An Intelligent Remote File Server

Kim Korner

AI TR86-39 December 1986

Support for this research was provided by the Army Research Office, under grant number ARO DAAG29-84-K-0060.
AN INTELLIGENT REMOTE FILE SERVER

APPROVED BY SUPERVISORY COMMITTEE:

[Signatures]
To those still in the throes.
AN INTELLIGENT REMOTE FILE SERVER

by

KIM M. KORNER, B.A.

DISSERTATION

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December, 1986
ACKNOWLEDGEMENTS

I am very grateful to all the members of my committee. Particular thanks are due to Mike Molloy, Jeff Brumfield and J.C. Browne. Mike Molloy and Jeff Brumfield provided the day to day encouragement, feedback, guidance and threats necessary to a successful dissertation attempt. Against this almost daily chore was the measured counterpoint of J.C Browne who took time from his busy schedule to offer support, perspective and direction.

Words fail in acknowledging my debt to Ija Korner for initially setting my feet on the path of knowledge. We share the joy of that quest. In large measure, this one is for him. My parents, Jean and Jim have been unflinching in their long support of this effort. Without their warmth and love, this would not have happened.

Writers of dissertations are often cruel and unusual human beings. It comes as a considerable surprise to discover how kind and forgiving one's friends can be despite the confusion of the situation. It is no surprise at all to discover how much richer and warmer life is as a result of their kind attentions. They know who they are and no thanks are adequate.

This research was supported by contract DAAG29-84-K-0060 from the U.S. Army Research Office.

Kim M. Korner

The University of Texas at Austin
December, 1986
PREFACE

In every circle, and truly, at every table, there are people who lead armies into Macedonia. ... These are great impediments to those who have the management of affairs. ... I am not one of those who think that commanders ought at no time to receive advice. ... If, therefore, anyone thinks himself qualified to give advice respecting the war which I am to conduct, which may prove advantageous to the public, ... he shall be furnished with a ship, a horse, a tent; even his travelling charges shall be defrayed. But if he thinks this too much trouble, and prefers the repose of the city to the trials of war, let him not, on land, assume the office of a pilot.

--Lucius Aemilius Paulus

Computer Science is increasingly a field in which civilian generals wage war gathered about coffee tables. Revered practitioners boast proudly, "I'm a theoretician - I haven't touched a machine in years." Such attitudes have led to a wide gulf between the theory of computer science and its practice. This is wrong.

The great legacy which is ours as scientists comes to us from predecessors who were brilliant theoreticians and who were possessed of a burning desire to solve problems and answer questions. The desire for solutions overcame whatever qualms there may have been about the dirtying of hands in research and the wrestling with the technologies necessary to obtain *real* rather than *toy* solutions. While computer science has more than its fair share of theory, it seems curiously short on burning questions.

The foregoing is not to suggest that this study probes to the heart of any burning questions central to computer science. This is the modest exercise of a fledgling computer scientist with a few of the right motivations and the barest set of notions about their specific application.

Let us as computer scientists together roll up our sleeves and resolve to get dirty. Theory is important, but theory unattached to the base data of the science is aimless philosophy. Disciplines within computer science that content themselves with "toy" problems, that always reside "just this side" of combinatorial explosion, that explain away all problems with the deus ex machina of improved future computing technologies all need to abandon their comfortable theoretical enclaves and pursue real issues which don't conveniently stop when the going becomes difficult. Let us cease to be armchair philosophers and behave as curious, questing computer scientists.
ABSTRACT

Limitations of current disk block caching strategies are discussed. A new model for providing remote file service using knowledge based caching algorithms is proposed. The knowledge based algorithms generate expectations of user process behavior which are used to provide hints to the file server. Surplus resources of the remote file server permit incorporation of these hints into caching algorithms. The research involves gathering trace data from a modified Unix kernel and trace driven simulation of remote file server models. Comparisons are made between conventional, knowledge based and optimal models. Further applications of knowledge based strategies in operating systems are discussed.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td>v</td>
</tr>
<tr>
<td>Preface</td>
<td>vi</td>
</tr>
<tr>
<td>Abstract</td>
<td>vii</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>viii</td>
</tr>
<tr>
<td>Chapter 1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1. Operating Systems and the Incursion of Intelligent Algorithms</td>
<td>1</td>
</tr>
<tr>
<td>1.2. Bridging Levels of Abstraction</td>
<td>3</td>
</tr>
<tr>
<td>1.3. Case in Point — An Intelligent File Server</td>
<td>7</td>
</tr>
<tr>
<td>1.4. The Current Study</td>
<td>11</td>
</tr>
<tr>
<td>Chapter 2. Literature Review</td>
<td>12</td>
</tr>
<tr>
<td>2.1. Introduction</td>
<td>12</td>
</tr>
<tr>
<td>2.2. Artificial Intelligence and Operating Systems</td>
<td>12</td>
</tr>
<tr>
<td>2.3. Remote Disk Servers</td>
<td>15</td>
</tr>
<tr>
<td>2.4. Unix</td>
<td>21</td>
</tr>
<tr>
<td>2.5. SUN NFS</td>
<td>23</td>
</tr>
<tr>
<td>2.6. Caching Studies</td>
<td>25</td>
</tr>
<tr>
<td>2.7. Research on File Types</td>
<td>27</td>
</tr>
<tr>
<td>2.8. Queueing Model Studies</td>
<td>28</td>
</tr>
<tr>
<td>2.9. Miscellaneous</td>
<td>30</td>
</tr>
<tr>
<td>Chapter 3. Design</td>
<td>34</td>
</tr>
<tr>
<td>3.1. General Approach</td>
<td>34</td>
</tr>
<tr>
<td>3.2. Strategy</td>
<td>35</td>
</tr>
<tr>
<td>3.3. Data Collection</td>
<td>35</td>
</tr>
<tr>
<td>3.4. Initial Analysis</td>
<td>37</td>
</tr>
<tr>
<td>3.5. Trace Analysis Algorithms</td>
<td>39</td>
</tr>
<tr>
<td>3.7. Simulations</td>
<td>40</td>
</tr>
<tr>
<td>3.7.1. Intelligent Simulation</td>
<td>40</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 4-1: Cache Sizes - Number of Units  87
Table 5-1: Validation Results  83
Table 5-2: Response Times (msec.) Simulated Process Access 87
           Sequence Data Set A
Table 5-3: Response Times (msec.) Simulated Process Access 88
           Sequence Data Set B
Table 5-4: Response Times (msec.) Simulated Process Access 89
           Sequence Data Set C
Table 5-5: Cache Hit Ratios - LRU Algorithm Simulated Process 96
           Access Sequence Data
Table 5-6: Cache Hit Ratios - Optimal Algorithm Simulated Process 96
           Access Sequence Data
Table 5-7: Cache Hit Ratios - Intelligent Algorithm Simulated Process 97
           Access Sequence Data
Table 5-8: Simulated Process Access Sequence Data Set A Simulation 98
           Resource Utilization (%)
Table 5-9: Simulated Process Access Sequence Data Set B Simulation 98
           Resource Utilization (%)
Table 5-10: Simulated Process Access Sequence Data Set C Simulation 99
            Resource Utilization (%)
Table 5-11: Simulated Process Access Sequence Data Set A Factor 101
            Analysis Response Times (msec.)
Table 5-12: Simulated Process Access Sequence Data Set B Factor 102
            Analysis Response Times (msec.)
Table 5-13: Simulated Process Access Sequence Data Set C Factor 103
            Analysis Response Times (msec.)
Table 5-14: Response Times (msec.) Simulated load Data Set A 111
Table 5-15: Response Times (msec.) Simulated Load Data Set B 112
Table 5-16: Response Times (msec.) Simulated Load Data Set C 113
Table 5-17: Cache Hit Ratios - LRU Algorithm Simulated Load Data 120
Table 5-18: Cache Hit Ratios - Optimal Algorithm Simulated Load Data 120
Table 5-19: Cache Hit Ratios - Intelligent Algorithm Simulated Load Data 121
Table 5-20: Simulated Load Data Set A Simulation Resource Utilization (%) 122
Table 5-21: Simulated Load Data Set B Simulation Resource 123 Utilization (%) Table 5-22: Simulated Load Data Set C Simulation Resource 123 Utilization (%) Table 5-23: Simulated Load Data Set A Factor Analysis Response 128 Times (msec.) Table 5-24: Simulated Load Data Set B Factor Analysis Response 127 Times (msec.) Table 5-25: Simulated Load Data Set C Factor Analysis Response 128 Times (msec.) Table A-1: Terminal Sessions 142 Table A-2: Inode Activity Types 155 Table A-3: Block Activity Types 156 Table B-1: Trace File Format 158 Table B-2: Simulation Parameters (msec.) 163
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>Unix Interface Organization</td>
<td>38</td>
</tr>
<tr>
<td>3-2</td>
<td>IFS Functional Decomposition</td>
<td>42</td>
</tr>
<tr>
<td>4-1</td>
<td>Trace Log Example</td>
<td>50</td>
</tr>
<tr>
<td>4-2</td>
<td>Pass 1 Summary Example</td>
<td>54</td>
</tr>
<tr>
<td>4-3</td>
<td>Annotated Session Log Example</td>
<td>54</td>
</tr>
<tr>
<td>4-4</td>
<td>Trace Log Formats</td>
<td>55</td>
</tr>
<tr>
<td>4-5</td>
<td>Trace Log after Second Pass - Example</td>
<td>56</td>
</tr>
<tr>
<td>4-6</td>
<td>Session Trace Processing</td>
<td>57</td>
</tr>
<tr>
<td>4-7</td>
<td>Generalization Algorithm Example</td>
<td>59</td>
</tr>
<tr>
<td>4-8</td>
<td>Trace Lamination Processing</td>
<td>60</td>
</tr>
<tr>
<td>4-9</td>
<td>Merging of Terminal Sessions by Session Offset</td>
<td>61</td>
</tr>
<tr>
<td>4-10</td>
<td>Composite Trace Example</td>
<td>62</td>
</tr>
<tr>
<td>4-11</td>
<td>Simulation Overview</td>
<td>64</td>
</tr>
<tr>
<td>4-12</td>
<td>Queueing Network Model</td>
<td>66</td>
</tr>
<tr>
<td>4-13</td>
<td>Transaction Flow Diagram LRU Model</td>
<td>68</td>
</tr>
<tr>
<td>4-14</td>
<td>Transaction Flow Diagram Optimal Model</td>
<td>70</td>
</tr>
<tr>
<td>4-15</td>
<td>Transaction Flow Diagram Intelligent Model</td>
<td>72</td>
</tr>
<tr>
<td>4-16</td>
<td>IFS Cache Partitioning</td>
<td>74</td>
</tr>
<tr>
<td>4-17</td>
<td>Experimental Simulation Design</td>
<td>79</td>
</tr>
<tr>
<td>4-18</td>
<td>Factor Analysis Simulation Design</td>
<td>81</td>
</tr>
<tr>
<td>5-1</td>
<td>Validation Error Frequency Distribution</td>
<td>84</td>
</tr>
<tr>
<td>5-2</td>
<td>Read Transactions Simulated Process Access Sequence Data Set A</td>
<td>90</td>
</tr>
<tr>
<td>5-3</td>
<td>Write Transactions Simulated Process Access Sequence Data Set A</td>
<td>91</td>
</tr>
<tr>
<td>5-4</td>
<td>Read Transactions Simulated Process Access Sequence Data Set B</td>
<td>92</td>
</tr>
<tr>
<td>5-5</td>
<td>Write Transactions Simulated Process Access Sequence Data Set B</td>
<td>93</td>
</tr>
<tr>
<td>5-6</td>
<td>Read Transactions Simulated Process Access Sequence Data Set C</td>
<td>94</td>
</tr>
<tr>
<td>5-7</td>
<td>Write Transactions Simulated Process Access Sequence Data Set C</td>
<td>95</td>
</tr>
<tr>
<td>5-8</td>
<td>Factor Analysis Read Transactions Simulated Process Access Sequence Data Set A</td>
<td>104</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

It is the intent of this dissertation to explore the utility of intelligent algorithms in remote disk server applications. These intelligent algorithms are distinguished by the use of knowledge gathered at a conceptually "high" level of the operating system and applied to very "low" level operations. The efficiency of the resulting system is enhanced due to the application of knowledge to the algorithms which govern primitive operations. The system is also inherently "more fair." Consideration is given to other areas in which intelligent algorithms may prove useful to operating system design.

1.1 Operating Systems and the Incursion of Intelligent Algorithms

Generally speaking, computer operating systems are reactive in nature -- demands are placed upon system resources and the operating system attempts to coordinate, order and satisfy the demands in some fashion, guided by some externally defined criteria of optimality. The reactivity is a direct result of the system's inability to predict future demand.

Most solutions to the dilemma of being unable to predict future demand (but needing to do so in any case) rely upon the use of modal or median values (which will, on the average, be correct predictions). Where variances are high, this solution is a poor one -- but still better than none at all. The modal or median values can be selected from varying periods in the past (and can be combined and weighted in various fashions) to allow for trends as the nature of the demands fluctuate. In large part, the attractiveness of average predictors is to be found in their low computational costs.
Artificial restrictions can be placed upon services provided to reduce variance. When service requests cease to be "average," the offending requestor may be denied service or shifted to another provider of services within whose context the offender is more average. In this fashion, variance is reduced and the predictive value of the canonical model is improved. As an example, consider a task in a multitasking operating system which may be removed from one service queue to another if its service demands diverge from those expected of tasks in that queue. This restriction of variance considerably extends the range of canonical models -- but at the cost of increased complexity in providing service. Such techniques are also not universally applicable (e.g., provision of non-preemptable resources pose certain distinct problems).

Average predictors fail badly (i.e., frequently fail to predict correctly) when variance is high (not tightly clustered about a mean or not contained in a single modal value). When the expense of failure is high, alternate methods of dealing with the uncertainty of the future become attractive.

When these alternate methods are computationally more expensive than those involving average predictors, such expense can be traded off against the high cost of incorrect prediction. Deadlock handling provides an interesting example. If, in a given situation, deadlock detection and resolution are deemed unacceptable then deadlock prevention must be employed. Prevention algorithms often are based on worst case scenarios which may idle significant percentages of possibly expensive resources or dominate other resources (such as communication) before deciding how to handle a request. If one could obtain accurate predictions of future resource demand, expensive resources could be more fully utilized. The increased costs of computing accurate predictions could be directly traded off against the savings realized in more efficiently used resources.

On the other hand, if the costs of more accurate prediction can be paid for in inexpensive coin, a different kind of economy can be invoked for abandoning canonical models. Recent shifts in technology (primarily the proliferation of cheap powerful processors in
networked environments and dramatic decreases in the price of fast short term memory) have provided surpluses of computing power which are generally untapped.

In a traditional mainframe environment, CPU cycles and memory are in constant demand. When used by the system, resources are unavailable to users. Any extra work done by the system is at the expense of the user community. In a distributed environment of single user work stations, unused computing resources are squandered. The organization of the environment is such that it is not economically attractive to put these surplus resources to direct use in performing user tasks (though the developers of truly distributed operating systems might take exception). The surplus resources are a potential rich bonanza for the system; and one of the primary uses could be in enhancing the predictive abilities of the system.

1.2 Bridging Levels of Abstraction

Prediction of random events in the future would require major rewriting of all the laws of causality – not a minor undertaking. It is unlikely that computer systems based upon such prediction are possible. But not all future events are random and such non random happenings one can predict with greater accuracy.

Consider a group of people in a building. Where will they be tomorrow? The canonical approach would examine the type of building, watch its occupants for a period of days and if sufficient unanimity of action were demonstrated, conclude a general expectation. An office building would be very well handled (5 days out of 7), but what of an airport? Yet the behavior of people at airports is clearly not random -- employees will return repeatedly, passengers will generally be found near the destinations specified on their tickets for the next couple of days and visitors seeing off or meeting passengers will probably be found in the same city with which they share the airport.

In the example above, greater accuracy of prediction is achieved by bringing to bear
knowledge about the behaviors of concern. What one knows about airports and the purposes they serve helps to divide people into differently behaving groups and to infer the probable behavior of each group. Note that inferring the likely behavior of a group member may rely upon different kinds of information — one need only know that an individual is an airport employee to guess that he'll be back to work on succeeding days, while one needs to look at the traveller's ticket to predict where he's likely to be. Note that while quality of prediction has been increased, predictions are still probabilistic in nature.

The above example illustrates the power of knowledge based approaches over canonical approaches. This power stems from deriving the underlying reasons for occurrences and thus transforming seemingly random events into the domain of well understood, purposeful behavior. Knowledge about why something is being done permits much better inference about what will be done next.

Due to the general acceptance of modular designs in constructing operating systems (and the widespread practice of designing modules which conform to horizontal stratifications of functionality), knowledge about the behavior of the system is highly stratified. This stratification isolates knowledge in respective layers which could be valuable to other layers. This isolation of knowledge at different levels of abstraction makes the task of enhancing prediction by application of knowledge difficult — but also explains why this approach has not been more rigorously investigated.

Users of computers perform operations at the highest level of abstraction available within the man-machine system. Users devote most of their cycles (in so far as humans can be said to have cycles) to activities of this nature. It is (presumably) at this level at which global plans, motivation and tasks interface to the "real" world. Users typically have goals relating to their use of the computer and little concern about how those goals are attained by the machine.

The computer's operation (primarily as manifested by the operating system) is at the
opposite end of the continuum of abstraction. Absent are any of the whys and hows the user deals with. Instead, the computer spends most of its time dealing with specific imperatives seemingly unrelated to other imperatives. Only rarely is behavior in one part of the operating system coupled to behavior in another part — and then rarely on the basis of knowing that the two activities are purposefully related.

Between these two extremes is a considerable spectrum. As one moves further away from the user's level of abstraction, actions performed by the system for the user "make less sense" and are less connected to global concepts guiding the user's behavior. Software designed to interface with users mirrors many of the users' abstractions in its algorithms and data structures. Operating system code that actually interfaces to hardware (as opposed to interfacing to other code) is organized around the imperatives of the machine architecture and deals with a much less sensible world — demands placed upon it appear much more random because they are stripped of the context of higher level abstractions.

The bulk of the operating system's consumption of machine resources involves operations at low levels of abstraction. If one is to improve operating system performance, the work must be done at these levels. If one could apply some of the context lost as one descends through the levels of abstraction, the greater knowledge would permit more efficient algorithms to be used. These algorithms would use knowledge about the context in which demands are placed upon them to decrease the apparent randomness of these demands.

Does this mean one needs to abandon traditional stratified designs for operating systems and pass full information throughout the system? Obviously not. Design, construction and maintenance of operating systems software is already too complex an undertaking, stressing the state of the art. Adding major complexity to the process is a poor idea at best. Major restructuring of systems is not required.

What is desirable is the bridging of levels of abstraction; discovering where higher level knowledge is concentrated and piping that down past intervening levels to the low level
algorithms where it can be most useful. The provision of specific purpose data paths between modules does not seriously jeopardize the safeguards inherent in modular hierarchical design methodologies.

A related issue deals with software reliability. Low level routines must be very reliable and robust — many higher level routines depend upon them. Higher level software tends to be more complex and less depended upon. If one links high level, possibly unreliable, software with low level, critical functions, how is one to ensure correct performance? Low level routines must be distrustful of information from higher levels of abstraction. This can be achieved by making the high level information be advisory rather than imperative. One restores the lost context at lower levels through hints from higher levels.

To summarize:

- One predicts better when one has knowledge about the underlying causality of events.

- Knowledge based approaches are more complicated than statistical approaches to prediction of "random" events — but this complication yields much better results and can often be paid for in "cheap" resources.

- This knowledge (in operating systems) clusters nearest the user of the system but is needed by algorithms furthest from the user.

- Without weakening the technique of modular hierarchical design, one can pipe high level knowledge to the low level routines which can benefit most from it. This is referred to as bridging levels of abstraction.

- Information from higher levels of abstraction should be in the form of hints such that lower levels may provide assurances of correct functionality. Hints provide absent context which can be used to improve performance; otherwise they may be ignored.
1.3 Case in Point -- An Intelligent File Server

File servers provide an excellent testbed for application of knowledge based algorithms. File servers are generally based upon rotating magnetic media – disks. Waiting for blocks of data from rotating media is temporally expensive. Making the media faster becomes prohibitively expensive very quickly. Inexpensive rotating media and more expensive random access memory (RAM) are frequently combined in caching schemes to provide performance intermediate to both.

Caching relies upon repeated access to data – data read from a disk once can be retained in RAM and future requests for the same data can be rapidly satisfied. If one has expectations of future requests, slack periods can be utilized to pre-fetch into RAM data which will be needed later. Because the RAM cache is much smaller than the disk, not everything read can be cached indefinitely – decisions must be made about what to cache and for how long. But such decisions must be made on the basis of activity yet to be performed. For caching to work well, it must correctly anticipate the future.

As has been previously discussed, typical approaches to the need to predict the future are canonical – and disk caching is no exception. Early students of operating systems were quick to observe that data access throughout the system frequently observes a convenient temporal locality. That is, recently used data are more likely to be needed again in the near future while data unused for longer periods are unlikely to be reused. This temporal locality is the canonical model for predicting data access in a disk cache; recently accessed data are cached and as space is needed to cache new references, data which have been unused longest are purged from the cache to provide space. This least recently used (LRU) algorithm is at the heart of most disk caching systems and for the most part it works quite well.

In two specific instances LRU does not work well. If data are not reused, they uselessly consume cache space until they become old enough to be purged from the cache. If data are accessed several times in a sequential fashion, the start of a data sequence can be
aged from the cache while recent references are retained — yet these purged data will be
needed before the data retained in the cache. In this case, LRU is the worst possible strategy.

It would be pleasant to be able to recognize when these situations are in effect and
modify caching strategies accordingly. Regrettably, this means abandoning the pure and
uncomplicated LRU strategy in favor of something much more complex with greater overhead.
This increased overhead is largely responsible for the absence of experimentation in alternate
caching strategies; traditionally, disk caching has been performed in mainframe environments
where increases in system overhead translate directly to loss of computing resources for users.

Network file servers suffer from no such tradeoff. Processing power is structured to
meet peak demands (which directly implies a surplus during non-peak times), can be
inexpensively expanded and its use in "overhead" tasks is not begrudged by its user
community. Resources are available for more costly caching strategies if they can be
effectively devised.

Aside from the spare computing resources to be found in file server environments
which allow more powerful prediction algorithms, file servers have several characteristics
which make them attractive objects of study.

- File servers are isolated by networks from other components of computing systems.
  This simplifies the task of analysis and leads to clearer results. Analysis of caching
  on mainframes has always been complicated by the tight coupling of user
  computing resources and those required for caching.

- File servers are an expensive shared resource. Improvements in their performance
  will yield real savings to an extensive user community.

More effective caching strategies are made possible by bridging levels of abstraction.
A network file server responds to requests at a disk block level — permissible operations
involve reading or writing single disk blocks. Knowledge which would serve to organize and
make sense of these low level requests is absent from this level of the operating system. But
elsewhere in the system is information which structures and, to a large extent, permits
prediction of block requests.
In many computer systems, a small number of system utility programs account for most of the user load placed on the system. Editors, compilers, text processors, linkers, loaders, data base subsystems and the like spring immediately to mind. These programs are well behaved and predictable with respect to their file operations. A typical compiler may scan its input file once, generate an intermediate file, read sequentially through this intermediate file several times, and finally generate a file of relocatable code. Knowledge about a program and its use of files permits prediction of file access patterns. This prediction of access patterns is what the file server so desperately needs to structure the seemingly random set of block requests.

Note that information about both programs and files is required. Simply knowing about one or the other is insufficient. A file has different access patterns across different programs (consider a source program being edited and then compiled) and a program can treat files quite differently (a loader reads object modules, scans libraries, generates absolute modules, reads initialization files, etc.). Knowledge about a program and the bindings of files it references to functions the program performs are the key to understanding access patterns. Both files and programs can be considered to have types. File access patterns are largely explainable by the interaction of file type and program type.

Note also that to a large extent access patterns are very difficult to discern from the static code of the program under consideration. The easiest (and for those denied access to source code of system utilities, only) means of determining program file access behavior is by observing its behavior. One exercises the program over a representative data set and generalizes about file types (grouping similar files together) and access patterns for each type. These generalizations then apply to subsequent invocations of the program.

Thus one can build a set of generalizations about the file access behavior of frequently used programs. These generalizations can then be used to anticipate access patterns when, at the time of actual execution, a file is opened for use and thus to provide hints to the file server.
A knowledge based approach also suggests what should be in the cache in the absence of other cache activity. As soon as one starts distinguishing between different types of information which exist in the cache, some types are seen to be of general use (i.e., of utility to all processes as opposed to specific processes). Directory information along common paths (in the Unix world, directories clustering around the root directory) and initial page sets of commonly used programs have a high likelihood of being used by all users. In slack periods where disk utilization is very low and cache contents have decreasing likelihood of being useful, a file server can start refilling the cache with this type of generally useful information. This is possible because one knows what has general utility and what has specific utility.

Thus, after a period of inactivity, the cache will be ready to respond rapidly to requests for executable blocks of commonly used programs and directory information of commonly traversed paths. A random process will have a very good chance of having many of its initial requests satisfied out of cache and as it continues execution, it can provide hints to allow the cache to respond intelligently to its specific behaviors.

Prefetching blocks of sequential files can also be more intelligently handled in the file server. It makes little sense to clutter the cache and extend the disk service queue in order to satisfy a prefetch request for a block which will be needed at some indeterminate period in the future while service demands are high. However, if the server is relatively idle, such prefetching makes a great deal of sense. Hints specifying blocks which could be prefetched on a resources available basis provide a solution to this dilemma. The server is given useful tasks to which it can turn its idle resources (if any) without being forced into load dependent counterproductive activity.
1.4 The Current Study

The current study is an attempt to investigate the above assertions. Trace data is collected from a modified Unix kernel. The trace data spans the desired level of abstraction—logical file operations and their physical file consequences are captured.

On the one hand, this trace data is fed to an intelligent analysis program which builds generalizations about file access patterns. On the other hand, this trace data drives simulations of file servers.

Three file server models are simulated:

- A generic "dumb" server using LRU caching methodologies.
- An optimal server which has precise knowledge of future file operations with which it can optimize cache operations. Note that this optimality applies only to cache replacement performance.
- An "intelligent" server of the sort previously proposed.

These three simulations provide a continuum in their use of predictor information. The "dumb" model marks what is currently done (a canonical approach). The optimal model sets a limit of what performance can be obtained with perfect predictors. The "intelligent" model demonstrates the gains of intelligent algorithms over conventional canonical ones. These gains are achieved by better cache management and knowledge based prefetching.

A more comprehensive discussion of the study is reserved for the Design and Method chapters.
Chapter 2

Literature Review

2.1 Introduction

While there is a distinct lack of literature pertaining directly to intelligent file servers, there are branches of the literature which apply to certain aspects of this study.

2.2 Artificial Intelligence and Operating Systems

In that this study is a marriage of AI concepts to OS problems, it is perhaps meaningful to review some of the few studies which have attempted similar unholy alliances.

Reiner [Reiner 80] explores "the role of learning techniques, motivated by approaches in artificial intelligence, in the context of operating system control." Working in a Univac 1108 environment, Reiner builds a system which experimentally determines cause and effect of parameter choice on the load performance space. These relationships are then used to tune system parameters adaptively in production. Artificial intelligence techniques are used to derive the effects of parameter variation from highly variant data and cross checked by rigorous statistical methods. Future effort is suggested for fully automating the system, performing simultaneous modification of several system parameters and automatic detection of dominant parameters. This study is a very productive application of AI techniques within the Operating System domain which yielded very real and positive performance results.

Levine [Levine 84a] deals with applications of expert systems to the domain of performance modelling. A great deal of this paper is involved in describing expert systems to those not familiar with them. This introduction is well done if a bit basic and is followed by a
rather uncomplicated description of performance model validation. Some interesting simplifications are possible due to the domain (limiting validation uncertainty by constraining the original modelling, automatic generation of correct models from fundamental model decisions, elimination of probabilistic reasoning and backtracking) — it is quite possible that these simplifications limit the ultimate power of this system. The high level design of the expert system is plausible if a bit vague and brief. Such a system if actually constructed would probably be useful.

In his B.A. Honors Thesis, he [Levine 84b] amplifies upon the above article. While still stopping short of implementation, he clarifies his domain (validation limited to analytic models based on data gathered from IBM MVS environments by a BGS data reduction tool CAPTURE/MVS, models to be validated on a queueing network solver, again a BGS product, BEST/1) and sets forth a detailed design for the expert system functions. A good deal of the expert knowledge acquisition seems to have been done. The simplifying assumptions are further justified. While not terribly familiar with the BGS products which play so critical a role in this proposal, it would seem that the generality of this proposed system is much less than it would at first appear — only validation of analytic models for a very specific OS is to be supported. This is still an ambitious project, but one which seems potentially rewarding. Extension from this very circumscribed domain to simulated models and/or other OSs would appear very difficult — a typical problem of expert systems.

Ennis et. al. [Ennis 84] are also interested in applications of expert systems. In this case a real time expert system (an unusual system) is used to monitor and control (frequently in conjunction with a human operator) a mainframe running a typically convoluted IBM operating system. The need for expert system assistance in this area is well supported (scarcity of good operators, need for high quality immediate decision making, desire for uniform operational characteristics across operators). Interface of the operating system to the expert system seemed not terribly involved. The actual achievements of this system seemed a bit trivial, especially in the face of claims for the capabilities of the next generation of the system. It is my understanding that the ambitious goals of the next generation of the system
were not met when the additional complexity of the expert system outstripped the dedicated expert system's computational resources (a full IBM 370 system) - causing the system to cease being "real time."

A companion article [Griesmer 84] gives a brief overview of the previously described expert system but concentrates on the design and implementation issues of interest to AI practitioners. Of particular interest are the modifications necessary to OPS5 (an expert system development and support language) to support real time usage. These include timed productions, an OPS5 communications phase (for real time contact with the "outside world") and priority based conflict resolution. Consideration is also given to the demands imposed by the need for continuous operation. The details of the ambitious goals for the next generation of the system are elaborated (extension of the domain and inclusion of a learning component). Given the reported size of the system reported upon (~500 rules) and the extensions desired, it is not surprising that subsequent phases of the project stalled — expert systems grow easily at first up to a certain point and past that point develop only with great cost and pain (such is the price of combinatorial explosion). In general the extensions of OPS5 to a real time environment are impressive, consistent with the spirit of the language and well thought out. The strength of this study lies in these extensions rather than in its ambitious stated goals of providing real help to system operators.

Rathi [Rathi 84] directly addresses the addition of intelligence to the gap in the memory hierarchy between main and secondary memories. He posits a large multi-processor system which sits between main memory and the disk subsystem called a Self Managing Secondary Memory (SMSM). This system is viewed largely within the context of the TRAC computing environment. While the SMSM is a virtual cache for the disk subsystem, it is a distinct level within the memory hierarchy, with named objects and directories. Much of the "intelligence" of the system lies in efficient search and garbage collection algorithms shared out among its multiple processors. This is a very specialized system designed for a very particular application — the results do not generalize well to contemporary storage arrangements. Nevertheless, it does point to the need for intelligence in secondary storage management and suggests some of the performance benefits obtainable from its application.
2.3 Remote Disk Servers

Much attention has been paid in the literature to file servers. The systems discussed cover a considerable range of function and complexity. The following section serves to localize the intelligent server design within that range. It is important to note that several have been attracted to what might be considered knowledge based approaches, but all have shied away from the complexity involved in such approaches.

Lazowska et. al. [Lazowska 84] discuss performance of diskless workstations. This article provided the inspiration for the current study. The authors examine the performance of three existing systems (SUN BSD Unix, SUN V Unix and DOMAIN software on Apollo computers) and use the results to parameterize queueing network performance models. The intent of the modelling is not to build capacity planning models but rather to "enhance our understanding of the various issues arising in the design of diskless workstations and their file servers." Issues considered include: data access latency, disk block size, CPU costs, network packet size, prefetching, caching, local secondary storage, multiple servers and CPU performance. They conclude that service from a shared file server can equal that of a lightly loaded local disk, that efficiency is intimately linked to volume transfers at every interface and that performance gains are best achieved by augmenting (both in hardware and algorithm design) the shared file server. Open questions remaining at the end of the study include: the need for further characterization of file access behavior, improved caching strategies (in particular alternatives to simple LRU), the impact of high performance CPUs in file servers and the effects of paging on network file servers. These questions are directly addressed in this study.

Birrell and Needham [Birrell 80] provide a brief introduction to the design issues which underlay network file service. They consider a spectrum of file servers ranging from remote disks to complex autonomous file systems with files as abstract objects, access controls, directory structures and interlocks. They briefly explore the nature of the necessary primitives for a variety of remote file services. Tradeoffs of some design features are
suggested. This is an early article, written without the benefit of much experience in network environments. Nonetheless, it draws a useful continuum within which to evaluate file system designs.

Turner et. al. [Turner 84] discuss a DEC research implementation of a network disk block server. The target machine for the server is remarkably small -- a PDP 11/24 with disks in the RL02 range. The goals of such a server system are necessarily modest (the criteria of acceptable server performance are given as being equivalent to a local floppy drive). Much of the study deals with possible alternatives for overcoming the CPU bottle neck. Several are discussed but not conclusively demonstrated either in the actual server or the loose analytic model developed. Disk block caching is mentioned in passing in a wistful tone (... if one had sufficient spare memory ...) as a possibly effective technique of avoiding rotational latencies engendered by CPU processing tardiness. It is surprising to see this recent a study dealing with such underpowered hardware and grappling with the inevitable performance limitations of such a system.

Spector [Spector 82] is interested in efficient communication techniques in local networks. To this end, a taxonomy of remote reference strategies is developed. The claim is then made that efficiency is maximized when a remote reference model is chosen which possesses the exact degree of functionality required. Implicit in this argument is the notion that while convenient, layered protocols are inherently inefficient. By way of example, a remote reference model (Only-Once-Type-1, Primary, Processor-synchronous) is chosen, designed and implemented. Implementation occurs in two forms (software and microcode). Performance results indicate that the software implementation is roughly three times faster than the fastest layered protocol and that the microcoded version was roughly 30 times faster than the software version. It is important to note that the use of the specific model from the taxonomy permits development of the microcoded implementation. Such efficient implementations are not possible in layered protocols.

Paxton [Paxton 79] provides a theoretical framework for extending simple file servers
to provide consistent atomic transactions over multiple servers. This paper is a powerful argument against those who claim that simple file servers (of the sort proposed in the current study) cannot serve in applications where more complex file functions are desired. This article establishes that simple file servers can be designed and tested and later extended as needed.

Richardson and Needham [Richardson 83] write about a small file server on the Cambridge Distributed Computing System. The net itself is a ring of 50-60 single user machines with very limited capabilities. The file server is capable of supporting up to fifteen concurrent users. Data is cached in a buffer space of \(\sim750K\) bytes. Two cache areas are distinguished from each other — there is a directory cache (with small 512 byte blocks) and a data cache (with much larger 2048 byte blocks). The server has state (files are opened and closed) thus read ahead and asynchronous writes can be supported. A simplified directory structure is extracted from the extended directory information and cached to facilitate look up — the simplified information bears a striking resemblance to Unix directories (character string/inode pairs). Cache blocks are indexed by a hash table (for rapid access) and held in an LRU stack (for allocation/deallocation). Hit rates of 90% are claimed for data. Directory cache hit rates of 65% are also claimed. Of specific interest to this study is their statement that "there is great advantage to be gained from having a front end to a file server that takes advantage of knowledge of the patterns of use expected of a particular subset of the file server's clients" (italics mine). This appears to have been careful and thoughtful work unfortunately constrained by meager hardware resources and the necessary concerns they generate.

A companion paper [Craft 83] discusses the high level resource manager for the Cambridge Processor Bank. This seems again a case of a great deal of excellent work devoted to too little hardware and too few resources. The resources at issue are high level components of the decentralized system (e.g., an editing session on a file). The system is largely concerned with generating these high level resources from smaller more primitive ones (e.g., authentication servers, system loaders, processors, etc.). This paper deals with objects several levels of abstraction above file servers - but does make the interesting point that network
performance can be improved by noticing popular resources and preparing in advance for their next use (e.g., preloading a vacant processor with a favorite editor). It is this kind of optimization this study attempts at the file server level.

Cheriton and Zwaenepoel [Cheriton 83] describe the distributed V kernel. The kernel is intended for use in a diskless work station environment and makes use of general purpose IPC mechanisms. The use of the general IPC facility, it is claimed, can replace page level, file transfer and remote terminal protocols -- reducing the number of protocols needed by a system. Service requests are handled directly by the kernel (rather than by a network server) and passed by a "raw" ethernet data link. *File access takes three different forms: random file page access, sequential file reading and program loading.* Sequential access is optimized by prefetching and asynchronous writing. Program loading is characterized by mass transfer of entire files and while not actually done, they claim "if the file server retained copies of frequently used programs in memory, much as many current timesharing systems do, could achieve ... performance ... independent of disk speed." They claim that the processor is the key limitation in file server systems as network bandwidth is plentiful and disk resources are identical to those encountered in conventional systems. This article is of interest due to its close attention to performance of the SUN work station in a diskless environment and its speculations about limits to that performance.

The WFS file server is described by Swinehart et. al. [Swinehart 78]. WFS is characterized as providing network file service through a concise set of file operations. It is a simple, stateless server which was implemented in under two man months. Its implementation, performance, design philosophy and capabilities are discussed. Extensions to provide privacy, security and efficiency are proposed. The authors conclude that "a very simple central file server, teamed with appropriate file system elaborations in the client host, can meet or exceed many of the capabilities of more comprehensive central facilities at acceptable cost to the client." This study provides a further argument in favor of simpler file servers of the types used in this study.
The Felix File Server is the focus of the paper by Fridrich and Older [Fridrich 81]. It reports the results of a project at Bell-Northern to develop a file server supporting virtual memory for work station, data sharing and secure storage. The resulting system is quite complex. There are file sets which can be atomically committed, and extended access methods to facilitate sharing. Access to files is capability based to avoid problems of user validation. A block orientation was chosen to support virtual memory, conventional file systems and data base applications. Security and sharing are implemented at the lowest levels of the server. The server is extremely functional but performance is only cursorily touched upon. This is a very lucid example of the kind of broadly functional system that this study does not concern itself with.

A similarly complex scheme is propounded by Arden [Arden 85]. While the overall design has little application to the current study, he does propose that computationally expensive information which may later become invalid but which also may be useful is worth retaining and treating as a hint. Thus, once computed, information is sometimes worth the overhead of retention even though it may be flawed. The concept of a hint — a useful but possibly incorrect piece of information is a noteworthy one.

Another fairly involved design is presented by Sturgis et. al. [Sturgis 80]. Here the emphasis is placed upon a system of servers which appear as a single logical file system. Sharing of files while maintaining consistency, and data locking are the major issues addressed. The server they propose is transaction based. A novel locking scheme is proposed to allow more efficient local caching of data (locks may be broken when the locked data has been read but not written, the lock has been in force for a *long* time, and another transaction is requesting a lock on the locked data). Servers have remarkable latitude in aborting transactions and clients have the greater part of the responsibility for dealing with aborted or failed transactions. The server system presented would appear complete, not overly efficient but highly adaptable to a wide variety of file service demands. Other conclusions of note in this paper: file servers allow one to "vary the ratio of processing power to disk capacity over a wide range to suit a variety of applications," the file server system can
configured so that "its effective bandwidth can be matched to the usage patterns of a given environment," file server systems can be based upon *specialized computers optimized for file service* and the distribution of function takes advantage of the distribution of processing power.

Mitchell and Dion [Mitchell 82] compare two fairly complex file servers in terms of "design goals, implementation issues, performance, and their relative successes and failures." Both servers are considerably more complicated than the ones envisioned in this study. The conclusions drawn are of interest only to elaborate file server models. Conclusions for future server design tend toward even greater complexity (architectures with non-volatile memory, nested transaction capabilities and *batched* disk operations). Such are the consequences of complex design.

The ITC Distributed File System is the subject of a paper by Satyanarayanan *et al.* [Satyanarayanan 85]. In it, they describe the ongoing effort at CMU to merge personal computing and timesharing paradigms into a distributed file system. While their basic assumption of transferring (and caching) whole files about the network makes their work less applicable to this study than otherwise might be the case, they make a number of relevant observations. Workstations, they claim, have cycles to burn, thus one should pursue applications to use these spare cycles. Files have specific classes (types) which can be used "for independent optimization, and hence improved performance, in a distributed file system design." It is also claimed that more attention should also be paid to caching in general and within file servers in specific. While our fields of application are very different, the authors would seem to have a high degree of agreement with several of the principles which underlay the current study — this perhaps reflects a growing awareness of the imperatives of networked file systems in general.

Another distributed file system which relies upon transferring and caching whole files is the Cedar system at Xerox PARC described by Schroeder *et al.* [Schroeder 85]. It is strongly organized around the problem of providing an immutable shared program
development environment to groups of programmers on a local network. In this goal it succeeds fairly well. The proudly reported performance statistics, however, are pretty dismal when compared with block based file servers.

2.4 Unix

The Unix operating system has had a pervasive influence on this study. Data was collected under this system, the simulations were modeled directly upon the manner in which a specific Unix system interacts with its file servers and some reliance is made upon the naming conventions which have developed in the extended Unix community. The following articles have been critical in understanding the environment in which this study is couched.

Unix is not a single monolithic operating system but rather a class of systems undergoing simultaneous evolution on a variety of hardwares. Frequently, it is difficult to impossible to understand how a specific operating system function is accomplished in a given system without examining the code of that system — and even then the design principles remain obscure. A considerable resource in understanding the flavor of Unix systems employed in this study (4.2BSD) has been the extensive overview provided by Quarterman, Silberschatz and Peterson [Quarterman 85]. This overview provided the framework within which individual investigations into the system source code were meaningful.

Hanson, Kraut and Farber [Hanson 84] examine command usage on Unix Systems. They collected usage data from 24,000 processes and 170 active users in a document preparation and program development environment. Their extensive analysis identified core commands, usage patterns, and key factors in user errors. The results are presented in a theoretical context which suggests a generic editing model for future interface design. Of particular interest to this study is the finding that the set of core commands used in a Unix environment is quite small — less than 10 percent of the 400 available commands account for almost 90 of usage. This corroborates local observations of command usage and is a central argument for the preloading of "popular" executables into cache by the intelligent file server algorithm.
The current study was performed in an environment which used 4.2BSD Unix. Work has continued on this flavor of Unix. McKusick, Karels and Leffler [McKusick 85] report on the characteristics of the next scheduled release (4.3BSD). 4.3BSD offers few new features but rather concentrates on performance enhancement. Performance bottlenecks were identified throughout the system requiring extensive rewrites. Overall, this article shows system tuning at its very best. Of interest to the current study is the finding that one quarter of 4.2 BSD system time was spent translating path names to inodes and that "one major source of name translations came from requests to execute system utilities or access system databases." The solution adopted in 4.3BSD is to provide a name cache within the kernel. This name cache reduced name translation overhead from 25% to less than 10%. This same problem (viewed from the perspective of disk caching) is addressed by the considerable attention paid to caching of directory information by the intelligent file server algorithm.

Zhou, Da Costa and Smith [Zhou 85] describe a file system trace package for Berkeley Unix. This package purports to: trace all relevant file operations and collect all (!) useful information, allow tracing of a single user, group of users, a single process or group of processes, trace long or short term behavior, have minimum performance impact, and provide convenient analysis. Their system entails kernel modifications and extensive post processing facilities. *Logical* rather than physical I/O is traced -- physical I/O being considered too difficult. Directory and paging I/O are not traced. Trace overhead is given as less that 10%. Their system is comprehensive if a bit limited in the kinds of information it is willing to deal with and very efficient. During the course of their study they discovered many things about the Unix file system also observed in the current study: path names of files are difficult to get during the trace and must be obtained in post processing, Unix file mechanisms make the simulation of the file system in post processing nontrivial, complete file names often indicate type and function of files, the number of active processes in a system does not fluctuate excessively, and I/O transfer patterns in Unix are quite bursty. These are all helpful confirmations of locally observed system behaviors and support many design decisions made in the development of the IFS.
A follow-on study by Ousterhout et al. [Ousterhout 85] uses the above trace mechanism to analyze file access characteristics and drive a simulation which investigates Unix cache performance. Their analysis of access patterns indicated that most files are characterized by short lifetimes, are sequentially accessed (90%) and frequently read or written in their entirety (66%). Most files are short but long sequential runs account for most data transfers. The simulation of disk block caching assumes a dedicated file server with most of its memory devoted to disk caching. Cache sizes as large as 64 megabytes are considered. All their simulation results are questionable in that their trace data captured neither the the large number of directory disk accesses (both inodes and directory blocks) nor the smaller but still very significant paging activity. The results do however suggest that large caches and large block sizes can yield impressive performance. This study, though very well intentioned, demonstrates clearly the limitations that inadequate trace data can impose on subsequent trace driven simulations.

The trace method which was used in the current study was adapted from some excellent work by Simonetti [Simonetti 85] and is fully described in the Methods chapter (see page 46).

2.5 SUN NFS

The Sun Network File System (NFS) is a full featured file service. Since it provides efficient file interchange across non-homogeneous hardware environments through public domain protocols, it is expected to become a standard of file interchange. All data for the current study were collected in a Sun network environment (1.2) that predated NFS. At the time of writing, NFS has replaced that environment. NFS would seem a logical place in which to extend the results of the current study.

Lyon et al. [Lyon 85] provide the most complete description of the SUN NFS. This is an overview that gives a good overall feel for the system without any great detail about operating specifics. File system operations are divided into client and server functions. A
given system can be either server, client or both. NFS is not a distributed operating system but rather a means of unifying very different file systems over a local area network. The NFS interface has been placed in the public domain to encourage its general use. Basic file operations are provided on stateless servers. Authentication and file directories are supported while locking, data replication and synchronization are not. The SUN interface to NFS is extremely well integrated into the Unix environment by the device of vnodes (a virtual inode).

West [West 85] presents a statement of intent for the SUN NFS. It is billed as a business overview, yet it is really an argument for NFS as a vendor independent distributed file system standard. While NFS has grown out of a BSD Unix environment, it is claimed to be transmission, protocol and machine/OS independent. The advantages of a generic network file system are discussed in terms of sharing, security, consistency, administration, storage savings, lower incremental costs and storage migration. Terms for distribution of NFS protocols and licensing of NFS sources are given.

The Remote Procedure Call Protocol Specification [SUN 85a] details the specifications for the NFS message protocol which implements the remote procedure call. The remote procedure call is a logical extension of local procedure calls. Terminology is defined, the model explained, independence from transport protocols, semantics and binding are established and requirements delineated.

The Remote Procedure Call Reference Manual [SUN 85b] is an explanation of the organization of the NFS remote procedure call (RPC). A three layer protocol is established. The links to eXternal Data Representations (XDR) are discussed at length. Examples of RPC usage are also given. RPC is used by the NFS to communicate file system requests between client and server processes.

The External Data Representation Reference Manual [SUN 85c] presents the XDR scheme used by NFS in providing data portability. This portability is necessary for implementing remote procedure calls across systems which may have fundamentally different
internal data representations. The manual consists of "a description of the XDR [C language] library routines, a guide to accessing currently available XDR streams, information on defining new streams and data types, and a formal definition of the XDR standard. XDR was designed to work across different languages, operating systems and machine architectures."

Sandberg et. al. [Sandberg 85] go into further detail regarding the design goals of the Sun NFS and elaborates on some of the system tuning that was involved in its development. Several features of NFS are most clearly stated in this paper — in particular the NFS is based upon a stateless protocol (a design decision this study follows). The history of NFS is interesting in its own right, but even more interesting is that to work around performance limitations, numerous caches and readahead capabilities were added. They concluded that they had met their design goals and that further performance increments were limited by CPU speed bottlenecks.

Chang [Chang 85] presents a brief proposal for the extension of NFS into a full distributed networking environment (SunNet). Proposed extensions include a distributed name service, a distributed time service, a distributed file service and a distributed mail service. These are layered on top of NFS and *two new primitives: the status monitor and the distributed synchronizer. These are two powerful primitives for simplifying distributed systems design ...* The proposal while not terribly specific is plausible and demonstrates the ease with which full featured network environments may be built upon stateless file servers.

2.6 Caching Studies

In that this study concerns itself with improving remote file server caching performance, several articles which deal exclusively with caching are of interest. Most of these articles refer to more traditional (i.e., "mainframe") environments but are applicable to distributed file servers as well.

Kaplan and Winder [Kaplan 73] provide an excellent summary of the theory of cache
based approaches to performance enhancement. Design tradeoffs are well presented as are performance measures.

Busch and Kondoff [Busch 85] detail the involved search for improved disk performance in a mainframe environment. After exhausting economical hardware solutions, caching strategies are developed. The authors make some small use of known file types and anticipated access patterns in their caching algorithms but stop short of full exploitation. The results of this investigation are suggestive for network server applications but not directly transferable. The authors briefly discuss the need to extend their results into "decentralized systems."

While not concerning himself exclusively with caching, Stonebreaker [Stonebreaker 81] makes some rather salient points about caching and its improvement in database applications. Database management systems frequently know in advance precisely the access patterns of the files they use. Even when highly inappropriate, the underlying operating system frequently insists upon canonical LRU caching of file accesses and, in some cases, prefetching of file blocks. Stonebreaker points out that modifying caching strategies to cache cyclically rereferenced sequential files with an MRU discipline and not caching singly referenced files at all could reduce cache miss ratios by 10-15%. In addition, he suggests that "except in rare cases INGRESS at (or very shortly after) the beginning of its examination of a block knows exactly which block it will access next. Unfortunately, this block is not necessarily the next block in logical file order." Improvement of cache performance through file specific alternate caching strategies and intelligent prefetching are two major strategies adopted in this study for use in a general computing (as opposed to a DBMS) environment.
2.7 Research on File Types

The notion of file types is not novel to this study. This section presents several diverse works which have independently discovered the usefulness of typing files.

Smith [Smith 81a] presents an analysis of file reference patterns. This analysis precedes his development of algorithms for file migration (presented in another paper). While his example is hindered by the machine under consideration (an IBM 370/168 and 360/91 installation, files considered were all Wylbur data sets), his use of exploratory statistical methods is excellent. The goal of the study was to "describe and characterize ... the file behavior patterns observed" and to "develop a basis for the specification of file migration algorithms." It is concluded that time since last reference, file size, file class and file age are useful in predicting the time to next reference. File size and time since last use are, not surprisingly, most useful. Previous file interreference intervals were not found to be useful. The data analyzed were also used as the trace for trace driven simulation of the migration algorithms developed in subsequent work. His analysis provides a clear base for his later theoretical work, as well as a clear model of proper investigatory methods upon which to base such later works.

The companion article [Smith 81b] details Smith’s investigations into replacement algorithms. Working from the carefully gathered data of the above study, Smith tests a number of migration algorithms through simulation techniques. His simulations are all bounded by simulation results from "optimal but unrealizable algorithms" — a useful technique used in the current study. The study concludes that algorithms which use the most information about file access behavior work best and leaves open for future investigation adaptive or intelligent algorithms.

Satyanarayanan [Satyanarayanan 81] is also concerned with the behavior of file systems. In this instance, the analysis is based on a more reasonable machine (a PDP-10 at CMU). The analysis is to be used to drive a "simulation or analytic model of a file system for
a local network* and as "the basis of implementation decisions and parameters." The study draws conclusions from a series of snapshots of the file system. Analysis is concerned with interactions of file sizes, *file types* and functional file lifetimes. Related studies are well referenced and an analytic approximation is developed. Useful conclusions for similar academic file systems are advanced.

Mogul [Mogul 84] develops a generalized paradigm for representing information about files. He stresses "a need for a flexible, uniform representation of information about files. This need arises in all systems, but is acute in heterogeneous distributed systems." One of the critical pieces of information currently absent from current systems is that of file *type*. Mogul considers such information to be self evidently important and its absence a major driving force behind his new paradigm for representing directory information.

### 2.8 Queueing Model Studies

Queueing models are of direct interest to this study. Several focus directly on remote file service. Others support the specific methodologies adopted.

Brice and Browne [Brice 78] present a method of integrating scheduling, called feedback scheduling, in which CPU and I/O scheduling are jointly modified by a processes' data-flow rate. It is argued that such scheduling algorithms bring to bear greater knowledge of process behavior and results in greater CPU efficiency due to greater overlap of CPU and I/O process intervals. An interesting distinction is also made between the preemptive portions of I/O (emptying and filling of user buffers) and nonpreemptive portions (actual device activity) — this permits novel round robin scheduling (and associated flexible quanta allocation) of the preemptive phase of I/O activity. The proposition that feedback scheduling is superior to static scheduling is tested in both analytic models and trace driven simulations. Trace data from the UT2D Cyber OS taken with three job mixes is used to validate the analytic models and to drive the simulations. In all instances, feedback scheduling is at least as good if not superior to static scheduling — sometimes by considerable margins. The results
while directly confined to Cyber style machines should generalize well to other mainframe OSs, given the availability of I/O processors. Of particular interest to this study is Brice's conclusion that I/O processing should be handled by flexible and schedulable processors rather than unintelligent channels. More intelligence in I/O processors yields significant performance gains.

Oldehoeft and Allan [Oldehoeft 85] provide an excellent paradigm for testing the efficiency of storage allocation algorithms. Trace data, distribution driven data and random data are used to drive simulations of different allocation algorithms. Numerous methodological similarities exist between this study and the current one.

Another study which is interested in the queueing model comparison of algorithms is that of Teorey and Pinkerton [Teorey 72]. Five scheduling policies for movable head disks are compared by seek time and waiting time. Both analytical and simulation results are obtained. This study is again of interest in that it guided in the construction of the current studies' methodology for the comparison of caching algorithms.

Ramakrishnan and Emer [Ramakrishnan 86] are involved in the design of a distributed system. In this paper they focus on the file server as the "most heavily used resource" of interest as "its performance is critical to the success of the system." They design a queueing network model and calibrate it with prototype data. The design of their model closely resembles that of the model developed for the current study. The parameters they arrive at also bear a close relationship to those of the current study. They do not opt for a trace driven simulation but rather make simplifications to the load to permit analytic solution of their model.
2.9 Miscellaneous

Various papers do not categorize well but nonetheless have bearing on this study. They appear in this section.

LOCUS is a distributed system rather than just a distributed file system. As one expands the ideas of intelligent algorithms past remote file service into other realms of operating systems, a LOCUS like environment would seem desirable. All the motivations which push one into working in distributed file server domains apply equally to general operating system domains.

The earliest of the LOCUS papers [Popek 81] describes the general functioning of the then not fully implemented system and reports on the performance of the almost functional file system. LOCUS is presented as a fully integrated distributed system which emphasizes transparency and efficiency of network operations. Network transparency is claimed to allow easier software development (application programs don't even realize that they're dealing with a non-local resource), incremental development of network hardware and software, greater reliability (binding of network resources can be flexible and hidden from the network users) and simpler user models of the abstract machine (the user sees resources without machine boundaries superimposed upon them). The unique and difficult accomplishments listed include a solution to the global naming problem, efficient network synchronization, automatic (partial) partition merging, file modification commit mechanisms, replicated file consistency, and high efficiency network behavior (obtained through problem oriented protocols, integrated functionality, lightweight process network request servers and local mode optimizations). Rudimentary performance measurements of I/O service indicate that local LOCUS operation is almost as efficient as non-networked Unix (Version 7) and distributed LOCUS suffers at worst a twofold performance decrement (a happy result). The authors conclude that such performance is impressive when one considers the functionality gained in networking and the much higher costs (usually an order of magnitude) normally associated with layered networking designs.
LOCUS seems a better way. The caveat is offered that the design is incomplete, that the performance measures were limited, and that the LOCUS system requires much larger than standard memories. Yet to be completed at the time of publication were remote process creation, full automatic recovery, and performance tuning.

The LOCUS paper from the same group a year later [Walker 83] speaks of a much more mature system. The system has moved from PDP 11s to VAXen and 6800 based work stations, and all components of the system discussed in the earlier article have been implemented. Close attention is still paid to the file system (*file system activity typically predominates in most operating systems and so high performance is critical*). The file replication systems proposed in the initial paper seem to work smoothly in actual practice as do the atomic file modification commits (a vast improvement over the standard Unix file system). Remote process creation is handled as an extension of local process creation and is completely transparent. Several optimizations have been introduced (most noticeable amongst them the run call which is similar to a fork followed by an exec) to enhance process performance in a distributed environment but which are also useful in purely local contexts. Much time and effort has been spent in making recovery from file system partitions and merges automatic and transparent to the user. This effort appears to have been impressively successful. An important concept developed in the recovery work is that of file types which allow the application of intelligent algorithms to known types of files. The article concludes much more strongly than the previous article; this is based upon a much more complete system and considerably greater experience with its operation. Measurements of the LOCUS system indicate performance equal to Unix in the local case and *that remote access in general, while somewhat slower than when resources are local, is close enough that no one typically thinks much about resource location because of performance reasons.* Network overhead is great enough to make swapping load images across the network uneconomical. Most user problems resulted from lack of complete transparency. Remote execution was primarily of use in load balancing. Storage replication, for all the bother of recovery in partition merges, was deemed necessary. The cost for the networking facility is put at roughly a third more code than Unix and considerably increased complexity. Further research is to be
focused on performance considerations as the system is scaled into larger networks. In that this paper speaks of real results on a running system, one cannot be other than impressed. The author's initial optimism would appear warranted.

A third LOCUS paper [Mueller 83] describes an existing nested transaction mechanism. This paper is contemporary with the second LOCUS paper just reviewed. The mechanism is an extension of Unix modular concepts into the domain of transactions. The mechanism is embedded in the OS to preserve network transparency and represents "the first actual implementation of nested transactions on a distributed system." I am not sufficiently versed in the database literature to judge the adequacy of the solutions presented; certainly they appear adequate to the task. What is obvious is the ease with which nested transactions were merged into the LOCUS system - with the resultant implication that LOCUS is, as claimed, a robust general purpose network environment which can be turned, with ease and elegance, to many applications not provided for in the original design.

Clark [Clark 85] introduces a novel program structure, the "Upcall." In discussing the performance problems of systems with layered levels of abstraction, he proposes that control flow should not always be downward through the layers or to directly adjacent layers. He offers as an alternative that "procedure flow should map on to the natural flow of control in the program, whether up, down, or sideways." While not dealing with issues of data reliability, this argument is similar to the information passing notions of "bridging levels of abstraction" (or hints) discussed in the Introduction.

Cooper [Cooper 83] draws conclusions similar to those of this study as regards the interaction of modularity and efficiency. In the specific domain of layered communication protocols, he argues the need for modularity (*layering* in protocol terminology) and at the same time attributes to layering considerable inefficiency. Modularity is required for correctness. The inefficiency stems from lack of knowledge about the use each layer is putting others to -- the *inability of the lower layers in the protocol hierarchy to understand what the higher layers are doing.* Cooper deals with the inefficiency by proposing that module
boundaries be made more permeable in a well stipulated fashion. He refers to this as "soft layering" while I call it knowledge bridging. He uses soft layering to communicate "a model of the way in which the [low level] protocol expects higher level protocols to use it." While this study uses knowledge bridging to transfer knowledge from high level to low level modules, Cooper's soft layering transfers knowledge in the opposite direction.

In a general perspective on distributed systems, Kleinrock [Kleinrock 85] notes that workstation processing power is an order of magnitude greater than that of installed mainframes. Yet, he adds, most of the time that processing potential is idle. He concludes that one should search for applications which tap this unused power. This study is one such application.

The current study claims that program behavior is characterizable, in particular that file access behavior of a program can be generalized. While this is generally accepted, it has not been extensively researched. Other work which has focused on characterizing program behavior has concentrated on either virtual memory usage (of which Alanko's work on program behavior measurement [Alanko 80] is a prime example) or working memory use by programs (see for example [Korn 85]). The consensus which emerges from such literatures is that programs do have reliable behaviors which can be captured by observational methods. While most programs are to some extent data dependent in their behavior, most have some consistent patterns and few have no predictability at all. This bodes well for predicting file access patterns of programs.
Chapter 3

Design

This chapter discusses the design issues of the study. The following chapter deals with the methodologies used to implement the design. Put slightly differently, this chapter addresses what the study was attempting to do; the following chapter details how it was done.

3.1 General Approach

In order to assess the impact of intelligent algorithms on remote disk server cache performance, several alternative research strategies were considered.

- An actual implementation of a remote server using intelligent algorithms was a very attractive possibility. Few demonstrations are as convincing as a fully functional prototype. Careful consideration of the scope of such a project and the resources available for its completion (the author could anticipate no assistance from others working on similar projects and sharing common development tasks) lead to its regretful abandonment.

- A stochastic simulation approach was considered. It is believed that the environment from which it was desirable to draw conclusions (a network disk server connected to a multi-host UNIX environment through a local area network) was not sufficiently characterized to permit reasonable stochastic models to be developed.

- A trace driven simulation was finally settled upon. Regardless of the experimental method chosen, it became clear that it would be necessary to study the actual physical environment of interest simply because it is so poorly understood. That study presented ample opportunity to collect trace data with which to drive a simulation. Hence the simulation responds to the historical demands of a real, well understood system.
3.2 Strategy

To test the hypothesis that intelligent algorithms using knowledge about file access patterns can improve remote disk server performance, several tasks had to be accomplished:

- Information about user activity and disk block activity had to be obtained and correlated.

- An initial analysis of these activities had to be performed to see if variant caching strategies could be effective and if intelligent algorithms could characterize access patterns and file types.

- Following a successful analysis, those characterization algorithms had to be developed.

- Using the characterization algorithms, the records of disk block activity had to be reduced and annotated to provide traces with which to drive the simulations.

- Three simulations had to be developed and run over a variety of trace data. A simulation using intelligent algorithms had to be developed and its performance contrasted with that of simulations using conventional algorithms (LRU) and those able to exploit full knowledge of future service demands (Optimal). The behavior of these three simulations with different cache sizes had to be explored.

- The simulations required some form of validation.

- The results of the simulations had to be compared and their differences explained.

- The behavior of the intelligent algorithm had to be be investigated in some factor analytic fashion.

3.3 Data Collection

In a Unix environment, all interaction with users is handled by one of a group of programs collectively known as shells. The choice of a specific shell is a matter of personal preference, but all users have one and it is through that shell that they communicate their intentions to the operating system (see figure 3-1). The shell accepts user commands and translates them into specific requests to the operating system.
Figure 3-1: Unix Interface Organization

The operating system of Unix is a monolithic collection of code modules collectively known as the kernel. A well defined set of system calls allow programs access to the functions of the kernel. It is in the lowest levels of the kernel that disk activity is performed. It is within the kernel that the logging of disk activity must be done.

While user intent is clearest within the shell, conjoining the monitoring of activities of arbitrary shells with kernel activities proved technically infeasible. User intents can, however, be fairly well determined (with a bit of effort) from the system calls to the kernel. System calls can be logged from within the kernel.

A mechanism was developed by which an arbitrary Unix process (and any child
processes, if desired) could be tagged. All system calls which affected the disk system by a
tagged process were logged to a file, along with the actual actual disk activity engendered by
that call.

3.4 Initial Analysis

Data for the the initial analysis were quite informally collected, largely as a
byproduct of exhaustively testing the trace collection tools. Little attempt was made to be
systematic; the initial analysis was simply intended to indicate whether a more rigorous study
was warranted.

Careful examination of trace data indicated several interesting features of file access
in typical Unix processes:

- Directory information (inode, indirect block and directory data block) accounts for
  a sizable portion of file I/O. The directory information accessed is not scattered
  randomly throughout the file system but clusters around commonly used system
  paths and user working directories.

- User files (as distinguished from temporary files created for the user) are almost
  always accessed sequentially. Knowledge of the processor opening the file is a very
  good predictor of the number of sequential passes through the file.

- Temporary files are typically small and frequently randomly accessed (and/or
  repositioned and rewritten). They can be reliably assumed to have much shorter
  lifetimes than user files.

- A few system executable files are responsible for most of the executable activity of
  the system. Many executable files page in a small well defined subset of their disk
  blocks before paging activity becomes data dependent.

- Disk blocks are quite frequently repeatedly accessed (see, for example tables A-2
  and A-3 in Appendix A). Good caching algorithms can make a considerable
  difference in performance.

These observations suggested that the study was worthy of further pursuit.
Additionally suggested were several guidelines for the design of the intelligent file server
algorithms:

- There should be separate cache partitions for each client process and for the
system at large. In this fashion, operations on processes (i.e., process creation and deletion) can be mapped directly to operations on the file server cache. Allocation of cache to partitions should be dynamic; this is necessary due to variation in the number of processes which a file server is serving at any given moment.

- As directory references occasion such a major portion of disk accesses, a fixed percentage of each partition should be devoted to directory information - both directory blocks and inodes. LRU replacement should be used to maintain the directory cache. The global system partition should cache information of frequent global use. This global directory information can be preloaded and maintained during lull periods. Process partitions should cache information not contained in the global cache but frequently encountered by the process.

- To allow for appropriate cache management strategies for files, subpartitions within process partitions should be provided for management of related groups of blocks. Thus all blocks of a file used by a process (i.e., not necessarily all the blocks of a file) should be grouped into a subpartition. Operations by a process on a file can then be mapped directly into cache operations on a subpartition. Due to the variability of process file sizes and numbers, the subpartitioning should be dynamic.

- User files should whenever possible employ read ahead. Single pass files should not be cached. Multi-pass sequential files should employ an MRU caching discipline.

- Wherever possible, temporary files should be cached in their totality (often possible due to their small size). Where this is not possible, LRU disciplines are desirable.

- File information should be flushed from a process' cache when the process deletes the file and no open file descriptors for that file exist. All of a process' cache partition may be scavenged when the process terminates.

- A mechanism should be provided to allow transferring a file from one process' cache partition to that of another process. This allows a process to hand off a file to another related process before it terminates.

- Frequently used system executable files should have their initial page set preloaded into the system partition. Preloading of initial page sets for user executable files would seem counter indicated.

- Spare disk service and cache resources can be used to maintain the contents of the system (general use) partition. Common directory information and system executables can be restored in periods of server inactivity.
3.5 Trace Analysis Algorithms

The data which comes from the disk activity logging mechanism requires considerable post processing to obtain a clear record of disk activity. This stems in large part from the absence of static state captured in the trace records — entries in the trace record are relative to the constantly changing state of the file system and a processes' interface to that file system. It is necessary to simulate the Unix file system and use the simulation's state to interpret each file's activity. Once a clear record of actions on files is obtained, types of file accesses can be characterized.

Therefore, generalizations about access patterns can be produced from the trace records to demonstrate that such generalization is possible.

Once each trace session is processed, they must be combined into larger, composite trace files. A number of passes through the data are required to laminate these smaller sessions together. Some of the passes are required by the mechanics of the trace driver for the simulation package. Others were needed to provide information about the whole trace needed by the optimal simulation.

3.6 Simulated Process Access Sequencing

The composite trace contains legitimate sequences of Unix process disk block accesses. The composite trace does permit what might be considered anomalous process behavior. That is to say, arrival times for transactions from a given process in a session trace are based upon the response characteristics of an unloaded disk system. When terminal sessions are combined in the composite trace, the response characteristics of the file server simulation are degraded — yet process requests continue to arrive at the original rate. When the file server becomes loaded, a process can have several disk service requests outstanding. In actual practice, processes wait for service requests to be satisfied before submitting new requests. Thus the synthetic nature of the composite trace causes it to be "illegal."
The solution to the issue of legality involves modification of the composite traces. Transactions from a process are not admitted into the simulated file server until any previous transaction of that process has exited from the simulation. Thus, whenever a transaction would be illegal, it’s arrival at the simulated file server is delayed until it will not engender an illegal condition.

This sequencing of process accesses creates new problems. Since legality is a dynamic quality related to the performance of a simulation (i.e., more responsive simulations will have to delay arriving transactions less), each simulation will have to perform varying degrees of process access sequencing. Thus the results cease to be directly comparable. To avoid this, it was decided to generate a new composite trace as a byproduct of legalization. This new simulated process access sequence composite trace is legal for the simulation which generated it. By progressively legalizing a composite trace through all simulations, it is possible to generate a trace which is legal for all simulations. All simulations can then be run on this one simulated process access sequence trace with directly comparable results.

3.7 Simulations

All the simulations share certain basic features:

- Two different types of information are being dealt with - inodes and disk blocks. They are maintained in separate caches.

- In keeping with the strategy employed by the system from which the trace data was collected, all requests (read and write) are synchronously handled. Write requests are performed with cache write through.

3.7.1 Intelligent Simulation

In dealing with the proposed structure for the intelligent file server (IFS), it is essential to have a clear understanding of the division of function in the file system. The work is based upon data gathered from Sun workstations; this strategy employs a remote disk block server. Thus the remote component of the IFS is also a block server. As such, the remote server deals with strategic abstractions at the level of disk blocks.
The intelligence of the IFS can be viewed as being in two parts (see figure 3-2). The first involves generating hints about which of several caching strategies is likely to be most applicable to the handling of a set of block requests from a given file — based upon a high level understanding of the nature of the process making the requests and the file being requested. The natural location for this decision making is on the host machine—near the files and processes involved. Other than demonstrating the feasibility of such decision making, this study does not concern itself with this element of the IFS. The second part concerns itself with satisfying the requests of all processes from all hosts the remote server is responsible to in some optimal fashion — based upon the hints and the current set of all requests. This clearly belongs in the server. The intelligent algorithms in the server are the central thrust of this study.
Figure 3-2: IFS Functional Decomposition
It could be argued that the entire IFS should reside in the remote server. Were the system under consideration a remote file server (as would have been the case if the trace data had been collected from the Sun NFS) such would surely be the case. The difficulty of transporting and maintaining the file abstraction to/on the remote block server is considerable and costly — one practically has to build a remote file server. Further, the high level information used to generate hints is not directly germane to block operations and is functionally quite removed from the kinds of information normally exchanged between host and disk block server. Duplicating the user abstraction in the remote server is expensive and unnecessary. The IFS, when based on remote block servers, naturally splits into client and server modules.

The initial data analysis suggested several design principles:

- The cache is partitioned into several distinct areas. A system partition exists to hold directory information (both block and inode) for common use paths and frequently used blocks of popular system programs. All other information is held in process partitions — one for each process on a remote host requesting service. Process partitions are dynamically managed.

- The partitioning is not inflexible. Partitions only come into play when there is no unused cache space — at that time, the partition most over its nominal size is selected to provide replacement cache elements. Thus partitions serve to balance cache use without requiring that elements be allocated inflexibly or, if unused, remain idle.

- Process cache partitions are created when the first transaction from that process is received by the server. Process cache partitions are scavenged after notification of process termination. Scavenged cache blocks are placed on a garbage list from which they can be reused (as still valid) by other processes.

- The system cache partition has a minimum size but is allowed to vary dynamically with the number of processes. Size of process partitions is directly determined by the number of processes requesting service.

- When a file is opened (an operation which does not generate any file activity on the specified file), the kernel of the client machine needs to negotiate a unique block group identifier with the server. This identifier is not the same as the file — it simply serves to group related blocks. All subsequent file operations specify this group id. File groups are contained in process cache subpartitions. Groups are transferable between process cache partitions.
- Service requests can be grouped into several categories – inodes, directory blocks, executable blocks, sequentially accessed file blocks (using an MRU replacement algorithm), randomly accessed file blocks (which use an LRU replacement algorithm) and file blocks which are accessed only once (which should not be cached).

- Directory information (both block and inode) can be determined to be either of global or process specific interest. In the case of process specific information, it is cached in a process partition. Otherwise, it is cached in the system partition. The system cache can be preloaded (and reloaded during server idle periods) with information known to be of general utility. Since all simulation periods are brief and fairly active, cache reloading is not actually simulated.

- Sequential access file and directory block requests (which are normally read in sequential fashion) specify with each request the next block in the sequence to be prefetched on a resources available basis.

- Commonly used executable blocks are kept in the system partition. These blocks can be preloaded and could be reloaded when the server is otherwise idle. As described above, reloading is not actually performed in the simulations.

3.7.2 LRU

LRU cache management is commonplace in almost all existing file server caches. Performance of this simulation is intended to reflect the expected behavior of contemporary systems within the simulation context.

The LRU simulation distinguishes between inodes and blocks. Separate caches are maintained for each. Caches are otherwise undistinguished. The most straightforward of LRU cache replacement algorithms is used in adding new elements to the cache – the cache element least recently accessed is chosen for replacement.

3.7.3 Optimal

Optimal algorithms are only of theoretical interest. They can only operate in synthetic situations in which full (and in real environments, unobtainable) information is available. This simulation provides a theoretical benchmark of the best results which can be obtained by cache management strategies alone.

The Optimal simulation directly follows the LRU simulation save that the cache
replacement algorithm is optimal ([Knuth 85]). The optimality is achieved by knowing the next reference time for each element in a cache. The element selected for replacement is that with the time of next reference furthest removed from the present.

3.8 Validation

Validation of the simulations is a critical component of any such study. Correctness of the simulation models is aided by careful design and is demonstrated by internal consistency checks and checkpointing through known states. The validity of the models is verified by comparison of their global performance measures with known systems. In addition, the specific behavior of the LRU simulation is verified against data recorded from an actual Unix system.

3.9 Simulation Runs

The simulation runs are organized as a 3 factor experiment in which algorithm (Intelligent, LRU or Optimal), cache size and data set are the 3 orthogonal independent variables. Response time and variance of the system being simulated are the dependent variables.

3.10 Factor Analysis

The simulation runs are organized as a 3 factor experiment in which contributing factors to the intelligent algorithm, cache size and data set are the 3 orthogonal independent variables. Response time and variance of the system being simulated are the dependent variables.
Chapter 4

Methods

4.1 Overview

This chapter describes in detail the activities performed in order to accomplish the design goals of the previous chapter.

The first part of the chapter describes the modifications to the Unix system kernel necessary to the gathering of data and the subsequent data gathering process. The steps necessary to transform the trace data into a form usable for driving simulations are detailed. The algorithm for generalizing file access patterns from trace data is specified. The simulations are described in detail. Model validation is discussed. The chapter concludes with a specification of the simulation runs performed.

The results of the simulations are presented in the following chapter.

4.2 Kernel Instrumentation

As has been previously discussed, it was necessary to modify the Unix operating system to collect data about user activity and the disk activity it engenders.

The modifications to the Unix kernel were built upon a system call trace facility developed by Simonetti at SUNY - Stony Brook ([Simonetti 85]). He describes this package as:

A solution to the problem of debugging user processes which interact closely with the kernel (i.e. make extensive use of system calls) is to create a utility to record the system calls made by any process, along with their arguments and results. The routines which
perform the recording function are implemented entirely within the kernel itself thus eliminating the need to alter the user code or its runtime environment. Access to the tracing mechanism is provided through the addition of a new system call. A user interface program is provided which makes use of the new system call in controlling the trace. One performs a system call trace by executing the user interface program to open a log file to which the trace will be written, carry out the desired tracing operations and close the log file on completion of the trace. The log file itself is written directly in a human readable format which may be viewed without the use of a post processor . . .

The means available for controlling the tracing process are critical in that the trace must not be allowed to get out of hand resulting in a flood of information, while at the same time providing a means to monitor a process's children. The available commands allow the user to monitor a process only, a process and all of its children, or its children only.

The system call trace facility implementation is fairly straightforward. Simonetti continues:

The realization of the trace facility thus consisted of the development of routines for implementing the trace itself in addition to minor modifications of the kernel routines for process creation and the handling of traps.

The bulk of the code added to the kernel consisted of the routines systrace, entrytrace and exittrace. The systrace routine implements the system call used for opening the trace file, enabling or disabling the trace for any given process and closing the trace file. The entrytrace routine is invoked on entry to the kernel just before a branch is made to the kernel code implementing the specified system call. The parameters to entrytrace are the system call number and the number of arguments. Entrytrace writes out a record to the trace file consisting of the name of the system call and its arguments. The exittrace routine is invoked before the kernel returns control to the user on completion of a system call and writes a record to the trace file consisting of the name of the system call and any results or error status returned. The system call number is passed as a parameter to exittrace . . .

Two one bit flags were added to the kernel data structure for each process. The SYSTRACE flag is set for any process whose system calls are to be traced. The CHILDTTRACE flag is set to indicate that any future children of the process are to have their SYSTRACE and CHILDTTRACE flags set. The process creation routine was modified so that if the CHILDTTRACE flag is set for the parent process, the SYSTRACE and CHILDTTRACE flags will be set for the child process.

The portion of the trap handler which uses the system call number to branch to an appropriate kernel routine was modified so that if the SYSTRACE flag was set in the
calling process's structure, then calls would be made to the *entrytrace* and *exittrace* routines before and after the call to the requisite kernel routine, respectively.

Both the original trace package and all specialized modifications made to for this study were written in the C programming language.

For the purposes of this study, there are several things which Simonetti's package does too well.

- It traces *all* system calls. For purposes of monitoring disk activity, only system calls which produce disk activity or change the state of the Unix file system need be traced. The package was modified so that only the following system calls were traced: access, chdir, chmod, close, creat, dup, dup2, execv, execve, exit, fchmod, flock, fork, fstat, fsync, ftruncate, link, lseek, lstat, mkdir, open, read, readlink, readv, rename, rmdir, stat, symlink, truncate, unlink, utimes, vfork, write and writev.

- System calls traced with the package reported *all* of their arguments and results — including all data transferred in file operations. While this is desirable for the debugging purposes proposed by Simonetti, it generates too much needless information and went too far in compromising the contents of user interactions in the context of this study. All such excess reportage was stripped from the trace package.

The trace package also had a number of deficits which had to be remedied before it could become useful in the current study.

- Simonetti was not in the least concerned about low level input/output activity. Trace code was added to the routines which perform the actual i/o operations on inodes, disk blocks and page blocks (executable files are comprised of "normal" blocks but are read into and out of memory by paging routines rather than by the usual disk block i/o routines). This trace code checked the SYSTRACE bit in any process for which i/o was requested and, if set, reported that i/o activity into the trace file. All requests, both synchronous and asynchronous, were traced. Local and remote disk accesses were both captured.

- The original trace program was willing (optionally) to timestamp all system calls with a granularity of one second. The modified package made timestamping the default and timestamped all transactions with the granularity of the system clock (10 milliseconds). The greater accuracy was required for the resultant traces to realistically drive the simulations.
The revised trace package was installed on Sun workstation running 4.2 bsd Unix (Sun version 1.2). This trace package (when operating) typically caused an overall slowdown of roughly 50% (directly related to a roughly 150% increase in system overhead) in disk intensive activities. While not negligible, this overhead was deemed acceptable.

In order to log a session, a volunteer would open a trace file and enable tracing on the child processes of his shell process. Thus all disk activity spawned by the volunteer's shell program would be captured. At the end of a period of activity, the volunteer would close the trace file and report its existence to the experimenter.

A brief segment of a trace produced by the package is given in figure 4-1 by way of example. System call level information appears in boldface type and disk block level information appears in normal typeface.
Figure 4.1:  Trace Log Example

4.3 Trace Gathering

Volunteers were solicited from individuals who had access to the network of machines which included the instrumented SUN workstation. Two faculty members, one secretary and two graduate students (including the experimenter) contributed session logs.

The volunteers were instructed in the mechanics of turning on and off session tracing. They were then asked to perform their normal work while tracing for brief periods (five to thirty minutes). Volunteers were not restricted in their choice of activities. Volunteers were
fully informed of the information content of the traces and allowed to examine the plain text results of session logging. Volunteers were encouraged not to submit any sessions which compromised sensitive information (to the best of the experimenter's knowledge, this situation never arose).

The session logs are characterized in Appendix A.

An unfortunate experimental artifact arises in the gathering of data for this study. The trace gathering system was developed under the Sun 2.0 version of Unix. Midway through this study, our system support staff elected to upgrade our Sun network to version 3.0 of the operating system. This new version is incompatible with the tracing package. Thus there came a certain point at which it was necessary to declare by fiat that a sufficiency of data had been collected and the rest of the study would make do with the data on hand. The system upgrade was delayed until there was agreement that this point had been reached. The system upgrade was then performed and further data gathering permanently suspended. The inability to augment available data was to have distinct methodological consequences.

4.4 Trace Generator System

4.4.1 Trace Analyzer

The traces produced represent a complete record of process file activity. But this information is not directly usable — some post processing is necessary to gather information distributed throughout the trace and provide the required coherence at any given point.

In the Unix environment, most system calls which engender file activity use file descriptors rather than file names. A file descriptor is a small integer which is used by a process to efficiently represent a file after it has been opened or created by that process. Frequently, file descriptors are reused by a process after the file they represent has been closed. In addition, file descriptors may be copied, the files they represent may be renamed, the files they represent may be deleted (leaving a file descriptor representing an unnamed file
- a valid Unix construct) or any of a host of other operations may be performed which confuse the mappings of file name to file descriptor. In short, there does not exist a static mapping of file descriptors (as found in trace records) to file names (the entities of interest). Much of the post processing in the trace analyzer involves simulating the interaction between user processes and the Unix kernel to dynamically obtain this desired mapping. This simulation is not trivial.

Post processing must also distinguish between commingled traces from concurrently executing (and hence trace logging) processes. This can easily be the case in a Unix environment where either two users are generating trace logs simultaneously (a condition which did not pertain in this study but was allowed for in the interests of generality) or a shell command spawns several concurrent processes (not a common occurrence, but by no means unheard of).

Frequently, a single user command will occasion the execution of several processes (usually in sequential fashion but occasionally concurrently). For example, invoking the C compiler typically yields sequential processes executing the C preprocessor, the C compiler, the assembler and the loader. Each of these processes communicate performs some work upon files and subsequent processes made use of some of those files. The trace analyzer must have some understanding of the structure of process execution and produce an analysis of file access patterns across related processes. In this experiment, it was necessary to chop up session logs by hand in that the trace analyzer had no knowledge of which processes were functionally related (e.g., that the C compiler processes described above were unrelated to a subsequent edit process in the trace).

As can be seen in figure 4-1, session logs contain two kinds of trace information - system call information (bold typeface) and block level information (normal typeface). The analyzer separates these two kinds of information and deals differently with each. The system call information is used to update the process' file state. The block level information (needed to drive the simulations) is annotated by the state information.
The first pass of the analyzer maintains state for each process it encounters. Processes freshly created are assumed to have the default Unix file descriptors. Processes forked from other processes correctly inherit their parent's file descriptors. As system call information is read from the session trace, the relevant process' file state is updated. The nature of the update is based upon a production rule system which mimics the behavior (in all its strangeness) of the Unix file system. The major goal of this file state is to allow a mapping of file descriptor to file name. Additionally, a record is maintained for each process of its actions on all files that it references. As block level information is encountered in the session trace, it is tracked back to the most recent system call of the process responsible for the activity and the block reference is annotated with file name information.

At the conclusion of the first pass, another production system is invoked on all files in all processes observed in the session trace. This production system determines access patterns from lists of accesses and positionings of the file in question. For example, a file which is opened, positioned to end of file, positioned to beginning of file, read sequentially for several blocks, positioned to the current file position, sequentially read for several blocks, positioned to beginning of file, sequentially read for several blocks and closed is determined to be a sequentially read file. Files are initially determined to be one of: single pass sequential read, multi pass sequential read, random access, and write only. Another production system merges access patterns for a file across all processes in which that file is referenced. Single pass sequential, opened but unaccessed and write only files are mapped to "don't" cache strategies. Multi pass sequentially read files map to MRU cache strategies. Randomly read files map to LRU cache strategies.

At this point, a summary of file activity (see figure 4-2), an annotated log of system calls (see figure 4-3) and a partial trace of block activity are available; process numbers, device/block numbers, whether block activity is related to file or directory operations (from context), whether directory activity is system or user (derived from a simple heuristic -- system path names start with a "+/", user pathnames don't) and file names are known and this information appears in the block activity trace. Further information to be included (see figure 4-4) is added in successive passes through the file.
Processes in trace -

733 /usr/ucl/vi 00:00:04.24 00:19:42.96

Composite File Transactions -

/tmp ((733 (NOP))) NO-CACHE
/tmp/Ex00733 ((733 (RANDOM-W 50176 82580))) NO-CACHE
/camd ((733 (RANDOM-R 1024 39220))) LRU
/tmp/Rx00733 ((733 (RANDOM-RW 10240 23608))) LRU

Figure 4-2: Pass 1 Summary Example

"733 Call execve ( /usr/ucl/vi, 0x23100, 0xfff8c )"
"733 Call dup ( 0, 0 ) StdIn"
"733 dup returns 3"
"733 Call close ( 0 ) StdIn"
"733 close returns 0"
"733 Call open ( /usr/prof/molloy/.exrc, 0, 0 )"
"733 open returns Error 2"
"733 Call open ( .exrc, 0, 0 )"
"733 open returns Error 2"
"733 Call close ( -1 )"
"733 close returns Error 9"
"733 Call stat ( /tmp, 0xffffcfc )"
"733 stat returns 0"
"733 Call creat ( /tmp/Ex00733, 384/ 0x180 )"
"733 creat returns 3 3 <- /tmp/Ex00733"
"733 Call close ( 3 ) /tmp/Ex00733"
"733 close returns 0"
"733 Call open ( /tmp/Ex00733, 2, 0 )"
"733 open returns 3 3 <- /tmp/Ex00733"
"733 Call write ( 1, , 7 ) StdOut"
"733 write returns 7"
"733 Call write ( 1, , 7 ) StdOut"
"733 write returns 7"
"733 Call unlink ( /tmp/Ex00733 )"
"733 unlink returns 0"
"733 Call close ( 3 ) /tmp/Ex00733"
"733 close returns 0"

Annotations appear in boldface

Figure 4-3: Annotated Session Log Example
Session Trace $\Rightarrow$ Final Trace Log

\[
\begin{align*}
time \ p n \ read-op \ (m,n) & \Rightarrow r \ b \ p n \ m+n \ nb \ file \ cd \ na \\
time \ p n \ write-op \ (m,n) & \Rightarrow w \ b \ p n \ m+n \ file \ cd \ na \\
time \ p n \ dir-read-op \ (m,n) & \Rightarrow r \ b \ m p n \ m+n \ nb \ 0 \ 4 \ na \\
time \ p n \ dir-write-op \ (m,n) & \Rightarrow w \ b \ m p n \ m+n \ 0 \ 4 \ na \\
time \ p n \ igr e t \ (m,n) & \Rightarrow r \ i \ m p n \ m+n \ na \\
time \ p n \ ip u t \ (m,n) & \Rightarrow w \ i \ m p n \ m+n \ na \\
time \ p n \ pagein \ (m,n,o) & \Rightarrow r \ b \ p n \ m+n \ 0 \ 0 \ 0 \ na \\
time \ p n \ pageout \ (m,n,o) & \Rightarrow w \ b \ p n \ m+n \ 0 \ 0 \ 0 \ na
\end{align*}
\]

Non Terminal Symbols

cd = cache discipline (see below)
dir-read-op = block read operation accessing directory information
dir-write-op = block write operation accessing directory information
file = file group
m = physical device number
mpn = 0 if from a root lookup, pn otherwise
n = inode or block number
na = time of next access (0 if no next access)
b = next block of a sequential file
o = page number of paged block
pn = process number
read-op = block read operation
write-op = block write operation

Cache Disciplines

\[
\begin{align*}
0 & \text{ swap file (implies file = 0)} \\
1 & \text{ don't cache} \\
2 & \text{ LRU} \\
3 & \text{ MRU} \\
4 & \text{ directory block (implies file = 0)}
\end{align*}
\]

Figure 4-4: Trace Log Formats

The code which performs this first pass is implemented in roughly 26 kilobytes of Interlisp-10 source which runs under the TOPS-20 environment on a DEC 2060.

A second pass through the data substitutes unique (within the context of this
experiment) numbers for the file names and determines (for sequential files and directory block sequences) the next block to be accessed. Cache disciplines (from the first pass summary) are verified by hand as accurate and included in the trace at this point. Process termination hints and transfer of files between processes hints were inserted by hand editing at this point. An example of the trace log at this point is given in figure 4-5.

```
rb 733 179834304 0 0 4 na
w 1 733 179806061 na
T 4.6
rb 733 134415064 0 0 0 na
T 4.7
rb 733 134415080 0 0 0 na
T 5.1
rb 0 179200136 0 0 4 na
ri 0 179200256 na
ri 0 179800002 na
rb 0 179800144 0 0 4 na
ri 0 179806349 na
rb 0 179835974 0 0 4 na
T 6.74
rb 733 179839720 179839720 886 2 na
T 6.84
rb 733 179834304 0 0 4 na
ri 1 733 179806924 na
w 1 733 179806924 na
T 6.88
rb 733 179839720 179839720 886 2 na
T 6.94
rb 733 179839720 179839720 886 2 na
T 7.02
rb 733 179839720 179839720 886 2 na
T 7.08
rb 733 179839720 179839784 886 2 na
T 7.14
rb 733 179839784 179839784 886 2 na
```

Time of next access (na) cannot be computed until after lamination

Figure 4-5: Trace Log after Second Pass - Example

The code which performs the second pass is implemented in roughly seven kilobytes of Interlisp-10 source which runs under TOPS-20.
At this point each session trace has been transformed into a roughly complete trace of block activity, ready to be laminated together with other similar traces. The operations performed on the original session trace to this point are graphically illustrated in figure 4-6. The lamination processed session traces and the small amount of additional processing required for a complete composite trace is discussed in the upcoming section on Trace Lamination.

---

Figure 4-6: Session Trace Processing
4.4.2 Trace Generalizer

The trace generalizer takes as input the list of file names and caching disciplines from the first pass of the trace analysis and generates rules for mapping file names into caching disciplines. These rules will not be exact but need to be generally correct. While these rules are not directly used in this study, the demonstration that such rules can be automatically generated is important when one argues that the overall approach to intelligent caching is possible and implementable.

Several more computationally involved procedures were implemented, but the following direct approach was found to give high quality results with least computational overhead.

The algorithm breaks file names up into three parts — directory, file name and file extension. Files are partitioned into groups by directory and file extension. The algorithm first attempts to find extensions which uniformly possess the same caching discipline. If any are found, rules are generated for these mappings and the files so mapped are removed from the list of files to be accounted for. The same procedure is repeated for directories.

After all uniform mappings have been exploited, if there remain files to be accounted for, one must develop mappings which are only generally correct. The algorithm goes through the remaining directories and extensions calculating the effects of performing mappings based upon that directory name or extension. The mapping accounting for the greatest number of files with the fewest errors is selected. The files thus accounted for are removed from the list of unmapped files and the entire procedure iterated. This iteration is necessary because the partitioning by directories and extensions is orthogonal. The iteration terminates when all files have been accounted for.

The algorithm produces a list of mappings from directories and file extensions to caching disciplines. The list is not minimal but it is maximally correct. Note that the order of the list is important — a file may be mapped by two rules (directory and extension), but the one which occurs first is most likely to produce a correct mapping.
An example is given of the operation of this algorithm on a list of files from a C compilation in figure 4-7. In this example, a list of files accessed by a C compilation - with their associated access pattern - are given to the generalizer. The generalizer outputs a list of rules which map file names to access patterns.

```
Input to Generalization Algorithm-

("/ccinit 1)
(mung.c 1)
(marf.c 1)
(/tmp/cc1 2)
(/tmp/cc2 2)
(mung.o 2)
(marf.o 2)
(neato.h 3)
(/usr/include/stdio.h 3)
(/usr/include/types.h 1)
(/usr/include/dir.h 1)
(/usr/include/dunno.h 1)
(/usr/local/lib/libc 2)
(/usr/local/lib/liba 2)
(a.out 1))

Output-

((EXTENSION out => 1)
(EXTENSION o => 2)
(EXTENSION c => 1)
(DIRECTORY /usr/local/lib/ => 2)
(DIRECTORY /tmp/ => 2)
(DIRECTORY "/" => 1)
(DIRECTORY /usr/include/ => 1)
(EXTENSION h => 3))
```

Figure 4-7: Generalization Algorithm Example

In actual practice, to form a set of mappings for a particular program, the program would be repeatedly invoked in such a fashion as to fully exercise its file behavior. Traces of these invocations would be analyzed and then fed collectively into the generalization algorithm.

The code which performs this algorithm is implemented in roughly eight kilobytes of Interlisp-10 source which runs under TOPS-20.
4.4.3 Trace Lamination

Once all session logs have been processed in the first two passes of the trace analyzer, it remains to laminate them together in order to simulate the effects of simultaneous activity by multiple network clients of the file server. Several more passes through the trace data are required for this lamination. These passes are illustrated in figure 4-8.

![Diagram](image)

**Figure 4-8: Trace Lamination Processing**

First, it is necessary to physically merge the separate terminal session logs. Each individual session log is given a time offset (all logs have timestamps which start at relative time zero) and all sessions are merged by timestamp plus offset (see figure 4-9). What results is a single file containing all individual terminal sessions spaced throughout the simulation.
period by their relative temporal offsets. This is performed by roughly four kilobytes of Interlisp code.

Figure 4-9: Merging of Terminal Sessions by Session Offset

The composite trace file contains all references to blocks for the course of the simulation. At this point, a parameter required by the optimal caching algorithm - time of next reference for a block - can be calculated. This involves passing through the file and building a list of reference times for each block then, on a second pass, annotating each reference to a block with the time of its next reference. This algorithm is not complicated but the size of the data set is quite large (composite files were typically 2.5 megabytes for this experiment). The list of reference times would not fit in any of the Interlisp systems available (TOPS-20, Unix or Xerox Dandelion/Dandetiger). The algorithm was rewritten in two kilobytes of finely honed Zetalisp code and run on a Symbolics 3600 lisp machine.

Finally, the trace driver of the simulation package required that at each timestamp,
the number of read, write and hint transactions occurring during that time period be known. Code to count the transaction types and annotate the timestamps amounted to three kilobytes of Interlisp code.

An excerpt from a composite file ready to drive a simulation is given in figure 4-10. Timestamps are followed by a count of the number of read, write and hint transactions in that time period. Transactions directly follow the timestamp.

```
87.41 3 0 0
r b 6280 179842446 0 0 4 87.55
r 1 6280 179807495 87.55
r b 6280 179842536 0 0 4 87.71
87.45 1 0 0
r b 6280 179842460 179842460 189 2 87.57
87.55 2 1 0
r b 6280 179842446 0 0 4 3399.83
r 1 6280 179807495 0.0
w 1 6280 179807495 0.0
87.57 1 0 0
r b 6280 179842460 179842460 189 2 87.65
87.65 1 0 0
r b 6280 179842460 179842536 189 2 3400.19
87.71 1 1 0
r b 6280 179842536 179842528 189 0.0
w b 6280 179842536 189 2 3000.83
87.77 0 1 0
w 1 6280 179807495 3399.87
382.39 1 0 0
r b 6280 179208328 179208484 190 1 382.45
```

Figure 4-10: Composite Trace Example

Characterizations of the session traces and the details of their lamination into the three composite traces used in this experiment are given in Appendix A.
4.5 Simulations

All simulations were written in the PAWS language as described in The Performance Analyst's Workbench System -- User's manual [PAWS 84]. Simulations were run on the VAX version of the PAWS system.

The three different simulations (LRU, Optimal and Intelligent) share certain features. These are described in this section. Their unique characteristics are discussed in subsequent subsections.

All simulations are run within a standard context (see figure 4-11). A composite trace is read by a trace driver module in PAWS, which passes transactions to a module which performs simulated process access sequencing. The sequencing module generates a new composite trace of transaction arrival times. Transactions then pass to the file server module where response times are measured.
PAWS is not normally trace driven. The language is, however, sufficiently flexible and extensible to allow a trace driver to be written. The trace driver developed for this experiment is described in detail in Appendix B. At this point, suffice it to say that the trace driver introduces into the PAWS simulation transactions from a composite trace log at the times indicated in the file which contain all the information from the trace file about that transaction.

Appendix B also details the operation of the process access sequencing module. Put most generally, sequencing ensures that any process has only one transaction active in the
simulation at any time. Transactions arriving at the simulation before the completion of the previous transaction are delayed. An attempt is made to maintain inter-burst intervals for transactions. The sequencing module generates a composite trace of transactions after any delays have been imposed. This trace is process access sequenced for the simulation which sequenced it (i.e., sequencing the sequenced trace would result in no delays and a process access sequenced trace identical to the first sequenced trace).

Each file server simulation shares the same basic structure (see figure 4-12). CPU and DISK service is apportioned by the model. In addition, cache is modeled by nodes InCache? (which checks for residency in the cache) and Cache_Update (which updates cache). It is assumed that each transaction requires CPU service to be received from the network. Write transactions update the cache, receive CPU service for the overhead of cache updating, receive DISK service (for the write to the disk) and receive CPU service for encoding a response back onto the network. Read transactions check for cache residency (with an associated CPU overhead), if not found they receive DISK service, then all update the cache (with associated CPU overhead) and encode a response (further CPU overhead) for network output.
The data structure used to represent all cache structures is hash table based. Hash tables of small size ($n = 100$) point at doubly linked lists of hash elements. The hash key is an element's device/block (or device/inode) number. All elements are also contained in a doubly linked list. The hash table is used to quickly check for occupancy in the cache structure. The linked list is used to maintain the ordering of elements for the replacement algorithms. Thus with minimal storage overhead, lookup of either specific elements or the next element to be replaced are quite rapid. Because the two structures are linked, maintenance of the cache structures is quite efficient.
Simulations terminate when the trace log reaches end of file and no further transactions remain in the simulation.

Each simulation type was run with three cache sizes. The three cache sizes used in the simulations are given in table 4-1. In addition, total cache size in megabytes is calculated (based on a block size of 4096 bytes and an inode size of 512 bytes).

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>300</th>
<th>500</th>
<th>800</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocks</td>
<td>300</td>
<td>500</td>
<td>800</td>
</tr>
<tr>
<td>Inodes</td>
<td>75</td>
<td>125</td>
<td>200</td>
</tr>
<tr>
<td>Cache Size (megabytes)</td>
<td>1.27</td>
<td>2.30</td>
<td>3.69</td>
</tr>
</tbody>
</table>

Table 4-1: Cache Sizes - Number of Units

4.5.1 LRU

A transaction flow diagram of the LRU simulation is given in figure 4-13. Transaction phases are annotated on the edges which connect nodes. Nodes of the same name represent single service nodes which are graphically duplicated for clarity of presentation (e.g., all CPU nodes represent a single CPU service queue).
Figure 4-13: Transaction Flow Diagram
LRU Model

The LRU simulation is quite simple. There are two caches – one for disk blocks and
one for inodes. Cache lookup involves hashing into the appropriate cache and searching down the linked list for the desired element. Cache update is performed in one of two ways. If the element was already in the cache it is removed from its current place in the replacement linked list and moved to the head. If the element is new to the cache, the element at the tail of the replacement list is moved to the head and filled with the new device/block or device/inode number.

Hint transactions are ignored.

4.5.2 Optimal

A transaction flow diagram of the Optimal simulation is given in figure 4-14. Transaction phases are annotated on the edges which connect nodes. Nodes of the same name represent single service nodes which are graphically duplicated for clarity of presentation (e.g., all CPU nodes represent a single CPU service queue).
Figure 4-14: Transaction Flow Diagram
Optimal Model

Optimal cache management cannot be achieved if exact times of arrival at the node
which updates cache are not known. In order for this to be the case, all sources of variance must be removed from the path between arrival at the simulation model and cache updating. Thus in the Optimal model, all service (CPU and DISK) is delayed until after cache updating. To prevent super optimal performance from the immediate cache update (where a transaction updates the cache and queues for disk service and a second transaction checks the cache, finds it updated and continues while the first transaction is still waiting for disk service) all transactions are held at Cache_Update while the block they reference is queued for disk service. Upon completion of disk service, a transaction passes to node Freeit which frees all transactions halted at Cache_Update.

The Optimal cache management is identical to the LRU simulation in all respects save one. The replacement list is maintained in order by time of next reference. Replacement is not performed if the time of next reference of the new element is greater than that of the tail of the reference list. If replacement is performed, the last element is filled with the new data and is inserted into the replacement list by its time of next reference.

Hint transactions are ignored.

4.5.3 Intelligent

A transaction flow diagram of the LRU simulation is given in figure 4-13. Transaction phases are annotated on the edges which connect nodes. Nodes of the same name represent single service nodes which are graphically duplicated for clarity of presentation (e.g., all CPU nodes represent a single CPU service queue).
Figure 4-15: Transaction Flow Diagram
Intelligent Model

The Intelligent simulation differs quite markedly from the two previous simulations.
The cache structure is more complex, the cache is preloaded, hint guided read ahead is employed and hints (in general) are used. These will be discussed in turn.

The cache of the Intelligent simulation is dynamically managed (see figure 4-16). The system partition is fixed at \((28 + \text{(total cache size} / 20))\) for blocks and 38\% of inodes. The remainder is allocated equally among however many processes are currently active. Partitions are not rigidly enforced — if a partition is not being fully used, other partitions may grow beyond their nominal size. However, when that under used partition begins to demand more space, it is given space, up to the partition size, from those partitions most over their nominal sizes.
Figure 4-16: IFS Cache Partitioning
Process partitions are internally dynamically subpartitioned. There are initial subpartitions for directory blocks and executable page blocks. Subpartitions are created for files which are accessed by the process. The cache replacement algorithm attempts to maintain a balance of internal subpartitions. The balance is achieved when all file subpartitions are the same size, the directory block subpartition is half the size of a file subpartition and the executable page subpartition is two thirds the size of a file subpartition. The system partition is a degenerate process partition with a directory block and executable page subpartitions but no file subpartitions.

Cache lookup is similar to that found in the LRU and Optimal simulations, save that the lookup is iterated over all partitions and subpartitions.

Updating the cache becomes a bit involved. If the element is already in the cache and is in a subpartition owned by the process initiating the transaction, the element is moved to the head of its replacement list. If the element is found in some other process' (or the system) subpartition then it is moved to the head of that subpartition's replacement list -- unless that subpartition is being managed by a MRU discipline, in which case it is left alone. If the element is found on the garbage list, it is removed from the garbage list and inserted into the proper sub partition of the requesting process' partition (see below).

If the element is not located in the cache, updating the cache involves obtaining a new element (usually by replacing something currently in the cache) and inserting it into the appropriate subpartition. Partitions and subpartitions which do not exist but which are required for insertion are created. Updates made by process 0 are directed to the system partition, all others update an appropriate process partition. Updates to directory block or executable page subpartitions within a process partition are indicated by respective cache discipline hints in the transaction (see figure 4-4). File subpartitions are indicated by a file type cache discipline hint and the specification of a file group specification in the transaction. Don't cache transactions are placed at the head of the garbage list (described below).
Finding a cache element for replacement involves several steps. First, if the number of elements of that type (inode or block) outstanding are less than the maximum cache size, a new cache element is created. If the number of elements on the garbage list is greater than some specified number (five for blocks, zero for inodes) the tail of the garbage list is chosen for replacement. Otherwise the partition most over its nominal size is selected, the subpartition within that process most over its nominal size is selected and the most expendable element within that subpartition selected for replacement. The most expendable element is based upon the cache discipline of the subpartition (which is essentially LRU for everything except MRU file partitions).

Two types of "pure" hint transactions exist — file group transfer and process termination. File group transfer is used to move a file subpartition from one process partition to another (usually in expectation of the demise of the first process). Process termination marks a process partition as terminated. All subpartitions without pending operations (i.e., transactions enqueued for disk service) are cleared and their contents transferred to the head of the garbage list. As pending operations complete, subpartitions are cleared and their contents added to the garbage list. When all a partition's subpartitions have cleared, the process partition is deleted. This is necessary in order that transactions returning from disk service do not recreate recently deleted partitions for which no future termination can be expected.

At the start of the simulation, there exists a list of 187 system inodes, 707 executable page blocks and 127 system directory blocks. Space permitting, the system inode cache is preloaded with the indicated inodes, system directory blocks are allowed to fill up to half the block cache and system executable blocks allowed to fill the rest. While allowed by the general statement of the algorithm, the system cache is not reloaded during idle periods. This is due to the rather brief duration of the simulation runs.

Sequential read requests specify the next block to be read after the current request. Reading of these blocks is handled in a background fashion. These next blocks are kept in a
list with most recent requests added to the tail of the list. The intelligent simulation keeps track of the disk queue. When there are no outstanding requests for disk service, it searches the list from the head, discards any requests for "next blocks" already in the cache (presumably they've already been formally requested and that request satisfied) and submits a single service to the disk queue at a lower priority (hence foreground requests will override background service). When the background disk request is satisfied, it is added to the requesting process' subpartition.

4.6 Validation

Validating the simulation models was both an ongoing effort and a criterion event.

The PAWS model was very similar for all three simulations and very simple for all. Models were all less than 6.5K bytes of paws code (see B.4. Few errors could be expected in the PAWS code past the initial debugging stage.

The cache management algorithms invoked by PAWS as USS modules were more complex (All USS modules for all simulations are contained in ~80K bytes of C source code). Considerable effort was spent in ensuring their correctness. Proper modular design techniques were used in their development, as was a fairly rigorous set of programming and commenting disciplines. Internal consistency checks and debugging aids were designed in from the beginning and remained active throughout the production runs.

Once debugged to the point of running, all simulations were stepped through known states with specific attention to boundary cases. This found most errors. All simulations in production made periodic dumps of their state. These dumps were carefully examined after the simulation run for anomalous conditions. On this basis, the author believes that the simulations ran with few if any deviations from the foregoing descriptions of the algorithms; any undetected errors are highly unlikely to have majorly impacted the results.
Parameters for the model were selected from the literature [Lazowska 84] and slightly tuned to obtain average response values comparable to those measured locally.

Finally, a validation experiment was performed. It was desirable to compare on a transaction by transaction basis the behavior of the LRU simulation with that of a real Unix system. The trace information captured by the kernel modifications did not include individual disk transaction response times (i.e., it was known when a disk request was made but not when it was completed). Completion times were, however, known for the system calls which engendered the disk activity. Thus one could compare the completion times of disk requests from the LRU simulation with the completion times of the bracketing system calls in the original trace log. Any disk activity completing after its bracketing system call indicated a simulation error.

The LRU simulation was modified to log the time of transaction exit from the simulation. The session trace log of a representative terminal session (composite simulation traces could not be used as they comprise a partially synthetic workload) was compared transaction by transaction with the log of disk request completion times which resulted. This comparison proved hard to automate and was done by hand in a rather tedious fashion.

The results of this validation experiment are presented in the following chapter.

4.7 Experimental Simulations

These simulations form the central thrust of this paper in that they address the prime issue of whether intelligent caching algorithms do exhibit superior performance over conventional caching algorithms.

The 3 data sets were process access sequenced for each of the 3 cache sizes. Sequencing for each cache size was required to permit minimum modification to the data sets -- larger cache sizes allowed faster response times and required less modification to be process legal.
Intelligent algorithms are compared to LRU and Optimal algorithms over 3 cache sizes and over three data sets. This experimental design is presented graphically in figure 4-17.

An identical set of simulations were run on the original data (simulated load data). While not strictly process legal, this data contains higher (and more characteristic for Unix) burst rates. The results of these simulations must be viewed with caution but are suggestive of performance under high rates of transaction arrival.

Figure 4-17: Experimental Simulation Design
4.8 Factor Analysis

If indeed the intelligent algorithms do perform better than conventional caching algorithms, it is desirable to know why. The performance augmentations of the intelligent caching algorithm (cache preload and background readahead) operate in addition to intelligent cache management. Thus one can selectively turn them off and observe the impacts on performance.

In a further set of simulations, performance is measured for optimal caching alone, optimal caching with cache preload and optimal caching with background readahead. These measures can be contrasted with those already obtained in the previous series of simulations - the full intelligent algorithm (which includes both cache preloading and background readahead with intelligent cache management) and the LRU algorithm. This experimental design is presented graphically in figure 4-18. These simulations were performed on both simulated process access sequence and simulated load data. Again, caution must be exercised in interpreting the results of simulations employing simulated load data.
Figure 4-18: Factor Analysis Simulation Design

The results of all simulation series are given in the next chapter.
Chapter 5

Results

This chapter discusses the results of the simulations described in the previous chapter. Results of the validation simulation, the central experimental simulations and the factor analysis simulations are described in turn, for both simulated process access sequence and simulated load data.

5.1 Validation Results

The validation simulation described in the previous chapter was performed on one of the more representative sessions (see terminal session KMK2 in Appendix A). The results are presented in table 5-1. A frequency distribution for the simulation misses (i.e., where disk transactions completed after the bracketing system calls) is given in figure 5-1.

The results from the simulation were judged to be in sufficient agreement with the observed data to validate the simulation.
<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responses Within Bound</td>
<td>797</td>
</tr>
<tr>
<td>Responses Out Of Bound</td>
<td>61</td>
</tr>
<tr>
<td>Error Percentage</td>
<td>7%</td>
</tr>
<tr>
<td>Average Error</td>
<td>76.72 msec.</td>
</tr>
<tr>
<td>Mean Read Response</td>
<td>35.283 msec.</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>55.322</td>
</tr>
<tr>
<td>Mean Write Response</td>
<td>111.747 msec.</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>66.800</td>
</tr>
</tbody>
</table>

Table 5-1: Validation Results
5.2 Simulation Results - Simulated Process Access Sequence Data

The results of the various simulations are presented in tables 5-2, 5-3, and 5-4. The same results are given graphically in figures 5-2 to 5-7. In the figures, the upper solid line represents LRU algorithm response, the lower solid line represents optimal algorithm response, and the dotted line is the response time of the Intelligent algorithm.

Standard deviations closely follow the mean response times given in tables 5-2 to 5-4.
The remainder of this study will discuss only means. It should be recognized that the discussions of the results also apply to variance measures of response time.

Similarly, write response times are provided but not discussed. Write response is of less interest to caching studies, has less variance and is provided only to demonstrate that nothing unexpected occurred in the simulations. Write response measures will not be mentioned further.

Regarding read response measures, in all instances (not unexpectedly), the LRU algorithm performed less well (i.e., response time was greater) than the optimal algorithm. The intelligent algorithm was inferior to the LRU in four of the nine test cases. Performance of the intelligent algorithm was intermediate (between LRU and optimal) in two simulations. In the three other cases the intelligent algorithm exhibited performance superior to that of the optimal algorithm.

It may at first glance appear paradoxical that the optimal algorithm should be outperformed. This reflects a difficulty in terminology. The optimal algorithm is performing optimal cache replacement. The intelligent algorithm is doing intelligent cache replacement and, additionally, using the preloading and prefetching permitted by the knowledge based algorithms. The optimal algorithm is optimal only in cache replacement strategy, while the intelligent algorithm employs additional strategies.

Caching algorithm behavior was similar for each data set. For small cache sizes, intelligent caching was worse than LRU caching by 5% to 28%. This is due to an innate characteristic of the intelligent algorithm; dynamic cache partitioning permits each process its fair share of cache. Sometimes the intelligent caching algorithms permit a process to make poor use of a portion of its cache share. With small cache sizes, the poor behavior of a process holding on to its cache share and using it unwisely can adversely effect average cache behavior. This is primarily manifested by interactive processes which acquire a partition and then remain inactive for long periods of time, idling a significant percentage of the cache.
As cache sizes become larger, the effects of occasional cache holding by processes becomes diluted by the generally superior intelligent strategies in the larger cache — a smaller percentage of the cache is idled and the rest is more efficiently used. Moderate cache sizes yield intelligent cache performances ranging from roughly equal to LRU to 9% improved over LRU. The largest cache size shows performance gains of 13% to 16%.

Observing the distribution of read response times between algorithms, it becomes clear that the load imposed by the simulated process access data set was not terribly severe. More data to further load the system would be desirable; unfortunately the need for process access sequencing was recognized after data gathering ceased to be possible (as has been previously discussed).
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache Size</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
<td>500</td>
<td>800</td>
</tr>
<tr>
<td>LRU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>13.9</td>
<td>13.3</td>
<td>10.7</td>
</tr>
<tr>
<td>(read)</td>
<td>19.9</td>
<td>22.0</td>
<td>16.9</td>
</tr>
<tr>
<td>(write)</td>
<td>47.8</td>
<td>50.5</td>
<td>48.1</td>
</tr>
<tr>
<td>(write)</td>
<td>18.6</td>
<td>26.6</td>
<td>20.1</td>
</tr>
<tr>
<td>Optimal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>10.5</td>
<td>10.5</td>
<td>10.0</td>
</tr>
<tr>
<td>(read)</td>
<td>15.7</td>
<td>16.6</td>
<td>16.0</td>
</tr>
<tr>
<td>(write)</td>
<td>44.9</td>
<td>47.6</td>
<td>47.5</td>
</tr>
<tr>
<td>(write)</td>
<td>15.8</td>
<td>20.9</td>
<td>20.0</td>
</tr>
<tr>
<td>Intelligent</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>17.8</td>
<td>13.5</td>
<td>9.2</td>
</tr>
<tr>
<td>(read)</td>
<td>24.6</td>
<td>19.5</td>
<td>13.3</td>
</tr>
<tr>
<td>(write)</td>
<td>52.5</td>
<td>49.4</td>
<td>46.2</td>
</tr>
<tr>
<td>(write)</td>
<td>21.4</td>
<td>18.5</td>
<td>16.3</td>
</tr>
</tbody>
</table>

mean

standard deviation

Table 5-2: Response Times (msec.)
Simulated Process Access Sequence Data Set A
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
</tr>
<tr>
<td>LRU</td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>8.5</td>
</tr>
<tr>
<td></td>
<td>10.9</td>
</tr>
<tr>
<td>(write)</td>
<td>34.8</td>
</tr>
<tr>
<td></td>
<td>4.9</td>
</tr>
<tr>
<td>Optimal</td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>7.1</td>
</tr>
<tr>
<td></td>
<td>9.1</td>
</tr>
<tr>
<td>(write)</td>
<td>34.8</td>
</tr>
<tr>
<td></td>
<td>5.0</td>
</tr>
<tr>
<td>Intelligent</td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>8.9</td>
</tr>
<tr>
<td></td>
<td>11.4</td>
</tr>
<tr>
<td>(write)</td>
<td>34.8</td>
</tr>
<tr>
<td></td>
<td>4.9</td>
</tr>
</tbody>
</table>

mean

standard deviation

Table 5-3: Response Times (msec.)
Simulated Process Access Sequence Data Set B
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
</tr>
<tr>
<td><strong>LRU</strong></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>8.8</td>
</tr>
<tr>
<td></td>
<td>11.8</td>
</tr>
<tr>
<td>(write)</td>
<td>35.7</td>
</tr>
<tr>
<td></td>
<td>6.5</td>
</tr>
<tr>
<td><strong>Optimal</strong></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>7.3</td>
</tr>
<tr>
<td></td>
<td>9.8</td>
</tr>
<tr>
<td>(write)</td>
<td>35.5</td>
</tr>
<tr>
<td></td>
<td>6.2</td>
</tr>
<tr>
<td><strong>Intelligent</strong></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>10.3</td>
</tr>
<tr>
<td></td>
<td>13.1</td>
</tr>
<tr>
<td>(write)</td>
<td>35.8</td>
</tr>
<tr>
<td></td>
<td>6.4</td>
</tr>
</tbody>
</table>

mean

standard deviation

Table 5-4: Response Times (msec.)
Simulated Process Access Sequence Data Set C
Figure 5-2: Read Transactions
Simulated Process Access Sequence Data Set A
Figure 5-3: Write Transactions
Simulated Process Access Sequence Data Set A
Figure 5-4: Read Transactions
Simulated Process Access Sequence Data Set B
Figure 5-5: Write Transactions
Simulated Process Access Sequence Data Set B
Figure 5-6: Read Transactions
Simulated Process Access Sequence Data Set C
Cache hit ratios are given for the LRU and optimal simulations in tables 5-5 and 5-6 respectively. These simulations did not capture detailed information about cache hits; cache hit ratios for both blocks and inodes are co-mingled.

Hit ratios for the intelligent simulation are given in Table 5-7. More detailed statistics were captured by the intelligent model. This was deemed necessary due to the number of types of data cached. The average hit ratio reported is, however, directly comparable to those reported in tables 5-5 and 5-6.
Response measures appear, not unexpectedly, directly related to cache hit ratios.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
</tr>
<tr>
<td>A</td>
<td>.85</td>
</tr>
<tr>
<td>B</td>
<td>.87</td>
</tr>
<tr>
<td>C</td>
<td>.87</td>
</tr>
</tbody>
</table>

Table 5-5: Cache Hit Ratios - LRU Algorithm
Simulated Process Access Sequence Data

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
</tr>
<tr>
<td>A</td>
<td>.91</td>
</tr>
<tr>
<td>B</td>
<td>.91</td>
</tr>
<tr>
<td>C</td>
<td>.92</td>
</tr>
</tbody>
</table>

Table 5-6: Cache Hit Ratios - Optimal Algorithm
Simulated Process Access Sequence Data
<table>
<thead>
<tr>
<th>Data Set</th>
<th>Cache Size</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
<td>500</td>
<td>800</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>swap</td>
<td>.20</td>
<td>.45</td>
<td>.70</td>
</tr>
<tr>
<td>nocache</td>
<td>.69</td>
<td>.72</td>
<td>.81</td>
</tr>
<tr>
<td>LRU</td>
<td>.82</td>
<td>.88</td>
<td>.93</td>
</tr>
<tr>
<td>MRU</td>
<td>.87</td>
<td>.93</td>
<td>.94</td>
</tr>
<tr>
<td>dir</td>
<td>.81</td>
<td>.88</td>
<td>.99</td>
</tr>
<tr>
<td>inodes</td>
<td>.83</td>
<td>.99</td>
<td>.92</td>
</tr>
<tr>
<td>average</td>
<td>.77</td>
<td>.84</td>
<td>.93</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>swap</td>
<td>.22</td>
<td>.50</td>
<td>.77</td>
</tr>
<tr>
<td>nocache</td>
<td>.84</td>
<td>.85</td>
<td>.85</td>
</tr>
<tr>
<td>LRU</td>
<td>.90</td>
<td>.92</td>
<td>.92</td>
</tr>
<tr>
<td>MRU</td>
<td>.92</td>
<td>.93</td>
<td>.94</td>
</tr>
<tr>
<td>dir</td>
<td>.93</td>
<td>.99</td>
<td>.99</td>
</tr>
<tr>
<td>inodes</td>
<td>.86</td>
<td>.92</td>
<td>.94</td>
</tr>
<tr>
<td>average</td>
<td>.85</td>
<td>.91</td>
<td>.94</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>swap</td>
<td>.17</td>
<td>.49</td>
<td>.74</td>
</tr>
<tr>
<td>nocache</td>
<td>.81</td>
<td>.83</td>
<td>.83</td>
</tr>
<tr>
<td>LRU</td>
<td>.89</td>
<td>.92</td>
<td>.93</td>
</tr>
<tr>
<td>MRU</td>
<td>.88</td>
<td>.94</td>
<td>.94</td>
</tr>
<tr>
<td>dir</td>
<td>.88</td>
<td>.98</td>
<td>.99</td>
</tr>
<tr>
<td>inodes</td>
<td>.85</td>
<td>.91</td>
<td>.94</td>
</tr>
<tr>
<td>average</td>
<td>.82</td>
<td>.90</td>
<td>.94</td>
</tr>
</tbody>
</table>

Table 5-7: Cache Hit Ratios - Intelligent Algorithm
Simulated Process Access Sequence Data

Utilization statistics for all simulations are given in tables 5-8 to 5-10.
Utilization is inversely proportional to cache hit ratios. The observed utilizations are not indicative of either CPU or Disk performance bottlenecks.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
</tr>
<tr>
<td>LRU</td>
<td></td>
</tr>
<tr>
<td>(CPU)</td>
<td>21.2</td>
</tr>
<tr>
<td>(DISK)</td>
<td>21.4</td>
</tr>
<tr>
<td>Optimal</td>
<td></td>
</tr>
<tr>
<td>(CPU)</td>
<td>19.9</td>
</tr>
<tr>
<td>(DISK)</td>
<td>18.8</td>
</tr>
<tr>
<td>Intelligent</td>
<td></td>
</tr>
<tr>
<td>(CPU)</td>
<td>22.9</td>
</tr>
<tr>
<td>(DISK)</td>
<td>24.7</td>
</tr>
</tbody>
</table>

Table 5-8: Simulated Process Access Sequence Data Set A
Simulation Resource Utilization (%)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
</tr>
<tr>
<td>LRU</td>
<td></td>
</tr>
<tr>
<td>(CPU)</td>
<td>10.3</td>
</tr>
<tr>
<td>(DISK)</td>
<td>10.1</td>
</tr>
<tr>
<td>Optimal</td>
<td></td>
</tr>
<tr>
<td>(CPU)</td>
<td>9.8</td>
</tr>
<tr>
<td>(DISK)</td>
<td>9.19</td>
</tr>
<tr>
<td>Intelligent</td>
<td></td>
</tr>
<tr>
<td>(CPU)</td>
<td>10.5</td>
</tr>
<tr>
<td>(DISK)</td>
<td>10.5</td>
</tr>
</tbody>
</table>

Table 5-9: Simulated Process Access Sequence Data Set B
Simulation Resource Utilization (%)
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache Size</th>
<th>300</th>
<th>500</th>
<th>800</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
<td>(CPU)</td>
<td>9.6</td>
<td>9.3</td>
<td>9.1</td>
</tr>
<tr>
<td></td>
<td>(DISK)</td>
<td>9.4</td>
<td>8.9</td>
<td>8.5</td>
</tr>
<tr>
<td>Optimal</td>
<td>(CPU)</td>
<td>9.1</td>
<td>9.0</td>
<td>8.9</td>
</tr>
<tr>
<td></td>
<td>(DISK)</td>
<td>8.5</td>
<td>8.3</td>
<td>8.1</td>
</tr>
<tr>
<td>Intelligent</td>
<td>(CPU)</td>
<td>10.0</td>
<td>9.3</td>
<td>9.0</td>
</tr>
<tr>
<td></td>
<td>(DISK)</td>
<td>10.37</td>
<td>8.7</td>
<td>8.2</td>
</tr>
</tbody>
</table>

Table 5-10: Simulated Process Access Sequence Data Set C Simulation Resource Utilization (%)

5.2.1 Factor Analysis Results

The results of the various factor analysis simulations on simulated process access sequence data discussed in the previous chapter are presented in tables 5-11, 5-12 and 5-13. The same results are given graphically in figures 5-8 to 5-13. In the figures, the upper solid line represents LRU algorithm response, the lower solid line represents full intelligent algorithm response, the upper dotted line is the response time of an algorithm using intelligent caching alone (without cache preload), and the lower dotted line is the response time of an algorithm employing intelligent caching and cache preload — in the plot for data set C read transactions, the lines for intelligent caching alone and intelligent caching with preload are identical. Since in all instances the response times of intelligent caching with background readahead are identical to those obtained with intelligent caching alone, those results are not graphed.

In all simulations, performance of the algorithm using intelligent caching and background readahead did not differ from that of intelligent caching alone; cache preloading always offered performance boosts over pure intelligent cache management.
In data set A, the intelligent algorithm performs less well than the LRU algorithm with cache size 300, similarly with cache size 500 and better with cache size 800. Intelligent caching with preload was superior to intelligent caching alone in all cases. The addition of background readahead to intelligent caching with cache preload resulted in slight performance improvements.

In data set B, algorithms employing intelligent caching outperformed the LRU algorithm in the largest two cache sizes. Performance of intelligent caching alone was intermediate between LRU performance and performance obtained with intelligent caching and cache preload in those cases. The addition of background readahead to intelligent caching with cache preload had no observable effect except with cache size 300 — where the effect was minimal.

In data set C, dramatic performance differences of the various algorithms relative to each other were observed. With the smallest cache size (300), the performance of all intelligent caching based algorithms was inferior to that of the LRU algorithm. With the median cache size (500), performance of all algorithms was comparable. With the largest size cache (800) intelligent caching alone is equal in performance to the LRU algorithm and with cache preloading performed substantially better than LRU. The addition of background readahead to intelligent caching with cache preload had no observable effect.

Intelligent caching alone performs less well than the LRU algorithm with the small cache size (due to the previously discussed holding effect) and roughly comparably to the LRU algorithm with the two larger cache sizes.

Cache preloading in addition to intelligent caching always provides substantial cache performance improvements. The degree of improvement increases with cache size.

Surprisingly, readahead does not provide major performance improvements; indeed it seems to have no effect save when some coupled with cache preloading. Presumably the
bursty nature of requests prevents readahead from occurring during periods when it is known what to read next. Cache preloading may be acting to reduce disk load and permit such readahead — in effect handling the bursts in the cache and not passing them along to disk.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
</tr>
<tr>
<td>LRU</td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>13.9</td>
</tr>
<tr>
<td>(write)</td>
<td>47.8</td>
</tr>
<tr>
<td>Intelligent Caching</td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>19.2</td>
</tr>
<tr>
<td>(write)</td>
<td>55.1</td>
</tr>
<tr>
<td>Intelligent Caching and Preload</td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>18.2</td>
</tr>
<tr>
<td>(write)</td>
<td>53.3</td>
</tr>
<tr>
<td>Intelligent Caching and Background Readahead</td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>19.2</td>
</tr>
<tr>
<td>(write)</td>
<td>55.1</td>
</tr>
<tr>
<td>Intelligent Caching, Preload and Background Readahead</td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>17.8</td>
</tr>
<tr>
<td>(write)</td>
<td>52.5</td>
</tr>
</tbody>
</table>

Table 5-11: Simulated Process Access Sequence Data Set A
Factor Analysis Response Times (msec.)
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
</tr>
<tr>
<td><strong>LRU</strong></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>8.5</td>
</tr>
<tr>
<td>(write)</td>
<td>34.8</td>
</tr>
<tr>
<td><strong>Intelligent Caching</strong></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>9.2</td>
</tr>
<tr>
<td>(write)</td>
<td>34.9</td>
</tr>
<tr>
<td><strong>Intelligent Caching and Preload</strong></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>9.0</td>
</tr>
<tr>
<td>(write)</td>
<td>34.9</td>
</tr>
<tr>
<td><strong>Intelligent Caching and Background Readahead</strong></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>9.2</td>
</tr>
<tr>
<td>(write)</td>
<td>34.9</td>
</tr>
<tr>
<td><strong>Intelligent Caching, Preload and Background Readahead</strong></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>8.9</td>
</tr>
<tr>
<td>(write)</td>
<td>34.8</td>
</tr>
</tbody>
</table>

Table 5-12: Simulated Process Access Sequence Data Set B
Factor Analysis Response Times (msec.)
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
</tr>
<tr>
<td><strong>LRU</strong></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>8.8</td>
</tr>
<tr>
<td>(write)</td>
<td>35.7</td>
</tr>
<tr>
<td><strong>Intelligent Caching</strong></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>10.4</td>
</tr>
<tr>
<td>(write)</td>
<td>35.8</td>
</tr>
<tr>
<td><strong>Intelligent Caching and Preload</strong></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>10.3</td>
</tr>
<tr>
<td>(write)</td>
<td>35.8</td>
</tr>
<tr>
<td><strong>Intelligent Caching and Background Readahead</strong></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>10.4</td>
</tr>
<tr>
<td>(write)</td>
<td>35.8</td>
</tr>
<tr>
<td><strong>Intelligent Caching, Preload and Background Readahead</strong></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>10.3</td>
</tr>
<tr>
<td>(write)</td>
<td>35.8</td>
</tr>
</tbody>
</table>

Table 5-13: Simulated Process Access Sequence Data Set C
Factor Analysis Response Times (msec.)
Figure 5-8: Factor Analysis Read Transactions
Simulated Process Access Sequence Data Set A
Figure 5-9: Factor Analysis Write Transactions
Simulated Process Access Sequence Data Set A
Figure 5-10: Factor Analysis Read Transactions
Simulated Process Access Sequence Data Set B
Figure 5-11: Factor Analysis Write Transactions
Simulated Process Access Sequence Data Set B
Figure 5-12: Factor Analysis Read Transactions
Simulated Process Access Sequence Data Set C
5.3 Simulation Results - Simulated Load Data

The following section describes results based upon simulations run on simulated load data. While these results are suggestive of high burst rate performance, they must be viewed with caution as they are not strictly legal in their process behavior.

The results of the various simulations using simulated load data are presented in tables 5-14, 5-15, and 5-16. The same results are given graphically in figures 5-14 to 5-19. In
the figures, the upper solid line represents LRU algorithm response, the lower solid line represents optimal algorithm response, and the dotted line is the response time of the Intelligent algorithm. Where a lower solid line is not visible, it is merged with the baseline due to lack of graphical resolution.

In all instances (not unexpectedly), the LRU algorithm performed less well (i.e., response time was greater) than the optimal algorithm. The intelligent algorithm was inferior to the LRU in only one of nine cases. Performance of the intelligent algorithm was intermediate (between LRU and Optimal) in the remaining simulations.

Performance of the intelligent algorithm on the simulated load data is impressive. With the smallest cache where the intelligent algorithm does worst (see the discussion of cache holding in small caches in the previous sections), response times for the intelligent algorithm are superior to LRU by 198% (data set A), 38% (data set B) and worse by 15% (data set C). Data set C was constructed to minimize performance of the intelligent algorithm — presumably it does so by encouraging cache holding behavior of some processes at the expense of other processes.

With moderate cache sizes, intelligent algorithm performance increases over LRU range from 3% to 238%. Large cache sizes result in performance gains of 69% to 340%.

Clearly, the wide range of response times both between algorithms and between cache sizes indicate that the simulated load data is more difficult for the simulated file server to handle; changes in caching efficiency have a much greater effect than in the simulated process access sequence data. The simulated load data preserve more typical burst rates which favors longer queue lengths which in turn strongly emphasize differences of cache efficiencies.

The problem of process access sequencing was not encountered until after preliminary simulations had been run on the simulated load data. The results of those simulations were sufficiently encouraging to allow discontinuation of data collection (as described in a previous
The results presented in this section were considered final until issues of legal process behavior were considered.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache Size</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
<td>500</td>
<td>800</td>
</tr>
<tr>
<td>LRU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>70700</td>
<td>46600</td>
<td>30900</td>
</tr>
<tr>
<td></td>
<td>129000</td>
<td>97100</td>
<td>73700</td>
</tr>
<tr>
<td>(write)</td>
<td>2244000</td>
<td>197000</td>
<td>170000</td>
</tr>
<tr>
<td></td>
<td>133000</td>
<td>105000</td>
<td>84600</td>
</tr>
<tr>
<td>Optimal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>201</td>
<td>166</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>495</td>
<td>443</td>
<td>333</td>
</tr>
<tr>
<td>(write)</td>
<td>607</td>
<td>543</td>
<td>436</td>
</tr>
<tr>
<td></td>
<td>1040</td>
<td>983</td>
<td>775</td>
</tr>
<tr>
<td>Intelligent</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>23700</td>
<td>13800</td>
<td>7030</td>
</tr>
<tr>
<td></td>
<td>41500</td>
<td>31700</td>
<td>21400</td>
</tr>
<tr>
<td>(write)</td>
<td>94200</td>
<td>78700</td>
<td>57900</td>
</tr>
<tr>
<td></td>
<td>48800</td>
<td>43300</td>
<td>34900</td>
</tr>
</tbody>
</table>

mean

standard deviation

Table 5-14: Response Times (msec.)
Simulated load Data Set A
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache Size</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
<td>500</td>
<td>800</td>
</tr>
<tr>
<td>LRU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>99</td>
<td>65</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>382</td>
<td>265</td>
<td>256</td>
</tr>
<tr>
<td>(write)</td>
<td>446</td>
<td>348</td>
<td>338</td>
</tr>
<tr>
<td></td>
<td>650</td>
<td>477</td>
<td>462</td>
</tr>
<tr>
<td>Optimal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>29</td>
<td>27</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>25</td>
<td>23</td>
</tr>
<tr>
<td>(write)</td>
<td>103</td>
<td>100</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>92</td>
<td>88</td>
<td>87</td>
</tr>
<tr>
<td>Intelligent</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>72</td>
<td>45</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>241</td>
<td>140</td>
<td>101</td>
</tr>
<tr>
<td>(write)</td>
<td>358</td>
<td>304</td>
<td>288</td>
</tr>
<tr>
<td></td>
<td>439</td>
<td>361</td>
<td>340</td>
</tr>
</tbody>
</table>

mean
standard deviation

Table 5-15: Response Times (msec.)
Simulated Load Data Set B
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache Size</th>
<th>300</th>
<th>500</th>
<th>800</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LRU</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td></td>
<td>169</td>
<td>103</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td></td>
<td>745</td>
<td>555</td>
<td>460</td>
</tr>
<tr>
<td>(write)</td>
<td></td>
<td>614</td>
<td>472</td>
<td>448</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1120</td>
<td>892</td>
<td>795</td>
</tr>
<tr>
<td><strong>Optimal</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td></td>
<td>30</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>29</td>
<td>27</td>
<td>25</td>
</tr>
<tr>
<td>(write)</td>
<td></td>
<td>102</td>
<td>101</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>85</td>
<td>84</td>
<td>83</td>
</tr>
<tr>
<td><strong>Intelligent</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td></td>
<td>195</td>
<td>100</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td></td>
<td>771</td>
<td>476</td>
<td>173</td>
</tr>
<tr>
<td>(write)</td>
<td></td>
<td>684</td>
<td>490</td>
<td>347</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1140</td>
<td>792</td>
<td>414</td>
</tr>
</tbody>
</table>

mean

standard deviation

Table 5-16: Response Times (msec.)
Simulated Load Data Set C
Figure 5-14: Read Transactions
Simulated Load Data Set A
Figure 5-15: Write Transactions
Simulated Load Data Set A
**Figure 5-16:** Read Transactions
Simulated Load Data Set B
Response Time (msec)

![Graph showing response time vs cache size with LRU and Intelligent lines.](image)

Figure 5-17: Write Transactions
Simulated Load Data Set B
Figure 5-18: Read Transactions
Simulated Load Data Set C
Figure 5-19: Write Transactions
Simulated Load Data Set C

Cache hit ratios are given for the LRU and optimal simulations in tables 5-17 and 5-18 respectively. Hit ratios for the intelligent simulation are given in Table 5-19. Comments about hit ratios made in the previous section apply here as well.

In contrast with hit ratios for simulated process access sequence data, hit ratios for simulated load data are smaller for data set A, slightly smaller for data set B and quite similar for data set C — across all algorithms.
<table>
<thead>
<tr>
<th>Data Set</th>
<th>Cache Size</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
<td>500</td>
<td>800</td>
</tr>
<tr>
<td>A</td>
<td>.69</td>
<td>.75</td>
<td>.80</td>
</tr>
<tr>
<td>B</td>
<td>.85</td>
<td>.89</td>
<td>.91</td>
</tr>
<tr>
<td>C</td>
<td>.86</td>
<td>.90</td>
<td>.92</td>
</tr>
</tbody>
</table>

Table 5-17: Cache Hit Ratios - LRU Algorithm
Simulated Load Data

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Cache Size</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
<td>500</td>
<td>800</td>
</tr>
<tr>
<td>A</td>
<td>.77</td>
<td>.82</td>
<td>.84</td>
</tr>
<tr>
<td>B</td>
<td>.89</td>
<td>.91</td>
<td>.93</td>
</tr>
<tr>
<td>C</td>
<td>.90</td>
<td>.92</td>
<td>.94</td>
</tr>
</tbody>
</table>

Table 5-18: Cache Hit Ratios - Optimal Algorithm
Simulated Load Data
<table>
<thead>
<tr>
<th>Data Set</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>swap</td>
<td>.26</td>
</tr>
<tr>
<td>nocache</td>
<td>.63</td>
</tr>
<tr>
<td>LRU</td>
<td>.37</td>
</tr>
<tr>
<td>MRU</td>
<td>.60</td>
</tr>
<tr>
<td>dir</td>
<td>.61</td>
</tr>
<tr>
<td>inodes</td>
<td>.78</td>
</tr>
<tr>
<td>average</td>
<td>.59</td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>swap</td>
<td>.23</td>
</tr>
<tr>
<td>nocache</td>
<td>.88</td>
</tr>
<tr>
<td>LRU</td>
<td>.84</td>
</tr>
<tr>
<td>MRU</td>
<td>.92</td>
</tr>
<tr>
<td>dir</td>
<td>.93</td>
</tr>
<tr>
<td>inodes</td>
<td>.88</td>
</tr>
<tr>
<td>average</td>
<td>.85</td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>swap</td>
<td>.17</td>
</tr>
<tr>
<td>nocache</td>
<td>.77</td>
</tr>
<tr>
<td>LRU</td>
<td>.82</td>
</tr>
<tr>
<td>MRU</td>
<td>.88</td>
</tr>
<tr>
<td>dir</td>
<td>.87</td>
</tr>
<tr>
<td>inodes</td>
<td>.85</td>
</tr>
<tr>
<td>average</td>
<td>.80</td>
</tr>
</tbody>
</table>

Table 5-19: Cache Hit Ratios - Intelligent Algorithm
Simulated Load Data

Utilization statistics for all simulations are given in tables 5-20 to 5-22.
Utilization is inversely proportional to cache hit ratios. The observed utilizations are not indicative of either CPU or Disk performance bottlenecks.

CPU utilization is roughly comparable to that observed in the simulated process access sequence data simulations. Disk utilization is roughly double that of the process access sequenced data simulations.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
</tr>
<tr>
<td>LRU</td>
<td></td>
</tr>
<tr>
<td>(CPU)</td>
<td>20.7</td>
</tr>
<tr>
<td>(DISK)</td>
<td>53.1</td>
</tr>
<tr>
<td>Optimal</td>
<td></td>
</tr>
<tr>
<td>(CPU)</td>
<td>19.3</td>
</tr>
<tr>
<td>(DISK)</td>
<td>47.1</td>
</tr>
<tr>
<td>Intelligent</td>
<td></td>
</tr>
<tr>
<td>(CPU)</td>
<td>17.2</td>
</tr>
<tr>
<td>(DISK)</td>
<td>37.4</td>
</tr>
</tbody>
</table>

Table 5-20: Simulated Load Data Set A
Simulation Resource Utilization (%)
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
</tr>
<tr>
<td><strong>LRU</strong></td>
<td></td>
</tr>
<tr>
<td>(CPU)</td>
<td>8.9</td>
</tr>
<tr>
<td>(DISK)</td>
<td>20.2</td>
</tr>
<tr>
<td><strong>Optimal</strong></td>
<td></td>
</tr>
<tr>
<td>(CPU)</td>
<td>8.6</td>
</tr>
<tr>
<td>(DISK)</td>
<td>18.8</td>
</tr>
<tr>
<td><strong>Intelligent</strong></td>
<td></td>
</tr>
<tr>
<td>(CPU)</td>
<td>8.9</td>
</tr>
<tr>
<td>(DISK)</td>
<td>19.9</td>
</tr>
</tbody>
</table>

Table 5-21: Simulated Load Data Set B
Simulation Resource Utilization (%)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
</tr>
<tr>
<td><strong>LRU</strong></td>
<td></td>
</tr>
<tr>
<td>(CPU)</td>
<td>8.8</td>
</tr>
<tr>
<td>(DISK)</td>
<td>19.8</td>
</tr>
<tr>
<td><strong>Optimal</strong></td>
<td></td>
</tr>
<tr>
<td>(CPU)</td>
<td>8.5</td>
</tr>
<tr>
<td>(DISK)</td>
<td>18.3</td>
</tr>
<tr>
<td><strong>Intelligent</strong></td>
<td></td>
</tr>
<tr>
<td>(CPU)</td>
<td>9.2</td>
</tr>
<tr>
<td>(DISK)</td>
<td>21.2</td>
</tr>
</tbody>
</table>

Table 5-22: Simulated Load Data Set C
Simulation Resource Utilization (%)
5.3.1 Factor Analysis Results

The following section describes results based upon simulations run on simulated load data. While these results are suggestive of high burst rate performance, they must be viewed with caution as they are not strictly legal in their process behavior.

The results of the various factor analysis simulations on simulated load data discussed in the previous chapter are presented in tables 5-23, 5-24 and 5-25. The same results are given graphically in figures 5-20 to 5-25. In the figures, the upper solid line represents LRU algorithm response, the lower solid line represents full intelligent algorithm response, and the dotted line is the response time of an algorithm using intelligent caching alone (without cache preload). A second, lower dotted line is the response time of an algorithm employing intelligent caching and cache preload -- in data sets B and C, that performance line exactly conforms to that of the full intelligent algorithm and is not graphed. Since in all instances the response times of intelligent caching with background readahead are identical to those obtained with intelligent caching alone, those results are not graphed.

In all simulations, performance of the algorithm using intelligent caching and background readahead did not differ from that of intelligent caching alone; cache preloading always offered performance boosts over pure intelligent cache management.

In data set A, all intelligent algorithms perform substantially better than the LRU algorithm. With cache size 300, intelligent caching with cache preload and background readahead slightly outperformed intelligent caching with only cache preload -- this is the only case using simulated load data in which this was observed. With the larger cache sizes, the addition of background readahead to intelligent caching with cache preload resulted in performance decrements.

In data set B, all algorithms employing intelligent caching outperformed the LRU algorithm. Performance of intelligent caching alone was intermediate between LRU performance and performance obtained with intelligent caching and cache preload. The
addition of background readahead to intelligent caching with cache preload had no observable effect.

In data set C, dramatic performance differences of the various algorithms relative to each other were observed. With the smallest cache size (300), the performance of all intelligent caching based algorithms was very similar and inferior to that of the LRU algorithm. With the median cache size (500), performance of all algorithms was comparable. With the largest size cache (800) intelligent caching alone was slightly inferior in performance to the LRU algorithm and with cache preloading performed substantially better than LRU. The addition of background readahead to intelligent caching with cache preload had no observable effect.

In data sets A and B, all algorithms utilizing some form of intelligent caching performed better than LRU strategies. Performance of the intelligent algorithm on data set C (which was constructed to minimize intelligent algorithm performance) is more reminiscent of that obtained on simulated process access sequence data sets (worse than LRU with cache size 300, equivalent to LRU with cache size 500 and superior at cache size 800).

As with the simulated process access sequence data, cache preloading is always useful; performance gains obtained with preloading increase with cache size. This reflects less extra information preloaded into the cache (preloads are all roughly the same size) than spare cache which can be used before one starts replacing the preloaded information.

Readahead provides occasional small performance increases. Surprisingly, it also provides appreciable performance decrements. This was observed in the data set with highest burst rates. Presumably the readahead algorithm was able to find brief periods of disk inactivity and schedule readahead. Background readahead, once scheduled, is non-preemptable. Background readahead progressed in high burst periods while more pressing foreground requests were queued. With the large queue lengths experienced, the effects of an occasional monopolization of the disk by a readahead transaction were considerably magnified.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
</tr>
<tr>
<td><strong>LRU</strong></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>70700</td>
</tr>
<tr>
<td>(write)</td>
<td>244000</td>
</tr>
<tr>
<td><strong>Intelligent Caching</strong></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>22100</td>
</tr>
<tr>
<td>(write)</td>
<td>91400</td>
</tr>
<tr>
<td><strong>Intelligent Caching and Preload</strong></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>19100</td>
</tr>
<tr>
<td>(write)</td>
<td>87100</td>
</tr>
<tr>
<td><strong>Intelligent Caching and Background Readahead</strong></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>22100</td>
</tr>
<tr>
<td>(write)</td>
<td>91400</td>
</tr>
<tr>
<td><strong>Intelligent Caching, Preload and Background Readahead</strong></td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>19000</td>
</tr>
<tr>
<td>(write)</td>
<td>76000</td>
</tr>
</tbody>
</table>

Table 5-23: Simulated Load Data Set A
Factor Analysis Response Times (msec.)
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
</tr>
<tr>
<td></td>
<td>500</td>
</tr>
<tr>
<td></td>
<td>800</td>
</tr>
<tr>
<td>LRU</td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>99</td>
</tr>
<tr>
<td>(write)</td>
<td>446</td>
</tr>
<tr>
<td>Intelligent Caching</td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>77</td>
</tr>
<tr>
<td>(write)</td>
<td>366</td>
</tr>
<tr>
<td>Intelligent Caching and Preload</td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>72</td>
</tr>
<tr>
<td>(write)</td>
<td>358</td>
</tr>
<tr>
<td>Intelligent Caching and Background Readahead</td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>77</td>
</tr>
<tr>
<td>(write)</td>
<td>366</td>
</tr>
<tr>
<td>Intelligent Caching, Preload and Background Readahead</td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>72</td>
</tr>
<tr>
<td>(write)</td>
<td>358</td>
</tr>
</tbody>
</table>

<p>| Table 5-24: Simulated Load Data Set B        |
| Factor Analysis Response Times (msec.)        |</p>
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
</tr>
<tr>
<td>LRU</td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>169</td>
</tr>
<tr>
<td>(write)</td>
<td>614</td>
</tr>
<tr>
<td>Intelligent Caching</td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>198</td>
</tr>
<tr>
<td>(write)</td>
<td>884</td>
</tr>
<tr>
<td>Intelligent Caching and Preload</td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>195</td>
</tr>
<tr>
<td>(write)</td>
<td>814</td>
</tr>
<tr>
<td>Intelligent Caching and Background Readahead</td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>198</td>
</tr>
<tr>
<td>(write)</td>
<td>888</td>
</tr>
<tr>
<td>Intelligent Caching, Preload and Background Readahead</td>
<td></td>
</tr>
<tr>
<td>(read)</td>
<td>195</td>
</tr>
<tr>
<td>(write)</td>
<td>814</td>
</tr>
</tbody>
</table>

Table 5-25: Simulated Load Data Set C  
Factor Analysis Response Times (msec.)
Figure 5-20: Factor Analysis Read Transactions
Simulated Load Data Set A
Figure 5-21: Factor Analysis Write Transactions
Simulated Load Data Set A
Figure 5-22: Factor Analysis Read Transactions
Simulated Load Data Set B
Figure 5-23: Factor Analysis Write Transactions
Simulated Load Data Set B
Figure 5-24: Factor Analysis Read Transactions
Simulated Load Data Set C
5.4 Comparison of Simulated Process Access Sequence and Simulated Load Results

Viewed from a cautious frame of mind, only the results from the simulations performed on simulated process access sequence data can be trusted. Nevertheless, conclusions supported by both data sets have extra credibility. Cautionary notes drawn from the simulated load data simulations are worthy of consideration. Performance measures from the simulated load simulations which are at variance with simulated process access sequence simulations are suggestive of performance with high burst rates but by no means conclusive.
The simulated process access sequence data loads the simulations too lightly to assess the true performance implications of intelligent caching. These simulations are representative of intelligent caching behavior under light loads and demonstrate valid performance improvements. The reasons why more data could not be obtained to increase the load have already been discussed. Other alternatives (such as duplicating existing data or generating synthetic loads) were considered but discarded for methodological reasons.

The performance of the simulations with simulated load data show more dramatic performance increases. These performance increases must be viewed with caution — the data driving the simulations has characteristic burst rates, characteristic block distributions and generates high loads but is derived from illegal process behavior. Thus the simulation results are suggestive of performance with high burst rates but not conclusive.

Conservatively, the performance improvements demonstrated in the simulated process access sequence simulations are likely to be realized in an actual implementation of the intelligent algorithms. Optimistically, the sizable performance gains of the simulated load data simulations are potentially realizable. In all likelihood, performance intermediate between the two simulation results would result from an actual implementation.

The performance gains from cache preloading are consistent between simulated process access sequence and simulated load data simulations. In particular, the increases in cache preloading effectiveness with cache size are comparable between simulations.

The detrimental effects of readahead encountered in the simulated load data simulations are noteworthy. There is high likelihood that similar performance decrements would occur in simulated process access sequence data with high loads and longer queue lengths. The finding that readahead may provide overall performance decrements should be considered valid even though it comes from the simulated load data sets.
Chapter 6

Discussion

6.1 Experimental Results

Clearly, intelligent caching as described does provide performance increments over LRU caching strategies — both simulated process access sequence and simulated load data sets support this conclusion. The extent of the performance increment and the cache size at which the increment is expected is not well defined — the caution with which one views the simulated load data simulations determines one's conclusions. A reasonable expectation of intelligent caching performance is some intermediate figure between the simulated process access sequence and simulated load data simulation results.

Intelligent caching works better with larger cache sizes. Even at their largest, cache sizes used in the simulations are reasonable for contemporary file service engines.

The important caution that cache background readahead can have detrimental performance implications is drawn from the simulated load data. Nonetheless, it is felt that such conclusions would be drawn from simulated process access sequence data with sufficiently high burst rates. This is an important finding unreported in the literature.

Cache preloading is demonstrated to be a beneficial to performance. It is important to note that the intelligent algorithm is central to knowing what to preload. Since cache reloading during idle server periods is not employed in this study, the estimates of what can be achieved with preloading/reloading are conservative.
A considerable surprise was the low server overhead of the intelligent algorithms — a cost that was easily amortized by performance increments.

6.2 Evaluation

The author is well content with the validity of the experimental conclusions. There are, however, some methodological nits to be picked with the study. This sections addresses such concerns.

It was majorly unfortunate that midway through the study, circumstances conspired to remove the capability to gather further data. The advent of a major software upgrade on the experimental machine (and all machines of that type) which invalidated the data gathering tool and the need to bring that machine into conformity with a network environment was to have considerable impact on the study. At the time, the decision to abandon data gathering was based upon the excellent results from the simulated load data simulations and the considerable storage costs of gathering further trace data. The decision came to be reconsidered many months later when the issue of illegal process behavior reared its ugly head and the simulated process access sequence data proved inadequate to load the simulations and demonstrate the presumed full performance increments of intelligent caching at representative file server loads. Regrettably, by that time, the decision was irrevocable without considerable delay in the study.

Too little consideration was given to the validation of the simulations at the beginning of the study. This resulted in the absence in the traces gathered of information useful for validation — specifically completion times for disk service operations. Thus the validation simulation was less detailed then it might have been.

The Unix file system uses a powerful set of heuristics to optimize placement of blocks on disk. Had time and effort permitted, it would have been desirable to directly simulate head movement of disks in estimating disk service time.
From the perspective at the conclusion of the study, hindsight suggests that an actual implementation of the intelligent file server might have been possible with comparable effort.

6.3 Further Research

This study demonstrates the feasibility of knowledge based caching algorithms. It falls short of providing full information about the expected performance of an actual intelligent file server. There remain a number of open questions.

- **Extension to a more random environment** – In this study all processes were well behaved, giving all available hints, notifying the server of their termination, etc. In real network environments such refined behavior is not to be expected. Network components and hosts will die without warning. Processes will be aborted by users. Extensions need to be made to the intelligent algorithm to allow timing out of dead processes and freeing up of the cache held by them. In this direction may also lie solutions to the cache holding problems of the intelligent algorithm which hinders it in small cache environments.

- **Evaluation of "fairness"** – The concept of "fairness," that each process has a claim to a fair share of the cache needs to be investigated. The downside of fairness, useless holding of cache by an inactive process has been explored. The beneficial effects have not been explored; in this sense fairness dictates that lightly active processes will not suffer because highly active processes have monopolized cache. Fairness evenly distributes the benefits and the costs of caching to all processes using the file server.

- **Performance in the absence of hints** – It has been claimed that in the absence of hints, intelligent caching degenerates into LRU caching. This is readily evident but has not been tested. What is of interest is the performance of intelligent caching algorithms under a mix of requests – some with hints and some without. It is assumed that performance would be intermediate between intelligent and LRU caching – but it is by no means self evident. Such information would be critical if one were to consider gradual implementation of intelligent file servers with incremental development of hint giving on network hosts.

- **Development of successful readahead strategies** – Readahead fails badly in the current study when burst rates are high. Background readahead finds small time periods when the disk is idle and issues non-preemptable requests while more critical requests queue waiting for the completion of readahead. Preemptable strategies and perhaps burst sensitive strategies need to be investigated.

- **Extension into "unusual" Unix environments** – The current study was performed in a Unix environment where program development and text processing form the normal load. It is not clear that other Unix environments where other workloads predominate (e.g., database and lisp applications) would provide the same
predictability for the knowledge based algorithms to work with. This should be tested.

6.4 Conclusions

Bridging knowledge from high level abstractions in the operating system to low level operations has demonstrable performance implications. Using high level knowledge to organize simple but costly operations works — specifically, knowledge about file access patterns and usages can dramatically boost caching performance.

Knowledge about which information in a cache is likely to be of general utility allows this information to be preloaded (and reloaded during periods of inactivity) into the cache.

Knowledge about file access patterns allows appropriate choices of caching strategies. Truly surprising amounts of information need not be cached and that which requires caching is sometimes better handled by MRU than LRU strategies.

Surprisingly, knowing what to read next in a sequential file doesn’t help much. Background readahead offered little performance gain and sometimes performance decrements. This would seem to be a case of too little knowledge being a dangerous thing. It is known that a given block will be needed, but not how important that block is relative to other requests that may arrive while that background request is being processed.

Dynamic partitioning of cache on a per process basis allows mapping of knowledge about process behavior to cache used by the process. This partitioning isolates the damage a process can do to cache functioning (i.e., an active process can’t lock other less active processes out of the cache) but allows a process to be foolish in its own management of its cache share. This gives rise to the uninvestigated phenomena of cache fairness. In small cache sizes, the damage a process can do to itself with poor cache management can adversely affect average cache behavior.
The cost of the extra processing required by intelligent cache management is surprisingly small and is readily amortized by the performance gains of intelligent caching.

The bridging of knowledge into file servers about the files being served occasions measurable performance gains and is worthy of further pursuit.
Appendix A.

Trace Data

This appendix describes the fifteen terminal sessions which were logged and which were combined to form the three traces which drove the simulations. The three traces are also characterized.

A.1 Terminal Sessions

The basic characteristics of the terminal sessions are given in table A-1.

For each terminal session, the name by which it is referenced in the next section is given. Start refers to how long after logging of the session was initiated the first disk activity commenced. End refers to the last reported activity of the session. The offset figures are those quantities added to each session time stamp to distribute the sessions through the trace.
<table>
<thead>
<tr>
<th>Number</th>
<th>Session</th>
<th>Start</th>
<th>End</th>
<th>Duration</th>
<th>Offset in Trace File</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ALT1</td>
<td>:36</td>
<td>27:13</td>
<td>0.0</td>
<td>6.0</td>
</tr>
<tr>
<td>2</td>
<td>ALT2</td>
<td>:25</td>
<td>10:17</td>
<td>9:52</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>CAROL1</td>
<td>:50</td>
<td>12:28</td>
<td>11:38</td>
<td>0.0</td>
</tr>
<tr>
<td>4</td>
<td>CAROL2</td>
<td>:35</td>
<td>:51</td>
<td>:16</td>
<td>0.0</td>
</tr>
<tr>
<td>5</td>
<td>HFK1</td>
<td>:59</td>
<td>11:30</td>
<td>10:31</td>
<td>0.0</td>
</tr>
<tr>
<td>6</td>
<td>KMK1</td>
<td>:37</td>
<td>9:07</td>
<td>8:30</td>
<td>0.0</td>
</tr>
<tr>
<td>7</td>
<td>KMK2</td>
<td>:32</td>
<td>7:21</td>
<td>6:49</td>
<td>0.0</td>
</tr>
<tr>
<td>8</td>
<td>KMK3</td>
<td>2:07</td>
<td>11:14</td>
<td>9:07</td>
<td>0.0</td>
</tr>
<tr>
<td>9</td>
<td>KMK4</td>
<td>:23</td>
<td>1:27</td>
<td>1:04</td>
<td>0.0</td>
</tr>
<tr>
<td>10</td>
<td>KMK5</td>
<td>:16</td>
<td>:20</td>
<td>:04</td>
<td>0.0</td>
</tr>
<tr>
<td>11</td>
<td>MM1</td>
<td>:04</td>
<td>5:36</td>
<td>5:32</td>
<td>0.0</td>
</tr>
<tr>
<td>12</td>
<td>MM2</td>
<td>:04</td>
<td>6:39</td>
<td>6:35</td>
<td>0.0</td>
</tr>
<tr>
<td>13</td>
<td>MM3</td>
<td>1:24</td>
<td>57:34</td>
<td>58:10</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>MM3A*</td>
<td>1:24</td>
<td>25:27</td>
<td>24:03</td>
<td>0.0</td>
</tr>
<tr>
<td>15</td>
<td>MM3B*</td>
<td>56:39</td>
<td>57:13</td>
<td>0:34</td>
<td>-29.0</td>
</tr>
<tr>
<td>16</td>
<td>MM4</td>
<td>:25</td>
<td>2:30</td>
<td>2:05</td>
<td>0.0</td>
</tr>
<tr>
<td>17</td>
<td>MM5</td>
<td>4:24</td>
<td>19:52</td>
<td>15:28</td>
<td>0.0</td>
</tr>
</tbody>
</table>

* Sessions 14 and 15 are roughly equal segments of session 13.

They were generated in order to allow trace B to have a shorter overall temporal length.

Table A-1: Terminal Sessions
A.2 Session Activity

The following section details the user activities performed in each terminal session.

Where available, shell histories are given. Where unavailable, execution system calls have been extracted from the session logs to give a feel for user activities.

Session - Alt1

7 mail margo@harvard
8 ls
9 rm #*
10 rm traceoff
11 rm traceon
12 ls -l

Session - Alt2

12 ed mailer.mss

Session - Carol1

28 emacs.gnu 85-10
29 dtbl -PIP 85-10 | ditroff -PIP
30 ipq
31 iprm 86
32 ps

Session - Carol2

10 emacs.gnu 85-10

Session - Hfk1

1948 Call execve ( "/bin/ls", 0x24234, 0x23b20 )
1949 Call execve ( "/bin/ls", 0x24438, 0x23b20 )
1950 Call execve ( "/usr/local/dtbl", 0x23ee4, 0x23b20 )
1951 Call execve ( "/usr/local/deq", 0x24194, 0x23b20 )
1952 Call execve ( "/usr/prof/hfk/bin/ditroff", 0x241f4, 0x23b20 )
1952 Call execve ( "/usr/local/ditroff", 0x241f4, 0x23b20 )
1953 Call execve ( "/usr/local/troff_p", 0x224d0, 0xfffd8 )
1954 Call execve ( "/usr/ucb/lpr", 0x22934, 0xfffd8 )
1955 Call execve ( "/bin/ls", 0x24468, 0x23b20 )
1956 Call execve ( "/usr/prof/hfk/bin/tex", 0x24798, 0x23b20 )
1956 Call execve ( "/usr/local/tex", 0x24798, 0x23b20 )
1958 Call execve ( "/bin/ps", 0x248a0, 0x23b20 )
1961 Call execve ( "/usr/ucb/mail", 0x23424, 0x23b20 )
1965 Call execve ( "/usr/lib/sendmail", 0x232b6, 0xffd00 )
1966 Call execve ( "/usr/ucb/lpq", 0x24ab4, 0x23b20 )
1973 Call execve ("lprm", 0x24c48, 0x23b20 )
1973 Call execve ("/usr/ucb/lprm", 0x24c48, 0x23b20 )
1977 Call execve ("/usr/ucb/lpq", 0x23424, 0x25b20 )
1978 Call execve ("/bin/ls", 0x23438, 0x23b20 )
1979 Call execve ("/bin/ls", 0x24b5c, 0x23b20 )
1980 Call execve ("/usr/ucb/more", 0x23454, 0x23b20 )
1981 Call execve ("/bin/ls", 0x23aa0, 0x23b20 )
1982 Call execve ("/bin/df", 0x2345c, 0x23b20 )
1983 Call execve ("/usr/ucb/rsh", 0x24494, 0x23b20 )

Session - Kmk1

26 l
27 rm talk.pic*
28 gcc showtc.c
29 ty showtc.c
30 cc showtc.c -o showtc -ltermlib
31 showtc
32 mail korner@sally
33 trace -n
34 cd -moll
35 cd -molloy
36 l
37 cd trace
38 l
39 ty trace.8
40 ps

Session - Kmk2

34 l avi
35 l fussell
36 l hfk
37 ty hfk/trace.session1.hfk
38 l
39 cd ..
40 l
41 cd -carol
42 pwd
43 l
44 dir
45 ty history.session1
46 cd
47 h
48 ps

Session - Kmk3

84 cd mazewar/usr
rm *.o
l
make
cd ../daemon
rm *.o
make
cd ../usr
make
ty README
make sun
l
mazewar
cd ../
l
ps

Session - Kmk4

22156 Call execve ("/bin/cc", 0xffe04, 0xffea4 )
22157 Call execve ("/lib/cpp", 0x10e08, 0xffea8 )
22158 Call execve ("/lib/ccom", 0x10e08, 0xffea8 )
22159 Call execve ("/bin/as", 0x10e08, 0xffea8 )
22160 Call execve ("/bin/ld", 0x10e08, 0xffea8 )

Session - Kmk5

17261 Call execve ("/bin/ls", 0xffed4, 0xffee0 )

Session - Mml1

596 Call execve ("make", 0x22c14, 0xfffd8c )
596 Call execve ("/bin/make", 0x22c14, 0xfffd8c )
597 Call execve ("cc", 0xffe5a8, 0xfffd90 )
597 Call execve ("/usr/prof/molloy/bin/cc", 0xffe5a8, 0xfffd90 )
597 Call execve ("/usr/cc", 0xffe5a8, 0xfffd90 )
597 Call execve ("/usr/local/cc", 0xffe5a8, 0xfffd90 )
597 Call execve ("/usr/cc", 0xffe5a8, 0xfffd90 )
597 Call execve ("/usr/ucb/cc", 0xffe5a8, 0xfffd90 )
597 Call execve ("/bin/cc", 0xffe5a8, 0xfffd90 )
598 Call execve ("/lib/cpp", 0x10e08, 0xfffd88 )
599 Call execve ("/lib/ccom", 0x10e08, 0xfffd88 )
600 Call execve ("/bin/as", 0x10e08, 0xfffd88 )
601 Call execve ("cc", 0xffe5a8, 0xfffd90 )
601 Call execve ("/usr/prof/molloy/bin/cc", 0xffe5a8, 0xfffd90 )
601 Call execve ("/bin/cc", 0xffe5a8, 0xfffd90 )
601 Call execve ("/usr/local/cc", 0xffe5a8, 0xfffd90 )
601 Call execve ("/usr/local/cc", 0xffe5a8, 0xfffd90 )
601 Call execve ("/usr/ucb/cc", 0xffe5a8, 0xfffd90 )
601 Call execve ("/bin/cc", 0xffe5a8, 0xfffd90 )
Call execve ("/lib/cpp", Ox10e08, Oxfffd88 )
Call execve ("/lib/com", Ox10e08, Oxfffd88 )
Call execve ("/bin/as", Ox10e08, Oxfffd88 )
Call execve ("cc", Oxffe758, Oxfffd90 )
Call execve ("/usr/prof/molly/bin/cc", Oxffe758, Oxfffd90 )
Call execve ("/bin/cc", Oxffe758, Oxfffd90 )
Call execve ("/usr/local/cc", Oxffe758, Oxfffd90 )
Call execve ("/usr/local/cc", Oxffe758, Oxfffd90 )
Call execve ("/usr/ucb/cc", Oxffe758, Oxfffd90 )
Call execve ("/bin/cc", Oxffe758, Oxfffd90 )
Call execve ("/bin/id", Ox10e08, Oxfffc88 )

Session - Mm2

Call execve ("make", Ox22d60, Oxfffd8c )
Call execve ("/bin/make", Ox22d60, Oxfffd8c )
Call execve ("cc", Oxffe5a8, Oxfffd90 )
Call execve ("/usr/prof/molly/bin/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/bin/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/usr/local/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/usr/local/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/usr/ucb/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/bin/cc", Oxffe5a8, Oxfffd90 )
Call execve ("make", Ox22d60, Oxfffd8c )
Call execve ("/bin/make", Ox22d60, Oxfffd8c )
Call execve ("cc", Oxffe5a8, Oxfffd90 )
Call execve ("/usr/prof/molly/bin/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/bin/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/usr/local/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/usr/local/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/usr/ucb/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/bin/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/bin/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/usr/local/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/usr/local/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/usr/ucb/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/bin/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/bin/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/usr/local/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/usr/local/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/usr/ucb/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/bin/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/usr/prof/molly/bin/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/bin/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/usr/local/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/usr/local/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/usr/ucb/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/bin/cc", Oxffe5a8, Oxfffd90 )
Call execve ("/usr/prof/molly/bin/cc", Oxffe5a8, Oxfffd90 )
628 Call execve ( "/usr/local/cc", 0xffe5a8, 0xffffd90 )
628 Call execve ( "/usr/local/cc", 0xffe5a8, 0xffffd90 )
628 Call execve ( "/usr/ucb/cc", 0xffe5a8, 0xffffd90 )
629 Call execve ( "/bin/cc", 0xffe5a8, 0xffffd90 )
630 Call execve ( "/lib/cpp", 0x10e08, 0xffffd84 )
631 Call execve ( "/bin/as", 0x10e08, 0xffffd84 )
632 Call execve ( "/cc", 0xffe5a8, 0xffffd90 )
632 Call execve ( "/usr/prof/molloy/bin/cc", 0xffe5a8, 0xffffd90 )
632 Call execve ( "/bin/cc", 0xffe5a8, 0xffffd90 )
632 Call execve ( "/usr/local/cc", 0xffe5a8, 0xffffd90 )
632 Call execve ( "/usr/ucb/cc", 0xffe5a8, 0xffffd90 )
633 Call execve ( "/bin/cc", 0xffe5a8, 0xffffd90 )
634 Call execve ( "/lib/cpp", 0x10e08, 0xffffd88 )
637 Call execve ( "/bin/as", 0x10e08, 0xffffd88 )
638 Call execve ( "/cc", 0xffe5a8, 0xffffd90 )
638 Call execve ( "/usr/prof/molloy/bin/cc", 0xffe5a8, 0xffffd90 )
638 Call execve ( "/bin/cc", 0xffe5a8, 0xffffd90 )
638 Call execve ( "/usr/local/cc", 0xffe5a8, 0xffffd90 )
638 Call execve ( "/usr/local/cc", 0xffe5a8, 0xffffd90 )
638 Call execve ( "/usr/ucb/cc", 0xffe5a8, 0xffffd90 )
639 Call execve ( "/bin/cc", 0xffe5a8, 0xffffd90 )
640 Call execve ( "/lib/cpp", 0x10e08, 0xffffd84 )
641 Call execve ( "/bin/as", 0x10e08, 0xffffd84 )
642 Call execve ( "/cc", 0xffe5a8, 0xffffd90 )
642 Call execve ( "/usr/prof/molloy/bin/cc", 0xffe5a8, 0xffffd90 )
642 Call execve ( "/bin/cc", 0xffe5a8, 0xffffd90 )
642 Call execve ( "/usr/local/cc", 0xffe5a8, 0xffffd90 )
642 Call execve ( "/usr/local/cc", 0xffe5a8, 0xffffd90 )
642 Call execve ( "/usr/ucb/cc", 0xffe5a8, 0xffffd90 )
643 Call execve ( "/bin/cc", 0xffe5a8, 0xffffd90 )
644 Call execve ( "/lib/cpp", 0x10e08, 0xffffd84 )
645 Call execve ( "/bin/as", 0x10e08, 0xffffd84 )
646 Call execve ( "/cc", 0xffe5a8, 0xffffd90 )
646 Call execve ( "/usr/prof/molloy/bin/cc", 0xffe5a8, 0xffffd90 )
646 Call execve ( "/bin/cc", 0xffe5a8, 0xffffd90 )
646 Call execve ( "/usr/local/cc", 0xffe5a8, 0xffffd90 )
646 Call execve ( "/usr/local/cc", 0xffe5a8, 0xffffd90 )
646 Call execve ( "/usr/ucb/cc", 0xffe5a8, 0xffffd90 )
646 Call execve ( "/bin/cc", 0xffe5a8, 0xffffd90 )
647 Call execve ( "/lib/cpp", 0x10e08, 0xffffd88 )
648 Call execve ( "/lib/cpp", 0x10e08, 0xffffd88 )
649 Call execve ( "/bin/as", 0x10e08, 0xffffd88 )
650 Call execve ( "/cc", 0xffe5a8, 0xffffd90 )
650 Call execve ( "/usr/prof/molloy/bin/cc", 0xffe5a8, 0xffffd90 )
650 Call execve ("/bin/cc", 0xffe5a8, 0xfffd90 )
650 Call execve ("/usr/local/cc", 0xffe5a8, 0xfffd90 )
650 Call execve ("/usr/local/cc", 0xffe5a8, 0xfffd90 )
650 Call execve ("/usr/ucb/cc", 0xffe5a8, 0xfffd90 )
650 Call execve ("/bin/cc", 0xffe5a8, 0xfffd90 )
651 Call execve ("/lib/cpp", 0x10e08, 0xfffd84 )
652 Call execve ("/lib/ccom", 0x10e08, 0xfffd84 )
653 Call execve ("/bin/as", 0x10e08, 0xfffd84 )
654 Call execve ("cc", 0xffe5a8, 0xfffd90 )
654 Call execve ("/usr/prof/molloy/bin/cc", 0xffe5a8, 0xfffd90 )
654 Call execve ("/bin/cc", 0xffe5a8, 0xfffd90 )
654 Call execve ("/usr/local/cc", 0xffe5a8, 0xfffd90 )
654 Call execve ("/usr/local/cc", 0xffe5a8, 0xfffd90 )
654 Call execve ("/usr/ucb/cc", 0xffe5a8, 0xfffd90 )
654 Call execve ("/bin/cc", 0xffe5a8, 0xfffd90 )
655 Call execve ("/lib/cpp", 0x10e08, 0xfffd88 )
656 Call execve ("/lib/ccom", 0x10e08, 0xfffd88 )
657 Call execve ("/bin/as", 0x10e08, 0xfffd88 )

Session - Mm3

6280 Call execve ("vi", 0x22eac, 0xfffd8c )
6280 Call execve ("/usr/ucb/vi", 0x22eac, 0xfffd8c )
6349 Call execve ("ps", 0x22ea4, 0xfffd8c )
6349 Call execve ("/bin/ps", 0x22ea4, 0xfffd8c )

Session - Mm4

591 Call execve ("vi", 0x2306c, 0xfffd8c )
591 Call execve ("/usr/ucb/vi", 0x2306c, 0xfffd8c )

Session - Mm5

733 Call execve ("vi", 0x23100, 0xfffd8c )
733 Call execve ("/usr/ucb/vi", 0x23100, 0xfffd8c )

A.3 Trace Composition

A.3.1 Terminal Sessions and Traces

Figures A-1 to A-3 illustrate the manner in which the terminal sessions were combined to produce the three initial simulated load composite traces. The line for each session is the plotted value of start plus offset to end plus offset. These values are taken directly from table A-1.
Figure A-1: Trace Composition
Trace A
Terminal Session

Figure A-2: Trace Composition
Trace B
Figure A-3: Trace Composition  
Trace C

A.3.2 Trace Activity Over Time

Figures A-4 to A-6 indicate the distribution of requests in each initial simulated load composite trace over time. Each point in the graph represents the number of service requests in the previous ten seconds for that trace. The data are presented for simulated load data sets -- simulated process activity sequenced data plots are quite similar to the simulated load plots.
Figure A-4: Trace Activity
Trace A
Figure A-5: Trace Activity
Trace B
A.3.3 Trace Activity Type

Each trace is generated from the same set of terminal sessions. Each trace differs from the other traces only in the distribution of the sessions in time. Hence, each trace contains identical service demands. Table A-2 contains a breakdown of inode requests by type. Table A-3 contains a breakdown of block requests by type.
<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Read Requests</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System</td>
<td>6779</td>
<td>187</td>
</tr>
<tr>
<td>User</td>
<td>1193</td>
<td>342</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>7972</td>
<td>491</td>
</tr>
<tr>
<td><strong>Write Requests</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System</td>
<td>2623</td>
<td>195</td>
</tr>
<tr>
<td></td>
<td>1376</td>
<td>362</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>3999</td>
<td>465</td>
</tr>
</tbody>
</table>

**Total Distinct Inodes - 499**

Table A-2: Inode Activity Types
### Read Requests

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pagein</td>
<td>2672</td>
<td>888</td>
</tr>
<tr>
<td>No Cache</td>
<td>2444</td>
<td>366</td>
</tr>
<tr>
<td>LRU</td>
<td>4773</td>
<td>347</td>
</tr>
<tr>
<td>MRU</td>
<td>3332</td>
<td>154</td>
</tr>
<tr>
<td>Directory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System</td>
<td>10815</td>
<td>154</td>
</tr>
<tr>
<td>User</td>
<td>3118</td>
<td>109</td>
</tr>
<tr>
<td>Total Directory</td>
<td>13933</td>
<td>214</td>
</tr>
</tbody>
</table>

### Write Requests

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pageout</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>No Cache</td>
<td>2115</td>
<td>102</td>
</tr>
<tr>
<td>LRU</td>
<td>1333</td>
<td>102</td>
</tr>
<tr>
<td>MRU</td>
<td>1398</td>
<td>61</td>
</tr>
<tr>
<td>Directory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System</td>
<td>3042</td>
<td>136</td>
</tr>
<tr>
<td>User</td>
<td>800</td>
<td>93</td>
</tr>
<tr>
<td>Total Directory</td>
<td>3842</td>
<td>175</td>
</tr>
</tbody>
</table>

**Total Distinct Blocks - 1897**

Table A-3: Block Activity Types
Appendix B.

Simulations

B.1 Trace Driver

The trace driver makes extensive use of the flexibility and extensibility of PAWS. The basic task of the trace driver is to introduce into the simulation system transactions of the type and at the time indicated by a file of trace information.

The format of the trace file is given in Table B-1.
<time> <nread> <nwrite> <nhint>
     <nread> <read transactions>
     ...
     <nwrite> <write transaction>s
     ...
     <nhint> <hint transaction>s

where

<read transaction> = one of
     r b <pn> <bn> <nb> <fg> <cd> <na>    ;read block
     r i <pn> <in> <na>                  ;read inode
<write transaction> = one of
     w b <pn> <bn> <fg> <cd> <na>       ;write block
     w i <pn> <in> <na>                 ;write inode
<hint transaction> = one of
     h z p <pn>                        ;zap process
     h t <opn> <nnpn> <fg>             ;transfer file group

<bn> = device/block number
<cd> = cache discipline
<fg> = file group number
<in> = device/inode number
<na> = time of next access for block/inode
<nb> = next device/block number
<nhint> = number of hint transactions to be entered at time <time>
<nnpn> = destination process
<nread> = number of read transactions to be entered at time <time>
<nwrite> = number of write transactions to be entered at time <time>
<opn> = source process
<pn> = process number
<time> = time at which to enter transactions into system

Table B-1: Trace File Format

The flow of transactions between the trace driver simulations is diagramed in figure B-1. Transaction phase numbers are marked on the edges connecting nodes.
Figure B-1: Trace Driver Nodes and Interconnections

The trace driver is initialized with a transaction of each type (read, write, hint) at node Source1. At node Comp1, the first local integer of each transaction is set to reflect the category of the transaction — this allows user supplied subroutines (USS) access to transaction category information. These three prototype transactions retain phase 100 throughout the simulation. The transactions go to a user supplied subroutine (USS) module which reads from the trace file, determines the next time that transactions will be entered into the simulation and puts the three prototype transactions in a queue (Delay1) until that time. When that time occurs in the simulation, the prototype transactions are returned to the USS which determines how many transactions to split off of each prototype and the transactions are split
(at node Split1) into the number of transactions required at that time. The three originals returned to Delay1 to be enqueued until the next activity period. The new split transactions (with phase 2) are routed to another USS which reads the transaction data from the trace file and inserts it into the simulation transaction's local data area. Filled transactions depart the USS with phase 3 for the nodes which perform simulated process access sequencing. All of these operations take zero simulation time. Thus the filled transactions join the simulation from the second USS at the desired time.

B.2 Simulated Process Access Sequencing

The simulation nodes which accomplish process access sequencing are diagramed in figure B-2. Edges which connect nodes are annotated with transaction phases. All transactions enter node Legal? from the trace driver nodes with phase 3. Legal? is a USS which checks to see if the process which submitted the transaction has a transaction currently active in the simulation. If so, it is delayed at Legal? until the currently active transaction clears node Legalout. If the process does not have any currently active transactions, Legal? either passes the transaction to node Logit or, if previous transactions of the requesting process have been delayed, delays the transaction at node Wait for a period equal to the sum of all delays. Transactions arriving at node Logit are ready to enter the simulation and are recorded in the new simulated process access sequence composite trace log; they then pass on to node Enter1 and join the file server simulation.

Transactions departing from the simulation leave node Exit1 and arrive at node Legalout. At Legalout, the next request from the process of the departing transaction (if any) is released from Legal? to node Wait. The released transaction waits for an interval described below. In addition, Legalout checks to see if the composite trace file is at end of file and and the current transaction is the last remaining one in the system: if so, a flag in the local data area of the transaction is set. At node Endit, this flag is checked and if set the simulation is terminated.
Figure B-2: Process Access Sequencing Nodes and Interconnections

Process access sequencing is performed in such a manner as to leave inter burst intervals relatively unchanged. Figure B-3 illustrates the principles used. Process access sequencing is necessary whenever a transaction from a process has not exited the simulation before the process submits another request. The second request must be delayed until the first request completes. Following this first delay, all transactions from the process must be offset
by this delay. In addition, all future delays must be added to the offset. Delays for transactions are selected such that the interval between two transactions (c) is the maximum of their original interarrival interval (a) and the time between arrival and completion of the first of the two transactions (b).

\[ c = \max(a, b) \]

Figure B-3: Process Access Sequenced Transaction Delays

B.3 Simulation Parameters

Parameters of the simulation models are given in table B-2.
<table>
<thead>
<tr>
<th>Model</th>
<th>LRU</th>
<th>Optimal</th>
<th>Intelligent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Decode</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Network Encode</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>Cache Check</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Cache Update</td>
<td>10.0</td>
<td>10.0</td>
<td>10.0</td>
</tr>
<tr>
<td>Hint Overhead</td>
<td></td>
<td></td>
<td>0.5</td>
</tr>
<tr>
<td>Disk Service</td>
<td>20.0</td>
<td>20.0</td>
<td>20.0</td>
</tr>
</tbody>
</table>

Table B-2: Simulation Parameters (msec.)

### B.4 Simulation Models

#### B.4.1 LRU Model

```
DECLARE
  REAL  NETIN, CACHE1, CACHE2, CACHE3, NETOUT,
       DSKSERV;
!
  Trace Driver Nodes
  NODES  SOURCE1, DELAY1, COMP1, USS, SINK1,
         SPLIT1, LEGAL, WAIT, LOGIT, ENTER1,
         EXIT1, LEGALOUT, ENDIT
!
  Simulation Nodes
  CPU, INCACHE, DISK, CACHEIT,
  COMP2;

CATEGORIES
  READ, WRITE, HINT;

TOPOLOGY
  SOURCE1  COMP1 <ALL, ALL> 1.0;
  COMP1    USS <ALL, ALL> 1.0;
  DELAY1   USS <ALL, ALL> 1.0;
  USS      SPLIT1 <ALL, 100> 1.0;
  SPLIT1   USS <ALL, 2> 1.0;
  SPLIT1   DELAY1 <ALL, 100> 1.0;
  USS      LEGAL <ALL, 3> 1.0;
  LEGAL    LOGIT <ALL, 3> 1.0;
  LEGAL    WAIT <ALL, 2> 1.0;
```
WAIT LOGIT <ALL, 2> 1.0;
LOGIT ENTER1 <ALL, ALL> 1.0;
! All transactions join the simulation at ENTER1
! Sink all hints for iru and optimal
ENTER1 EXIT1 <HINT, ALL> 1.0;
ENTER1 CPU <READ, ALL> 1.0;
ENTER1 CPU <WRITE, ALL> 1.0;
! Read path
CPU INCACHE <READ, 3> 1.0; ! Decode network transmission
INCACHE CPU <READ, 4> 1.0; ! In the cache?
INCACHE CPU <READ, 5> 1.0; ! Yes
CPU DISK <READ, 4> 1.0; ! Need to read
CPU CACHEIT <READ, 5> 1.0; ! Cache check CPU overhead
DISK CACHEIT <READ, 4> 1.0; ! Update cache
CACHEIT CPU <READ, 6> 1.0; ! Update + network encoding
CACHEIT CPU <READ, 7> 1.0; ! Network encoding
CPU EXIT1 <READ, 6> 1.0; ! Not found in cache
CPU EXIT1 <READ, 7> 1.0; ! Found in cache
! Write path
CPU CACHEIT <WRITE, 3> 1.0; ! Update cache
CACHEIT CPU <WRITE, 4> 1.0;
CPU DISK <WRITE, 4> 1.0; ! Update disk
DISK COMP2 <WRITE, 4> 1.0;
COMP2 CPU <WRITE, 6> 1.0;
CPU EXIT1 <WRITE, 6> 1.0;
! Exit stage left
EXIT1 LEGALOUT <ALL,ALL> 1.0;
LEGALOUT ENDT <ALL,ALL> 1.0;
ENDT SINK1 <ALL,ALL> 1.0;
DEFINE
! Trace Driver Nodes
SOURCE1
    TYPE SERVICE
    QUANTITY 1
    QD DELAY
    REQUEST
        <ALL, ALL> CONSTANT (0.0);
DELAY1
    TYPE SERVICE
    QUANTITY 1
    QD DELAY
    REQUEST
        <ALL, ALL> CONSTANT (LR[1]);
COMP1
    TYPE COMPUTE
    REQUEST
        <READ, ALL> LETEQ LI[1] 1
<WRITE, ALL> LESEQ LI[1] 2
<HINT, ALL> LESEQ LI[1] 3;

US5
   TYPE USER
   REQUEST  
      <ALL, 100> 1
      <ALL, 2>  2;

SINK1
   TYPE SINK;

SPLIT1
   TYPE SPLIT
   REQUEST  
      <READ, 100> CONSTANT (LI[2]) <READ, 2>
      <WRITE, 100> CONSTANT (LI[2]) <WRITE, 2>
      <HINT, 100> CONSTANT (LI[2]) <HINT, 2>;

LEGAL
   TYPE USER
   REQUEST <ALL, ALL> 10;

WAIT
   TYPE SERVICE
   QUANTITY 1
   QD DELAY
   REQUEST  
      <ALL, ALL> CONSTANT (LR[5]);

LOGIT
   TYPE USER
   REQUEST <ALL, ALL> 11;

ENTER1
   TYPE BRANCH;

EXIT1
   TYPE BRANCH;

LEGALOUT
   TYPE USER
   REQUEST <ALL, ALL> 12;

ENDIT
   TYPE COMPUTE
   REQUEST  
      <ALL, ALL>
      GOTOIF 1 LB[1]
      EXIT
      LABEL 1
      PAUSE;

! Simulation nodes
CPU
   TYPE SERVICE
   QUANTITY 1
   QD PS
REQUEST
  <READ, 3> CONSTANT (NETIN) ! Network decoding
  <READ, 4> CONSTANT (CACHE1) ! Cache check cost
  <READ, 5> CONSTANT (CACHE1) ! Cache check cost
  <READ, 6> CONSTANT (CACHE3) ! Cache update + network encode
  <READ, 7> CONSTANT (NETOUT) ! Network encode
  <WRITE, 3> CONSTANT (NETIN) ! Network decoding
  <WRITE, 4> CONSTANT (CACHE2) ! Cache update
  <WRITE, 6> CONSTANT (NETOUT); ! Network encoding

INCACHE
  TYPE USER
  REQUEST <ALL, ALL> 3; ! 3 if LRU, 5 if optimal

DISK
  TYPE SERVICE
  QUANTITY 1
  QD FCFS
  REQUEST
    <ALL, ALL> CONSTANT (DSKSERV);

CACHEIT
  TYPE USER
  REQUEST <ALL, ALL> 4; ! 4 if LRU, 6 if optimal

COMP2
  TYPE CHANGE
  REQUEST
    <WRITE, 4> 6 1.0;

! End Simulation Nodes

INITIAL
  POPULATION ! Setup for the trace driver
    <READ, 100> 1 SOURCE1
    <WRITE, 100> 1 SOURCE1
    <HINT, 100> 1 SOURCE1;

STATISTICS REPORT
  RESPONSE ENTER1 EXIT1 GRAN 100.0;

RUN
  LETEQ NETIN 1.0 ! Network decoding cpu cost
  LETEQ NETOUT 2.0 ! Network encoding cpu cost
  LETEQ CACHE1 1.0 ! Cache check cpu cost
  LETEQ CACHE2 10.0 ! Cache update cpu cost
  ADD CACHE3 CACHE2 NETOUT ! Cache update + network encode cpu cost
  LETEQ DSKSERV 20.0 ! Disk service cost
    GO 21600000.0 6000000.0 ! Run for 6 hours, 10 min. notify
    DUMP
    EXIT;

END;
B.4.2 Optimal Model

DECLARE
REAL NETIN, CACHE1, CACHE2, CACHE3, NETOUT,
DSDKSERV;
!
Trace Driver Nodes
NODES SOURCE1, DELAY1, COMP1, USS, SINK1,
SPLIT1, LEGAL, WAIT, LOGIT, ENTER1,
EXIT1, LEGALOUT, ENDIT
!
Simulation Nodes
CPU, INCACHE, DISK, CACHEIT,
FREEIT, COMP2;
CATEGORIES
READ, WRITE, HINT;

TOPOLOGY
SOURCE1 COMP1 <ALL, ALL> 1.0;
COMP1 USS <ALL, ALL> 1.0;
DELAY1 USS <ALL, ALL> 1.0;
USS SPLIT1 <ALL, 100> 1.0;
SPLIT1 USS <ALL, 2> 1.0;
SPLIT1 DELAY1 <ALL, 100> 1.0;
USS LEGAL <ALL, 3> 1.0;
LEGAL LOGIT <ALL, 3> 1.0;
LEGAL WAIT <ALL, 2> 1.0;
WAIT LOGIT <ALL, 2> 1.0;
LOGIT ENTER1 <ALL, ALL> 1.0;

All transactions join the simulation at ENTER1
!
Sink all hints for lru and optimal
ENTER1 EXIT1 <HINT, ALL> 1.0;
ENTER1 INCACHE <READ, ALL> 1.0;
ENTER1 CACHEIT <WRITE, ALL> 1.0;

! Read path
INCACHE CACHEIT <READ, ALL> 1.0; ! In the cache?
CACHEIT CPU <READ, 7> 1.0; !Update cache
CACHEIT DISK <READ, 6> 1.0; ! Need to read
DISK FREEIT <READ, 6> 1.0; ! Free pending
FREEIT CPU <READ, 6> 1.0; ! Update cache
CPU COMP2 <READ, 7> 1.0; ! Cache check CPU overhead
CPU COMP2 <READ, 6> 1.0; ! Cache check & update CPU overhead
COMP2 CPU <READ, 8> 1.0;
CPU EXIT1 <READ, 8> 1.0;

! Write path
CACHEIT CPU <WRITE, 4> 1.0; ! Update cache
CPU DISK <WRITE, 4> 1.0; ! Update disk
DISK COMP2 <WRITE, 4> 1.0;
COMP2 CPU <WRITE, 6> 1.0;
CPU EXIT1 <WRITE, 6> 1.0;
! Exit stage left

EXIT1 LEGALOUT <ALL,ALL> 1.0;
LEGALOUT ENDIT <ALL,ALL> 1.0;
ENDIT SINK1 <ALL,ALL> 1.0;

DEFINE

! Trace Driver Nodes

SOURCE1
  TYPE SERVICE
  QUANTITY 1
  QD DELAY
  REQUEST
    <ALL, ALL> CONSTANT (0.0);

DELAY1
  TYPE SERVICE
  QUANTITY 1
  QD DELAY
  REQUEST
    <ALL, ALL> CONSTANT (LR[1]);

COMP1
  TYPE COMPUTE
  REQUEST
    <READ, ALL> LEPEQ LI[1] 1
    <WRITE, ALL> LEPEQ LI[1] 2
    <HINT, ALL> LEPEQ LI[1] 3;

USS
  TYPE USER
  REQUEST
    <ALL, 100> 1
    <ALL, 2> 2;

SINK1
  TYPE SINK;

SPLIT1
  TYPE SPLIT
  REQUEST
    <READ, 100> CONSTANT (LI[2]) <READ, 2>
    <WRITE, 100> CONSTANT (LI[2]) <WRITE, 2>
    <HINT, 100> CONSTANT (LI[2]) <HINT, 2>;

LEGAL
  TYPE USER
  REQUEST <ALL, ALL> 10;

WAIT
  TYPE SERVICE
  QUANTITY 1
  QD DELAY
  REQUEST
    <ALL, ALL> CONSTANT (LR[5]);

LOGIT
TYPE USER
REQUEST <ALL, ALL> 11;
ENTER1
TYPE BRANCH;
EXIT1
TYPE BRANCH;
LEGALOUT
TYPE USER
REQUEST <ALL, ALL> 12;
ENDIT
TYPE COMPUTE
REQUEST
<ALL, ALL>
GOTOIF 1 LB[1]
EXIT
LABEL 1
PAUSE;

! Simulation nodes
CPU
TYPE SERVICE
QUANTITY 1
QD PS
REQUEST
<READ, 6> CONSTANT (CACHE1) ! Decode + check + update
<READ, 7> CONSTANT (CACHE2) ! Decode + check
<READ, 8> CONSTANT (NETOUT) ! Decode + check
<WRITE, 4> CONSTANT (CACHE3) ! Cache update
<WRITE, 6> CONSTANT (NETOUT); ! Network encoding
 INCACHE
TYPE USER
REQUEST <ALL, ALL> 5; ! 3 if LRU, 5 if optimal
DISK
TYPE SERVICE
QUANTITY 1
QD FCFS
REQUEST
<ALL, ALL> CONSTANT (DSKSERV);
CACHEIT
TYPE USER
REQUEST <ALL, ALL> 6; ! 4 if LRU, 6 if optimal
FREEIT
TYPE USER
REQUEST <ALL, ALL> 13; ! Free pending requests after disk read
COMP2
TYPE CHANGE
REQUEST
<READ, 6> 8 1.0
<READ, 7> 8 1.0
<WRITE, 4> 6 1.0;
!
End Simulation Nodes

INITIAL
    POPULATION ! Setup for the trace driver
        <READ, 100> 1 SOURCE1
        <WRITE, 100> 1 SOURCE1
        <HINT, 100> 1 SOURCE1;

STATISTICS
    REPORT
        RESPONSE ENTER1 EXIT1 GRAN 15.0;

RUN
    LETEQ NETIN 1.0 ! Network decoding cpu cost
    LETEQ NETOUT 2.0 ! Network encoding cpu cost
    LETEQ CACHE1 12.0 ! Cache check cpu cost (1.0) + netin (1.0) + update (10.0)
    LETEQ CACHE2 2.0 ! Cache check cpu cost (1.0) + netin (1.0)
    LETEQ CACHE3 11.0 ! Cache update (10.0) + netin (1.0)
    LETEQ DSKSERV 20.0 ! Disk service cost
        GO 21800000.0 600000.0 ! Run for 6 hours, 10 min. notify
        DUMP
        EXIT;

END;

B.4.3 Intelligent Model

DECLARE
    REAL NETIN, CACHE1, CACHE2, CACHE3, NETOUT,
        HINTIN, DSKSERV;
!
    Trace Driver Nodes
    NODES SOURCE1, DELAY1, COMP1, USS, SINK1,
        SPLIT1, LEGAL, WAIT, LOGIT, ENTER1,
        EXIT1, LEGALOUT, ENDIT
!
    Simulation Nodes
    CPU, INCACHE, DISK, CACHEIT,
    COMP2, RAHEAD;

CATEGORIES
    READ, WRITE, HINT, BACKGROUND;

TOPOLOGY
    SOURCE1 COMP1 <READ, ALL> 1.0;
    SOURCE1 COMP1 <WRITE, ALL> 1.0;
    SOURCE1 COMP1 <HINT, ALL> 1.0;
    SOURCE1 RAHEAD <BACKGROUND, ALL> 1.0;
    COMP1 USS <ALL, ALL> 1.0;
    DELAY1 USS <ALL, ALL> 1.0;
    USS SPLIT1 <ALL, 100> 1.0;
    SPLIT1 USS <ALL, 2> 1.0;
    SPLIT1 DELAY1 <ALL, 100> 1.0;
    USS LEGAL <ALL, 3> 1.0;
    LEGAL LOGIT <ALL, 3> 1.0;
LEGAL  WAIT <ALL, 2> 1.0;
WAIT  LOGIT <ALL, 2> 1.0;
LOGIT  ENTER1 <ALL, ALL> 1.0;
!
All transactions join the simulation at ENTER1
!
Sink all hints for lru and optimal
ENTER1  CPU <HINT, ALL> 1.0;
ENTER1  CPU <READ, ALL> 1.0;
ENTER1  CPU <WRITE, ALL> 1.0;
!
Hint path
CPU  INCACHE <HINT, ALL> 1.0;
INCACHE  EXIT1 <HINT, ALL> 1.0;
!
Read path
CPU  INCACHE <READ, 3> 1.0;  ! Decode network transmission
INCACHE  CPU <READ, 4> 1.0;  ! In the cache?
INCACHE  CPU <READ, 5> 1.0;  ! Yes
CPU  DISK <READ, 4> 1.0;  ! Need to read
CPU  CACHEIT <READ, 5> 1.0;  ! Cache check CPU overhead
DISK  CACHEIT <READ, 4> 1.0;  ! Update cache
CACHEIT  CPU <READ, 6> 1.0;  ! Update + network encoding
CACHEIT  CPU <READ, 7> 1.0;  ! Network encoding
CPU  EXIT1 <READ, 6> 1.0;  ! Not found in cache
CPU  EXIT1 <READ, 7> 1.0;  ! Found in cache
!
Write path
CPU  CACHEIT <WRITE, 3> 1.0;  ! Update cache
CACHEIT  CPU <WRITE, 4> 1.0;
CPU  DISK <WRITE, 4> 1.0;  ! Update disk
DISK  COMP2 <WRITE, 4> 1.0;
COMP2  CPU <WRITE, 6> 1.0;
CPU  EXIT1 <WRITE, 6> 1.0;
!
Read ahead path
RAHEAD  DISK <BACKGROUND, ALL> 1.0;
DISK  RAHEAD <BACKGROUND, ALL> 1.0;
!
Exit stage left
EXIT1  LEGALOUT <ALL, ALL> 1.0;
LEGALOUT  ENDIT <ALL, ALL> 1.0;
ENDIT  SINK1 <ALL, ALL> 1.0;
DEFINE
!
Trace Driver Nodes
SOURCE1
  TYPE SERVICE
  QUANTITY 1
  QD DELAY
  REQUEST
    <ALL, ALL> CONSTANT (0.0);
DELAY1
  TYPE SERVICE
  QUANTITY 1
QD DELAY
REQUEST
<ALL, ALL> CONSTANT (LR[1]);

COMP1
TYPE COMPUTE
REQUEST
<READ, ALL> LETEQ LI[1] 1
<WRITE, ALL> LETEQ LI[1] 2
<HINT, ALL> LETEQ LI[1] 3;

USS
TYPE USER
REQUEST
<ALL, 100> 1
<ALL, 2> 2;

SINK1
TYPE SINK;

SPLIT1
TYPE SPLIT
REQUEST
<READ, 100> CONSTANT (LI[2]) <READ, 2>
<WRITE, 100> CONSTANT (LI[2]) <WRITE, 2>
<HINT, 100> CONSTANT (LI[2]) <HINT, 2>;

LEGAL
TYPE USER
REQUEST <ALL, ALL> 10;

WAIT
TYPE SERVICE
QUANTITY 1
QD DELAY
REQUEST
<ALL, ALL> CONSTANT (LR[5]);

LOGIT
TYPE USER
REQUEST <ALL, ALL> 11;

ENTER1
TYPE BRANCH;

EXIT1
TYPE BRANCH;

LEGALOUT
TYPE USER
REQUEST <ALL, ALL> 12;

ENDIT
TYPE COMPUTE
REQUEST
<ALL, ALL>
GOTOIF 1 LB[1]
EXIT
LABEL 1
PAUSE;

! Simulation nodes
CPU
TYPE SERVICE
QUANTITY 1
QD PS
REQUEST
  <HINT, ALL> CONSTANT (HINTIN) ! Network decode + CPU cost
  <READ, 3> CONSTANT (NETIN) ! Network decoding
  <READ, 4> CONSTANT (CACHE1) ! Cache check cost
  <READ, 5> CONSTANT (CACHE1) ! Cache check cost
  <READ, 6> CONSTANT (CACHE3) ! Cache update + net encode
  <READ, 7> CONSTANT (NETOUT) ! Network encode
  <WRITE, 3> CONSTANT (NETIN) ! Network decoding
  <WRITE, 4> CONSTANT (CACHE2) ! Cache update
  <WRITE, 6> CONSTANT (NETOUT); ! Network encoding

INCACHE
TYPE USER
REQUEST <ALL, ALL> 7; ! 3 if LRU, 5 if optimal, 7 if intelligent

RAHEAD
TYPE USER
REQUEST <ALL, ALL> 9;

DISK
TYPE SERVICE
QUANTITY 1
QD PRIORITY
REQUEST
  <READ, ALL> 1.0 CONSTANT (DSKSERV)
  <WRITE, ALL> 1.0 CONSTANT (DSKSERV)
  <BACKGROUND, ALL> 2.0 CONSTANT (DSKSERV);

CACHE1
TYPE USER
REQUEST <ALL, ALL> 8; ! 4 if LRU, 6 if optimal, 8 if intelligent

COMP2
TYPE CHANGE
REQUEST
  <WRITE, 4> 6 1.0;

! End Simulation Nodes
INITIAL POPULATION ! Setup for the trace driver
  <READ, 100> 1 SOURCE1
  <WRITE, 100> 1 SOURCE1
  <HINT, 100> 1 SOURCE1
  <BACKGROUND, 1> 1 SOURCE1;

STATISTICS REPORT
  RESPONSE ENTER1 EXIT1 GRAN 15.0;

RUN
LETEQ NETIN 1.0  ! Network decoding cpu cost
LETEQ NETOUT 2.0  ! Network encoding cpu cost
LETEQ CACHE1 1.0  ! Cache check cpu cost
LETEQ CACHE2 10.0  ! Cache update cpu cost
ADD CACHE3 CACHE2 NETOUT  ! Cache update + network encode cpu cost
ADD HINTIN NETIN 0.5  ! Network decoding cpu cost + update cost
LETEQ DSKSERV 20.0  ! Disk service cost
                     GO 21600000.0 600000.0  ! Run for 6 hours, 10 min. notify
                     DUMP
                     EXIT;

END;
Bibliography

[Alanko 80] Alanko, T. O., Haikala, I. J. and Kutvonen, P. H.
Methodology and Empirical Results of Program Behavior Measurements.
In Performance Evaluation Review, pages 55-68. ACM Sigmetrics, ACM,
May, 1980.

[Arden 85] Arden, M.J.
DFS92s: A Distributed File System in a Workstation/LAN Environment.
Technical Report UCB/CSD 85/236, University of California - Berkeley,

[Birrell 80] Birrell, A.D. and Needham, R.M.
A Universal File Server.
IEEE Transactions on Software Engineering SE-6(5):450-453, September,
1980.

[Brice 78] Brice, R.S., and Browne J.C..
Feedback Coupled Resource Allocation Policies in the Multiprogramming-
Multiprocessor Computer System.

Disc Caching in the System Processing Units of the HP 3000 Family of
Computers.

[Chang 85] Chang, J.
SunNet.
In Conference Proceedings of the 10th Anniversary Usenix Conference,

[Cheriton 83] Cheriton, D.R, and Zwaenepoel, W.
The Distributed V Kernel and its Performance for Diskless Workstations.
In Proceedings of the Ninth ACM Symposium on Operating System
Principles, pages 129-140. Stanford University, ACM, 11 West 42nd St.,
NY, NY, October, 1983.

[Clark 85] Clark, D.D.
The Structuring of Systems Using Ucalls.
In Proceedings of the Tenth ACM Symposium on Operating Systems
[Cooper 83] Cooper, G.H.

[Craft 83] Craft, D.H.
Resource Management in A Decentralized System.

[Ennis 84] Ennis, R.L., Griesmer, J.H., et. al..

[Fridrich 81] Fridrich, M. and Older, W.
THE FELIX FILE SERVER.

[Griesmer 84] Griesmer, J.H., Hong, S.J., et. al..

[Hanson 84] Hanson, S.J., Kraut, R.E., and Farber, J.M.
Interface Design and Multivariate Analysis of UNIX Command Use.

Cache-based Computer Systems.

[Kleinrock 85] Kleinrock, L.
Distributed Systems.

[Knuth 85] Knuth, D.E.
An Analysis of Optimum Caching.

[Korn 85] Korn, D.G. and Vo, K.
In Search of a Better Malloc.
[Lazowska 84] Lazowska, E.D., Cheriton, D.R., and Zwaenepoel, W.
File Access Performance of Diskless Workstations.

[Levine 84a] Levine, A.P.
In Proceedings CMG XV, International Conference on the Management
and Performance Evaluation of Computer Systems, pages 227-233. BGS

[Levine 84b] Levine, A.P.
B.A. Honors Thesis.
May, 1984

[Lyon 85] Lyon, B., et. al..
Overview of the Sun Network File System.
1985.

[McKusick 85] McKusick, M.K., and Karels, M.
Performance Improvements and Functional Enhancements in 4.3BSD.
In Conference Proceedings of the 10th Anniversary Usenix Conference,

A Comparison of Two Network-Based File Servers.

[Mogul 84] Mogul, J.
Representing Information About Files.
Submitted for Publication ->, 1984.

A Nested Transaction Mechanism for LOCUS.
In Proceedings of the Ninth ACM Symposium on Operating System
Principles, pages 71-89. UCLA, ACM, 11 West 42nd St., NY, NY,
October, 1983.

[Oldehoeft 85] Oldehoeft, R.R and Allan, S.J.
Adaptive Exact-Fit Storage Management.

[Ousterhout 85] Ousterhout, J.K., Da Costa, H., et. al..
A Trace Driven Analysis of the UNIX 4.2BSD File System.
In Proceedings of the Tenth ACM Symposium on Operating Systems

[PAWS 84] PAWS 2.0 Performance Analyst's Workbench System
[Paxton 79] Paxton, W.H.  
A Client-Based Transaction System to Maintain Data Integrity.  
In *Proceedings of the Seventh Symposium on Operating Systems Principles*,  

[Popek 81] Popek, G., Walker, B. et. al..  
LOCUS - A Network Transparent, High Reliability Distributed System.  
In *Proceedings of the Eighth Symposium on Operating Systems Principles*,  

[Quarterman 85] Quarterman, J., Silberschatz, A. and Peterson, J.L.  
The UNIX SYSTEM: 4.2BSD.  
Accepted for publication: *Computing Surveys *, 1985.

[Ramakrishnan 86] Ramakrishnan, K. K., and Emer, J. S.  
A Model of File Server Performance for a Heterogeneous Distributed System.  

[Rathi 84] Rathi, B.D.  
The Design and Performance Analysis of a Self Managing Secondary Memory.  

[Reiner 80] Reiner, D.S.  

[Richardson 83] Richardson, M.F., Needham, R.M.  
The TRIPOS Filing Machine, a front end to a File Server.  

[Sandberg 85] Sandberg, R., Goldberg, D. et. al..  
Design and Implementation of the Sun Network Filesystem.  
In *Conference Proceedings of the 10th Anniversary Usenix Conference*,  

[Satyanarayanan 81] Satyanarayanan, M.  
A Study of File Sizes and Functional Lifetimes.  
In *Proceedings of the Eighth Symposium on Operating Systems Principles*,  
Satyanarayanan, M., Howard, J.H., et al.
The ITC Distributed File System: Principles and Design.

Schroeder, D., Gifford, D., and Needham, R.
A Caching File System for a Programmer's Workstation.

Simonetti, J.D.
*A System Call Trace Facility*.

Smith, Alan Jay.
Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms.

Smith, A.J.

Spector, Alfred Z.

Stonebreaker, M.
Operating System Support for Database Management.

Sturgis, H., Mitchell, J. and Israel, J.
Issues in the Design and Use of a Distributed File System.

Sun Microsystems.
1985.

Sun Microsystems.
1985.

Sun Microsystems.
External Data Representation Reference Manual.
1985.
[Swinhart 79] Swinhart, D., McDaniel, G., and Boggs, D.
WFS: A Simple Shared File System for a Distributed Environment.
In *Proceedings of the Seventh Symposium on Operating Systems Principles*,

[Teorey 72] Teorey, J.J. and Pinkerton, T.B.
A Comparative Analysis of Disk Scheduling Policies.

Performance of a DECNET Based Disk Block Server.

The LOCUS Distributed Operating System.

[West 85] West, A.
The Sun Network File System, NFS — Business Overview.
1985.

[Zhou 85] Zhou, S., Da Costa, H. and Smith, A.J.
A File System Tracing Package for Berkeley UNIX.
In *Conference Proceedings of the 10th Anniversary Usenix Conference*,
VITA

Kim 'M' Korner was born in Salt Lake City, Utah, on July 4, 1953, the son of Jean Tomlinson Korner and Dr. Ija Nome Korner. He graduated Summa cum Laude from the University of Wisconsin at Green Bay with a B.S. in Analysis Synthesis in June of 1973. From September of 1973 until June of 1978 he was a graduate student of Clinical Psychology in the Department of Psychology at the University of Utah. The following year he was employed as a helicopter skiing guide by Canadian Mountain Holidays in Banff, Alberta. He was a graduate student in Computer Science at the University of Wisconsin at Madison during the spring term of 1978. While there, he was employed by the Madison Academic Computing Center to test and evaluate statistical software systems. He entered The Graduate School of The University of Texas, Computer Science department, in the fall of 1978. While a graduate student, he has been employed by the University of Texas Computation Center as a systems programmer, and is currently a consultant to the MCC Human Interface Group in Austin. His graduate tenure at Texas has also been supported by a departmental ARO research fellowship.

Permanent Address: 8806 Clarewood Dr.
Austin, Texas 78758

This dissertation was typed by Kim Korner.