK and LRU do not perform as well as working set. This is because these global policies cannot insure that the block of memory allocated to a program minimizes that program's space time. ... It is false economy to limit the

hardware support for memory management to usage bits and interval timers, for the savings in hardware are canceled by performance losses (relative to the working set dispatcher) or by additional mechanism elsewhere in the operating system.

Having carefully studied this and other literature, We built PFF, working set and CLOCK replacement algorithms into the bread-board kernel.

The working set maintenance is performed on a per process basis by use of a segment locality list. The list is updated after every process stop by checking the reference bits of the segments the process could have referenced. If the reference bit is found on, an entry for the segment gets inserted into the process' locality list if not already there, and the current process' virtual time is recorded in the entry. Thus, the entry contains the approximate virtual time of the last reference made by the process to the segment.

A microcoded search and map instruction is used to speed up the reference bit check. It terminates only for a segment which has the reference bit set.

After the list has been updated, elements in the list which have last referenced times differing by more than T ms from the process' current virtual time are trimmed from the list.

In order to keep from releasing a segment from memory that leaves the working set of one process but still resides in the working set of another process, a counter is associated with each segment which counts the number of processes in the current multiprogramming set which have

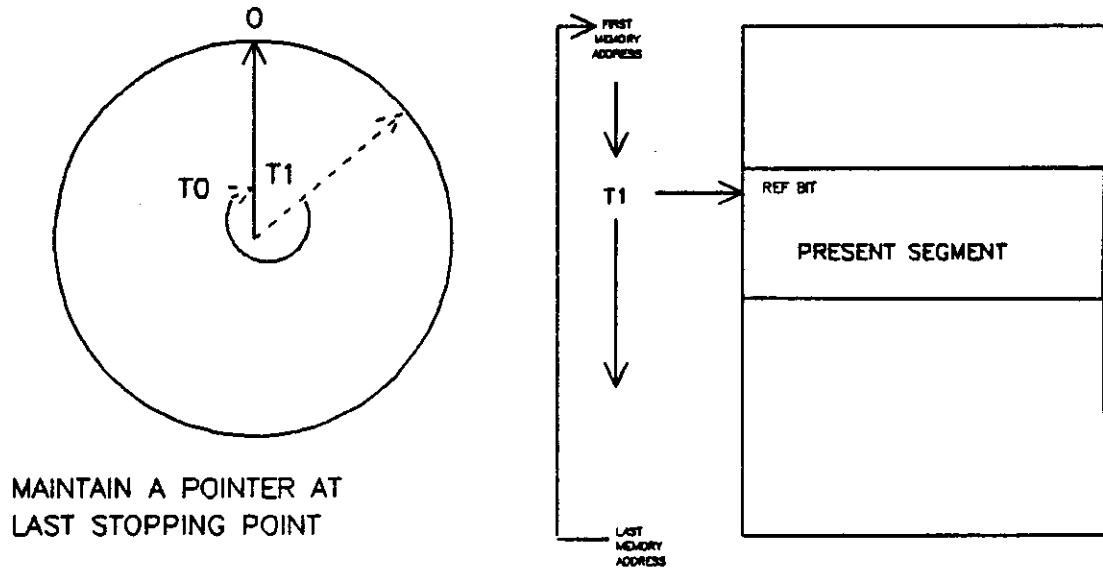
the segment in their locality lists. When a segment is tossed from the locality list of a process the counter is decremented. The segment is released to the overlay pool only if the counter falls to zero. When a process is swapped out, the counters of the segments in its locality list are decremented, and those that fall to zero are released.

When a segment is released to the free space pool, it is combined with any adjacent free regions and the new larger free region is placed on the free space list structure for the placement algorithm. The segment is not lost track of though. If the segment is referenced again before it is actually overlayed, the free region of which it is a part is split, the sub-region occupied by the segment gets marked assigned, and the segment flagged present.

The PFF algorithm is implemented using the same locality list structure, but a segment is added to the locality list only on a segment fault. Elements are trimmed from the list at fault time if their time since last reference exceeds the PFF critical inter-fault time. Sharing is not accounted for.

The implementation of the CLOCK algorithm is depicted in Figure 25. The algorithm is implemented by advancing the CLOCK pointer sequentially through memory from memory region to region, skipping free and reserved region, and turning off the reference bits of segments in assigned regions. When the end of memory is hit, the scan wraps around to the beginning.

CLOCK ALGORITHM



610207R

Figure 25: CLOCK Algorithm in Segmented System

The working set algorithm maintenance required four times as long to save state as did the PFF and CLOCK algorithms. The working set specific code for tracking sharing and all of its ramifications

required 1500 extra lines of code. The implementation of the CLOCK and PFF algorithms each required less than 100 lines in total.

The behavior of the working set algorithm with respect to the working set window size is shown in Figure 26. Notice that the fault rates drop rapidly as the working set window size increases, but the processor utilization and transaction rates peak as the memory management traffic required to support the larger working sets takes its toll. Although large values of the working set window size result in low fault rates, they force a low level of multiprogramming and require excessive memory management disc activity to swap in and out the larger working sets.

Effects of Working Set Window Size

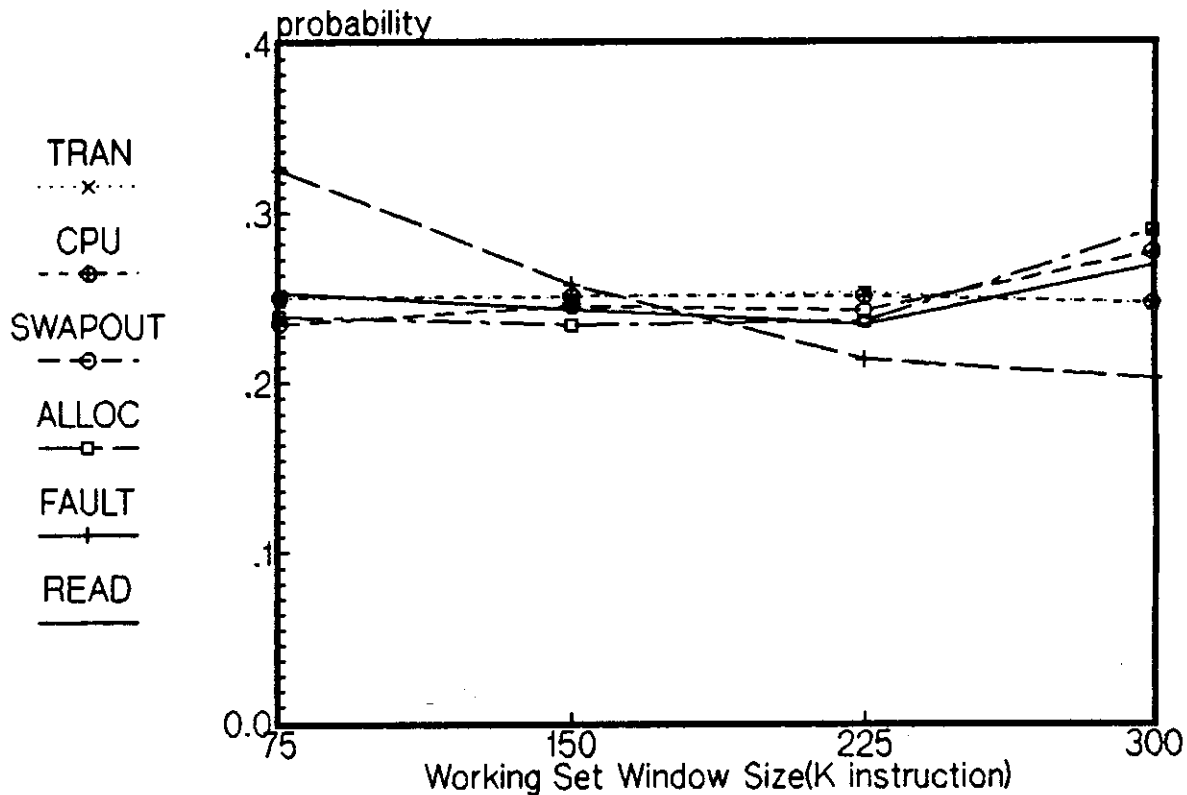


Figure 26: Effects Of Working Set Window Size

We tried out many variations with the working set memory management algorithm. Several things helped, each on the order of 2%.

Two space utilization enhancements were : don't swap out the process' entire working set when deleting a process from the current multiprogramming set, only swap out the space you need; and perform local compaction as you release a segment to get the advantages of localized hole recombinations.

One surprising enhancement was that when adding a process to the current multiprogramming set, just swap in its stack, current extra data segment, and current code segment, rather than its entire working set. A scatter read facility would have probably eliminated this benefit, but locality set swapping to a contiguous place on the disc is hard to implement when data is shared. Allocating a disc swap area on the fly, remapping the objects to point to their new homes, and invalidating the swapped copy when it gets referenced by another process is messy.

Another enhancement was to place explicit load control mechanisms on the swapping. One mechanism that helped was to delay swapping out a process when swapping in a process if there is currently a higher priority process waiting for its scheduled disc activity to complete. This last feature of holding off was needed since the memory manager is a highly parallel server, with the capability of releasing segments, initiating writes, allocating space, and initiating reads of many segments in a short time (9< k inst / memory allocation request total). Load control mechanisms were required to keep things from getting out of hand. Delaying on the order of 50 ms was helpful, but delays over 100ms degraded performance, except at small working set window sizes. We'll see in the next chapter that even in large memories the parallelism of the memory manager pays off in managing cached write posting.

The working set algorithm benefits significantly from both background garbage collection and from local compaction during deallocation. This

disc drives. They quote access times of 4 ms on a hit, with up to an 85% hit ratio. It is packaged with 4 RAM cards and an LSI-2 I/O processor.

The use of external caching, either locally or through an intermediate storage level, reduces the effective secondary store access time on read hits, and potentially decreases the access time on writes, provided immediate physical update of the disc media is not required on a write access before signaling transfer completion to the processor. The performance impact of external disc caching as a function of processor speed, disc access time, and number of discs is shown in the following two figures with and without waiting for write posting to the disc media.

Without waiting for write posts, effective processor utilization is achieved with 4 25 ms discs through 8 MIPs, and 10 ms discs and the dependency on disc access time and number of discs is significantly reduced over the system without disc caching.

If all writes must be waited on, effective processor utilization is limited to 4 MIPs with 25 ms discs and 8 MIPs with 10 ms discs. The dependency on disc access time and number of discs becomes much more significant.

Replacement Algorithm Invocation Frequency

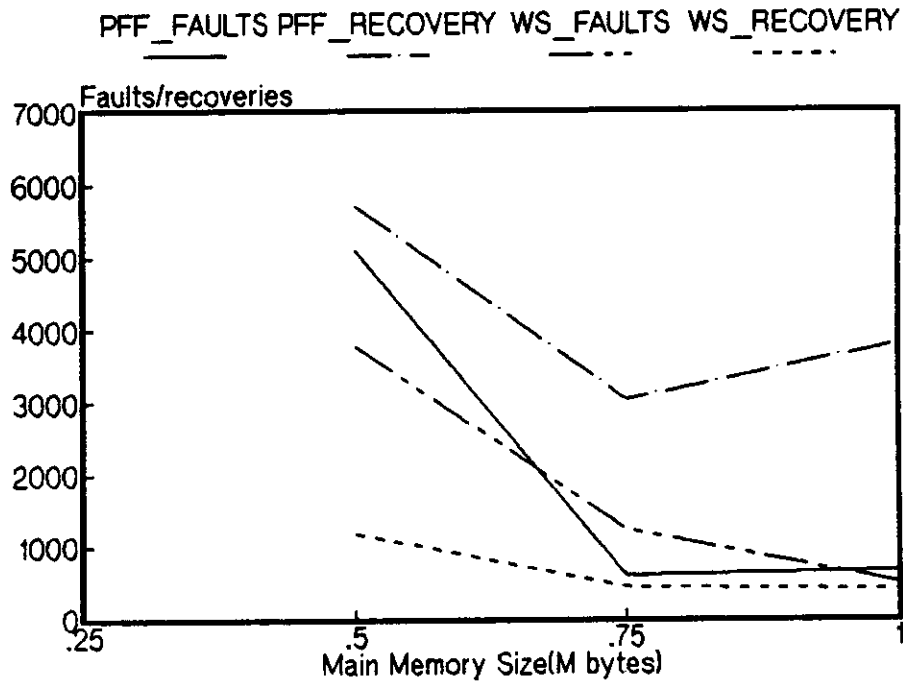


Figure 27: PFF vs WS With Moderate Load

This high recovery rate makes it highly counterproductive to do background garbage collection with the PFF. As the garbage collection move threshold (maximum size of assigned space it will move) increases, the number of moves it performs increases, and therefore the number of overlays of replacement candidates increases. The background garbage collection eats up overlay candidates so that the PFF can't recover them and must go to the disc to fetch them in again. With the working set algorithm, since sharing is accounted for, the replacement candidates really are not needed. The combining of the scattered holes

reliance on garbage collection is caused by the opening up of holes randomly in memory when a process is swapped out since the segments in a process locality list are collected based on the process' referencing behavior and not for the convenience of the placement algorithm.

The PFF algorithm performs better than working set under light memory loading. This is because the working set algorithm releases segments from a locality list even though there is no memory pressure, and they do get referenced again after some time. The additional process state save overhead was costing too much, and buying nothing.

Figure 27 indicates the effect of the PFF's failing to account for sharing. The load generator was creating a moderate load. The test was rerun at increasing memory sizes, once with PFF and once with working set. Notice the higher fault rate and very high recovery rate of PFF, even in the larger memory sizes. Since a segment is only in the locality list of the process that trapped on it, and most of the code segments and a fair percentage of the data segments are heavily shared, the segments get recovered immediately when released from the locality list of a process. Thus under moderate loading the PFF algorithm is doing work for nothing.

replacement, resulting in more combinations of adjacent holes so more rapid creation of a hole of combinations of adjacent holes so more rapid creation of a hole of adequate size with fewer replacement required. This effect is further enhanced through local compaction at deallocation time, plus background garbage collection accomplishes more with less work. The swapping out of a working set doesn't create opportunities for such a high degree of clustered recombinations. The allocation time of clock is about 30% longer than working set due to the moves in combining, but fewer deallocations, and so eventual allocations, are required.

Figure 28 compares the CLOCK, PFF and working set (WS) algorithms in action with a moderate load. Although the fault rates of CLOCK are twice those of the working set, the number of space allocations and segment transfers incurred by CLOCK is less even though it performs more transactions.

Both CLOCK and WS outperform PFF in all measures. PFF is getting hurt due to failure to account for sharing in a workload that shares heavily.

created by the random locations of segments selected for replacement during otherwise idle periods pays off.

It should be noted that a PFF algorithm taking sharing into account using the mechanism built for the working set algorithm support produced behavior analogous to the working algorithm. The PFF behavior cited here is primarily for the comparison of algorithms that take sharing into account and those that don't.

The best policy for light or moderate loading of main memory was found to be the Multics CLOCK algorithm. It has the low cost of the PFF and accounts for sharing as in the working set, but implicitly.

The CLOCK algorithm has zero cost until a replacement is required, so in moderate and light loading memory management overhead doesn't get in the way. When the algorithm is invoked, it flips from one segment to the next in memory looking for a segment which has its reference bit off. As segments are encountered with reference bits on, they are skipped and their reference bits cleared. When a segment is found with reference bit off, the segment is selected for replacement, flagged absent, and the region it covers combined with adjacent holes and returned to the free space list. The algorithm is extremely simple to implement, and its execution is very fast.

In a segmented system, with the CLOCK algorithm implemented as here based on memory order, the CLOCK algorithm has a very important advantage over working set type policies. Unlike working set replacements, it tends to select segments close to each other for

the CLOCK during non-bursty intervals, as well as getting the benefits of CLOCK's localized replacements and recombinations,

This new algorithm measures the memory pressure based on the time since last invocation of the replacement algorithm. When this interval is large, the simple CLOCK replacement algorithm is used. When the interval is small (indicating frequent invocation of the replacement algorithm and therefore heavy memory loading) an interesting combination of CLOCK and working set principles is used.

When memory pressure is detected, during a process state save the segments which obviously belong to its locality are added to a mini working set for the process. This includes the segments required to relaunch the process (registers point at them), and the next couple of code segments the process will need (found by checking its last stack markers). Every T ms of virtual time the mini working set is chopped back to its minimum so that it doesn't grow too large. The maintenance of this mini-working set costs only a fraction of an accurate working set accounting procedure.

As the processor dispatcher scans its priority ordered queue of pending activities, instead of just skipping over a process which is waiting for a disc access to complete, it first sets the reference bits of the segments in that process' mini working set. This prevents the CLOCK algorithm from consuming the locality of a more urgent process in the multiprogramming set when it is invoked to swap in a less urgent process.

Replacement Algorithm Comparisons

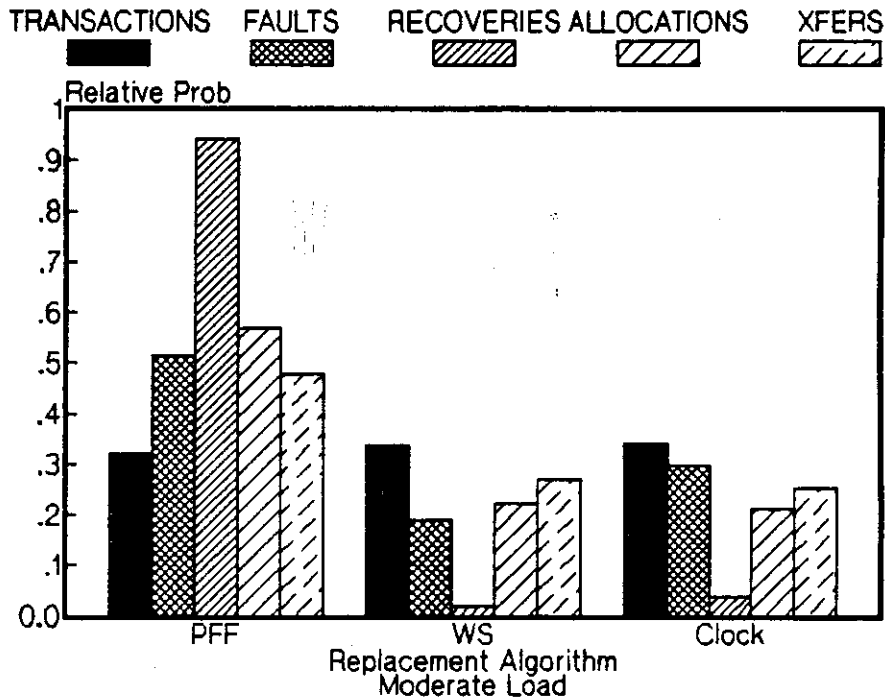
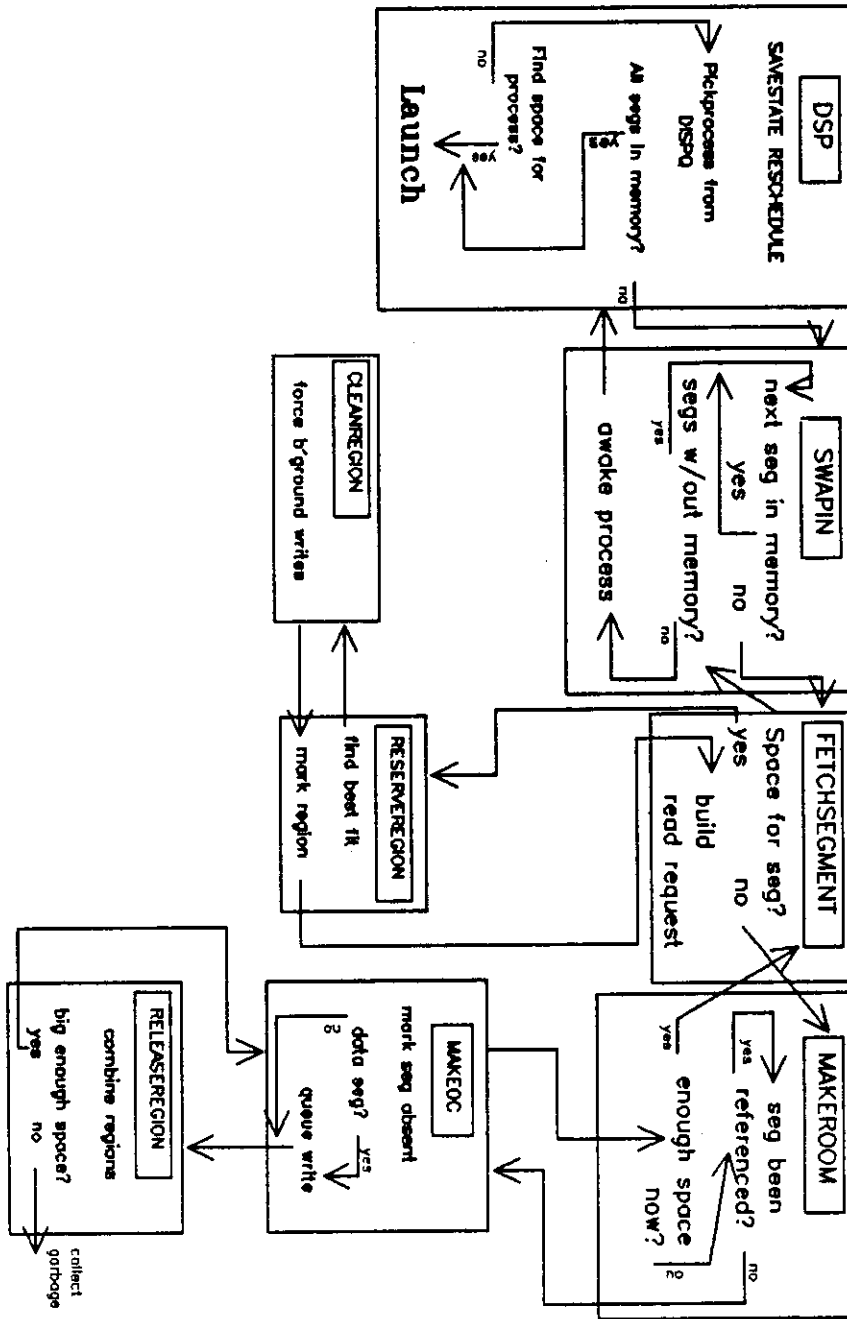


Figure 28: Comparison of CLOCK, WS, PFF in Action

A tuned per process working set policy taking sharing into account performed the best among the various replacement algorithms under heavy loading. It is a stable algorithm. The CLOCK algorithm was found to degrade under heavy loading. The fault rate grew to several times that of the working set policies, and it was difficult to control by limiting the CLOCK scan rate.

A hybrid algorithm was developed which incorporates the stability of the working set approach under heavy loading with the low overhead of

CPU/MEMORY/DISC MANAGEMENT



9004590

Figure 29: Bread-Board CPU, Memory and Disc Management

Carr and Hennessy [Car 81] independently developed a hybrid CLOCK and working set algorithm which they call WSClock. Their version uses the clock scanning to apply the working set policy by comparing the current virtual time of a page pointed at by the clock hand with the last virtual time of reference of the page by the process that owns the page. This approach requires that a unique association exists between a page in memory and a process, so sharing is not accounted for. This loses the advantage of the CLOCK algorithms implicit accounting of sharing, but it does protect the memory locality of the processes in the multiprogramming set. The same algorithm is applied independent of memory loading.

An overview of the interactions between processor, main memory, and disc management in the bread-board kernel is given in Figure 29. The resulting production kernel is described in [Bus 82].

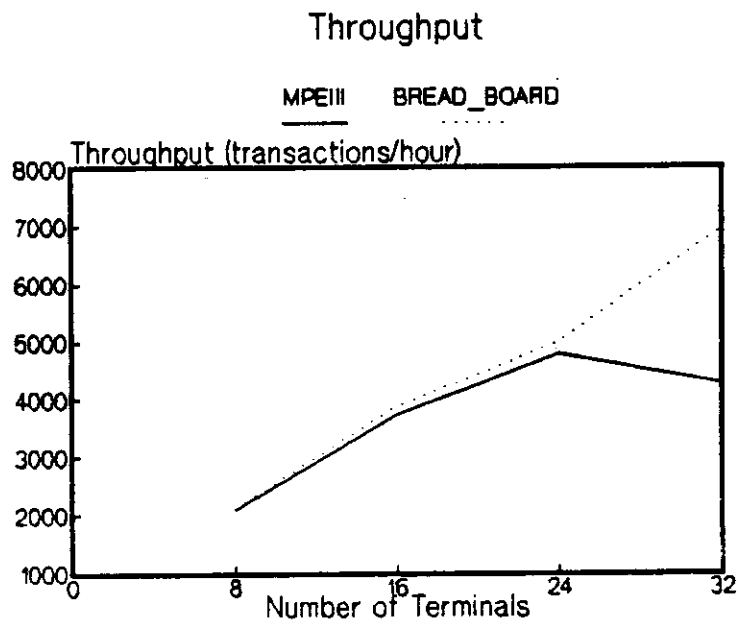
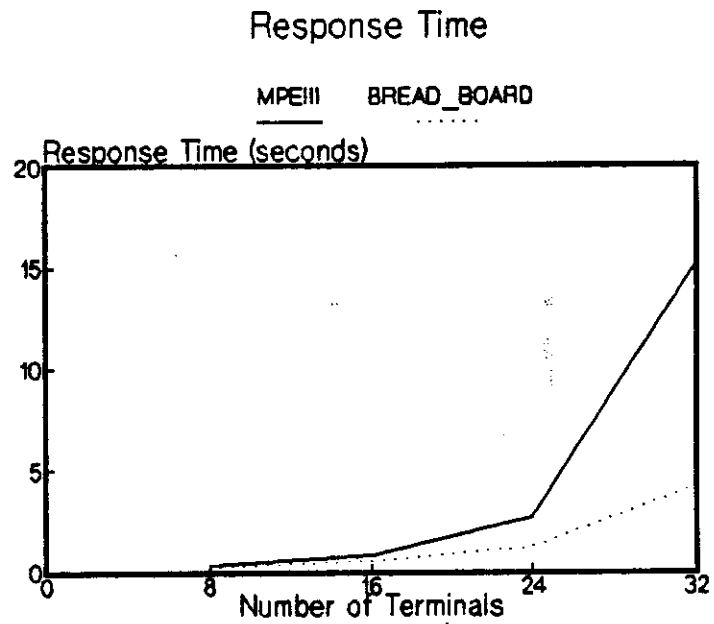


Figure 30: Performance Impact Due to Integrated Algorithms

With the relatively low cost of main memory in relation to total system cost (1 Mbyte < 1 % of system cost), and the relatively small

The resulting hybrid replacement algorithm has the stability and performance of the best working set algorithms under heavy loading, and the low overhead of the CLOCK under moderate and light loading. Figure 30 shows the system performance improvements realized by this algorithm coupled with the other integrated algorithms presented in this chapter relative to the then-current production operating system for the case study family. Response time reductions to a quarter and throughput doubling was found across all family members in the system configurations then supported.

It is interesting to note that DEC's VAX family doesn't even support reference bits. This was an overkill probably, and operating systems designers had to work around it. Babaoglu and Joy [Bab 81] had to emulate the reference bits by sampling so that memory replacement in a system with sharing (Unix) would operate efficiently.

The trend though is towards less in hardware to make it faster, easier to build, and amenable to VLSI implementations. Some RISC machines [Fit 81] even leave the TLB fault handling to software since it is a low frequency event.

In the next chapter we will see that large memories can be efficiently exploited by extensive file and database caching through the memory manager. Hardware support for working sets in this environment would not pay off, since the inter-reference times of cached disc domains are on the order of seconds while those of transient objects are on the order of ms. Thus, a working set window choice that would work well would be difficult to obtain, and a low cost policy that favors the transient objects (which occupy a small total space in a large memory) is just fine.

5.6 Conclusions

In order to address contention by adding capacity, the capacity needs to be utilized by generating service demand, and the service demand needs to be kept balanced across the servers. This was seen especially with disc management requiring parallelism and balance in

space requirements for code and data objects (100 Kbytes for 1 user, but highly sub-linear with respect to number of users due to high degree of sharing), it is cost effective to provide enough primary memory to keep the replacement frequency of transient objects low.

The justifiable overhead to be incurred for supporting a replacement algorithm is small under such situations since the frequency of invocation is normally relatively slow. We don't want to spend a lot to perform a perfect job of memory management as measured by space-time products or fault rates under such stable situations. However, highly bursty periods still occur during which the memory capacity is too small to handle the load at the low stable level, and these situations must be adequately handled.

Denning argues in [Den 80] :

Recent technological advances make working set detecting hardware even more attractive. Such hardware would simplify the operating system and reduce the overheads of job scheduling and memory management; it would do this by replacing a considerable amount of mechanism that would otherwise be in the operating system software.

It is questionable whether such hardware support for memory management is justified from economic or performance perspectives. The hardware likely slows down the processor instruction rate, adds to the hardware complexity and cost, and would be present to support a relatively infrequent event (replacement of transient objects in large memories). The cost of the hardware development would be incurred each time a new family member is added, whereas the memory management code is developed just once, and is easily fixed and enhanced.

Idle periods can be applied to constructive activity that would otherwise have to be done during busy periods. Background garbage collection was found to be useful in segmented systems, provided the replacement algorithm was doing a good job in selecting overlay candidates.

Algorithms for resource management can be integrated by studying their interactions and maximizing those effects that contribute most towards system performance. This was found in the study of disc access scheduling.

Algorithm integration impacts the structure and local goals of algorithms, requiring them to communicate and adjust in an on-going manner, and requiring them to perform actions which do not optimize local goals. For example, disc access scheduling needs to support priority changing, request cancellation, and performing disc accesses which are not the closest to the current head position. Memory management and garbage collection need to give up without completing when more urgent activities become pending.

Assign urgency globally to reflect system objectives, and reflect locally. We found this to be a useful policy in integrating all local decisions towards the global goal.

Disc scheduling, memory replacement, memory allocation, garbage collection, processor and semaphore management need to be integrated, not just replacement and multiprogramming set control.

the memory management implementation in order to utilize available capacity.

Don't locally optimize blindly. It can cause more damage than it does good. This was found to be the case with disc scheduling based on minimizing head movement and with memory management based on minimizing the space-time product.

Go with cheap algorithms if the resource demand is not too heavy. Support hybrid algorithms to handle bursty situations. This was found with replacement algorithms, where low cost algorithms performed very well under light memory loading, while more expensive algorithms were beneficial under heavy loading.

Algorithms need to take the workload characteristics into account. This was especially found with the replacement algorithms, where performance was degraded significantly when sharing was not accounted for.

CLOCK type algorithms have a distinct advantage over working set algorithms in a segmented system due to localized hole creations. Local garbage collection at replacement time supplements this. The hybrid replacement algorithm introduced here has the advantages of working set (protection of urgent processes even under loading) along with the advantages of clock (easy implementation, low overhead, localized hole creation).

data is kept in structured permanent and temporary objects including files, databases, stacks and heaps. Concurrent data access is managed through locking or versioning. Recovery from transaction aborts or system failures is handled through check-pointing, write-ahead logging, and roll-forward/roll-back recovery.

In order to exploit the system price/performance potential offered by evolving technology, basic kernel management of main memory, disc, and processor management needs to be extended and integrated with subsystem and application data and recovery management. This was well demonstrated in our research with the bread-board system and the case study family.

After having integrated the management strategies for the main memory, disc, processor and semaphore resources to produce balanced resource utilization and significant performance improvement across the family, a new high-end system was introduced. The processor was twice as fast and memory 4 times as large as the previous top of the line. The system performance improvements realized with this computer were, however, sub-linear with respect to processor speed relative to the previous top of the line system. System performance with this high-end computer was found to be very sensitive to disc subsystem throughput and access times, but relatively insensitive to main memory capacity. In spite of the integrated strategies of the bread-board kernel, the system performance was not scaling as the family extended to significantly more powerful systems.

Chapter 6

Integrating Data Management

The price/performance of processors and cost of semiconductor memory have been falling rapidly, while moving head discs are getting denser but not much faster. This chapter examines alternatives to exploit these technology trends to provide significant system level price/performance improvements. Alternative architectures and integrated data management and kernel algorithms are examined. Results from modelling and a bread-board implementation are presented. Applicability to decentralized configurations are discussed in the next chapter.

6.1 Introduction

In the preceding chapter we examined alternative algorithms for processor, main memory, disc, and semaphore management. We endeavored to understand interactions between algorithms managing these basic system resources, and to determine an algorithm set for these resources which provides good performance through algorithm integration.

Above the management of these basic system resources, subsystems and applications manage data to provide extended system functionality. The

and the interactions between disc cache management and other system resource management. System level measurements of the bread-board internal disc cache and integrated data management/kernel algorithms demonstrate the projections of the modelling and the potential of the integrated approach to exploit current trends in processor, disc and semiconductor memory technology to significantly improve system level price/performance.

The chapter begins with an overview of memory hierarchies to provide background and perspective. The need for memory / disc balancing is then discussed in light of evolving technology.

This is followed by a discussion of alternative balancing approaches. The approaches considered include external caching of disc storage through disc buffers or an intermediate storage level, and internal caching of discs in the primary store through local or global file buffering, explicit internal caching, or file mapping. Methods of improved database concurrency control are surveyed, and limitations imposed by transaction, recovery schemes are discussed. Integrated kernel and data management algorithms providing recovery support with minimum performance penalty are presented.

Modelling results are presented which provide insight into the relative performance of the alternatives. The variables considered include processor speed, disc access time and number of discs, read hit ratios, write wait probabilities, and effective multiprogramming level.

We investigated this scalability problem, and found it to be due to a lack of scalability in the algorithms used for subsystem and application level buffering and recovery management. Processor utilization was limited by subsystems waiting processes for disc accesses to complete. The disc write accesses were being generated by local buffer replacement and write-ahead log management. The disc read accesses were initiated to resolve local buffer read misses.

We wished to extend the basic algorithm integration for processor, main memory, semaphore and disc management to exploit current technology trends in order to realize significant improvements in system price/performance. We examined disc buffering, intermediate storage levels between primary and secondary store, disc caching in excess primary memory of the system processing unit, and integrating kernel resource management with higher level data management. We modelled the performance of these alternatives with varying processor speeds, disc speeds, number of discs, and read hit and write wait probabilities.

In order to gain insight into the interactions and to discover tradeoffs and improved algorithms, we bread-boarded and analyzed one alternative : integrated algorithms providing explicit global caching of discs in excess primary memory.

The analysis and bread-boarding effort gave insight into the alternatives to exploit processor and storage technology trends, the differences between caching discs and caching main memory, the requirements on disc caching imposed in transaction management systems,

and transient data objects, degree of parallelism, translation mechanisms, fetch, replacement, and write handling policies, interfaces between data management subsystems and the kernel, and external tuning controls. Measurements from the bread-board are provided which show the performance as a function of processor speed and the number of discs with and without intern caching enabled, as well as cache occupancy and read hit rates on a per disc basis.

Finally, the conclusions which may be drawn from this research into integrating data management are summarized.

6.2 Overview of Memory Hierarchies

Storage hierarchies and evaluation techniques for their optimization are discussed by Mattson, Gecsei, Slutz and Traiger in [Mat 70] and by Chow in [Cho 74].

Figure 31 shows a standard computer system storage hierarchy. A storage hierarchy provides a cost effective system organization for computer systems. Each successive level of a storage hierarchy uses lower cost, but slower, memory components. By retaining frequently accessed code and data in the higher speed memories, the system can operate at speeds close to the access times of the fastest memories but at costs approaching those of the slowest memories. The price and performance of a computer system is often dominated by the organization and management of its storage hierarchy.

These projections were obtained using the system model described in Appendix E. The workload parameters for the model runs and the workload for the measurement data on the bread-board implementation were obtained using a Production Management (PM) benchmark. HP Production Management 3000 is an application package which aids manufacturing in scheduling, tracking, and capacity planning using workorders for fixed quantities of specific products with individual start or completion dates. PM 3000 uses one Image database almost exclusively. The Shop Floor Control database consists of 29 datasets, 14 masters, and 15 details. There were approximately 5000 parts, 6000 orders, and 39,000 routings in the database which occupied over 500 Mbytes of disc storage.

The measured processor service requirement of the benchmark was 192 kinstr per disc reference. The disc subsystem service requirements were 24 disc references per transaction, a disc read:write ratio of 7:3, disc references distributed uniformly across the discs, and a mean transfer size of 1 kbyte.

The overhead for disc access channel program construction and interrupt handling was assumed to be 1 kinstr in the modelling. Variations on these disc and processor service demands produce an analogous family of curves, albeit shifted in one or more dimensions.

Following the discussion and analysis of alternatives, the design and measurements from a bread-board implementation are presented. Design issues considered include location of cache in main memory, degree of integration of cache management with the management of code

COMPUTER STORAGE HIERARCHY

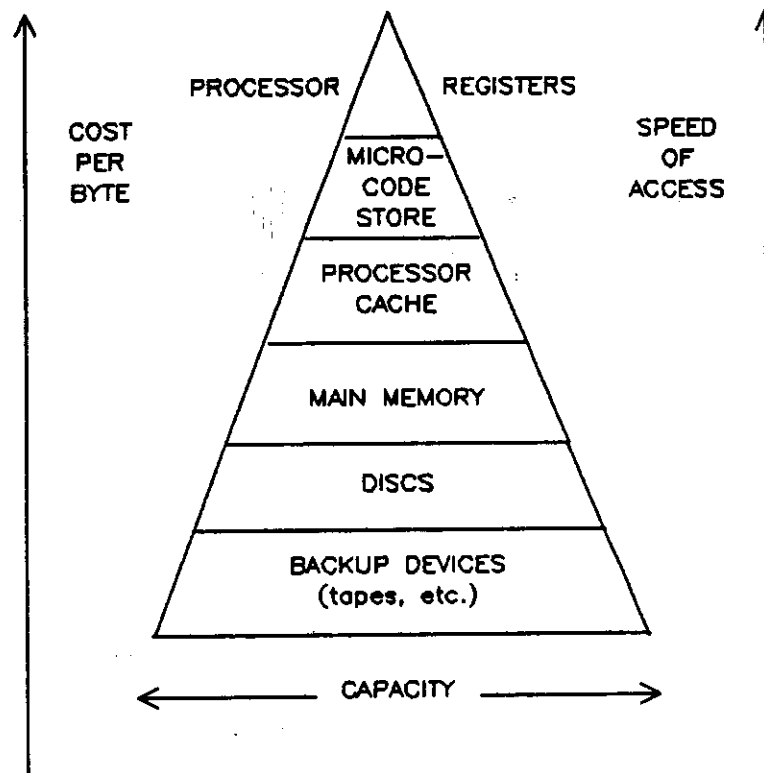


Figure 31: Standard Computer System Storage Hierarchy

Achievable system performance is a direct function of processor speed and utilization. Processor utilization is limited primarily by its waiting time caused by misses at various levels of the storage hierarchy. Thus, for optimal system price/performance, the processor speed and the capacity, speed, and management of the levels of the storage hierarchy must be matched. In order to fully utilize the processor capacity, the system must achieve a sufficiently high

These projections were obtained using the system model described in Appendix E. The workload parameters for the model runs and the workload for the measurement data on the bread-board implementation were obtained using a Production Management (PM) benchmark. HP Production Management 3000 is an application package which aids manufacturing in scheduling, tracking, and capacity planning using workorders for fixed quantities of specific products with individual start or completion dates. PM 3000 uses one Image database almost exclusively. The Shop Floor Control database consists of 29 datasets, 14 masters, and 15 details. There were approximately 5000 parts, 6000 orders, and 39,000 routings in the database which occupied over 500 Mbytes of disc storage.

The measured processor service requirement of the benchmark was was 192 kinstr per disc reference. The disc subsystem service requirements were 24 disc references per transaction, a disc read/write ratio of 7:3, disc references distributed uniformly across the discs, and a mean transfer size of 1 kbyte.

The overhead for disc access channel program construction and interrupt handling was assumed to be 1 kinstr in the modelling. Variations on these disc and processor service demands produce an analogous family of curves, albeit shifted in one or more dimensions.

Following the discussion and analysis of alternatives, the design and measurements from a bread-board implementation are presented. Design issues considered include location of cache in main memory, degree of integration of cache management with the management of code

Caches between primary memory and discs are discussed in [Kra 82, Hug 82], and in detail later in this chapter.

Smith [Smi 78] discusses long term file reference patterns and the use of automatic file migration systems which moves files between the discs and the next lower level of the storage subsystem. An example of such a storage level is the IBM 3850 [Hem 77] Mass Storage Subsystem. It consists of two walls of pigeon holes containing tape cartridges, a cartridge accessor which moves tapes between the pigeon holes and recording devices, and processors and staging discs which locate and move the data between the host the host and the storage subsystem using a virtual address interface.

6.3 Need For Memory/Disc Balancing

Processor speeds are increasing, and costs are dropping. There has been an order of magnitude improvement in processor price/performance in recent years. Current 32 bit microprocessors such as the 68020 [Ele 82] and National 32032 [Ele 84c] operate at cycle times of less than 100 nanoseconds. RISC architectures [Fit 81, Wik 82, Patt 82] offer the potential for significant performance improvements over these, with the direct execution of an instruction each cycle with cycle times approaching 10 nanoseconds.

These order of magnitude advances in processor speeds have not been matched by proportional advances in disc access speeds. There is an access time gap of four orders of magnitude, and widening, between a main memory reference and a disc reference in most current computer

probability of finding data when referenced at the highest levels rather than having to go to the lower levels of the hierarchy.

Traditional solutions for a low hit ratio at a certain level of the storage hierarchy include improving the management policies of the levels, increasing the capacity of the level incurring the low hit rate, speeding up the access time of the next lower level of the hierarchy, and introducing a new level into the storage hierarchy. Cost and technology determine which alternative or combination of alternatives is optimal.

The advantages of memory hierarchies have been exploited at different levels of computer systems.

The need for and design of high speed buffer storage between the processor and primary memory in large scale systems is discussed in [Con 69, Mea 70, Bel 73]. Processor/main memory caches are now common in medium and large scale systems.

Clark, Lampson and Pier [Cla 81] discuss the cache design for the Dorado personal computer. The cache is pipelined, yielding a cache access each cycle.

Lindsay [Lin 81] discusses the developing need for caches in microprocessor systems.

Raymond and Pucknell [Ray 81] discuss extensions of the cache concept to the microcode store level.

This trend in disc technology exacerbates the hierarchy imbalance. Since the disc subsystem often comprises over 50% of the system cost, the tradeoff towards a small number of high capacity discs is attractive. However, the demand for a disc grows with its capacity. Thus, realizing the cost advantages of a few large capacity discs rather than several small capacity discs reduces the potential for parallel service. This increases the mean disc queue lengths, and thereby the expected values for disc service response time.

The IBM 3380 is a current state-of-the-art disc storage device. It has an areal density of 10^7 bits/cm² with a total multidisc capacity of 2 Gbytes, an average seek time of 16 ms, and a data transfer rate of 3 Mbytes/sec. The Gartner Group reported at the recent IBM Large Computer Conference that IBM will introduce double density versions of its 33XX in 1984 providing 4, 6 and 20 Gbyte drives, with quad density drives expected in the 1987-88 time frame. Further, they report that, due to performance reasons, disc space utilization of IBM 3380 drives in IBM installations is currently limited to 50-55% of capacity.

Magneto-optic disc storage devices [Tog 82, Bel 83] offer the potential for significant improvements in read/write disc technology. The primary limitation on disc density is the loss of signal amplitude due to inaccuracies in the positioning of the read/write mechanism. In magneto-optic devices, this is limited only by optical properties. Current such devices have densities in excess of 10^8 bits/cm². The mean access time of such devices is limited by inertia of the waveguide positioning, and mean access times will be on the order of 10 ms.

system storage hierarchies. In comparison to this access time gap between these storage levels, the access time gap between a processor cache reference and a main memory reference is normally only a factor of three to five.

Disc densities are rapidly increasing, and large capacity discs offer significant price advantages over small capacity discs. As is shown in Figure 32, large capacity discs cost one fourth as much per Mbyte as small capacity discs.

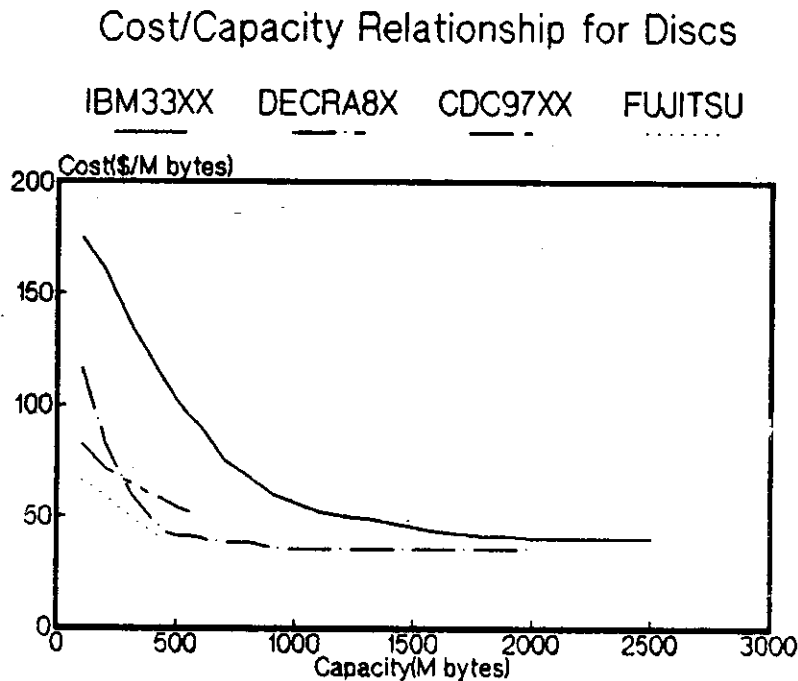


Figure 32: Current Cost vs Capacity for Discs

This effect is shown in Figure 33. The top graphs show transaction throughput as a function of processor speed for a fixed number (4) of 25 ms and 10 ms discs. The bottom graphs show transaction throughput as a function of the number of 25 ms and 10 s discs at a fixed processor speed of 3 MIPs. A fixed channel bandwidth of 2 Mbytes/sec is assumed for all the runs.

We see that without secondary store caching and with 4 25 ms discs, effective processor utilization extends only to 1 MIP, and beyond 1 MIP performance is linear with respect to the number of discs. With faster discs (4 10 ms discs), effective processor utilization extends through 4 MIPs with a sharp dependency on the number of 10 ms discs for the higher multiprogramming levels.

This indicates that in order to effectively utilize higher speed processors with conventional storage hierarchies and management techniques, faster discs, more discs, and higher effective multiprogramming levels are required. As discussed above, the technology trends in secondary store devices do not support this direction.

Access to tracks within 15 track positions of the current write head position can be achieved by deflection of the laser beam, providing access times on the order of 1 ms in this range. Data rates in excess of 6 Mbytes/sec have been demonstrated with this technology [Bel 83].

Replacing discs with secondary storage devices employing CCD, magnetic bubble, or semiconductor RAM technology has shown limited cost effectiveness. Bubbles and CCD have not been able to keep pace with the density improvements and resulting drop in cost per byte of semiconductor RAM, while the cost per Mbyte of semiconductor RAM is still two orders of magnitude greater than that of discs. Magnetic bubble memories are beginning to be used however as floppy disc replacements [Ele 84a], costing twice as much as floppy drives while providing access times a third of those of floppies.

Recent trends and projections on semiconductors are presented in [Ele 84b]. CCDs have been falling in consumption and are expected to continue to do so. Current consumption of magnetic bubble memories is ten times that of CCDs, but one tenth as much as random access semiconductor memory and discs. The use of magnetic bubble, semiconductor memory and disc systems are projected to grow at a similar rate relative to one another over the next five years.

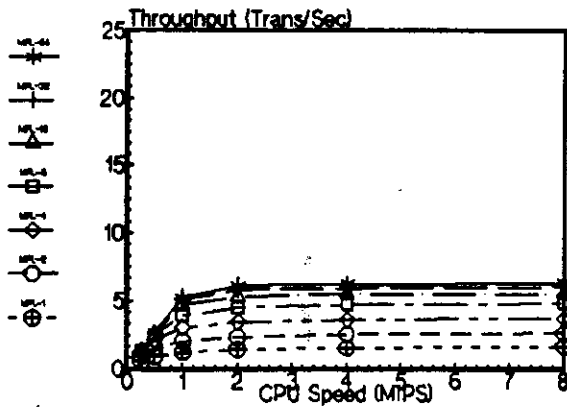
The large access time gap between semiconductor primary memory and magnetic disc secondary memory in current storage hierarchies causes non-linear system performance relative to processor speed. This limits the exploitation of high speed, low cost processors and high density discs to realize improved system level price/performance.

Utilization of processor capacity depends not only on mean disc access times and the number of discs, but also on sustaining a high effective multiprogramming level in order to exploit processor multiplexing opportunities when secondary store service delays are incurred. The effective multiprogramming level is limited by load and by concurrency control mechanisms.

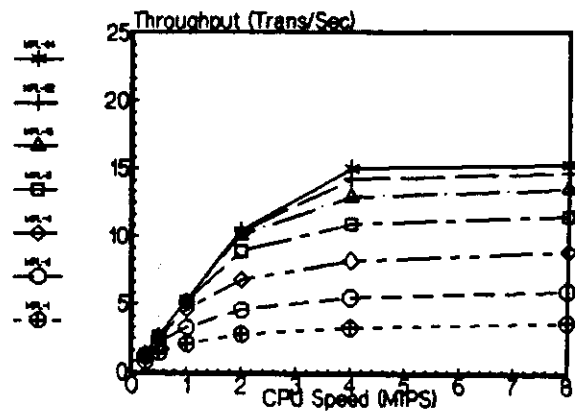
The impact of concurrency control on the effective multiprogramming level can limit the exploitation of high speed processors with multiuser database intensive workloads. In transactional database systems, a transaction commit must transform the database from a consistent state to a consistent state. When transactions are allowed to execute concurrently against the same database, consistency is maintained by share locking the data read and exclusive locking the data written by each transaction until the transaction commits. When popular structures are held locked during secondary storage access, the effective multiprogramming level is reduced due to the "convoy effect." This effect is described by Gray [Fly 78] in his "Notes on a Database Operating System." In this state, most of the processes are queued on the semaphore while the holding process spends its time waiting for disc transfers. Processor multiplexing opportunities are thereby reduced, and system performance rapidly flattens out with respect to increasing processor speed.

6.4 Alternative Balancing Approaches

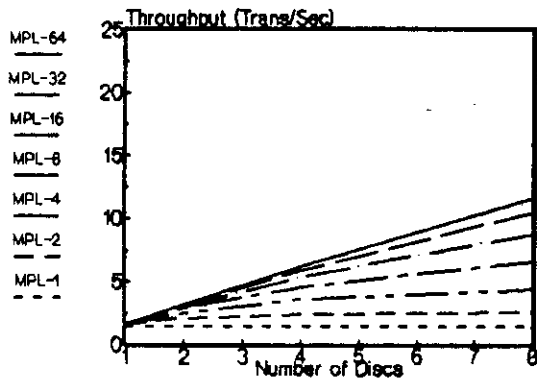
NO DISC CACHING WITH 4 25 ms DISCS



NO DISC CACHING WITH 4 10 ms DISCS



25ms DISC DEPENDENCY WITHOUT CACHING
3 MIP CPU With 25 ms Discs



10ms DISC DEPENDENCY WITHOUT CACHING
3 MIP CPU With 10 ms Discs

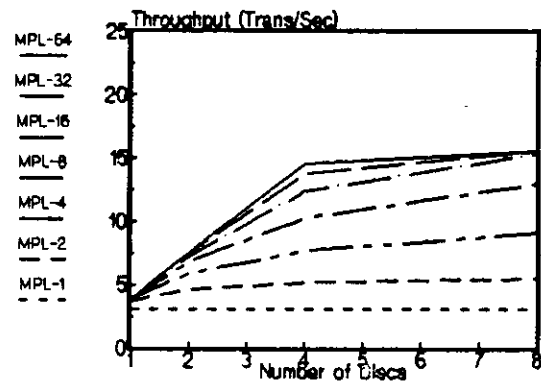


Figure 33: Conventional Hierarchy Management Performance

Read Hit Percentages vs Cache Size

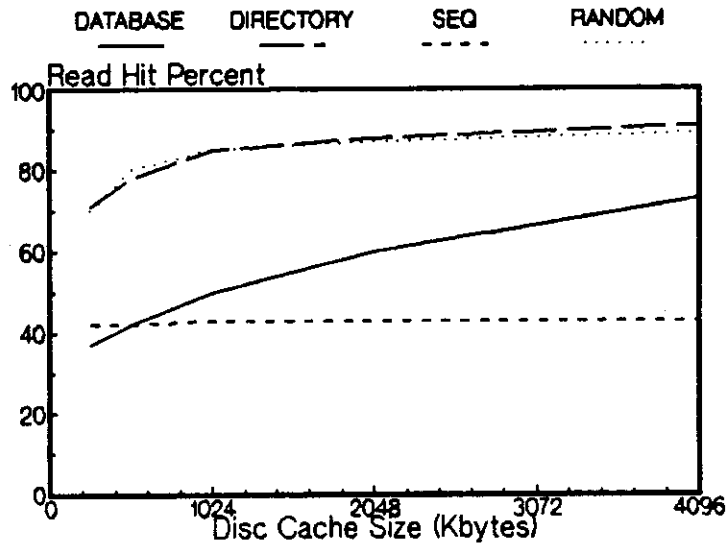


Figure 34: Access Method Hit Rate vs Cache Size

6.4.1 External Caching Techniques

Caching techniques have been applied to the disc subsystem in order to reduce the effective disc access time to secondary store. Caching has been implemented in discs, controllers and as a standalone system component interfacing one or more secondary storage devices to the main memory.

The use of a local cache per disc is depicted in Figure 35. Smith [Smi 76] discusses buffering discs using bubbles, CCDs, or electronic beam memories. He concludes that three buffers each a cylinder in size would produce a hit ratio on the order of 96%, with LRU working well as a replacement policy. IBM [Com 83] announced an intelligent cached

As shown in Appendix E, system throughput is directly proportional to effective multiprogramming level, and inversely proportional to processor and disc response times and disc visit frequencies. Consequently, efforts to overcome the limitations in exploiting the trends in processor and memory technologies have focused on : reducing the effective secondary store access time through external disc caching techniques; reducing the number of secondary store visits per transaction through internal disc caching in the primary memory; and sustaining large effective multiprogramming levels through improved concurrency management schemes. These alternatives are studied in this section.

The potential of secondary store caching schemes is indicated in Figure 34. This figure shows the read hit rates which can be achieved by the various access methods as a function of cache capacity. The data for this graph was using the disc cache simulation model on disc access traces obtained from the marketing support installation. Hit rates of 85% for the non-sequential access modes are achievable. This correlates with results using trace data from other installations as well, and is in line with hit rates reported in external cache implementations described below. The mean cache residency time in the larger cache sizes exceeds 5 minutes. In the comparisons of disc caching alternatives below, mean read hits of 85% are used. The dependencies of the alternatives on read hit rate are examined as well.

INTERMEDIATE STORAGE LEVEL

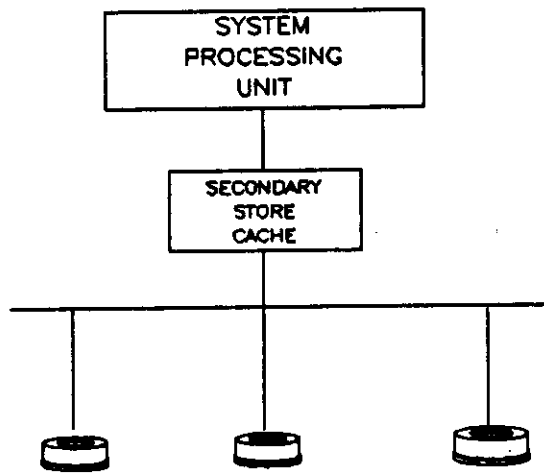


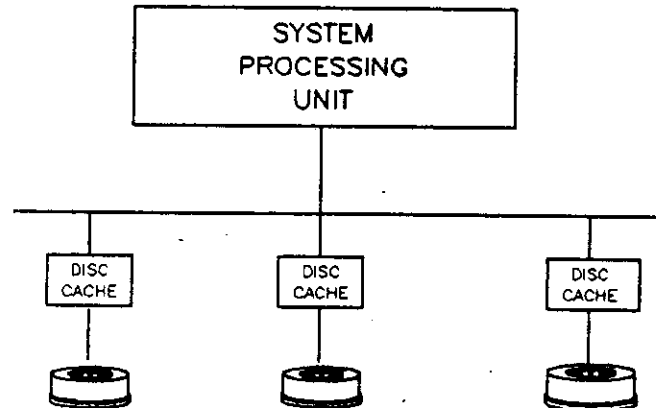
Figure 36: External Caching : Intermediate Level

The use and organization of a CCD memory level which could serve to close the gap between semiconductor and disc access times is discussed in [Cro 76, Reg 76, and Pan 76], and the use of magnetic bubble memory systems for this purpose is discussed in [Cha 74, Jul 76 and Chen 78]. But since bubbles and CCDs have not been able to keep pace with the density improvements and drop in cost per byte of semiconductor RAM, they have not qualified in gap filling efforts. Rather, the technology of choice for external disc caches has been semiconductor random access memory.

Hugelshofer and Schultz [Hug 82] describe such a semiconductor disc cache marketed by Computer Automation Inc. consisting of 2 Mbytes of RAM placed between the processing system and up to four moving head

disc featuring a 384 kbyte microprocessor driven controller that optimizes seeks as well as caches recently referenced data. Krastins [Kra 82] discusses a cache which is integrated with the disc controller consisting of 1-2 Mbytes of RAM. The cache buffers full physical tracks. They report a hit rate of 85% and a mean access time of 8 to 12 ms, with hit processing time less than 1 ms.

CACHE/DISC

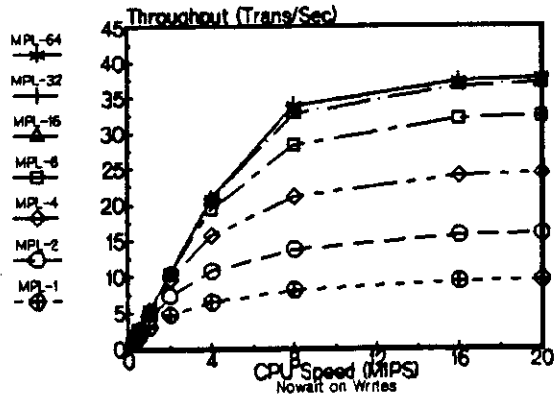


dlocext

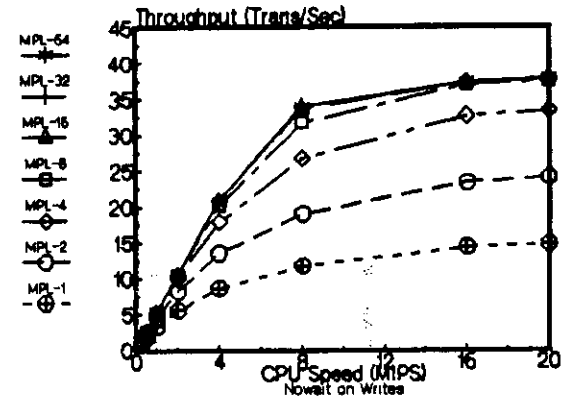
Figure 35: External Caching : Buffer Per Disc

The use of a cache front-ending the disc subsystem is depicted in Figure 36. This can be viewed as inserting a new level into the storage hierarchy.

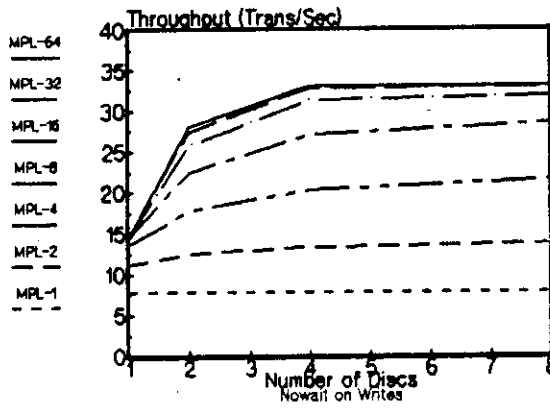
EXTERNAL CACHING CPU DEPENDENCY, 25 MS DISCS
4 25 MS Discs, RH=.86, WW=0



EXTERNAL CACHING CPU DEPENDENCY, 10 MS DISCS
4 10 MS Discs, RH=.85, WW=0



EXTERNAL CACHING DISC DEPENDENCY AT 7 MIPS
7 MIP CPU, RH=.85, WW=0, 25 ms discs



EXTERNAL CACHING DISC DEPENDENCY AT 7 MIPS
7 MIP CPU, RH=.85, WW=0, 10 ms discs

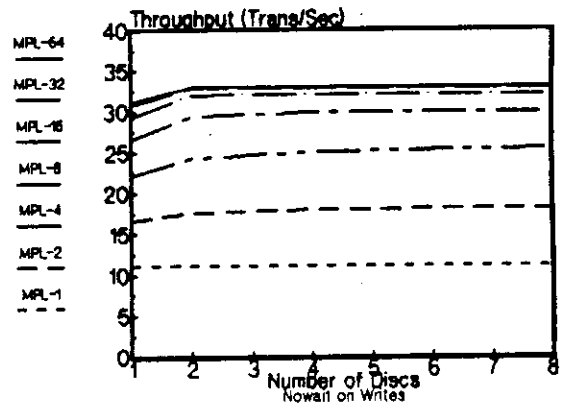
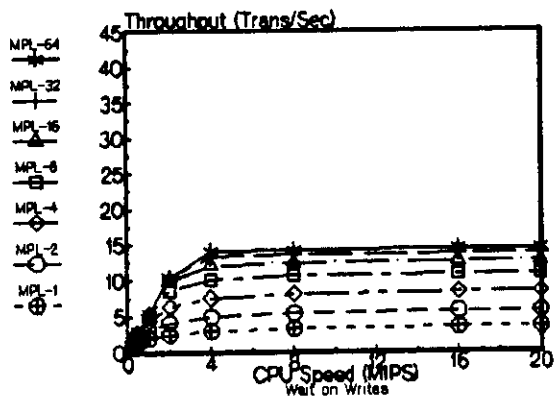
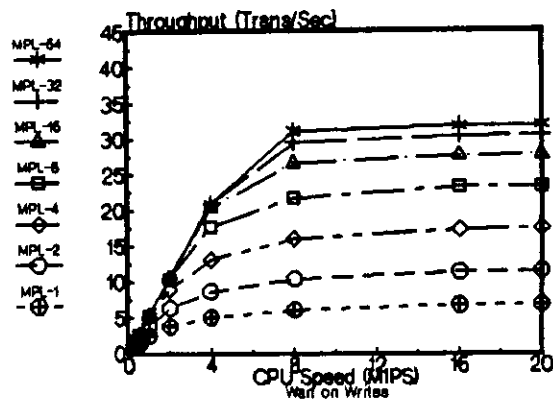


Figure 37: External Caching Without Write Wait

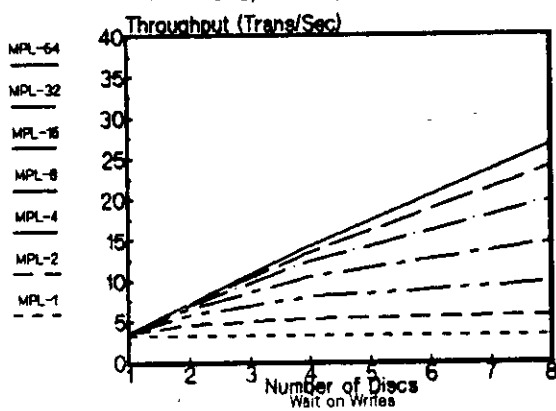
EXTERNAL CACHING CPU DEPENDENCY, 25 MS DISCS
4 25 MS Discs, RH=.85, WW=1



EXTERNAL CACHING CPU DEPENDENCY, 10 MS DISCS
4 10 MS Discs, RH=.85, WW=1



EXTERNAL CACHING DISC DEPENDENCY AT 7 MIPS
7 MIP CPU, RH=.85, WW=1, 25 ms discs



EXTERNAL CACHING DISC DEPENDENCY AT 7 MIPS
7 MIP CPU, RH=.85, WW=1, 10 ms discs

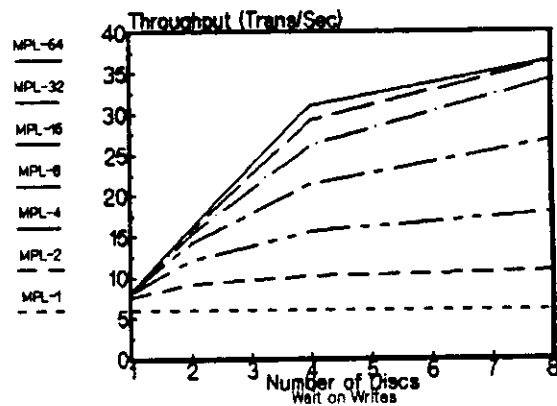
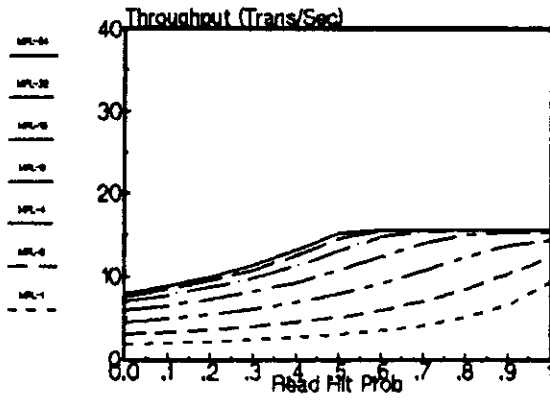


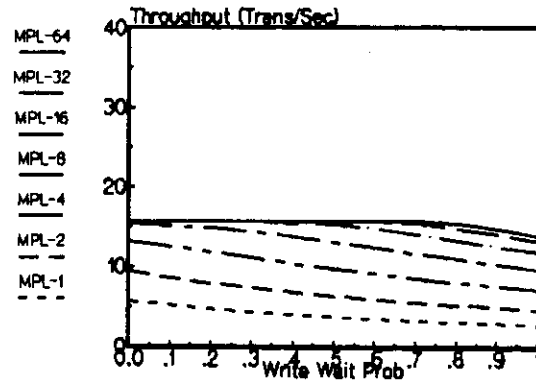
Figure 38: External Caching With Write Wait

Figure 39 shows the sensitivity of external caching to the read hit and write wait probabilities. These are shown for two processor speed slices, 3 mips and 7 mips, with 4 25 ms discs. The sensitivity to read hit and write wait probabilities increases with processor speed, with rapid loss of performance in high speed processors as the read hit probability drops and the write wait probability increases.

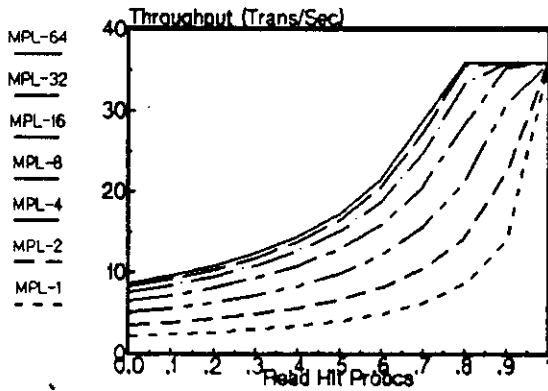
READ HIT DEPENDENCY WITH EXT CACHE, 3 MIPS
3 MIP CPU With 25 ms Discs and Write Wait Prob = (



WRITE WAIT DEPENDENCY WITH EXTERNAL CACHING
3 MIP CPU With 25 ms Discs and Read Hit Prob = .85



READ HIT DEPENDENCY WITH EXTERNAL CACHING
7 MIP CPU With 25 ms Discs and Write Wait Prob = (



WRITE WAIT DEPENDENCY WITH EXTERNAL CACHING
7 MIP CPU With 25 ms Discs and Read Hit Prob = .85

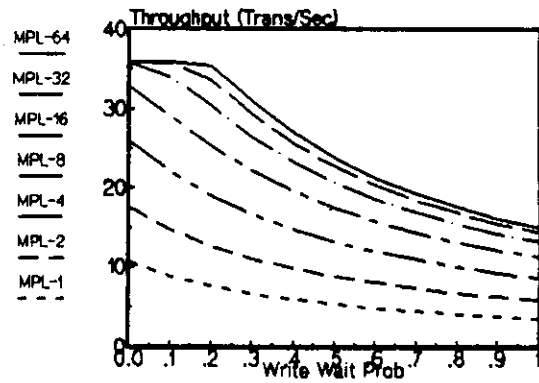


Figure 39: External Caching Sensitivity to RH and WW Prob

6.4.2 Using Large Primary Memories

With the improvements in memory densities and access times, very large main memories can be cost effective provided they can be exploited to reduce the traffic between main and secondary store. Techniques to exploit main memory for this purpose include auxiliary local buffering in applications and subsystems and global disc caching through explicit caching or file mapping.

Systems have conventionally provided limited caching of the discs in main memory through buffering on a subsystem, file, or application basis as shown in Figure 40.

LOCAL FILE BUFFERING

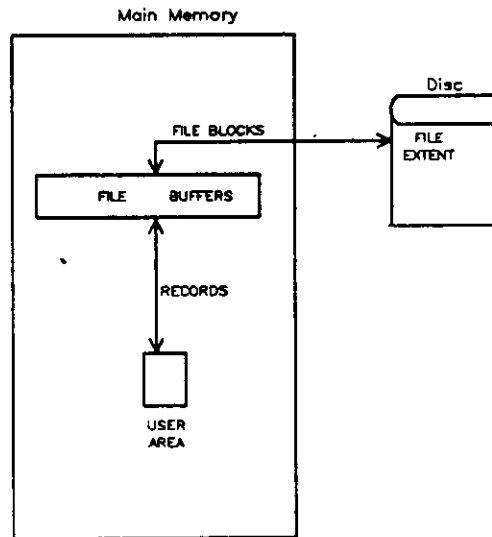


Figure 40: Local File/Application Caching

Centralized buffering schemes are employed in the Berkeley 4.2 Unix

file system for the DEC Vax computer family [Mck 82] and in a product for the IBM PC [Tec 83]. As shown in Figure 41, a fixed portion of main memory is set aside for global file buffers. When a file block read from disc is requested, the global buffers are checked first. If the requested block is present in a global file buffer, the read is satisfied with a move from the buffer to the user's address space. Otherwise, a global buffer is freed up, a disc read of the block is initiated into the selected global buffer, and when the read completes the data is moved into the user's space. File block writes are performed through global buffers as well.

GLOBAL FILE BUFFERING

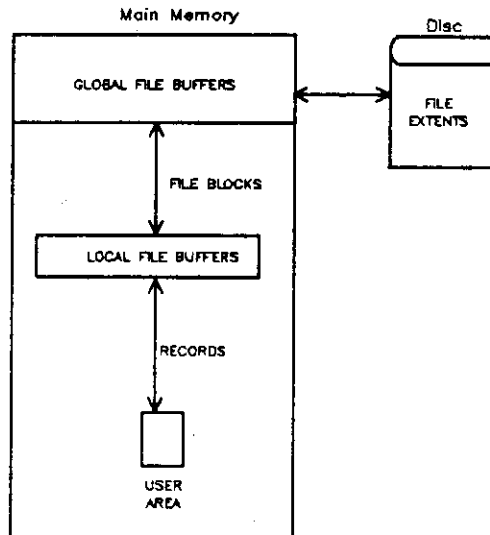


Figure 41: Global File/Application Caching

locates the required cached disc domain in main memory and moves the data between the cached region of the disc and the data area of the access method. This approach was implemented in the bread-board system for experimentation purposes, as described in later sections of this chapter.

With architectures supporting large virtual address spaces, pieces of files or discs or entire files or discs can be mapped directly into the address space. The location of a piece of a file or disc in main memory is then handled by the virtual to physical address translation hardware, and the normal memory management mechanisms handle fetching and replacing of pages of files or discs. This approach to secondary store caching is depicted in Figure 43. This approach was first employed in Multics [Org 73], and recently in [Bas 82, Red 80].

In global disc caching, main memory partitions disappear and cached disc regions containing file data are centrally managed with the pieces of transient objects by the main memory manager. In this approach to disc caching in main memory, pieces of the disc are mapped as data objects, and placed and replaced by the normal memory management algorithms as those used for code, stacks, etc. This approach is shown in Figure 42.

GLOBAL DISC CACHING

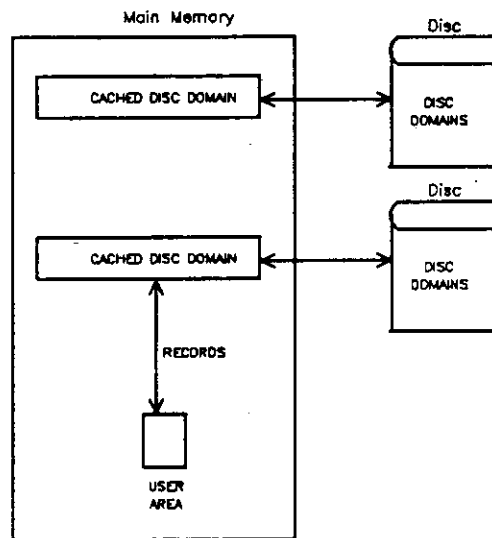


Figure 42: Explicit Global Disc Caching in Main Memory

Global disc caching in main memory can be either explicit beneath the disc access interface or implicit through the use of file mapping.

With explicit disc caching in main memory, the access methods continue to address the discs, but a software translation mechanism

FILES MAPPED IN VIRTUAL SPACE

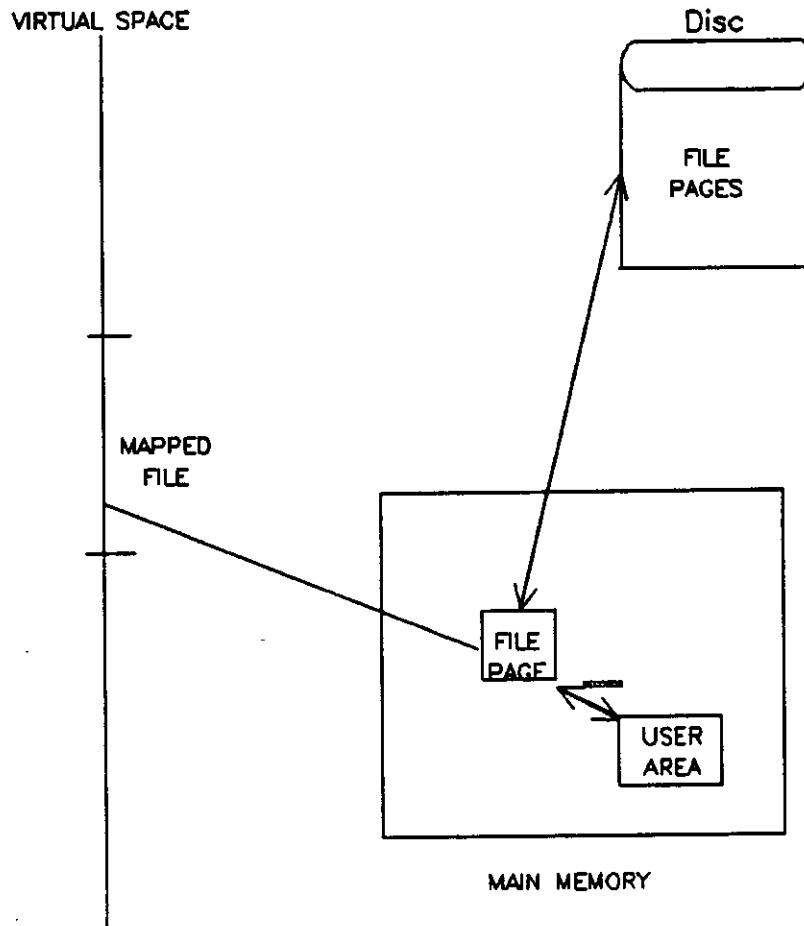
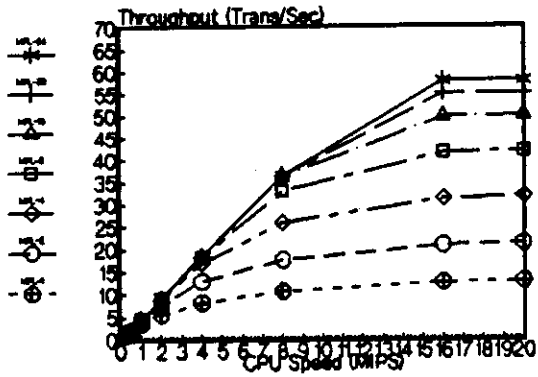


Figure 43: Internal Caching Through File Mapping

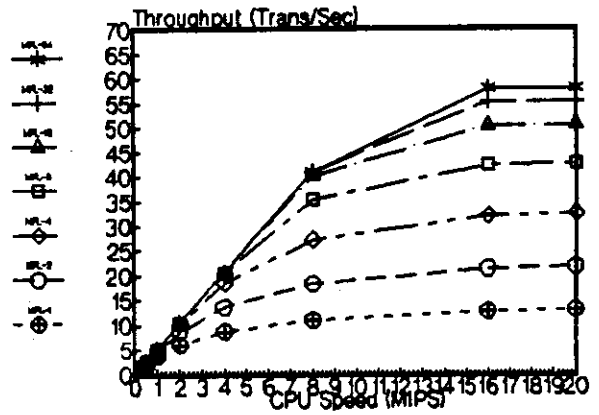
The performance characteristics of file mapping relative to explicit internal caching are shown in Figure 44. The top charts show the performance of the two schemes as a function of processor speed for a broad range of processor speeds, and the bottom charts a blowup for low

processor speeds. The performance characteristics for the two schemes are seen to be very similar for higher speed processors where the translation overhead of explicit internal caching becomes negligible. File mapping has the advantage for low speed processors.

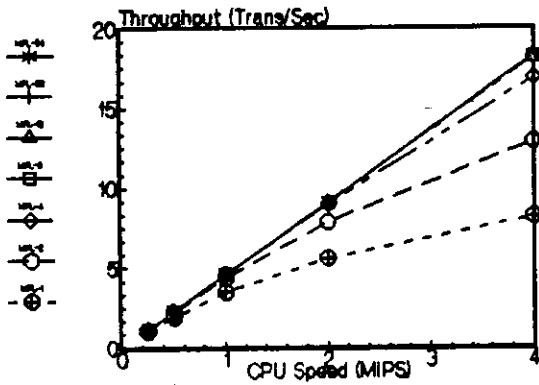
EXPLICIT INTERNAL CACHE WITH 4 25 MS DISC:
RH=.85, WW=0



FILE MAPPING WITH 4 25 ms DISCS
RH=.85, WW=0



EXPLICIT INTERNAL CACHE W/O WRITE WAIT
4 25 ms Discs, RH=.85, WW=0



FILE MAPPING W/O WRITE WAIT
4 25 ms Discs, RH=.85, WW=0

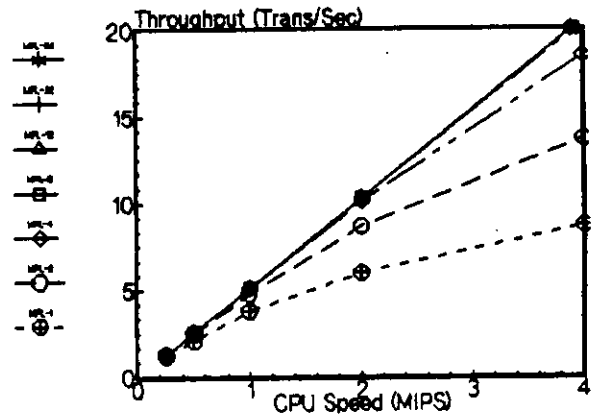
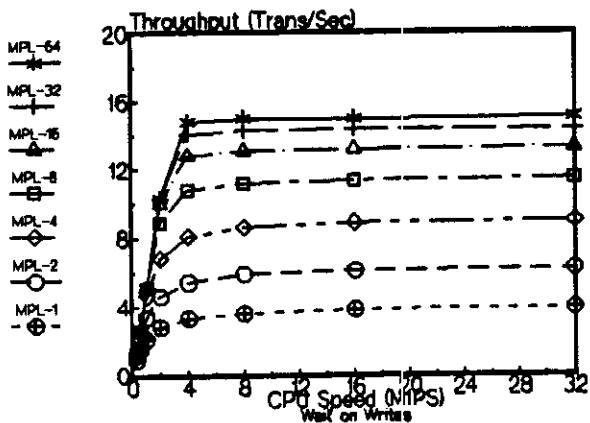


Figure 44: Comparison of Explicit Internal/File Mapping

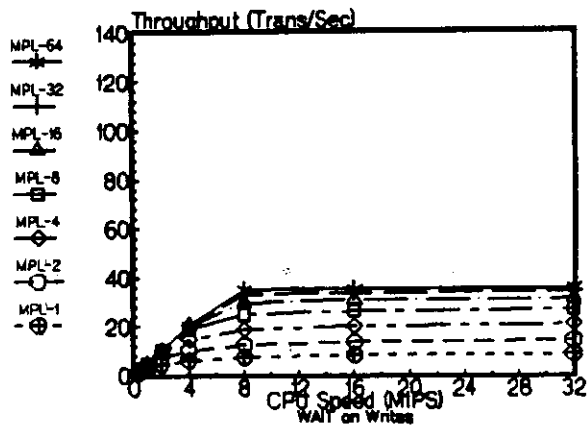
Since the performance characteristics of explicit internal caching of secondary store and file mapping are similar, only the full characterization of file mapping is presented here. The performance characteristics of file mapping as a function of processor speed, disc access time, and number of discs with and without waiting for write posts to complete are presented in the following two figures.

Note that without write wait file mapping provides effective processor utilization through 16 MIPs with 4 25 ms discs and through 32 MIPs with 4 10 ms discs. Waiting for writes causes a sharp drop in the effective processor utilization and a sharp increase in the dependency on the disc access time and the number of discs.

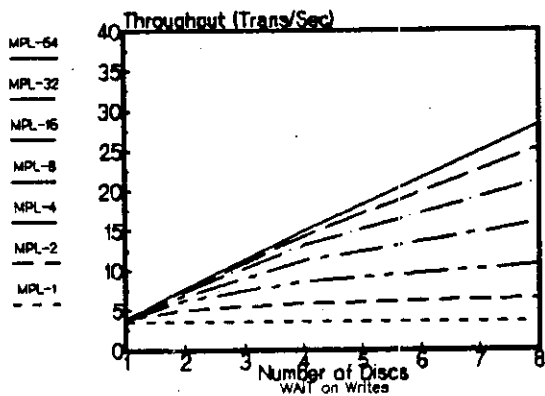
FILE MAPPING CPU DEPENDENCY, 25 MS DISCS
4 25 MS Discs, RH=.85, WW=1



FILE MAPPING CPU DEPENDENCY, 10 MS DISCS
4 10 MS Discs, RH=.85, WW=1



FILE MAPPING DISC DEPENDENCY AT 7 MIPS
7 MIP CPU, RH=.85, WW=1, 25 ms discs



FILE MAPPING DISC DEPENDENCY AT 7 MIPS
7 MIP CPU, RH=.85, WW=1, 10 ms discs

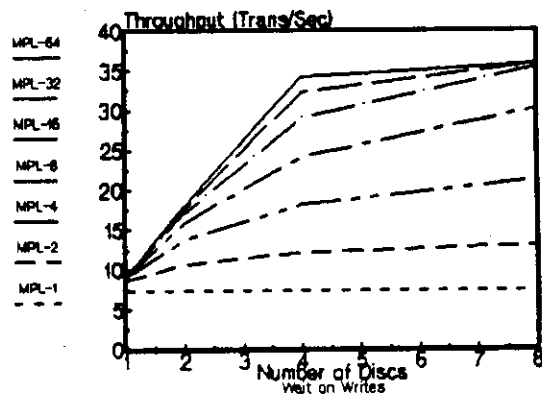


Figure 46: File Mapping With Write Wait

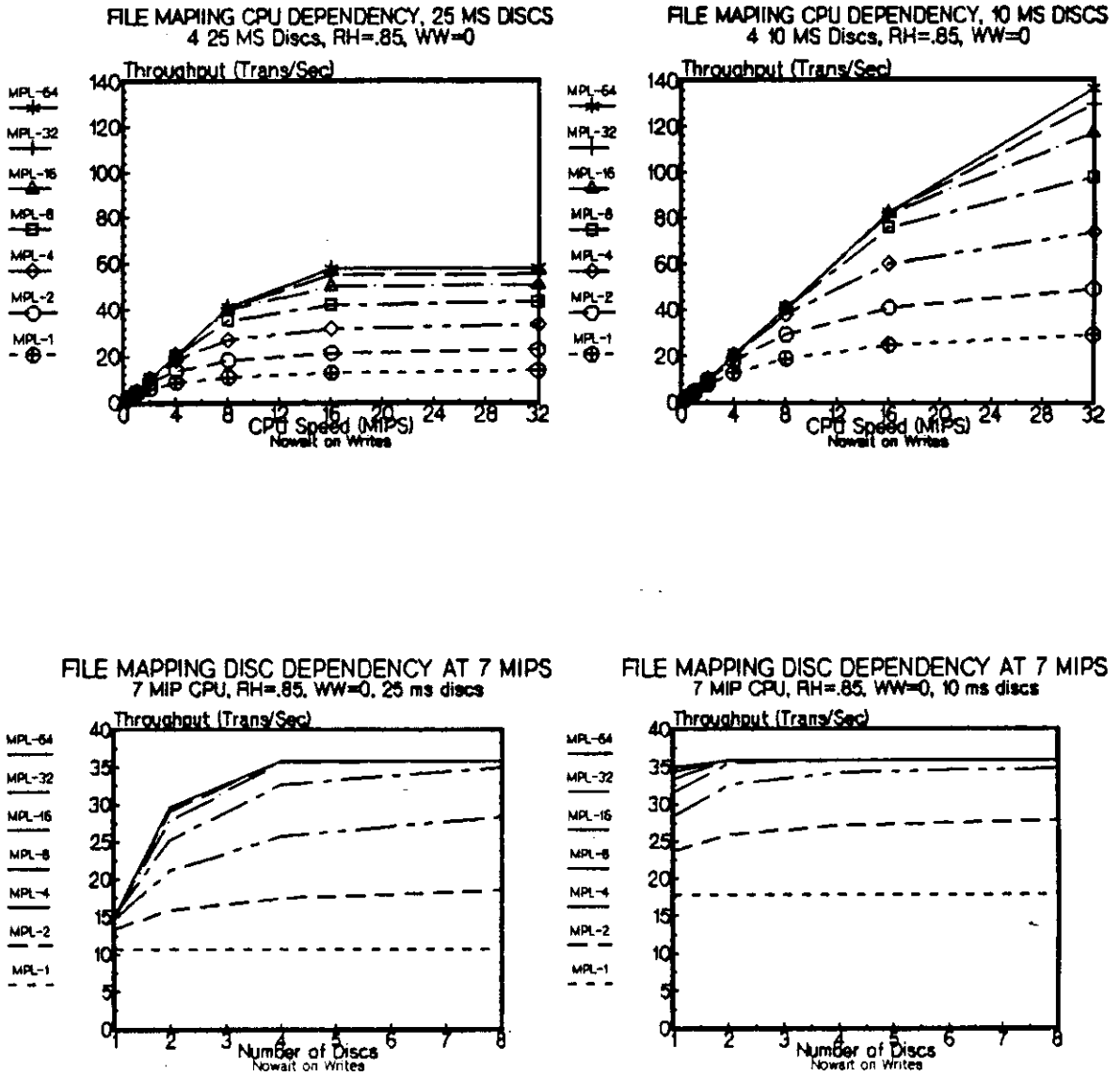


Figure 45: File Mapping Without Write Wait

Figure 47 shows the sensitivity of file mapping to the read hit and write wait probabilities. These are shown for two processor speed slices, 3 mips and 7 mips. The sensitivity to read hit and write wait probabilities increases with processor speed, with rapid dropping in effective processor utilization for high speed processors as the read hit probability decreases and the write wait probability increases.

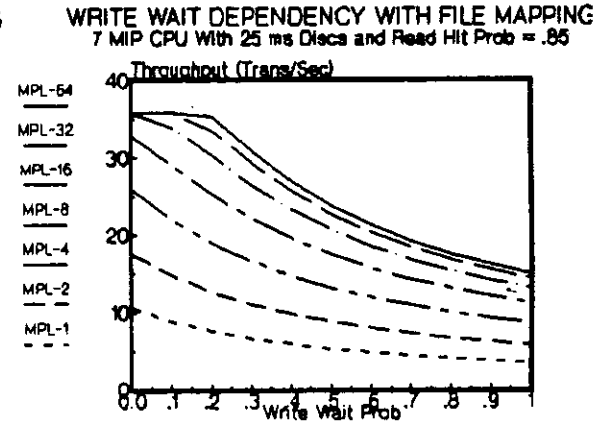
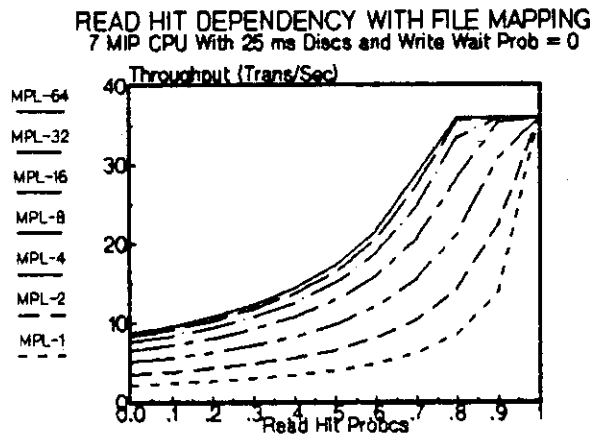
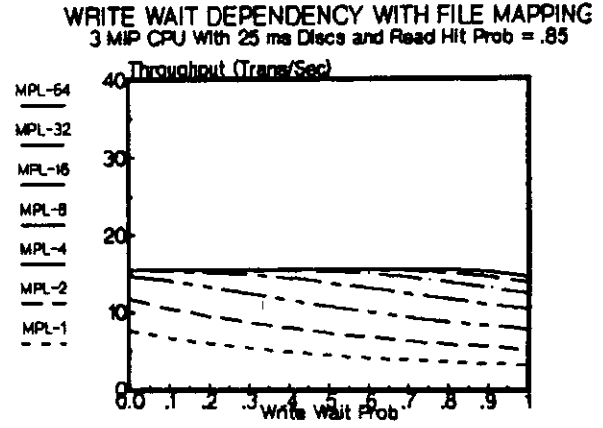
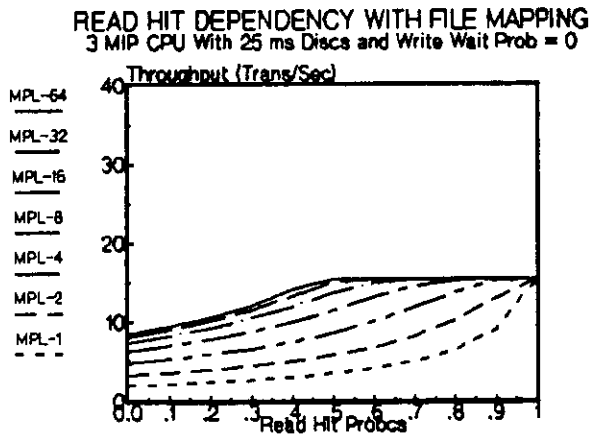


Figure 47: File Mapping vs Read Hit/Write Wait Prob

6.4.3 Improved Concurrency Control

We saw in the preceding graphs that system throughput varies strongly not only with processor and disc subsystem capacity, but also with the sustained multiprogramming level. Concurrency control mechanisms can severely limit the effective multiprogramming level through the convoy effect described earlier. Even with the methods of secondary store caching to increase parallelism and reduce access time to disc data, processor utilization of higher speed processors may still be limited due to the convoy effect caused by read misses and write waits incurred while holding popular semaphores. Advances in concurrency control mechanisms to provide maximum concurrency while still guaranteeing data consistency are surveyed in this section.

Eswaran, Gray, Lorie and Traiger discuss the concepts and necessary conditions for consistency in database systems in [Esw 74]. Transactional database systems guarantee that data which is used during a transaction is still valid when the transaction commits. This data consistency requirement is normally supported by share locking data which is read and exclusive locking data which is written.

Lock conflict frequencies are a function of lock granularity and lock bind time. The finer the lock granularity and the later the lock bind time, the lower the lock conflict frequency. However, refined locking increases the number of locks which must be obtained, thereby increasing lock management overhead. Also, the later the lock bind time, the greater the probability for deadlock.

In predicate locking [Esw 74], predicate locks correspond to logically related data items. The predicate locks are not normally related to the physical location of the data, and frequently cross data sets. Predicate locking results in the early locking of more data than is used by a transaction.

Gray [Fly 76] discusses the locking scheme in system R. Locks are accumulated as a transaction proceeds, and released at commit time. All data items which are read during a transaction are share locked, and all data items written by a transaction are exclusive locked for the duration of the transaction. Deadlock detection and recovery must be performed to support this locking scheme. This locking scheme locks the minimum set of data required for transaction consistency, and waits to lock the data until the last possible moment.

Low cost deadlock detection schemes are discussed in [Agr 83]. Rather than using watchdog timers to detect deadlocks and to restart transactions, "waits-for-graphs" are maintained and periodically checked for cycles indicating deadlocks.

Optimistic concurrency control mechanisms are discussed by Kung and Robinson in [Kun 81]. They discuss non-locking protocols which maintain lists of read and modified data, with checking the lists at commit time to see if the data used in the transaction had been modified by another transaction. Conflicts are resolved by restarting the transaction.

Chan and Gray [Cha 83] discuss the use of old time-stamped logged versions of data to provide a consistent view of the database without locking for read-only (query) transactions.

These techniques can help sustain high multiprogramming levels. The locking can be integrated with the kernel's management of resources by applying priority bumping to the holder of a lock when a more urgent process queues for the lock.

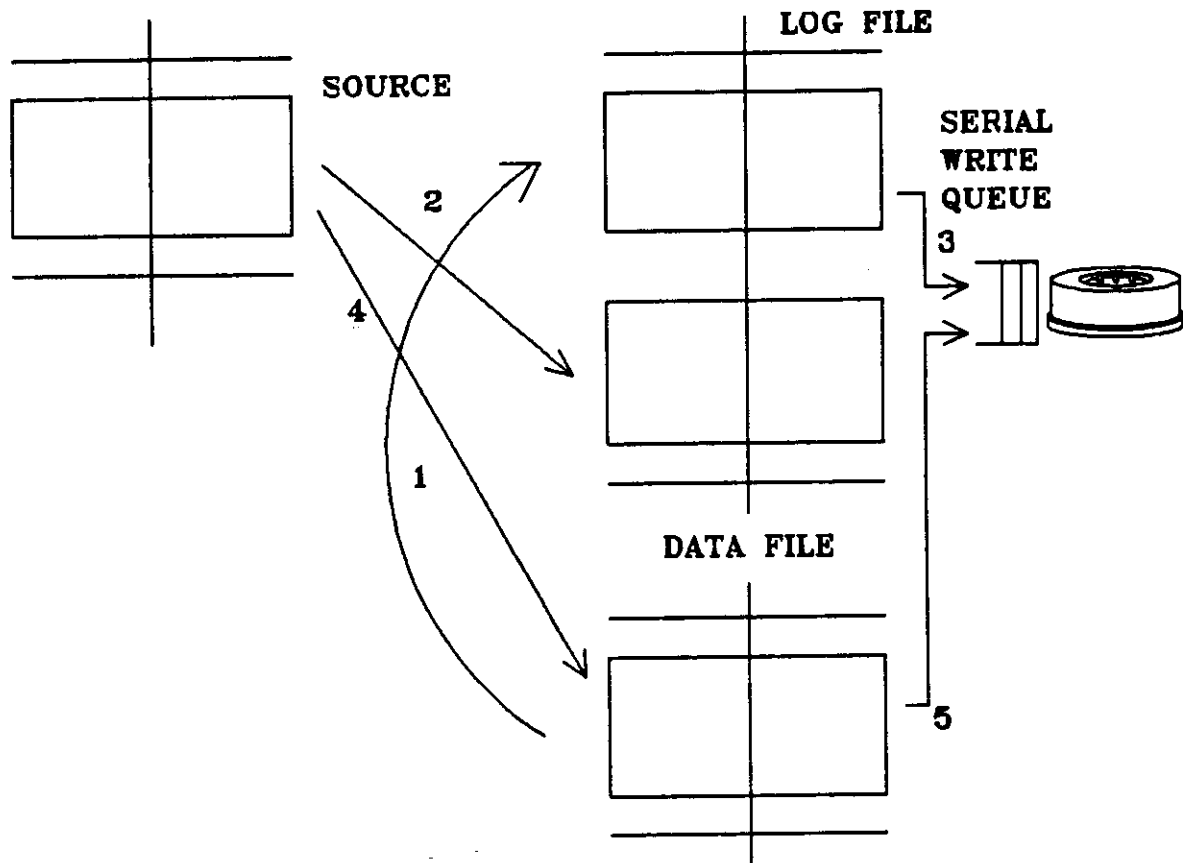
6.4.4 Requirements of Transaction Recovery

Transactional database systems guarantee that the database is left in a consistent state in the event of transaction aborts or system/disc failure. The standard mechanism used to achieve transaction recovery is through the use of a write-ahead log.

Write ahead logging is discussed in [Gra 76]. Before the data in the database is modified, a copy of the old image and new image of the data is written to a log file, along with identification information. If the transaction aborts, the old images of data modified by the transaction are read from the log and restored into the database, thereby "undoing" the actions of the transaction. If the system crashes, when it is restarted a utility is run which goes through the log file and undoes any actions of uncommitted transactions. If a disc fails, the database is restored to its last backed-up state, and the actions of committed transactions are redone using the log file.

In order for transaction recovery through write-ahead-logging to work, the before and after images of the data along with identifying information must be posted to the disc prior to the update of the database with the new values. Most database systems issue the log write, then wait for the physical post to complete before issuing the database write.

We saw in the preceding sections the significant impact of having to wait for writes. In order to minimize the probability of waiting for disc writes to complete, we developed a special mechanism which allows the specification of posting order constraints at post time or on a file basis so that posting can proceed without wait if only a post order constraint exists. With this facility, the post of the log can be issued nowait, and the database write can be executed immediately. The kernel guarantees that posting order within a serial write queue matches the chronological order of post initiation. The sequence for write-ahead log use of this facility is depicted in Figure 48.

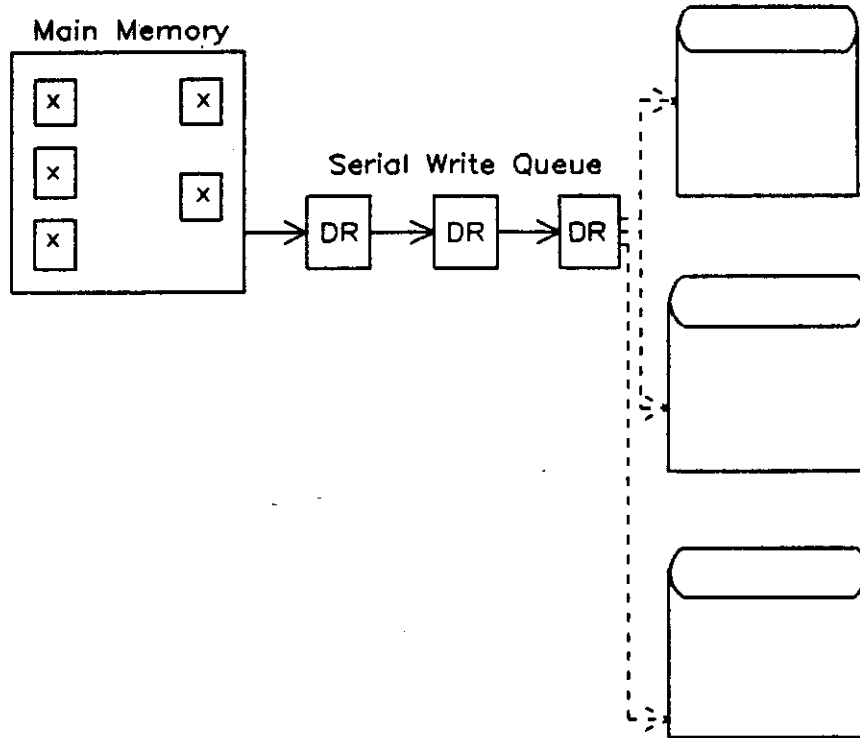


WRITE AHEAD LOGGING WITH SERIAL POSTING

Figure 48: Nowait Logging via Kernel Serial Posting

In order to insure disc consistency, only one write access for a serial write queue can be pending at a time. This is depicted in Figure 49. Since this limits parallel use of available disc drives, multiple serial write queues for unrelated postings are beneficial.

SERIAL WRITE POSTING FOR DISC CONSISTENCY



DBERVQ

Figure 49: Serial Write Queue Management

6.5 Evaluation of Alternatives

In this section we evaluate the alternatives discussed above based on performance, cost and reliability. We identify complementary

approaches, and identify workload/configuration sensitivities effecting the suitability of alternatives.

Evaluating with respect to performance, we saw in the preceding section that the ability to effectively utilize processor capacity drops rapidly with increasing processor speed if secondary store caching is not performed. Faster discs and more discs provide benefit in uncached systems, but the benefits are not commensurate to those obtained through secondary store caching. The dependency on disc access time and the number of discs is dramatically reduced with the secondary store caching alternatives.

All of the secondary store caching alternatives are able to provide effective processor utilization at processor speeds several times those which can be effectively utilized without caching with equivalent processor/disc configurations.

Note however that when the processor is fully utilized without caching in a balanced configuration, explicit internal caching degrades performance relative to no caching. Figure 50 compares the effects of no caching, external caching, explicit internal caching and file mapping on the low end of processor relative to not caching. The added overhead of locating the disc region in memory and moving the data to the target area does not pay off when the processor utilization is high. This effect was observed in the bread-board measurements. The low end family members degraded in performance when explicit internal disc caching was enabled. External caching out performs internal explicit in high utilization ranges as well, since the data transfer is

performed in parallel with processor usage, so the move overhead is not consuming valuable processor time.

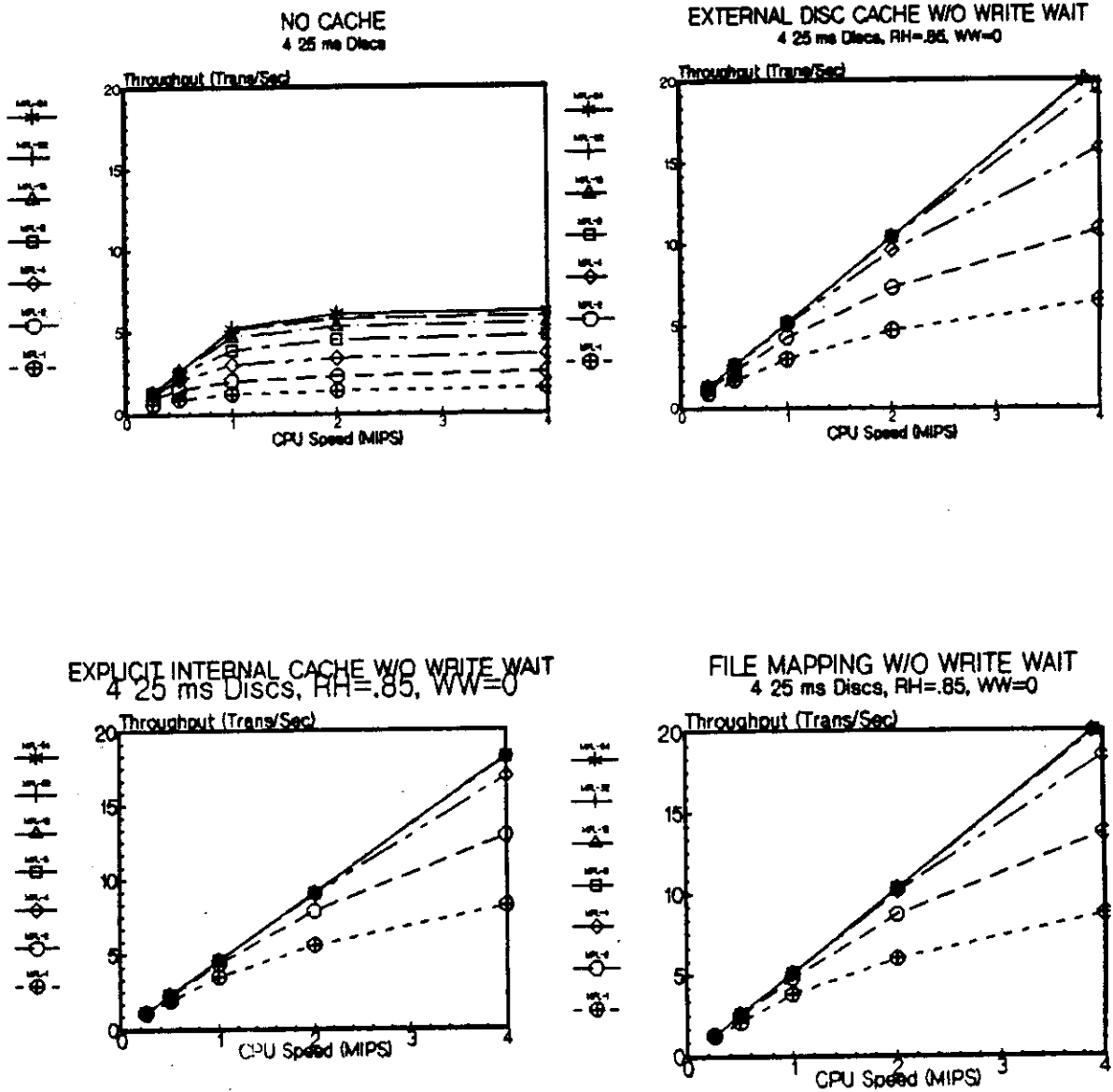


Figure 50: Alternatives in High Utilization Range

Figure 51 compares the performance of the alternative caching schemes across a broad range of processor speeds. Internal disc caching, either explicit or through file mapping, is able to provide effective processor utilization at speeds well above those effectively utilized by external caching with the same overheads and hit rates assumed. The overhead of internal caching becomes negligible as processor speeds increase, whereas the overhead in external caches stays fixed.

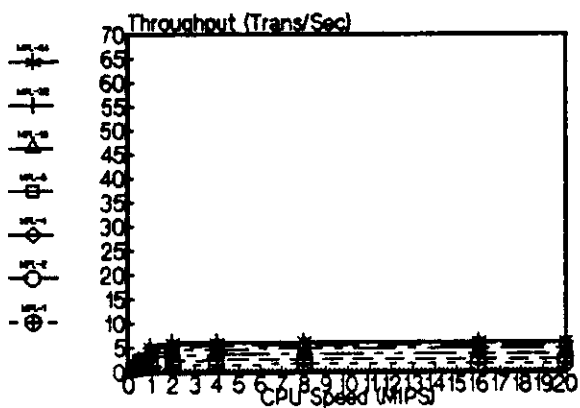
Special mechanisms can be employed to achieve high read hit rates and low write wait rates. Since the secondary store caching alternatives are sensitive to these parameters, it is important to exploit them.

Integrating the cache fetch, replacement and write handling with transaction management requirements is simple with internal caching, but would be difficult with external caching. Consequently, cache memory utilization and the sustained effective multiprogramming level of internal caching can be superior to that achievable in external caching.

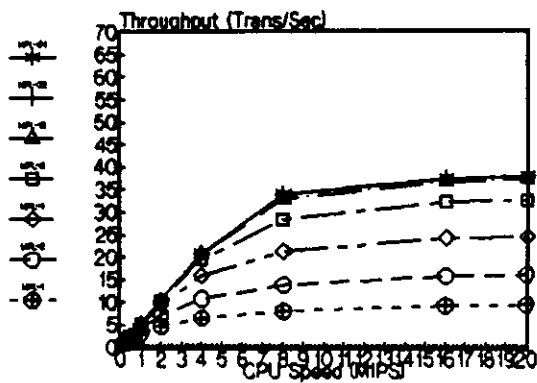
With internal caching, the size of a fetched disc domain on a read miss can be tailored to the structure of the data based on knowledge of the storage layout (e.g. fetch extents vs fetching tracks which contain unrelated data or only pieces of the required extent). The replacement policy can exploit operating system knowledge of access patterns (e.g. the policy can flush a cached disc domain from the cache memory after sequential reference and on file purging). Write posting orders and

write and read priorities can be adjusted to meet the current needs of transaction management.

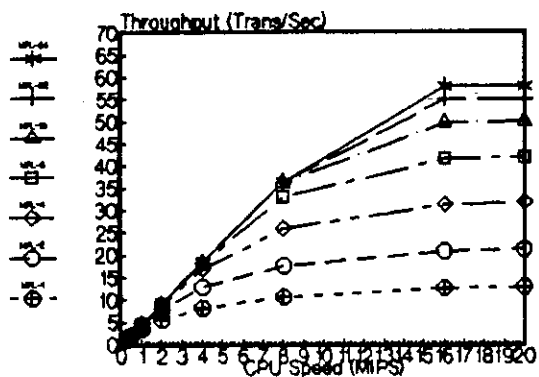
NO DISC CACHING WITH 4 25 ms DISCS



EXTENAL DISC CACHING WITH 4 25 MS DISCS
RH=.85, WW=0



EXPLICIT INTERNAL CACHE WITH 4 25 MS DISC:
RH=.85, WW=0



FILE MAPPING WITH 4 25 ms DISCS
RH=.85, WW=0

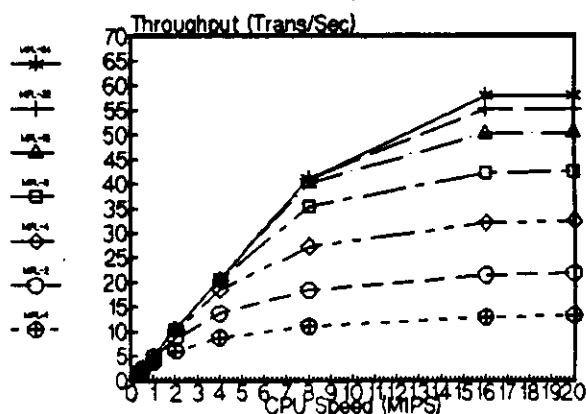


Figure 51: Alternatives Across Broad Processor Range

We now compare the alternate methods of internal caching of secondary store.

Development of multiple local buffer management schemes is redundant, and the buffering capacity is localized and unresponsive to current memory loading conditions. The amount of memory devoted to the buffering of a specific file or subsystem would likely be either excessive or insufficient at any given moment, depending on the current memory availability and the current workload demand and priority structure. Its the old fixed partition problem.

Global file buffers require a fixed partition of main memory between swap space and buffer space, so are not responsive to memory loading conditions. They suffer from the fixed partition problem as do local file buffers. Furthermore, they provide memory management functions, including space allocation, replacement, and disc access initiation and interrupt fielding. Thus, developing efficient algorithms for the global buffer manager is redundant to the development of the manager of memory for transient code and data. Moreover, as is discussed in the next chapter, they are not amenable to supporting efficient local caching in decentralized systems.

Explicit internal disc caching and file mapping overcome the fixed partition problems, so they can be expected to achieve higher read hit rates. Further, it is easier to integrate disc access post order and priority adjustments with these schemes.

Explicit internal disc caching in main memory requires a translation mechanism to locate a required cached disc domain in main memory. In large main memories there can be several thousand cached disc domains present at any moment, and locating a cached domain through a translation of disc to main memory address can be expensive in processor cycles. Additionally, if the processor has a cache, the location sequence can flush a large portion of the cache while chasing through list structures. However, explicit disc caching can be performed with any processor architecture.

File mapping would appear to offer the best of the alternatives for caching of discs in main memory. It leverages the main memory management mechanisms, has no limits on main memory space applied to file caching, and has no overhead for location of file domains in main memory. Applicability of this approach is limited however to specific processor architectures.

Stonebraker [Sto 83] discusses the issues involved in transaction management in architectures binding files into a user's address space. He identifies problems with such systems due to requiring the lock unit to be page multiples, the necessity of limiting mapping-in to only those pages of a file which are currently locked in order to maintain consistency maintenance for multiple updaters, and the complication in page replacement since pages of the write ahead log must precede permanent file page replacement in order to maintain integrity for transaction recovery. Stonebraker concludes that "without hardware

support specific to a given concurrency control scheme, all options appear to have substantial performance problems."

These problems are overcome if the processor architecture supports an address space which is sufficiently large to map in all the files concurrently and provides capability checking on a per page basis, and if the operating system and database cooperate to meet the transactional requirements. Supporting an address space of 64 or 96 bits need not significantly complicate the hardware or increase the cycle time or memory requirements. For example, hashing into a cache of virtual to physical address translations on translation look-aside buffer misses is performed on the the IBM System 38 [Hou 78] rather than using page tables to support its 48 bit address space. The database can invoke post requests of the log, and the kernel can adhere to post order constraints between the log and the database during disc scheduling.

Evaluating with respect to cost, using faster discs and more of them to overcome the performance limitations without caching is clearly not cost effective. External caching is more expensive than internal caching since additional power, cooling, cabinetry, and electronics is required in addition to the extra memory required for the caching. Also, more memory would be required to achieve the same hit rates in external caching unless the cache management is integrated with the operating system policies.

Evaluating with respect to reliability, internal caching introduces no new hardware components into the system. The reliability is identical to the uncached system whereas any of the peripheral cache architectures necessarily degrade system reliability due to their introduction of hardware components. The software/firmware complexity of explicit internal and external caching is roughly the same, so reliability degradation due to cache management is comparable. File mapping is simpler. Since the posting strategy of the peripheral cache is not integrated with the system posting strategy, a consistent level of integrity is not guaranteed for transactional database systems.

6.6 The Bread-Board Disc Cache

The kernel bread-board was used to investigate principles and integrated approaches to caching discs in the main store. Algorithm interactions were observed, and improvements developed. Differences between disc and main memory caching and areas in which architectural improvements would be of benefit were noted. The production disc cache system is described in [Bus 83].

The kernel resource management mechanisms and strategies presented in the previous chapter provided an efficient, extensible research base. These strategies were extended to support explicit disc caching in primary memory. The resulting mechanisms and strategies integrate kernel and data management. Access method knowledge of file structure and access type are exploited to enhance prefetch and replacement decisions for disc domains. Data recovery protocols are supported

without wait for posting through kernel post order constraint adherence. Priorities of disc accesses are adjusted to reflect the changing urgency of read and write requests due to data management locks or commits. Process priorities are adjusted to reflect lock states.

The overall structure is shown in Figure 52. The user program requests records. The file system maintains local file buffers, and on buffer misses or replacements accesses the I/O system to initiate buffer reads or writes. Beneath the I/O interface, pieces of the disc are cached in main memory. Actual disc transfers are initiated by the cache manager.

EXPLICIT INTERNAL DISC CACHE INTERFACES

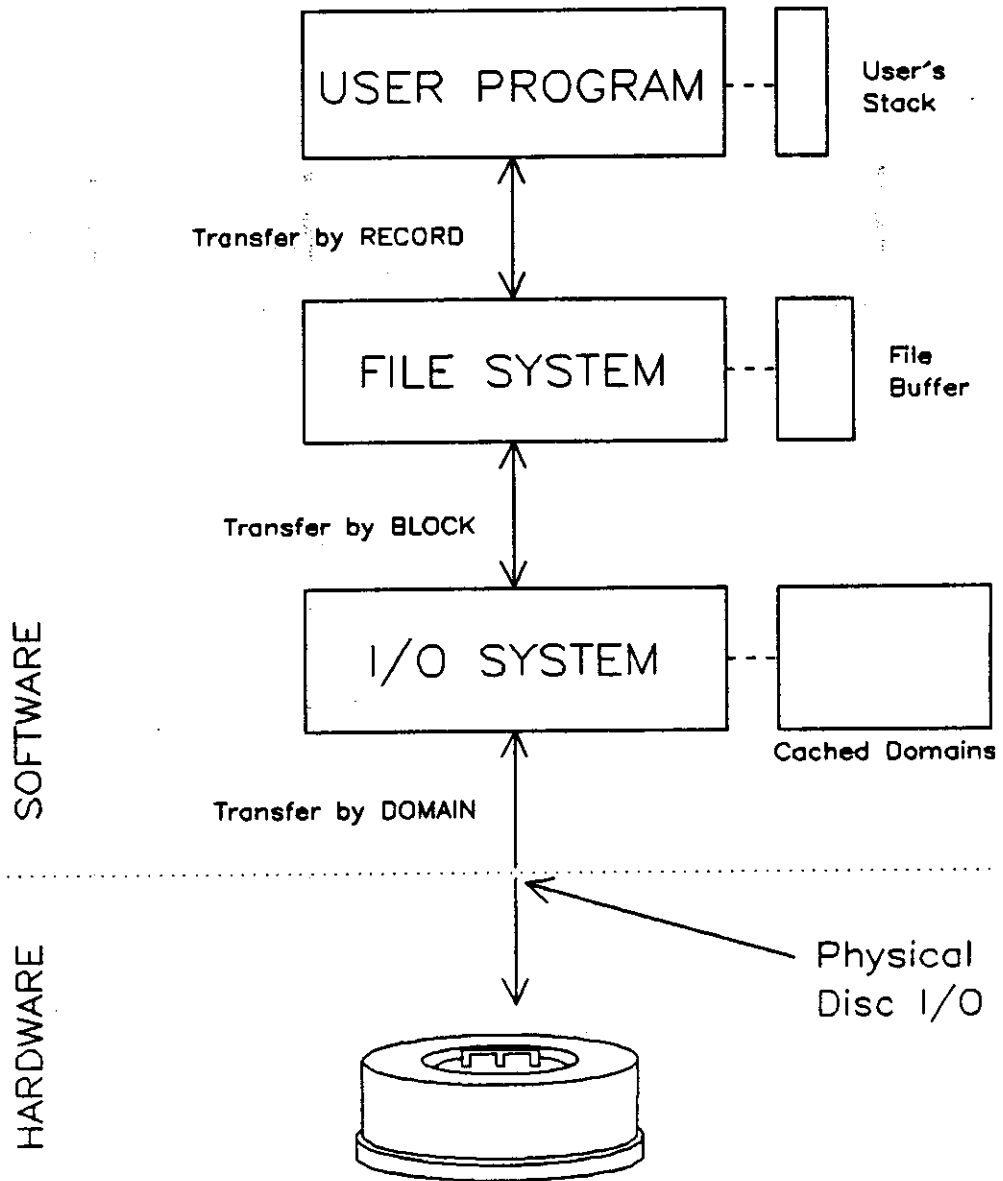


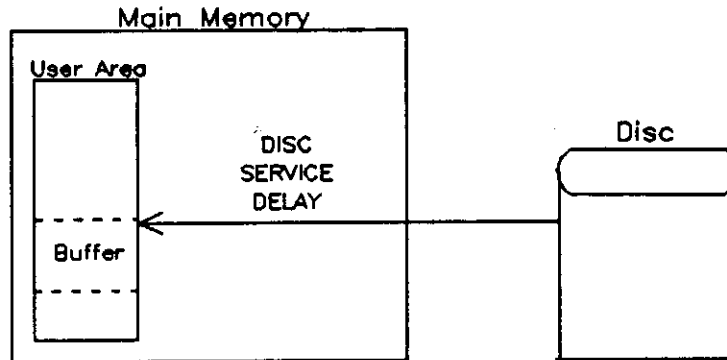
Figure 52: Bread-Board Internal Disc Cache Interface

6.6.1 Read Handling

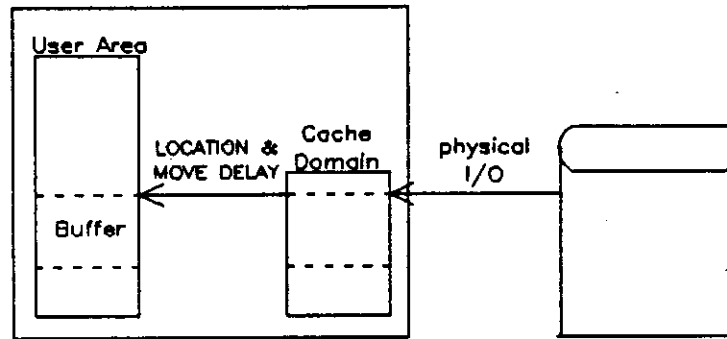
Figure 53 shows read processing with and without caching enabled. Without caching, the disc transfers directly to the buffer, but a disc access delay is incurred. With caching, the disc transfers to an area of memory reserved for the disc domain, then the data is moved to the buffer using the processor. On read hits, data access requires locating the data in memory and moving it, rather than having to incur a disc delay. This is performed on the current processes stack without a process switch.

READ HANDLING

UNCACHED READ



cached read



DCREAD

Figure 53: Disc Read Caching in the Bread-Board

6.6.2 Locating Cached Disc Regions

The cached disc domain location mechanism employed in the bread-board kernel is depicted in Figure 54. A separate list is maintained for each disc which identifies the memory regions corresponding to currently cached domains from the disc. The list is ordered by increasing disc address, and a microcoded link-list-search instruction is used to locate a required region in the list.

This location scheme requires about 500 instructions for setup and cleanup, plus 2 memory references and 2 compares for each cached domain in the list. Thus, the overhead of translation increases with the memory size. This is not a particularly good feature. Thousands of domains can be cached for each disc in a large memory, so the overhead of translation can become significant. In hindsight, more attention should have been paid to the location mechanism. Architectures supporting file mapping in virtual space eliminate this overhead altogether.

CACHE DISC DOMAIN LOCATION MECHANISM

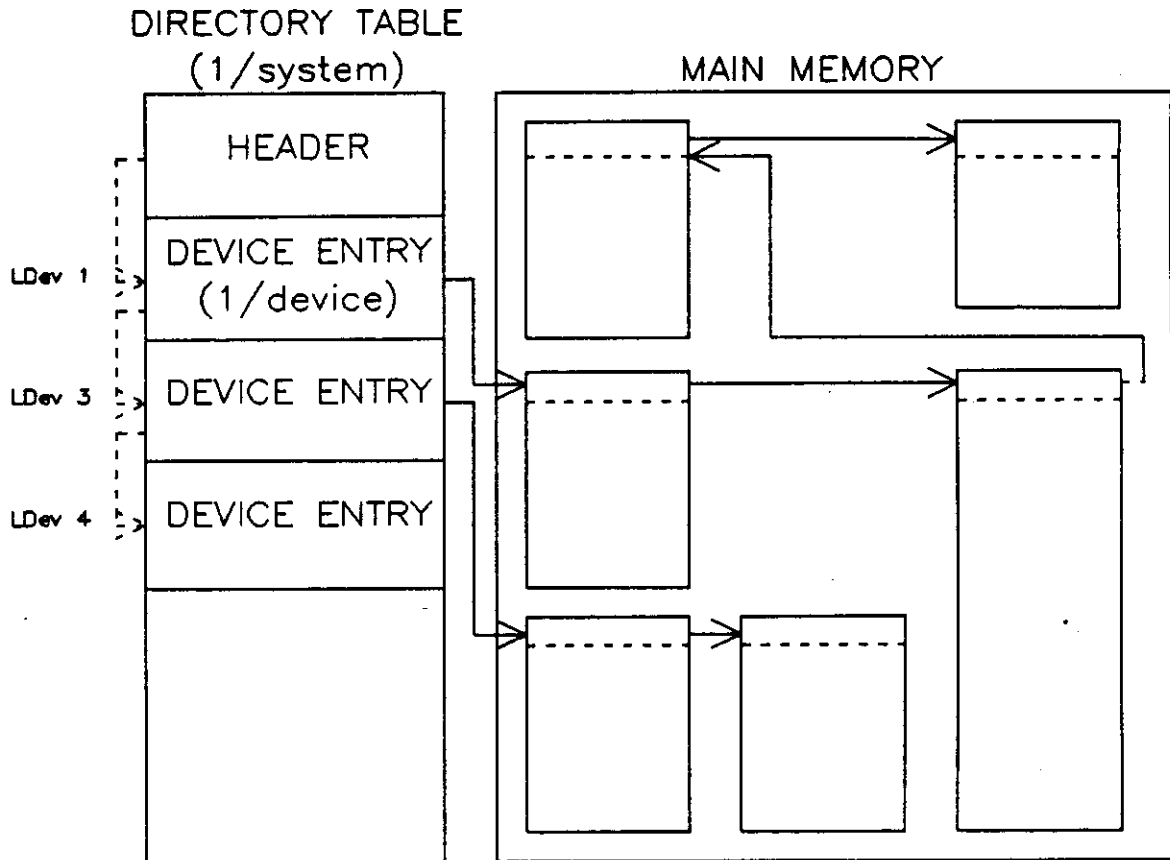
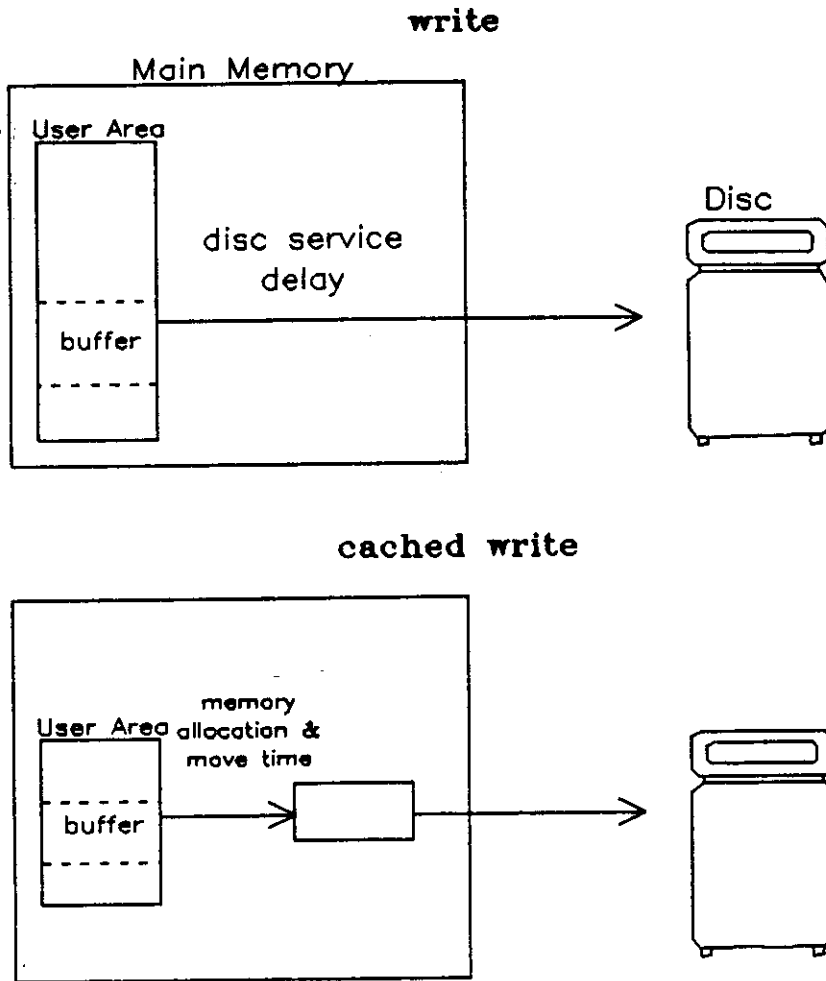


Figure 54: Cached Disc Domain Location Mechanism

6.6.3 Write Handling Mechanisms and Policies

Write handling with and without internal disc caching is shown in Figure 55. Without caching, the transfer occurs directly to the disc from the buffer. The transfer can be initiated with or without wait, with explicit completion synchronization occurring later. With internal disc caching, a free area of memory is allocated when the write is initiated, and the data is moved into that area. A write to the disc is issued, and the process can continue to execute. The write is considered complete once the memory move occurs unless the file has been specified as requiring physical post before completion notification.

WRITE HANDLING



dechw

Figure 55: Disc Write Caching in the Bread-Board

If there is currently a write pending against the specified disc domain, the process' request is queued until the pending write is posted to disc. If the disc domain to be written is not currently cached, an available region of memory is obtained which is used to map the corresponding disc image - i.e., no fetch of the disc domain to be written is required. When the move effecting the write takes place from the process' data area to the cached image of the disc, a post to the disc is initiated. Only the portion of the cached disc image which is modified by the write is posted. After the move to the disc image is performed and the post to disc is initiated, the writing process is allowed to continue running without having to wait for the physical post update to complete. This is all handled on the current process' stack, without even a process switch.

Precedency queues are provided to control posting order. Through the use of such queues, disc integrity can be assured without having to wait for physical posts to complete. Transaction recovery mechanisms employing write-ahead logging can initiate log posts, and, without wait, modify the permanent data using these precedency queues. This provides transaction recovery support with minimal delay.

Smith [Smi 79] discusses alternative write handling policies for processor caches, comparing swap policies (wait until space is needed before writing) and write through policies (write operations made immediately to the cache and memory simultaneously). He recommends write through over swap, with buffering of the writes to main memory.

A write through policy was chosen for our implementation of internal disc caching. The post request to disc is issued at a background disc priority. The priority of a pending post request is raised if the process waits for the post to complete or the region is required by the main memory replacement algorithm. Thus, issuing a physical post when the write is performed rather than waiting for replacement has no negative impact. Only idle disc capacity is used.

Performing write-through is also beneficial since the transaction recovery mechanisms require synchronization on physical commit at some point, so performing these early saves delays.

The write protocol is shown in Figure 56.

WRITE COMMITMENT PROTOCOL TO DISC

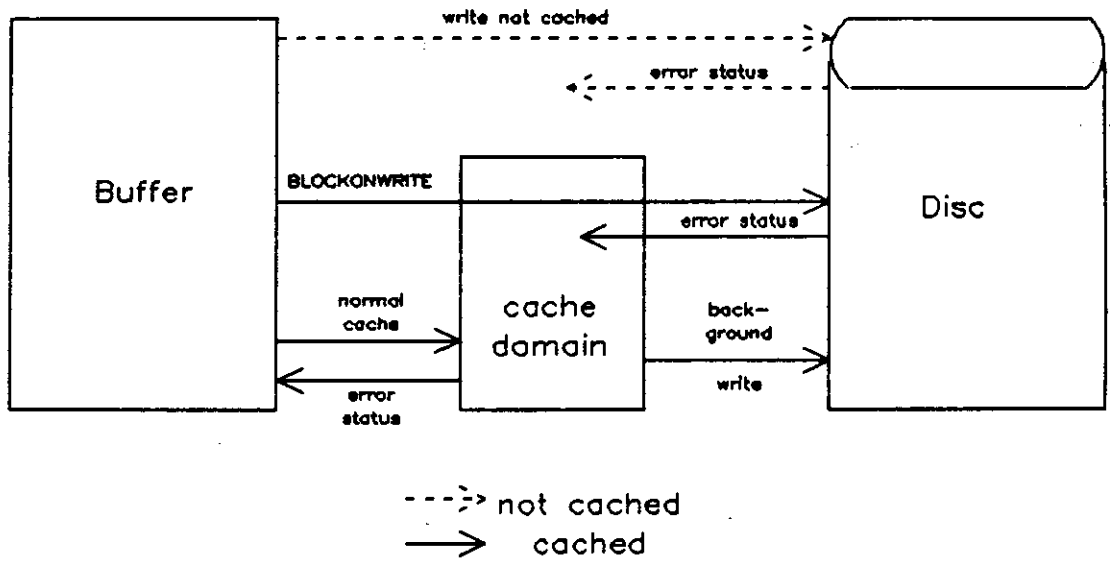


Figure 56: Internal Caching Write Protocol

In paged systems employing file mapping, a fetch of the data range from disc is required on subpage write misses. This fetch is not required with segmented architectures. The frequency of such write misses was found to be about 40% of the write attempts with the workloads observed. The write miss rate depends heavily on the storage algorithms employed in the access methods.

6.6.4 Cache Fetch, Placement and Replacement

The kernel's main memory placement and replacement mechanisms were extended to handle cached disc domains in the same manner as segments. Thus, cached disc domains can be of variable size, fetched in parallel with other segments or cached disc domains, garbage collected, and replaced in an integrated manner with stacks, data segments, and code segments. The relative allocation of main memory between stack, data, code and cached disc domain objects is entirely dynamic, responding to the workload's current requirements and current memory availability.

Fetching and replacing differ from processor caching of main memories. Rao [Rao 78] discusses the impact of replacement algorithms on cache performance for caching of main memory, finding that the replacement algorithm has a secondary effect on cache performance. Smith [Smi 75] discusses sequential prefetching of programs for processor caches.

With internal disc caching, when a request is made to read data which is not currently cached, the fetch strategy uses knowledge of the file blocking, extent structure, access method, and current memory loading to select the optimal size of disc domain to be fetched into memory. The fetch is performed in an unblocked manner so that the requesting process or another process can run in parallel with the cache fetch from disc. With processor caches, the processor must idle on a cache miss since the process switch time exceeds the cache miss processing time.

No special treatment of cached domains was required for the replacement algorithm. It naturally protects the transients objects in smaller memories due to their smaller inter-reference times, and uses large memories for extensive write and read caching with the relative amount of cached disc space per device skewing naturally to the heavily referenced devices.

Figure 57 shows the cache hit ratios across discs, the read to write ratio across the discs, and the partitioning of main memory allocated for cached domains across the discs. The normal LRU type replacement algorithm does fine in responding to the variable demand requirements of the various disc volumes.

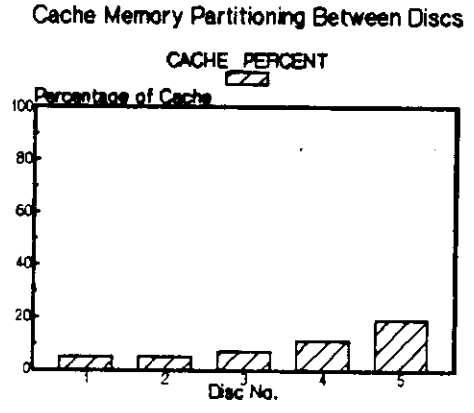
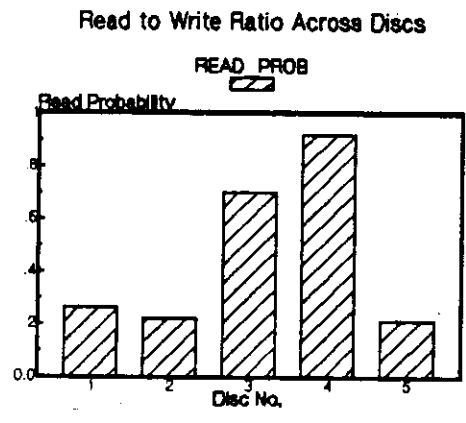
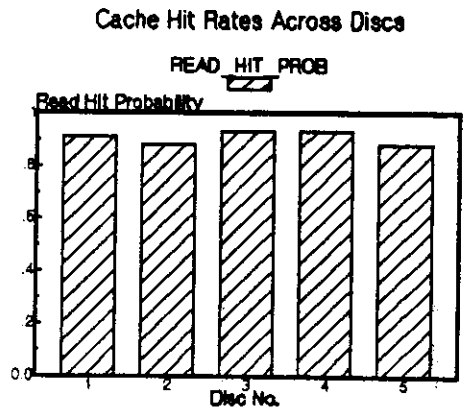


Figure 57: Dynamic Partitioning of Cache Across Discs

Explicit prefetching and flushing for sequential access was found to improve hit rates through simulation studies on standard trace files, whereas special prefetching and flushing for other access modes did not.

This policy was implemented in the kernel. When a process completes referencing a cached disc domain in sequential mode, the domain is flushed immediately from main memory since it won't be needed again. In this way, memory utilization is improved over that achievable with the kernel's standard LRU type replacement algorithm.

6.6.5 External Caching Controls

The external controls for the caching are shown in Appendix F. These allow caching to be enabled/disabled against specific discs, display the current status of disc caching, set posting policies on a system and file basis, and control the roundoff fetch sizes for random and sequential access.

Defaults for the tuning parameters were selected based on simulations of disc access traces using the simulation model. Good defaults for random fetch sizes were found to be 4 kbytes and for sequential 24 kbytes. Large prefetches were found to payoff big for sequential, but not for random type accesses. Rounding the fetching above the requested block was found to be superior for all access methods to fetching below the requested block or centering on the requested block. The choice of tuning parameters is, as always, an

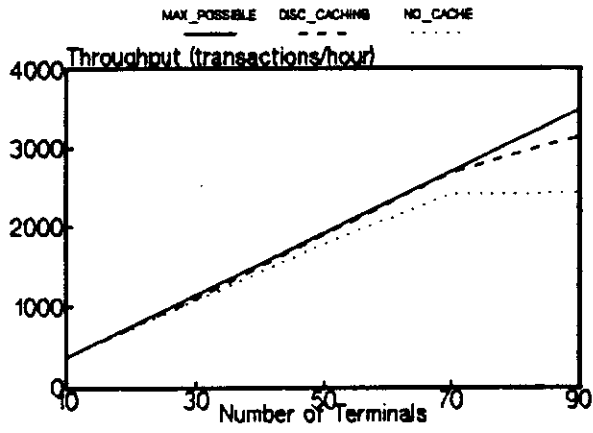
adjustment to the access patterns of the particular subsystems and databases.

6.6.6 Performance of the Bread-Board Caching

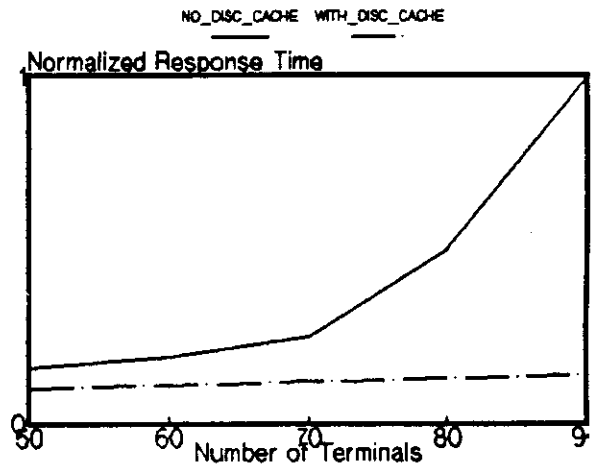
With these mechanisms and strategies, the bread-board kernel significantly reduced the traffic between the main memory and secondary disc storage and significantly reduced delays to read or write disc information. Read hit rates of 85% are common for file, database and directory buffer fills. These read hits eliminate roughly 65% of the disc accesses (5:1 read to write ratio). Due to the caching of writes, most delays for posting are eliminated. Together, the read hits and cached writes eliminate 90% of process delays due to disc accessing. This dramatically reduces semaphore holding times.

The impact on system performance over the non-cached kernel is shown in Figure 58. Throughput improvements of 50% and response time reductions of 5:1 are standard on the high-end (Series 64) while the mid-end gets about 25%, and the low end degrades, thus demonstrating the scaling of performance with processor speeds. However, the mid-range system with the kernel caching outperforms the high end machine without kernel caching of the discs.

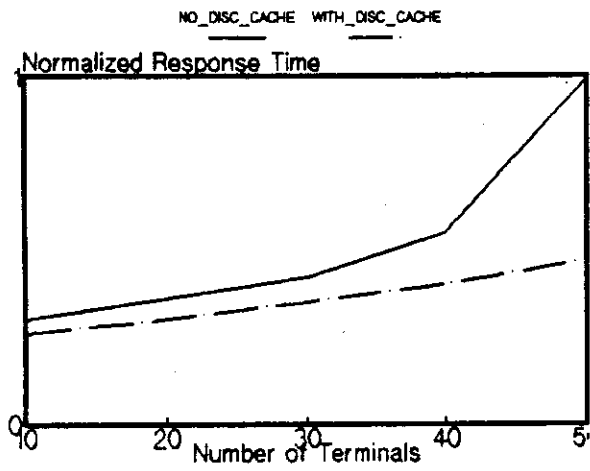
HP Series 68 Throughput



HP Series 68 Response Time



HP Series 48 Response Times



HP Series 48 & HP Series 68 Response Time
4Mb Main Memory

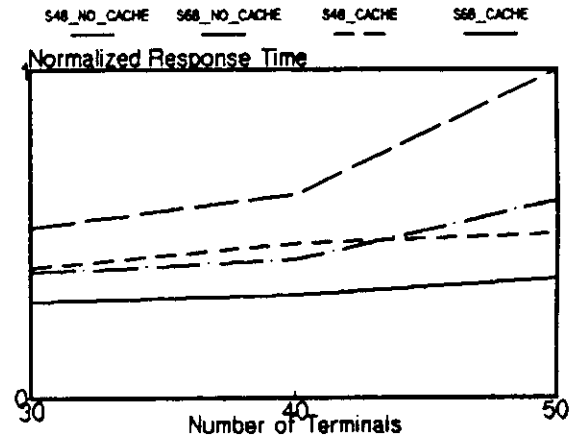


Figure 58: Kernel Caching Performance Impact

Figure 59 compares the kernel with and without caching for configurations with 4 125 Mbyte discs and 2 400 Mbyte discs. The cached system with 2 disc servers outperforms the uncached system with four disc servers. Moreover, the cached system performance is the same with 2 or 4 discs, thereby demonstrating that SPU caching can exploit the system cost advantages of large capacity discs.

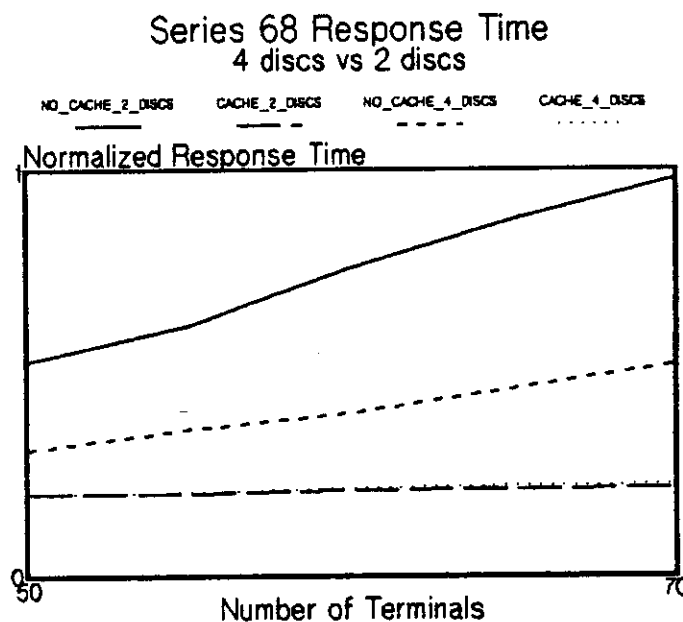


Figure 59: Effects of Multiple Discs and Disc Caching

A benchmark comparison [HP 83b] was made between an IBM 3033 (5 MIPS) and an HP 3000/64 (1 MIP) with and without the bread-board kernel applying disc caching in main store. The benchmark ran CompServ's Materials Request and Planning (MRP) software developed for both the HP 3000 and IBM 3033 systems. The benchmark consisted of a materials

request and planning batch job which performs a large database implosion in order to determine the parts, lead times, and orders to manufacture a specified set of end-products subject to a master production schedule. This type of an application tends to be highly I/O intensive since it accesses the database in an inverted manner, starting with schedules and searching for qualifying parts.

The benchmark ran in 28 hours on the IBM 3033, 49 hours on the HP 3000/64 without the kernel's caching of the discs in excess main store, and in 27.4 hours on the HP 3000/64 with kernel disc caching. That the HP 3000/64 could outperform the IBM 3033 in spite of the 5 x difference in processor speed indicates that caching by the kernel of disc domains in excess main store with the cache management policies introduced here rather than applying locally managed, limited caching of data items by database or file systems can be constructively applied to systems outside the case study family. Database and file system caching is limited to a fixed capacity and applies policies which optimize for the standard access approach. When more resources are available, as main memory for a stand-alone batch job, and access is non-standard, as in this inverted access case, localized caching policies do not respond. In small memory systems under memory pressure and with slow processors, disc caching degrades performance. The cost of locating the cached disc domain in main memory and moving the domain gets added to the disc access time. With low hit rates and slow processors this overhead exceeds the benefits of caching the discs. This overhead is not present in architectures supporting file mapping.

6.7. Conclusions

This chapter examined alternative approaches to exploit current technology trends in processors, discs, and semiconductor memory to realize significant improvements in system price/performance.

Caching secondary store in primary main memory through explicit internal caching or file mapping, coupled with integrated kernel and data management algorithms, was found to provide the most cost effective and best performing alternative.

External caching of discs by providing disc buffers or an intermediate storage level using CCD, bubble or semiconductor RAM technology was found to be inferior to internal caching of discs in the system processing unit using large primary memories based on cost, performance, and reliability measures. Difficulties in integrating external cache management algorithms with operating system knowledge, and having to cross the process switch boundary and access through the I/O system, causes the performance of external caching to be significantly less than that achievable with internal caching techniques. The additional cost of electronics, cooling, power, and cabinetry cause the cost of external caching to be greater. The added components cause the reliability to be less. However, external caching can be added to existing systems with the minimum of additional development.

Alternatives for the caching of discs in primary memory include local buffering, global buffering, explicit disc caching, and file

mapping. Local and global buffering techniques suffer from fixed partition problems, and require buffer management techniques which are redundant to the kernel's management of transient objects. Explicit disc caching and file mapping provide disc caching in a manner which scales with memory size and leverages the kernel management algorithms for main memory.

Secondary store cache management can be integrated with data management to obtain significant incremental performance improvements. Specific improvements were identified by integrating write-ahead logging with disc access scheduling through the adherence to post order constraints, bumping the priority of effected disc post requests at commit time, and using extent boundary and access type knowledge for selecting the fetch size on read misses and flushing on sequential access.

Utilizing high speed processors requires sustaining a high multiprogramming level. Improved concurrency control schemes, such as late binding, granular locking for updates and versions for queries, help achieve this. Database locking can be integrated with kernel resource management by bumping the priority of a lock holder when a more urgent process queues on its lock.

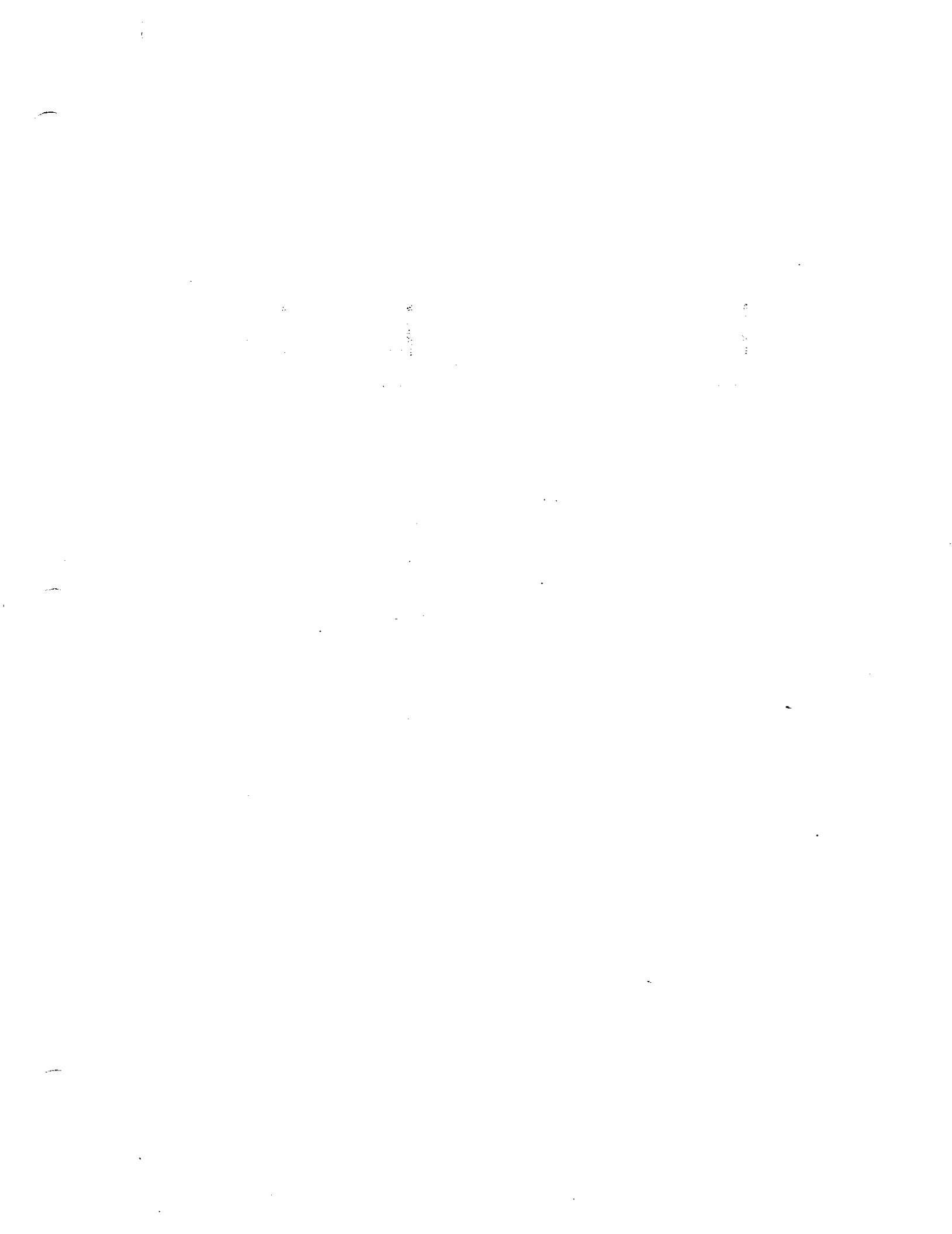
Providing caching of discs in main memory differs from caching main memory for processors in several ways. The delay in resolving a read miss is highly state dependent, requiring disc queueing and head positioning components. Since the read miss resolution time is long compared with the time to state save the current process and launch

another process, other ready processes can be run if the effective multiprogramming level is sufficiently high, allowing productive processor utilization while resolving read misses. Special knowledge of access patterns can readily be exploited to improve cache fetch and replacement strategies. Write misses can be resolved without requiring a fetch from the backup store if variable sized allocation is used, thereby thereby utilizing large caches to eliminate process waits on write misses. Special constraints exist on the write handling policies for updating modifications to the backing store due to the requirements of write-ahead logging and shadow paging in transactional systems.

The modelling techniques and algorithms for processor caches supply a base that is useful when considering other caching problems.

Intuition into the problems of secondary store caching was acquired through the bread-boarding effort, and the bread-board graphically demonstrates the results which the analysis predicts. The effective utilization of higher speed processors and reduced dependency on the number of discs was clearly demonstrated.

The bread-board demonstrates further that integrated internal caching of secondary store can be accomplished with conventional hardware and software architectures. Data management integration with internal cache management can be achieved in existing systems by providing special functions to allow subsystems and applications to influence the management of caching, disc scheduling, and main memory management so as to meet their performance and recovery objectives.



Chapter 7

Further Research

The scope of empirical and analytic research described in the preceding chapters was restricted to centralized computer systems. This chapter discusses applicability of the concepts in decentralized systems, and our current research in this area.

A major goal in decentralized systems is to provide transparent, controlled sharing of resources among the component computers. In order to share data among computers in decentralized systems, new problems of security, access control, data consistency, reliability and availability are introduced. Various approaches to providing shared access to files in decentralized systems have been proposed, some have been analyzed, and some have been built. Alternative configurations and algorithms differ in terms of their degrees of flexibility, transparency, performance, reliability, and availability.

We focus on examining the potential of our integration concepts for improving performance of transactional distributed configurations of current interest without degrading their availability, reliability, transparency or security. We examine configurations with hosts sharing a secondary storage subsystem, file servers, and workstations. We consider extensions of our integrated resource management algorithms to

improve the performance of these decentralized configurations by integrating scheduling and data concurrency, consistency and recovery management.

7.1 Integrated Resource Scheduling in Decentralized Systems

In the previous chapters we discussed the use of a global priority and assignment and adjustment algorithms as a means of integrating local resource management decisions in a centralized computer system. This mechanism appears to extend well to achieve integrated management of shared storage subsystems and shared file servers.

An example of an architecture targeted towards providing high speed, shared access to a common set of discs is DEC's HSC50 mass storage subsystem [Pla 83]. The shared storage subsystem manager services the requests for disc access from a number of hosts, deciding service priorities based on minimum seek time.

We discussed in previous chapters that, within a centralized system, significant performance improvements can be achieved by priority servicing of disc requests, where the priority assignments reflect current system urgency. Rather than having the storage subsystem locally optimize its throughput by applying a minimum-service-time-next policy, as is done in the HSC50, a protocol and service discipline can be employed which better supports system performance objectives. Priorities can be assigned in each system and disc service requests can be tagged with priorities in a manner similar to that forwarded for the secondary storage management in this research.

This scheme would require a global system concept of priority among the hosts connected to the storage subsystem. Such a requirement is natural and would be easy to implement, even if the host operating systems are non-homogeneous. The expression of performance objectives through a global tuning command and subsequent priority assignments and adjustments were performed with the algorithms forwarded in this research, the host/storage subsystem protocol would need to support status checking, cancellation, and priority adjustment of outstanding requests.

Some storage subsystems take on responsibility for file or database access and control. The Tripos filing machine [Ric 83] in the Cambridge Distributed Computing System [Mit 82] provides such a file server. It is accessed transparently through local file system stubs which execute remote procedure calls to invoke file system services. A small amount of local buffering is provided, with the filing machine itself performing extensive buffering.

Cheriton and Zwaenepoel [Che 83] describe a network of discless workstations built on SUN computers connected to file servers. All secondary storage is provided by back-end file servers, and general purpose IPC facilities are used rather than specially tuned file access protocols. They find that the performance is close to the lower bound given by the network penalty, so specialized protocols would provide minimal improvement.

Tagging service requests for file servers with global priorities could help to guide the file server's local management decisions on lock, processor, memory and disc scheduling local management decisions in the same manner applied in the centralized resource management examined in this research.

7.2 Integrated Data Management in Decentralized Systems

In the previous chapter we saw the significant advantages of extensive internal caching of secondary store using integrated data management algorithms to exploit current trends in processors and memories. When we consider applying these to decentralized systems in order to realize concomitant benefits, we must address the requirements of distributed transaction management and solve the problems of inter-computer cache coherency management.

Consistency and recovery mechanisms for distributed transaction processing have been given attention in recent years. The approaches taken to provide decentralized transaction consistency and recovery have a major impact on system performance in decentralized systems.

Traiger, Gray, Galtieri and Lindsay [Tra 79] define the concepts of transaction and data consistency for distributed systems. They show that consistency management for non-replicated data is amenable to the techniques used for centralized data. In particular, they show that if transaction execution is well formed (all data read during a transaction is shared locked, all data written in a transaction is write locked) and two-phase (all locks are held until the end of the

transaction), consistency is guaranteed. Applying this at each node through a local lock manager for objects residing at that node guarantees distributed transaction consistency. Transaction atomicity is guaranteed by following a two phase commit protocol where all nodes must agree to commit before the commit log record is written.

Reed [Ree 83] discusses distributed transaction management using intention lists which hide new values of data involved in a transaction by a version mechanism until the transaction commits. He presents a distributed synchronization mechanism consisting of a two-phase commit type protocol for implementing atomic commits. This approach provides an "update semantic on top of immutable versions."

Chan and Gray [Cha 84] discuss the use of old versions of data for read only transactions to minimize synchronization delays while still providing a consistent view of the database in distributed systems.

Gifford [Gif 83] describes synchronization for commit and abort processing in decentralized systems using transaction coordinators. He discusses the use of local shadow paging to save new, uncommitted values for transactions in progress until commit time. Recovery from conflicts on commit takes place by discarding the new values in the shadow pages, then restarting the transaction.

The Locus distributed file system [Wal 83] supports file replication and nested distributed transactions [Mue 83] in a network of VAX computers running homogeneous copies of an extended UNIX operating system. Shadow paging is used for transaction recovery. At commit

time the changes are first committed to one copy of the file then replicated to other copies through a centralized synchronization mechanism. They report that a substantial degree of "performance transparency" is achieved in the Locus network. Their local performance equals UNIX in the local case, and is "close enough" in the non-local case in a 5 computer network with 30-40 users without file replication or nested transactions.

Exploring alternatives to exploit the performance advantages of extensive internal caching of secondary store while allowing concurrent distributed execution of transactions presents a challenge that offers significant potential. The distribution of functionality between system components, and the structure and algorithms of subsystems and applications to accommodate and efficiently exploit such environments, can be expected to change from those of current systems.

Popek and Thiel [Pop 83] discuss distributed data management issues in the UCLA Locus distributed system. Popek concludes:

Repeatedly it appears that one can profitably avoid solving the same problem at multiple levels in computing systems. To do so however requires understanding the tasks to be accomplished at each level, so that commonality and restructuring can be accomplished.

We are currently researching functional partitioning and algorithms for integrated data management in decentralized transaction management systems. We are bread-boarding and developing analytic and simulation models to investigate these topics.

The bread-board configuration is shown in Figure 60. It consists of a set of shared discs accessible to a number of multiple hosts which synchronize their access to shared storage structures through a global shared memory (if available) or through explicit communications. This configuration allows incremental growth with shared files through the addition of homogeneous computers. The goal of resource management in this system is to provide near linear system performance increase with respect to total processor capacity. The problem is analogous to multiprocessors with a common shared main memory.

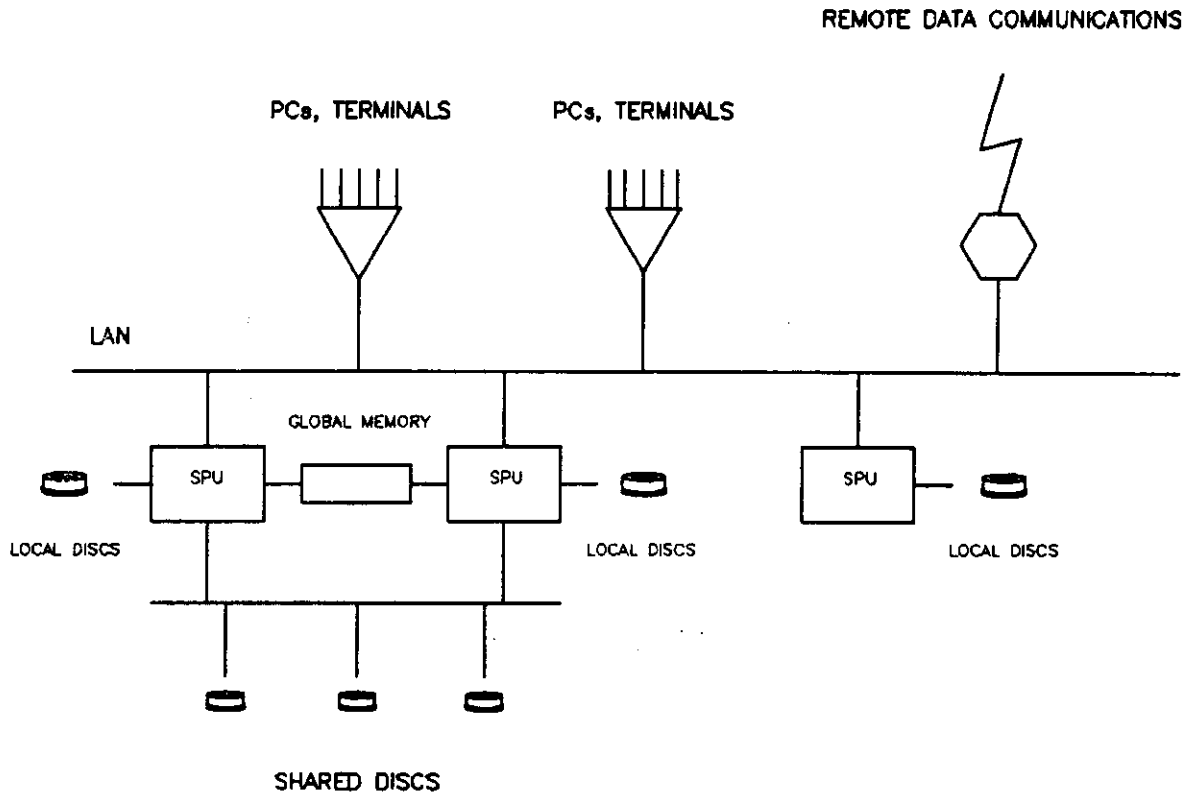
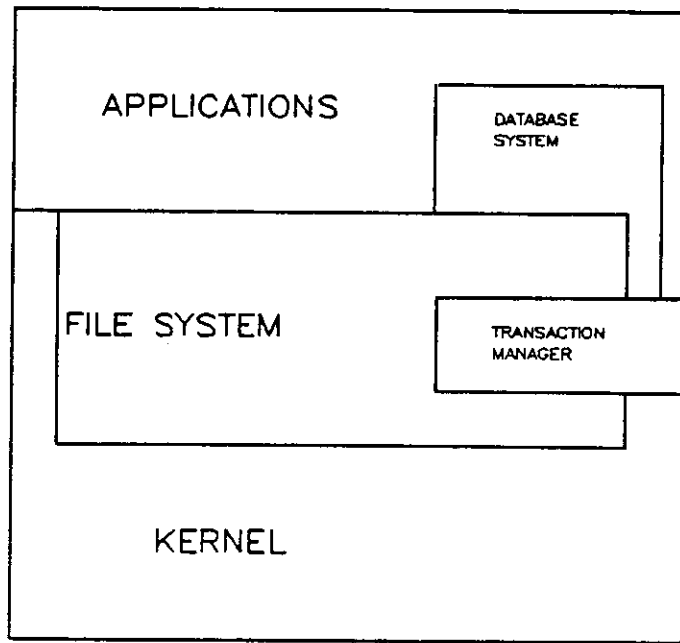


Figure 60: Current Research Configuration

The software architecture is shown in Figure 61. Responsibility for buffering and transaction consistency and recovery management resides beneath the database system and is performed by a cooperative effort of the file system, kernel, and processor traps. This architecture limits the problems of data coherency and decentralized locking to the lowest levels of the system where special components and algorithms can be used to minimize the costs imposed by decentralization.



SOFTWARE ARCHITECTURE

S200J85

Figure 61: Software Architectural Components

The processor architecture and the kernel provide access to shared distributed data at processor speed through file mapping and remote page faulting. Shared files are mapped into a common portion of the virtual space, while local files and transient code and data objects are mapped independently in the remainder of the virtual space. This is depicted in Figure 62.

VIRTUAL SPACE USE IN DATA SHARING CONFIGURATIONS

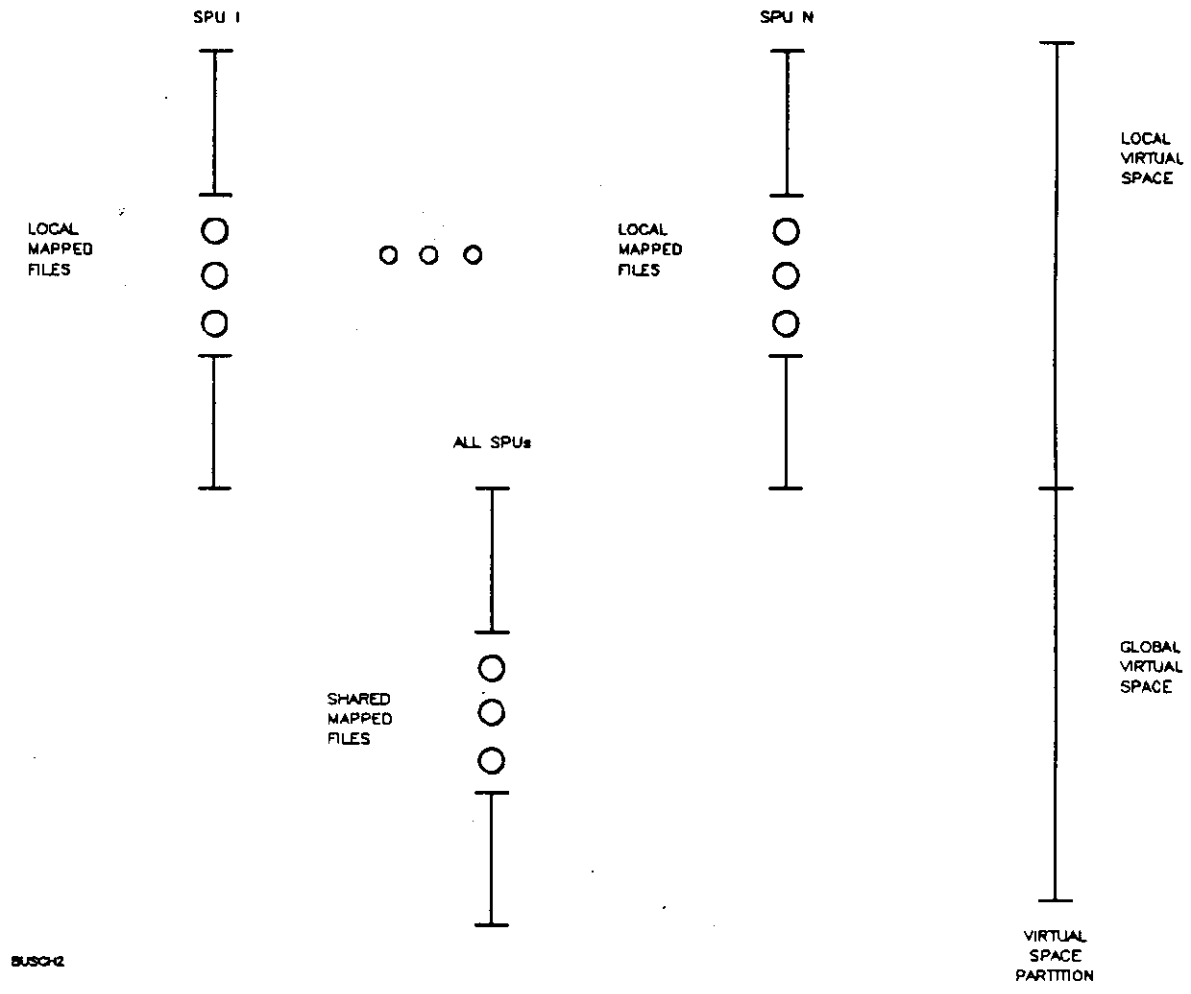


Figure 62: Shared/Local Virtual Space Partition

Multiple copies of pages are cached for read access, but only one copy exists when any computer has write access. This cache coherency management is enforced by the use of the read protect bit. To prevent writers from accessing a page when multiple copies are cached for read access, the read only bit of each copy is set to force a trap to the kernel on the first write access attempt. The kernel's distributed virtual storage manager then has the responsibility to decide when to invalidate other copies and when to allow the page to float to other computers.

On top of this cache coherency mechanism, the kernel provides locking on a virtual range basis with distributed queueing and deadlock detection and recovery. By decoupling locking from cache coherency in a manner similar to the use of semaphores above basic cache coherency in a multiprocessor system, cache coherency maintenance is invisible to data management. Further, since locking is performed by the kernel, decentralized queueing concerns are also transparent to data management. The kernel can exploit the configuration to perform these functions most efficiently.

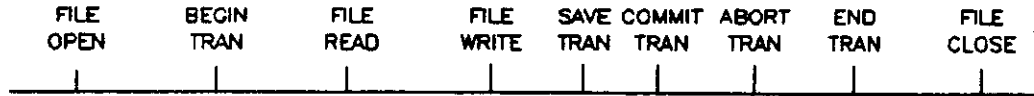
The file system uses the kernel locking mechanisms to share lock data read by transactions and to exclusive lock data written by transactions. The locking is performed implicitly beneath the access methods. This approach provides maximum concurrency since the binding is late and the locking is granular. Before and after images of modified data are logged to a local journal, with data modification

proceeding without wait due to adherence to posting order constraints between the log and database objects.

The resulting partition of functionality is shown in Figure 63.

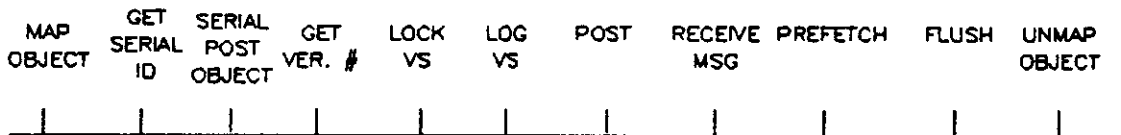
FUNCTIONAL PARTITION AND INTERFACES

DATABASE / APPLICATIONS
(INVOKE SERVICES)



FILE & TRANSACTION MANAGEMENT

(SECURITY, DATA CONSISTENCY
WITHIN TRANSACTIONS)



KERNEL

(ACCESS CHECKING, FILE MAPPING, STORAGE
MANAGEMENT, CACHED PAGE COHERENCY)

S206JB7

Figure 63: Bread-Board Partition of Functionality

With this partition of functionality and set of basic mechanisms, the maintenance of decentralized data coherency and consistency is entirely transparent to the file and database subsystems. The algorithms for managing locking and inter-computer secondary store cache coherency are encapsulated.

Stonebraker [Sto 83] proposes a split in functionality for a centralized system using file mapping which migrates locking and logging responsibility to the very lowest levels of the system. In Stonebraker's proposal, locking and logging are performed implicitly as a process involved in a transaction loads and stores from its address space. A process' first read reference to a page since the last commit increments a shared lock count, and its first write reference to a page forces a trap to log the current version of the page and obtain an exclusive lock for it. Read references to exclusively locked pages and write references to shared lock pages generate interrupts which perform deadlock detection and queue the process for the page. A vector of accessed and modified page bits is maintained for each process, and at commit time all modified pages are forced to the disc and all accessed and modified pages are unlocked for the process.

Requiring microcode or hardware assist for locking/logging complicates the hardware, resulting in an increased basic cycle time. The invocation of lock and logging services beneath the file access methods is low overhead, requires no additional hardware support, and provides transparency of distributed locking and cached secondary store coherency.

When examining alternative algorithms for cache coherency control in decentralized configurations, we can leverage off of the work done for multiprocessor systems with private caches, since this presents an analogous problem. However, the relatively long interconnection delays in decentralized configurations employing secondary store caching can

be expected to change the tradeoffs between alternative cache management algorithms from those found in multiprocessor systems. Such differences were found in the last chapter when comparing processor cache management with secondary store cache management.

Censier and Feautrier [Cen 78] discuss alternatives for maintaining cache coherency in multicache systems. One alternative is to have each cache broadcast the identification of any cache block which is to be modified so that the other caches can invalidate local copies. This approach generates intercache traffic even if only one copy of the data exists. A second approach which eliminates this unnecessary traffic is to maintain a private flag with each cache block so that intercache traffic is only introduced when there is a genuine conflict. If a mask of processors containing a data block is maintained either in the main memory or redundantly in each cache containing the block, invalidation requests and fielding of requests need only be sent and serviced by caches directly involved in the conflict.

The ELXSI multiprocessor system [Sha 83] uses software convention to maintain cache coherency. All shared data is delivered through messages, while semaphores used to control message queueing are not cached.

The Synapse multiprocessor system [Fra 84] uses a distributed ownership protocol which allows data to shift between caches.

Variations on these approaches can apply in decentralized systems as well. The impact of the interconnect delays and required traffic patterns influence the algorithmic choices.

Bitmasks indicating which computers currently have copies of pages and locking structures can be stored in a shared global memory. Then inter-computer cooperation is required only on lock conflicts or when writes conflict with other cached copies of a page. In the non-conflicting cases, each computer achieves the performance of the integrated data management system employing internal secondary store caching through file mapping described in the last chapter.

Since the performance of such a system is dependent on lock and write conflict frequencies, we expect granular locking, unstructured storage of permanent data as in relational systems, and dispersed storage of shared management structures to contribute significantly towards approaching the performance goal.

Without a shared global memory, explicit communication between the distributed virtual space and lock managers is required to maintain cache coherency and enforce decentralized locking.

We can leverage the techniques used in analyzing multiprocessor systems to analyze the alternatives to achieve decentralized caching of secondary store.

Sauer and Chandy model the impact of processor service distributions and scheduling algorithms on multiple processor systems in [Sau 79]. They used a cyclic queue model with common I/O queues and a central server model with multiple I/O queues in their analysis. They compute the throughput ratios of single and multiple processor systems as a function of degree of multiprogramming, interference level, and

preemptive scheduling policies. They find that sophisticated processor scheduling algorithms buy relatively little in multiple processor configurations.

Patel [Pat 81,82] analyzes multiprocessors with private cache memories using an approximate analytic model. His model computes performance measures including processor utilization, average wait time, and memory traffic with and without store through and load through policies. His model does not however capture the impact of write conflicts between the caches.

Shedler and Slutz [She 76] discuss the use of merged access streams of multiple programs to predict hit rates in multiple processor systems. Logical reference streams from multiple computers could be used to predict (and isolate) conflict frequencies, as well as to predict hit rates.

Goldberg, Lavenberg, and Popek present a queueing model of a local area distributed system in [Gol 82]. Terminals, processors, discs and data network service centers are supported, with processor sharing discipline for the processors, and first come, first served discipline for the discs and the network. Background activity such as file replication is approximated. A closed product form network solution is employed. Their model was validated using Locus, with measurements tracking predictions until non-local access exceeds 40%.

We are building trace driven simulation models to predict conflict and hit rates for alternative data storage and cache coherency

approaches. We are building analytic models which capture the significant configuration and workload characteristics, hit and interference rates, and overhead of remote operations so that alternatives can be compared with current and new technologies.

7.3 Conclusions

Several of the concepts and algorithms for integrating resource management in centralized systems developed in this research appear to apply well in decentralized system problems.

The use of our global priority assignment and adjustment algorithms to reflect system urgency in local decisions can be used in a shared disc server to schedule disc accesses and in a file server in lock, processor, memory, and disc access scheduling.

Our current research is directed at determining architectures, functional partitions, and integrated algorithms which allow the benefits of integrated data management and internal secondary store caching to be realized in decentralized systems. Mapping shared files into virtual space, providing multiple system cache coherency of cached file pages through write protect traps, and providing logging and locking implicitly through access method invocation of distributed kernel services on a late binding, granular basis, appears to offer good potential of achieving this goal. We can leverage the algorithms and analysis techniques used for multiple processor systems in researching this problem.

Chapter 8

Conclusions

Integrating resource management algorithms is a never-ending, always expanding problem. It follows the evolution of system components as technology improves, as new uses of computer systems come into existence, and as innovative concepts for new ways to build computer hardware and software architectures are introduced.

This research effort used a conventional computer system family in order to perform a case study in integrated algorithms. The bread-board constructed for this research effort was used to figure out the main interactions between the algorithms and some unifying principles for them. Interesting behavior of some known algorithms was uncovered. Some new integrated algorithms were proposed, implemented and analyzed which appear to have the potential of providing improved performance in other centralized systems. Some of the principles and algorithms appear to extend well to distributed systems as well. Our follow-on research is directed towards investigating architectures, functional partitioning, and algorithms for realizing concomitant benefits in decentralized systems.

The research proceeded by building an operating system kernel and performance tools that were designed for change. Algorithm interactions were observed, and improved algorithms were developed to optimize system performance objectives.

Resource managers were constructed independently of each other, and internally structured from primitives which make minimal assumptions on eventual service policies. The system was designed to be measurable under change by keeping the algorithm implementations, the measurement subsystem, and the display and analysis tools independent of each another.

This method of constructing the operating system and performance tools to accommodate changes in algorithms did prove useful in examining alternative strategies. Algorithms were easy to change and evaluate. The performance and development cost impacts of designing for change and measurability were minimal.

The major performance benefits came from getting the right split in functionality, providing parallel service where there is contention, and having the algorithms cooperate to optimize system, not local, objectives. Substantial second order improvements were realized through subsequent, appropriately selected local optimizations.

Designing for change, designing for measurability, obtaining a careful functional split, providing parallel, cooperating algorithms, and performing local optimizations only after global optimization has been achieved are all principles which can be applied when building new

systems so that they achieve their performance potential and evolve as technology improves.

Workload characteristics change as processor and system architectures change and as the algorithms employed in the applications, compilers, databases, file systems and kernels change. Algorithms need to accommodate the workload characteristics. Further, workload characteristics can be exploited to select and tune algorithms, resulting in significant performance impact for a given system and workload.

For algorithms to cooperate towards system objectives, they need on-going hints on service priority, and they need to make local decisions unselfishly.

The algorithms introduced here for expressing urgency and assigning, adjusting, and communicating priorities in support of the global objectives, provide a simple integrating factor for cooperative resource management which appear to extend well to decentralized resource management.

The need to strive for higher objectives impacts algorithm structure. Being prepared to change internal service priorities in a responsive manner, and to submit to global priorities which limit what is locally achieved, appear repeatedly. In the research, this was seen with disc access scheduling, main memory allocation, garbage collection and replacement.

Efficiently supporting broad ranges of workload demand and resource capacity calls for hybrid algorithms which exploit resource availability tradeoffs. Low cost algorithms win big when resources are abundant, but more careful algorithms payoff when demand is heavy. This was seen especially with main memory placement and replacement algorithms where fast but sloppy or elegant, sophisticated algorithms can cost considerably more than the benefit they bring.

In segmented systems, variants of the MULTICS clock algorithms have distinct advantages over working set type replacement algorithms due to their localized rather than scattered hole creation characteristics. Local compaction during replacement compounds this effect. Background garbage collection which operates globally to make large holes larger by moving small adjacent assigned regions to small holes improves the free space distribution during otherwise idle time. The algorithm supplements well a good replacement algorithm, but degrades the performance of poor replacement algorithms due to overlaying replacement candidates which could be recovered.

A major limiting factor in many systems is the disc access speed, and improvements in secondary store technology are unlikely to even keep up with processor speed advancements. Integrated algorithms for extensive secondary store caching in the primary memories provides the best solution for balancing the memory hierarchy with the current trends in processor, memory and disc technologies.

Centralizing caching of discs in the primary store managed in an integrated manner by the kernel, rather than distributing it between subsystems or providing it in peripheral devices, provides the best memory utilization, fastest access to the data, and the best performance for the posting requirements in transactional database systems. Such internal secondary store caching, when coupled with improved integrated methods of concurrency control, can allow us to realize the advantages of processor and memory technologies to deliver significant system level price/performance improvements.

The performance of systems employing extensive secondary store caching can be further improved by having the access methods pass hints on storage structure and reference patterns.

Exploiting the performance advantages of integrated internal secondary store caching is desirable in distributed systems as well. Simulation and analytic modelling will help to determine hit and interference rates for alternative multi-cache management algorithms and to evaluate them against current and new technologies. Bread-boarding will help to discover things our models and intuitions miss.

Extensive secondary store caching supported through file mapping and remote page faulting coupled with transaction consistency control on dispersed data through late binding, granular locking could significantly reduce communication traffic and delays required to support efficient execution of concurrent transactions in homogeneous decentralized systems. Our follow-on research is directed at

investigating architectures, functional splits and integrated algorithms for such systems.

Appendix A

Measurement Subsystem Specification

The measurement subsystem consists of data structures, mechanism control procedures, and data access procedures. Kernel code knows about the current specification of the interface, and supports its current definition by gathering and storing the supported information into the appropriate buffers. The information is gathered as events occur, or during sampling or trace interrupts.

Statistics Updating Procedure

```
Procedure UPDATESTATISTICS(Class,Subclass,Subclassentry,Startingitem,  
    Newvalueflag,Valuechange,Doubleitemflag);
```

Interface Control Procedures

```
Procedure STARTSTATISTICS(Classmask);
```

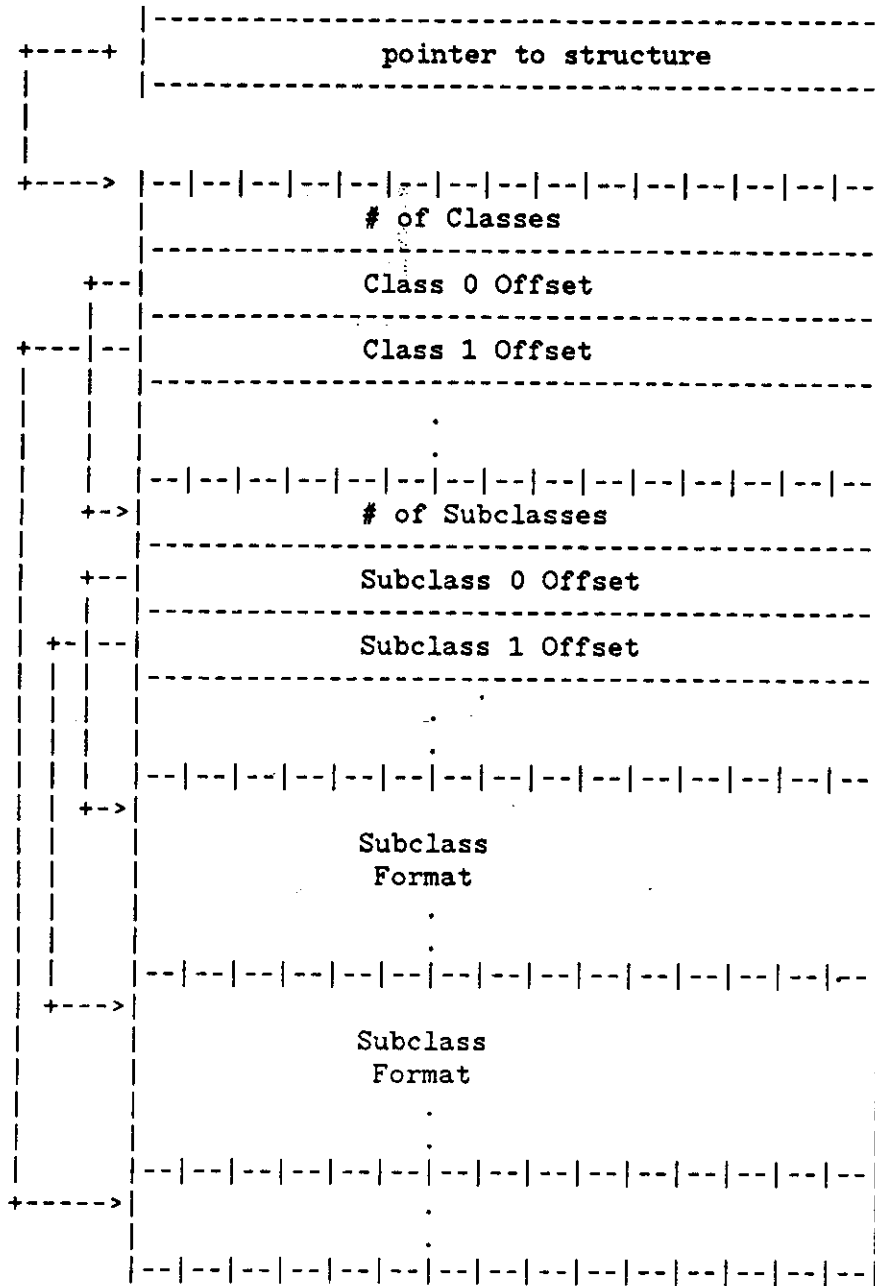
```
Procedure STOPSTATISTICS(Classmask);
```

Interface Access Procedures

```
Procedure GETSTATISTICS(Class,Subclass,Startingitem,Wordcount,Where));
```

Information Structures For Measurement Storage

Mechanism Global Structure



Appendix B

Instrumentation Formatting

LOG FILE IDENTIFICATION: dag cache on
LOG FILE CREATION: THU, DEC 22, 1983, 1:45 PM
DATE OF DATA REDUCTION: WED, DEC 28, 1983, 10:14 PM

PROCESS LAUNCH AND STOP INFORMATION

EVENT IDENTIFICATION	EVENT CNT	RATE/SEC	PROB
-----	-----	-----	-----
PROCESS LAUNCH	1880	6.3	100.0%
PROCESS PREEMPTION	233	0.8	12.4%
STOP: SL CODE SEG FAULT	5	0.0	0.3%
STOP: PROG FILE SEG FAULT	3	0.0	0.2%
STOP: DATA SEG FAULT	64	0.2	3.4%
STOP: CACHE DOMAIN FAULT	0	0.0	0.0%
STOP: BLOCKED DISC I/O	137	0.5	7.3%
STOP: UNBLOCKED DISC I/O	75	0.3	4.0%
STOP: TERMINAL READ I/O	26	0.1	1.4%
STOP: TERM I/O (NON-READ)	33	0.1	1.8%
STOP: MISC BLOCKED I/O	280	0.9	14.9%
STOP: BUSY SIR	0	0.0	0.0%
STOP: IMPEDED	156	0.5	8.3%
STOP: QUANTUM EXPIRATION	9	0.0	0.5%
STOP: STACK OVERFLOW	11	0.0	0.6%
STOP: PCBX EXPANSION	2	0.0	0.1%
STOP: DL AREA EXPANSION	0	0.0	0.0%
STOP: DB-Z EXPANSION	1	0.0	0.1%
STOP: DATA SEG EXPANSION	5	0.0	0.3%

MEMORY MANAGEMENT RELATED INFORMATION

EVENT IDENTIFICATION	COUNT	RATE/SEC
-----	-----	-----
PROCESS SWAP-IN	373	1.2
MEMORY ALLOCATION	381	1.3
SL CODE SEG RECOVERY	0	0.0
PROG FILE SEG RECOVERY	0	0.0
DATA SEG RECOVERY	0	0.0
CACHE DOMAIN RECOVERY	0	0.0
SEG RECOVERY (SWAP-IN)	0	0.0
SEGMENT IN MOTION IN	2	0.0
FOUND FREE SPACE	340	1.1
MAKEROOM SUCCESSFUL	27	0.1
DEFERRAL: SWAP-IN FAILED	0	0.0
GIVE-UP: MORE URGENT ACT	6	0.0
HARD REQUEST SUCCEEDED	0	0.0
MEM MGMT CLOCK CYCLE	0	0.0
CODE SEGMENT RELEASED	1	0.0
DATA SEGMENT RELEASED	2	0.0
CACHE DOMAIN RELEASED	266	0.9
SEGMENT LOCK REQUEST	1	0.0
SEGMENT FREEZE REQUEST	0	0.0
PCBX AREA CONTRACTION	0	0.0
DL AREA CONTRACTION	0	0.0
DB-Z CONTRACTION	8	0.0
DATA SEG CONTRACTION	3	0.0

TIME IN CPU STATE INFORMATION

CPU STATE DESCRIPTION	TIME (SEC)	% OF INTERVAL
BUSY: PROCESSES	34.421	11.5%
BUSY: SEGMENT MEM ALLOC	1.999	0.7%
BUSY: BKGRND GARBAGE COLL	0.000	0.0%
BUSY: CACHE MGT-PRCS STK	7.407	2.5%
BUSY: CACHE MGT-ICS	0.325	0.1%
PAUSED: USER/CACHE I/O	4.405	1.5%
PAUSED: SEG SWAP I/O ONLY	0.345	0.1%
PAUSED: USER/CACHE * SWAP	0.000	0.0%
PAUSED: CACHE DOMAIN SWAP	0.000	0.0%
PAUSED: IDLE	256.014	85.4%
GARB COLL AND MEM ALLOC	0.338	0.1%

CPU/DISPATCHER ACTIVITY INFORMATION

EVENT IDENTIFICATION	COUNT	RATE/SEC
PAUSE: USER/CACHE I/O	211	0.7
PAUSE: SEG SWAP I/O ONLY	11	0.0
PAUSE: USER/CACHE * SWAP	0	0.0
PAUSE: IDLE	463	1.5
BKGRND GARB COLL ATTEMPT	0	0.0
GARB COLL GIVE-UP	0	0.0
BKGRND GARB COLL MOVE	0	0.0
MEM ALLOC GARB COLL MOVE	17	0.1
CACHE DATA MOVE	1175	3.9

CACHE HIT / IO ACCESS ACTIVITY

EVENT IDENTIFICATION	COUNT	RATE/SEC
I/O READ ATTEMPTS	884	2.9
CACHE READ HITS	783	2.6
I/O WRITE ATTEMPTS	291	1.0
CACHE WRITE HITS	112	0.4
READ HITS / READ ATTEMPTS	0.89	
WRITE HITS/WRITE ATTEMPTS	0.38	
CACHE HITS / IO ATTEMPTS	0.76	

DISC ACTIVITY INFORMATION

DISC 1

DISC ACTIVITY DESCRIPTION	CUMULATIVE	
	COUNT	RATE
CODE SEGMENT READ	6	0.0
DATA SEGMENT READ	15	0.1
CACHE DOMAIN READ	40	0.1
SEGMENT WRITE: BACKGROUND	0	0.0
SEGMENT WRITE: FORCED	1	0.0
DOMAIN WRITE: BACKGROUND	9	0.0
DOMAIN WRITE: FORCED	1	0.0
BLOCKED READ	0	0.0
UNBLOCKED READ: NO AWAKE	0	0.0
UNBLOCKED READ: AWAKE	0	0.0
BLOCKED WRITE	23	0.1
UNBLOCKED WRITE: NO AWAKE	0	0.0
UNBLOCKED WRITE: AWAKE	0	0.0
CONTROL OPERATION I/O	3	0.0
ABSENT BUFFER TRAP	0	0.0
QUEUE ON BUSY CONTROLLER	10	0.0
ALL USER I/O READS	0	0.0
ALL USER I/O WRITES	23	0.1
ALL USER I/O	23	0.1
ALL TRANSIENT OBJECT READS	21	0.1
ALL TRANSIENT OBJECT WRITES	1	0.0
ALL MEM MGMT I/O	22	0.1
ALL CACHE I/O READS	40	0.1
ALL CACHE I/O WRITES	10	0.0
ALL CACHE I/O	50	0.2
ALL I/O READS	61	0.2
ALL I/O WRITES	34	0.1
ALL I/O	95	0.3

QUEUE LENGTH DESCRIPTION	CUMULATIVE	
	COUNT	%
QUEUE LENGTH: EMPTY	94	95.9%
QUEUE LENGTH: 1	4	4.1%
QUEUE LENGTH: 2	0	0.0%
QUEUE LENGTH: 3	0	0.0%
QUEUE LENGTH: 4	0	0.0%
QUEUE LENGTH: 5	0	0.0%
QUEUE LENGTH: > 5	0	0.0%

Appendix C

Disc Workload Characterizer

The disc workload analyzer program queries for a string to identify the run, the name of the disc access trace file, and name of the file to which the statistical report should be outputted. A description of the output and a sample output follows.

Reference Breakdown

Breaks down the disc access trace into access class and function. The reference type column breaks the access trace first into overall, read and write, then into image, ksam, directory, message, other, sequential, or direct, then into each of these access modes independently for read and write. The # Occurrences column gives the number of trace records found corresponding to the reference type, and the Prob column gives the fraction of total disc reference in the trace which the reference type represents.

Transfer Size Distribution

Histogram and moments of the transfer sizes of all non memory mgt disc accesses in the trace file.

Inter Disc Reference Interval Distribution

Histogram and moments of the time between disc accesses experienced in the trace. This indicates how busy and how bursty the disc usage for file accesses was during the period of tracing.

Extent Size Distribution

Histogram and moments of the size of the extents that were active during the trace period.

Inter Extent Reference Interval Distribution

Histogram and moments of the time between reference to the same extent. This measures temporal and spatial locality of reference to the disc. A mean comparable to the mean inter disc reference interval indicates that the same extent is referenced closely in time so that by caching a large part of the extent in memory and keeping it there for a while would result in read hits and thereby avoid disc accesses.

Stack Depth Distribution of Inter Extent References

Histogram and moments of the lru stack depth of interextent references. The extents are kept in an lru stack, and at each reference, the referenced extent is moved to the head of the stack and the histogram of its old location is bumped. The mean lru stack depth gives a measure of the locality of references to extents, with a small mean lru stack depth (<10) indicating that extents are typically repeatedly referenced in close time proximity, and that caching extents on an lru basis will be effective in catching most of the references.

Sample Output

*** Disc Workload Characteristics ***

Reference Breakdown :

Reference Type	# Occurrences	Prob
any ref	53000	1.00
read ref	36773	.69
write ref	16227	.31
image ref	0	.00
ksam ref	0	.00
dirc ref	9886	.19
msg ref	3617	.07
other ref	11341	.21
seq ref	12646	.24
direct ref	15510	.29
image read	0	.00
ksam read	0	.00
dirc read	8535	.16
msg read	3617	.07
other read	5333	.10
seq read	6677	.13
direct read	12611	.24
image write	0	.00
ksam write	0	.00
dirc write	1351	.03
msg write	0	.00
other write	6008	.11
seq write	5969	.11
direct write	2899	.05

*** Disc Workload Characteristics (ctnd) ***
 Transfer Size Distribution :

Transfer Size (bytes)	# Occurrences	Prob
0	813	.02
128	5258	.10
256	14924	.28
384	1623	.03
512	1333	.03
640	20	.00
768	11086	.21
896	1382	.03
1024	6147	.12
1152	685	.01
1280	6643	.13
1408	3	.00
1536	71	.00
1664	2	.00
1792	377	.01
>= 1920	2633	.05

Samples = 53000
 Distribution Mean = 784.51 (bytes)
 Distribution Std Dev = 1110.66 (bytes)
 Histogram Median = 768 (bytes)

Inter Disc Reference Interval Distribution :

Inter Ref Time (ms)	# Occurrences	Prob
0	5394	.10
10	18438	.35
20	7521	.14
30	4119	.08
40	3285	.06
50	2282	.04
60	1674	.03
70	1519	.03
80	1084	.02
90	884	.02
100	665	.01
110	543	.01
120	484	.01
130	377	.01
140	376	.01
>= 150	4355	.08

Samples = 53000
 Distribution Mean = 64.31 (ms)
 Distribution Std Dev = 279.42 (ms)
 Histogram Median = 20 (ms)

*** Disc Workload Characteristics (ctnd) ***
 Extent Size Distribution :

Extent Size (sectors)	# Occurrences	Prob
0	1267	.75
16	149	.09
32	61	.04
48	60	.04
64	31	.02
80	2	.00
96	7	.00
112	13	.01
128	15	.01
144	10	.01
160	9	.01
176	1	.00
192	0	.00
208	5	.00
224	0	.00
>= 240	70	.04

Samples = 1700
 Distribution Mean = 44.72 (sectors)
 Distribution Std Dev = 169.78 (sectors)
 Histogram Median = 0 (sectors)

Inter Extent Reference Interval Distribution :

Inter Ref Time (ms)	# Occurrences	Prob
0	19781	.39
50	6013	.12
100	2652	.05
150	1680	.03
200	1081	.02
250	1100	.02
300	1129	.02
350	841	.02
400	662	.01
450	551	.01
500	487	.01
550	484	.01
600	455	.01
650	431	.01
700	326	.01
>= 750	12821	.25

Samples = 50494
 Distribution Mean = 26318.89 (ms)
 Distribution Std Dev = 158527.06 (ms)
 Histogram Median = 50 (ms)

*** Disc Workload Characteristics (ctnd) ***

Stack Depth Distribution of Inter Extent References :

LRU Stack Depth	# Occurrences	Prob
0	34025	.67
5	4815	.10
10	2434	.05
15	1319	.03
20	987	.02
25	675	.01
30	511	.01
35	413	.01
40	359	.01
45	240	.00
50	270	.01
55	240	.00
60	225	.00
65	234	.00
70	196	.00
>= 75	3551	.07

Samples = 50494
 Distribution Mean = 28.82
 Distribution Std Dev = 112.17
 Histogram Median = 0

Appendix D

Disc Cache Simulator

The disc cache simulator simulates various strategies for caching pieces of the disc referenced by the access methods in excess main memory. The simulation is instrumented and analysis is performed to determine the behavior and performance of the specified caching strategy.

Input

Inputs to the program are disc reference trace files obtained from trace calls placed in the operating system and run on representative workloads.

The user selects cache sizes and strategies. Under user control are : the subset of access modes to be cached, the cache size range, the fetch size round off policy, the flush policy, and wait for post control.

The default fetch policies are :

Round Off Above Request (on a disc cache miss, we fetch from the disc the larger of the requested size or the size specified in fetch size for sequential or random. We fetch it starting at the requested location as opposed to rounding around the requested

location or rounding down to end at the end of the requested block).

Stop At Extent Boundary in Round (When we fetch on a cache miss, we normally fetch more than requested. We stop however at the extent boundary of the related request.)

No Fetch on Write Miss (On a write attempt to a disc domain which is not currently cached, we just grab a hole in memory, write into it, and fire off a disc write. There is no need to fetch the domain before we write).

The default replacement policies are as follows :

Flush After Seq Ref (On sequential access, after the cached portion of the effected extent has been completely used (ie last block referenced), we make the cached block available to the replacement algorithm since the likelihood of the domain being needed again soon is low since it is being sequentially referenced. If referenced again soon, it still has a chance of being recovered.

Output

The characteristics of the reference trace and the performance and behavior of the cache under the specified policies are computed.

For each main memory cache size requested, the program outputs histograms and moments of the distributions of the number of elements in the cache, the cache residency time, and the cache lru stack depth. The hit ratios overall and for each access method and function are also provided.

The cache performance statistics report breaks down the references into read or write and separately into image, ksam, directory, message, sequential, direct, and other. The number of each access is given in the count column, and the portion of the total that the access type represents is given in the probability column. The number of partial hits refers to a reference to a disc domain which is only partially cached. When these occur, we flush the cached region and treat it as a miss. The number of hits and the hit ratio for each access type are given in the last two columns.

The percent of disc accesses eliminated by caching in the specified memory size is listed, along with the number of cache replacements required to perform the simulation. All read hits eliminate disc accesses altogether. All write misses eliminate the need to wait for the disc access from the issuing process' point of view, but the disc access is still performed in background mode. Most of the write hits also eliminate a wait, unless a write is already active against the cached disc domain, in which case the write request gets its priority bumped and the process must wait until the prior write completes.

The cache entry count distribution shows a histogram of the number of disc domains cached in memory during the simulation.

The cache lru stack depth distribution gives a histogram of the depth of reference into an lru stack of the cached disc domains. When a disc cached disc domain is referenced, it is pulled from its lru stack location and put at the head of the stack. The histogram slot corresponding to its old stack depth is bumped. The simulation also

gives timing information such as residency times of cached domains and inter-reference times.

Implementation

The cache is simulated as an lru stack. The real replacement algorithm (modified clock) is a very close approximation to lru, so the use of an lru stack as the model should be accurate. Although the real implementation uses a floating virtual boundary between cache and segment real memory, the approximation of a fixed cache size used in this simulation is a good first order estimate of the behavior and performance to be realized by adding incremental memory of the given cache sizes.

The lru stack simulating the cache is implemented as a doubly linked list, and a set of general list maintenance manipulation primitive procedures and functions are provided to test list status, add or delete entries to lists & to manipulate an auxiliary pointer associative the list.

The cache control block contains pointers to the head and tail of the lru stack, a count of the number of elements in the cache, the cache size, the amount of valid data in the cache, and an auxiliary pointer used by the list manipulation primitives.

A cache element contains fields identifying the access type, access function (R/W), device number, base disc address of a cache block, size of cache block, time of insertion into cache, time of last reference, and previous and next in lru list cache block pointers.

The cache is maintained as follows : when a disc block is referenced, the lru stack is searched to see if the disc domain is already present. If so, the associated lru stack list element is moved to the front of the lru stack, and the appropriate distributions and instrumentation are updated. If the disc domain is partially present, any list elements partially overlapping the desired domain are deleted. The reference is then serviced as a cache miss.

In the case of a cache miss, the specified fetch policy determines the fetch size round off policy, and sufficient bottom of stack elements are deleted until enough space is made available for the referenced block and its round-off to be placed in the cache. The list element related to the new disc domain is inserted at the head of the lru list.

In the case of sequential access, once a boundary is reached, the element is deleted from the stack. This flushing is performed to take advantage of the knowledge that sequentially accessed domains are not needed after their last block is referenced.

Throughout, effected distributions and instrumentation are updated.

Sample Output

Simulation ID : lab timeshare machine trace
Trace File Name : smurftr (disc trace file name)
Output File Name : simout (ascii disc file)

Simulation Initialization Values

Specified Disc Cache Size Range :

Initial Cache Size (kbytes) : 1000 (1 Mbyte)
Final Cache Size (kbytes) : 2000 (2 Mbytes)
Cache Size Increment (kbytes) : 1000

Subset of Access Modes To Be Cached : All

Specified Fetch Policy :

Min Random Fetch Size (sectors) = 32
Min Seq Fetch (sectors) = 96

System Default Fetch Policies :

Round-Off Above Extent
Stop At Extent Boundaries In Round
No Fetch On Write Miss

System Default Replacement Policies :

Flush After Seq Ref
No Flush On Write Hit

*** Cache Performance Statistics ***

Size of File Cache (bytes) : 1024000

Ref Type	Count	Prob	# Part Hits	# Hits	Hit Ratio
any	53000	1.00	827	40714	.77
read	36773	.69	780	31172	.85
write	16227	.31	47	9542	.59
image read	0	.00	0	0	.00
image write	0	.00	0	0	.00
ksam read	0	.00	0	0	.00
ksam write	0	.00	0	0	.00
dirc read	8535	.16	52	8165	.96
dirc write	1351	.03	0	1343	.99
msg read	3617	.07	249	3175	.88
msg write	0	.00	0	0	.00
seq read	6677	.13	11	5429	.81
seq write	5969	.11	31	591	.10
direct read	12611	.24	468	10222	.81
direct write	2899	.05	13	1938	.67
other read	5333	.10	0	4181	.78
other write	6008	.11	3	5670	.94

Percent of Disc Accesses Eliminated : 59

Number of Cache Replacements : 5010

*** Cache Behavior Statistics ***

Cache Entry Count Distribution :

Cache Entry Cnt	# Occurrences	Prob
0	363	.01
100	7112	.13
200	39179	.74
300	6346	.12
400	0	.00
500	0	.00
600	0	.00
700	0	.00
800	0	.00
900	0	.00
1000	0	.00
1100	0	.00
1200	0	.00
1300	0	.00
1400	0	.00
>= 1500	0	.00

Samples = 53000
 Distribution Mean = 244.38
 Distribution Std Dev = 46.31
 Histogram Median = 200

Cache LRU Stack Depth Distribution :

LRU Stack Depth	# Occurrences	Prob
0	26081	.64
5	3961	.10
10	2066	.05
15	1305	.03
20	848	.02
25	635	.02
30	510	.01
35	476	.01
40	366	.01
45	302	.01
50	243	.01
55	236	.01
60	237	.01
65	253	.01
70	204	.01
>= 75	2991	.07

Samples = 40714
 Distribution Mean = 17.28
 Distribution Std Dev = 39.58
 Histogram Median = 0

Appendix E

Analytic System Model

This model calculates performance and behavior statistics for a computer system with a processor, optionally a disc cache, and a specified number of parallel disc servers. This model is useful for examining disc and processor contention and utilization under various system configurations and workload characteristics.

The model calculates performance and behavior statistics for a computer system with one service class, a processor, a disc cache, and a specifiable number of parallel disc servers. The workload demands and system component service rates are specified through the input parameters. From these, a closed queueing network is constructed consisting of J service centers and N identical customers. Each service center has a single server.

Input

Inputs are solicited by the program for the following workload and system parameters :

Processor Subsystem Parameters :

Eff Processor Speed (MIPS)
Mem to Mem Move Rate (Mbytes/sec)
Instructions / Transaction(x 1 Million)

cpu<->disc subsystem circulations per transaction

Disc Subsystem Parameters :

disc subsystem visits per transaction
Mean Req Xfer Size
Prob of Disc Read vs Disc Write
I/O Program Overhead
Number Of Parallel Discs Selectd
Mean Disc Access Times
Channel XferRate
Disc Cache Overhead
Disc Cache Read Hit Ratio
Disc Cache Write Wait Prob
Prob of Wait For Post on Write
Mean Disc Cache Fetch

multiprogramming level

These inputs are obtained as follows :

Effective Processor Speed

This parameter includes typical instructions seen by the cpu and processor cache and pipeline effects. The processor speeds are stated relative to IBM MIPS.

Memory to Memory Move Rate

This parameter is needed since some machines can move blocks around in memory much faster than by successive load and store instructions and disc caching uses memory to memory moves extensively.

Instructions per Transaction

This value is obtained from the formatted instrumentation report by taking the cumualtive cpu time spent on processes and dividing it by the cumulative number of terminal reads to obtain cpu time per transaction, then multiply this value by the effective number of

instructions per second that the processor on which the measurement was made can sustain (effective processor speed x 1 million). If only one customer class rather than the aggregate mean for the entire workload, obtain this value by a similar calculation on the process cpu time report for a process's transaction class.

CPU <-> Disc Circulations Per Transaction

This is the number of visits to the disc that would be generated for a typical transaction without disc caching.

Number of Parallel Discs

The number of master disc volumes on the system. It is assumed that the disc accesses generated are spread evenly across the volumes. This tends to favor uncached system performance since it uses the disc subsystem far more. If the requests are skewed, lower the number of parallel discs inputted here to compensate.

Mean Disc Access Time

Seek + Rotational latency. Depends on disc type.

Channel Transfer Rate

Mbytes per Second. Depends on channel type.

Mean Transfer Size Without Disc Caching

Obtained from Disc Workload Characterizer.

Disc Cache Overhead

In kiloinstructions for internal cache (not including the data move) In
milleseconds for external cache

Channel Program Overhead

In kiloinstructions. Includes setup and interrupt service time.

Percent of Disc Accesses Which Are Reads

Obtained from disc workload characterizer.

Disc Cache Read Hit Percentage

Obtained from the disc cache simulation performance report.

Write Wait Probability

Determined By Write Ahead Log Management Policy

Mean Disc Cache Fetch

Minimum amount fetched on a read miss (in kbytes).

Effective Mean Multiprogramming Level

of processes which can execute in parallel assuming loaded system.

If a serial database, only 1 process can be circulating between the
disc and cpu at a time (the one with the database lock). If more than
one process is mapped to a single effective multigramming level,
response times must be multiplied by number of processes mapped to
obtain correct response times. This approximation is valid only for a
fully loaded system.

These configuration and workload parameters are used to construct the central server model consisting of a processor station, optionally a disc cache, and the number of specified parallel disc servers. The service times and transition probabilities for the queueing model are computed to obtain the central server model parameters. If internal caching is modelled, the model is reduced to an equivalent model without an explicit cache. When a disc cache is present in the configuration, it is assumed that all file accesses go through cache, so the cache service time is added to the disc service time to obtain the effective miss read time.

Output

The output parameters describe the system performance and behavioral characteristics. These include the utilization, throughput, steady state response time, and steady state number of customers at each service center.

$U_j(i)$ = utilization of service center j with MPL i
 $T_j(i)$ = throughput of service center j with MPL i
 $E[r_j(i)]$ = response time of service center j with MPL i
 $E[n_j(i)]$ = mean steady state queue length of service center with MPL i .

Algorithm

The input parameters describing the workload and configuration are used to build a central server model. The setup of this model is now described.

Calculation of transition probabilities :

$$p[1] = \text{prob of transaction completion} = 1 / (\text{cpu_visits_per_trans})$$

$$p[2] = \text{prob of visting disc cache} = \text{prob of visiting disc subsystem} \\ \text{and not generating a read miss}$$

$$\text{or not generating a write wait}$$

$$= (1-p[1]) * ((\text{read_prob} * \text{read_hit_prob}) + (1-\text{read_prob}) * (1- \\ \text{write_wait_prob}))$$

FOR j := 3 TO service_center_count

$$p[j] = \text{prob of visiting disc } j = (1-p[1]-p[2]) / \text{par_disc_count}$$

(Note that these transition probabilities model cached writes that don't require waits as not causing a physical disc transfer. This approximation is justified when physical disc posts are not issued due to write caching with repeated references fielded in cache, and when they are performed as background traffic utilizing otherwise idle disc capacity.)

Calculation of Service Times

$$s[1] = \text{cpu service time} / \text{visit}$$

$$= \text{secs_per_trans} / \text{cpu_visits_per_trans}$$

```

s[2] = service time of disc cache
      = IF internal_cache
          THEN dc_instr_overhead / cpu_speed
              + disc_req_size / mem_move_rate
          ELSE (external cache) = dc_ms_overhead
              +io_program_instr_overhead/cpu_speed
              +disc_req_size/eff_chan_xfer_rate

```

```

FOR j := 3 TO service_center_count DO

```

```

s[j] = service time of disc
      = IF NOT disc_cache
          THEN disc_req_size/eff_chan_xfer_rate
              + disc_access_time
              + io_program_instr_overhead/ nm_cpu_speed
          ELSE (disc cache) = dc_fetch_size/eff_chan_xfer_rate
              +io_program_instr_overhead/ nm_cpu_speed
              +disc_access_time

```

Construction Of Equivalent Model

```

IF internal_cache THEN Absorb Disc Cache Into Processor Station

```

```

BEGIN

```

```

(adjust transition probabilities)

```

```

disc_visits_per_trans = disc_visits_per_trans
                        -disc_visits_per_trans
                        *((read_prob*read_hit_prob)

```

```

cpu_visits_per_trans = disc_visits_per_trans + 1
p[1] = 1/cpu_visits_per_trans (completion prob)
FOR i := 3 TO service_center_count
DO p[i] = (1-p[1])/par_disc_count (disc visit prob)
          +(1-read_prob)*(1-write_wait_prob))

(adjust cpu service time to include cache overhead for each
disc visit since everything thru the cache, and for the
fewer # of circulations)

s[1]=(cpu_per_trans+s[2]*olddiscvisits)/cpu_visits_per_trans

END

```

```

IF external_cache THEN Adjust Disc Service Time For Cache Misses
FOR j:=3 to service_center_count
DO s[j] = s[j] + s[2] (suffer cache overhead on misses)

```

Model Computation

Once the input has been interpreted and reduced to a standard central server model, an iterative technique presented in [Lav 83] is used to solve the queueing network.

The queueing discipline at service center 1 (cpu) is processor sharing. The queueing discipline at server $j \neq 1$ is FCFS and the service demands are assumed to have an exponential distribution.

A customer completing service at service center 1 immediately enters service center j with probability P_j where $P_1 + \dots + P_J = 1$. Service center 1 is the central server. A customer that completes service at service center j for $j \neq 1$ immediately enters service center 1.

The procedure is recursive with respect to the number of customers in the network. The algorithm is presented below.

For $j = 1 \dots J$ and $i = 1 \dots N$

$$E[n_j(0)] = 0 \text{ for } j = 1 \dots J$$

For $i = 1 \dots N$

$$R_j(i) = E[S_j] * (1 + E[n_j(i-1)])$$

$$T_1(i) = i / (R_1(i) + p_2 R_2(i) + \dots + p_j R_j(i))$$

$$T_j(i) = p_j T_1(i)$$

$$E[n_j(i)] = T_j(i) E[r_j(i)]$$

$$U_j(i) = T_j(i) E[S_j]$$

Sample Output

Disc Cache System Model (V 3.4)

Model ID : 1 ms ext cache, 4 10 ms discs, rh=85, ww=0
Output File Name : *modelout
Xact Graph File Name : t1e410n
Util Graph File Name : u1e410n

Processor Subsystem Parameters :

Processor Speeds Selected :
0.25 0.50 1.00 2.00 4.00 8.00 16.00 32.00 64.00
CPU Speed This Run : 0.25
Instructions / Transaction(x 1 Million) : 0.192000

CPU usage / Transaction (Secs) : 0.768000

Disc Subsystem Parameters :

disc subsystem visits per transaction 24.00
Mean Req Xfer Size (kbytes) : 0.25
Prob of Disc Read vs Disc Write : 0.7000
I/O Program Overhead (kinstr) : 2.000000
Number Of Parallel Discs Selected : 4
Mean Disc Access Time (ms) : 10.000000
Channel Xfer Rate (Mbytes/sec) : 2.00
Disc Cache Overhead (ms) : 1.00
Disc Cache Read Hit Ratio : 0.85
Disc Cache Write Wait Prob : 0.00
Prob of Wait For Post on Write : 0.0000
Mean Disc Cache Fetch (kbytes): 2.00

Calculated Service Times (sec):

cpu service time per circulation : 0.030720
disc cache service time : 0.009122
disc service time : 0.019000

Calculated Transition Prob :

prob of transaction comp after cpu : 0.04
prob of disc cache visit after cpu : 0.86
prob of visiting a disc after cpu : 0.025

multiprogramming level = 64

*** Output Statistics ***

mpl = 1

CPU Statistics

cpu response time = 0.031
cpu utilization = 0.742
cpu throughput = 24.159
cpu queue length = 0.742

Disc Cache Statistics

dc response time = 0.009
dc utilization = 0.189
dc throughput = 20.757
dc queue length = 0.189

Per Disc Statistics

disc response time = 0.028
disc utilization = 0.017
disc throughput = 0.609
disc queue length = 0.017

Overall Transaction Response Time : 1.035
Overall Transaction Throughput : 1.0



Appendix F

Kernel Tuning Commands

1. TUNE

Changes the filter and/or priority limits of the circular subqueues.

Syntax

```
      CQ
:TUNE [minclockcycle][;{DQ} = [base][,[limit][,[min][,[max]]]]]
      EQ
```

Parameters

minclockcycle	Minimum value (in milliseconds) for replacement algorithm cycle through memory. If algorithm cycles through memory in less time than this value, memory allocation is delayed. (thrash prevention mechanism.)
base	Priority at which C, D, or E processes will begin in queue.
limit	Worst (highest number) priority C, D, or E processes can attain.
min	The minimum value indicating the length of time (filter value) in milliseconds which a process can have the CPU before its priority is reduced.
max	The maximum value indicating the length of time (filter value) in milliseconds which a process can have the CPU before its priority is reduced.

Operation

This command changes the filter or priority limits of a circular subqueue, and is used primarily in on-line tuning of the system to best accommodate the current load.

A CS process is given a priority of CBASE when it begins. When a process stops (for disc I/O, terminal I/O, preemption, etc.), a new priority is determined so that it may be re-queued for the cpu. If the

process has completed a transaction (a transaction is defined as the time between terminal reads), the priority become CBASE. The value of an "average short transaction" is then recalculated. If the CS process has not completed a transaction, and if the process has exceeded the average short transaction filter value since its priority was last reduced, the priority is increased (made worse) by 1.

DS and ES processes begin at DBASE and EBASE respectively, and are re-scheduled according to the same criteria as used for CS processes, with the exception that a fixed value (the value of max which has been specified for the subqueue) is used in place of the average short transaction value, which is used for CS processes only.

If the values specified for max in the CS, DS, and ES queues are too large, system response may become erratic. If they are too small, excessive swapping may result. Recommended settings for min and max for the CS subqueue are 0 and 300, as it is desirable to favor short transactions in most cases. A value for min and max of 10,000 usually produces efficient system operation for the DS and ES subqueues.

The minclockcycle parameter is used by the memory manager to determine when thrashing (excessive memory management activity) is occurring. Making this value smaller increases the possibility of thrashing. Recommended value for this parameter is 1000.

2. STARTCACHE

Enables disc file caching for a logical device.

Syntax

```
:STARTCACHE ldn
```

Parameters

ldn The logical device number of a particular disc to have disc caching enabled.

This command allows a previously uncached disc to be cached in memory by the kernel.

Example

To enable caching of logical device number 10 to function enter:

```
:STARTCACHE 10
```

3. STOPCACHE

Syntax

```
:STOPCACHE ldn
```

Parameters

ldn The logical device number of a particular disc to have disc caching enabled.

Operation

If the device is being cached, the :STOPCACHE request will be satisfied when the last access is complete.

Example

To remove logical device number 20 from the cached disc set enter:

```
:STOPCACHE 20
```

4. SHOWCACHE

Reports on disc caching status.

Syntax

:SHOWCACHE

Parameters

None

Operation

The :SHOWCACHE command reports the following information pertaining to the state of disc caching in the system: (a) a list of the cached ldevs, (b) total cache requests, (c) percent of cache hits and reads (d) thousands of bytes of memory used as cache domains, (e) percent of memory being used, (f) the total number of cache domains, and (g) the amount of memory used as overhead to support caching.

Example

DISC LDEV	CACHE REQUESTS	READ HIT%	WRITE HIT%	PROCESS READ%	PROCESS STOPS	K-BYTES	% OF MEMORY	CACHE DOMAINS
5	56356	82	22	66	7281	0	0	0
1	88328	87	62	84	11195	690	11	154
2	150363	82	44	60	22699	1931	32	171
3	21609	78	14	44	2068	696	11	41
4	25514	85	9	50	1901	879	14	48
Total	342170	84	36	66	45144	4196	69	414

55% of user I/Os eliminated.
Data overhead is 113K bytes.
Sequential fetch quantum is 96 sectors.
Random fetch quantum is 16 sectors.
Block on Write = NO.

5. CACHECONTROL

Modifies the global caching parameters for the system.

Syntax

```
:CACHECONTROL {RANDOM = n          } [...[;...]]
                {BLOCKONWRITE = YES/NO} [...[;...]]
                {SEQUENTIAL = n       } [...[;...]]
```

Parameters

- RANDOM** Assigns the number of sectors read from disc on a cache read miss of a random access file. The disc read will stop at extent boundaries and will fetch at least the amount required. The number sectors must be between 1 to 96. The system default is 16.
- BLOCKONWRITE** Specifies whether or not to "block" the process until the posting of cache buffers to disc completes. The NO option is the default and allows the posting to complete in the background. This only applies to writes since reads do not alter the contents of the cache buffers.
- SEQUENTIAL** Assigns the number of sectors read from disc on a cache read miss of a sequential file. The disc read will stop at extent boundaries and will fetch at least the amount requested. The number of sectors must be between 1 to 96. The system default is 96.

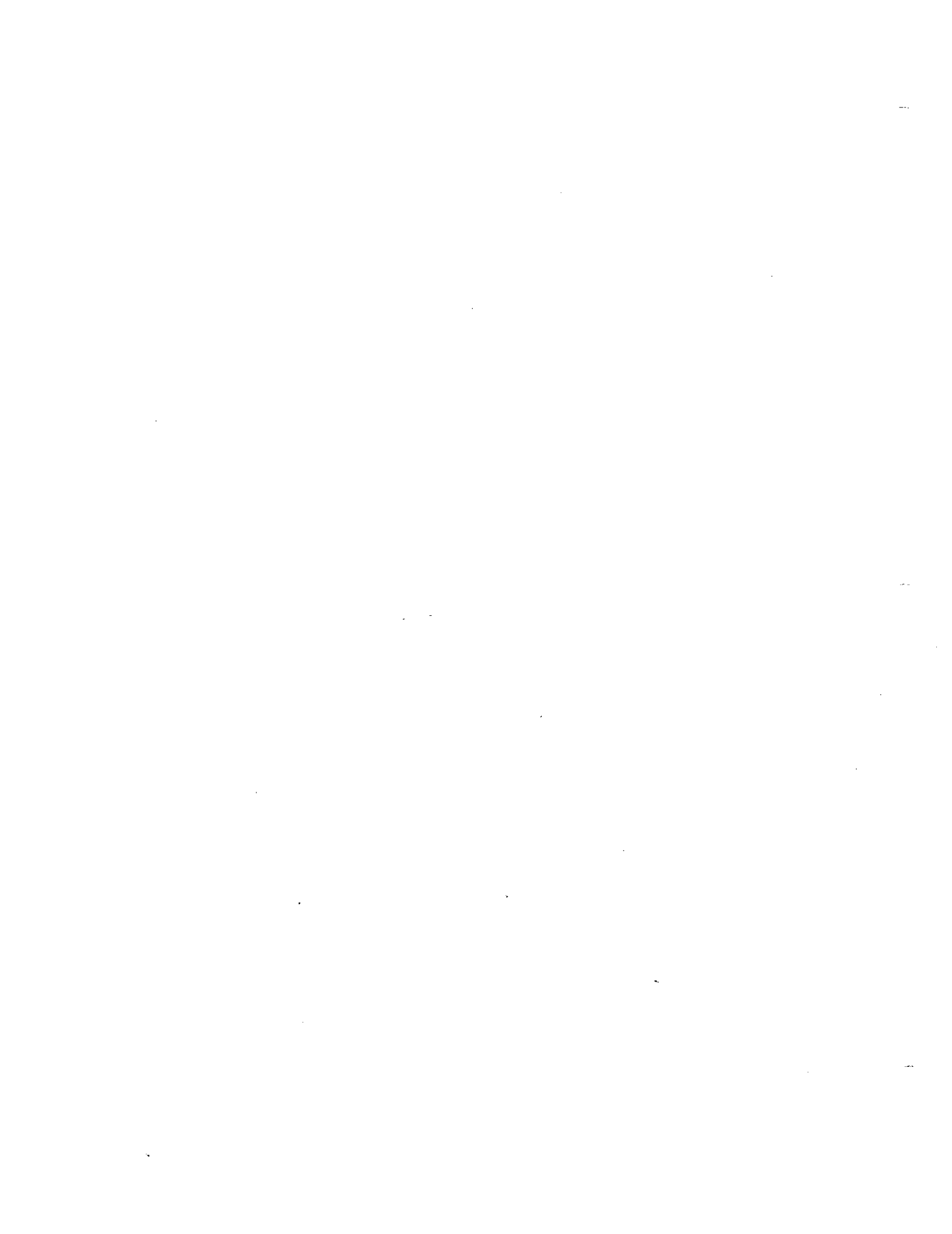
Operation

This command is used to tune the performance of caching on a running system.

Example

To alter the number of sectors read on a sequential fetch to be 90, and the number of sectors on a random fetch to be 10, the command would be:

```
:CACHECONTROL SEQUENTIAL=90;RANDOM=10
```



Bibliography

In the following references, *CACM* stands for *Communications of the Association For Computing Machinery*, *SOSP* stands for *Symposium on Operating System Principles* and *IEEE* stands for *Institute of Electronic and Electrical Engineers*.

- Agr 83 Agrawal, R., Carey, M.J., DeWitt, D.J., *Deadlock Detection is Cheap*, Draft, Computer Science Department, University of California, Berkeley, 1983
- Bab 79 Babaoglu, Ozalp and Joy, W., *Design and Implementation of the Berkeley Virtual Memory Extensions to the Unix Operating system*, Draft, Computer Science Department, University of Calif., Berkeley, December 1979
- Bar 73 Bard, Yonathan, *An Experimental Approach to System Tuning*, IBM Scientific Center, Cambridge, Mass.
- Bar 75 Bard, Y., *Application of the Page Survival Index (PSI) to Virtual-Memory System Performance*, IBM J. Res. Develop., May 1975, pp 212-220
- Bas 82 Basart, Ed, D. Folger, and B. Shellooe, *Pipelining and New OS Boost Mini to 8 MIPS*, Mini-Micro Systems, Sept. 1982, pp 258-269
- Bat 70 Batson, A. and Ju, S., *Measurements of Segment Size*, Comm of the ACM, Vol 13, No 3, March 1970, pp 155-159
- Bat 76 Batson, A., *Program Behavior at the Symbolic Level*, Computer, November 1976, pp 21-26
- Bat 77 Batson, A.P. and Brundage, R.E., *Segment Sizes and Lifetimes in Algol 60 Programs*, Comm of the ACM, Vol 20, No 1, January 1977, pp 36-44
- Bau 75 Baudet, Gerard and Boulenger, J., *Analysis of a Drum with Bulk Arrivals*, International Computing Symposium 1975
- Bea 72 Beausoleil, W. F., D. T. Brown, and B. E. Phelps, *Magnetic Bubble Memory Organization*, Communication, Nov. 1972, pp 587-591
- Bel 74 Bell, James, D. Casasent, and C. Gordon Bell, *An Investigation of Alternative Cache Organizations*, IEEE Transactions on Computers, Vol. C-23, No. 4, April 1974, pp 346-351

- Bel 83 Bell, Alan E., *Critical Issues In High-Density Magnetic and Optical Data Storage*, Laser Focus/Electro-Optics, Part 1 : August 1983 pp 61-66, Part 2 : September 1983 pp 125-136
- Ber 78 Beretvas, T., *Performance Tuning in OS/VS2 MVS*, IBM System Journal, Vol. 17, No. 3, 1978, pp 290-313
- Bla 77 Blake, Russel P., *Exploring a Stack Architecture*, Computer, May 1977, pp 30-39
- Bob 71 Eobeck, Andrew H., and H. E. D. Scovil, *Magnetic Bubbles*, Scientific American, Vol. 28, No. 78, June 1971, pp 78-90
- Boy 70 Boyle, W. S., and G. E. Smith, *Charge Coupled Semiconductor Devices*, Bell System Technical Journal, April 1970, pp 587-593
- Bra 74 Brandwajn, A., *A Model of a Time Sharing Virtual Memory System Solved Using Equivalence and Decomposition Methods*, Acta Informatica 4, 11-47, 1974
- Bri 81 Brigg, Faye A., Kai Hwang, K. S. Fu, and Benjamin W. Wah, *PUMPS Architecture for Pattern Analysis and Image Database Management*, IEEE 1981, pp 178-187
- Bul 77 Bulman, David M., *Stack Computers*, Computer, May 1977, pp 14-28
- Bun 77 Bunt, R.B. and Hume, J.N.P., *Adaptive Processor Scheduling Based On Approximating Demand Distribution*, Infor, vol.15, No.2, June 1977, pp 135-147
- Bur 80 Bursky, Dave, *Special report : Memories Pace Systems Growth*, Electronic Design, Sept. 1980, pp 63-78
- Bus 82 Busch, John R., *The MPE IV Kernel : History, Structure and Strategies*, Proceedings of the HP 3000 International User's Group Conference, Orlando, April 27, 1982, F-9; reprinted in Journal of the HP 3000 International User's Group Conference, Inc, Vol 5, No 3,4, July 1982
- Bus 83 Busch, John R., *Disc Caching in the System Processing Units of the HP 3000 Family of Computers*, Proceedings of the HP 3000 International User's Group, Edinburgh, October 1983; reprinted in Journal of the HP International Users Group, Vol 7, No 1, January 1984, pp 21-24
- Buz 73 Buzen, J.P. and Gagliardi, U.O., *The Evolution of Virtual Machine Architecture*, National Computer Conference 1973, pp 291-299
- Buz 80 Buzen, J.P. and Denning, P.J., *Measuring and Calculating Queue Length Distributions*, Computer, April 1980, pp 33-44

- Car 81 Carr, Richard W. and Hennessy, John L., *AWSClock - A Simple and Effective Algorithm for Virtual Memory Management*, ACM 0-89791-062-1-12/81-0087, 1981
- Cen 78 Censier, Lucien M., and Paul Feartrier, *A New Solution to Coherence Problems in Multicache Systems*, IEEE Transactions on Computers, Vol. C-27, No. 12 Dec. 1978, pp 1112-1118
- Cha 73 Chamberlin Donald D. and Fuller, Samuel H., *A page Allocation Strategy for Multiprogramming Systems*, Symp. on Operating Principles, 1973, pp 66-72
- Cha 73 Chamberlin, D.D. and Fuller, S.H., *An Analysis of Page Allocation Strategies for Multiprogramming Systems with Virtual Memory*, IBM J. Res. Develop., Sept 1973
- Cha 83 Chan, A. and Gray, R., *Implementing Distributed Read Only Transactions*, submitted for publication, 1983
- Cha 77 Chandy, K.M. and Reiser, M., *Computer Performance*, North-Holland Publishing Company, 1977
- Cha 74 Chang, Hsu, *Capabilities of the Bubble technology*, National Computer Congerence, 1974, pp 847-855
- Che 78 Chen, Tien Chi, *Magnetic Bubble Memory and Logic*, Advances in Computers, Vol. 17, 1978, pp 224-282
- Che 83 Cheriton, D. R., and W. Zwaenepoel, *The Distributed V Kernel and its Performance for Diskless Workstations*, ACM, Vol. 17, no. 5, Oct. 1983, pp 129-140
- Chi 79 Chin, Y. H. and Yu, S.H., *A Mathematical Model for Distributed Free Space*, National Computer Conference, 1979
- Cho 75 Chow, C. K., *Determination of Cache's Capacity and its Matching Storage Hierarchy*, IEEE Transactions on Computers, Vol. C-25, No. 2, Feb. 1976, pp 157-164
- Cho 75 Chow, C. K., *Determining the Optimum Capacity of a Cache Memory*, IBM Technical Disclosure Bulletin, Vol. 17, No. 10, March 1975, pp 3163-3166
- Cho 77 Chow, We-Min and Chiu Willy W., *An Analysis of Swapping Policies in Virtual Storage Systems*, IEEE Transactions on Software Engineering, vol.se-3, No.2, March 1977, pp 150-156
- Chr 70 Christensen, C. and Hause, A.D., *A Multiprogramming, Virtual Memory System for A Small Computer*, Spring Joint Computer Conference, 1970, pp 683-690
- Chu 69 Chu, Wesley W., *Optimal File Allocation in a Multiple Computer System*, IEEE Transactions on Computers, Vol. C-18, No. 10, October 1969, pp 885-889

- Chu 76a Chu, Wesley W. and Opderbeck, Holger, *Analysis of the PFF Replacement Algorithm via a Semi-Markov Model*, Communication of the ACM, vol.19, No.5, May 1976, pp 298-303
- Chu 76b Chu, Wesley W. and Opderbeck, Holger, *Program Behavior and the Page-Fault-Frequency Replacement Algorithm*, Computer, November 1976, pp 29-38
- Chu 79a Chu, Wesley W., and Peter P. Chen, *Tutorial : Centralized and Distributed Data Base Systems*, IEEE Computer Society.
- Chu 79b Chu Wesley W. and Paul Hurley, *A Model for Optimal Query Processing for Distributed Data Bases*, UCLA, 1979
- Chu 80 Chu, Wesley W., *Distributed Data Base Systems*, Computer Science Dept., UCLA, Nov. 1980.
- Cla 81 Clark, Douglas W., B. W. Lampson, and Kenneth A. Pier, *The Memory System of A High-Performance personal Computer*, IEEE Transactions on Computers, Vol. C-30, No. 10, Oct. 1981, pp 715-733
- Cla 83 Clark, Douglas W., *Cache Performance in the VAX-11/780*, ACM, Vol. 1, No. 1, Feb. 1983, pp 24-37
- Cof 73 Coffman, Edwards G. Jr., and Peter J. Denning, *Operating Systems Theory*, Prentice-Hall, 1973
- Cov 72 Covill, D. L., *B6700 Virtual Memory*, 1972, pp 130-139
- Com 83 Computerworld, *Series/1 Gets Top CPU, Extended Networking; System/38 Also Capped*, Vol. 17, No. 15, April, 1983.
- Con 69 Conti, C. J., *Concepts for Buffer Storage*, Computer Group News, March 1969, pp 9-21
- Cra 75 Cranston, B. and Thomas, R., *A Simplified Recombination Scheme for the Fibonacci Buddy System*, Comm of the ACM, Vol 18, No 6, pp 331-332
- Cro 76 Crouch, Harry R., and John B. Cornett Jr., *CCDs in Memory Systems Move into Sight*, Computer Design, Sept. 1976, pp 75-80
- Den 68 Denning, P. J., *The Working Set Model for Program Behavior*, CACM, Vol 11, No 5, May 1968, pp 323-333
- Den 70 Denning, Peter J., *Virtual Memory*, Computing Surveys, vol.2, No.3, Sept. 1970, pp 153-189
- Den 75 Denning, Peter J. and Graham, G. Scott, *Multiprogrammed Memory Management*, Proceedings of the IEEE, vol.63, No.6, June 1975, pp 924-938

- Den 76 Denning, Peter J., *Program Behavior, Working Sets, and Multiprogramming, Draft of a paper for the book Software Modeling and its Impact on Performance*, June 1976
- Den 78 Denning, P. and Slutz, Donald R., *Generalized Working Sets for Segment Reference Strings*, Communication of the ACM, vol.21, No.9, Sept. 1978, pp 750-759
- Den 80 Denning, Peter J., *Working Sets Past and Present*, IEEE Transactions on Software Engineering, vol.se-6, No.1, Jan. 1980, pp 64-84
- Den 81 Denning, Peter J. and Kahn, Kevin C., *An L-S Criterion for Optimal Multiprogramming*, Computer Science Dept., Purdue University, West Lafayette
- Doy 75 Doyle, M. S., and J. W. Graham, *Some Parameters Affecting the Performance of Paged Storage Hierarchies*, Information, Vol. 13, No. 2, June 1975, pp 197-207
- Eas 77 Easton, M.C. and Bennett, B.T., *Transient-Free Working Set Statistics*, Comm of the ACM, Vol 20, No 2, Feb 1977, pp 93-99
- Eis 72 Eisenberg, Murray A., *Further Comments on Dijkstra's Concurrent Programming Control Problem*, Communications of the ACM, Vol. 15 No. 11, Nov. 1972
- Dij 67 Dijkstra, E. W., *Solution of a Problem in Concurrent Programming Control*, Communications of the ACM, Vol. 10, No. 3, March 1967, pp 569-572
- Ele 82 *The Motorola 68020 32-Bit Microprocessor*, Electronics, March 10, 1983, p 40
- Ele 83a *256- and 512-Kbyte Bubble Cards Seek Niche in IBM Personal Computer Market*, Electronics, January 12, 1984, pp 206-208
- Ele 84b *1984 World Markets Forecast*, Electronics, January 12, 1984 pp 123-154
- Ele 84c *Fields, Stephen W., National Hits With 32-Microprocessor*, Electronics, March 8, 1984, pp 97-98
- E11 75 Ellis, Clarence A., *Optimal Scheduling of Homogeneous Job Systems*, Information Sciences 9, 1975, pp 323-359
- Epl Epley, Donald L., *Scheduling Problems in Computer Systems*, University of Iowa, Iowa
- Esw 74 Eswaran. K.P., Gray, J.N., Lorie, R.A., Traiger, I.L., *On the Notions of Consistency and Predicate Locks in a Database System*, IBM Research, 1974

- Fen 74 Fenton, J.S., and Payne, D. W., *Dynamic Storage Allocation of Arbitrary Sized Segments*, Information Processing 74, North Holland Publishing Company, 1974, pp 344-348
- Fer 74 Ferrari, D., *Improving Program Locality by Strategy Oriented Restructuring*, Information Processing 74, North Holland Publishing Company, 1974, pp 266-270
- Fer 76 Ferrari, D., *Program Behavior*, Computer, November 1976, pp 7-13
- Fer 76 Ferrari, D., *The Improvement of Program Behavior*, Computer, Nov. 1976, pp 39-47
- Fer 76 Ferrari, D., *The Improvement of Program Locality*, Computer, November 1976, pp 39-47
- Fer 78 Ferrari, D., *Computer Systems Performance Evaluation*, Prentice Hall, Inc, 1978
- Fit 81 Fitzpatrick, D.T., Ketevenis, M.G. et al, *VLSI Implementations of a Reduced Instruction Set Computer*, CMU Conference on VLSI Systems and Computations
- Fly 78 Flynn, M. J., Gray, J.N., Jones, A.K., Legally, K., Opderbeck, H., Popek, G.J., Randell, B., Saltzer, J.H., Wiehle, H.R., *Operating Systems : An Advanced Course*, Lecture Notes In Computer Science, Springer-Verlag, 1978
- Fra 73 Frailey, Dennis J., *A Practical Approach to Managing Resources and Avoiding Deadlocks*, Communications of the ACM, Vol. 16, No. 5, may 1973, pp 323-329
- Fra 84 Frank, S.J., *Tightly Coupled Multiprocessor System Speeds Memory Access Times*, Electronics, January 12, 1984, pp 164-169
- Ful 75 Fuller, Samuel H., *An Analysis of Drum Storage Units*, Journal of the Association for Computing Machinery, Vol. 22, No. 1, Jan. 1975, pp 83-105
- Gel 80 Gelenbe, E., and I. Mitrani, *Analysis and Synthesis of Computer Systems*, Academic Press, 1980
- Gha 75 Ghanem M.Z., *Dynamic Partitioning of the Main Memory Using the Working Set Concept*, IBM J. Res. Develop., Sept. 1975, pp 445-450
- Gha 75 Ghanem M.Z., *Study of Memory Partitioning for Multiprogramming Systems with Virtual Memory*, IBM J. Res. Develop., 1975, pp 451-457
- Gif 77 Gifford, D. K., *Hardware Estimation of a Process' Primary Memory Requirements*, Comm. of the ACM, Vol 20, No 9, September 1977, pp 655-663

- Gif 82 Gifford, David K., *Information Storage in a Decentralized Computer System*, CSL-81-8, June 1981; revised 1982
- Gol 82 Goldberg, Authur, Steve Lavenberg, and Gerald Popek, *A Validated Distributed System Performance Model*, Nov. 1982.
- Gom 74 Gomaa, H., *An Exercise in Resource Management, Software Practice and Experience*, Vol 4, 1974, pp199-213
- Gup 78 Gupta, Ram K. and Franklin, Mark A., *Working Set and Fault Frequency Paging Algorithms:*, IEEE Transactions on Computers, vol.c-27, No.8, August 1978, pp 706-712
- Hab 76 Habermann, A. N., *Introduction to Operating System Design*, Science Research Associates, Inc., 1976
- Hem 77 Hemy, H., *IBM 3850 Mass Storage Subsystem, Performance Evaluation Using a Channel Monitor*, Computer Performance, North Holland Publishing Company, 1977, pp 177-196
- Hat 71 Hatfield, D.J. and Gerald, J., *Program Restructuring for Virtual Memory*, IBM Syst Journal, No 3, 1971, pp 168-192
- Hik 81 Hikita, Sadayuki, Haruaki Yamazaki, Kiyoshi Hasegawa, and Yutaka Matsushita, *Optimization of the file access method in content addressable database access machine (CADAM)*, National Computer Conference, 1981, pp 507-513
- Hin 75 Hinds, J. A., *An Algorithm for Locating Adjacent Storage Blocks in the Buddy System*, Comm of the ACM, Vol 18, No 4, April 1975, pp 221-222
- Hir 73 Hirschberg, D.S., *A Class of Dynamic Memory Allocation Algorithms*, Comm. of the ACM, Vol 16, No 10, October 1973
- HoA 74 Hoare, C. A. R., *Monitoring: An Operating System Structuring Concept*, Communications of the ACM, Vol. 17, No. 10, Oct. 1974 pp 549-557
- HoA 78 Hoare, C. A. R., *Communicating Sequential Process*, Communications of the ACM, Vol. 21, No. 8, August 1978, pp 666-677
- Hod 82 Hodor, Ken M., and Malcolm E. Woodward, *A High-Performance Memory System with Growth Capability*, HP Journal, March 1982, p p 15-17
- Hou 78 Houdek, M.E. and Mitchell, G.R., *Translating a Large Virtual Address*, IBM S/38 Technical Developments, IBM General Systems Division, 1978
- Hou 82 Houston, Velina, *H-P User : Quallex Backup Nets Big Saving*, Management Information Systems Week, August 1982.

- HP 83a Hewlett-Packard Corporation, HP 3000 Configuration Guideline, 1983
- HP 83b Hewlett-Packard Corporation, *Series 64 With Disc Caching Beats IBM 3033 in Batch MRP Run*, HP Computer News, October 1, 1983
- Hug 82 Hugelshofer, Will, and Bruce Schultz, *Cache Buffer for Disk Accelerates Minicomputer Performance*, Electronics, Feb. 1982, pp 155-159
- Int 84 Interact Editor, *1983 Customer Survey*, HP 3000 International User's Group, Inc, January, 1984
- Iso 71 Isodo, S. and Goto, E., *An Efficient Technique for Dynamic Storage Allocation of 2**n-Word Blocks*, CAEM, Vol 14, No. 9 September 1971
- Jul 76 Juliussen, J. Egil, *Magnetic Bubble Systems Approach Practical Use*, Computer Design, Oct. 1978, pp 81-91
- Kah 81 Kahn, Kevin C., and Fred Pollack, *An Extensible Operating System for the Intel 432*, IEEE, 1981, pp 398-406
- Kle 75 Kleinrock, L, *Queueing Systems Vol I,II*, John Wiley and Sons, Inc, 1975
- Kno 65 Knowlton, K.C., *A Fast Storage Allocator*, CAEM, Vol 8, No 10, October 1965, pp 623-625
- Knu 68 Knuth, D.E. *The Art of Computer Programming* , Vol I : Fundamental Algorithms, Addison-Wesley, Reading, Mass, 1968
- Kob 78 Kobayashi, H., *Modeling and Analysis: An Introduction to System Performance Evaluation Methodology*, Addison-Wesley Publishing Company, 1978
- Kra 82 Krastins, Uldis, *Cache Memories Quicken Access to Disk Data*, Electronic Design, May 1982, pp 41-44
- Kra 75 Krause, K.L. and Shen, V.Y., *Analysis of Several Task-Scheduling Algorithms for a Model of Multiprogramming Computer Systems*, Journal of the ACM, Vol 22, No 4, October 1975, pp 552-550
- Kub 75 Kubo, H and Kobayashi, M., *A Cost Oriented Approach to Optimal Page Size*, Second USA-Japan Computer Conference, Session 12-2-1, 1975
- Kub 76 Kubo, H and Kobayashi, M., *Evaluation of Optimal Page Size and Initial Loading Under a System-Wide Criterion*, NEC Research and Development, No 41, April 1976

- Kun 81 Kung, H.T. and Robinson, John T., *On Optimistic Methods for Concurrency Control*, ACM Transactions on Database Systems, Vol. 6, No. 2, June 1981, pp 213-226
- Lam 74 Lamport, Leslie, *A New Solution of Dijkstra's Concurrent Programming Problem*, CAEM, Vol. 17, No. 8, Aug. 1974, pp 453-455
- Lam 78 Lamport, Leslie, *Time, Clocks, and the Ordering of Events in a Distributed System*, CAEM, Vol. 21, No. 7, July 78, pp 558-565
- Lam 83 Lamson, B. W., *Hints for Computer System Design*, ACM, Vol. 17, No. 5, Oct. 1983, pp 33-48
- Lav 83 Lavenberg, Stephen S., *Computer Performance Modeling Handbook*, Academic Press, N.Y., 1983
- Lav 81 Lavi, Y, *Benefits of NS16082 Memory Management Unit*, Wescon/81 Professional Program Session record 9
- Lev 82 Levy, Henry, and Peter Lipman, *Virtual Memory management in the VAX/VMS Operating System*, Computer, March 1982, pp 35-41
- Lei 83 Leitner, Gerald, and Gerald J. Popek, *An Algebraic Model for the Specification and Verification of Distributed Systems*, UCLA
- Ler 76 Leroudier, Jacques and Potier Dominique, *Principles of Optimality for Multiprogramming*, Symp. ACM, Harvard, March 1976, pp 211-218
- Lev 75 Levin, K. Dan, and Howard Morgan, *Optimizing Distributed Data Bases - A framework for research*, National Computer Conference, 1975.
- Lin 81 Lindsay, Donald C., *Cache Memory for Microprocessors*, Dynalogic Corporation, Ottawa, Canada
- Mal 82 Mallach, Efrem G., *What does cache buy for you?*, Mini-Micro System, May 1982, pp 267-271
- Mar 81 Markowitz, R., *iAPX 286 : Virtual Memory and Distributed Computing*, Wescon/81 Professional Program Session record 9
- Mc 82 Mc Kusick, Marshall Kirk, Samuel J. Leffler, and William N. Joy, *The Implementation of a Fast File System for UNIX*, Computer Systems Reserch Group, Dept. of EECS, UC Berkeley.
- Mea 70 Meade, Robert M., *On Memory System Design*, Fall Joint Computer Conference, 1970, pp 33-43

- Mit 83 Mitchell, James G., and Jeremy Dion, *A Comparison of Two Network-Based Servers*, Xerox Palo Alto research Center & Cambridge University computer lab.
- Moo 83 Moore, Johanna D., and Erik T. Mueller, and Gerald J. Popek, *Nested Transactions and Locus*, UCLA.
- Mue 83 Muller, E. T., J. D. Moore, and G. J. Popek, *A Nested Transaction Mechanism for LOCUS*, ACM, Vol. 17, No. 5, Oct. 1983, PP 71-89
- Mye 78 Myers, G. J., *Advances in Computer Architecture*, John Wiley and Sons, 1978
- Nie 77 Nielsen, Norman R., *Dynamic Memory Allocation in Computer Simulation*, Comm. of the ACM, Vol 20, No 11, November 1977, pp 864-873
- Ous 79 Ousterhout, John K, and Donald A. Scelza, and Pradeep S. Sindhu, *Medusa: An Experiment in Distributed Operating System Structure (Summary)*, ACM, 1979, pp 115-116
- Ols 83 Olson, Rovbert A., B. Kumar, and Leonard Shar, *Messages and Multiprogressing in the ELXSI System 6400*, IEEE, 1983, pp 21-24
- Pan 76 Panigrahi, G., *Charge-Coupled memories for Computer Systems*, *Computer*, April 1976, pp 33-42
- Par 81 Parker, D. Stott, Gerald J. Popek, Gerald Rudisin, Allen Stoughton, Bruce Walker, Evelyn Walton, Johanna Chow, David Edwards, Stephen Kiser, and Charles Kline, *Detection of Mutual Inconsistency in Distributed Systems*, Proceedings 5th Berkeley Workshop on Distributed Data Management & Computer Networks, Calif., Feb. 81.
- Pat 81 Patel, Janak H., *A Performance Model for Multiprocessors with Private Cache Memories*, IEEE, 1981, pp 314-316
- Par 72 Parnas, D. L., *On the Criteria To Be Used in Decomposing Systems into Modules* CACM, Vol. 15, No. 12, Dec. 1972 pp 1053-1058
- Pat 82 Patel, Janak H., *Analysis of Multiprocessors with Private Cache Memories*, IEEE Transactions on Computers, Vol. C-31, No. 4, April 1982, pp 296-3
- Patt 82 Patterson, David A., *A RISCy Approach To Computer Design*, IEEE Spring Compcon 82, Feb 1982, pp 8-14
- Poh 75 Pohm, Arthur V., OM P. Agrawal, and Ronald N. Monroe, *The Cost and Performance Tradeoffs of Buffered Memories*, Proceedings of the IEEE, Vol. 63, No. 8, August 1975, pp 1129-1135

- Pop 83 Popek, Gerald J., and Greg Thiel, *Distributed Data Management Issues in the LOCUS System*, UCLA.
- Pop 83 Popek, Gerald J., and Bruce J. Walker, *Transparency and its Limits in Distributed Operating Systems*, UCLA.
- Pop 83 Popek, G., B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel, *LOCUS A Network Transparent, High Reliability Distributed System*, UCLA.
- Pur 70 Purdom, Paul W. Jr. and Stigler, Stephen M, *Statistical Properties of the Buddy System Journal of the ACM*, Vol 17, No 4, October 1970, pp. 683-697
- Ran 79 Randell, B., *A Note on Storage Fragmentation and Program Segmentation*, Journal of the ACM, Vol 12, No 7, July 1979
- Rao 78 rao, Gururaj S., *Performance Analysis of Cache Memories*, Journal of the Association for Computing Machinery, Vol. 25, No. 3, July 1978, pp 378-395
- Ree 83 Reed, David P., *Implementing Atomic Actions on Decentralized Data*, ACM Transactions on Computer Systems, Vol. 1, No. 1, Feb. 1983, pp 3-23
- Rei 72 Reiter, A, *A Resource-Oriented Time-Sharing Monitor, Software-Practice and Experience*, Vol 2, 1972, pp 55-71
- Ric 83 Richardson, M. F., and R. M. Needham, *The TRIPOS Filing Machine, a Front End to a File Server*, ACM, Vol. 17, No. 5, Oct. 1983, pp 119-128
- Rod 76 Rodriguez-Rosell, Juan, *Empirical Data Reference Behavior in Data Base Systems*, Computer, Nov. 1976, pp 9-13
- Rya 74 Ryan, Thomas A. Jr. and Coffman, Edward G. Jr., *A Problem in Multiprogramming Storage Allocation*, IEEE Transactions on Computers, vol. c-23, No.11, November 1974, pp 1116-1122
- Sad 75 Sadeh, E., *An Analysis of the Performance of the Page Fault Frequency (PFF) Replacement Algorithm*, Proc. of the 5th Symp. on Operating System Principles, ACM SIGOPS, Nov. 1975, pp6-13
- Sai 81 Saito, Masato, *Evaluation of Cache Memory, Store Buffers and Memory Interleaving on Single processor Systems and Their Application to Multiprocessor System Environments*, Nec Research & development, No. 62, July 1981, pp 65-77
- Sau 79 Sauer Charles H., and K. Mani Chandy, *The Impact of Distributions and Disciplines on Multiple Processor Systems*, Communications of the ACM, Vol. 22, No. 1, Jan. 1979

- Ste 83 Stephenson, C. J., *Fast Fits : New Methods for Dynamic Storage Allocation*, ACM, Vol 17, No. 5, Oct. 1983, pp 30-32
- Sto 83 Stonebraker, M., *Virtual Memory Transaction Management*, Memorandum No. UCB/ERL M83/74, Electronics Research Laboratory, University of California, Berkeley, December 19, 1983
- Svo 76 Svobodava, L., *Computer Performance Measurement and Evaluation Methods: Analysis and Applications*, Elsevier North-Holland, Inc., 1976
- Tec 83 Techne Software Corp., *Cache/Q by Techne the Software Accelerator*, advertisement, 1983
- Teo 72 Teorey, T.J. and Pinkerton, T.B., *A Comparative Analysis of Disk Scheduling Policies*, CACM, Vol 15, No 3, March 1972, pp 177-184
- Tog 82 Togami, Yuji, *Megneto-Optic Disc Storage*, IEEE Transactions on Magnetics, Vol. Mag-18, No. 6, November 1982, pp 1233-1237
- Tra 79 Traiger, I. L., Gray, J.N., Galtieri, C.A., Lindsay, B.G., *Transactions and Consistency in Distributed Database Systems*, IBM Research Report, RJ2555(33155), 6/5/79
- Tun 70 Tung, Chin, *On the Apparent Continuity of Processing in a Paging Environment*, IEEE transactions On Computers, vol. c-19, No.11, November 1970, pp 1047-1054
- Tur 77 Turner, R. and Strecker, B., *Use of the LRU Stack Depth Distribution for Simulation of Paging Behavior*, CACM, Vol 20, No 11, November 1977, pp 795-798
- Wal 83 Walker, Bruce J., and Gerald J. Popek, *The LOCUS Distributed File System*, UCLA.
- Wal 83 Walker, B., G. Popek, R. English, C. Kline, and G. Thiel, *The LOCUS Distributed Operating System*, ACM, Vol. 17, No. 5, Oct. 1983, pp 49-70
- Wet 76 Wettstein, Karlsruhe H., *Ein integriertes Hardware-Software-System zur Steuerung der Ein/Ausgabe*, Computing, Sept. 1975, pp 1-20
- Wil 82 Wilkes, M.V., *The Processor Instruction Set*, Proc. of 15th Annual Workshop on Microprogramming, Oct 5-7, 1982, pp 3-5
- Whe 73 Wheeler, T.F. Jr., *IBM OS/VS1 - An Evolutionary Growth System*, National Computer Conference, 1973, pp 395-400
- Ypm 75 Ypma, John E., *Bubble Domain Memory Systems*, National Computer Conference, 1975, pp 523-528

- Sca 81 Scales, H.L., *Implementing a Virtual Memory System Using the MC68451 Memory Management Unit*, Wescon/81 Professional Program Session record 9
- She 76 Shedler, G.S. and Slutz, D.R., *Derivation of Miss Ratios for Merged Access Streams*, IBM Journ. Res. Devel., Sept 1976, pp 505-517
- She 74 Shen, K. K., and Peterson, J. L., *A Weighted Buddy Method for Dynamic Storage Allocation*, Comm. of the ACM, Vol 17, No 10, October 1974, pp 558-562
- Sho 75 Shore, John E., *On the External Storage Fragmentation Produced by First-Fit and Best-Fit Strategies*, Comm. of the ACM, Vol 18, No 8, Aug 1975, pp 433-440
- Sho 77 Shore, John E., *Anomalous Behavior of the Fifty-Percent Rule in Dynamic Memory Allocation*, Comm. of the ACM, Vol 20, No 11, August 1977, pp 812-820
- Shr 74 Shrevastava, S. K., *A View of Concurrent Process Synchronisation*, , The Computer Journal, Vol. 18, No. 4, 1974, pp 375-379
- Smi 76 Smith, Alan J., *A Modified Working Set Paging Algorithm*, IEEE Transactions on Computers, vol.c-25, No.9, Sept. 1976, pp 907-913
- Smi 78 Smith, Alan Jay, *Directions for Memory Hierarchies and Their Components: Research and development*, IEEE, 1978, pp 704-709
- Smi 78 Smith, Alan Jay, *Long Term File Migration - Part I: File Reference Patterns*, UC Berkeley, 1978
- Smi 78 Smith, Alan Jay, *Long Term File Migration - Part II: File Replacement Algorithms*, UC Berkeley
- Smi 78 Smith, Alan Jay, *Sequential Program Prefetching in Memory Hierarchies*, IEEE, Dec. 1978, pp 7-21
- Smi 79 Smith, Alan Jay, *Characterizing the Storage Process and Its Effect on the Update of Main Memory by Write Through*, Journal of the Association for Computing Machinery, Vol. 26, no. 1, Jan. 1979 , pp 6-27
- Spi 72 Spirn, J.R. and Denning, P.J., *Experiments with Program Locality*, , Fall Joint Computer Conference, 1972, pp 611-621
- Spi 77 Spirn, J.R., *Program Behavior: Models and Measurements*, Elsevier North-Holland, Inc, 1977
- Spi 76 Spirn, J, *Distance String Models for Program Behavior*, Computer, Nov 1976, pp 14-20

