ADAPTIVE CACHING IN A DISTRIBUTED FILE SYSTEM

BY
SWEE BOON LIM
B.S., George Washington University, 1987
M.S., University of Illinois, 1991

THESIS
Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois
ADAPTIVE CACHING IN A DISTRIBUTED FILE SYSTEM

Swee Boon Lim, Ph. D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1995
Roy H. Campbell, Advisor
Michael W. Condry, Advisor

A distributed computing environment includes the hosts, the applications, and the network. Since computing environments can be very diverse, a distributed operating system must have flexible services that can adapt to different computing environments. The specific service being considered here is that of file system caching.

The importance of file system caching is widely recognized. An effective file system caching service can reduce local disk accesses and remote file server accesses significantly. The thesis is that a flexible and adaptive file system caching service is practical and important for performance. A flexible file system service permits performance tuning through customized caching strategies. An adaptive file system caching service adapts to the computing environment by selecting strategies suitable for the environment.

To validate the thesis, a distributed file system with a flexible and adaptive file caching service has been implemented in the Choices operating system. This distributed file system allows each file to have its own caching strategy. In addition, the data from a single file may be cached in additional secondary caches. A secondary cache supports caching of file data in a tertiary storage device, such as a local disk, in addition to main memory. The Choices file system caching framework defines how new caching strategies can be added to the system. It also defines the interaction between a file's in-memory cache and its secondary caches.

The Choices distributed file system provides a set of caching strategies that are suitable for the most common file access patterns. In addition, the system chooses the appropriate strategy for each file based on how the file is being accessed, and how it had been accessed in the past. This adaptive caching service does not depend on the application or the user to provide caching hints.

Experiments have been conducted using two different workloads to evaluate the benefits of adaptive file system caching. These workloads have distinct file access behaviors. These experiments show that Choices' adaptive file system caching consistently outperforms fixed caching strategies. Adaptive caching improves both main memory and secondary cache utilization.
To my parents
Acknowledgements

I wish to express my deepest gratitude to all those who have supported me through this effort. I especially thank my advisors, Professor Roy Campbell and Professor Michael Condry, whose technical and financial support made this thesis possible. Besides Roy and Michael, I thank the other members of my committee, Professor Dennis Mickunas, and Professor Jane Liu.

I thank Dave Raila for keeping the computers running, and for his work on the SPARCstation 2 port of Choices.

I acknowledge the assistance provided by the members of the Choices project: Aamod Sane, Theron Thock, See-Mong Tan, Amitabh Dave, Ellard Roush, and Lun Xiao.

I thank Anda Harney and many of the non-academic staff members who work for the Department of Computer Science for their professional support.

I thank my parents for their love and support. They made my education in the United States possible.
# Table of Contents

## Chapter

1 Introduction .......................................................... 1
   1.1 Contributions ................................................. 2
   1.2 Thesis Outline ................................................ 3

2 Motivation .......................................................... 5
   2.1 Need for Flexibility .......................................... 5
       2.1.1 Hosts .................................................. 6
       2.1.2 Applications .......................................... 7
       2.1.3 Networks ............................................. 7
       2.1.4 Performance Objectives .............................. 8
   2.2 Need for Adaptability ....................................... 9
   2.3 Summary ...................................................... 10

3 Object Orientation ............................................... 11
   3.1 Data Encapsulation .......................................... 11
   3.2 Data Abstraction ............................................ 12
   3.3 Interface and Code Sharing ................................. 12
   3.4 Polymorphism ............................................... 13
   3.5 Design Sharing ............................................. 13
   3.6 Summary ...................................................... 14

4 Related Work ...................................................... 15
   4.1 Distributed File Systems .................................. 15
   4.2 File Access Behaviors ..................................... 17
   4.3 Exploiting File Access Behaviors ....................... 18
   4.4 Client and Server Caches ................................ 20
   4.5 Network Characteristics .................................. 20
   4.6 Other Cache Optimizations ................................ 21
   4.7 Flexible Virtual Memory and File Caches ............. 22
   4.8 Adaptive File Systems ..................................... 24

5 The Foundation ..................................................... 25
   5.1 Conventions .................................................. 26
   5.2 The Virtual Memory Management System Framework ..... 26
       5.2.1 Virtual Memory Hardware Abstraction ............ 27
       5.2.2 Physical Memory Management ..................... 27
       5.2.3 Logical Memory Management ....................... 30
       5.2.4 Caching Support ..................................... 30
       5.2.5 Virtual Address Space Management ............... 31
   5.3 The File System Framework ................................. 31
9 Performance Evaluation .......................................................... 148
   9.1 The Experiments ............................................................ 148
      9.1.1 Trace-driven Workloads ........................................ 151
      9.1.2 Workload Characteristics ..................................... 154
      9.1.3 The Experimental Computing Environment ................ 158
      9.1.4 Summary ............................................................ 161
   9.2 Observations ............................................................... 161
      9.2.1 Building the Choices kernel ................................. 161
      9.2.2 SMS ............................................................... 167
      9.3 Summary ............................................................... 170

10 Conclusion .......................................................................... 171

Appendix ................................................................................. 173

Bibliography ........................................................................... 173

Vita ......................................................................................... 184
List of Tables

7.1 Time required to write a large file ........................................ 111
8.1 File access behaviors and their caching strategy preferences .................. 124
8.2 Encoding of file access behavior attributes in hints. .......................... 132
8.3 The open flags for file access behavior hints. ................................ 132
9.1 File size distribution of header files. ........................................ 154
9.2 File size distribution of source files. ........................................ 155
9.3 File size distribution of object files, excluding the kernel .................... 155
9.4 File size distribution of archive files. ....................................... 155
9.5 File size distribution of makefiles. ........................................... 156
9.6 File size distribution of dependency files. .................................... 156
9.7 File size distribution of all files accessed. ................................... 156
9.8 Programs invoked to build the Choices kernel. ............................... 157
9.9 Files accessed by SMS. ....................................................... 158
9.10 Building the Choices kernel with different caching strategies ............... 161
9.11 Differences between LCACHE 16 Mb and ADAPTIVE 16 Mb. .................. 163
9.12 Time needed to build the Choices kernel on SunOS and with local files .... 166
9.13 Replaying the SMS workload with different caching strategies .............. 168
9.14 The SMS workload and different caching strategies ......................... 168
9.15 Time needed to complete the SMS workload on SunOS and with local files .. 169
List of Figures

5.1 The original Choices virtual memory management system framework .................. 28
5.2 The class hierarchy of the original Choices virtual memory management system .. 29
5.3 The class hierarchy of the original Choices file system .................................. 33
5.4 The original Choices file system framework .................................................... 34
5.5 The corresponding entities in the BSD file system ........................................... 35
6.1 The architecture of a flexible file caching service .......................................... 39
6.2 Stacking secondary cache ................................................................................. 43
6.3 Distributed file system caching ......................................................................... 44
6.4 The Choices flexible file system caching framework ......................................... 47
6.5 The Choices flexible file system caching class hierarchy .................................... 48
6.6 The pseudo code for doWriteBehind ............................................................... 77
6.7 The pseudo code for doFreeBehind ................................................................. 78
6.8 The pseudo code for doReadAhead ................................................................. 78
6.9 The new MemoryObjectCachingContainer class and the class hierarchy ........ 86
7.1 Remote access to persistent object ..................................................................... 98
7.2 Remote access to persistent storage (or persistent memory object) ................. 99
7.3 The distributed file system proxy classes in the Choices class hierarchy ........... 100
7.4 Differences between the thread model and the continuation model ..................... 109
7.5 An example: creating and passing continuations ............................................. 113
7.6 Parent-child relationships among continuations .............................................. 114
7.7 Executing continuations ................................................................................... 115
8.1 The pseudo code for setHints and clearHints ................................................. 134
9.1 As file access behavior ................................................................................. 157
9.2 SMT's file access behavior ............................................................................ 159
9.3 SMS's file access behavior ............................................................................ 160
Chapter 1

Introduction

A distributed file system allows a host on a network to access files on another host[60]. File system caches in a distributed file system contain data that are likely to be accessed in the near future. The data may be from a local disk[107] or from a remote host[43, 75]. These caches significantly reduce accesses to the local disks[11, 95, 108], and the remote file server[4, 9, 43, 71, 75]. A file caching strategy defines how data is stored in local memory, how data is fetched from the local disk or the remote server, and how data is removed from local memory. Local memory on a host includes its main memory and any tertiary memory, such as disks and non-volatile random access memory (NVRAM).

My thesis is that a flexible and adaptive file system caching service is both practical and important for performance. A flexible file system caching service can support numerous different file caching strategies. It permits performance tuning through customized caching strategies that take into account the characteristics of the computing environment. These characteristics include the applications and their file access patterns, the configurations of the hosts, and the characteristics of the network. An adaptive file system caching service adapts to a particular computing environment by choosing the appropriate caching strategies for environment.

To validate my thesis, I built a prototype distributed file system in the Choices operating system[15, 17, 69, 97, 99]. This distributed file system has a flexible and adaptive file caching service. This file caching service derives its flexible from allowing each file to have its own caching strategy, and its own secondary cache(s). Furthermore, a file may have a different caching strategy on each client, as well as, on its file server. Secondary caches exploit the memory hierarchy of a host. A secondary cache supports caching of a file’s data on a tertiary storage device, such as a disk. The Choices distributed file system uses secondary caches to support client local disk caching of remote file data.

An object-oriented framework identifies the different types of components in the Choices file caching service. It also describes the relationships between the various types of components. Applications
and users of the distributed file system select a caching strategy for each file by plugging together appropriate components of each type. A flexible file caching service should also be extensible. It should permit new caching strategies to be integrated into the distributed file system in an orderly manner. This framework provides extensibility by defining how new components representing new strategies can be added to system.

Using this framework, several caching strategies have been implemented. These strategies are optimized for most common file access patterns observed in academic and research environments[4, 80]. A secondary cache that supports caching of remote or local file data on a local disk has also been implemented.

The Choices distributed file system is adaptive because it uses simple heuristics to select appropriate caching strategies for each file from a set of implemented caching strategies. Caching strategy selection for a file is based on the current dynamic behavior of the file, as well as, how the file was accessed in the past. The distributed file system stores, as part of the meta-data of the file, access behavior history information. The distributed file system uses this information as hints when selecting caching strategies. Server and client access behavior information are stored separately. This allows the system to select different strategies for the the client and the server. In addition, users and applications may provide additional hints to aid the selection of an appropriate caching strategy. User and application hints are particularly useful for new or temporary files because these files may not have prior access behavior history information.

To demonstrate the benefits of an adaptive distributed file system, experiments were conducted on the prototype file system. These experiments show that adaptive file system caching consistently performs better than non-adaptive caching file system caching.

1.1 Contributions

To summarize, the contributions of this thesis are:

1. A framework for flexible file system caching that supports multiple caching strategies. In particular, each file can have its own caching strategy. Furthermore, a file’s caching strategy may be different on each client and its server.

2. This framework also takes advantage of the memory hierarchy of each host. Each file may be cached in main memory, as well as, any number of tertiary storage devices on the clients and the server.

---

1A file’s meta-data is data which is not part of the contents of the file. For example, a UNIX file’s meta-data includes its size, its type, its owner, its group, and access permissions.
3. Simple heuristics for selecting caching strategies based on previous access behaviors and current
dynamic access behaviors.

4. This thesis presents our experience with the prototype file system and demonstrates the perform-
ance benefits of adaptive file system caching.

5. Our experiments also demonstrate the benefits of two unique features of the Choices flexible file
system caching framework and distributed file system. The first is the use of zombies to reclaim
main memory. The second is the use of continuations to support asynchronous remote file accesses.

1.2 Thesis Outline

The rest of this thesis is divided into the following chapters:

2. Motivation -
   This chapter presents the need for a distributed file system with flexible and adaptive file caching.

3. Object orientation -
   Flexibility requires customizability and extensibility. This chapter describes the software engi-
eneering techniques employed by object-oriented systems and their benefits. These benefits include
customizability and extensibility.

4. Related work -
   This chapter presents related work which includes a survey of distributed file systems, client and
server caching strategies, cache optimizations, file access characteristics, adaptive systems, and
user-level paging.

5. The foundation -
   It presents a brief overview of the foundation frameworks that have been integrated to build an
adaptive distributed file system. These frameworks include the Choices file system framework and
virtual memory system framework.

6. Flexible file caching -
   It describes an architecture and a framework for flexible file caching.

7. Distributed access and caching hints -
   It describes extensions to the Choices frameworks to support distributed file access. It also covers
various optimizations to improve distributed file access performance. One of these optimizations
is the use of continuations to support asynchronous remote file accesses.
8. *Adaptive file caching* -

It discusses caching strategies implemented in the *Choices* distributed file system, and the heuristics used to select a caching strategy for each file.

9. *Experiments and results* -

It describes the experiments that have been conducted. It also presents the key observations.

10. *Conclusion* -

The final chapter concludes this thesis and summarizes the benefits of adaptive file caching.
Chapter 2

Motivation

This chapter presents the motivation for building a distributed file system with flexible and adaptive file caching. First, it presents the need for flexibility. Then, it presents the need for adaptability.

2.1 Need for Flexibility

A traditional distributed file system[60] implements a single caching strategy. The designers of the distributed file system select an appropriate strategy based on their design goals for their file system and their model of a “typical” computing environment. Their model of a “typical” computing environment is usually derived from observations made in their own computing facility[4, 9, 80]. It normally consists of file servers and clients connected by a local area network.

The problem with this approach is that these strategies may not perform well outside the designer’s “typical” computing environment. In some cases, they may not work at all in a different environment. For example, the Andrew File System (AFS)[43] was designed to operate in an environment where a small number of file servers must serve many clients. Hence, scalability is very important. The Andrew File System maximizes scalability by reducing server load and network traffic. It always caches the entire remote file on the client’s local disk. Such a system will not work in an environment with diskless workstations. On the other hand, Sun Microsystem’s Network File System (NFS)[100, 101] was designed to support diskless workstations. It does not use client disks for caching even though they are available.¹ As a result, NFS does not perform well in a computing environment where scalability is crucial[43, 75].

A distributed computing environment includes the hosts, the applications, and the networks. Different hosts have different configurations. For example, they may have different amounts of main memory

¹Although local disk caching can be added via servers outside the file system[74], it is not part of the file system. In addition, Solaris 2.3 supports some client disk caching options via the Cache File System[100].
and some hosts may be diskless. In addition, different hosts may have different roles, a host may be a client, a server, or both. Different applications have different file access characteristics. A file may be accessed sequentially, randomly, repeatedly, once-only, and/or temporarily. Different networks have different bandwidth and latency characteristics. As illustrated in the previous example, distributed file systems with fixed caching strategies cannot be expected to perform well in other possible computing environments. Since computing environments can be very diverse, a distributed file system should have a flexible file caching service that can support numerous caching strategies.

The rest of this section presents some characteristics of the various components in a distributed computing environment and show why a flexible file system caching service is needed to take advantage of these characteristics to obtain better performance.

2.1.1 Hosts

A good file system caching service should be able to exploit the hardware configuration of each host in the distributed system. The memory hierarchy on each host is especially important. Local disks on clients can be used to cache remote file data[43]. Client local disk caching increases the effective size of the client cache. It reduces cache misses and reduces the load on the server. It can result in lower client response times and increase server scalability[43].

Client disk caching can also reduce the effects of network latency. A client may get better response times by caching data stored on remote file servers in local disks. For example, the average access times\(^2\) of most disk drives are between 8 to 16 milliseconds. The minimum network latency for a round-trip message between the east and the west coast is 32 milliseconds.\(^3\) This network latency is two to four times more than the average access times of local disks. Hence, client local disk caching reduces response time when the file server is far away.

Alternatively, client local disk caching may not be appropriate when the server is nearby and the server has large main memory caches. For example, the time required to exchange request and reply messages on most local area networks is less than 2 milliseconds.\(^4\) This latency is significantly less than the average access times of most disk drives. Depending on the amount of data transferred, the bandwidth of the network, the transfer rate of the client disks, and the server load, the client may be able to page data out to the file server’s memory faster than paging to a local disk[75].

In summary, caching of remote data on client disks may be appropriate for far away servers but may be inappropriate for nearby servers. Since a client may access files on both nearby and distant servers, the client should be able to cache remote file data on local disks on a per file basis. Furthermore, some

---

\(^2\)The access time is the time required to reach the data to be accessed. It includes the seek time and the rotational latency. It does not include the time required to transfer the data.

\(^3\)This is based on a distance of 3000 miles each way at the speed of light. Real networks take much longer.

\(^4\)This timing can be easily verified using 'ping'.
distributed computing environments may have clients that are diskless. A file caching service should be flexible enough to cache remote file data in local disks when local disks are available on the client and when appropriate.

The client or server role of individual hosts can also affect the selection of appropriate cache replacement policies. In a distributed file system, caches can be present on both the client and server[43, 60, 75]. In such a system, the server only “sees” references that have not been satisfied by the client cache[71]. For a sufficiently large client cache, this filtering eliminates the temporal locality of references seen in the original reference stream[71, 125]. Because of this filtering, Willick[125] found that a frequency based cache replacement policy, such as Least-Frequently-Used (LFU)[29] or Frequency-Based-Replacement (FBR)[92], is more suitable for the server, while a temporal locality based policy, such as Least-Recently-Used (LRU)[29], is more suitable for the client. This implies that a file system caching service should be able to support different cache replacement policies on the client and the server.

Since a particular host can be both a client of another server and a server to other clients, the file caching service should allow different cache replacement policies to coexist on the same host. For example, a host should be allowed to use a frequency based replacement policy for local files accessed by remote clients, and a temporal locality based replacement policy for purely local files and remote files.

2.1.2 Applications

The file access characteristics of individual applications can also influence file system caching strategies. For example, if a file is accessed sequentially, a Most-Recently-Used (MRU)[29] cache replacement policy is more appropriate than a LRU cache replacement policy[53]. In addition, prefetching may be appropriate for some sequentially accessed files[5]. Similarly, caching could be turned off or minimized if the data in a file is accessed only once[53].

Other characteristics of file access behaviors are average request sizes and the request inter-arrival times[88]. These characteristics can be used to select appropriate block sizes for transfers between the storage device and the server, as well as, for transfers between the client and the server. Since each application may open files that require different transfer sizes, data fetch policies, and cache replacement policies, a flexible file system caching service could improve file system performance.

2.1.3 Networks

The primary characteristics of a network are bandwidth and latency. Networks with high bandwidths are usually known as high speed networks. Studies of several high speed networks have found that the actual measured throughputs of these networks depend on the data transfer size[23]. In particular, higher speed networks do not provide a significant performance advantage over slower networks when
Transfer sizes are small. The throughput of a network usually increases proportionally with transfer size until the maximum throughput is attained. The transfer size at which the maximum throughput is attained varies depending on the network. For file system network transfers, Lazowska et al. [55] suggest that a transfer size of 8 to 16 kilobytes is adequate for Ethernet while Baud et al. [5] indicate that a transfer size of 128 kilobytes is required for optimal performance on UltraNet [6]. This means that a file system caching service should support different transfer sizes. This capability is especially important in a computing facility where each individual host may be connected to more than one network. For example, the Tapestry computer lab has file and compute servers that are connected to each other by a FDDI network. These servers are also connected to client workstations by an Ethernet network. The file system caching service on these servers should be able to use different transfer sizes for transferring data over FDDI and Ethernet.

The latency characteristics of a network influence the selection of an appropriate transfer size. Network latency is usually determined by the host protocol processing overhead and the distance between the communicating hosts. Network latency affects the file system throughput since each file system I/O operation typically requires at least one request message and one reply message. A large transfer size can minimize the effects of large latencies by reducing the number of I/O operations. A large transfer size can also reduce the host protocol processing overhead [55].

2.1.4 Performance Objectives

Different computing environments may have different performance objectives. Possible performance objectives are maximum scalability, minimum response times, and maximum throughput. Each of these objectives has its own special requirements. Very often, a particular objective will have requirements that conflict with the requirements of other objectives. For example, maximizing scalability may conflict with minimizing response time. If scalability is crucial, then the caching strategy should reduce server load [43, 55]. For example, the Andrew File System reduces server load by copying the entire file from the server to a client disk when the file is initially accessed [43]. This allows subsequent client accesses to the file to be satisfied by the copy cached on the client’s disk without contacting the server. However, this strategy increases response time. If response time is more important, then caching remote data in main memory only may be more suitable (assuming that the client main memory cache is sufficiently large) [75]. Because of these conflicting requirements, it is unlikely that a file system with a fixed caching strategy, like the Andrew File System, designed with a particular objective in mind, will perform well in a different computing environment that has a radically different objective.

---

5 This may not be true for networks using small fixed-size cells, such as ATM networks which can achieve 3.4 gigabits per second. We are currently studying the throughput characteristics of these networks.
2.2 Need for Adaptability

A flexible file caching service introduces a new problem. This is the problem of how to select the appropriate caching strategies. One solution is to force the users and the applications to choose desirable caching strategies. This solution is inadequate for the following reasons:

1. The applications need to be modified to specify the desirable caching strategies. The file access characteristics of these applications may not be known. It may be impossible to modify some applications because the applications’ source code is not available. Furthermore, it is also tedious to modify a lot of existing applications.

2. The developers of an application may not have intimate knowledge of the runtime environment of the application. For example, the developers may not know whether the runtime files will be local or remote, whether the hosts will have local disks, or the amount of memory that the hosts will have.

3. Even if the developers of an application know about the initial runtime environment of the application, this runtime environment may change and evolve with time. The strategies selected for the initial computing environment may not be appropriate in a newer environment. Hence, the developers may have to modify the application to select different caching strategies whenever the computing environment evolves.

4. Application developers may need to support different versions of the same application. Each version specifies different caching strategies to optimize for a particular computing environment.

5. The users and the application developers may not be sophisticated enough to select the appropriate strategies. It is hard to take into account the numerous variables that contribute to the selection of suitable caching strategies.

A better solution to this problem is to build a distributed file system with an adaptive file caching service. A distributed file system with flexible file caching provides a set of caching strategies. A distributed file system with flexible and adaptive file caching attempts to adapt to a computing environment by selecting appropriate caching strategies from a set of implemented caching strategies. Users and applications of the distributed file system may provide hints to help select suitable caching strategies. However, a distributed file system with adaptive caching should not depend on these hints to perform well.
2.3 Summary

A distributed file system with flexible file caching is desirable because computing environments are diverse. This system should also be adaptive to take advantage of different strategies/options provided by its flexibility, without depending on user and application hints/intervention.
Chapter 3

Object Orientation

From the previous chapter, it is obvious that a distributed file system with flexible and adaptive file caching is desirable. A flexible file caching service implies that it should also be customizable and extensible. Customizability permits the use of different caching strategies to meet the challenges of different computing environments. Extensibility permits new caching strategies to be integrated into the distributed file system in an orderly manner for new computing environments.

An object-oriented approach is ideal for building such a file caching service. An object-oriented approach provides the benefits of customizability and extensibility through the use of several software engineering techniques[46, 67, 97]. In addition, these techniques also provide reusability, portability, and maintainability[46, 67, 97]. These techniques include data encapsulation, data abstraction, interface and code sharing, polymorphism, and design sharing[67].

3.1 Data Encapsulation

Data encapsulation isolates a software component’s behavior from the component’s state[67, 76]. An object is a software entity that consists of a set of state data and a set of operations[76, 122] on the object’s state data[76, 122]. Objects support data encapsulation by enforcing the use of operations to effect state changes[32, 67, 79]. Hence, an object’s operations define the behavior of the object[76]. These operations determine how the object interacts with other objects and insulate other objects from changes to the structure of the object’s state data. Data encapsulation is sometimes also called information hiding[76]. Data encapsulation encourages software designers to decompose systems into objects by minimizing the impact resulting from state data structure changes[67].

In Smalltalk[41] and Objective-C[76], invoking an operation on an object is known as sending a message to the object. The object is called the message receiver[76]. The code executed in response to
the message is known as a *method*. In C++[34, 116], methods are known as *member functions*. Public member functions define an object’s external behavior. Private and protected member functions define an object’s internal behavior. An object’s state is composed of its *instance variables*[79], which are also called the *data members*[34, 116] of the object.

### 3.2 Data Abstraction

A class is a template for building similar objects[79]. A class specifies the operations and state data that all objects of the class will have. Classes support *data abstraction* by describing the behavior and structure shared by a set of objects[34, 41, 67, 116]. In other words, the different kinds of components in a system are represented by different classes of objects. A class characterizes the behavior and structure of a kind of components in a system.

Classes differ from traditional user-defined type mechanisms because they support the notion of *sub-types*[79]. A class defines a sub-type of a second class if instances of the first class can be used anywhere that instances of the second class can be used[67].

### 3.3 Interface and Code Sharing

Classes belong to a *class hierarchy*. They share behavioral and structural specifications using *inheritance*[67]. A *child class* in a hierarchy, known as a *subclass or derived class*, inherit specifications from one of more *parent classes*, also known as its *superclasses* and *base classes*. If a child class can inherit from more than one parent class, the class hierarchy supports *multiple inheritance* otherwise it supports *single inheritance*.

If a child class inherits the *interfaces* of its parent classes, the class hierarchy supports *interface inheritance*. The interface of a class defines the set of operations the class supports, *i.e.*, it specifies the behavior of the class. Interface inheritance supports interface sharing. Some languages, such as the Object Management Group’s Interface Definition Language[26], only support interface inheritance.

If a child class inherits both the behavioral and structural specifications of its parent classes, the class hierarchy supports *implementation inheritance*. Implementation inheritance supports both interface and code sharing. Most programming languages, such as SmallTalk[41], C++[34, 116], and Objective-C[76], support implementation inheritance.

When implementation inheritance is used, some classes are designed such that only other classes can inherit from them. These *abstract classes* group methods and instance variables that will be used by other classes into a common definition[76]. Abstract classes do not have any instances[41]. They are not useful by themselves but provide code and definitions that are inherited by their subclasses. They help
reduce the implementation burden of their subclasses[76]. Since abstract classes must have subclasses, they are sometimes called abstract superclasses[76].

A concrete class provides a complete implementation of an interface. This interface may be specified by its abstract superclasses. Concrete classes have instances.

### 3.4 Polymorphism

The ability of different objects to respond, each its own way, to identical messages is called polymorphism[76]. Object-oriented languages provide polymorphic functions, which can be applied to objects of various types[32, 79]. Polymorphism simplifies programming by permitting conventions to be established that can be reused in class after class[76]. It makes programs more general by allowing them to manipulate more kinds of objects[67]. It also makes programs more extensible by allowing them to be applicable to as yet unspecified objects[67].

Polymorphism is supported by dynamic binding[76] or late binding[34, 76, 116]. Dynamic binding is usually used by dynamically typed languages like Smalltalk and Objective-C. With dynamic binding, sending a message requires the run-time support for the language to resolve the type of the message receiver to determine the method to be invoked[41, 76].

C++, a statically typed language, supports late binding[34, 116]. Late binding requires the message receiver to be statically typed in the source code, but does not require the type to be exact[76]. The message receiver is statically typed in the source code by a declared class. The actual run-time message receiver must be an object typed to the declared class or to any class that it inherits from. While late binding is "dynamic", it carries with it strict compile-time type constraints. For example, the C++ mechanism that provides late binding is known as a virtual function call[34, 116]. Each object that supports virtual function calls has one or more tables of pointers, known as vtables.\(^1\) When a virtual function call takes place, the function invoked is determined by fetching an address from a slot assigned to the function in a vtable belonging to the invoked object.

### 3.5 Design Sharing

The most important form of software reuse is reusing the architecture or function decomposition of systems within the same application domain[30, 67]. An abstract class is a reusable design for a kind of component in a system[126]. A framework is a reusable design for entire systems or subsystems[67].

A framework consists of a set of classes and a specification for combining objects to build a working system[46]. Such a working system is called an ensemble[46]. A framework describe how a system is

\(^1\)The number of vtables an object has depends on the object's inheritance lattice, which is known at compile time.
decomposed into components[30] and how there components of different kinds interact[67]. A framework permits reuse of design, as well as, implementation[126].

A framework is like an architectural specification, except polymorphism allows components of a subtype to be substituted for components of a specified type. Frameworks are useful for both characterizing (design) and building (implementation) a system[67].

3.6 Summary

Customizability and extensibility are desirable in a flexible file system caching service. An object-oriented approach provides both customizability and extensibility. The above techniques allow a complex system, like a file system caching service, to be decomposed into a set of objects with different properties and attributes. A class hierarchy specifies the kinds of the objects in the system. A framework defines how objects of different classes can be plugged together.

An object-oriented approach provides customizability because different classes of objects, which implement different properties and attributes of various caching strategies, can be integrated into a distributed file system. It constructs a customized caching strategy for a file by choosing and combining objects that have desirable properties and attributes.

An object-oriented approach provides extensibility because polymorphism and a design with an appropriate set of abstractions allow new kinds of objects to be added to the distributed file system in an orderly manner[67]. Polymorphism permits an object to be substituted by another object of a similar kind, which provides new functionality, in the future.

The additional benefits of object-orientation includes maintainability, portability, and resuability. These benefits allow rapid prototyping of ideas. A distributed file system prototype was completed in less than three months to test some of ideas presented in this thesis[25]. This prototype provided invaluable insight into the design and implementation of an efficient distributed file system.
Chapter 4

Related Work

This chapter discusses related work which includes:

1. a survey of distributed file systems,
2. file access behaviors,
3. exploiting file access behaviors to improve performance,
4. client and server caches,
5. various file system cache optimizations,
6. Flexible virtual memory and file caches, and
7. adaptive file systems.

4.1 Distributed File Systems

With the increased availability of workstations and local area networks, distributed file systems have become increasingly popular. Early distributed file systems such as the Newcastle Connection[13] does not support caching of remote data. The Newcastle Connection uses a simple Remote Procedure Call[7] mechanism to access file data on remote hosts.

Then, Sun Microsystems developed the Network File System [100, 101], widely known as NFS, to provide effective file access to diskless workstations. It has a stateless server and allows clients to cache remote data. A stateless server does not maintain any client related information. It does not know how many clients are connected to it, how many clients are actively using a file, or what data are cached by its clients. NFS clients fetch data in fixed sized blocks from the server on demand. The client cache
semantics are not defined by the NFS protocol\[77, 78\]. Some implementations write all changes to the server immediately\[110\], while others delay writes\[60\], or "write-on-close"\[101\]. However, the server is required to commit writes to disk before returning to the client\[100\]. UNIX\[1\] semantics are not maintained.

AT&T's Remote File System (RFS)\[91\] is similar to NFS. However, a RFS client always writes all changes to the server immediately to enhance data consistency\[2\]. RFS maintains a stateful connection between the client and the server, and it provides full UNIX semantics.

The Sprite distributed file system\[75\] extends the NFS concept, it caches blocks on both the client and the server. It uses a more elaborate cache consistency protocol that allows both the client and the server to delay writes when there is no write sharing\[75\]. Write sharing occurs when several clients concurrently access a file and at least one client writes to the file. When a file is opened, the Sprite client always contacts the file’s server. Each file server maintains sufficient state to determine whether write sharing is about to occur. When write sharing of a file is detected, the server notifies the file’s clients to stop caching the file. This causes the clients to write changes to the server immediately and to always read from the server. Since write sharing is rare\[4\], Sprite’s caching strategy significantly reduces server load, and hence scales better than NFS.

The Andrew File System\[42, 43, 48\], also known as AFS, improves scalability by further reducing server load. It caches whole files on client disks. Once a file is cached on a client, the client can access the contents of the file without referring to the server. AFS maintains cache consistency\footnote{AFS ensures that a client opening a file will always obtain the latest changes that have been closed. It does not provide cache coherence. Cache coherence is not maintained when two clients concurrently write to the same file.} using callbacks\[48\]. A callback is a promise by the server to notify the client when a file cached by the client has to be invalidated. Callbacks eliminates the need to contact the file server on each request to open a file. AFS reduces server load by increasing the effective size of the client cache and by reducing the number of messages/server requests needed to main cache consistency.

The Coda File System\[51, 103, 104\] extends the AFS to support disconnected operation. It also uses whole file caching. The Cedar File System from Xerox also caches whole files on client disks. However, the Cedar File System only supports immutable shared remote files\[39\].

SunSoft introduced the Cache File System in Solaris to provide client local disk caching for NFS and local disk caching of data for slower storage devices like CD-ROM\[109\]. In Solaris, a user enables local disk caching when he/she mounts a cached file system. A cached file system is a virtual file system\[52\], i.e. it uses the kernel vnode file system interface. It represents a back file system, that may be a NFS file system or a CD-ROM resident file system. It caches data from the back file system in a local UNIX file system. Like AFS, it enables caching for all files in the back file system.
The AutoCacher[74] is similar to the Cache File System but it is designed to cache immutable NFS files. The AutoCacher enables caching by path name instead of by file system. It emulates an NFS server, like the automounter[14], instead of a virtual file system.

In Spring of 1992, a prototype distributed file system was developed as part of an advanced operating systems class at the University of Illinois. This prototype demonstrated that Sprite’s memory based client caching mechanism can coexist with Andrew’s disk based client caching mechanism on the same host[25].

The Amoeba distributed system assumes most files are small and most of their contents are completely accessed[90]. It attempts to improve response time by caching whole files in server memory[89]. Applications in Amoeba communicates directly with the file server. They always read and write whole files. Amoeba client hosts do not cache remote file data.

The LOCUS distributed operating system[121] has a location-transparent distributed file system. It extends the UNIX file system to support file replication to increase availability and performance.

4.2 File Access Behaviors

In 1985, Oustershout et al. [80] analyzed the file access behaviors of UNIX 4.2 BSD file systems[72] in a time-sharing academic/research environment. User-level file access activities were recorded and analyzed. They found that most files were open for a short time and those that were opened again were opened again very shortly after they were closed. Most files were accessed sequentially. Lengths of most sequential runs were short (a few kilobytes) because most files were small. Although long sequential runs (more than 25 kilobytes) were rare, they accounted for 30% of the total data transfer volume. Most new information was deleted a few minutes after creation. They also found throughput requirements per user to be low (300-600 bytes/sec). Their block cache simulations show that moderately sized caches used in UNIX systems can reduce disk traffic significantly. Their simulations also indicate that disk accesses could be reduced significantly if larger block sizes (16 kilobytes or more) were used. The Sprite distributed file system was built and optimized around the file access characteristics observed in this study[75].

In 1991, Baker et al. [4] repeated the above study by instrumenting the Sprite distributed file system. Similar file access characteristics were observed. However, throughput requirements per user had increased 20 to 30 times because of increases in raw CPU speed. Another difference is that more very large (multi-megabyte) files were accessed. Long sequential runs of 1 megabyte or more were also more common (more than 10% of total transfer operations), partially because larger files were accessed. Long sequential runs also accounted for more of the total data transfer volume (55-80%). These observations suggest that optimizing for long sequential runs is important, especially because even more and longer sequential runs were observed by us [62]. Sprite’s client caching strategy was found to be effective. It
reduced server traffic volume by 50%. Sprite's ability to vary the amount of memory dedicated to file system caching was also found to be useful. The study found that write sharing is rare. The authors also found their applications which used write-sharing performed better when revised to use server processes to support sharing. This is because the revised applications took advantage of their individual sharing semantics to reduce the number of messages required to remain consistent.

Biswas and Ramakrishnan had similar observations in their characterizations of file access behavior in several production VAX/VMS environments[8]. The production environments studied were scientific timing sharing, program development, office applications and decision support airline reservations, order-entry and report generation, and on-line betting. Like the BSD and Sprite studies, they found that most files were open for a very short time (half were open less than a second) and re-opens, if present, occurred soon (40-60% of reopens took place within a minute of the last close). Very few files (less than 5-8%) were simultaneously shared between users, i.e. open at the same time. Over 70% of file opens were for read, except in a transaction processing environment. In a transaction processing environment, writes were more common (over 60%). More significantly, their characterizations show that there are significant differences in access behaviors across different environments.

4.3 Exploiting File Access Behaviors

Baud et al.'s RFIO[5] uses file access hints to improve data transfer throughput and response time. RFIO was designed to transfer data from servers to clients quickly and efficiently. It classifies file accesses into three different categories; sequential, pseudo-sequential, and random. The application supplies the file access category as a hint to the server when it opens a file. The server uses this hint to determine how data is fetched from disk and transferred to the client. These hints were found to be useful and provided significant improvements in throughput and response time. Even though RFIO does not cache, it does show that application hints can aid in the selection of appropriate fetch and transfer strategies to improve performance.

Korner[53] investigated using a knowledge based approach to exploit file access characteristics to improve performance. First, he logged file accesses on a production system. Then, he analyzed the logs to obtain knowledge about the expected access behaviors of files residing in different directories and files with different extensions. The access characteristics considered were whether a file is likely to be accessed again soon, and whether it is likely to be accessed sequentially. For example, he classified files with .o extensions as being more likely to be accessed randomly and files in the /usr/include directory as being more likely to have repeated accesses. Then, he integrated the obtained knowledge into a simulated file system to provide caching hints. The simulated file system does not cache data from a file if the file is expected to be accessed only once. If the file is cached, the cache replacement policy used depends on
whether the file is expected to be accessed sequentially. The MRU cache replacement policy was used for sequentially accessed files and the LRU cache replacement policy was used for randomly accessed files. For example, caching was disabled for files in /usr/include and MRU was used on files with .o extensions. His simulations show that his approach can reduce response time significantly.

The primary difference between the two approaches is how the hints are obtained. In Baud et al.'s approach, hints are provided by the application programmers. In Korner's approach, hints are derived from off-line processing of file access activities logged earlier.

The Choices distributed file system uses both user hints and prior file access behavior information to choose caching strategies. Korner's approach appears impractical for real distributed file systems because it requires logging of file access activities, off-line processing of the logged data, and integrating the hints into the file system. Choices' adaptive file system improves on his approach by dynamically analyzing the access behavior of a file and encoding the observed behavior into the file's meta-data for later use.

Blaze and Alonso analyzed a week-long trace of file accesses at DEC/SRC[9]. They identified three properties of file access from this trace data. These properties are locality, inertia, and entropy. Locality says that files tend to be accessed from the same places from which they have been recently accessed. Inertia says that files tend to be accessed in the same manner as previous accesses. Entropy says that files become more permanent over time and tend to become read-only as they are read more often. They used these properties to devise a strategy for expiring cached files to reduce the number of messages required to maintain cache coherence. The main idea is to expire a file before it is about to be "recalled" by its server.

While the previous studies concentrated on all file accesses in the file system, we have studied the access behaviors of various individual scientific applications. We logged file access activities of supercomputing scientific applications running on the Cray-2 at the National Center for Supercomputing Applications[62]. Our analysis of the logged data shows that these applications had file access behaviors which tend to be consistent across runs. Each application appeared to have its own file access behavior "finger print". This consistency suggests that behaviors observed earlier is likely to be repeated.

Our observations, together with Blaze and Alonso's observations, indicate that file access behaviors are highly predictable. Predictability permits optimizations based on prior file access history. An adaptive file system can take advantage of this property by selecting strategies that take into account prior access history.
4.4 Client and Server Caches

Makaroff[71] investigated the influence of different client and server cache sizes on cache performance. His simulations of a LRU server cache and a LRU client cache, using trace data from time-sharing UNIX 4.2 BSD[58, 86] systems, show that file server cache performance is significantly different than that of client caches. He found LRU client caches to be very effective, i.e. low cache miss ratios can be achieved by small client caches. Server caches were less effective. The server cache miss ratios were much higher than the client cache miss ratios for similarly sized client caches. He also found that a large client cache diminishes the usefulness of the server cache. This is primarily because the temporal locality of references in the original reference stream has been filtered out by the client cache.

A later study by Willick et al. [125] used trace data obtained from a time sharing HP-UX system to drive simulations to compare the effectiveness of a temporal locality based replacement policy (LRU) with the effectiveness of frequency based replacement policies (LFU and FBR[92]). The simulations confirmed Makaroff's findings that LRU was very effective for client cache. For a reasonably sized cache, LRU’s cache miss ratios were close to the optimal cache miss ratios.\(^3\) They found LFU to be totally inappropriate for client caches. FBR is a hybrid replacement policy that attempts to capture the benefits of both LRU and LFU. Its replacement policy is primarily based on frequency counts, but it takes locality into account. It performed as well as LRU on the client.

Their simulations also show that LRU is inappropriate for server caches when client caches are present (because of locality filtering by the client caches). Frequency based replacement policies (LFU and FBR) were found to provide much better server cache hit ratios than LRU when client caches were present.

4.5 Network Characteristics

Partridge[81] argued that larger transfer sizes will always improve network throughput until the maximum throughput is attained. Comer et al. [23] characterized four different networks at the National Center for Supercomputing Applications. The networks were UltraNet (1 Gbps)[6], FDDI (100 Mbps)[94], HyperChannel (50 Mbps)[37], and Ethernet (10 Mbps)[35]. The observed characteristics of these networks support Partridge’s arguments. Comer et al. found that higher speed networks tend to achieve near maximum throughput at larger transfer sizes. For small transfers (less than 8 kilobytes), UltraNet had a lower throughput than Ethernet. Large transfers of 32 kilobytes or more were required to take advantage of UltraNet’s higher bandwidth. Similar studies of UltraNet by Clinger[22] and Baud et al. [5] found that transfer sizes of 128 kilobytes or more were required to obtain the best throughput. These

\(^2\)HP-UX is Hewlett-Packard's variant of UNIX.

\(^3\)The optimal miss ratio is obtained by using the optimal cache replacement policy[29]. It is the best cache miss ratio that can be achieved for a particular reference stream on a fixed size cache.
studies suggest that a file system should transfer data over a network using transfer sizes that are suitable for the network.

4.6 Other Cache Optimizations

Traditional file system caches deal with individual blocks and do I/O one block at a time. This results in low throughputs[72]. To improve throughput, Mekusick[72] and Laworska et al. [55] suggested using larger block sizes to reduce the number of I/O operations and increase transfer sizes. However, Peacock[82] and Mcvoy et al. [73] still found performance to be poor even when a larger block size is used. They investigated the benefits of coalescing I/O operations into clusters to increase transfer sizes. Their approach was to approximate the behavior of extent-like file systems by grouping I/O operations, that transfer data into multiple consecutive blocks of a file, into a single operation. Peacock found that this technique can achieve larger transfer sizes than simply using a larger block size. Mcvoy et al. found that a single coalesced I/O operation may take the place of 15-30 block I/O operations, resulting in significantly better sequential file access performance.

In systems with memory-mapped files[40, 119], virtual memory assists have been used to manage their main memory caches[10, 56, 59]. Braunstein et al. [12] investigated the effects of using virtual memory assists in managing file buffer caches in UNIX, which has a stream file system[67]. They found that virtual memory assists can speed up transfers to and from the cache by 20-50%. In addition, transfers to and from disks were also faster (10-25%).

Ruemmler and Wilkes[95] investigated the benefits of non-volatile disk caches. Their trace data of low level disk access activities on HP-UX systems show that most disk operations are writes;\textsuperscript{4} disk accesses are rarely sequential, and I/O activities can be very bursty. Using a simulator, they investigated the effects of write caching at the disk level. They found that a small non-volatile cache at each disk can effectively absorb write bursts;\textsuperscript{5} and thus allow writes to be serviced much faster.

Baker and Sullivan[3] investigated the benefits of using non-volatile memory to provide high-availability through fast recovery. They used non-volatile memory to cache BSD file system meta-data[58] and information about connections between the file server and its clients.\textsuperscript{6}

\textsuperscript{4}Reads can be effectively filtered out by a large cache. Meta-data writes cannot be cached.

\textsuperscript{5}This means that the non-volatile cache can effectively cache the data from the write bursts without writing to disk. The actual writes to disk can take place later when the disk is idle.

\textsuperscript{6}The Sprite distributed file system used in the study maintains stateful connections between the file server and its clients.
4.7 Flexible Virtual Memory and File Caches

The Mach operating system has a virtual memory management system that supports user-level pagers[87]. The primary focus of the Mach virtual memory management system is to provide a high degree of machine architecture independence and to permit fast message-passing through virtual memory manipulations such as copy-on-write. The Mach kernel manages the address spaces of each process and the physical memory on a host. Logically, the Mach virtual memory management system associates a physical memory cache with each memory object. A Mach memory object is repository for data, indexed by byte. Each memory object in Mach has an associated pager. A pager may be shared by more than one memory object. The kernel identifies the memory object, on which an operation is to be performed, for the pager. The Mach virtual memory management system provides a well-defined interface between physical memory caches (maintained by the kernel) and their pager(s). This interface is based on Mach’s message passing mechanism, known as ports, and allows pagers to be implemented in user-level processes. Pagers in Mach perform basic page-in and page-out operations. User-level pagers in Mach provide some flexibility for customizing these page-in and page-out operations.

The Spring operating system also has a flexible virtual memory management system[49, 50]. The Spring operating system uses this virtual memory management system to cache file data. The Spring virtual memory management system provides additional functionality that is beyond Mach’s virtual memory management system[50]. In particular, it allows different components of the memory management system to reside in different protection domains. It has well-defined object-oriented interfaces to each component of the virtual memory management system. Its well-defined interfaces and ability to divide its components among different protection domains provide additional flexibility. The main focus of the Spring virtual memory management system is to provide cache coherence support between caches.

The developers of the Mach and Spring virtual memory management system have yet to apply and take advantage of their available flexibility to improve file system performance by providing or utilizing different file caching strategies. It is unclear whether these virtual memory management systems can support multiple page replacement policies[57], and how they will support different pre-fetch and write-back policies.

In addition to the above systems, Cao et al. [18, 19] and Lee et al. [57] extended existing operating systems to implement application controlled cache replacement policies. Lee et al. extended the Mach (OSF/1) kernel to allow applications to provide their own specific page replacement policies. An application specifies a page replacement policy with a program written in the HiPEC command set. The HiPEC command set is like a CPU’s instruction set. Each HiPEC command consists of an operation and one or more operands. Examples of HiPEC commands include DeQueue to remove a page from a queue of pages, EnQueue to add a page to a queue of pages, LRU to pick the least recently used page from a
queue for replacement, *Flush* to flush a page to disk, *Ref* and *Mod* to test the referenced and modified status of a page. An application associates a HiPEC program with each region of virtual memory that requires a specialized page replacement policy. When a memory region with a specific replacement policy requires page replacement, the virtual memory system executes the region’s HiPEC program. The primary advantage of HiPEC programs are efficiency, security, and portability. It is efficient since the HiPEC instructions are decoded and executed by the kernel. Hence, there is no need to cross the kernel boundary. It is secure since kernel data structures are only accessed by the kernel-provided routines that implement the HiPEC command set. It is portable because details of the operating system internals are shielded from application programmers.

Cao et al. extended the Ultrix file system to support application controlled file cache block replacement. This system implements two-level replacement, a scheme that allows the kernel to control allocation among caches and the applications to control their own cache replacement. Its kernel allocation policy, known as LRU-SP, reduces the impact of foolish applications that choose foolish replacement policies. LRU-SP may override an application’s replacement choice if it considers the application to be foolish. An application is foolish if its chosen replacement policy results in more cache misses than LRU.

In this system, LRU-SP chooses the target process for block replacement and the target process chooses the block to replace. The target process controls its block replacement policy by assigning the files it accesses to various priority groups and associating different replacement policies with these priority groups. Each process has a set of priority groups and each priority group is associated with a page replacement policy. This policy can be either LRU or MRU. The kernel maintains a LRU or MRU queue for each priority group. This queue contains in-memory blocks that belong to files in its priority group. A process can move specific blocks from one of its queues to another by assigning “temporary” priorities to these blocks. The block to be replaced is obtained from the lowest non-empty priority queue. For example, if the lowest non-empty priority queue is a LRU queue, then the least recently used block in the queue is chosen for replacement.

Both Lee et al.’s and Cao et al.’s systems provide application control over page replacement for individual processes. These systems show that application controlled replacement can improve performance by reducing number of cache misses and reducing the number of I/O’s. Both systems require applications to be modified to take advantage of different cache replacement policies. Furthermore, they do not address applications that may span multiple processes. In contrast to their approaches, our approach addresses page replacement policies, as well as, other transfer policies, on a per file basis. We believe a per file approach can better address applications that span multiple processes.
4.8 Adaptive File Systems

The iPcrest file system is an adaptive file system developed at Princeton University[112]. It optimizes for better performance by clustering heavily accessed files near the center of a disk. It introduces the notion of file temperature. The temperature of a file is determined by the number of I/O operations performed on the file over a period of time divided by the file size. Hot files are more frequently and heavily accessed than cold files.

The iPcrest file system tracks the file temperature of each file and periodically reorganizes its disks by file temperature. It divides the space on each disk into regions. The regions are symmetrical about the center of the disk, and grow exponentially in size as one moves away from the center. Files are placed in each region according to their file temperatures, with the hottest files placed in the center region and the coolest files placed in the outermost regions. In other words, it adapts to changes in file temperatures and attempts to reduce the average seek distances to the hottest files. Staelin et al. ’s benchmark show that this optimization can improve average I/O time per operation by 7-17% depending on the size of the disk and the amount of dormant data.
Chapter 5

The Foundation

This chapter describes the foundation on which an adaptive distributed file system will be built. The *Choices* object-oriented operating system[15, 17, 97, 99] has been chosen as the foundation for the following reasons:

1. *Choices* is object-oriented and has all the benefits of object-orientation as discussed in Chapter 3. The most important of these benefits are ease of extensibility and customizability. These benefits permit rapid prototyping and compliment any flexible and adaptive system.

2. It has a rich set of beautifully engineered frameworks that are easy to refine and extend. These frameworks include a file system framework[16, 66, 67, 68, 70], a virtual memory management framework[98], a process management framework[96], a networking framework[127], a device management framework[34], and a message passing framework[44, 45].

3. It has a virtual memory management system framework that provides some flexibility. It can be refined to provide the features desired in a flexible file system caching service.1

To build an adaptive file system on *Choices*, the most relevant and important frameworks are the virtual memory management system framework and the file system framework. The virtual memory management system framework will be refined to support flexible file system caching. The file system framework will be extended to support distributed access. In addition, both frameworks will cooperate to track file access behavior and to select appropriate caching strategies.

This chapter presents a brief overview of the original *Choices* virtual memory management system framework and the original *Choices* file system file system framework.

---

1These features will be presented in the following chapters.
5.1 Conventions

The following typographic conventions will be used this thesis to identify classes, instances, and methods:

1. **Class names** -
   A word in *sans serif* font starting with an uppercase letter denotes a class name. For example, `Fruit` and `TropicalFruit` are class names.

2. **Instances** -
   A word or a string of consecutive words in *sans serif* font starting with a lowercase letter denotes an instance or instances of a class. For example, a fruit is an instance of the `Fruit` class. A `tropical fruit` is an instance of the `TropicalFruit` class and `tropical fruits` are multiple instances of the `TropicalFruit` class.

3. **Method names** -
   A word in fixed width *typewriter* font denotes a method name. For example, `texture` and `color0fSkin` are method names.

5.2 The Virtual Memory Management System Framework

The *Choices* virtual memory management system takes advantage of virtual memory hardware provided by most modern computer architectures[29, 84]. A virtual memory management system allows efficient and secure sharing of main memory (also known as physical memory) between applications, permits an application’s address space to exceed the amount of physical memory on a host. The most relevant feature the *Choices* virtual memory management system to this thesis is its ability to cache data from backing store in main memory.

The *Choices* virtual memory management system framework divides its objects into the following five layers:

1. **Virtual memory hardware abstraction** -
   Objects in this layer abstract and manage the virtual memory hardware.

2. **Physical memory management** -
   Objects in this layer manage the physical memory in a host.

3. **Logical memory management** -
   Objects in this layer represent logical memories. A logical memory is collection of related data aggregated together[97], such as a file or a disk. In * Choices*, logical memories represent backing stores.
4. **Caching support** -
   Objects in this layer support caching of backing store data in physical memory. They also manage
   the transfer of data between backing stores and physical memory.

5. **Virtual address space management** -
   Objects in this layer provide each application with a virtual address space.

   Figure 5.1 illustrates the original *Choices* virtual memory management system framework. Figure 5.2
   presents the class hierarchy of the *Choices* virtual memory management system. The rest of this
   section discusses the classes and objects in each layer of the *Choices* virtual memory management system.

### 5.2.1 Virtual Memory Hardware Abstraction

The abstract `AddressTranslation` class defines a hardware independent interface to the machine architec-
ture dependent virtual memory address translation hardware. Concrete subclasses of `AddressTranslation`
support different computer architectures. Instances of these concrete subclasses, known as address
translations, interact with the machine-dependent address translation hardware to map virtual memory
addresses to physical memory addresses.

The `AddressTranslation` class has the following methods:

1. `addMapping` to add virtual-to-physical address mappings at a given protection level;

2. `removeMapping` to remove mappings for a virtual address range; and

3. `changeProtection` to change protection for a virtual address range.

### 5.2.2 Physical Memory Management

An instance of the `PhysicallyAddressableUnit` class, known as a **physically addressable unit**, represents a
block of continuous physical memory. In most systems, a **physically addressable unit** represents a page
of physical memory. The `PhysicallyAddressableUnit` class has methods to return the physical address of
a **physically addressable unit**, to read and modify the attributes of a **physically addressable unit**. These
attributes include the `referenced` and `modified` status of the unit. **Address translations** update these
attributes in `removeMapping`. Each **physically addressable unit** also has a `next link` attribute which is used
to link several **physically addressable units** into a list. A **physical memory chain** encapsulates a linked list
of **physically addressable units**.

Each host has an instance of the `Store` class, known as the **store**. The **store** manages allocation of
**physically addressable units** on the host. It owns the collection of all unallocated **physically addressable
units**. Its clients invoke its `alloc` method with a size parameter to obtain **physically addressable units** from
Figure 5.1: The original *Choices* virtual memory management system framework.
Figure 5.2: The class hierarchy of the original *Choices* virtual memory management system.
the store. The store remembers all its clients and the number of physically addressable units each client possesses. The store's clients invoke the store’s dealloc method to return previously allocated physically addressable units to the store. In most cases, the store's clients are instances of the MemoryObjectCache class.

The store also maintains a system daemon, known as the system pager. When there are few unallocated physically addressable units on the host, the store activates the system pager. The system pager invokes the pageOut method of the store's clients. This causes the clients to page out physically addressable units that contain stale data and return these physically addressable units to the store.

5.2.3 Logical Memory Management

Instances of the MemoryObject class, known as memory objects, represents logical memories. A memory object in Choices represents a storage entity (or logical memory) that consists of a sequence of identically sized storage units. The attributes of a memory object include the size of each of its units, also known as its unit size, and its length in units. The MemoryObject class has methods to read and modify these attributes. It also has read and write methods to read and write the storage entity. Arguments to these methods include a starting unit number, number of units to read or write, and a virtual address. Subclasses of the MemoryObject class support different types of logical memories, such as disks and files.

5.2.4 Caching Support

An instance of the MemoryObjectCache class, known as a memory object cache, allows a memory object to be mapped into one or more continuous virtual address regions and allows portions of the memory object to be cached in physical memory. Methods of the MemoryObjectCache class operate on units which are the same size as the physically addressable unit. When a virtual address in one of these regions is accessed and the virtual address is not addressable, a page fault occurs and the cache method of a memory object cache is invoked. The cache method ensures that the unit associated with the faulting address is resident in physical memory and makes the unit addressable by the faulting process. If the unit is not resident in main memory, the cache method obtains free memory from the store and initializes the memory by invoking the read method of the memory object. The cache method makes the unit addressable by adding the appropriate virtual-to-physical address mappings to the faulting domain’s address translation. The mcache method is similar to the cache method except it operates on more than one unit.

The MemoryObjectCache class has a pageOut method that is invoked by the store. This method is responsible for removing stale data from the cache and returning memory occupied by the stale data to the store. Subclasses of the MemoryObjectCache class support different caching schemes and

---

2The definition of stale data depends on the client's cache replacement policy.
provide some flexibility. For example, instances of the `SpecialMemoryObjectCache` class are used to cache kernel code and data. They never page out. Their `pageOut` methods do nothing. Instances of the `PagedMemoryObjectCache` class will page out stale data. Subclasses of the `PagedMemoryObjectCache` class implement different cache replacement policies.

### 5.2.5 Virtual Address Space Management

An instance of the `Domain` class, known as a `domain`, manages the virtual address space of an application. It maintains a collection of bindings between `memory objects` and virtual address regions. Each `domain` has an associated `address translation` that manages the virtual-to-physical address mappings of the `domain`. The `Domain` class has a `add` method to map a `memory object` to a virtual address region, i.e. to add a binding to the `domain`. It also has a `remove` method that is used to unmap a `memory object` from a virtual address region, i.e. to remove a binding from the `domain`.

A `memory object` may be mapped into more than one `domain`. When a page fault occurs, the exception handler calls the `repairFault` method of the faulting `domain`. The faulting `domain` is the `domain` whose `address translation` is active when the fault occurred. The `repairFault` method examines the `domain`’s collection of bindings to determine which `memory object` is mapped at the faulting address. Then, it invokes the `cache` method of the `memory object cache` associated with the `memory object` mapped at the faulting address. As discussed earlier, this method makes the faulting address addressable.

The `Domain` class also has a `makeResident` method and a `constructChain` method. These methods take a virtual address offset and a length in bytes as their arguments. The `makeResident` method makes sure that the virtual address region described by its arguments is resident and addressable. The `constructChain` method is mostly used by I/O device drivers. It makes sure that the virtual address region described by its arguments is resident and locked so that it cannot be selected for replacement. It returns a `physical memory chain`. This `physical memory chain` encapsulates the list of `physically addressable` units that have been locked. Destroying this `physical memory chain` releases the acquired lock.

### 5.3 The File System Framework

The `Choices` file system framework divides its objects into the following three layers:

1. Storage management -
   Objects in this layer represent and help implement logical memories that are persistent.

2. Persistent Object -
   Objects in this layer encapsulate and interpret the data stored in logical memories.
3. Application Interface -

Objects in this layer interact with objects in the storage management layer and persistent object layer. They provide high-level interfaces that are useful to applications.

Figure 5.3 describes the class hierarchy of the original Choices file system. Figure 5.4 shows how objects of various classes combine to implement a BSD file system. Figure 5.5 illustrates the corresponding entities in the BSD file system represented by objects in Figure 5.4. Each of the three different symbols in Figure 5.4 represents objects in each of the three layers of the file system. Rectangles represent objects in the storage management layer. Ovals represent objects in the persistent object layer. Hexagons represent objects in the application interface layer.

5.3.1 Storage Management

The most important class of objects in this layer belong to the PersistentMemoryObject class. The PersistentMemoryObject class is a subclass of the MemoryObject class. Its instances are represented by rectangles with solid outlines in Figure 5.4. A persistent memory object represents a persistent logical memory and it has all the properties of a memory object. Like a memory object, a persistent memory object has a unit size and a length in number of units. A persistent memory object is essentially an uninterpreted sequence of persistent unit-sized blocks, i.e., its read and write methods retrieve and store unit-sized blocks of persistent data. Different subclasses of the PersistentMemoryObject class support different kinds of persistent memory objects. All PersistentMemoryObject subclasses have the same public\(^3\) interface.

The Disk class is a subclass of the PersistentMemoryObject class. It represents physical disk devices. The FileObject class represents a persistent memory object whose persistent storage is provided by another persistent memory object. The MemoryObjectPartition class, the ArEntry class, and the UNIXinode abstract class are subclasses of the FileObject class. Concrete subclasses of the UNIXinode class include the BSDinode class and the SVIDnode class (not illustrated in Figure 5.4). A BSD inode\(^4\) represents a BSD file system inode and its storage is provided by a disk partition. A disk partition represents a partition on a disk drive and its storage is provided by a disk. An ar entry represents storage occupied by an entry in an ar archive and its storage is provided by an UNIX inode\(^5\).

Other classes in this layer help to implement specific kinds of persistent memory objects. For example, the UNIXinodeIndirect class compliments the UNIXinode class. Instances of the UNIXinodeIndirect class cache indirect blocks\([1, 58]\) belonging to UNIX inodes.

---

\(^3\) As in public methods in C++\([34]\).

\(^4\) A BSD inode is an instance of the BSDinode class.

\(^5\) An UNIX inode is an instance of the UNIXinode class.
Figure 5.3: The class hierarchy of the original *Choices* file system.
Figure 5.4: The original Choices file system framework.
Figure 5.5: The corresponding entities in the BSD file system.
5.3.2 Persistent Object

The most important class of objects in this layer belong to the PersistentObject class. Its instances are represented by ovals with solid outlines in Figure 5.4. Other classes in this layer compliment specific subclasses of the PersistentObject class.

An instance of the PersistentObject class is a persistent object. A persistent object is always associated with a persistent memory object. A persistent object encapsulates and interprets persistent data stored in its persistent memory object. It invokes its persistent memory object’s read and write methods to access persistent data stored in its persistent memory object. There are three main types of persistent objects or subclasses of the PersistentObject class. They are the MemoryObjectContainer class, the MemoryObjectDictionary class, and the PersistentArray class.

An instance of the MemoryObjectContainer class, known as a memory object container, contains an indexed collection of persistent memory objects. It has open, create, and close methods to open, create, and close member (or contained) persistent memory objects. For example, an instance of the DiskContainer class contains a collection of disk partitions, and an instance of the BSDContainer class contains a collection of BSD inodes.

An instance of the MemoryObjectDictionary class, known as a memory object dictionary, implements a naming service. A memory object dictionary contains a collection of string key to number associations. This number is an index into a memory object container’s collection of persistent memory objects and it is used to open the persistent memory object associated with a particular key. Methods of the MemoryObjectDictionary class include open to open a persistent memory object associated with a given key; keys to list all the keys in a memory object dictionary; associations to enumerate key to index number associations; add to associate a persistent memory object with a new key; and remove to delete an association. For example, the BSDDirectory is a subclass of the MemoryObjectDictionary class. Each of its instances represent a BSD file system directory.

The PersistentArray class has methods that allow a persistent memory object to be accessed like an array. Its methods include at to retrieve data and atPut to store data.

Some memory object containers may contain a dynamic number of persistent memory objects or they may contain persistent memory objects whose size may change. The BlockAllocator class compliments these memory object containers by defining a protocol for allocating and deallocating units from the container’s underlaying persistent memory object to the contained persistent memory objects. Subclasses of the BlockAllocator class compliment different subclasses of the MemoryObjectContainer class. Each subclass of the BlockAllocator class implements an allocation policy that is specific to its associated MemoryObjectContainer class. For example, the BSDAllocator class compliments the BSDContainer class and implements the BSD file system block allocation policy.
5.3.3 Applications Interface

Objects and classes in this layer define addition interfaces useful to applications. The FileSystemInterface class and the RecordStream classes are the most popular classes in this layer.

The FileSystemInterface class defines a per-process interface to the file system. A file system interface keeps track of the current directory and maintains a name space for referencing files in the system. It has methods to open, create, and unlink\(^6\) a file given a path name. The name space maintained by a file system interface can be modified using mount and umount methods.

The RecordStream class defines an interface for stream access to data held in a persistent memory object. It introduces a notion of current file position. The RecordStream class is an abstract class. Its subclasses supports different stream protocols. For example, the PersistentArrayStream concrete subclass implements a byte stream. The FixedRecordStream concrete subclass implements a stream of fixed-sized records. The VariableRecordStream concrete subclass implements a stream of variable-sized records.

5.4 Status and Summary

In summary, Choices provides two different object-oriented frameworks for the virtual memory management system and the file system. Although these frameworks are complete for each of their respective functionality, they have not been integrated to support file system caching.

The original file system does not support caching. Even though a persistent memory object can be mapped into a domain and cached by a memory object cache, the file system does not take advantage of this capability. The file system uses a persistent memory object's read and write methods to retrieve data from, and store data into the persistent memory object directly. The primary problem is that the virtual memory management system framework was designed to support virtual memory paging only. It does not address the needs of the file system. For example, the memory object caches in the virtual memory management system assume that the size of a memory object does not change once it is mapped into a domain. This is not a realistic assumption for a file system.

The following chapter presents how the original virtual memory management system framework and the file system framework have been integrated and refined to implement flexible file system caching.

\(^6\)The unlink method removes an association between a name and a file.
Chapter 6

Flexible Caching

This chapter describes the goals and the architecture of a flexible file system caching service. In addition, it presents an implementation of this architecture in Choices. This implementation extends and refines the Choices virtual memory management system framework and file system framework.

6.1 Goals

The design goals for a flexible file system caching service include:

1. Flexibility -
   A flexible file system caching service should support numerous different caching strategies. These strategies should be able to coexist on the same host.

2. Customizability -
   A flexible file system caching service should allow the computing environment to select and use caching strategies that best suits its requirements.

3. Extensibility -
   A flexible file system caching service should allow new caching strategies to be added easily and in an orderly manner.

4. High performance -
   A flexible file system caching service should perform better than conventional non-flexible file system caching services.

6.2 Architecture

38
Figure 6.1: The architecture of a flexible file caching service.
This section describes the architecture of a flexible file system caching service that meets the above goals. Figure 6.1 presents an overview of this architecture. In this figure, File is the in-memory representative of a file on the host. It is similar to a vnode[101] in SunOS\(^1\) and a file’s object descriptor in Sprite[123]. It may represent either a local file or a remote file.

### 6.2.1 Host Memory Manager

Each host has a single Host Memory Manager. A Host Memory Manager manages the allocation of main memory on the host. It distributes cache memory to each cached file on the host. The memory allocated to each file is managed by the file’s File Cache. In other words, the Host Memory Manager distributes available cache memory to the File Caches. When the host is about to run out of memory, the Host Memory Manager will ask some or all of the File Caches to return some of the memory allocated to them.

Different hosts can have different Host Memory Managers which implement different main memory allocation policies. These policies determine how main memory is distributed to the various File Caches on the host. One possible policy is to allocate memory to each File Cache in proportion to the frequency of new memory allocation requests from the File Cache. Other possible policies are to allocate an equal amount of memory to each File Cache on the host or to allocate memory according to the LRU-SP policy[18].

### 6.2.2 File Cache

Each file cached on a host has a File Cache. A File Cache consists of modules that determine how data from its file is cached in main memory, the transfer strategy used to move data in and out of main memory, and the cache replacement policy. Different kinds of modules provide different functionalities that are required of a File Cache.

A File Cache has attributes. These attributes include the size of the File Cache, the number of resident units in the File Cache, the number of reads into the File Cache, the number of writes out of the File Cache, and the cache unit size. The cache unit size of a File Cache is equivalent to the block size in other distributed file systems. Data transfers will always take place in sizes which are multiples of the cache unit size. A large cache unit size forces large transfers.

### 6.2.3 Cache Memory Manager

A Cache Memory Manager performs policy-free general “housekeeping” functions required by all File Caches to keep track of memory that has been allocated by a Host Memory Manager to its File Cache.

\(^1\)SunOS is Sun Microsystems Inc.’s variant of UNIX.
In particular, it keeps track of which part of its file is cached, and where it is cached in main memory. It interacts with the Host Memory Manager to obtain more memory for the cache and to return allocated memory to the Host Memory Manager. It always allocates and frees memory in multiples of its cache unit size.\textsuperscript{2}

### 6.2.4 Migration Policy

A Migration Policy module is policy-rich. It defines the data transfer strategies and the cache replacement policy of a File Cache. These data transfer strategies determines when and how data from the file is transferred in and out of the File Cache. Prefetch strategies, such as \textit{fetch on-demand, pre-fetch next unit}, and \textit{pre-fetch all on initial access}, are data transfer strategies. Write strategies, such as write-through, delayed-write, and write-on-close\textsuperscript{60}, are also data transfer strategies.

A Migration Policy module may implement a \textit{group} or a \textit{local} cache replacement policy. A \textit{local} cache replacement policy selects unit(s) for replacement from within a single File Cache. A \textit{group} cache replacement policy selects unit(s) for replacement from all the File Caches on single host that are using the same or a related family of cache replacement policies. Although each File Cache has its own Migration Policy module, group cache replacement policies can be supported because multiple Migration Policy modules can collaborate to define a group policy. For example, a \textit{global} LRU cache replacement policy\textsuperscript{3} may be implemented by attaching the same kind of group Migration Policy modules to all File Caches on the host. These modules cooperate to determine which unit is the least recently used unit in their File Caches.

Different Migration Policy modules implement different data transfer strategies and replacement policies. For example, a Migration Policy module optimized for large sequential file accesses may implement the MRU cache replacement policy and the \textit{pre-fetch next unit} data transfer strategy. Another Migration Policy module optimized for small files may implement the LRU cache replacement policy and the \textit{pre-fetch all on initial access} data transfer strategy.

The main reason for providing different modules in a File Cache for policy-free versus policy-rich functionality is to permit the policy-free modules to remain unchanged when the policy-rich modules are replaced or substituted to adapt to different environments. This separation allows a File Cache to change its cache replacement policy and data transfer strategy by replacing its Migration Policy module without loosing track of the memory allocated to the File Cache.

\textsuperscript{2}This is the cache unit size of the Cache Memory Manager’s File Cache.

\textsuperscript{3}A global cache replacement policy selects unit(s) for replacement from all the allocated units in the host.
6.2.5 Secondary Cache

An optional Secondary Cache allows data from a file to be cached in tertiary storage. It caches data in tertiary storage on behalf of its File Cache. It allows a tertiary storage system to act like a backing store[98] for the File Cache. Different types of Secondary Cache(s) support different types of tertiary storage systems. Multiple Secondary Caches may be "stacked" to mirror the memory hierarchy of the host.

Figure 6.2 illustrates how different kinds of Secondary Caches may be stacked to cache file data using different memory devices. In this example, the original file resides on a slow optical disk such as a CD-ROM. The Disk Secondary Cache caches the file's data in a local disk. The Solid State Disk Secondary Cache caches the file's data in a faster solid state memory device. Finally, the the File Cache caches the file's data in primary memory. Each of these caches may cache the entire file or only part of the file.

The Secondary Cache allows the file system caching service to exploit the memory hierarchy of a host. A File Cache only has to attach the appropriate Secondary Cache(s) to itself to use the available tertiary storage systems on the host for caching. A good implementation of the architecture should allow Secondary Cache(s) to be attached and detached at any time. For efficiency, it should also allow the File Cache to page data directly from the file into main memory without copying the data into the Secondary Cache(s) first.

6.2.6 Distributed Caching Example

Figure 6.3 presents an example of how this architecture works in a distributed system. In this example, the client invokes the server to open a file. This file resides on a CD-ROM in the server. First, the server opens the file by creating an in-memory representative for the file, i.e. the File on the server. Then, the server attaches a File Cache to the File. The File Cache allows data from the file to be cached in main memory. In addition, the server also attaches a Secondary Cache to the File Cache. This particular Secondary Cache caches the file's data in a disk that is faster than the CD-ROM. This capability enables faster subsequent accesses to the contents of the CD-ROM resident file by the same or different clients. This Secondary Cache is optional and is used primarily to improve performance.

The Migration Policy module of the File Cache on the server determines how the file's data is transferred from the CD-ROM into the server's main memory and into the disk cache. One such module may copy the entire file into the disk cache on initial access. A different Migration Policy module may bypass the disk cache and page data directly into the server's memory on demand. It will write to the disk cache only when it has to page data out to free memory.
Figure 6.2: Stacking secondary cache
Figure 6.3: Distributed file system caching.
After the server has opened the file successfully, it sends a reply message containing a token or an identifier which represents the file to the client. When the client receives the reply from the server, it encapsulates the returned token or identifier in the Remote File. The Remote File represents the CD-ROM resident file on the client. Then, the client attaches a File Cache to the Remote File so that it can cache remote file data in its main memory. The client also attaches a Secondary Cache to the File Cache. This particular Secondary Cache allows the client to cache remote file data in a local disk. It increases the amount of memory available on the client for caching remote file data. The client File Cache’s Migration Policy module determines how data from the remote file is transferred from the file server into the client’s main memory and into the client’s local disk cache.

6.2.7 Summary

This architecture is flexible because each file can have a different caching strategy. In addition, the caching strategy used on the server need not be the same as the caching strategy used on the client. It is customizable because the application and the distributed operating system can “mix and match” components and modules of different types to define caching strategies that best suit the needs and requirements of the distributed computing environment. In addition, it separates the policy-rich components from the policy-free components. For example, it allows the migration policy associated with a file to change without losing track of the memory that has been allocated to cache the file’s data. Furthermore, it is also extensible since new components that implement new policies can be added easily. It is expected to perform well because it can exploit host, network, and application file access characteristics with customized caching strategies. For example, the architecture can address network latency problems with large cache unit sizes, client disk caching, and suitable pre-fetch strategies. It also permits stacking of secondary caches to take advantage of the memory hierarchy on a host.

6.3 The Flexible File System Caching Framework

This section describes a framework that implements the flexible file system caching architecture described in the previous section. This framework is known as the flexible file system caching framework. It integrates and extends the original Choices virtual memory management system framework and file system framework.

This section describes the flexible file system caching framework. First, it discusses the motivation for extending the original Choices virtual memory management framework to support flexible file system caching. Then, it presents the classes in this framework and some implementation details. Implementation details discussed include:

- How different migration policies are supported.
• How secondary caches are implemented.

• How a hierarchy of file containers (as described in Section 5.3) affect caching.

### 6.3.1 Motivation

It is desirable to extend the *Choices* virtual memory management system framework to support flexible file system caching because the *Choices* virtual memory management system framework is already reasonably close to the proposed architecture. The advantages of object-oriented programming make rapid evolution of this framework to support flexible file system caching easy.

In addition, operating systems that use their virtual memory management systems to cache file system data have the following advantages:

1. The ability to dynamically vary main memory allocation between application programs\(^4\) and the file system. SunOS has a similar capability[40]. Baker et al.[4] found this feature to be useful in Sprite.

2. The ability to take advantage of virtual memory hardware to improve performance[12].

3. The ability to leverage distributed virtual memory to maintain cache coherence[61].\(^5\)

### 6.3.2 Overview

Figure 6.4 illustrates the *Choices* flexible file system caching framework. Figure 6.5 presents the class hierarchy of this framework. The following maps the different components of the flexible file system caching architecture to objects in this framework:

1. **File** -
   
   It maps to a memory object. To be more specific, it should map to a persistent memory object implemented by the file system, such as an UNIX inode.

2. **File Cache** -
   
   It maps to an instance of the MemoryObjectCache class. A memory object cache in the revised framework has component objects that manage memory and determine the memory object cache’s migration policy.

3. **Cache Memory Manager** -
   
   It maps to two new classes of objects. The first is the Map class. A map keeps track of memory

\(^4\) An application program uses main memory to store its executable code, stack, and data.

\(^5\) Many thanks to Aamod Sane for his effort in integrating his distributed virtual memory protocols into the flexible file system caching framework.
Figure 6.4: The Choices flexible file system caching framework.
Figure 6.5: The Choices flexible file system caching class hierarchy. New or significantly revised classes are shown in italics.
allocated to its memory object cache. The second is the TranslationMap class. A translation map keeps track of the virtual-to-physical address mappings associated with units in its memory object cache.

4. Migration Policy -

It maps to an instance of the new MigrationPolicy class. A migration policy belongs to a memory object cache. It implements the memory object cache's cache replacement policy and data transfer strategy.

5. Secondary Cache -

It maps to an instance of the new SecondaryCache class. A secondary cache can attach itself to other secondary caches to form a stack of secondary caches (as described in Section 6.2.5). Each memory object cache has such a stack.

6. Host Memory Manager -

It maps to a store. A store allocates physical memory to the memory object caches on a host.

The remainder of this section describes each of these classes in further detail. In addition, it addresses some of the shortcomings of the original system such as concurrency control.

6.3.3 MemoryObjectCache

A memory object cache corresponds to a File Cache in the flexible file system caching architecture. It has several component objects of different classes. These objects provide different functionalities. A memory object cache is the central coordinator for activities involving the cache, such as paging in a unit, flushing a page to disk, and mapping a unit into an address space. It is like a traffic controller that directs the flow of data to and from the cache, as well as, between its component objects.

The original MemoryObjectCache class implements all the requirements of a memory object cache, i.e. a monolithic memory object cache provides all the functionality required of a memory object cache. The new MemoryObjectCache class does not implement all the required functionality. Some of the original functionality has been delegated to the Map class, the TranslationMap class, and the MigrationPolicy class. In the new framework, each memory object cache has three component objects; a map, a translation map, and a migration policy. Its map and translation map implement the functionality of the Cache Memory Manager. The migration policy implements the functionality of the Migration Policy module. The new MemoryObjectCache class has a new setMigrationPolicy. This method changes the migration policy of a memory object cache.

Since a memory object cache's migration policy is determined by an independent object, it can change its migration policy with greater ease than the original monolithic memory object cache. In the original
frameworks, the only way to change a memory object cache's migration policy is to destroy the memory object cache and replace it with another memory object cache. This is inefficient because destroying a memory object cache also destroys the data structures that keep track of the memory allocated to the memory object cache. Hence, data cached by the memory object cache must be flushed before the memory object cache can be destroyed.

In addition to being a coordinator, a memory object cache is also a repository for attributes that are necessary and useful to its component objects and to the framework. A memory object cache has the following attributes:

1. **cache unit size** -
   A memory object cache and its component objects operate in units of this size. For example, the lengths of all I/O operations to and from the memory object cache are in multiples of this size. Hence, it determines the memory object cache's smallest transfer size. Memory allocation and deallocation requests also take place in multiples of this size. The original memory object cache has a cache unit size that is always the same as the size of a physically addressable unit. The new memory object cache may have a cache unit size that is any power-of-two multiple of the physically addressable unit size. The power-of-two restriction allows shifts to be used instead of multiplications and divisions. Shift instructions are usually faster on most architectures.\(^6\)

2. **cache size** -
   It is the maximum number of units that the memory object cache can manage.

3. **I/O window size** -
   The I/O window size is the size of an I/O window for a memory object cache. An I/O window is a locked memory object view that is mapped into the I/O domain. It provides a protected virtual address region that is dedicated to I/O. (Section 6.3.11.1 describes I/O windows and the I/O domain in greater detail.)

4. **various statistics** -
   These attributes record the number of memory faults, the number and volume of various kinds of data movements to and from the cache.

The new memory object cache preserves all the methods of the original memory object cache. However, it delegates certain method invocations to its component objects. For example, a memory object cache delegates a pageOut invocation to its migration policy. In addition, the following methods have been added to support file system caching:

\(^6\)This restriction may be relaxed if necessary to use multiplications and divisions.
1. **cacheRead** and **cacheWrite**.  
The `read` and `write` methods of a memory object invoke these methods. The memory object may use these methods to retrieve data from and store data into its memory object cache. The arguments to these methods include a virtual address, a byte offset and a length. For the `read` method, a memory object cache copies the specified length bytes of data starting from the specified byte offset of its memory object to a buffer specified by the virtual address. In doing so, the memory object cache caches units from its memory object in main memory. For the `write` method, a memory object cache copies the specified length bytes of data from a buffer specified by the virtual address into its cache starting at the specified byte.

Some systems do not provide dedicated methods to copy data in and out of virtual memory based caches. In order to copy data out of the cache, these systems map the desired source units into a virtual address region and use standard data access machine instructions to copy the source data from the mapped virtual address region. This is commonly known as memory-mapped I/O. It causes a loss of semantic information that is necessary for some optimizations. For example, if the original access is to read ten consecutive units of data, memory-mapped I/O reduces this read operation to individual data access instructions. This may cause ten individual page faults and result in ten separate I/O operations. If the virtual memory management system knows about the intent to read ten units, it may obtain all ten units with a single I/O operation. Another example is writing ten consecutive units. If all ten units are overwritten completely, the virtual memory management system need not page in these ten units before writing over them.

The lost semantic information may also contain useful hints. For example, one of the LRU replacement policies in *Choices* treats units brought into the cache with one I/O operation as a cluster, and it frees memory by paging out the least recently used cluster. In other words, this replacement policy treats consecutive units that are paged in together as a single “location”. Although memory-mapped I/O may be used to transfer data in and out of a memory object cache, the `cacheRead` and `cacheWrite` methods are necessary since they provide additional semantic information.

2. **setNumberOfUnits**.  
The original frameworks assume that the size of a memory object does not change once it has been added to a domain. As a result, the size of a memory object cache is fixed at creation. This is a problem because the size of some persistent memory objects, such as the UNIX inode, is likely to change. This method allows the size of a memory object cache to change. It also propagates a size change request to all the component objects of the memory object cache. This allows each of these components to adjust their data structures accordingly. For example, a component object may free some memory if the size of the memory object has been reduced.
The original MemoryObjectCache class has two I/O methods to move data between main memory and its memory object. These methods are fromMemoryObject and toMemoryObject. With the introduction of secondary caches, these methods are no longer sufficient. A new memory object cache does all its I/O through its top-most secondary cache. The top-most secondary cache is the secondary cache that is directly attached to the memory object cache. The new and revised I/O methods of the new MemoryObjectCache class are:

1. fromSecondaryCache -
   It serves the same function as the original fromMemoryObject method. A memory object cache uses this method to read data into main memory. This method invokes the read method of the top-most secondary cache.

2. toSecondaryCache -
   A memory object cache invokes this method before it reclaims main memory occupied by cached data. It coordinates the transfer of affected (usually dirty) data out of main memory. It allows any attached secondary cache(s) to cache the data that is about to be removed from main memory. This method eventually invokes the write method of the top-most secondary cache. An argument of the write method receives the referenced and modified status of each of the affected units. The modified status is required to maintain cache consistency. It indicates whether a unit has been altered since it had been brought into main memory. For example, if none of the secondary caches chooses to cache a modified unit, then this modified unit must be written to backing store memory object. Similarly, a modified unit held in a secondary cache needs to be written to the backing store memory object when the secondary cache is flushed. If a unit is not modified, a secondary cache that already has the unit cached can avoid writing the same data into the secondary cache again.

3. toMemoryObject -
   The toMemoryObject method directs the commit of in-memory cached units to the backing store memory object. Typically, this method is invoked the synchronize method of the MemoryObjectCache class. The synchronize method commits cached data that are in both main memory and the memory object cache’s secondary cache(s).

Units committed by this method must be written to persistent storage. In other words, the data in committed units must be able to survive system failures. This method invokes the rawWrite method of the top-most secondary cache. Like the toSecondaryCache method, it provides the referenced status and the modified status of each of the affected units to the rawWrite method. It also allows any attached secondary cache(s) to cache data being committed. In most cases, modified units are written to the memory object.
In addition to the above methods, the revised `synchronize` method has to interact with the memory object cache’s top-most secondary cache. As mentioned earlier, this method commits data cached in main memory and the memory object cache’s secondary caches to the backing store memory object. In addition to invoking the `toMemoryObject` method to commit data that has been cached in main memory, it has to invoke the top-most secondary cache’s `synchronize` method to commit data cached in the memory object cache's secondary caches. Each secondary cache always propagates a `synchronize` message to the next secondary cache on the memory object cache's secondary cache stack.

Similarly, the revised `invalidate` method has to interact with the memory object cache’s top-most secondary cache. This method invalidates units specified by its arguments. It invalidates both units cached in main memory and units cached by its secondary caches. It is intended to be used by cache coherence protocols for invalidating stale data. Like the `synchronize` method, it invokes the `invalidate` method of the memory object cache's top-most secondary cache, and each secondary cache on the memory object cache’s secondary cache stack propagates this message down the stack.

### 6.3.4 Map

The Map class and its subclasses provide part of the functionality of the Cache Memory Manager. Each memory object cache has exactly one map. A map keeps track of memory allocated to a memory object cache.

A map maintains a database that remembers which units are cached in main memory and the physically addressable units that are used to cache these units. The Map class has methods to access, query, and modify this database. Logically, this database is an array of \( n \) elements, where \( n \) is the size of the memory object in cache units. Each array element in this array represents a cache unit and contains an ordered list of physically addressable units. If a unit is not cached in main memory, the ordered list associated with the unit is empty. Otherwise, the ordered list contains the physically addressable units that cache data belonging to the unit.

The cache unit size determines the number of physically addressable units in each list.\(^7\) If the cache unit size is the same as the size of a physically addressable unit,\(^8\) then each ordered list contains either zero or one physically addressable unit. If the cache unit size exceeds the size of a physically addressable unit then each ordered list contains zero or more than one physically addressable units.

Since a cached unit may be cached in more than one physically addressable unit, the Map class also defines utility methods that obtain the referenced status and the modified status of each unit. Typically, these utility methods traverse the ordered list of physically addressable units associated with each unit to determine the unit’s status. An unit is valid if it is cached in main memory, i.e. its ordered list is

\(^7\)This number is the cache unit size divided by the size of each physically addressable unit.

\(^8\)All physically addressable units on a host have the same size.
non-empty. Otherwise it is invalid. An unit has been referenced if it is valid and any of its associated physically addressable units has been referenced. Similarly, an unit has been modified if it is valid and any of its associated physically addressable units has been modified.

The Map class defines the following methods:

1. residentSize -
   This method returns the number of units cached in main memory.

2. setSize -
   This method sets the maximum number of units that the database can manage. The memory object cache invokes this method when the size of its backing store memory object changes.

3. pauList -
   This method returns an ordered list of physically addressable units associated with the units specified by its arguments.

4. addPauList -
   This method associates physically addressable units with cached units. In other words, it makes these units valid. Its arguments include a range of units and an ordered list of physically addressable units.

5. findAndRemove -
   This method finds valid units within the specified range of units. It removes the physically addressable units associated with these units from the map. Then, it returns the removed physically addressable units as an ordered list. Typically, these physically addressable units are returned to the store as free memory.

6. exists, modified, and referenced -
   These methods determine if the specified unit is valid, modified, and referenced, respectively.

7. setReferenced, setModified, and setReferencedModified -
   These methods set the specified units' referenced, modified, referenced and modified status to the specified boolean value, respectively.

8. findNextExist, findNextNonExist, and findNextModified -
   These methods find, within the range of specified units, the first consecutive sequence of valid, invalid, and modified units, respectively. These methods are not strictly necessary since it is possible to simulate their functionality using other methods provided by the class. However, the Map class defines these methods because they permit examination of an instance's data structure at a higher semantic level, which provides opportunities for data structure specific optimizations for these commonly used operations.

54
Several subclasses of the Map class have been implemented. Each subclass uses a different data structure to maintain its database. For example, the TableMap class uses a linear array to maintain this database. Each element in the array represents a physically addressable unit. In other words, each cache unit size number of elements represent a cache unit. A table map may increase the size of its array in response to a setSize message. It does so by allocating a new and larger array, copying the data in the old array to the new array, and releasing the old array. The ConstantSizeTableMap class is a subclass of the TableMap class. Its instances are associated with memory objects whose sizes do not change. The MultiLevelTableMap class is a subclass of the Map class. It maintains its database using a tree. Its instances are associated with memory objects that change their sizes frequently or that are sparsely cached in main memory.

The PremappedMap class is a special subclass of the Map class. It does not use a data structure to implement the database. It is associated with a specific kind of memory object that represent physical memories whose virtual addresses are always the same as their physical addresses. A premapped map can always locate the physically addressable units associated with a given cache unit once the base address of its backing memory object is known. It can either obtain this base address from its special backing memory object or from its memory object cache. It locates the physically addressable units associated with a cache unit by computing the physical addresses of these units and obtaining the corresponding physically addressable units for these addresses from the kernel.

6.3.5 TranslationMap

The TranslationMap class and its subclasses provide the Cache Memory Manager’s remaining functionality that is not provided by the Map class. Each memory object cache has a translation map. Since a Choices memory object can be mapped into multiple domains, virtual-to-physical address mappings for the memory object may exist in multiple address translations. The translation map interacts with these address translations. It maintains a database of which units have mappings in which address translations. It consults this database when its memory object cache directs it to unmap one or more units from all address translations. This normally occurs before the memory object cache is about to page these units out.

The TranslationMap class defines the following methods:

1. translatable -

   Its arguments include a range of cache units, a virtual memory protection level, an address translation, and a virtual address. It makes the range of cache units addressable at the specified base address of the specified address translation. It also associates the desired protection level with these
units in the address translation. It may add or update an entry in the translation map's database to remember any new virtual-to-physical address mappings.

2. untranslatable -

Its arguments include a range of cache units, an address translation, and a virtual address. The latter two arguments are optional. This method consults the translation map's database to remove virtual-to-physical address mappings that have been established earlier. If only a range of cache units is specified, it makes all virtual addresses associated with these units unaddressable. If an optional address translation is specified, then it makes virtual addresses associated with these units in the specified address translation unaddressable. Similarly, if a virtual address is also specified, it makes the range of units associated with the specified virtual address in the specified address translation unaddressable.\footnote{This is necessary since an unit can be mapped into more than one virtual address in any address translation.}

Like the Map class, different subclasses of the TranslationMap class use different data structures to maintain their databases.

6.3.6 SecondaryCache

The SecondaryCache class implements the Secondary Cache described in Section 6.3.6. As mentioned earlier, a secondary cache caches data from the backing store memory object in a tertiary storage device. Typically, this storage device has better performance characteristics than the storage device on which the backing store memory object is stored.

Furthermore, secondary caches can be stacked. A memory object cache only interacts with its top-most secondary cache. Each secondary cache interacts with its parent and its child. Its parent is the secondary cache above itself on the stack. Its child is the secondary cache below itself on the stack. The top-most secondary cache has no parent and the bottom-most secondary cache has no child. The SecondaryCache class defines the protocol used by a secondary cache to coordinate caching activities between its parent, its child, and itself.

The methods of the SecondaryCache class include:

1. read -

The fromSecondaryCache method of the MemoryObjectCache class invokes this method to read data into main memory. Its arguments include a starting unit, the number of units requested, and the starting address of the buffer that will receive the incoming data. If the requested units are available in the cache of the invoked secondary cache, the invoked secondary cache should satisfy the read request with data from its cache.
If the requested units are not cached by the invoked secondary cache, then the action taken depends on the implementation of the invoked secondary cache. In most implementations, the invoked secondary cache delegates the read invocation to its child by invoking the child’s read method. This allows the child to write the requested data directly into main memory. Upon return from the child’s read invocation, the secondary cache may choose to cache the requested data before returning to its caller.

This method’s interface allows a secondary cache with the requested units to write the requested data directly into main memory. In other words, no additional buffers are required. If a higher secondary cache needs to access data written to memory by a lower secondary cache, it can always locate this data with simple address calculations.

2. write.

It is invoked to write uncommitted data\(^1\) that are about to be removed from main memory. The toSecondaryCache method of the MemoryObjectCache class invokes this method to flush data cached in main memory. This method allows a secondary cache to cache this data before it is flushed from main memory. Like the read method, its arguments include a starting unit, the number of units to be flushed, and the starting address of the buffer that contains the data that is about to flushed. However, it has an extra argument. This argument provides the referenced status and the modified status of each of the affected units. This status information is required to maintain cache consistency.

A secondary cache may delegate a write invocation to its child. In other words, it does not cache any of the affected data. Alternatively, it may cache the data and propagate the write invocation to its child so that its child can cache the data as well. It also may cache the data itself and not propagate the write invocation. When a memory object cache invokes this method, it expects dirty data to be either cached by at least one of its secondary caches or be written back to the backing store memory object.

3. rawWrite.

The toMemoryObject method of the MemoryObjectCache class invokes this method to write committed data\(^1\) to the backing store memory object. It takes the same arguments as the write method. In most cases, write behaves like rawWrite except that dirty data are usually written back to the backing store memory object.

A secondary cache with persistent storage, that can survive system failures, can cache dirty data. It need not write the dirty data to the backing store memory object immediately, provided the

---

\(^1\) Uncommitted data are not expected to survive system failures.

\(^1\) Committed data must survive system failures.
secondary cache can update the backing store memory object eventually. Such a secondary cache may need to participate in a cache coherence protocol to maintain cache consistency.

4. invalidate -
It invalidates units that are currently cached by a secondary cache. A secondary cache always propagates an invalidation request to its child. For example, when a parent secondary cache caches an unit which contains dirty data from main memory as a result of a write operation and it does not propagate the write message to its child, it must invalidate the same unit in all the secondary caches below it. It does so by invoking the invalidate method of its child. This starts a chain of invalidation requests that propagates through all the secondary caches below the secondary cache that initiated the invalidate operation. If the parent secondary cache does not invalidate the secondary caches below itself, then an inconsistency may occur. This is because one of the lower secondary caches may have cached the same unit and believe it is still dirty. When this lower secondary cache is synchronized, it might write this out-of-date unit to the backing store memory object.

5. synchronize -
The synchronize method of the MemoryObjectCache class invokes this method to force a memory object cache's secondary caches to write dirty units that have been cached in their caches to the backing store memory object. Each secondary cache in a stack propagates the synchronize message down to its child so that all the secondary caches in the stack have a chance to write their dirty cached units to the backing store memory object. A good implementation should not write a unit to the backing store memory object if the unit in its cache is clean, or if the unit has already been updated earlier, such as by a higher secondary cache.

6. moRead and moWrite -
These methods read from and write to the backing store memory object. They translate arguments from the cache unit size to the memory object's unit size. Usually, they are invoked by read, write, rawWrite, and synchronize.

7. stack manipulation methods -
These methods are used to insert and remove secondary caches from a stack.

The abstract SecondaryCache class has two subclasses: the DummySecondaryCache class and the RealSecondaryCache class. The DummySecondaryCache class is an implementation convenience. It allows a memory object cache to do all its I/O through a secondary cache. It ensures that at least one secondary cache is attached to a memory object cache. The dummy secondary cache "caches" data in the backing store memory object. It is always the bottom-most secondary cache in a secondary cache stack. Its read
method reads from the backing store memory object and its write and rawWrite methods write modified units to the backing store memory object. It does nothing for invalidation and synchronization requests.

RealSecondaryCache

Subclasses of the RealSecondaryCache class implement secondary caches that actually do cache data. The RealSecondaryCache class is an abstract superclass. It implements some basic functionalities and a basic caching strategy that its concrete subclasses may inherit. One of these functionalities is to track the status of each unit. At the very least, each real secondary cache must know which units it has cached and which of these units have been modified. A modified unit contains dirty data, i.e. the dirty data has not been written to the backing store memory object. As mentioned in Section 6.3.3, this status information is required to maintain cache consistency.

The RealSecondaryCache class implements a basic caching strategy that can be inherited by its subclasses. This strategy does not cache units that are being read into main memory. Instead, it caches units that are being removed from memory or units that are being written to its backing store memory object. The main advantage of not caching on read is that it may save on writes to the real secondary cache's storage device. For example, it saves on writes when the units being read into main memory will be modified or deleted soon. It may also save on writes for frequently used units that are not likely to flushed or synchronized. These units are likely to remain cached in main memory once they are brought into main memory. It is not likely that the real secondary cache will have a chance to cache these units.

The following methods implement the above strategy:

1. read -
   First, the real secondary cache attempts to satisfy the read request with units that it has cached. Then, it invokes the read method of the its child for those units that it had not cached. For example, a real secondary cache is invoked to read units 0 to 6 into a buffer starting at address 0x0000 and it only has units 0 to 3 cached. First, it obtains the data for units 0 to 3 from its cache and copies this data into the buffer at 0x0000 to 0x3FFF (assuming each unit contains 0x100 bytes). Then, it invokes its child’s read method to read units 4 to 6 into the buffer at 0x4000 to 5FFF, and it does not cache the data provided by it’s child secondary cache.

2. write -
   The real secondary cache does nothing for units that it has already cached and have not been modified while in main memory. The modified status of these units remains unchanged because a modified unit in the real secondary cache still contains dirty data.

\(^{12}\)The 0x prefix indicates that the following number is in hexadecimal.
For units that it has cached and have been modified while in main memory, it updates these units in its cache with the new data from main memory and tags these units with the modified status.

For units that it has not previously cached, it caches these units by writing them to its storage device and tags units that have been altered while in main memory with the modified status.

In general, the write method tags cached units that have been modified while in main memory with the modified status. The modified status of a unit is cleared usually after the unit has been written to the backing store memory object.

This write method also invokes the invalidate method on the secondary cache's child to invalidate units that have been marked as modified in its own cache. This invalidation request informs the lower secondary caches that they no longer have the latest copy of these units.

3. rawWrite -

This method does its work in three phases. In the first phase, the real secondary cache identifies the units that must be written out to the backing store memory object. These units either have been modified while in main memory or have been tagged as having been modified in the real secondary cache.

In the second phase, the real secondary cache identifies the units that it should cache or update. Like the write method, it caches units that it has not cached and it updates cached units with new data from main memory. The set of units identified in the first phase are not necessarily disjoint from the set of units identified in the second phase. Unlike the write method, this method clears the modified status of all the affected units identified by the arguments to this method since the data in these units will have been written to the backing store memory object when the method returns.

In the third phase, also known as the I/O phase, the real secondary cache invokes I/O methods on various objects. It writes units identified in the first phase to the backing store memory object. It writes units identified in the second phase to the storage device that caches its data. It also invokes the invalidate method on its child secondary cache to invalidate units identified in the second phase. Delaying I/O until the last phase allows all the necessary I/O operations to be issued asynchronously at the same time. An asynchronous rawWrite operation returns after issuing these asynchronous I/O operations. A synchronous rawWrite operation waits for all these asynchronous I/O's to complete.

The real secondary cache always issues asynchronous I/O's because asynchronous I/O's increase concurrency. In particular, if an unit has to be written to both the backing store memory object

---

13 An asynchronous operation is one that returns to its caller before it completes the operation. Asynchronous operations will be discussed in further details in Section 7.5.1.
and the secondary cache's storage device, the two write operations can occur concurrently. For example, a unit belonging to a remote file with a local client disk cache can be transmitted by the client and written to the local disk at the same time.\textsuperscript{14}

This method is also optimized to eliminate memory copies. It allows multiple concurrent asynchronous I/O operations to obtain their data from the same memory location. In other words, it does not setup separate buffers for I/O operations that write the same data to different devices.

4. synchronize -

This method writes modified units held in the real secondary cache to the backing store memory object. The synchronize method of the MemoryObjectCache class invokes this method for units that are not currently cached in main memory. For units that are currently cached in main memory, the memory object cache invokes its top-most secondary cache's rawWrite method.

If modified units exist, this method needs a buffer to transfer dirty data from the real secondary cache's storage device into main memory and from main memory to the memory object. It allocates a buffer only if another secondary cache in the same stack has not allocated a buffer. Otherwise, it reuses the previously allocated buffer.

In other words, this buffer is allocated only once and it is shared by all the secondary caches belonging to the same stack. This should be pretty big in order to maximize the amount of data transferred per I/O request, i.e. to reduce I/O fragmentation. Hence, it is not desirable for each secondary cache to allocate its own buffer.

5. allocate -

The write and rawWrite methods invoke this method. This method reserves storage from the real secondary cache's storage device for new data. The RealSecondaryCache class does not implement a cache replacement policy. Its concrete subclasses are expected to provide their own cache replacement policies. This method could initiate cache replacement in these subclasses. When invoked, it should not return until space for the requested number of units has been allocated.

6. basicRead and basicWrite -

The read, write, and rawWrite methods invoke these methods to transfer units between main memory and the real secondary cache's storage device. Concrete subclasses of the RealSecondaryCache class must provide an implementation for these methods. Implementations of the basicWrite method are expected to allocate space for new units that were not previously cached. If the write and rawWrite methods require space allocation, they invoke the allocate method before

\textsuperscript{14}This assumes that there is no other common system resource bottlenecks.
they invoke the `basicWrite` method. This ensures that sufficient space has been reserved for the new data that will be written out by `basicWrite`.

**UNIX_inodeSecondaryCache**

The `UNIX_inodeSecondaryCache` class is a concrete subclass of the `RealSecondaryCache` class. An UNIX inode secondary cache\(^\text{15}\) caches data in an UNIX inode. It is used normally to cache data from a remote memory object on a client host's local UNIX file system. It inherits the caching strategy of the `RealSecondaryCache` class. The following static or class attributes of the `UNIX_inodeSecondaryCache` class limit the amount of resources that UNIX inode secondary caches on a host may consume:

1. `maxCachedFile`
   - It is the maximum number of UNIX inode secondary caches permitted on the host.

2. `maxCachedBytes`
   - It is the maximum number of bytes that can be allocated from an UNIX file system to UNIX inode secondary caches for caching. Byte is unit of choice since different UNIX inode secondary caches can have different cache unit sizes.

3. `thresholdCachedFiles`
   - This is the number of UNIX inode secondary caches that can be used for caching before cache replacement starts. The default value for this attribute is 90% of `maxCachedFiles`.

4. `thresholdCachedBytes`
   - This is the number of bytes that can be allocated from an UNIX file system to UNIX inode secondary caches for caching before cache replacement starts. The default value for this attribute is 90% of `maxCachedBytes`.

5. `cachedFiles`
   - This is the number of UNIX inodes that is currently used by UNIX inode secondary caches for caching.

6. `cachedBytes`
   - This is the number of bytes that is currently allocated from an UNIX file system to UNIX inode secondary caches for caching.

The `UNIX_inodeSecondaryCache` class implements a LRU cache replacement policy. It reclaims resources from the least recently used UNIX inode secondary cache on the host. Cache replacement occurs

\(^{15}\) A UNIX inode secondary cache is an instance of the UNIX_inodeSecondaryCache class.
when `cachedFiles exceed thresholdCachedFiles` or when `cachedBytes exceeds thresholdCachedBytes`. The `UNIXInodeSecondaryCache` class implements the following methods:

1. **cleaner**.
   It is a static or class method. It contains the main loop of a daemon process known as the `UNIXInodeSecondaryCache`'s *cleaner daemon*. This daemon is active only when cache replacement is required. It synchronizes and flushes the cached contents of the least recently used `UNIX inode secondary cache` on the host. Then, it destroys this least recently used `UNIX inode secondary cache`.

2. **constructor**.
   If `maxCachedFiles` has been reached, this method returns with an error. If `maxCachedFiles` has not been reached, it creates a new `UNIX inode` for caching units from the `UNIX inode secondary cache`'s backing store *memory object*. Then, it increments `cachedFiles`. If `cachedFiles exceeds thresholdCachedFiles` and the cleaner daemon is inactive, it activates the cleaner daemon.

3. **destructor**.
   It decrements `cachedFiles`. It also updates `cachedBytes` to reflect the amount of space freed by reclaimed from the `UNIX inode secondary cache`.

4. **allocate**.
   It updates `cachedBytes` and activates the cleaner daemon when `cachedBytes exceeds threshold-MaxBytes`.

5. **read, write, and rawWrite**.
   Each of these methods updates the LRU queue maintained by the `UNIXInodeSecondaryCache` class and invokes the corresponding `read, write, or rawWrite` method of the `RealSecondaryCache` class.

6. **basicRead and basicWrite**.
   These methods read from and write to the `UNIX inode` that caches data on behalf of the `UNIX inode secondary cache`.

**Summary**

In summary, a *memory object cache* co-operates with its *secondary caches* to cache file data in tertiary storage devices. The `SecondaryCache` class defines the abstract interface for *secondary caches*. It also specifies how *secondary caches* in stack cooperate to maintain cache consistency. The `Dummy-SecondaryCache` class is a special subclass of the `SecondaryCache` class. It simplifies the framework by permitting the `MemoryObjectCache` class to delegate its I/O operations to the `SecondaryCache` class. It does not actually cache data in any tertiary storage devices. The `RealSecondaryCache` class provides a
simple caching strategy that is useful to **secondary caches** that do cache data in tertiary storage devices. The UNIX_inodeSecondaryCache class inherits this strategy and it implements caching of file data on local UNIX file systems.

6.3.7 **MigrationPolicy**

Subclasses of the MigrationPolicy class implement different cache replacement policies and different data transfer strategies. Each memory object cache has an associated migration policy. The MigrationPolicy class has methods that represent different events. A memory object cache invokes these methods when these events occur. These methods include:

1. **unitsFromStore** -
   A memory object cache invokes this method before it requests for more main memory from the store.

2. **unitsToStore** -
   A memory object cache invokes this method before it returns previously allocated main memory to the store.

3. **preCacheRead** and **preCacheWrite** -
   A memory object cache invokes these methods before processing cacheRead and cacheWrite requests.

4. **postCacheRead** and **postCacheWrite** -
   They are similar to **preCacheRead** and **preCacheWrite**, except that a memory object cache invokes them after processing cacheRead and cacheWrite requests.

5. **postPageIn** -
   A memory object cache invokes this method after paging data into main memory, i.e. after invoking fromSecondaryCache.

6. **pageOut** -
   A memory object cache invokes this method when the system pager invokes its pageOut method. In other words, the memory object cache delegates pageOut requests to its migration policy. This method has a single argument. This argument indicates the number of cache units that should be paged out. This method may only page out eligible units that have not been locked.\(^{16}\) An unit may be locked if an I/O operation is pending on it, or if it contains vital kernel data that cannot be paged out.

\(^{16}\)This requires the pageOut method to consult the memory object's associated range lock.
Subclasses of the MigrationPolicy class implement different migration policies. In general, immediate subclasses of the MigrationPolicy class implement various cache replacement policies. Subclasses of these classes implement additional data transfer strategies. Although multiple inheritance may be used to combine policies and strategies, it is not used since the Choices framework and the initial Choices C++ compiler does not support multiple inheritance. However, this thesis will discuss the use of multiple inheritance where appropriate.

The remainder of this subsection describes a few subclasses of the MigrationPolicy class and how they implement various cache replacement policies and data transfer strategies. First, it describes the simplest subclasses. These subclasses implement local replacement policies. Then, it present a subclass that implements a group replacement policy. Finally, it describes subclasses that implement data transfer strategies.

6.3.7.1 Local Replacement Policies

The SimpleReplacementPolicy class implements a very simple and straightforward local replacement policy. As mentioned in Section 6.2, a local replacement policy chooses units for replacement from a single memory object cache. A simple replacement policy scans its memory object cache’s associated map object in ascending unit order for eligible units. It pages the requested number of units out if available. On subsequent invocations of its pageOut method, it begins its scan from where it last stopped its scan. This policy is simple because each instance only requires an instance variable to remember where the previous scan stopped and it only needs to implement the pageOut method.

The BoundedReplacementPolicy class is an abstract class. It limits the number units that a memory object cache may have resident in main memory. It has two instance variables, one for the maximum number of resident units and one for the current number of resident units. It implements the following methods.

1. maxResidentUnits and SetMaxResidentUnits.
   They retrieve and set the maximum number of units that the memory object cache may cache in main memory. In other words, they are accessor methods for the protected maximum number of resident units instance variable.

2. unitsFromStore.
   This method invokes the pageOut method if acquiring the desired number of new units from the store causes too many units to be resident in main memory. It also increments the instance variable that counts the current number of resident units.

\footnote{The current number of resident units can be obtained from the map object. It’s value is accessed frequently. It is duplicated in each instance of bounded replacement policy for efficiency to eliminate the overhead of several procedure calls to obtain it from the map.}
3. unitsToStore -
   It decrements the instance variable that counts the current number of resident units.

The FIFOReplacementPolicy class is a subclass of the BoundedReplacementPolicy class. It implements the First-In-First-Out (FIFO) replacement policy for units cached by a memory object cache. It maintains a queue of resident units. The head of the queue has the unit paged in first and the tail of the queue has the unit paged in last. The following methods are implemented by the FIFOReplacementPolicy class.

1. constructor -
   It must initialize an instance's queue with units that are already resident in main memory when the FIFO replacement policy is created. Since a memory object cache can change its migration policy, units may be already cached in main memory when the migration policy is constructed. In general, a subclass of the MigrationPolicy class that maintains additional information for each resident unit must overload its constructor to take into account units that are resident already.

   The constructor for the FIFOReplacementPolicy class initializes an instance's queue by adding all resident units cached by the memory object cache to the queue. Since it is not possible to determine the order in which units were originally brought into main memory, these units are added to the tail of the queue in ascending order by unit number. In other words, it assumes that the resident unit with the smallest unit number has been paged in first, and the resident unit with the largest unit number has been paged in last.

2. postPageIn -
   It adds the units specified by its arguments to the tail of the queue.

3. unitsToStore -
   It removes the units specified by its arguments from the queue.

4. pageOut -
   It removes the specified number of eligible units from the head of the queue and pages them out.

The LIFORelationshipPolicy class is a subclass of the FIFOReplacementPolicy class. It implements the Last-In-First-Out (LIFO) replacement policy for units cached by a memory object cache. It is a subclass of the FIFOReplacementPolicy. It overloads the postPageIn method to add the specified units to the head of the queue instead of the tail of the queue.

6.3.7.2 Group Replacement Policies

As mentioned in Section 6.2, a group replacement policy chooses units for replacement from memory object caches that use the same or a related family of replacement policies. The units chosen for replacement may or may not be cached by the memory object cache that received the pageOut message. In
general, migration policy objects that have the same group cache replacement policy cooperate to choose units for replacement. They usually update a shared object. This shared object contains information necessary to determine which units should be paged out first. Typically, this shared object maintains a table or a queue of units that are cached by memory object caches that use the same group replacement policy.

The GroupLRUReplacementPolicy class implements a group LRU replacement policy. It does not implement a LRU replacement policy that always chooses the least recently used unit for replacement. In other words, it does not implement a strict LRU replacement policy. The performance penalty for implementing a strict LRU replacement policy would be too high. On most conventional computer architectures, each memory reference to a different page must be detected by the kernel in order to maintain a strict LRU queue. Typically, the kernel detects each reference to a different page by manipulating the virtual memory hardware to generate a page fault when a different page has been accessed. This is very inefficient since the kernel would have to handle a page fault for each distinct page referenced. Like many other virtual memory management systems, the GroupLRUReplacementPolicy class implements an approximation of the LRU replacement policy known as the two-handed clock algorithm\cite{58}. As suggested by its name, the two handed clock algorithm involves two “hands” and a “clock”.

The shared object maintained by the GroupLRUReplacementPolicy class is known as the GroupLRUInfoTable. The GroupLRUInfoTable is a table of GroupLRUInfo entries. It represents the face of a clock and the GroupLRUInfo entries represent tick marks on the face of the clock. Each GroupLRUInfo represents a cluster that is resident in main memory. A cluster in this context is a set of consecutive units that have been paged into main memory together. Three attributes uniquely identify a cluster. These attributes are the first unit in the cluster, the last unit in the cluster, and a pointer to an instance of the GroupLRUReplacementPolicy class. This pointer indirectly and uniquely identifies the memory object cache that caches the cluster.

Like the BSD implementation of the two-handed clock algorithm, the GroupLRUReplacementPolicy class has a daemon process. This daemon process, known as the LRU pager, scans the GroupLRUInfoTable for the least recently used cluster. Initially, the LRU pager is inactive, i.e. it is blocked on a semaphore. The LRU pager is activated or unblocked with the help of the system pager daemon (discussed in Section 5.2.2). When there is little free memory on the host, the store activates the system pager daemon. When the system pager invokes the pageOut method on a memory object cache that has a group LRU replacement policy, this memory object cache delegates the pageOut message to its migration policy object which its group LRU replacement policy. This method activates the LRU pager.

The LRU pager maintains two pointers that point to GroupLRUInfo entries. Each pointer references an GroupLRUInfo in the GroupLRUInfoTable. These pointers represents the two hands of the clock. Each hand points to a tick mark on the face of the clock. In this variant of the two-handed clock
algorithm, the second hand always lags the first hand by a quarter of a revolution of the clock. In this
discussion, the cluster identified by the first hand is known as the first cluster, the cluster identified by
the second hand is known as the second cluster.

The LRU pager executes a loop consisting of three different stages. It operates on the first cluster
in the first stage. It operates on the second cluster in the second stage. In the third stage, it ensures
that the two hands revolve around the clock at a reasonable speed.

In the first stage, the LRU pager clears the referenced status from the units in the first cluster with
the help of the cluster’s memory object cache’s map and translation map. This clears the reference bits
of the physically addressable units that cache this unit, and updates the appropriate machine dependent
address translation tables. At the end of the first stage, the LRU pager advances the first hand to point
to the next tick mark on the face of the clock, i.e. the next GroupLRUInfo in the GroupLRUInfoTable.

At the beginning of the second stage, the LRU pager reads the referenced status of the second cluster,
with the help of the cluster’s memory object cache’s map and translation map. If the second cluster is
eligible for replacement and it has not been referenced, it is paged out. To page out a cluster, the LRU
pager invokes the toSecondaryCache method on the cluster’s memory object cache. As mentioned in
Section 6.3.3 and Section 6.3.6, the toSecondaryCache method eventually invokes the write method on
the memory object cache’s top-most secondary cache. The secondary cache may cache these units.

The new framework supports both synchronous and asynchronous I/O operations.\textsuperscript{18} Most I/O
methods in this framework support both synchronous and asynchronous operations. A synchronous I/O
operation returns to its caller after the necessary device I/O requests have been completed. An asyn-
crchromous I/O operation returns to its caller after the required device I/O requests have been scheduled
for execution. It does not wait for the completion of the device I/O requests. The LRU pager invokes the
toSecondaryCache method asynchronously. This allows the LRU pager to continue scanning for more
clusters for replacement after the necessary device I/O requests have been scheduled. The physically
addressable units occupied by the paged-out clusters will be freed when these I/O requests complete. At
the end of the second stage, the LRU pager advances the second hand to next tick mark.

At the beginning of the third stage, the LRU pager calculates the current scan rate. The scan rate
is the number of revolutions that each hand completes in a second. The following algorithm determines
the scan rate of the LRU pager.

\[
\begin{align*}
\text{mainStoreSize} &= \text{total amount of main memory} \\
\text{free} &= \text{amount of free main memory} \\
\text{lotsFree} &= \text{mainStoreSize}/8 \\
\text{desFree} &= \text{mainStoreSize}/16 \\
\text{minFree} &= \text{mainStoreSize}/32
\end{align*}
\]

\textsuperscript{18}Section 7.5.1 discusses the need for asynchronous operations and how this framework implements them.
\[ \text{fastScan} = 1000 \text{ revolutions/sec} \\
\text{slowScan} = 100 \text{ revolutions/sec} \\
\text{verySlowScan} = 1 \text{ revolution/sec} \]

if free < lotsFree then
    scanRate = 0
if lotsFree < free < desFree then
    verySlowScan < scanRate < slowScan and
    scanRate is inversely proportional to free
if desFree < free < minFree then
    slowScan < scanRate < fastScan and
    scanRate is inversely proportional to free

The LRU pager uses a higher scan rate when there is very little free memory, and a slower scan rate when there is more free memory. The LRU pager deactivates itself by blocking on a semaphore when the scan rate is zero. The current scan rate determines the desired time delay between each advance of the two hands. In other words, the current scan rate determines the amount of time each iteration of the loop should take. The LRU pager sleeps at the end of the third stage if the time consumed by the current iteration is less than the desired loop iteration time.

The primary differences between the \text{GroupLRUReplacementPolicy} and the BSD two-handed clock are:

1. If the length of each tick mark on the face of the clock indicates the amount of the data represented by the tick mark, then the BSD clock face has tick marks of fixed and equal length while the \text{GroupLRUReplacementPolicy} clock has tick marks of variable lengths. In other words, the amount of data scanned in each iteration is fixed in BSD and variable in \textit{Choices}. The \textit{Choices} LRU pager operates on clusters containing different number of units and units of different sizes. The BSD pager works operates on fixed size pages.

2. If the number of ticks on the face of the clock determines the size of the clock, then the size of the BSD clock is fixed and the size of the \text{GroupLRUReplacementPolicy} clock is variable. The amount of pageable memory on the host determines the size of the BSD clock. On the other hand, the number of clusters under the control of the group LRU replacement policy determines the size of the \text{GroupLRUReplacementPolicy} clock. Since the number of clusters under the control of the group LRU replacement policy may change over time, the size of the \text{GroupLRUReplacementPolicy} clock varies over time too.

These differences complicate the LRU pager and the \text{GroupLRUReplacementPolicy} class. However, the performance benefits of clustering justifies the additional complications. Clustering takes advantage of spatial and temporal locality. It assumes units that are brought into main memory “together” are “related” and hence likely to be referenced “together”. Clustering reduces the number of I/O requests
since there should be fewer clusters to scan and page out. Fewer I/O requests reduces the number of
disk seeks. Clustering also increases the size of each I/O request. A large I/O operation is usually more
efficient than a few smaller I/O operations when transferring the same amount of data. In summary,
clustering increases I/O efficiency.

The GroupLRUReplacementPolicy class overloads the following methods of the MigrationPolicy class:

1. `unitsToStore` -
   It removes GroupLRUInfo entries that represent the specified units from the GroupLRUInfoTable.
   It splits a cluster if it removes one or more units from the middle of a cluster.

2. `postPageIn` -
   It adds a GroupLRUInfo entry that represents the specified units to the GroupLRUInfoTable.

3. `pageOut` -
   It activates the LRU pager if the LRU pager is not already active.

A group MRU replacement policy could be implemented in a similar way.

### 6.3.7.3 Partitioning Memory

When multiple cache replacement policies are present on a host, the strategy for partitioning main
memory among the various cache replacement policies needs to be addressed. The following strategies
have been suggested:

1. **Fixed sized partitions** -
   Allocate or reserve a fixed amount of memory for each cache replacement policy or memory object
cache[57].

2. **No worse than some global cache replacement policy** -
   Cao et al. developed such a strategy for file caches (refer to Section 4.7). This strategy ensures
that a system that has individual cache replacement policies do not incur more page faults than
a system that has only a single system-wide LRU cache replacement policy. This strategy has
been evaluated for use in this framework. Unfortunately, it cannot be applied efficiently to virtual
memory management systems since it requires the exact memory reference pattern to be known
in order to establish an LRU queue.\(^\text{19}\)

*Choices* employs the “free competition” strategy. Individual caches acquire main memory as required.
When the amount of free memory is low, the store activates the system pager. The system pager is part

\(^{19}\text{In particular, LRU-SP stands for Least-Recent by-Used with Swap and Place-holder. The Swap operation requires the position of individual units in the LRU queue to be known.}\)
of a store. A method of the store implements the system pager. In the current implementation, the
system pager requests each memory object cache to free an amount of main memory that is in proportion
to the amount of memory consumed by the memory object cache. The following equation computes the
amount of main memory that the system pager requests each memory object cache to free when the
system pager invokes the pageOut method of the memory object cache.

\[ R = (C \times F) \div T \]

where
- \( R \) = amount of memory that the memory object cache should free
- \( C \) = total amount of memory consumed by the memory object cache
- \( T \) = total amount of pageable memory on the host
- \( F \) = total amount of memory that the store wants to reclaim

The amount requested by the system pager is a hint. Each individual memory object cache may free
more or less than the requested amount. For example, a memory object cache that believes it has a
higher priority may free half the requested amount. Similarly, a memory object cache that believes it has
a lower priority may free twice the requested amount. In the case of a group LRU replacement policy,
the LRU pager is activated and the LRU pager frees memory using the two-handed clock algorithm
discussed in Section 6.3.7.2. The system pager continues to loop through all the memory object caches
until the desired amount of main memory has been reclaimed.

In this strategy, memory object caches “compete” for main memory when main memory is available
for allocation. Eventually, available memory becomes exhausted. Then, these memory object caches
“compete” to reclaim main memory. The current system pager provides some degree of fairness by
requesting larger consumers of memory to free more memory than smaller consumers of memory. To
avoid possible deadlock or livelock, each memory object cache should at least reclaim one unit when
possible.

6.3.7.4 Data Transfer Strategies

As mentioned in Section 6.2.4, a migration policy determines the cache replacement policy and the data
transfer strategies of its memory object cache. The previous subsections presented subclasses of the
MigrationPolicy class that implement various cache replacement policies. This subsection illustrates how
new data transfer strategies can be added to the new framework by subclassing these classes.

The SmarterLRUReplacementPolicy class is a subclass of the GroupLRUReplacementPolicy class. Its
primary goal is to optimize for sequential file access performance. It extends the GroupLRUReplacement-
Policy class by adding optimizations that improve sequential file access performance. These optimiza-
tions include clustering, read-ahead, write-behind, and free-behind. These optimizations are inspired by
Mcvoy’s and Kleiman’s extensions to the UNIX file system to achieve sequential file access performance similar to extent-based file systems[73].

**Read Clustering**

Clustering improves I/O efficiency by reducing the number of I/O’s and increasing I/O sizes. A smarter LRU replacement policy performs clustering by attempting to issue I/O operations at the optimal I/O data transfer size of its backing store memory object whenever possible. In other words, the optimal I/O data transfer size of a memory object determines the optimal size of the smarter LRU replacement policy’s clusters. The optimal I/O data transfer size of a memory object is the smallest amount of data that an I/O operation should transfer in order to obtain near peak throughput for the memory object. It depends on the characteristics of the memory object’s underlying storage device. For example, the optimal I/O data transfer size of a disk on a Sun SPARCstation 2 is 64 kilobytes. 64 kilobytes is the size of the largest Direct Memory Access (DMA) transfer that a SPARCstation 2 can perform to and from a disk. The optimal I/O data transfer size of a memory object also may depend on how the memory object stores its data on the device. For example, each file system has its own disk layout. Hence, the optimal I/O data transfer size of a file depends on the file system’s disk layout.

A smarter LRU replacement policy clusters read operations by reading data into main memory a cluster at a time, even though the individual read requests are smaller than a cluster. When an application issues a read request for the initial unit, the smarter LRU replacement policy pages in the entire cluster that contains the unit. Since sequential read access is expected, the application should read the rest of the units in this cluster eventually. The smarter LRU replacement policy pages in the next cluster when the application reads the unit that starts the next cluster. The smarter LRU replacement policy’s read clustering strategy is beneficial to sequential read access because it performs data prefetching. However, the amount of data it prefetches is always less than a cluster. It also less likely to read unnecessary data into main memory as part of a cluster since most applications are likely to access all the data brought into main memory as part of a cluster[73, 80].

**Cluster Read-ahead**

In addition to prefetching performed as part of clustering read operations. A smarter LRU replacement policy also implements an optimization known as cluster read-ahead. It applies this optimization when the desired amount of data prefetching exceeds the optimal size of a cluster. An instance attribute specifies the desired amount of data to prefetch. This attribute is known as the readAheadLag. It contains the desired number of read-ahead units. Cluster read-ahead attempts to prefetch the next

---

20 This initial unit could be the first unit of multi-unit read request.
21 Unless the application does not read all the data in the cluster.
**readAheadLag** units using read operations of the optimal cluster size. It issues these read operations asynchronous and concurrently when necessary.

An alternative to cluster read-ahead is to fetch the desired amount of read-ahead units using a single large read operation. This alternative is less desirable than cluster read-ahead because it increases latency and reduces concurrency.

Its inferiority is best illustrated by an example. A large read operation may read 128 units. While the read operation is in progress, all 128 units are inaccessible to the memory object cache since the memory object cache does not know which units contain valid data, *i.e.* which units have been fetched into main memory by the read operation. Hence, an application cannot access any prefetched data until all 128 units have been read. This alternative has a higher latency because the application must wait for all 128 units before it can access the first unit, even though the first 127 units have been fetched into main memory already.

Cluster read-ahead avoids this latency problem. It divides the 128-unit read into eight 16-unit read operations (assuming an optimal cluster size of 16). It issues these read operations concurrently. A cluster becomes accessible once its read operation completes. Cluster read-ahead has a lower latency because the application need only wait for the 16 units of the first cluster before it can access the first unit. Cluster read-ahead also allows for more concurrency. It allows the application to work on the data in the first cluster while the remaining clusters are being fetched.

**Write Clustering**

A smarter LRU replacement policy clusters write operations by writing data out a cluster at a time, even though the individual write requests are smaller than a cluster. The SmarterLRUReplacementPolicy class implements write clustering by accumulating smaller write requests until an entire cluster can be written.

When an application issues a write request to the initial unit, the memory object cache caches the data written to this unit and the smarter LRU replacement policy "opens" a new cluster that begins with this unit. Since sequential write access is expected, the application eventually writes additional consecutive units. Like the initial unit, the memory object cache caches these units. After the application has written to an entire cluster, the smarter LRU replacement policy "closes" the cluster and writes the cluster out to the memory object cache’s top-most secondary cache. The secondary cache stack of the memory object cache determines how the cluster will be written to persistent storage. This delayed write action is commonly known as a write-behind. The smarter LRU replacement policy does not flush the cluster from main memory after the cluster has been written. Like read clustering, the smarter LRU

---

22 This initial unit could be the first unit of multi-unit write request.
23 Unless the application does not write to the entire cluster.
replacement policy “opens” new cluster for writing when the application writes to the unit immediately following the previous cluster.

**Write-behind**

Write-behind is particularly beneficial when the data written are not likely to be overwritten in the near future. It improves I/O performance by reducing latency, balancing I/O load, and maintaining the spatial locality of writes. Without write-behind, a dirty unit is written out only when cache replacement is required. With write-behind, a dirty unit is written out and becomes clean soon after an application wrote into it. When cache replacement is required, a clean unit can be reclaimed much faster than a dirty unit since the data in a clean unit need not be saved before reclaiming the unit’s memory. Therefore, write-behind reduces latency required to reclaim memory.

Write-behind also helps to spread or “load balance” writes over time. It writes new or modified units to a storage device soon after an application wrote into these units. It does not wait for a pager to perform these writes. In most cases, write-behind occurs before memory reclamation. Hence, it increases the write load when the pager is inactive. However, it reduces the write load during memory reclamation, *i.e.* when the pager is active, when I/O bandwidth is in higher demand for both paging in new data and writing out dirty data.

A problem with some cache replacement algorithms is that they lose the original spatial relationship between clusters. This fragments and scatters I/O requests and may result in additional disk seeks. An example of such a algorithm is the two-handed clock algorithm presented in Section 6.3.7.2. This algorithm pages out eligible clusters in the order they have been scanned. This order may or may not reflect the original order in which dirty clusters have been written. In the case of sequential write access, the two-handed clock algorithm may write the cluster two before the cluster one. Write-behind helps to maintain the original spatial relationship between clusters. It writes clusters out in the proper sequential order, and it reduces the likelihood that the pager will need to perform these writes.

**Free-behind**

The SmarterLRUReplacementPolicy class also implements an optimization known as free-behind. This optimization attempts to reduce memory contention by regulating the number of units that a smarter LRU replacement policy’s memory object cache may cache. When memory contention is high, it allows the memory object cache to cache fewer units. When memory contention is low, it allows the memory object cache to cache more units. A smarter LRU replacement policy uses the elapsed time between invocations of its pageOut method to determine the level of memory contention. If the elapsed time is short, then the system pager is very active. This implies that memory contention is high. If the elapsed time is very long, then the system pager is quite inactive. This implies that memory contention is low.
Like a bounded replacement policy, a smarter LRU replacement policy voluntarily removes units from its memory object cache to ensure that the memory object cache does not cache too many units. The bounded replacement policy removes units by invoking its `pageOut` method. Unlike the bounded replacement policy, the smarter LRU replacement policy selects units for removal by taking advantage of its knowledge of the expected access pattern. Since the expected access pattern is sequential, the smarter LRU replacement policy chooses units that have already been accessed. These units have smaller unit numbers than the most recently accessed unit. In other words, the smarter LRU replacement policy frees units "behind" the most recently accessed unit. This optimization is desirable because it takes advantage of the expected sequential access pattern to regulate the memory usage of individual memory object caches to reduce memory contention.

SmarterLRUReplacementPolicy

A smarter LRU replacement policy has various attributes that control its data transfer strategies. These attributes determine when and how the smarter LRU replacement policy performs the abovementioned optimizations. The attributes are:

1. `clusterSize` -
   It contains the optimal number of units per cluster for maximum I/O efficiency. It’s value is computed from the optimal I/O data transfer size of the smarter LRU replacement policy’s backing store memory object.

2. `readSequential` and `writeSequential` -
   They are boolean variables. The `readSequential` variable indicates if the memory object is sequentially read. Prefetching is disabled if `readSequential` is false. The `writeSequential` variable indicates if the memory object is sequentially written. Write-behind is disabled if `writeSequential` is false.

3. `readPointer` and `writePointer` -
   They contain unit numbers and are updated only when sequential access has been detected. The `readPointer` points to the next sequential unit that the smarter LRU replacement policy expects an application to read. The `writePointer` points to the next sequential unit that the smarter LRU replacement policy expects an application to write.

4. `readAheadLag` -
   It contains the desired number of units to prefetch.

5. `readAheadPointer` -
   It points to the first unit of the next cluster to prefetch. Usually, it points to the unit after the last unit prefetched by the last cluster read-ahead operation.
6. *enableWriteBehind* -
   It is a boolean variable. It determines if write-behind should be disabled. It allows subclasses of the `SmarterLRUReplacementPolicy` class to disable write-behind.

7. *nextWriteBehind* -
   It points to the first unit that should be written out as part of the next write-behind operation. Units before `nextWriteBehind` should be clean. Either these units have not been touched, or their modified data has been written out earlier as part of the previous write-behind operation.

8. *freeBehindLag* -
   It determines the maximum number of resident units that the memory object cache may cache before triggering the free-behind operation. The free-behind operation frees memory by removing from the memory object cache units that are furtherest away from either the current `readPointer` or the current `writePointer`. As mentioned earlier, a smarter LRU replacement policy adjusts its `freeBehindLag` periodically to reflect the amount of memory contention.

9. *minFreeBehindLag* and *maxFreeBehindLag* -
   They determine the minimum and maximum value of `freeBehindLag`, respectively. They allow subclasses of the `SmarterLRUReplacementPolicy` class to modify the default behavior of its free-behind strategy. For example, a subclass can assign a fixed value to `freeBehindLag` by setting `minFreeBehindLag` and `maxFreeBehindLag` to the same value. The default value of `minFreeBehindLag` is twice the `clusterSize`. The default value of `maxFreeBehindLag` is computed from the host’s main memory size.

10. *readFreeBehindPointer* and *writeFreeBehindPointer* -
    When `readSequential` is true, `readFreeBehindPointer` points to a unit before the `readPointer`, and this unit is the first unit that a free-behind operation will free. When `writeSequential` is true, `writeFreeBehindPointer` points to a unit before the `writePointer`, and this unit is the first unit that a free-behind operation will free.

In addition to the above variables, the `SmarterLRUReplacementPolicy` class adds the following new methods. These methods implement the abovementioned optimizations.

1. *doWriteBehind* -
   First, it determines if write-behind is desirable. It does nothing if write-behind is not desirable. Write-behind is desirable if there is at least `clusterSize` units behind the `writePointer`. If write-behind is necessary, it determines which cluster should be written and it issues a single write operation to write these clusters. Unlike cluster read-ahead, there is little to be gained by dividing
the single write operation into smaller cluster-sized write operations. This is because applications usually do not wait for these write operations to complete. Figure 6.6 presents the simplified pseudo code for the `doWriteBehind` method.  

```
SmarterLRUReplacementPolicy.doWriteBehind()
    writeCount = writePointer - nextWriteBehind;
    if writeCount < freeBehindLag then
        return;
    writeCount = [ writeCount ÷ clusterSize ] × clusterSize;
    if writeCount = 0 then
        return;
    firstUnit = nextWriteBehind;
    lastUnit = nextWriteBehind + writeCount - 1;
    nextWriteBehind = lastUnit + 1;
    invoke my memory object cache to write firstUnit to lastUnit;
```

**Figure 6.6**: The pseudo code for `doWriteBehind`.

2. `doFreeBehind` -

First, it determines if free-behind is desirable. It does nothing if free-behind is not desirable. Free-behind is desirable if the memory object cache caches more units than `freeBehindLag`, and the free-behind operation can free at least `clusterSize` units. Freeing at least `clusterSize` units at a time helps to ensure proper clustering of write operations, just in case the units to be reclaimed contain dirty data. Figure 6.7 presents this method's simplified pseudo code.

3. `updateFreeBehindLag` -

It increases `freeBehindLag` if there is little memory contention and `freeBehindLag` is less than `maxFreeBehindLag`. It computes the elapsed time between the current invocation of this method and the last invocation of its `pageOut` method. A long elapsed time increases `freeBehindLag`.

4. `doReadAhead` -

It implements cluster read-ahead. Figure 6.7 presents its simplified pseudo code. A cluster becomes accessible once it has been paged in by the memory object cache.

The `SmarterLRUReplacementPolicy` class overloads the following methods of the `GroupLRUReplacementPolicy` class:

---

24To simplify the pseudo code, logic that handle concurrency control and unexpected conditions, such as file truncation, and non-sequential accesses, is excluded. For example, an application may have a loop that overwrites a particular file once every iteration. A smarter LRU replacement policy is appropriate for this file since access to the file is predominantly sequential. The smarter LRU replacement policy will encounter an unexpected non-sequential access at the start of each new iteration when the application writes to the beginning of the file. The actual implementation of `doWriteBehind` handles these unexpected conditions.
SmarterLRUReplacementPolicy::doFreeBehind( nextToFree, current )
if number of resident units ≤ freeBehindLag then
    return;
newNextToFree = current - freeBehindLag;
if newNextToFree - nextToFree < clusterSize then
    return;
firstUnit = nextToFree;
lastUnit = newNextToFree - 1;
nextToFree = newNextToFree;
invoke my memory object cache to free firstUnit to lastUnit;

Figure 6.7: The pseudo code for doFreeBehind.

SmarterLRUReplacementPolicy::doReadAhead()
lag = readAheadPointer - readPointer;
if lag ≥ readAheadLag or readAheadPointer ≥ numberOfUnits then
    return;
newReadAheadPointer = newReadAheadPointer + readAheadLag;
if newReadAheadPointer > numberOfUnits then
    newReadAheadPointer = numberOfUnits;
firstUnit = readAheadPointer;
endUnit = newReadAheadPointer - 1;
readAheadPointer = newReadAheadPointer;
do
    lastUnit = firstUnit + clusterSize;
    if lastUnit > endUnit then
        lastUnit = endUnit;
    invoke my memory object cache to page in first to lastUnit asynchronously
    (if not already paged-in);
    firstUnit = lastUnit + 1;
while firstUnit ≤ endUnit;

Figure 6.8: The pseudo code for doReadAhead.
1. **precacheRead.**
   It invokes the memory object cache to page in the first cluster if the cluster is not already resident.

2. **postCacheRead.**
   First, it updates `readSequential`, `readPointer`, `readFreeBehindPointer`, and `readAheadPointer`. If `readSequential` is false, it does nothing. Otherwise, it invokes `doReadAhead` to perform cluster read-ahead. Then, it invokes `updateFreeBehindLag` to increase `freeFreeBehindLag` if possible. Finally, it invokes `doFreeBehind`.

3. **postCacheWrite.**
   It updates `writeSequential`, `writePointer`, and `writeFreeBehindPointer`. If `writeSequential` is false, it does nothing. Otherwise, it invokes `updateFreeBehindLag` to increase `freeFreeBehindLag` if possible. Then, it invokes `doFreeBehind`. If `enableWriteBehind` is true, it invokes `doWriteBehind`.

4. **pageOut.**
   First, it decreases `freeBehindLag` if necessary. It decrease `freeBehindLag` when the elapsed time between the previous and current invocation of `pageOut` is small. If free-behind cannot reclaim any memory, it invokes the superclass’s `pageOut` method. Otherwise, it invokes either `doFreeBehind (readFreeBehindPointer, readPointer)` or `doFreeBehind (writeFreeBehindPointer, writePointer)`. The choice depends on the value of `readSequential`, `writeSequential`, the distance between `readPointer` and `readFreeBehindPointer`, and the distance between `writePointer` and `writeFreeBehindPointer`.

The main differences between this implementation and Mcvoy et al.’s implementation of the above optimizations and strategies are:

1. **Mcvoy et al.** added these optimizations to the UNIX file system. Hence, they are specific to the UNIX file system. In this framework, these optimizations are implemented in the `SmarterLRUReplacementPolicy` class. It is more general since any memory object can benefit from these optimizations by attaching a `smarter LRU replacement policy` to the memory object’s memory object cache. Hence, many file systems can benefit from these optimizations. This difference emphasizes the “mix and match” advantage of an object-oriented framework.

2. The `SmarterLRUReplacementPolicy` class maintains separate pointers for read, write, read free-behind, and write free-behind. Mcvoy et al.’s extensions only maintains single set of seek and free-behind pointers. The Choices implementation allows for independent free-behind operations for sequential read access and sequential write access.

---

25 The distance between two units is the number of units between these two units.
3. Mcvoy et al. implemented single-cluster read-ahead. Their cluster size is fixed and optimized specifically for the UNIX file system. The SmarterLRUReplacementPolicy class supports multi-cluster read-ahead. It is also more configurable and determines its optimization parameters, such as clusterSize, more dynamically from the system configuration.

AdaptiveLRUReplacementPolicy

The AdaptiveLRUReplacementPolicy class is a subclass of the SmarterLRUReplacementPolicy class. An adaptive LRU replacement policy takes advantage of the hints provided by applications and its memory object to optimize for the expected access pattern. Chapter 8 will describe this class in greater detail.

6.3.7.5 Multiple Inheritance

An approach for combining different cache replacement policies and various data transfer strategies has been presented in the previous subsections. However, this approach is less than ideal. This is primarily due to certain limitations in the Choices programming environment. This subsection discusses alternative approaches. In particular, it discusses how multiple inheritance would provide a better approach.

Given the single-inheritance limitation of the Choices programming environment, the current approach implements cache replacement policies in immediate subclasses of the MigrationPolicy class. Subclasses of these classes add new data transfer strategies. The problem with this approach that it is hard or very inconvenient to apply a single data transfer strategy to many different cache replacement policies. For example, it is difficult to add the data transfer strategies of the SmarterLRUReplacementPolicy class to another MigrationPolicy class, such as the FIFOReplacementPolicy class, even though these data transfer strategies are quite independent of the cache replacement policy. If the same data transfer strategies were to be added to the FIFOReplacementPolicy class, then a subclass of the FIFOReplacementPolicy class would have to be created. This subclass would contain essentially the same code that is present in the SmarterLRUReplacementPolicy class. In this approach, cache replacement policies are more “generic” than data transfer strategies.

An alternative approach is to make data transfer strategies more “generic” than cache replacement policies. This alternative implements data transfer strategies in immediate subclasses of the MigrationPolicy class, and derives from these classes to add cache replacement policies. However, this alternative has the same shortcoming as the current approach. It makes applying a single cache replacement policy to many data transfer strategies difficult. On the other hand, it does make applying a data transfer strategy to many cache replacement policies simple. This alternative is rejected because each MigrationPolicy class must have a replacement policy. Therefore, priority has been given to making cache replacement policies more “generic”.

80
A third approach considered, given the single-inheritance limitation, is to decompose a migration policy into component objects. Each of the component object would have a specific role or function. For example, classes of component objects might include ReplacementPolicy, ReadAheadOptimization, and WriteBehindOptimization. Like a memory object cache, a migration policy would delegate invocations to appropriate component objects. The primary advantage of this approach is that it permits dynamic run-time composition of migration policies from component objects of different classes. However, this approach has several significant problems. The main problem is that there is no clear decomposition of a migration policy object into component objects that would be general enough to implement most migration policies. Another problem is that run-time composition of migration policies may cause component objects with conflicting strategies or optimizations to be composed into a single migration policy. A possible solution to this problem is to provide some form of compatibility matrix or criteria to determine compatibility, but it introduces additional complexity.

A more ideal fourth approach is to take advantage of multiple inheritance, if available. In this approach, each cache replacement policy or data transfer strategy would be implemented by a subclass of the MigrationPolicy class. These subclasses overload the appropriate methods of the MigrationPolicy class as necessary to implement their individual functionality. Multiple inheritance provides a mechanism for “combining” these functionality to define a migration policy. For example, the optimizations implemented in the SmarterLRUReplacementPolicy class would be implemented in three subclasses of the MigrationPolicy class, known as the ReadAheadOptimization class, the WriteBehindOptimization class, and the FreeBehindOptimization class. A SmarterTransferStrategy class would be derived from these three classes. The SmarterLRUReplacementPolicy class would be derived from both the GroupLRUReplacementPolicy class and the SmarterTransferPolicy class. It would inherit its replacement policy from the GroupLRUReplacementPolicy class and its data transfer strategies from the SmarterTransferStrategy class. Similarly, a SmarterFIFOREplacementPolicy class could be derived both from the FIFOReplacementPolicy class and the SmarterTransferPolicy class. This SmarterFIFOREplacementPolicy class would have a FIFO replacement policy and the data transfer strategies of the SmarterLRUReplacementPolicy class. This approach allows both data transfer strategies and cache replacement policies to be equally “generic”.

Like the third approach, certain combinations could cause conflicts and compatibility problems. Unlike the third approach, knowledgeable operating system gurus combine these classes to design new migration policies. They should be able to identify and resolve conflicts, as well as, reject silly combinations. In addition, the compiler helps to identify conflicts and reject certain combinations. For example, if both the SmarterTransferStrategy class and the GroupLRUReplacementPolicy class implement the pageOut method, then a C++ compiler would not permit a subclass to be derived from these two classes without redefining the pageOut method. This forces the operating system guru to resolve any potential conflicts resulting from combining these two classes.
In summary, multiple inheritance would improve the framework. It reduces the effort required to create new migration policies from existing classes that implement various cache replacement policies and data transfer strategies.

6.3.8 MemoryObjectView

A memory object view represents a continuous region of a memory object. It is defined by an offset into a memory object and a length which determines the size of the region. In the new framework, only memory object views can be added to a domain. To map an entire memory object into a domain, a memory object view representing the entire memory object is added. The new framework needs Memory object views for the following reasons:

1. To allow large memory objects to be memory-mapped -
   When a memory object is too large to be memory-mapped because of a lack of virtual address space, memory object views allow parts of the memory object to be memory-mapped.

2. To allow efficient loading of applications -
   Choices stores the binary code that represents an application program in a persistent memory object. There are at least two sections in an application program; the text section and the bss section. The text section contains read-only executable code. A memory object view allows the text section of an application program to be made addressable by memory-mapping (no copying required). Without memory object views, the text section of an application program must be copied to a new memory object before it can be memory-mapped.

3. To allow a memory object to change its size after mapping -
   In the original virtual memory management system framework, the size of a memory object determines the amount of virtual address space reserved for the memory object in a domain when it is memory-mapped. This causes problems if a memory object changes its size while it is memory-mapped. Adding a memory object view to a domain instead of a memory object separates virtual address space reservation from the size of the memory object. While the size of a memory object may change, the size of a memory object view never changes. A memory object view also allows a memory object that is expected to grow in size to reserve more virtual address space in a domain than the current size of the memory object, by adding a memory object view that is larger than the memory object to the domain.

The abstract MemoryObjectView class has two concrete subclasses: the NormalMemoryObjectView class and the LockedMemoryObjectView class. An application uses a normal memory object view to map a
region of a memory object into a domain. The new framework uses locked memory object views to support I/O (refer to Section 6.3.11.1).

6.3.9 MemoryObject

The MemoryObject class has been revised significantly to support file system caching. In particular, concurrency control has been improved and new I/O methods have been added. This subsection presents these revisions.

Concurrency Control

In the original frameworks, there are two different locks that may be acquired for each physically addressable unit. Each physically addressable unit has lock. This lock is acquired if the physically addressable unit's locked attribute is true. The original memory object cache has a lock manager. This lock manager knows which units in the cache are locked. Locking a unit in the cache indirectly locks the physically addressable units that caches the unit. Having two independent locks per physically addressable unit complicates concurrency control. Programmers of the virtual memory management system must be careful to avoid deadlocks. It is also inefficient since one lock should be sufficient. In addition, these locks do not assist in concurrency control for the memory objects. As a result, the original memory objects did not have sufficient concurrency control to ensure integrity and consistency in a multi-processing environment.

The new framework addresses these concurrency control problems. It requires each memory object to have a lock manager. This lock manager is an instance of the Rangelock class. A new memory object cache no longer has its own lock manager, it uses the lock manager of its memory object. Sharing a single lock manager allows the memory object and the memory object cache to cooperate better with each other to maintain mutual consistency. Good cooperation is essential since a method in one of these objects will frequently invoke methods in the other to accomplish its task. For example, a memory object cache servicing a page fault may need to read data from its memory object.

In the original frameworks, the lock in each physically addressable unit prevents a physically addressable unit from being selected for replacement. In the new framework, this lock has been removed. It is no longer required. Once a physically addressable unit has been allocated to a memory object cache, the physically addressable unit belongs to the memory object cache. The memory object cache determines whether the physically addressable unit is eligible for replacement.

Since the memory object and the memory object cache, share a single lock manager, it is easier to eliminate deadlocks and concurrency control problems.
Cached versus Raw I/O

The original MemoryObject class has two I/O methods: the read and write methods. These methods read and write to the memory object directly. With the addition of file system caching, these methods are no longer sufficient. At least two sets of I/O methods are required. The first set of I/O methods move data directly between main memory and the memory object’s backing store, like the original read and write methods. The second set of I/O methods supports cached I/O. They read and write through a memory object cache if the memory object has a memory object cache.

In the new framework, the original read and write methods of the MemoryObject class have been renamed rawRead and rawWrite. The renamed methods transfer data directly between main memory and the memory object’s backing store. Then, new read and write methods are defined. If the memory object has a memory object cache, they invoke the cacheRead and cacheWrite methods of the memory object’s memory object cache. Otherwise, they invoke the memory object’s rawRead and rawWrite methods.

Support for Containers

As discussed in Section 5.3, a memory object container is backed by a memory object, known as the containing memory object, and it contains memory objects, known as the contained memory objects. A contained memory object translates an I/O request to one or more I/O requests to its containing memory object. For example, a disk is backed by a disk container and a disk container contains disk partitions. A disk partition backs a BSD container and a BSD container contains BSD inodes (refer to Figure 5.4). A BSD inode translates a rawWrite operation to one or more write operations on its containing disk partition. Similarly, a disk partition translates a rawWrite operation to a write operation on its containing disk.

The new framework permits both the containing and the contained memory objects to be cached, i.e. each has a memory object cache. This introduces a need to distinguish writes of uncommitted data from writes of committed data, much like the secondary caches do. Uncommitted data may be cached by the memory object cache of the containing memory object. On other hand, committed data should be written through the containing memory object’s memory object cache. For example, committed data from a BSD inode should be written through the memory object cache of the BSD inode’s disk partition.

In summary, the following methods write into a memory object:

---

26 The decision to rename the original read and write methods is based on the number of lines of source that must be changed. In this case, there are fewer references to these methods in the virtual memory management system than the file system.

27 Actually, a memory object container is backed by a persistent memory object.

28 Actually, a memory object container contains persistent memory objects.

29 Assuming disk partition is cached.
1. **write**

The file system invokes this method to write uncommitted data. The write method of the MemoryObject class implements its default behavior. If a memory object cache is attached, its default behavior invokes the cacheWrite method of the memory object cache to cache the new data. Otherwise, it invokes the rawWrite method of the memory object. Subclasses of the MemoryObject class may overload this method to modify its default behavior.

2. **writeThrough**

The file system invokes this method to write committed data. Usually, this method is invoked by the rawWriteThrough method of a contained memory object. The writeThrough method of the MemoryObject class implements its default behavior. If a memory object cache is attached, its default behavior caches the new data and writes the new data through the cache. It invokes the cacheWrite method of the memory object cache to cache the new data. Then, it invokes the synchronize method of the memory object cache to force the memory object cache to write the committed data through the cache. If the memory object is not cached, its default behavior invokes the rawWriteThrough method of the same memory object. Subclasses of the MemoryObject class may overload this method to modify its default behavior. An alternative behavior may invalidate the memory object cache and write the new data directly to the memory object.

3. **rawWrite**

The virtual memory management system invokes this method to write uncommitted data to a memory object. Usually, this method is invoked by the write method of the DummySecondaryCache class. The rawWrite method of the MemoryObject class implements its default behavior. If the rawWrite method is invoked on a contained memory object, its default behavior translates the rawWrite invocation to one or more write invocations on the containing memory object. A subclass of the MemoryObject class may overload this method to provide an alternate behavior.

4. **rawWriteThrough**

The virtual memory management system invokes this method to write committed data. Usually, this method is invoked by the rawWrite method of the DummySecondaryCache class. The rawWriteThrough method of the MemoryObject class defines its default behavior. If the rawWriteThrough method is invoked on a contained memory object, its default behavior translates the rawWriteThrough invocation to one or more writeThrough invocations on the containing memory object. This ensures that committed data is propagated and written through all the containing memory objects’ memory object caches. Like write, a subclass of the MemoryObject class may overload this method to provide an alternate behavior.
In summary, the revised memory object and memory object cache simplify concurrency control and supports file system caching. Unlike many file system designs, the new framework supports caching at each level in a hierarchy of containers.

6.3.10 MemoryObjectCachingContainer

*Choices* uses reference counting for garbage collection. When an application opens a file, it returns a pointer object to the application. Usually, this pointer object references a persistent object. When the application destroys this pointer object, the reference count of the persistent object is decremented. When the reference count reaches zero, the persistent object is destroyed. Hence, an application “closes” a file by destroying the pointer object returned by the file system. Destroying a persistent object also destroys its persistent memory object and its memory object cache since the persistent object references its persistent memory object and its persistent memory object references its memory object cache. Destroying a memory object cache flushes the persistent memory object’s cached data from main memory. This is not desirable since another application may open the same file again soon and the same data may be needed again.

The new framework addresses this problem with zombies. Zombies are objects that are not currently referenced and should be garbage collected. The MemoryObjectCachingContainer class is a subclass of the MemoryObjectContainer class. A memory object caching container caches zombies. When a persistent memory object contained in a memory object caching container is “closed”, its persistent memory object and its persistent object become zombies (with the help of a reference counting trick). A zombie persistent memory object allows its memory object cache to stay “alive” because it still holds a reference to its memory object cache. When an application opens the file again, the file’s zombie persistent memory object and persistent object will become referenced and be “revived”.

![Figure 6.9: The new MemoryObjectCachingContainer class and the class hierarchy. Additions and changes are shown in italics.

Figure 6.9 illustrates how the MemoryObjectCachingContainer class fits into the class hierarchy of the new framework. Subclasses of the MemoryObjectCachingContainer class inherits the zombie caching property of the MemoryObjectCachingContainer class. These subclasses usually represent containers that contain objects that are likely to be cached. The UNIXContainer class is a subclass of the Memory-
ObjectCachingContainer class since Choices usually caches UNIX inodes.\textsuperscript{30} The ArContainer class is not a subclass of the MemoryObjectCachingContainer class because it contains ar entries and ar entries are seldom cached.\textsuperscript{31} There is no benefit in caching zombie ar entries. Ar entries are not cached because they are contained in UNIX inodes and UNIX inodes are usually cached. Caching both an UNIX inode, whose persistent object is an ar container, and its contained ar entries would cache the same data twice. This is inefficient use of main memory.

In general, caching two persistent memory objects at two different levels in a container hierarchy is not desirable. However, there are some exceptions. One exception would be the caching a disk partition and its contained persistent memory objects. The memory object cache for the disk partition could have a special migration policy to exploit hardware specific characteristics of the physical disk device and specific knowledge of the disk partition’s data layout to improve performance. Such a migration policy suitable for a disk partition that holds a log-structured file system[93, 113, 114] would perform track caching, prefetching and write-behind may be It would determine the number of tracks to cache and prefetch based on the size of on-disk cache and the size of each log segment. It should also limit the amount of memory dedicated to track caching to reduce the amount of redundant data that are cached at different levels in the containment hierarchy at the same time.

Even though a disk partition could be cached, the disk container is not a memory object caching container. If disk containers were memory object caching containers, then a disk partition would become a zombie only when the partition is unmounted. In general, data in unmounted partitions are not likely to needed again soon. Furthermore, caching data in an unmounted partition may interfere with certain actions, like physical removal of the disk containing the partition or consistency checks on the partition (such as “fsck”).

With the introduction of zombie persistent memory objects, two new problems have to be addressed. The first problem is “how to reclaim system resources from zombies”? If all previously active\textsuperscript{32} but currently inactive\textsuperscript{33} persistent memory objects become zombies, then the number of zombies in a host can only grow. Eventually, there will be a large number of zombies and they will consume a lot of system resources. The primary resource of concern in this case is main memory used to maintain the state of each zombie (and the zombie’s related objects) and data cached by the zombie memory object caches. As implied by its name, the MemoryObjectCachingContainer caches zombie persistent memory objects. Like most kinds of caches, it must define and implement mechanisms to reclaim system resources that it consumes.

\textsuperscript{30}The default is to enable caching for UNIX inodes.
\textsuperscript{31}The default is not to cache ar entries.
\textsuperscript{32}An active persistent memory object is currently in use by some application or the operating system. It is equivalent to an inode with open file descriptors in UNIX.
\textsuperscript{33}An inactive persistent memory object is currently not in use by some application and the operating system.
The second problem is "how to deal with dirty data belonging to zombies?". When a persistent memory object becomes inactive, it can either become a zombie or be destroyed immediately. If the persistent memory object is not contained in a memory object caching container, then it and its related objects are destroyed immediately. This causes its memory object cache to synchronize its dirty data and flush all its cached data from main memory. If the persistent memory object is contained in a memory object caching container, then the persistent memory object becomes a zombie. The zombie persistent memory object's memory object cache may cache dirty data. If this dirty data gets synchronized only when the memory object cache is destroyed, then this dirty data will not be synchronized as long as the persistent memory object is not consistently revived before the the MemoryObjectCachingContainer has a chance to destroy it. This is not desirable since the persistent memory object's cached changes will be lost when the system crashes.

There are two solutions to this problem. One solution makes sure that only migration policies with reasonable guarantees regarding dirty data are employed by persistent memory objects contained in a memory object caching container. In the new framework, a memory object can determine which kinds of migration policies are compatible. The MemoryObjectCachingContainer class provides no additional support for synchronizing dirty data. It depends on suitable migration policies to provide the necessary synchronization support. An other solution builds a mechanism for synchronizing dirty data into the MemoryObjectCachingContainer class. Both solutions are feasible in this framework. The new framework favors the latter solution because the latter solution allows for more flexible "mixing and matching" of migration policies.

The MemoryObjectCachingContainer class defines the default strategies that address the two above-mentioned problems. Subclasses of the MemoryObjectCachingContainer class may implement other strategies. The remainder of this subsection presents how the the MemoryObjectCachingContainer class solves these problems. First, it presents the strategy for reclaiming system resources from zombies. Then, it discusses the strategy for synchronizing dirty data.

**Reclaiming Resources from Zombies**

The MemoryObjectCachingContainer class classifies its zombies into three different categories. These three categories are fresh, stale, and extern. When a persistent memory object contained in a memory object caching container becomes inactive, it becomes a fresh zombie. It remains a fresh zombie for a fixed period of time, known as the persistent memory object's `keepFreshTime`. The MemoryObjectCachingContainer does not reclaim system resources occupied by fresh zombies. In other words, it does not destroy fresh zombies.

---

34 This depends on the migration policy of the memory object cache.
If a fresh zombie remains a zombie for keepFreshTime, then it becomes a stale zombie. A memory object caching container reclaims system resources, especially main memory, by eliminating stale zombies. It eliminates a stale zombie by either destroying the stale zombie or transforming the stale zombie into an extern zombie. The action taken depends on the zombie’s becomeExtern attribute. If zombie’s becomeExtern attribute is false, the memory object caching container destroys the zombie. As mentioned earlier, destroying the zombie persistent memory object also destroys the zombie’s related objects. This flushes the zombie’s cached data from main memory, and reclaimed main memory occupied by these objects. If the zombie’s becomeExtern attribute is true, the memory object caching container does not destroy the stale zombie. The memory object caching container flushes the zombie’s cached data from main memory, and transform the stale zombie into an extern zombie.

In either case, the memory object caching container reclaims main memory used to cache the stale zombie’s data. A memory object caching container eliminates its stale zombies in LRU order, i.e. the oldest stale zombies are eliminated first. A memory object caching container starts to eliminate its stale zombies when one of the two following conditions is present.

1. The total number of stale zombies in the system exceeds a specified threshold. This condition limits the amount of system resources allocated to stale zombies.

2. There is little free memory left on the host. The store has been modified to notify the MemoryObjectCachingContainer when this condition is present. This allows the MemoryObjectCachingContainer to reclaim memory by eliminating stale zombies.

The rationale behind this strategy is to allow the MemoryObjectCachingContainer to reclaim memory from stale zombies before any memory pager is activated. This strategy assumes that a stale zombie is not likely to be active or needed again in the near future. Therefore, each zombie’s keepFreshTime should be set to an appropriate value to reflect this assumption.

This strategy has several advantages.

1. **Lower CPU utilization** -
   The MemoryObjectCachingContainer maintains an LRU queue of all the stale zombies in a host. It can quickly identify the oldest stale zombie and reclaim the memory used to cache the zombie’s data. This is more CPU efficient than activating the memory pagers to initiate cache replacement.

2. **Improved I/O efficiency** -
   When the MemoryObjectCachingContainer eliminates a stale zombie, it flushes the zombie’s entire memory object cache. This allows the memory object cache to preserve the spatial relationships between the units in the cache when writes are necessary. Preserving spatial relationships improve transfer sizes. The memory object cache coalesces writes to sequential units. It may also improve
storage layout efficiency. For example, the BSD file system attempts to place sequential units on consecutive disk blocks so that these units can be read back using a single read operation.[72]. Memory pagers tend to ignore spatial relationships even when consecutive units of a persistent memory object are chosen for replacement.

The MemoryObjectCachingContainer eliminates stale zombies one at a time. Hence, it groups I/O bursts by memory object. This reduces disk seeks on file systems that place blocks belonging to the same file close together. For example, the BSD file system divides a disk partition into cylinder groups[72]. Each cylinder group contains both inode meta-data and data. The BSD file system attempts to minimize disk seeks by placing an inode’s meta-data and data on the same cylinder group when possible. Memory pagers tend to generate shorter I/O bursts to the same persistent memory object and to scatter I/O among persistent memory objects.

3. Better LRU approximation -

The MemoryObjectCachingContainer also improves the resolution and accuracy of LRU cache replacement when LRU is desirable. It knows exactly when a persistent memory object becomes a zombie. Since a zombie is inactive by definition, the MemoryObjectCachingContainer also knows when the zombie’s cached data was last referenced. The keepFreshTime of a zombie is typically in the tens of seconds. The maximum resolution of the LRU pager is at most a few seconds. Hence, the MemoryObjectCachingContainer performs LRU cache replacement on units that have not been referenced for at least keepFreshTime seconds. The LRU pager is activated only when no additional main memory can be reclaimed from stale zombies. Hence, the LRU pager performs replacement on units that have been referenced within the last keepFreshTime seconds.

In summary, the MemoryObjectCachingContainer implements a two level cache replacement policy. The MemoryObjectCachingContainer performs the first level of replacement by reclaiming memory from stale zombies that are not likely to be used in the future. When there are no more stale zombies, it depends on the memory pagers to reclaim memory.

The first level of replacement may be disabled for an individual persistent memory object by setting the persistent memory object’s keepFreshTime to a very large value.\(^{35}\) However, this introduces another problem. If all persistent memory objects have large keepFreshTimes, then these persistent memory objects never become stale zombies and they will never be destroyed. To solve this problem, the MemoryObject-CachingContainer forcefully “downgrades” the oldest fresh zombie to a stale zombie when the number of fresh zombies in the system exceeds a specified threshold. This threshold is computed from the size of main memory, and the approximate main memory requirement of a typical zombie and its related

\(^{35}\) On a 32-bit processor, the largest keepFreshTime is approximately 68 years.
objects, excluding the zombie’s cached data. This threshold is large enough so for fresh zombies with reasonable *keepFreshTimes* to expire before the threshold is exceeded.

As mentioned earlier, a stale zombie eventually becomes a extern zombie if its *become Extern* attribute is true. An extern zombie does not cache any of its data in main memory to minimize its memory consumption. It is destroyed under external control. The `MemoryObjectCachingContainer` class provides a method to destroy zombies. The extern zombie keeps its persistent memory object and its related objects “alive” because these objects may contain information that should last longer than in-memory cached data. For example, a zombie may have a secondary cache that uses in-memory data structures to keep track of units cached in the secondary cache. If the zombie were destroyed, then these data structures would be lost. Such a secondary cache initiates the destruction of its extern zombie persistent memory object when it is reclaimed. For example, the `UNIXNodeSecondaryCache` class reclaims its UNIX inode secondary caches when one of its specified thresholds is exceeded. To prevent unbounded growth in the number of extern zombies, the `MemoryObjectCachingContainer` also limits the number of extern zombies that a host can support.

**Synchronizing Dirty Data**

The `MemoryObjectCachingContainer`’s strategy for synchronizing dirty data is to synchronize a persistent memory object within *syncDelayTime* after the persistent memory object becomes inactive. *SyncDelay-Time* is an attribute of a persistent memory object.

Logically, each persistent memory object has a timer, known as its *syncDelay* timer. When a persistent memory object becomes inactive for the first time, its *syncDelay* timer is initialized to its *syncDelayTime* and started. This timer does not stop when the persistent memory object becomes active again. When the timer expires, the `MemoryObjectCachingContainer` synchronizes the persistent memory object.

A persistent memory object may become inactive while its *syncDelay* timer is running. This occurs when the persistent memory object becomes active and inactive again after the timer has been started but before the timer expires. When this occurs, the `MemoryObjectCachingContainer` does not re-initialize the timer to the persistent memory object’s *syncDelayTime*. The timer continues to run with whatever time is left on it. If a persistent memory object’s *syncDelay* timer were to be re-initialized whenever the persistent memory object becomes inactive, then the persistent memory object might never be synchronized. The persistent memory object would never be synchronized if the persistent memory object were consistently reactivated before its *syncDelay* timer could expire.

This strategy reduces writes to persistent memory objects that will be deleted soon. When a persistent memory object is deleted, its cached changes are discarded. Hence, no synchronization is required. In the most ideal case, such a “transient” persistent memory object would be deleted before its *syncDelay* timer expires. This strategy takes advantage of file access behaviors observed on several UNIX-like systems.
Ousterhout et al. [80] and Baker et al. [4] observed that a large percentage of new files are deleted very soon after creation. In other words, most new files have very short lifetimes and the syncDelayTimes of these transient files should reflect their expected lifetimes.

In the current implementation, the syncDelayTime of a persistent memory object is the same as the the persistent memory object’s keepFreshTime. This implementation prefers a “transient” persistent memory object to be deleted before it becomes a stale zombie. If a “transient” persistent memory object became a stale zombie, then it would be eligible for reclamation and its cached changes might be synchronized before it is deleted. This implementation has another benefit. It ensures that all stale zombies are synchronized. This reduces the latency required to reclaim memory from stale zombies since no writes are required before before reclaiming memory from stale zombies.

Summary

In summary, the MemoryObjectCachingContainer class increases the lifetime of cached data. It allows the virtual memory management system to cache data belonging to an inactive persistent memory object. It keeps an inactive persistent memory object “alive” as a zombie. It also defines the default strategy for reclaiming system resources from zombies, and the default strategy for synchronizing dirty data.

6.3.11 Miscellaneous Issues

The new framework also addresses the following shortcomings of the original frameworks:

6.3.11.1 I/O fragmentation

The original frameworks cause a multi-page\textsuperscript{36} I/O operation to be fragmented into several page-sized I/O operations. This is extremely inefficient.

In order to understand this problem, a brief description of how the original memory object cache transfers data between main memory and a memory object is necessary. When a memory object cache reads data into a continuous virtual address region, it must first read the data into main memory before adding the appropriate virtual-to-physical address mappings to the address translation of the faulting domain. If memory object cache adds the virtual-to-physical address mappings first, then other processes in the same domain may access this virtual address region before the actual data has been read. As mentioned earlier, the original memory object cache invokes the read method of its memory object\textsuperscript{37} to read data into main memory. This read method takes a virtual address, that points to a buffer, as one

\textsuperscript{36}To simplify the discussion in this subsection, a page is synonymous to a physically addressable unit.

\textsuperscript{37}In the new framework, the memory object cache invokes the read method of its top-most secondary cache.
of its argument. Hence, the memory object cache needs to map pages involved in the read operation into another virtual address space before invoking the read method of its memory object.

Similarly, the memory object cache also needs to map pages involved in an write operation into another virtual address space before invoking the write method of its memory object.\(^{38}\) Before the memory object cache writes an unit to its memory object, it must remove the unit from all address translations in the host. Otherwise, this unit may be modified while the write operation is in progress. This is a problem because the written data may or may not reflect the changes made while the write operation is in progress. If the memory object cache is about to remove the unit from main memory, then the unit may contain dirty data when it is removed. If the memory object cache is synchronizing the contents of the unit, then the unit may contain dirty data after the unit's referenced status has been cleared.

The original virtual memory management system framework maps each page of main memory to a well-known address. Typically, the well-known address of a page is the physical address of the page.\(^{39}\) These mappings are known as the one-to-one mappings. Since continuous virtual addresses are seldom mapped into continuous physical addresses, the memory object cache must fragment a multi-page read into smaller page-sized reads to transfer data into a continuous virtual address region. This is a problem because a single large transfer is more usually efficient than many small transfers.

A solution to this problem is to map every active memory object into the I/O domain. The I/O domain manages an virtual address space that is dedicated to data transfer between main memory and memory objects. When a memory object cache has to transfer data between main memory and its memory object, it uses virtual addresses in the I/O domain. Mapping every active memory object into the I/O domain ensures that separate virtual addresses will always be available for I/O, and consecutive units of a memory object are mapped into consecutive virtual addresses. Hence, I/O fragmentation is not a problem. Unfortunately, this solution is inadequate. The total virtual address space of the I/O domain limits the number of memory objects that can be mapped, and limits the size of the largest memory object.

Another solution to this problem is to map pages that are involved in an I/O operation into the I/O domain. Instead of reserving virtual address space in the I/O domain for an entire memory object, a memory object cache reserves virtual address space in the I/O domain only for the part of its memory object that is actually involved in an I/O operation. The memory object cache reserves virtual address space in the I/O domain by mapping a locked memory object view into the I/O domain. This locked memory object view holds units of the memory object that are involved in the I/O operation. This locked memory object view is also known as an I/O window.

\(^{38}\)In the new framework, the memory object cache invokes either the write or the rawWrite method of its top-most secondary cache.

\(^{39}\)The exception is Virtual Choices.
The new framework implements the latter solution. In addition, it has several optimizations that minimize overhead associated with the mapping and un-mapping of memory object views. In the first implementation, each memory object cache caches a single I/O window. In other words, a memory object cache does not remove or unmap the I/O window from the I/O domain upon completion of an I/O operation. It attempts to reuse the same I/O window for the next I/O operation if possible. It unmaps its cached I/O window and creates a new I/O window only when the current I/O window cannot be reused. This occurs when a new I/O operation requires a different part of the memory object to be mapped.

To increase the probability of an I/O window cache hit, the memory object cache creates I/O windows that are larger than necessary. For example, a memory object cache can create an I/O window that is 16 pages in length to reserve 16 pages of virtual address space even though the actual I/O operation only requires a single page of virtual address space to be reserved. The I/O window size attribute of a memory object cache determines the size of I/O windows created by the memory object cache. The I/O window size of a memory object cache is usually a multiple of the optimal I/O size of the memory object cache’s memory object.

There are several problems with the initial implementation. One problem is reduced concurrency since I/O operations that cannot share the same I/O window must be serialized. Another problem is that the data structures used to maintain an I/O window per memory object cache consumes too much memory. Furthermore, the amount of virtual address space available to the I/O domain may also be problem if there are too many active memory object caches that acquire large I/O windows.

The new framework addresses these problems by maintaining a system-wide cache of I/O windows. In other words, all the memory object caches share a pool of I/O windows. The IOWindowCache class implements this cache, and there is an I/O window cache per host. When a memory object cache needs an I/O window, it invokes the acquire method on the I/O window cache to acquire the desired I/O window. This method searches the I/O window cache for the requested I/O window. If the requested I/O window is found in the cache, it increments the reference count of the I/O window found. When the memory object cache no longer needs a previously acquired I/O window, it invokes the release method on the I/O window cache to decrement the I/O window’s reference count. If the acquire method does not find a requested I/O window in the cache, it removes the least recently used I/O window with a zero reference count. Then, it adds the requested I/O window to the cache, and increments the newly created I/O window’s reference count.

In summary, the new framework solves the I/O fragmentation problem of the original frameworks. The I/O domain and I/O windows eliminate fragmentation, and the I/O window cache reduces overhead associated with mapping and un-mapping of I/O windows.
6.3.11.2 Block Allocation

A large I/O request consisting of many units may have to be fragmented to several smaller disk I/O operations because the blocks required to satisfy the request are not placed consecutively on disk. For example, a 10-unit write request will be fragmented into 2 separate disk writes if the first 5 units and the last 5 units are located on different parts of the disk. To minimize fragmentation of large I/O requests, the block allocation algorithms of *Choices*’ implementation of UNIX file systems, such as the AIX file system, the BSD file system, and the SVID file system, have been revised to place consecutive units of a file on consecutive disk blocks whenever possible. This prevents large transfers originating from a memory object cache from being fragmented by these file systems.

6.3.11.3 The System Pager

The system pager has been examined and revised to improve utilization of available memory and to improve performance. The original system pager starts to page once half of available memory is used. It is also a performance bottleneck because it does not permit paging and allocations from the *store* to take place at the same time. Once the pager is activated, all subsequent allocation requests are blocked until the pager has recovered enough memory to reach a preset low watermark. This is bad because allocation requests are blocked even though there is enough free main memory available to satisfy the allocation requests. The new virtual memory management system allows more main memory to be allocated before the system pager is activated. The revised system pager also does paging in the “background”, i.e. allocation requests that can be satisfied will not block while the pager is paging.

6.4 Summary

This chapter presented the goals for flexible file system caching service. It also described an architecture for such a service and discussed how the architecture addresses these goals. To demonstrate the viability of the proposed architecture, it also presented an object-oriented framework that implements this architecture.

In summary, the following features of the architecture and the framework support flexible file system caching:

1. *Secondary cache* -

   *Secondary caches* exploit the memory hierarchy on a host. They increase the amount of memory available on a host for caching. Furthermore, a stack of *secondary caches* can be dynamically reconfigured, i.e. *secondary caches* can be inserted and removed from the stack at any time.
2. *Migration policy* -
Each cached file can have its own migration policy. A file’s migration policy determines the file’s cache replacement policy and its data transfer strategies. These data strategies include read-ahead, write-behind, and free-behind. In addition, the framework allows a file to change its migration policy at run-time.

3. *Memory object container* -
The new framework allows caching at each level in the container hierarchy. For example, disk caching can be enabled by attaching a memory object cache to a disk. Disk partition caching can be enabled by attaching a memory object cache to a disk partition. Similarly, a BSD inode in a disk partition can be cached by attaching a memory object cache to the BSD inode.

In addition, various performance optimizations have also been discussed. Some of these optimizations provide significant performance gains. Chapter 9 will discuss these optimizations and their performance benefits.
Chapter 7

Distributed File Access

This chapter describes the Choices distributed file system. First, it presents the design goals of the distributed file system. Then, it describes extensions to the Choices file system framework to support distributed file access. Finally, it also describes a few unique and important optimizations in this distributed file system.

7.1 Goals

The goals of the distributed file system are:

1. *Uniformity* -
   The distributed file system should allow uniform use of method invocation to access both local and remote files.

2. *Generic support* -
   The Choices file system supports user-defined persistent objects. The distributed file system should provide distributed access to both system-defined and user-defined persistent objects.

3. *Maximum reuse* -
   The distributed file system should reuse any suitable mechanisms introduced to support persistence.

4. *Performance* -
   The distributed file system should support flexible file system caching. This includes client and server caching. It should also allow hints to help select appropriate caching strategies.
7.2 Distributed File Access Strategies

Several possible extensions that provide distributed file access have been evaluated [64]. The Choices distributed file system implements two of these extensions. These extensions are remote access to persistent object and remote access to persistent storage.

As mentioned in Section 5.3, each Choices file is represented by a persistent memory object and a persistent object. The persistent memory object represent the file’s persistent logical memory or persistent storage. The persistent object encapsulates and interprets the data held in the persistent logical memory.

The first extension is remote access to persistent object. It permits remote access to the methods of the persistent object. A remote client remotely invokes the methods of a persistent object using an RPC-like[7] scheme. For example, a remote client invokes the add and remove methods of a remote memory object dictionary to manipulate the dictionary. Implementing this extension is straightforward; a persistent object is represented by a proxy object[105] on each client.

![Diagram of remote access to persistent object]

**Figure 7.1:** Remote access to persistent object

Figure 7.1 illustrates an example of this extension. In this example, the file is a directory in a BSD file system. Its proxy on the client is a remote dictionary. Method invocations on the remote dictionary is forwarded to the BSD directory on the server. NFS implements directory operations in a similar way.

Major advantages of this extension are that it preserves the exact semantics of the Choices file and does not require additional concurrency control support from the distributed file system. A disadvantage is that network latency, parameter marshaling, and server contention can make it inefficient. The server for a persistent object may become a bottleneck since the server is required to handle every method invocation on the persistent object. This extension also does not support client caching.

98
The second extension is remote access to persistent storage. It permits remote access to the persistent storage of a *Choices* file. This is also accomplished through the use of proxy objects. A proxy object, known as a remote memory object, represents a remote persistent memory object on each client. A client can access the persistent storage of the remote persistent memory object by invoking on the I/O methods of the client’s remote memory object. Alternatively, it can memory-map the remote memory object and access the persistent storage using memory reads and writes. A remote memory object is just like any other memory object. It can be cached by attaching a memory object cache to itself. This allows a client to cache data from a remote persistent memory object. In summary, this extension supports client caching of remote file data.

![Diagram of remote access to persistent storage](image)

**Figure 7.2**: Remote access to persistent storage (or persistent memory object)

Figure 7.2 illustrates an example of this extension. In this example, the *Choices* file is a normal file in the BSD file system. Its persistent object is a persistent array. Method invocations on the remote memory object is forwarded to the BSD inode on the server. The client invokes the I/O methods of the remote memory object to transfer persistent data in and out of the remote memory object’s memory object cache. LOCUS uses a similar approach to cache directory data[106].

The main advantage of this extension is that it permits client caching. However, remote access to persistent storage also introduces additional cache coherence and concurrency control problems when more than one client cache the same file. There are many ways to address these problems. These problems are well understood and a solution suitable for *Choices* has been proposed[64].

These two extensions achieve the first three goals. They uniformly use remote invocations to access remote files. They support both system and user defined persistent objects. Finally, they extend the
existing *Choices* frameworks and reuse existing classes. Section 7.5 addresses the fourth goal which is good performance.

### 7.3 Implementation

![Diagram](Image)

**Figure 7.3:** The distributed file system proxy classes in the *Choices* class hierarchy. Additions are shown in *italics*.

Figure 7.3 illustrates the additions to the *Choices* class hierarchy to support distributed file access. By default, *Choices* uses remote access to persistent object to access dictionaries and containers. In other words, the client accesses all remote *memory object dictionaries* through proxy dictionaries, known as *remote dictionaries*. It accesses all remote *memory object containers* through proxy containers, known as *remote containers*. This simplifies cache coherence and concurrency control. Furthermore, it is also convenient and easy to implement. All subclasses of the *MemoryObjectDictionary* class have the same public interface as the *MemoryObjectDictionary* class. Hence, a single *RemoteDictionary* class can represent every subclass of the *MemoryObjectDictionary* class. The *RemoteDictionary* class is itself a direct subclass of the *MemoryObjectDictionary* class. Similarly, all subclasses of the *MemoryObjectContainer* class have the same public interface as the *MemoryObjectContainer* class. Hence, a single *RemoteContainer* class can also represent every subclass of the *MemoryObjectContainer* class. The *RemoteContainer* class is actually a subclass of the *MemoryObjectCachingContainer* class (refer to Section 6.3.10). This allows a *remote container* to cache zombies.

By default, *Choices* uses remote access to persistent storage to access all other kinds of files. In other words, the client accesses all these files by accessing their persistent storage via *remote memory objects*. This reduces the number of kinds of proxy objects that the distributed file system must implement. This is because these files may be represented by different kinds of *persistent objects* with diversely different interfaces. If remote access to persistent object was employed, then the developer of each new
kind of persistent object must also implement a new kind of remote proxy for these persistent objects.\(^1\) Remote access to persistent storage eliminates this problem since all subclasses of the PersistentMemoryObject class have the same public interface. Hence, a single RemoteMemoryObject class can represent all subclasses of the PersistentMemoryObject class. The RemoteMemoryObject class is itself a subclass of the PersistentMemoryObject class. In Choices, this extension is most commonly used to access normal (UNIX) files represented by the PersistentArray class.

Each PersistentObject class may override its default remote file access extension. However, non-dictionary and non-container persistent objects must provide their own proxies if they choose to use remote access to persistent object.

### 7.4 Protocol and Cache Consistency

The current implementation employs a stateless protocol like the NFS protocol. In general, a stateless protocol does not provide full cache coherence. However, a stateless protocol is easier to implement on both the client and the server than a stateful protocol. An early prototype of the Choices distributed file system employed a stateful protocol\(^2\). This stateful protocol ensures cache coherence but it complicates recovery from client and server failures. Since the primary focus of this thesis is to investigate various caching strategies, a stateless protocol has been selected. It has fewer protocol related issues and problems.

The Choices distributed file system protocol is loosely based on the NFS protocol\(^3\). This protocol is based on RPC. It is a request reply protocol. The Choices distributed file system maps each a public operation of a proxy class to a RPC operation. Like the NFS protocol, a client re-transmits a request to the server until a reply has been received from the server. Exponential retransmission time back-off avoids network congestion.

The Choices distributed file system provides loose cache coherence for files that have been cached by a client. Each file in the distributed file system has a logical version number. In the case of the BSD file system, a counter in each file's inode provides the file's logical version number. The BSD file system increments this counter whenever the inode has zero links, i.e. whenever the inode has been deleted. Other file systems may use the last modification time of a file as the logical version number of the file. The distributed file system depends on this logical version number to check the consistency of the client cache. When a client opens a file for the first time, it obtains the logical version number of the file from the server. No additional checks take place while the file is active. In most cases, the file becomes a zombie when it is closed.\(^4\) On subsequent re-activations, the the client checks the consistency of the

---

\(^1\) Assuming that distributed access is required for this kind of persistent objects.

\(^2\) The RemoteContainer class is a subclass of the MemoryObjectCachingContainer class.
file’s cache by comparing the logical version number of the cached file with the logical version number of the file on the server. If these logical version numbers do not match, the client invalidates the file’s cache. Otherwise, the file’s cache is assumed to be valid.

If the file changed on the server while the file is active on the client, then the server rejects operations requested by the client. The client transmits the client’s perceived logical version number of the file with each request. The server compares the client’s perceived logical version number of the file with the actual logical version number of the file. If these logical version numbers do not match, the server does not perform the requested operation. It sends a reply to the client indicating that there is an inconsistency. The client passes the inconsistency error to the application as an I/O error. Otherwise, the server performs the requested operation.

The above cache consistency semantics is sufficient for this distributed file system since no experiments with concurrent write sharing will be conducted on the distributed file system. Concurrent write sharing are also rare[4, 8, 80] in most computing environments.

### 7.5 Optimizations

The following optimizations help to achieve good performance (the fourth goal):

#### 7.5.1 Asynchronous Operations and Continuations

Like most conventional distributed file systems[60], the Choices client and the Choices server interact using a request-reply protocol. The request identifies the operation desired and the reply contains the result of the operation. The conventional model for supporting such a protocol needs two threads for each operation; a client thread and a server thread. The client thread initiates a remote operation by sending a request message and blocks until it receives the reply. When the request arrives at the server, it is assigned to a server thread. The server thread performs the requested operation on the client’s behalf and transmits the result of the operation to the client. The client thread unblocks when it receives the reply from the server.

This model has a few shortcomings. It ties up system resources unnecessarily and reduces the number of concurrent operations. These shortcomings became apparent with the introduction of the read-ahead, write-behind, and free-behind data transfer strategies (described in Section 6.3.7.4). These strategies increase data transfer sizes. They can also generate a lot of asynchronous client requests because of aggressive read-aheads and write-behinds. Since the client threads are likely to block waiting for replies from the server and the server threads are likely to block waiting for disks, these threads become exhausted easily. Furthermore, the system resources occupied by these threads are wasted when these threads are blocked.
The rest of this subsection explains the shortcomings of this model in greater detail and presents a better alternative model based on continuations. First, it discusses the client. Then, it discusses the server. It also illustrates the differences between the traditional thread model and the new continuation model. Finally, it presents some performance data and implementation details.

7.5.1.1 The Client Side

An I/O operation is synchronous when the client thread that initiated the I/O operation must block until the operation has completed. For example, a read operation resulting from a page fault is a synchronous operation since the faulting thread cannot proceed until the faulting page has been fetched from the server. A read operation resulting from an application thread calling the read system call is also synchronous since the application thread cannot proceed until the requested data has been read and copied into its buffer. If all client initiated I/O operations were synchronous, then the thread that initiated the synchronous operation could send the request to the server and wait for the reply.

Unfortunately, synchronous I/O operations alone are not sufficient. Asynchronous I/O operations are required to achieve good performance. An I/O operation is asynchronous if the client thread that initiated the operation does not block while the operation is in progress. The client thread continues executing. Asynchronous I/O operations are important because the the read-ahead and write-behind data transfer strategies need them[73].

The Problem

In the traditional thread model, a distributed file system usually creates a pool of threads to support asynchronous operations. They are known as the asynchronous support threads. Depending on the file system, the number of threads in this pool may be fixed or variable. For example, Sun’s NFS implementation has a fixed number of asynchronous support threads.\(^3\) The first implementation of the Choices distributed file system also has a fixed number of asynchronous support threads.

When a thread on the client initiates an asynchronous I/O operation, it hands the I/O request to an available asynchronous support thread.\(^4\) This asynchronous support thread issues the remote I/O operation and blocks waiting for the reply on the initiating thread’s behalf. This is a problem since the maximum number of concurrent asynchronous I/O operations that a client can issue is limited to the total number of threads in the fixed-size pool.

One solution to this problem is to create more asynchronous support threads when the pool is exhausted. This solution is not desirable for two reasons. Firstly, each additional thread consumes

---

\(^3\)They are known as the “bio” daemons.

\(^4\)An available asynchronous support thread is one that is not currently processing a request or blocked waiting for the reply.
additional system resources. In particular, each additional thread consumes memory for its stack and other state maintained by the kernel, such as its context. In most systems, the stack must be at least one page. When a thread blocks while waiting for a reply from the server, it is consuming resources that could be put to better use. For example, the memory occupied by the thread stack can be used cache file system data.

Secondly, file access is bursty by nature[4, 80, 95]. Threads created to meet peak demands during bursts will be under-utilized the rest of the time. Baker et al. [4] and Ruenmller et al. [95] suggest that hundreds of threads may be needed in the near future to meet peak demands during bursts. If these threads were not ‘garbage collected’, the system resources they consume would be wasted. On the other hand, it is hard to predict when the next burst will occur. Premature or aggressive garbage collection could be costly because new threads would be created again for the next burst.

In summary, the thread model consumes too much system resources, in particular, memory. It also does not handle bursts very well.

The Solution

A better model is the continuation model. A continuation is an independent data structure that tracks the state of an operation. Continuations are cheap to create and destroy. Usually, they are allocated on the heap and consume very little memory. Most continuations in Choices consume less than 64 words\(^5\) of memory. The continuation model consumes much less system resources than the thread model.

In the continuation model, each asynchronous client operation has a continuation. When a client thread initiates an asynchronous operation, it creates a request message and a continuation. It queues the request message for transmission and registers the continuation with the distributed file system. Typically, each remote I/O operation has an unique request identifier, the distributed file system uses this identifier as a key to register and locate a continuation. When the client receives the reply to the request, the I/O operation’s continuation is located and executed. This continuation contains sufficient information to un-marshall the reply message and to complete the request.\(^6\)

Initiating an Asynchronous Operation

In Choices’ initial implementation of the continuation model, an initiating client thread hands its asynchronous operation to another thread, known as the starter thread. This allows the initiating thread to continue executing without waiting for the request message to be constructed and transmitted. The starter thread constructs the request message, builds the continuation, and queues the request message for transmission on the initiating thread’s behalf. Once the request message has been queued for trans-

\(^5\)On a 32-bit processor, like SPARC, a word is 32 bits or 4 bytes.

\(^6\)This usually means releasing locks that may be held.
mission, the starter thread is ready to handle another asynchronous operation. Since Choices has been tuned to minimize the probability that a starter thread may block, each client need maintain only one or two starter threads.

The problem with this implementation is that it does not perform well on an uniprocessor and it may be sensitive to processor scheduling overhead on a multi-processor. On an uniprocessor host, the starter thread may have a lower priority than the initiating thread. This means that the starter thread will not execute until the initiating thread has blocked. This is not desirable since we would like the asynchronous operation to be started as soon as possible to increase concurrency and minimize latency. The earlier the server receives the request, the earlier the server can start to process the request. In the worst sequential access case, the starter thread will not have a chance to issue the asynchronous operation for a read-ahead until the initiating thread has blocked waiting for the data requested by the read-ahead. This may not be a problem on a multi-processor host if one of the many processors is idle and available to run the starter thread. However, a processor may not be always available.

Hence, to ensure that an asynchronous operation will be started as soon as possible, it is necessary to give the starter thread a higher priority than the initiating thread. This is extremely inefficient on an uniprocessor host since the processor will have to switch to the starter thread immediately and then switch back to the initiating thread later. It is more efficient to allow the initiating thread to construct the request message, build the continuation, and queue the message for transmission. This eliminates the need for the starter threads and the context switching overhead associated with them.

Even though starter threads provide more concurrency on a multiprocessor host, the benefits of the additional concurrency are limited. Concurrent execution of the initiating thread and the starter thread is possible only when the starter thread is scheduled to execute immediately. If no idle processors are available, concurrent execution incurs additional latency and processor overhead to preempt some other thread. If an idle processor is available, concurrent execution incurs additional latency to load the context of a starter thread. Since the amount of time required to start an asynchronous operation is not much more than the time needed to schedule another thread for immediate execution, the additional concurrency that can be achieved by starter threads is offset by the scheduling and thread switching overhead. Hence, on a multiprocessor, it is still appropriate for the initiating thread to construct the request message, build the continuation, and queue the message for transmission.

In the current implementation, starter threads have been eliminated. This is appropriate for Choices because the Choices file system framework has been tuned to minimize the probability that a thread may block while initiating an asynchronous operation. In other distributed file systems, starter threads might be more appropriate if the probability that a thread may block while initiating an asynchronous operation is high. Unlike asynchronous support threads, starter threads do not block while waiting for the server to service their requests. A starter thread can service another asynchronous request once it
has transmitted its request message to the server. Hence, a distributed file system that uses starter threads to build continuations will still consume less resources than a distributed file system that uses threads to wait for server replies.

Completing an Asynchronous Operation

As mentioned earlier, the distributed file system on the client host executes or runs the continuation associated with an asynchronous I/O operation when it receives the operation's reply from the server. There are two ways to execute this continuation. The first is to execute the continuation while in the network driver's interrupt handler. The second is to hand the continuation off to another thread for execution. The latter incurs additional latency for thread scheduling and context switching. However, Choices employs the latter since there is a major problem with the former.

The problem is that executing a continuation may take a substantial amount of time especially for a read operation. Typically, the data returned by the server will need to be reassembled\(^7\) and copied into appropriate buffers on the client. Although very unlikely, the thread executing the continuation may also block. Executing the continuation in the network driver’s interrupt handler is not desirable because it may disable the network interrupt for too long causing packets to be dropped.

The Choices distributed file system maintains a small pool of threads, known as the client continuation threads to execute continuations. The number of threads in this pool depends on the bandwidth capabilities of the network interfaces and the number of processors in the host. If a single processor on a host can process continuations faster than all the network interfaces on the host can receive replies, then a single continuation thread is sufficient per the host. On the other hand, if the network interfaces can receive replies faster than a processor can process continuations, then a multiprocessor will benefit from having more threads. The additional threads allow several continuations to execute in parallel. On an uniprocessor host, having additional continuation threads is less beneficial since a continuation thread almost never blocks. Hence, having additional continuation threads on an uniprocessor host is unlikely to increase concurrency. In most current systems, one thread in this pool is sufficient since network bandwidth is the primary bottleneck.

To ensure that replies are processed as soon as possible to reduce latency, the continuation threads on a client have a high priority.

In summary, the continuation model allows the client to support many concurrent asynchronous operations with very few continuation threads. In most cases, only one continuation thread is sufficient per client host.

\(^7\)Reassembly is usually required since most network’s Maximum Transfer Unit (MTU) is smaller than the file system block size.
7.5.1.2 The Server Side

In the thread model, the server also creates a pool of threads to service client requests. Each incoming request is assigned to a thread in this pool. This thread parses the request and issues disk I/O requests to read or write data as required by the requested operation. Since these are file server threads, the probability that a request will require disk I/O’s are quite high. When a disk I/O is required, the server thread blocks after it has issued the disk I/O request to the disk scheduler. It blocks until the disk I/O has been completed. Since the server thread may block for a long time while waiting for the disk I/O to complete, it suffers from the same shortcomings as the thread model on the client.

The Problem

Unlike the client, all server requests are synchronous since a server thread cannot send a reply to the client until the result of an operation is known. In the thread model, the number of concurrent requests that a server can service is limited by the number of server threads. When there is a burst of requests, these server threads may become exhausted very quickly since they are likely to be blocked waiting for the disk. For same reasons presented in Section 7.5.1.1, creating more threads to handle bursts are not undesirable.

The Solution

Since the interaction between a server thread and a disk is similar to the interaction between a client and its server, the continuation model is also appropriate for the file server. In the continuation model, each client request is still assigned to a server thread for parsing. This thread creates an continuation for each disk I/O request. Then, it submits the disk I/O request and its associated continuation to the disk scheduler. When a disk I/O completes, its continuation is executed.

Building Continuations

Building continuations on a server is more complex than on a client since a file server operation may generate several disk I/O requests. Each of these disk I/O requests must have a continuation. Each continuation must contain sufficient information to determine if the client requested operation is complete. It must also contain sufficient information to send a reply to the client when all disk I/O’s generated by the client requested operation have completed. Both pieces of information are necessary since the server thread that constructed the continuations cannot pre-determine the order in which the disk I/O requests will be serviced.\footnote{These threads are known as “nfsd” daemons in NFS.}

\footnote{In some cases, the order of the disk I/O’s may be determined by the server. However, the order of actual disk I/O’s should be determined by the disk scheduler.}
For example, a read operation may require two separate disk reads to obtain data from different parts of a disk. One of these disk requests will complete before the other. When its continuation is executed, no reply should be sent since the other disk read has not completed. Eventually, the other disk read will complete and its continuation will be executed. When this occurs, the data read by both disk requests must be marshaled into a reply message and transmitted to the server.

**Executing Continuations**

Like the client, there are two ways to execute a continuation upon disk I/O completion. The first executes the continuation in the disk interrupt handler. The second hands the continuation to a separate thread for execution. For reasons similar to those mentioned in Section 7.5.1.1, Choices employs the latter. The threads that execute continuations on the server are known as the *server continuation threads*.

**Resource Requirements**

Each server requires two different kinds of threads: server threads that build continuations, and server continuation threads that execute continuations. The appropriate number of server threads and server continuation threads depends on the the bandwidth capabilities of the network interfaces, the number of processors in the host, and the bandwidth of the disk subsystem on the server. In most cases and in an ideal implementation, only one thread of each kind is required.

Unfortunately, the *Choices* distributed file system needs four server threads. This is because some server operations have not been converted to use continuations. These server operations are complex and hard to convert. The additional server threads are required because some of these operations may block. In practice, this is not a big problem because these operations seldom block once the necessary supporting data have been cached in memory. All such operations involve path name resolution.

Although both server threads and server continuation threads are still needed in the continuation model, these threads are better utilized. Unlike the server threads in the thread model, the server threads in the continuation model do not block while waiting for the disk. They are available to handle another client request once the necessary disk I/O requests have been submitted to the disk.

**7.5.1.3 Differences and Advantages**

Figure 7.4 illustrates the differences between the thread model and the continuation model. The advantages of the continuation model are:

1. *Lower memory overhead* -

   Continuations consume less memory than a blocked thread. On a Sun SPARCstation 2 running Choices, each blocked thread consumes at least one page (4 kb) for its stack. Hence, memory
1. Concurrency not limited by number of async client and server threads, provides for longer server disk queues.
2. Less resource usage, continuations are ‘cheaper’ than threads.

**Figure 7.4. Differences between the thread model and the continuation model**

<table>
<thead>
<tr>
<th>Thread model</th>
<th>Continuation model</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 blocked threads</td>
<td>No blocked threads</td>
</tr>
</tbody>
</table>
consumed by an operation that is waiting for a server disk is at least 4 kb on the server, and another 4 kb on the client. Typically, the “smallest” threads consume two to four pages each for their stacks. For an asynchronous operation that transfers 8 kb of data,\textsuperscript{10} the additional memory overhead for the operation is 50-100%. The minimal overhead is 50% (4 kb ÷ 8 kb) because the client thread blocks to wait for the server’s reply. The overhead is 100% (2 × 4 kb ÷ 8 kb) if the server thread also blocks to wait for the disk.\textsuperscript{11} This is not acceptable since the additional memory overhead is incurred when demand for memory is highest, \textit{i.e.} during bursts when the client and server threads are most likely to be blocked.

On the other hand, the \textit{Choices} distributed file system usually consumes substantially less than 512 bytes in continuations for each asynchronous I/O operation. Therefore, the memory overhead is less than 6% (512 ÷ 8 kb).

2. \textit{More concurrency and longer disk queues} -

Due to the lower memory overhead of the continuation model, the continuation model can support a greater number of concurrent remote operations. The maximum number of concurrent remote operations that can be supported is limited by the amount memory available on the server to sink or source data. The continuation model allows remote requests to “wait” for service at the remote server’s disk scheduler. It increases the disk scheduler’s queue lengths. This increases throughput. In the thread model, memory and other resource constraints may limit the number of available threads on the client and the server. When these threads become exhausted during bursts, remote requests will be waiting for threads to become available instead of waiting at the remote disk queue.

3. \textit{Lower processor overhead} -

The continuation model can reduce the number of context switches. In the case of the asynchronous client request, one context switch is eliminated since the client need not hand the request to an asynchronous support thread. Furthermore, if several server replies are queued at the network interface, a single client continuation thread can execute the continuations associated with these replies without context switching. This occurs when the client is busy with higher priority tasks when the replies arrive from the server. In the thread model, a context switch is required for each reply.

The same is true for the server. If several client requests are queued at the network interface on the server waiting for the server thread, the server thread can service all these requests, \textit{i.e.}\textsuperscript{10}

\textsuperscript{10}8 kb is the most common NFS block size.

\textsuperscript{11}Although memory consumed by these thread stacks can be reclaimed by swapping or paging the stacks, it is costly since it generates additional I/O requests.

110
<table>
<thead>
<tr>
<th>transfer size</th>
<th>8 kb</th>
<th>48 kb</th>
<th>minimum</th>
</tr>
</thead>
<tbody>
<tr>
<td>concurrency</td>
<td>NFS</td>
<td>8</td>
<td>unlimited</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>file size</th>
<th>time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Mb</td>
<td>66.6</td>
</tr>
<tr>
<td>16 Mb</td>
<td>139</td>
</tr>
<tr>
<td>32 Mb</td>
<td>287</td>
</tr>
<tr>
<td>64 Mb</td>
<td>626</td>
</tr>
<tr>
<td>128 Mb</td>
<td>1317</td>
</tr>
</tbody>
</table>

Table 7.1: Time required to write a large file

submit their I/O's to the disk scheduler, without context switching. Similarly, multiple queued continuations can be executed by a single server continuation thread without context switching.

Because of these advantages, the continuation model performs better during bursts when good performance is most important. Table 7.1 presents the amount of time required to create and write a file of a given size using various write block sizes at various levels of concurrency. The data presented in this table was obtained from a pair of Sun SPARCstation 2's. Each SPARCstation has 16 Mb of memory and a Micropolis 4110 hard drive. Ethernet connects these workstations. The transport protocol is UDP with checksum enabled.\(^{12}\) The first column presents the size of the output file. The second column presents the amount of time taken by NFS (SUNOS 4.1.3). The NFS server in this experiment has 8 threads. The third column presents the amount of time taken by 8 kb transfers with at most 8 concurrent requests using continuations. For this experiment, the distributed file system was modified to restrict the number of concurrent requests from the client. The Choices distributed file system performs much better than NFS because of its more aggressive write-behind and free-behind data transfer strategies (discussed in Section 6.3.7.4).

The fourth column presents the amount of time taken using 8 kb transfers but with unlimited concurrency. It shows that improved concurrency does indeed increase the write throughput at 8kb. Hence, the continuation model does provide better performance. The fifth column presents the amount of time taken using 48 kb transfers with unlimited concurrency. The final column presents the minimum amount of time required to transfer the data to the server using the UDP protocol. At the time of this experiment, the maximum UDP throughput for Choices is approximately 6 megabits per second at 8 kb, and 6.68 megabits per second at 48 kb. The continuation model achieves near network optimal throughput at 48 kb. In other words, the client’s asynchronous write-behinds are saturating the network. Hence, the network is the bottleneck and not the disk.

\(^{12}\)This TCP/IP implementation is derived from the BSD 4.4.
At the 8kb transfer size, the network is not the bottleneck. The bottleneck is the disk because the server writes two to three distinct locations on a disk for each write request. The first disk write transfers the client’s new data to disk. Then, another one to two writes update the file’s meta-data, i.e. the file’s inode and indirect blocks[1, 58, 72]. At the 8 kb transfer size, the file’s meta data is updated once for every 8 kb of data transferred. At the 48 kb transfer size, there are one-sixth as many meta-data updates.

Table 7.1 shows that the Choices distributed file system does benefit from increased concurrency, especially in the presence of data transfer strategies that generate large bursts and uses large transfer sizes. These strategies include clustering, read-ahead, and write-behind. The continuation model permits more concurrency at a lower cost.

7.5.1.4 Implementation Details

To support asynchronous I/O operations with continuations in Choices, I/O methods have been revised. A parameter has been added to each I/O method. The type of this parameter is a pointer to a continuation. This continuation is known as the caller’s continuation. A continuation is an object that consists of a pointer to a function, known as the continuation function, a result instance variable, and data specific to the continuation. The result instance variable contains the result of the asynchronous operation. When a caller invokes an I/O method, the caller’s continuation pointer may be null or it may reference a continuation object. If the pointer is null, then the caller invoked this method to perform a synchronous I/O operation, i.e. the caller waits for all the necessary device I/O’s to complete. If the pointer references a valid continuation, then the caller invoked this method to perform an asynchronous I/O operation, i.e. the caller does not wait for all the necessary device I/O’s to complete. When all the necessary device I/O’s complete, a continuation thread executes the caller’s continuation by invoking the continuation function with the caller’s continuation as its only argument.

When an I/O method, known as the parent method, is invoked to perform an asynchronous I/O operation, it usually invokes other I/O methods, known as the child methods. For example, a contained persistent memory object’s rawWrite method invokes the containing persistent memory object’s writeThrough method. In this example, rawWrite is the parent method and writeThrough is the child method. If no additional processing is required when the child method’s asynchronous I/O operation completes, the parent method passes its caller’s continuation to the child method. When the child method’s I/O operation completes, the continuation thread executes the child method’s caller’s continuation, which is also the parent method’s caller’s continuation.

If additional processing is necessary when the child method’s asynchronous I/O operation completes, the parent method creates a new child continuation that encapsulates the additional processing, and passes the new child continuation to the child method. This child continuation becomes the child
UNIXNode::rawWrite(0, 7, 0x10000000, U)
    (unit size = 1024, disk partition unit size = 512)
create common state P_C
create P_0 (P_C) for units 0 to 3, disk partition units 100 to 107
create P_1 (P_C) for units 4 to 7, disk partition units 200 to 207

DiskPartition::writeThrough( 100, 107, 0x10000000, P_0 );
    DiskPartition::rawWrite( 100, 107, 0x10000000, P_0 );
        (disk partition unit size = disk unit size)
create D_0
Disk::writeThrough( 10100, 10107, 0x10000000, D_0 );
    Disk::rawWrite( 10100, 10107, 0x10000000, D_0 );
schedule disk I/O
    return
return
return
DiskPartition::writeThrough( 200, 207, 0x10001000, P_1 );
    DiskPartition::rawWrite( 200, 207, 0x10001000, P_1 );
create D_1
Disk::writeThrough( 10200, 10207, 0x10001000, D_1 );
    Disk::rawWrite( 10200, 10207, 0x10001000, D_1 );
schedule disk I/O
    return
return
return

Figure 7.5: An example: creating and passing continuations.

methods’ caller’s continuation. The parent method’s caller’s continuation is also known as the parent continuation. When the child method’s asynchronous I/O operation completes, the continuation thread executes the child continuation. The child continuation performs the additional processing and executes its parent continuation. Creating and passing continuations between calling and called objects is necessary because of encapsulation - the parent method can make no assumptions about its child method, and the child method can make no assumptions about its parent method.

Figure 7.5 illustrates how continuations are created and passed from one I/O method to another. To simplify the illustration, only relevant parameters are shown. Figure 7.6 presents the parent-child relationships among the continuations in Figure 7.5. Figure 7.7 illustrates how these continuations eventually get executed. In the current implementation, storage for P_C, P_0, and P_1 are contiguous and is allocated by a single memory allocation request from the heap. This reduces overhead associated with heap memory allocation. The memory overhead for all the continuations in this example, excluding U, is less than 192 bytes.
The Choices framework defines some standard continuation classes. These classes represent the most popular continuations, \emph{i.e.} continuations that most parent I/O methods create for their child I/O methods. When the standard continuation classes are inadequate, the desired new functionality and requirements are added, in most cases, by subclassing existing continuation classes. The Choices framework provides the following kinds of continuations:

1. \textit{Dummy} -
   It does nothing. A parent method invokes a child method with this continuation if it does not care about the result of the child method’s asynchronous I/O operation.

2. \textit{Release lock} -
   Unlock a range of units belonging to a memory object.

3. \textit{Translate unit size} -
   Translate the result of an I/O operation from the child method’s unit size to the parent method’s unit size.

4. \textit{Translate bytes to units} -
   Translate the result of an I/O operation from bytes to the parent method’s unit size.

5. \textit{Sum result} -
   Sum the result of an I/O operation with a given constant. It is used when a part of an I/O request can be completed without any device I/O. For example, reading units that map to holes (or unallocated blocks) from an UNIX inode does not generate any device I/O’s. The UNIX file system writes zeros into the buffer for these units.

6. \textit{Synchronize units} -
   Synchronize units that have just been written. Typically, this continuation is executed after an asynchronous \texttt{MemoryObjectCache::cacheWrite} operation has completed. The cacheWrite method copies changes from a buffer into the memory object cache. It is invoked asynchronously because it may page in units from an I/O device. This continuation synchronizes these changes.

7. \textit{Fragmented I/O complete} -
   A parent I/O method creates one of these continuations, known as a fragmented I/O continuation,
... disk interrupt - disk write ( 10200, 10207 ) done
Disk::release( 10200, 10207 ) to unlock disk units
(Disk::rawWrite( 10200, 10207 ) completes)
queue $D_1$ on for service by continuation thread
exit interrupt handler
...

continuation thread executes $D_1$

$D_1$ invokes DiskPartition::release( 200, 207 )
(DiskPartition::rawWrite( 200, 207 ) completes)
$D_1$ executes $P_1$

$P_1$ updates $P_C$ - one of two writes to partition has completed
$P_1$ done
$D_1$ done
dequeue next continuation or block if queue empty
...

disk interrupt - disk write ( 10100, 10107 ) done
Disk::release( 10100, 10107 ) to unlock disk units
(Disk::rawWrite( 10100, 10107 ) completes)
queue $D_0$ on for service by continuation thread
exit interrupt handler
...

continuation thread executes $D_0$

$D_0$ invokes DiskPartition::release( 100, 107 )
(DiskPartition::rawWrite( 100, 107 ) completes)
$D_0$ executes $P_0$

$P_0$ updates $P_C$ - both writes to partition have completed
$P_0$ invokes UNIXNode::release( 0, 7 )
(UNIXNode::rawWrite( 0, 7 ) completes)
$P_0$ executes $U$
...

$U$ done
$P_0$ done
$D_0$ done
dequeue next continuation or block if queue empty
...

**Figure 7.7**: Executing continuations.
for each of its child I/O method invocations if it invokes the child I/O method more than once. Each child method invocation performs a portion of the parent method’s I/O request. Fragmented I/O continuations created by the same parent I/O method share common state. The fragmented I/O continuation is executed when a fragmented I/O completes. It updates the common state. If all the fragmented I/O’s have completed, it executes the parent continuation. In the previous example, \( P_0 \) and \( P_1 \) are fragmented I/O continuations of \texttt{UNIXNode::rawWrite}, and \( P_C \) is their common state.

The above kinds of continuations address most of the needs of the \textit{Choices} file system and virtual memory management system. The exception is the \texttt{MemoryObjectCache} class. The \texttt{MemoryObjectCache} class needs additional kinds of continuations. These continuations perform functions that are specific to virtual memory management, such as releasing I/O windows, removing address translations, and changing access permissions.

In summary, adding continuations and asynchronous I/O operations to the \textit{Choices} framework is very straightforward. Less than 200 man hours were expended to change the source and debug the changes. This shows that supporting asynchronous I/O operations with continuations is also easy to implement.

### 7.5.2 File Access Hints

The \textit{Choices} distributed file system uses file access hints\(^\text{13}\) to help select appropriate caching strategies for a file. These hints may be supplied by the application, the \texttt{persistent memory object}, and/or the \texttt{persistent object}.

The application may know about the expected access characteristics of the file, such as whether the file will be opened again soon and whether the file will be accessed sequentially. For example, the first stage of a compiler who writes an intermediate file for the second stage knows that the second stage will open the file again soon to read it. The designer of a \texttt{persistent object} can also provide hints via the \texttt{persistent object}. For example, the access characteristics of a BSD directory may depend on the design and implementation of the BSD\texttt{Directory} class. Hence, a BSD directory may provide hints to the file system to help select appropriate caching strategies for the directory. Similarly, the \texttt{persistent memory object} can provide useful hints.

With distributed file access, client hints are propagated to the file server. This allows both the server and the client to take advantage of these hints. In some cases, the client may modify these hints to reflect the effects of caching on the client. For example, if repeated opens of a file are likely, the client may cache the whole file on a local disk and the server may not notice the repeated opens.

\(^{13}\)These hints will be presented in Section 8.2.
7.5.3 Prefix caching

The distributed file system also implements prefix caching[124]. Prefix caching reduces remote directory lookups. In Choices, remote memory objects and remote dictionaries on a host collaborate to maintain a prefix table[124]. A prefix table caches associations of path names to remote memory objects.

The RemoteDictionary class implements the prefix cache. It assumes that cached prefixes are valid for thirty seconds.

7.6 Summary

Choices supports two different remote file access extensions: remote access to persistent object and remote access to persistent storage. Remote access to persistent storage permits client caching. An application can select an appropriate extension for each of its files according to its requirements. These remote file access extensions together with the flexible file system caching framework provide Choices with a flexible distributed file system with numerous caching options. For example, a client may cache the server disk, the disk partition, and/or the BSD inode, and it can use a different caching strategy for each of these persistent memory objects. Furthermore, several optimizations have been incorporated to improve performance, and to support flexible file system caching. These optimizations include continuations and file access hints.
Chapter 8

Adaptive Caching

The flexible file system caching framework, presented in the previous chapters, provides flexibility in implementing and choosing caching strategies for individual files. Unfortunately, developers or users of applications have to select appropriate caching strategies to realize the benefits of the available flexibility. This is either hard or extremely inconvenient (refer to Section 2.2). Adaptive file system caching provides an easier and more convenient way to realize these benefits. It uses prior and current access behaviors of each file to select an appropriate caching strategy for the file. Therefore, it allows unmodified applications to benefit from the additional flexibility provided by a flexible file system caching service.

This chapter discusses adaptive file system caching in detail. First, it discusses problems associated with modifying existing applications to take advantage of the Choices flexible file system caching service. Then, it presents a solution to these problems. This solution, known as adaptive file system caching, consists of three steps:

1. The file system caching service defines a set of attributes for describing expected file access behaviors. Application developers may modify or annotate their applications to provide expected file access behaviors using these attributes.

2. The file system caching service selects an appropriate caching strategy for each file based on the file’s expected access behavior.

3. The file system caching service detects at runtime the access behavior of each file. It describes the observed behavior using the above attributes, and it also remembers the observed behavior. Since files are likely to be accessed the same way, the observed access behavior of a file is also its expected access behavior.

In summary, the adaptive file system caching service observes file accesses, uses the observed file access behavior to choose an appropriate caching strategy for the file.
Finally, this chapter presents the *Choices* implementation of an adaptive file system caching service. It discusses other possible implementations, performance considerations, and additional requirements introduced by distribution.

## 8.1 Problems

The initial thesis proposal is to demonstrate the advantages of a flexible file system caching service. A prototype flexible file system caching service has been implemented. Upon completion of a prototype, I would run applications and replay trace data\(^1\) that have been annotated to select appropriate caching strategies to demonstrate the effectiveness of flexible file system caching. Soon after I started to annotate a few applications and annotate some collected trace data, I realized that it would be hard to modify a lot of existing or legacy applications to take advantage of different caching strategies. There are several problems.

The first and most immediate problem is that the application itself has to be modified. This may not be possible if the source code for the application is not available. The source may not be available because it has been lost. Alternatively, the application may be purchased or licensed from a software vendor that does not provide source to end users.

Although it may be possible to modify an application without its source code by patching its binaries,\(^2\) it is seldom done in practice because it is very tedious to modify binaries. Another way to affect the selection of caching strategies is through environment or context information associated to a process at runtime. For example, environment variables may be used in UNIX to select caching strategies. However, this is not desirable because it depends on the end user to provide the correct environment and context. Furthermore, the end user may not have sufficient knowledge of the computing environment or the application to provide the appropriate environment and context. In general, modifying an application is best left to the developer of the application. At the very least, the developer would have reasonable knowledge of the application.

Even if the sources for most applications were available, it would be tedious to modify a lot of these applications.

The second problem is that computing environments are diverse and may evolve over time. The developer of an application is unlikely to be able to select caching strategies that are appropriate for all computing environments. In particular, the developer cannot foresee future changes in the computing environment.

---

\(^1\)Since *Choices* has few useful native applications and it does not provide sufficient emulation of any popular operating system, I had to record the file access activities of useful applications in the applications' native operating system and replay the recorded activities in *Choices* to generate the same file access activities.

\(^2\)One way to do this is to substitute dynamically loaded libraries.
In order for a flexible file system caching service to be generally useful, these problems must be addressed. A flexible file system caching service should benefit both modified and unmodified applications. There following two sections presents two modifications to the flexible file system caching service that will address these two problems.

8.2 Annotate with File Access Characteristics

As mentioned in the previous section, application developers may modify their applications to select appropriate caching strategies. In other words, they annotate their applications with caching strategies. This first modification adds another kind of annotation. The new kind of annotation is the application’s file access characteristics. It is also the preferred kind of annotation. This modification addresses the second problem.

To help the file system caching service select caching strategies, the developer annotates an application with the application’s file access characteristics instead of annotating the application with desirable caching strategies. If the developer specifies a particular caching strategy for each file, then the developer is embedding knowledge of both the computing environment and the application’s file access characteristics into the application. Since computing environments are diverse and may evolve over time, the application developer should not be required to make any assumptions regarding the application’s computing environment. Therefore, the developer should not embed any knowledge of the computing environment into the application.

On the other hand, the developer should have intimate knowledge of the application’s file access characteristics. Therefore, the developer should embed knowledge of the application’s file access characteristics into the application whenever possible. The developer does so by annotating the application with the expected file access behavior of each individual file. This modification allows the flexible file system caching service to select different caching strategies for the same file access behavior in different computing environments. For example, a given file access behavior may employ two distinct caching strategies depending on the bandwidth or latency of the network connecting the server and the client. In essence, this modification permits delayed binding of caching strategies to individual files until the computing environment is known.

The application developer specifies an expected file access behavior for each file using a fixed and limited set of attributes. These attributes describe and distinguish the most common file access behaviors. The file system caching service understands these attributes and selects a caching strategy for each file based on these attributes and the computing environment. As the file system caching service evolves, new caching strategies may be added. When new caching strategies are added, the file system
caching service should be modified to select the new caching strategies whenever the new strategies are appropriate for particular file access behaviors.

Although it is possible to support an unlimited set of attributes, the current implementation does not do so. This is because any algorithm or heuristic that selects suitable caching strategies can only take into account a fixed set of variables, i.e., attributes.

On the other hand, additional attributes may be added as the file system caching service evolves. When additional attributes are added, the algorithms and heuristics that selects caching strategies should also evolve to take advantage of the additional information provided by the additional attributes. In general, attributes should be added whenever existing attributes cannot distinguish important, but distinct, file access behaviors that should employ different caching strategies. The goal is to have a set of attributes that is large enough to cover the most commonly observed and most important file access behaviors.

Unfortunately, a fixed set of attributes may not be able to describe and distinguish some uncommon file access behaviors. Furthermore, the flexible file system caching service may have special and specific caching strategies that are dedicated to these uncommon file access behaviors. In these special and hopefully rare cases, an application should still be able to specify specific caching strategies for the files that exhibit these uncommon file access behaviors. For example, a file access behavior that is hard to describe using a fixed set of attributes might be reading every other consecutive unit. If a dedicated caching strategy is available to prefetch data for this file access behavior, an application should be able to specify this caching strategy for the files that exhibit this behavior.

In summary, the improved flexible file system service understands two different kinds of annotations for each file. The first kind of annotation is a specific caching strategy. The second kind of annotation is an expected file access behavior. If an application specifies a particular caching strategy for one of its files, then the flexible file system service honors this specification. In other words, the flexible file system caching service attaches the specified caching strategy to the file.

If the application does not specify a specific caching strategy, it may specify the expected file access behavior of the file. The improved flexible file system caching service uses the expected file access behavior to select a caching strategy for the file. This modification addresses the second problem, i.e., the developer of an application is unlikely to be able to select caching strategies that are appropriate for all computing environments. With this modification, the application developer provides the file access characteristics of an application and the flexible file system caching service provides knowledge of the current computing environment. In general, the file access characteristics of an application is unlikely to change with the computing environment. For example, the UNIX program cat always accesses files sequentially. On the other hand, the flexible file system caching service is expected to evolve with the
computing environment so that it can select appropriate caching strategies for the evolving computing environment.

The improved flexible file system caching service treats file access behavior annotations as hints since actual file accesses may deviate from the specified file access behaviors. If an annotation is incorrect, then an inappropriate caching strategy may be selected and performance may suffer. The file system caching service should not fail or abort an application if the application’s annotations are incorrect.

If the application does not specify either a caching strategy or an expected file access behavior, then the improved flexible file system caching service selects a default caching strategy. In some cases, the type of the file and the file’s file system type may influence the selection of the default caching strategy. For example, the designer of a BSD file system knows how BSD directories and symbolic links are likely to be accessed, and may encode their expected file access behaviors into the implementation.

8.3 Adaptive File System Caching

The second modification introduces adaptive file system caching. It addresses the first problem, i.e. applications have to be modified to take advantage of the available flexibility.

Since it is hard to annotate a lot of existing applications, some other mechanism is needed to select appropriate caching strategies for unmodified applications. Adaptive file system caching is one such mechanism. It detects the file access behavior of each file. It uses the detected file access behavior of a file to select an appropriate caching strategy for the file. A flexible file system caching service that implements adaptive file system caching is also known as an adaptive file system caching service.

This modification builds on the first modification to implement adaptive file system caching. The first modification introduced a set of attributes that describes file access behaviors. The second modification observes the file access activities of each file and describes the observed file access behavior using the set of attributes introduced by the first modification. In other words, the adaptive file system caching service attempts to annotate each file with the file’s observed file access behavior. This allows the caching strategy selection algorithms and heuristics discussed in the previous section to select appropriate caching strategies for these files. In summary, adaptive file system caching allows an unmodified application to take advantage of different caching strategies.

The adaptive file system caching service assumes that the future file access behavior of a file is likely to be the same as its past file access behavior. Blaze and Alonso’s observations support this assumption[9]. In fact, they used “inertia” to describe this property. Our own study of file access characteristics of supercomputing applications also support this assumption[62].

Unfortunately, adaptive file system caching does have a shortcoming. The primary shortcoming is that the expected file access behavior of a newly created file is unknown since the file has no prior file
access history. In the current implementation, the file is not annotated and a default caching strategy is selected (as discussed in Section 8.2). A good adaptive file system caching service should be able to determine quickly the file access behavior of a new file, provide the necessary annotations, and replace the default caching strategy with a more appropriate caching strategy if necessary.

In summary, adaptive file system caching allows unmodified applications to take advantage of the additional flexibility provided by a flexible file system caching service.

8.4 An Implementation

The following subsections present an implementation of adaptive file system caching in Choices. The first subsection presents the set of attributes defined in this implementation and how they influence caching strategy selection. The second subsection discusses how applications specify caching strategies and provide file access behavior hints. The third subsection presents how Choices observes file access activities and detect various file access behaviors. The fourth subsection discusses how different computing environments affect caching strategies. The fifth subsection describes how Choices remember dynamically observed or application-provided file access behaviors. The sixth subsection discusses additional requirements introduced by distribution. Finally, the last subsection discusses extensions for different computing environments.

8.4.1 Attributes and Caching Strategies

Choices uses the following boolean attributes to describe file access behaviors:

1. Temporary versus Permanent -
   The value of this attribute is true if the file is likely to be deleted in the near future. Otherwise, the value of this attribute is false.

2. Single-access versus Multi-access -
   The value of this attribute is true if the data in this file is likely to be accessed only once. In other words, data in this file is unlikely to be needed again soon after it has been accessed. The value of this attribute is false if the data in this file is likely to accessed more than once within a short time period.

3. Large versus Small -
   The value of this attribute is false if the file accessed is small, will be small, or small amounts of data are accessed at a time. The value of this attribute is true if the file accessed is large, will be large, or large amounts of data are accessed at a time.
<table>
<thead>
<tr>
<th>Properties</th>
<th>write-behind</th>
<th>free-behind</th>
<th>read-ahead</th>
<th>prefetch file</th>
<th>lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>temporary</td>
<td>*size</td>
<td>*size</td>
<td></td>
<td></td>
<td>2×</td>
</tr>
<tr>
<td>permanent</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>single-access</td>
<td>yes</td>
<td>small</td>
<td></td>
<td></td>
<td>1½×</td>
</tr>
<tr>
<td>multi-access</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1×</td>
</tr>
<tr>
<td>large</td>
<td>yes</td>
<td>large</td>
<td></td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>small</td>
<td>no</td>
<td>no</td>
<td></td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>random sequential</td>
<td>no</td>
<td>yes</td>
<td></td>
<td></td>
<td>*size</td>
</tr>
</tbody>
</table>

**Table 8.1:** File access behaviors and their caching strategy preferences.

4. **Random versus Sequential** -

The value of this attribute is false if the data in this file is usually accessed in sequential order. Otherwise, the value of this attribute is true.

5. **Whether a secondary cache is beneficial** -

This attribute does not really describe a file access behavior. The adaptive file system caching service uses this attribute to describe whether a secondary cache might be beneficial for this file. Section 8.4.3 discusses this attribute in greater detail.

The value of an attribute identifies a file access behavior property. In the following discussion, file access behavior properties are presented in *italics*.

The various file access behavior properties and their caching strategy preferences appear in Table 8.1. These preferences include data transfer strategies³ and file caching parameters. The AdaptiveLRUReplacementPolicy class is a subclass of the SmarterLRUReplacementPolicy class. It inherits the write-behind, free-behind, and read-ahead data transfer strategies of the SmarterLRUReplacementPolicy class (described in Section 6.3.7.4). In addition, it implements file prefetching and adds new functionality to select data transfer strategies and change file caching parameters according to Table 8.1.

The rows in this table represent different properties. The columns in this table represent the various data transfer strategies and file caching parameters. An empty entry indicates that a particular property is neutral to a particular data transfer strategy or file caching parameter. The write-behind column indicates if write-behind is appropriate. The free-behind column indicates if free-behind is appropriate. A “large” entry in this column indicates that free-behind is appropriate, and freeBehindLag (refer to

³Section 6.3.7.4 describes these data transfer strategies.
Section 6.3.7.4) should be large. Similarly, a “small” entry in this column indicates that free-behind is appropriate but freeBehindLag should be small. A “no” entry indicates that free-behind should be avoided. The read-ahead column indicates if read-ahead is desirable. The prefetch file column indicates whether the entire contents of a file should be read into memory when the file is opened for the first time. The lifetime column indicates the desirable minimum zombie lifetimes, *i.e.*, keepFreshTime (refer to Section 6.3.10). The MemoryObjectCachingContainer class defines a default keepFreshTime. The desirable keepFreshTime for a file is computed by multiplying the default keepFreshTime by a factor in this column. A “*size” entry in this table indicates that the value of the entry depends on the large versus small attribute of the file. For example, write-behind is desirable for temporary large files but inappropriate for temporary small files. Similarly, free-behind should be “large” for temporary large files and disabled for temporary small files.

In general, the properties in Table 8.1 are enumerated in decreasing importance. Some combinations of file access behavior properties may result in conflicting preferences. For example, the table indicates that write-behind is both appropriate and inappropriate for a small permanent single-access file. The file’s single-access property desires write-behind, but its small property contradicts this preference since the small property prefers to disable write-behind. These conflicts are resolved according to the relative importance of the conflicting properties. In other words, the preferences of a more important property override the preferences of a less important property. In this example, write-behind is desirable since single-access is more important than small.

There is an exception to the enumerated order of importance in Table 8.1. This exception is represented by “*size” entries in the table. The exception is that the large and small properties are more important than the single-access property for temporary files. In other words, the preferences of the large and small properties override the preferences of the single-access property for temporary files.

The remainder of this subsection discusses the rationale behind the preferences presented in Table 8.1.

**Single-access versus Multi-access**

A single-access file should have a short zombie lifetime. In addition, write-behind and free-behind should be enabled for a single-access file.

A single-access file should have a shorter than normal zombie lifetime. In other words, when a single-access file becomes a zombie, it should be reclaimed sooner than other zombies of similar age. The keepFreshTime of a zombie determines when a zombie becomes eligible for reclamation. A zombie with a shorter keepFreshTime becomes eligible for reclamation sooner than another zombie of similar age, but with a longer keepFreshTime. Hence, the keepFreshTime of a single-access file should be less than that of a multi-access file. In the current implementation, Choices sets the keepFreshTime of a single-access file to one half of the system default keepFreshTime.
In an ideal situation where a *single-access* file is never accessed in the near future, then *single-access* files should not become zombies. They should be reclaimed once they become inactive. Unfortunately, annotations provided by applications are only hints (refer in Section 8.2). The *single-access* annotation suggests that a file is unlikely to be accessed in the near future. Since application hints may be wrong, the current implementation is more conservative. It chooses to shorten the zombie lifetime of a *single-access* file, instead of reclaiming a *single-access* file once the file becomes inactive.

Since new data written to a *single-access* file is unlikely to be overwritten in the near future, the new data should be synchronized soon after it has been written into the file’s cache. Synchronizing dirty data early, *i.e.* write-behind, is desirable because reclaiming memory from clean units is easier and faster than reclaiming memory from dirty units. Clean units need not be synchronized before being reclaimed. Hence, write-behind should be enabled for *single-access* files.

Similarly, recently accessed data is unlikely to be accessed again in the near future. Hence, memory occupied by recently accessed data should be reclaimed soon after the data has been accessed. Therefore, free-behind is also desirable for *single-access* files. In addition, the $freeBehindLag$ of a *single-access* file should be small so that memory is reclaimed eagerly (refer to Section 6.3.7.4). As mentioned earlier, each file’s $freeBehindLag$ changes according to the amount of memory contention. The range of adjustment is bounded by $maxFreeBehindLag$ and $minFreeBehindLag$. To ensure a small $freeBehindLag$, the file system caching service sets the $maxFreeBehindLag$ of a small file to a small value. In the current implementation, this small value is a single cluster.

A *multi-access* file has an average zombie lifetime. Unlike the *single-access* property, the *multi-access* property is neutral to write-behind and free-behind. Whether write-behind and free-behind are appropriate or inappropriate for a *multi-access* file depends on the other file access behavior properties of the file. For example, write-behind and free-behind are desirable for a *large* *multi-access* file but inappropriate for a *small* *multi-access* file.

**Large versus Small**

The *small* property identifies a small file that is likely to fit in main memory without stressing memory resources. Similarly, if only a small portion of a huge file is accessed, then this file is also considered to be *small*. I/O transfers to and from a *small* file are small since there isn’t much data to access. Unfortunately, the actual determination of whether a file is *small* depends on the amount of main memory on the host and the number of active files that the host must support. For example, a *small* file to a 1 gigabyte main memory host is might be a *large* file to a 16 kilobyte main memory host. Similarly, a *small* file to a host with few active files might be a *large* file to the same host with thousands of active files. Therefore, an application needs help from the file system to determine if a file is *small*. The file system should maintain data regarding typical file system usage. This data includes the load level, *i.e.*
the usual number of active files. The file system should also know the amount of main memory available to file caching. The file system determines the maximum size of a small file from the load level and the amount of main memory. For example, the Choices file system uses the following formula to determine the maximum size of a small file.

\[ S = C \times \frac{M}{A} \]

where

- \( S \) = maximum size of a small file
- \( C \) = a constant, to adjust for a large number of tiny or huge files
- \( M \) = amount of memory available for file caching
- \( A \) = average number of active files

In summary, an application obtains the maximum size of a small file from the file system. It annotates files, whose expected file size or expected amount of data accessed is less than the obtained maximum size, as having the small property.

Write-behind is not desirable for small files since it is more efficient to write all the data in a small file back to the file’s storage device when the whole file is eventually synchronized. Typically, a whole file is synchronized while it is a zombie when there are no active I/O’s to and from the file (refer to Section 6.3.10). Delaying writes until file synchronization is more efficient because it permits the file system to optimize placement of file blocks on the file’s storage device and to increase write sizes.

Since a small file is not expected to stress the memory resource, write-behind is not necessary. In addition, write-behind should be avoided because it may induce writes to synchronize dirty data before reclaiming memory occupied by the dirty data.

Similarly, since a small file is not expected to stress the memory resource, it is desirable to prefetch an entire file upon initial read access for files whose sizes are less than the maximum size for small files. This is desirable even though the whole file might not be accessed. This is because the Choices computing environment is expected to be similar to the computing environments of the BSD study[80] and the Sprite study[4]. Observations from these studies indicate that whole file access is likely (refer to Section 4.2). In addition, since these files are small, an application should be able to consume their data quickly. In other words, long time delays between accesses are not expected. Hence, the prefetched data is unlikely to be flushed from memory before it is needed.

For files whose sizes are more than the maximum size, then a small hint suggests that up to the maximum size amount of data will be accessed. This data could be similarly prefetched upon initial read access. The initial access provides the starting file offset for the prefetch operation.

---

4 In the current implementation, the Choices file system does not maintain load level statistics. A static expected load level is encoded in the file system.
Unlike small files, large files are expected to stress the available memory resource. Hence, it is desirable to voluntarily limit the resource usage of large files. This implies that free-behind is desirable. Although free-behind is most applicable to sequentially accessed files, it is also applicable to large files because applications, that do not access large files sequentially, tend to access large chunks of consecutive data at a time from these files. Hence, free-behind can operate on the sequential access parts of these randomly accessed file.

To avoid reclaiming memory prematurely, i.e. before available resources are utilized, the file system caching service should allow a larger amount of data from a large file to be cached before initiating a free-behind operation. Hence, a large file should have a large freeBehindLag. As mentioned before, Choices adjusts the actual freeBehindLag of a file according to the amount of memory contention. To ensure a reasonable large freeBehindLag for a large file, it sets the minFreeBehindLag of the file to a larger value. In the current implementation, this value is two clusters.

If there are writes to a large file, the file system caching service expects a large amount of data to be written. Hence, write-behind is desirable because it spreads out writes to the file’s storage device over time to avoid bursts (refer to Section 6.3.7.4). Write-behind is also appropriate because free-behind is desirable and it permits faster reclamation of memory by free-behind operations. It increases the probability that a free-behind operation can reclaim memory without having to synchronize dirty data first. Similarly, it also allows the memory pagers to reclaim memory faster when they are activated. When there are many large files, these pagers are also more likely to be activated.

Unlike small files, large files should not to be prefetched upon initial read access. This is because these files may be too big to be prefetched into main memory. Since prefetching a large file requires more data to be transferred, the probability of reading unnecessary data is higher. In this case, unnecessary data includes data that will not be needed until much later in the future, and is likely to be flushed from memory before being accessed. Furthermore, the cost of such a prefetch operation is also higher since more data is read. In other words, the risk of file prefetching is higher for large files. Instead of file prefetching, a large file should depend on read-ahead to prefetch data incrementally from the file, if necessary. Like write-behind, read-ahead is more effective because it spreads out read operations over time. It is also more likely to read the required data into memory right before the data is needed.

In summary, a small file does not prefer write-behind and free-behind but prefers file prefetching. On the other hand, a large file prefers write-behind and free-behind with a large freeBehindLag, but does not prefer file prefetching.

Random versus Sequential

A sequential file is one that is sequentially accessed most of the time. This includes some randomly accessed files that are accessed using long runs of sequential accesses with few rare random accesses
between runs. For example, a file that is accessed entirely repeatedly should be classified as *sequential* even though there are some random-access seeks to the beginning of the file. Read-ahead is desirable for *sequential* files but not desirable for *random* files.

When read-ahead is desirable, the desirable amount of data to prefetch, *i.e.* the *readAheadLag*, depends on the *small versus large* attribute. In other words, a *large* file has a larger *readAheadLag* than than a *small file*. The default *readAheadLag* for a *large* file is four clusters. The default *readAheadLag* for a *small* file is one cluster.

**Temporary versus Permanent**

A *temporary* file is usually a recently created file that will be deleted in the near future. It usually contains transient data that is deleted after the computation is complete. For example, most files created in the `/tmp` directory on UNIX systems are temporary, and compilers create *temporary* files to pass information between different compiler stages. Similarly, merge sort algorithms create *temporary* files that contain intermediate merge results. A *temporary* file has a long zombie lifetime and the *large versus small* attribute of the file determines its write-behind and free-behind preference.

A *temporary* file should have a long zombie lifetime to avoid writes to the file’s storage device. Any such writes would be wasted since the data written would be deleted soon. Therefore, it is desirable to cache dirty data for as long as possible. This increases the probability, and amount of data loss that would occur, when there is a system failure. The increased risk is acceptable since *temporary* files typically do not contain crucial data. For example, files in `/tmp` must not contain crucial data since all files in `/tmp` are deleted during system startup. In most cases, the data in *temporary* files can be recreated from some other source files. For example, compiler and merge sort intermediate files can be recreated from their original input files. As mentioned in Section 6.3.10, the *keepFreshTime* of a file also determines the *syncDelayTime*. In summary, a large *keepFreshTime* allows dirty data to remain unsynchronized for a longer period of time.

In addition, a large zombie lifetime is also desirable because it delays zombie reclamation and it avoids reconstructing the in-memory state associated with each *temporary* file multiple times during the short lifespan of the *temporary* file. This state includes the file’s *persistent object*, *persistent memory object*, *memory object cache*, *migration policy*, and *secondary cache*. Reconstructing this state should be avoided because it is expensive. Typically, several I/O’s are required to obtain the necessary meta-data to construct these objects.

Table 8.1 shows that the zombie lifetime preference of the *temporary* property is more important than the zombie lifetime preferences of the *single-access versus multi-access* attribute. The *temporary* property also overrides the write-behind and free-behind preferences of the *single-access* property. The driving principle behind this assignment of importance to the above preferences is, the same as before,
to avoid writing to temporary files when possible. This means delaying synchronization of dirty data for as long as possible. Therefore, a large zombie lifetime is desirable regardless of whether a temporary file is also single-access or multi-access.

The temporary property overrides the write-behind and free-behind preferences of the single-access property because the preferences of the single-access property are inconsistent with the above principle. Write-behind and free-behind remains undesirable regardless of whether a temporary file is single-access or multi-access.

On the other hand, it is appropriate for the temporary property to follow the write-behind and free-behind preferences of the large versus small attribute even though the large property enables write-behind and free-behind. This is because writes may not be avoidable for large temporary files due to their memory resource consumption. Furthermore, it allows a large file to cache more data before activating free-behind than a single-access file. A small file does not prefer write-behind and free-behind. These preferences are consistent with the preferences of a temporary file to delay writes.

Unlike the temporary property, the permanent property does not impose any preferences. In other words, the other attributes determine the preferences of a permanent file.

Summary

This subsection presented the attributes that describe file access behaviors in Choices. It also shows how these attributes influence caching strategy preferences.

8.4.2 Specifying Caching Strategies and File Access Behaviors

As mentioned in Section 8.2, an application can affect caching strategy selection for each of its files in two different ways. First, it can directly specify the desirable caching strategy for a file. Alternatively, it can describe the expected file access behavior of a file using the abovementioned attributes, and the file system caching service chooses an appropriate caching strategy based on the expected file access behavior. This subsection presents how a Choices application specify caching strategies and expected file access behaviors.

First, an application may directly specify a caching strategy for a file by connecting together appropriate objects belonging to various classes in the flexible file system caching framework (described in Section 6.3). In particular, the most relevant classes are the MigrationPolicy class and the SecondaryCache class. When connected, these objects implement the desired caching strategy. The Choices flexible file system caching framework provides the necessary methods for connecting these objects. In particular, the PersistentMemoryObject class defines and implements the following instance methods:
1. **setMigrationPolicy** -
   This method connects the *migration policy* specified by its argument to the *memory object cache* of the receiver *persistent memory object*.

2. **addSecondaryCache** -
   This method adds the *secondary cache* specified by its argument to the *memory object cache* of the receiver *persistent memory object*. Each subclass of the *PersistentMemoryObject* class may overload this method to disallow or permit only certain kinds of *secondary caches* to be associated with one of its instances. For example, an UNIX *inode* will not permit an UNIX *inode cache* to be attached to it’s *memory object cache*.

In addition, an application can get access to some of these objects to read and alter parameters that modify the behavior of these objects.

For the reasons presented in Section 8.1, specifying a caching strategy directly is not the preferred way to select a caching strategy. To specify a caching strategy directly, an annotated application must have sufficient knowledge of the framework and the computing environment in order to select the appropriate objects to compose a suitable caching strategy. Furthermore, the application requires modifications to invoke the above methods to make the necessary connections between objects. As mentioned before, this is hard to do. In addition, such modifications would be highly specific to a particular operating system, such as *Choices*. These modifications are unlikely to be portable or useful in other operating systems.

Alternatively, the application does not directly associate a specific *migration policy* with a file. By default, *Choices* attaches an *adaptive LRU replacement policy* to a file’s *memory object cache*. The *adaptive LRU replacement policy* implements the data transfer strategies, as well as, the strategy and parameter preferences described in Section 8.4.1. In other words, it determines the file’s data transfer strategies and file caching parameters from the file’s expected access behavior. Therefore, the application indirectly selects caching strategies by describing the expected access behaviors of its files to the file system caching service.

With the addition of expected file access behavior hints, two new instance variables have been added to the *PersistentMemoryObject* class. The first of these two new instance variables is *hints*. It contains the expected file access behavior of a *persistent memory object*. Each bit in this variable represents an attribute. Table 8.2 illustrates how the file access behavior attributes are encoded in this variable. An *adaptive LRU replacement policy* uses the expected file access behavior encoded in this instance variable to select strategies and modify file caching parameters.

In *Choices*, each application process has a *file system interface* (described in Section 5.3.3). An application invokes the *open* method of its *file system interface* to open a file. When it does so, it may also provide file access behavior hints to the file system. In particular, one of the arguments of the *open*
<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (0x01)</td>
<td>H_TEMP</td>
<td>temporary (true) versus permanent (false)</td>
</tr>
<tr>
<td>1 (0x02)</td>
<td>HRANDOM</td>
<td>random (true) versus sequential (false)</td>
</tr>
<tr>
<td>2 (0x04)</td>
<td>H_LARGE</td>
<td>large (true) versus small (false)</td>
</tr>
<tr>
<td>3 (0x08)</td>
<td>H_SINGLE</td>
<td>single-access (true) versus multi-access (false)</td>
</tr>
<tr>
<td>4 (0x10)</td>
<td>H_LCACHE</td>
<td>secondary cache is (true) or is not (true) beneficial</td>
</tr>
</tbody>
</table>

**Table 8.2:** Encoding of file access behavior attributes in *hints*.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 (0x00010000)</td>
<td>O_PERM</td>
<td>true if permanent</td>
</tr>
<tr>
<td>17 (0x00020000)</td>
<td>O_TEMP</td>
<td>true if temporary</td>
</tr>
<tr>
<td>18 (0x00040000)</td>
<td>O_SEQ</td>
<td>true if sequential</td>
</tr>
<tr>
<td>19 (0x00080000)</td>
<td>O_RANDOM</td>
<td>true if random</td>
</tr>
<tr>
<td>20 (0x00100000)</td>
<td>O_SMALL</td>
<td>true if small</td>
</tr>
<tr>
<td>21 (0x00200000)</td>
<td>O_LARGE</td>
<td>true if large</td>
</tr>
<tr>
<td>22 (0x00400000)</td>
<td>O_MULTI</td>
<td>true if multi-access</td>
</tr>
<tr>
<td>23 (0x00800000)</td>
<td>O_SINGLE</td>
<td>true if single-access</td>
</tr>
<tr>
<td>26 (0x04000000)</td>
<td>O_LCACHE</td>
<td>true if secondary cache is beneficial</td>
</tr>
<tr>
<td>27 (0x08000000)</td>
<td>O_NLCACHE</td>
<td>true if secondary cache is not beneficial</td>
</tr>
</tbody>
</table>

**Table 8.3:** The open flags for file access behavior hints.

The encoding of file access behavior attributes using *hints* can be seen in Table 8.2. An open method is a word containing various flags, known as the open flags. These flags indicate to the file system how the file should be opened and accessed. Traditionally, these flags indicate whether symbolic links should be followed, whether the file should be truncated, whether the specified file should be created if it does not already exist, whether write access should be permitted. With the addition of file access behavior hints, new flags have been added to the open flags to specify the expected file access behavior. Each file access behavior attribute is represented by two new flags. Table 8.3 enumerates these new flags.

Two flags are required for each file access behavior attribute because an application may specify three different preferences for each file access behavior attribute. An application may have a positive preference, a negative preference, and a neutral preference. A positive preference specifies that the value of this attribute should be true. A negative preference specifies that the value of this attribute should be false. When an application specifies a positive or negative preference, this preference is honored until another open request specifies a different preference. A neutral preference does not affect the current value of this attribute. Unlike a positive or negative preference, a neutral preference allows the file system caching service and other objects in the system to change the value of this attribute. In particular, the file system caching service may change the value of this attribute to reflect the currently observed behavior.

For example, the O_PERM and O_TEMP flags influences the value of the temporary versus permanent attribute, *i.e.* it affects the value of the H_TEMP bit of the persistent memory object’s hints instance.
variable. If the application sets the O_TEMP flag, it specifies a positive preference. This sets the
H_TEMP bit in hints. This bit remains true until another open method invocation clears it by specifying
a negative preference. If the application sets the O_PERM flag, it specifies a negative preference. This
clears the H_TEMP bit in hints. Similarly, this bit will remain false until another open method invocation
sets it by specifying a positive preference. If the application clears both the O_PERM and the O_TEMP
flags, it specifies a neutral preference. The H_TEMP bit in hints is not affected, i.e. it retains its current
value. In addition, the application allows the value of this bit to be modified by the file system caching
service if necessary. In the current implementation, if an application sets both the O_PERM nor the
O_TEMP flags, then the positive preference dominates the negative preference. Setting both flags is
discouraged since future implementations may assign a new meaning to this case.

Since the file system caching service honors positive and negative preferences specified by an appli-
cation, it must also remember the specified preferences. The current implementation stores application
supplied preferences in the second new instance variable of the PersistentMemoryObject class, known as
the openFlags.

The persistent memory object has two new public methods, known as setHints and clearHints. Figure 8.1 presents the pseudo code for these methods. These methods modify hints, but they do not
override positive and negative application preferences remembered by openFlags. In addition, these
methods invoke hintsChanged method whenever the value of hints changes. The hintsChanged method
notifies other relevant objects of changes in the file's expected access behavior.

An application uses setHints and clearHints to modify the expected access behavior of a file
after it has opened the file. Similarly, other objects in Choices uses these methods to change hints. In
particular, the adaptive file system caching service uses these methods to update hints to reflect the
observed access behavior of a file.

Since some applications may not be modified to provide expected file access behaviors, Choices
provides an alternate but more generic mechanism for specifying expecting file access behaviors for
these applications. This mechanism is inspired by UNIX's umask system call and shell command[1, 58].
In particular, it added two new instance variables to each file system interface. These variables are
openFlagsClearMask and openFlagsSetMask. In addition, four new accessor functions read and write
these variables.

When an application invokes the open method of its file system interface, the open method modifies
the open flags specified by the application in the following way before the open flags are processed or
passed to the rest of the file system:

    open flags = ( open flags and ( not openFlagsClearMask ) ) or openFlagsSetMask

133
PersistentMemoryObject::setHints(mask)
oldHints = hints;
if ( mask and H_LCACHE ) and not ( openFlags and O_NLCACHE )
hints = hints or H_LCACHE;
if ( mask and H_SINGLE ) and not ( openFlags and O_MULTI )
hints = hints or H_SINGLE;
if ( mask and H_LARGE ) and not ( openFlags and O_SMALL )
hints = hints or H_LARGE;
if ( mask and H_RANDOM ) and not ( openFlags and O_SEQ )
hints = hints or H_RANDOM;
if ( mask and H_TEMP ) and not ( openFlags and O_PERM )
hints = hints or H_TEMP;
if ( hints ≠ oldHints )
hintsChanged( oldHints );

PersistentMemoryObject::clearHints(mask)
oldHints = hints;
if ( mask and H_LCACHE ) and not ( openFlags and O_LCACHE )
hints = hints and ( or H_LCACHE );
if ( mask and H_SINGLE ) and not ( openFlags and O_SINGLE )
hints = hints and ( or H_SINGLE );
if ( mask and H_LARGE ) and not ( openFlags and O_LARGE )
hints = hints and ( or H_LARGE );
if ( mask and H_RANDOM ) and not ( openFlags and O_RANDOM )
hints = hints and ( or H_RANDOM );
if ( mask and H_TEMP ) and not ( openFlags and O_TEMP )
hints = hints and ( or H_TEMP );
if ( hints ≠ oldHints )
hintsChanged( oldHints );

Figure 8.1: The pseudo code for setHints and clearHints.

When an application process, known as the parent process, creates another new application process, known as the child process, a new file system interface is created for the child process. The file system interface of the child process inherits its openFlagsSetMask and openFlagsClearMask from the file system interface of the parent process. This allows a parent process to impose specific modifications to all the open flags supplied by the child process. Hence, the parent process can specify an expected file access behavior for all files opened by an unmodified application running in the child process. Since the modifications to the open flags specified by the parent process applies to all files opened by the child process, these modifications should be neutral with regards to an attribute unless met file accesses of the child process share a common property of the attribute. For example, if the child process opens an equal number of single-access and multi-access files, then the supplied masks should be neutral with regards to the single-access versus multi-access attribute. In other words, the parent process should not override the O_SINGLE flag of the child process. Therefore, both the H_SINGLE flag and the H_MULTI
flag should be false in both `openFlagsSetMask` and `openFlagsClearMask`. If the child process opens a lot more `single-access` files than `multi-access` files, then the supplied masks should set the O_SINGLE flag to true. Therefore, the `O_SINGLE` flag of `openFlagsSetMask` should be set and the `O_MULTI` flag of `openFlagsClearMask` should be cleared.

Like the UNIX shell, the `Choices` shell provides commands to modify the shell’s `openFlagsSetMask` and `openFlagsClearMask`. Applications executed by this shell will inherit these masks. These commands provide a simple interactive facility for specifying expected file access behaviors for unmodified applications.

In summary, an annotated application can specify a particular caching strategy for a file directly by connecting the appropriate objects together. Alternatively, it can describe the expected file access behavior of the file, and the file system caching service selects an appropriate caching strategy. This application may specify file access behaviors when it opens a file via the open flags, or it may invoke `setHints` and `clearHints` to alter a file’s expected file access behavior. Finally, an unmodified application may inherit expected file access behavior hints from its parent process.

### 8.4.3 Detecting File Access Behaviors

As mentioned in Section 8.3, an adaptive file system caching service attempts to detect the file access behavior of each file. It assumes that the observed access behavior of a file is likely to be repeated in the future. It detects the presence of various file access behavior properties and updates the `hints` instance variable of the file to reflect its observations. Since it invokes the `setHints` and the `clearHints` methods to modify `hints`, specific preferences, i.e., positive or negative preferences, specified by an application will always have a higher priority than detected properties. This subsection presents how the `Choices` adaptive file system caching service detects various file access behaviors.

**Temporary versus Permanent**

`Choices` does not automatically detect this attribute.

**Random versus Sequential**

In the current implementation, a file’s adaptive LRU replacement policy determines if the file is random or sequential. The `SmarterLRURedplacementPolicy` class is a superclass of the `AdaptiveLRUReplacementPolicy` class. A adaptive LRU replacement policy uses the `readPointer` and the `writePointer` instance variables of this superclass to dynamically determine if its file is accessed sequentially. It updates the `Il_RANDOM`
bit in *hints*. Then, it automatically adjusts the *memory object*’s data transfer strategies and file caching parameters, as described in Section 8.4.1.

The primary limitation of this implementation is that these properties are only detected if the file’s migration policy is an *adaptive LRU replacement policy*. An alternate and more general implementation modifies the *MemoryObject* class and its subclasses to detect these properties. The current implementation is adopted because *AdaptiveLRUReplacementPolicy* is the only client of the *random versus sequential* attribute, and the necessary logic is already present to detect these properties. There is no benefit in duplicating this functionality in the *MemoryObject* class. The alternative and more general implementation may be adopted in the future if necessary.

**Large versus Small**

*Choices* uses the current size of a file to determine the appropriate value for the file’s *large versus small* attribute. When the size of a file changes, the new size of the file is checked against the maximum size of a *small* file, and the *H_LARGE* bit in *hints* is updated if necessary.

**Single-access versus Multi-access**

*Choices* uses a simple criteria to determine if a file is *single-access* or *multi-access*. If the number of bytes accessed from the file’s cache exceeds the size of the file, then *Choices* assumes that the file is *multi-access*.

Each *memory object cache* maintains a counter for the number of bytes transferred between the *memory object cache* and other areas of main memory. In other words, this counter does not count transfers from main memory and various tertiary storage devices. This instance variable that holds the value of this counter is called *bytesAccessed*. If *bytesAccessed* exceeds the size of the file, then *Choices* assumes that the file is *multi-access*. Otherwise, it is *single-access*.

Since a *memory object* may be memory-mapped, not all memory accesses can be detected by its *memory object cache*. Hence, *bytesAccessed* actually contains an approximation of the actual number of bytes accessed. The approximation method is as follows:

1. Virtual memory access -

   The *cache* and the *mcache* methods of the *memory object cache* handle virtual memory faults (refer to Section 5.2.4). These faults include address translation faults and protection faults. For each invocation, these methods increment *bytesAccessed* by the number of cache units serviced by the fault.

2. File system access -

   The *cacheRead* and the *cacheWrite* methods of the *memory object cache* handles file system
cache accesses (refer to Section 6.3.3). One of their parameters specify the exact number of bytes requested. Hence, they increment bytesAccessed by the requested number of bytes.

The frequency of update for this attribute is also important. If it is updated too early, then a false single-access behavior may be detected. On other hand, if it is updated too infrequently, then detection of the multi-access behavior will be delayed. The current implementation updates this attribute whenever the file becomes inactive, i.e. when the file's persistent memory object's reference count goes to zero.

**Whether a Secondary Cache is Beneficial**

The appropriate value for attribute is also determined by a simple criteria. A secondary cache is beneficial if the data it caches will be read more than once. If the data in a secondary cache is read only once, the secondary cache actually incurs additional overhead to save a copy of data that will not be needed again in the future.

For example, a secondary cache is probably not beneficial for the following kinds of files:

1. **Write-only files** -
   It is probably more efficient to send dirty data directly to the files’ storage devices. If the written data is not to be read again, then there is no need to cache the written data in a secondary cache.

2. **Extremely frequently accessed files** -
   These files be cached in main memory most of the time. Data cached in their secondary caches are unlikely to be accessed.

3. **Infrequently accessed read-once files** -
   These files are accessed infrequently, perhaps once a day, or once a week. When accessed, each file is read entirely once. The secondary caches for these files are likely to be reclaimed before they are accessed the next time.

Examples of files that might benefit from having a secondary cache includes:

1. **Very large multi-access files** -
   Data in these files are more likely to be paged out since these files are more likely to stress the memory resource. Because they are multi-access, data that have been paged out earlier are more likely to be needed and paged in again. Since secondary caches caches this data in faster tertiary storage devices than the files’ storage devices, paging to and from these secondary caches should be faster and more efficient than paging to and from the files’ storage devices.

2. **Quite frequently read files** - These files are not frequently read enough to stay in main memory. However, they are frequently read enough that their secondary caches are not likely to be reclaimed before they are read again.

137
In the current implementation, *Choices* assumes that a file may potentially benefit from having a secondary cache, if it transfers more data into main memory from tertiary storage devices (either the files' storage device or the files' secondary cache) than the size of the file. A memory object cache transfers data from tertiary storage devices into main memory by invoking its `fromSecondaryCache` method. As mentioned earlier, this method invokes the memory object cache's top-most secondary cache to read data into main memory. This secondary cache might be a dummy secondary cache that represents the backing store memory object, or a real secondary cache that does cache data in some other memory object (refer to Section 6.3.6). If the amount of data transferred by the `fromSecondaryCache` method exceeds the size of the file, then some data in the top-most secondary cache must have been read more than once.

Each memory object cache has an instance variable that counts the number of cache units transferred into main memory by the `fromSecondaryCache` method. The instance variable is called `fromSecondaryCacheUnits`. For each invocation of the `fromSecondaryCache` method, `fromSecondaryCacheUnits` is incremented by the number of units transferred. If `fromSecondaryCacheUnits` is greater than the size of the file (expressed in cache units), then the file may potentially benefit from having a secondary cache.

The frequency of update of this attribute is also important. Like the single-access versus multi-access attribute, this attribute is updated whenever the file becomes inactive.

Although applications may specify hints or preferences for this attribute using the mechanisms described in Section 8.4.2, applications typically do not specify such hints and preferences. This is because the appropriate value for this attribute depends on both the amount of contention for main memory and the amount of contention for secondary cache memory, in addition to the file access characteristics of the application. Application developers probably would not know the amount of expected contention. Hence, it is more appropriate for the adaptive file system caching service to determine the value of this attribute for each files.

Unlike the other attributes, this attribute does not directly affect caching strategy selection. In other words, it does not cause a secondary cache to be attached whenever its value is true. As mentioned earlier, it merely indicates whether an observed file access behavior may benefit from having a secondary cache. Whether a particular file may benefit from having a secondary cache, also depends on the file itself. Furthermore, the file also determines which kinds of secondary caches are suitable. For example, a UNIXinode cache would not be suitable for a UNIXinode. On the other hand, a UNIXinode cache would be appropriate for a remote file object. Hence, each `PersistentMemoryObject` subclass implements the necessary functionality to observe this attribute and attach suitable secondary caches to their memory object caches when appropriate.

Currently, the remote file object is the only kind of persistent memory object that attaches a secondary cache to its memory object cache when a secondary cache is beneficial. A remote file object observes this
attribute and it attaches an UNIX inode secondary cache to its memory object cache when this attribute is set.

### 8.4.4 Remembering File Access Behaviors

As mentioned in Section 8.4.2, the expected access behavior of a file is held in the hints instance variable of the persistent memory object that represents the file. This means that the file system caching service remembers the expected access behavior of a file as long as its persistent memory object remains alive. In other words, the expected file access behavior of a file will be lost when the persistent memory object is eventually reclaimed, or when the operating system is rebooted. Typically, a persistent memory object becomes a zombie when it becomes inactive. The zombie persistent memory object will be reclaimed if it remains inactive. Since the same file access behavior is likely to be repeated when the file is accessed again, it is desirable to remember the expected access behavior of the file even after its persistent memory object has been reclaimed. This means that the adaptive file system caching service should make the expected file access behavior of a file persistent. The expected file access behavior to remember might be specified by an application as described in Section 8.4.2 or automatically detected by the adaptive file system caching service.

Making expected file access behaviors persistent is especially important for the single-access versus multi-access attribute and the whether a secondary cache is beneficial attribute. If these attributes were not made persistent, then the values of these attributes for a file that is being opened for the first time is unknown. As mentioned in Section 8.4.3, the first update of these attributes will occur only after the file becomes inactive for the first time. In the initial implementation, default values are assigned to these attributes. In the current implementation, the previously saved values of these attributes are available before the “first” update of these attributes.

In current implementation, Choices saves hints as part of the meta-data of a file whenever possible. In the case of an existing file type, it is only possible to save hints if there are sufficient spare space available in the data structure that represents the file on the file’s storage device. For example, the on-disk inode data structure of a BSD inode contains several unused spare words. Hence, the BSD_inode class can save the expected access behavior of a BSD inode in one of these words. In general, the amount of spare space available in existing file types are limited. Choices describes and encodes file access behaviors using binary attributes because of this lack of spare space for saving a file’s expected access behavior.

Other methods to describe and encode expected file access behaviors have been investigated. An alternative method describes and encodes an expected file access behavior using a general purpose
property or attribute list.\(^6\) It's main advantage is flexibility. It can support a variable number of attributes, and the size of each attribute value may also be variable. It is also extensible since new attributes may be added easily. It's main disadvantage is that data structures of different sizes would be required to record these attributes. Variable-size data structures are usually less efficient to maintain and access than constant-size data structures. Typically, more CPU and I/O resources will be required to manipulate variable-size structures.

Another method describes and encodes each attribute with some numeric value. This numeric value may be an weighted average that favors the most dominant file access behavior observed for a file. If there is a fixed number of attributes, then constant-size structures may be used to store an expected file access behavior. As mentioned earlier, the current implementation uses a binary encoding. It is a specialization of this method in which each attribute can have only two possible values. Choices uses this encoding because it is compact. An expected file access behavior can be encoded in the limited amount of spare space available in existing file systems for each file's meta-data. It is also more efficient since the encoded expected file access behavior can be saved and retrieved as part of the file's meta-data. Furthermore, binary encoding are easy to manipulate using existing binary operators. This is important since the Choices adaptive file system caching service accesses and modifies attributes frequently.

In general, an adaptive file system caching service should avoid introducing additional I/O's that only save and retrieve expected file access behaviors from tertiary storage devices. One such source of additional I/O's is uncoordinated or poorly coordinated in-memory updates of a file's expected access behavior with updates that save this expected file access behavior to a tertiary storage device. As mentioned earlier, a BSD inode saves its expected file access behavior, i.e. hints, in its on-disk inode data structure. It should not incur additional I/O's that just modifies its on-disk hints. It should "piggy-back" modifications to the on-disk hints onto existing required inode updates.

Coordinating updates is one of the reasons for updating the single versus multi-access attribute and the whether a secondary cache is beneficial attribute when a persistent memory object becomes inactive (refer to Section 8.4.3 and Section 8.4.3). When a persistent memory object becomes inactive, it may be destroyed immediately, i.e. it does not become a zombie. Usually, the persistent memory object is synchronized before it is destroyed. Since updates to the abovementioned attributes occur before the persistent memory object is synchronized, modifications to these attributes may be piggy-backed onto writes that modify the persistent memory object's meta-data, such as the file's last access time or last update time. On the other hand, if updates to the above attributes take place after the persistent memory object has been synchronized, then an additional I/O will be required to save the latest changes to these

\(^6\)Such as an "environment" in UNIX systems\(^3\). An UNIX "environment" contains a list of strings. Each string contains the name of an attribute and the value of the attribute. For example, "SHELL=/bin/sh" is one such string. UNIX provides `getenv()` and `putenv()` functions to manipulate environments.
attributes. Alternatively, if the persistent memory object becomes a zombie when it becomes inactive, updating these attributes when the persistent memory object becomes inactive is also appropriate. This is because there will be no further application accesses to the zombie persistent memory object. Hence, the values of these attributes will not change after they have been updated. Eventually, modifications to these attributes will be saved when the zombie persistent memory object is reclaimed and destroyed.

In summary, an adaptive file system caching service should remember expected file access behaviors. However, it should minimize additional overhead resulting from making expected file access behaviors persistent. Choices does so by using binary file access behavior attributes and saving these attributes in the spare bits of each file's meta-data. In addition, it coordinates in-memory and persistent updates of these attributes with updates to the file's meta-data.

8.4.5 Distribution

Distribution introduces the need to support more than one expected access behavior per file. This is because a file's client-side behavior may be very different from its server-side behavior. Hence, distribution requires the adaptive file system caching service to detect and remember both client-side and server-side file access behaviors. This subsection discusses these needs and requirements in greater detail. It also describes how the current implementation and other implementations may address these needs and requirements.

Client-side and Server-side File Access Behaviors

In a distributed file system, an application running on the client may access a remote file residing on a file server. The client may cache data belonging to this file in its main memory or its tertiary storage devices. When this occurs, the client observes all file accesses generated by the application. The server only observes file accesses generated by the client on behalf of the application. The server does not observe client application file accesses that can be satisfied with data residing in client cache. Hence, the client and the server observes or detects different file access behaviors for the same file. Therefore, a single expected file access behavior for each file is insufficient. If there is a single expected file access behavior per file, then the single expected file access behavior is either the file's client-side expected access behavior or the file's server-side expected access behavior, but not both.

For example, a client may cache a read-only multi-access file in main memory or a local disk. This file might be a system header file, such as "stdio.h", that is included by many source files. When these source files are compiled as part of a "make", each source file that includes it accesses it once. If the compiler on the client accesses this file frequently enough, then the file's data will not become stale in

\[\text{In an UNIX system, "stdio.h" contains the descriptions and function signatures of the standard I/O routines. These routines are commonly used to perform I/O and generate formatted output.}\]

141
the client’s cache. The client will observe the many accesses to the same data by the compiler. However, the same data is accessed only once on the server when server accesses the data on behalf of the client. In other words, this header file has two different observed file access behaviors. It is multi-access to the client and single-access to the server. Similarly, a file may be random to the client but sequential to the server. The probability of observing such differences in file access behaviors increases with the size of the client cache. In particular, the scenario discussed in this example is very likely to occur when the client caches remote data in secondary caches since these secondary caches increase the amount of memory available for caching on the client.

Detecting File Access Behaviors

The advantages of the Choices object-oriented design and frameworks naturally allows a file’s access behaviors to be detected independently on the client and the server. As mentioned in Section 7.3, a remote file represented by a remote file object on the client. Since the RemoteMemoryObject class is a subclass of the PersistentMemoryObject class, it inherits hints and openFlags from the PersistentMemoryObject class. A remote file object uses these instance variables to track the file’s client-side expected access behavior. (as described in Section 8.4.2). An adaptive LRU replacement policy may also be attached to the remote file object to select strategies and assist in file access behavior detection (as described in Section 8.4.3).

Similarly, the server threads on the file server accesses a remotely accessed file on the client’s behalf like any other application on the server. For example, if the file resides on a BSD file system, then a BSD inode represents this file on the server. Since the BSDinode class is a subclass of the PersistentMemoryObject class, the BSD inode also uses hints and openFlags to track the file’s server-side expected access behavior. In addition, it makes its server-side expected file access behavior persistent by storing hints in a spare word of its on-disk inode structure. Similarly, it may employ an adaptive LRU replacement policy to select strategies, and assist in file access behavior detection.

In summary, the existing file system caching service provides the necessary functionality and features to detect a file’s client-side and server-side access behaviors. It can also remember the file’s server-side access behavior on the server. However, it does not remember the file’s client-side access behavior. Basically, the client-side access behavior of a file may be stored on the client or the server.

Storing Client-side File Access Behaviors on the Client

To store client-side access behaviors on the client, the existing file system implementation may be extended to write these behaviors to a persistent storage device on the client. For example, a remote file object may be modified to write the client-side access behavior of a remote file to a local disk when it is reclaimed. When the distributed file system instantiates a remote file object to represent the same
remote file again, the **remote file object** retrieves the remote file’s client-side access behavior from the local disk.

This implementation has several problems. Persistent storage may not be available on some clients, such as diskless workstations. This implementation also introduces additional accesses on the client to save and retrieve client-side file access behaviors.

Additional accesses on the client may result in significant performance degradation, especially for small files. For example, a small file may need only two accesses on the server to retrieve all its data. The first retrieves its meta-data and the second retrieves the data in the file. In addition to the two server accesses, the client may need an additional access to retrieve the file’s client-side access behavior. This implementation introduces 50% more accesses for small files. Another problem is garbage collection of saved client-side file access behaviors. For example, saved file access behaviors belonging to files deleted by other clients become garbage. This problem may be solved by building a cache for client-side file access behaviors. This cache would limit the amount space occupied by saved client-side file access behaviors. It would probably also implement a cache replacement policy to remove stale cache entries.

**Storing Client-side File Access Behaviors on the Server**

To store client-side access behaviors on the file server, the client sends the client-side access behaviors to the server. The server stores the client-side access behaviors in its storage devices on behalf of the client. When required, the server also retrieves and sends the stored client-side access behaviors to the client.

A server can also store and retrieve client-side access behaviors more efficiently. For example, no additional accesses is required to retrieve a file’s client-side access behavior if the client-side access behavior is stored in the same block as the file’s meta-data. Garbage collection is also trivial. The file system simply deletes all stored access behaviors, client-side access behaviors included, associated with a file when the file is deleted.

A server determines how it will store and retrieve client-side file access behaviors. There are basically two different ways that a server may store and associate client-side expected file access behaviors with each file.

**Storing Many Client-side File Access Behaviors per File**

The first way stores a client-side access behavior for each file and each client. In other words, the server stores \( n \) client-side access behaviors for a file with \( n \) clients. Typically, this requires the file system to maintain variable-size data structures to remember each file’s client-side access behaviors. As mentioned before, variable-size data structures consume more space and are less efficient to access. The amount of
storage consumed by client-side access behaviors should also be bounded. Hence, some garbage collection mechanism would be needed to reclaim storage occupied by stale client-side access behavior data.

**Storing One Client-side File Access Behavior per File**

The second way stores a single client-side access behavior per file. This client-side access behavior applies to all of the file’s clients. Since the file system remembers a single client-side access behavior for all the file’s clients, it may use a fixed-size data structure to remember this behavior data. Like, the file’s server-side access behavior, the client-side access behavior data may be encoded in the file’s meta-data. For example, the client-side access behavior of a BSD inode can be stored in another spare word in the inode’s data structure.

Unfortunately, storing a single client-side access behavior per file results in a loss of behavior information. However, this loss of information may be acceptable in computing environments where sharing is rare, or where “inertia”\(^8\) is dominant. Alternately, weighted averaging techniques may be applied so that the most dominant client-side access behavior is saved.

**Current Implementation**

The *Choices* distributed file system permits storing both one or many client-side file access behaviors per file. A *Choices* client sends a file’s client-side access behavior, *i.e.* the remote file object’s hints, to the server. The server passes this client-side access behavior to the persistent memory object that represents the file on the server. This persistent memory object determines how it will store the client-side access behavior.

For example, a persistent memory object may create a tuple that contains the client’s identity and the file’s client-side access behavior, and store this tuple in the file’s meta-data. Alternatively, it may use the received client-side access behavior to update weighted averages for each access behavior attribute. In the simplest form, it simply stores the received client-side access behavior, and this behavior applies to all clients until a new client-side behavior is received.

In the current implementation, the BSD_inode class is the only PersistentMemoryObject subclass that stores and retrieves client-side file access behaviors. The simplest form is implemented. It is sufficient for the initial prototype since little sharing is expected in the initial computing environment and “inertia” is expected. Furthermore, it allows for efficient encoding of client-side access behavior data in the limited number of spare bits available in the on-disk inode data structure.

---

\(^8\)Refer to Section 8.3.
In addition, remote containers and remote dictionaries also send open flags to the server. The server passes these open flags to the corresponding remote containers and remote dictionaries on the server. Hence, the server can take advantage of hints provided by client applications via open flags.

**Summary**

In summary, distribution introduces client-side and server-side file access behaviors. An adaptive file system caching service should be able to detect both client-side and server-side file access behaviors. It should also provide the necessary mechanisms to remember client-side and server-side file access behaviors. In particular, a client should be able to share a file’s client-side access behavior with the file’s file server. This allows the server to remember the file’s client-side access behavior on the client’s behalf. The server may also take advantage of additional client-side knowledge to optimize its own file accesses.

### 8.4.6 Extensions for Different Computing Environments

The previous subsections discussed the current prototype implementation of an adaptive file system caching service in *Choices*. This prototype’s target computing environment is a local area network of workstations. Its primary users are researchers in the computer science department. This subsection discusses some possible future extensions to this prototype to support different computing environments.

#### Network Bandwidth

Some high speed networks require large data transfers to achieve maximum throughput (refer to Section 2.1.3). The prototype implementation may be modified to generate large transfers on these networks.

As mentioned before, each remote file object represents a remote file residing on the file server. This remote file object indirectly controls the size of data transfers across the network. It determines the cache unit size and the cluster size of its memory object cache and migration policy. The cache unit size provides the lower bound on the smallest amount of data transferred. The cluster size provides the optimal data transfer size for migration policies that implement clustering.

To generate large transfers on high speed networks, the RemoteFileObject class may be modified to choose an appropriate cache unit size and cluster size for each remote file according to the bandwidth characteristics of the network. For example, each network interface would define a method that provides access to the characteristics of the network. When a revised remote file object is instantiated, it would invoke this method to obtain the network’s bandwidth characteristics and compute a suitable cache unit size and cluster size for its memory object cache and migration policy.
Network Latency

In a wide-area network where the client and the file server are far apart, network latency is much larger than in a local area network (refer to Section 2.1.1 and Section 2.1.3). The prototype implementation may be extended to reduce the effects of high network latencies.

For example, the distributed file system server and client could cooperate to compute the network latency between the client and the file server. This latency may be computed by the underlying communications protocol or by the distributed file system. In either case, a remote file object would have access to the computed network latency and it would affect caching strategies accordingly. When network latency is high, it could increase its data transfer size by selecting a large cache unit size and cluster size. It could also increase concurrency by increasing data prefetching, i.e. read-ahead, and write-behind. In addition, it could employ a secondary cache to cache remote data (refer to Section 2.1.1).

8.5 Summary

It is difficult and impractical to modify a lot of existing applications to select appropriate caching strategies for each file. One problem is that the application developer may not have sufficient knowledge of the computing environment and the computing may evolve over time. Another problem is that some applications cannot be modified. In addition, it is tedious to modify existing applications.

Adaptive file system caching solves these problems. With adaptive file system caching, a modified application may provide the expected access behaviors to the file system caching service. The adaptive system caching service selects an appropriate strategy for each file based on the expected access behavior of the file. This file system caching service may select different strategies for the same expected file access behavior in different computing environments. Alternatively, each computing environment may have its own customized file system caching service that implements and selects strategies that are best suited to the environment. Similarly, when a computing environment evolves, its file system caching service could evolve with the environment. In other words, a modified application provides knowledge of its files’ expected access behaviors, and the file system caching service provides knowledge of the computing environment.

In the case of an un-modified application, an adaptive file system caching service observes the file accesses of the application, and it attempts to detect the expected access behaviors of the application’s files. Then, it uses the detected expected file access behaviors to select appropriate caching strategies. Hence, it allows an un-modified application to take advantage of the various strategies in the file system caching service.

A prototype implementation of an adaptive file system caching service has been presented. It describes expected file access behaviors with binary attributes. The prototype uses these attributes to select
caching strategies and change file caching parameters. It also detects both client-side and server-side file access behaviors. Its file servers store a single client-side access behavior for each file.

The following chapter describes the experiments conducted on the prototype to demonstrate the advantages of adaptive file system caching.
Chapter 9

Performance Evaluation

To demonstrate the benefits of adaptive file system caching, several experiments have been conducted on the Choices distributed file system. These experiments use two workloads to compare Choices’ adaptive caching strategy to other non-adaptive caching strategies. Each workload represents a computational task, and these workloads have distinct file access characteristics. The experimental result shows that the adaptive caching strategy consistently outperforms the non-adaptive caching strategies.

This chapter describes the experiments, the caching strategies evaluated, and the criteria used to evaluate the different strategies. It also describes the file access behaviors of the two distinct workloads. Finally, it analyzes the experimental data, presents several key observations, and shows the benefits of adaptive file system caching.

9.1 The Experiments

Due to a lack of native applications in Choices, these experiments depend on trace-driven workloads to evaluate different caching strategies. A trace-driven workload represents the file access activities of one or more application programs that have been executed to complete a computational task. It consists of a sequence of trace records, and each trace record in this sequence represents a file access system call that has been issued by a program. Replaying a workload in Choices simulates the file access activities required to perform a computational task in Choices.

These experiments use two distinct workloads that represent two different computational tasks. These tasks have very different and distinct file access behaviors. The two workloads are:

1. Building the Choices kernel -
   It contains the file access activities of the various programs required to build the Choices kernel. These programs include the GNU C++ compiler, GNU make utility, and the linker.
2. SMS -

SMS is one of the programs in the *perfect benchmarks* [83, 27]. The perfect benchmarks evaluate the performance of supercomputers. Its file access activities dominate the file access activities of the perfect benchmark.

These experiments evaluates following caching strategies:

1. Two-handed clock LRU (LRUCLK) -

   Many existing virtual memory management systems and file systems implement this strategy. It performs block level LRU replacement of cached data. The *GroupLRUReplacementPolicy* class implements this strategy (refer to Section 6.3.7.2). When a file becomes inactive, it becomes a zombie. However, resources consumed by the zombie is not reclaimed using the zombie reclamation optimization described in Section 6.3.10. In other words, LRUCLK treats data cached by active files and zombie files equally.

2. LRUCLK with zombie reclamation (ZOMBIE) -

   In addition to LRUCLK, it includes the zombie reclamation optimization. It attempts to reclaim memory from stale zombies before activating the LRU pager. It uses the two-handed clock LRU replacement algorithm only when it cannot reclaim memory from stale zombies. It reclaims the least-recently-active stale zombie first.

3. ZOMBIE with read-ahead, write-behind, and free-behind (SMART) -

   In addition to ZOMBIE, it adds the read-ahead, write-behind, and free-behind data transfer strategies (refer to Section 6.3.7.4). The *SmarterLRUReplacementPolicy* class implements this strategy.

4. SMART with local disk caching enabled (LCACHE) -

   In addition to SMART, it caches remote files on a local client disk. In these experiments, a client caches remote file data on local disk. It uses a UNIX inode secondary cache to cache data from a remote file in a local BSD inode. Section 6.3.6 describes how the client manages and reclaim resources consumed by its UNIX inode secondary caches. Usually, the client reclaims the least-recently-used UNIX inode secondary cache first.

5. Adaptive file caching (ADAPTIVE) -

   This is the *Choices* implementation of adaptive file system caching as described in Section 8.4. It selects an appropriate caching strategy for each file based on the file’s access behavior. This includes caching remote files on a local client disk when appropriate.

   In each experiment, the file server and the client always employ the same caching strategy, and only the client runs the workload.
The following performance measures will be used to evaluate each caching strategy.

1. Cache miss indicator -
   It is the amount of data read into main memory from external storage while executing a workload. A more commonly used measure of cache efficiency is the cache miss ratio or the cache hit ratio. The cache miss ratio is essentially the cache miss indicator divided by the total amount of data accessed. The denominator is not necessary since the total amount of data accessed while executing a given workload is always a constant. In addition, it is difficult to determine accurately the actual amount of data or memory accessed because some files are memory mapped.

   A cache miss indicator is obtained by instrumenting the MemoryObjectCache class. It is the total amount of data read by each invocation of the read method on a memory object cache's top-most secondary cache. The unit of this measure is the physical memory page size. This is because the smallest entity that the virtual memory management system can independently manipulate is a physical memory page, i.e. the smallest possible cache unit size is the physical memory page size, and all larger cache unit sizes must be multiples of the physical memory page size.

   Comparisons based on cache miss indicators also avoid an ambiguity when read-ahead is employed. When read-ahead is employed, a unit may be paged into memory before it is accessed. When this unit is eventually accessed, it is already present in main memory. Some studies do not include this unit in their cache miss ratios since accessing this unit does not trigger a cache miss fault. As a result, these studies usually observe very low cache miss ratios when read-ahead is employed. The cache miss indicator includes this unit since the unit was not originally in memory and had to be paged into memory. Read-ahead reduces latency, its effectiveness is evaluated better by runtime.

2. Network load -
   It is the total number of bytes of file data transferred over the network while executing a workload. Fewer bytes transferred indicate a better strategy. Since the goal is to comparing caching strategies, network traffic resulting from directory operations is excluded.

   Since file servers are either the sink or source file data transferred over the network, the network load also indicates the demand for server resources, i.e. the server load. Hence, a higher network load indicates that more file data has been transferred over the network. Therefore, more server resources is required to sink or source this data.

3. Runtime -
   It is the time required to complete a workload. The runtimes obtained from two different strategies may differ even when the two strategies have similar cache miss ratios and network loads. This is because one of these strategies may have higher latencies. For example, read-ahead reduces
latencies and results in lower runtimes. When appropriate, runtimes for the a workload running on SunOS 4.1.3 and NFS will be provided for comparison.

The remainder of this section describes how the trace-driven workloads were obtained, the advantages and disadvantages of trace-driven workloads, and the file access behaviors of the two experimental workloads in greater detail. The next section analyzes the experimental data and presents key observations.

### 9.1.1 Trace-driven Workloads

The experimental workloads were obtained with application profiling tools[24, 62]. The application profiling tools include:

1. **the logging library** -
   
   It replaces the standard system library, i.e. libc, and is linked into application programs that are to be traced. It intercepts system calls and writes a trace record to a named pipe[1], known as the logging pipe, for each traced system call. A trace record describes a system call event. It typically contains a number that represents the system call, a process identifier (pid), a process virtual time, a process elapsed time, relevant input parameters, output parameters, and a return status. Since pipes are buffered by the kernel, the additional overhead incurred by the logging library is very low (rarely more than 0.3%)[24]. The logging library does not record I/O activities resulting from virtual memory faults.

   For systems without dynamic linking, the logging library is linked into the application at link time. For systems with dynamic linking, such as SunOS, the logging library is a shared library[117]. A dynamically linked application loads and links the logging library instead of the standard system library at runtime.

2. **the logging daemon** -
   
   It reads the logging pipe and writes the trace records out to log files. It maintains an index file so that trace records from each run of a particular application program can be located quickly.

3. **the analysis programs** -
   
   They help to analyze collected data and to characterize the traced applications. We use them to study the file access characteristics of the traced applications. They can also manipulate a stream of trace records. For example, a new stream of trace records may be created from an existing stream of trace records, by selecting specific types of trace records, or by combining trace records to coalesce consecutive sequential I/O’s.

4. **the log file player** -
   
   It takes a stream of trace records as input and replays the I/O events described by the trace
records. It has been ported to Choices. It reproduces I/O activities recorded on another operating system, such as UNIX, in Choices.

As mentioned earlier, a trace-driven workload contains file access events required to complete a computational task. A workload is obtained by tracing application programs that are invoked as part of the computational task and extracting the application programs’ trace records from the log files. The extracted trace records may be manipulated to eliminate unnecessary or unsupported system calls events.

Advantages

A lack of native Choices applications is the primary reason for using trace-driven workloads. However, replaying trace-driven workloads also has the following advantages over running native applications:

1. Repeatability -

A trace-driven workload can be replayed many times and its file accesses will always be the same. Running native applications is not as consistent. For example, many studies use the Andrew benchmark[43] compare caching strategies and file system implementations. This benchmark copies files, compiles source files, and links object files. Depending on the copy program, compiler and linker used, vastly different file access behaviors may be observed. The file access behaviors of Sun’s and GNU’s C compilers are different. Similarly, the Choices GNU-derived copy program (cp) does not have the same file access behavior as SunOS 4.1.3’s copy program. Therefore, benchmarks that depend on native programs might compare program implementations instead of comparing caching strategies and file system implementations. This is undesirable for a test workload or benchmark that could be used to compare more than one operating system.

2. Fairness -

As far as possible, only file access performance should be compared. A trace can be manipulated to remove system call events that are not related to file access, such as fork, exec, console I/O, and pipe I/O. This minimizes effects due to differences in the handling of these events by different operating systems. For example, the exec system call loads and runs an executable program. The content of an executable program depends on the operating system, as well as, the processor architecture. Hence, this program may be loaded in different ways.

For example, SunOS 4.1.3 supports dynamic loading but UNICOS and Choices do not. SunOS 4.1.3 is likely to generate significantly fewer file accesses to load an executable program than UNICOS or Choices. It can “reuse” the shared libraries have already been loaded as part of loading another application. Similarly, even if the same binary format, such as COFF or ELF,
is used, different processor architectures have different executable binaries for the same program, resulting in differences in file accesses generated to load the program. Since the goal is to compare file system caching strategies and not the performance of program loading, the effects of program loading should be minimized. Replaying a trace-driven workload only loads a single program, the log file player.

3. **Flexibility** -
   A trace-driven workload can be manipulated to try other possibilities, such as combining trace records to coalesce consecutive I/O’s, and changing the timing relationship between events to speed up or slow down computations.

**Disadvantages**

The primary disadvantage associated with trace-driven workloads is that gains and loses observed in experiments may not reflect real gains or loses observed when running native programs. The following factors affect the applicability of observations derived from trace-driven workloads to real tasks:

1. **Trace data** -
   The log file player introduces additional file accesses to read trace data. For the two experimental workloads, the additional amount of trace data read by the log file player is less than 5% of the total amount of data accessed by each workload. The log file player is optimized to read trace data in large blocks to reduce the number of additional read system calls to load trace data.

2. **Other system activities** -
   As mentioned earlier, some system activities have been removed from each workload. These activities include fork and exec. Therefore, file accesses associated with these activities are excluded.

3. **Time between file access events** -
   Due to elimination of some system activities, the actual time between two file access activities also changes.

   In some experiments, computation time between two events may be ignored. The time between two events depends on the speed of the processor on which the trace data is obtained. Therefore, it may be desirable for the workloads to be independent of processor speed.

4. **Memory consumption** -
   When replaying trace-driven workloads, operating systems with dynamically sized file system caches allows main memory that would be required to cache native application code and data to be used for caching file system data. The net effect is that the host appears to have more mem-
ory available for caching file data. Both Choices and SunOS have dynamically sized file system caches.

Despite the above disadvantages, replaying trace-driven workloads is suitable for these experiments. Replaying trace-driven workloads allows us to evaluate different caching strategies with file access activities derived from real computational tasks without porting the tasks’ application programs to Choices. They are also more appropriate for comparing different file system implementations and caching strategies.

9.1.2 Workload Characteristics

The two workloads used in these experiments are building the Choices kernel and SMS. They have diversely different file access behaviors. SMS is a single program. It opens few files and one of the files it accesses is very large and dominates the its file accesses. Building Choices involves many programs. It accesses many small files and most files are accessed sequentially. The rest of this subsection describes the file access behaviors of these test workloads in greater detail.

Building the Choices kernel

This workload represents a partial build of the SPARCstation 2 version of the Choices kernel on a SPARCstation 2 (ss2t1.cs.uiuc.edu) running SunOS 4.1.3.

This workload accesses the following kinds of files:

1. header file (*.h) -
   A header file is a text file that is included by other files. It may also include other header files. It contains preprocessor, C++, and/or assembly language statements. These statements contain declarations that will be required by other source files or header files. These declarations include constants, type definitions, and function signatures. Table 9.1 shows the file size distribution of header files. Most (95.4%) of these files are very small (less than 8kb).

<table>
<thead>
<tr>
<th>size (bytes)</th>
<th>&lt; 8k</th>
<th>≥ 8k</th>
<th>≥ 16k</th>
<th>≥ 32k</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>352</td>
<td>14</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 9.1: File size distribution of header files.

2. source file (*.s, *.cc) -
   A source file is also a text file. Like a header file, it also contains preprocessor, C++, and/or assembly language statements. Typically, these statements implement functions, classes, and types declared in header files. A source file usually includes a few header files. Unlike header files, source
files are not included by other source or header files. Table 9.2 shows the file size distribution of source files. These files are usually small (98.6% less than 64kb) and accessed only once.

<table>
<thead>
<tr>
<th>size (bytes)</th>
<th>&lt; 8k</th>
<th>≥ 8k</th>
<th>≥ 16k</th>
<th>≥ 32k</th>
<th>≥ 64k</th>
<th>≥ 128k</th>
<th>≥ 256k</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>163</td>
<td>33</td>
<td>16</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Table 9.2:** File size distribution of source files.

3. **object file (*.o, Choices)** -

An object file is the result of compiling a source file, or linking one or more object files. It contains binary data. The binary data includes native machine code, static data, constants, symbol information for linker/loader relocation and debugging. Table 9.3 shows the file size distribution of object files, excluding the *Choices* kernel. The file size of *Choices* kernel is 7426640 bytes, it is also the largest file accessed.

<table>
<thead>
<tr>
<th>size (bytes)</th>
<th>&lt; 8k</th>
<th>≥ 8k</th>
<th>≥ 16k</th>
<th>≥ 32k</th>
<th>≥ 64k</th>
<th>≥ 128k</th>
<th>≥ 256k</th>
<th>≥ 512k</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>40</td>
<td>46</td>
<td>53</td>
<td>43</td>
<td>21</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 9.3:** File size distribution of object files, excluding the kernel.

4. **archive file (*.a)** -

An archive file is a file that contains a collection of object files. It is also known as a library. It is accessed by the linker. Table 9.4 shows the file size distribution of archive files.

<table>
<thead>
<tr>
<th>size (bytes)</th>
<th>&lt; 128k</th>
<th>&lt; 256k</th>
<th>&lt; 512k</th>
<th>&lt; 1M</th>
<th>≥ 1M</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 9.4:** File size distribution of archive files.

5. **makefile (Makefile, *.mk)** -

A makefile is a text file that contains rules for building object files from source files, and the kernel from object files. A makefile may include other make files to inherit rules defined in the other make files. Table 9.5 shows the file size distribution of makefiles.

6. **dependency file (*.dep)** -

A dependency file contains a list of other files that a particular make target depends on. For example, the dependency file for an object file abc.o contains abc.cc and stdio.h if it is compiled from a source file named abc.cc and abc.cc includes stdio.h. Dependency files are created by the preprocessor, *i.e.* **cpp**, and read by the make utility, *i.e.* **gnumake**. Table 9.6 shows the file size distribution of dependency files.
Table 9.5: File size distribution of makefiles.

<table>
<thead>
<tr>
<th>size (bytes)</th>
<th>&lt; 8k</th>
<th>&lt; 16k</th>
<th>≥ 8k</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>7</td>
<td>&lt; 4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 9.6: File size distribution of dependency files.

<table>
<thead>
<tr>
<th>size (bytes)</th>
<th>&lt; 8k</th>
<th>&lt; 16k</th>
<th>≥ 8k</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>263</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 9.7: File size distribution of all files accessed.

<table>
<thead>
<tr>
<th>size (bytes)</th>
<th>&lt; 8k</th>
<th>&lt; 16k</th>
<th>&lt; 32k</th>
<th>&lt; 64k</th>
<th>&lt; 128k</th>
<th>&lt; 256k</th>
<th>&lt; 512k</th>
<th>&lt; 1M</th>
<th>&lt; 2M</th>
<th>≥ 2M</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>1036</td>
<td>214</td>
<td>136</td>
<td>68</td>
<td>35</td>
<td>7</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 9.8 describes each program invoked to build the Choices kernel. The make utility (gnumake) initiates a build of the Choices kernel. First, gnumake reads a master makefile, and its 10 included makefiles. These makefiles indicate which target object files are need to link the Choices kernel. Then, gnumake reads 311 dependency files. It reads each makefile and dependency file once and sequentially. To build the Choices kernel, it invokes shell scripts specified by rules in the makefiles to generate the target object files and to link these object files into the Choices kernel. It also invokes other programs such as sed to generate source files, and cmp to verify the contents of certain header and source files.

The shell script that compiles each of the 200 C++ source files invokes echo to output a message to the console, rm to remove the target object file’s current dependency file, and the compiler driver (gcc) to compile the source file. The compiler driver invokes cpp to preprocess the C++ source file, cplusplus to compile the preprocessed C++ source into assembly language code, and as to compile the assembly language code into object code. These programs communicate via pipes. Cpp reads the C++ source file and each of its included header files once sequentially. Then, it creates a dependency file for the target object file. It writes sequentially to the dependency file once. Cplusplus performs no file accesses, it only reads from and writes to its input and output pipes. As opens an output file and two temporary files. Figure 9.1 illustrates how as accesses its files.

The shell script that compiles each of the 7 assembly language source files invokes echo to output a message to the console, rm to remove the target object file’s current dependency file, cpp to preprocess the assembly language source file, and as to compile the preprocessed source into the target object file.

The final step in building the Choices kernel is to link the object files compiled earlier into the executable kernel. The script that does this invokes echo to print a message to stdout. Then, it invokes gcc with a list of object files and libraries to link. First, gcc invokes cplusplus, cplusplus invokes ld to
<table>
<thead>
<tr>
<th>Program</th>
<th>Count</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gnumake</td>
<td>1</td>
<td>GNU make utility</td>
</tr>
<tr>
<td>sed</td>
<td>1</td>
<td>stream editor</td>
</tr>
<tr>
<td>nm</td>
<td>1</td>
<td>utility to examine an object file’s symbol table</td>
</tr>
<tr>
<td>ld</td>
<td>2</td>
<td>SunOS 4.1.3 linker</td>
</tr>
<tr>
<td>cmp</td>
<td>5</td>
<td>compare two files, reads the two input files sequentially</td>
</tr>
<tr>
<td>cc</td>
<td>203</td>
<td>GNU C++ compiler</td>
</tr>
<tr>
<td>gcc</td>
<td>203</td>
<td>GNU C++/C compiler driver</td>
</tr>
<tr>
<td>cpp</td>
<td>209</td>
<td>GNU preprocessor</td>
</tr>
<tr>
<td>as</td>
<td>209</td>
<td>assembler</td>
</tr>
<tr>
<td>rm</td>
<td>209</td>
<td>delete file</td>
</tr>
<tr>
<td>echo</td>
<td>209</td>
<td>echo, display line on console</td>
</tr>
<tr>
<td>sh</td>
<td>217</td>
<td>Bourne shell</td>
</tr>
</tbody>
</table>

Table 9.8: Programs invoked to build the Choices kernel.

create the target object file
write sequentially to the target object file
create 2 temporary files
interleaved sequential writes to the two temporary files
sequentially read all of the first temporary file
write sequentially to the target object file
sequentially read all of the second temporary file
write sequentially to the target object file
delete the 2 temporary files

Figure 9.1: As’ file access behavior.

perform a trial link. Upon completion of the trial link, cc| plus invokes nm to read the symbol table contained in ld’s output file. Cc| plus and nm communicate via pipes. Cc| plus reads the symbol table to generate a temporary source file to satisfy some missing symbols, and it invokes gcc again to compile this generated source. Finally, it invokes ld again to link the kernel with the object file compiled from the source file generated by cc| plus included in the list of object files to be linked.

The file access behavior of ld is as follows:

1. Read the first 8kb of each object file -
   In the first phase, ld opens each object file, reads the first 8 kb or less if the object file is less than 8kb, then closes the object file.

2. Randomly read each object file -
   In the second phase, ld opens each object file, and randomly reads the object file in 8 kb blocks. It may read some portions of the object file more than once. Then, it closes the object file. In other words, the file access behavior for each object file in this phase is random multi-access read.

157
3. Read object files again and write output -

In last phase, 1d creates the output file. Then, it opens each object file, and random multi-access read each object file, like in the second phase. Usually, it writes randomly to the output file in 8 kb blocks usually after several reads. It may write portions of the output file more than once. In other words, the output file’s access behavior is random multi-access write. While in this phase, it also creates a temporary file. At the end, it writes 16 bytes to this file and reads all 16 bytes back almost immediately. Finally, it closes the output file, renames the output file, and deletes the temporary file.

In summary, building the Choices kernel involves many files and many program invocations. In general, the smaller files are text files and the larger files are object and archive files. Text files are sequentially accessed, and header files tend to accessed more than once. When object files are created initially, they are written sequentially. When they are linked, they are read randomly.

**SMS**

SMS is one of the programs in the Perfect Benchmarks. The Perfect Benchmarks are a set of scientific programs that have been chosen to evaluate the computational performance of supercomputers. SMS computes seismic migrations. It is written in Fortran. The SMS workload is traced on the Cray-2 (u2.ncsa.uiuc.edu) at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign[62].

<table>
<thead>
<tr>
<th>Filename</th>
<th>File Size</th>
<th>Bytes Accessed</th>
<th>Block Size</th>
<th>Access behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMI41</td>
<td>76551</td>
<td>76551</td>
<td>32768</td>
<td>sequential read, single access</td>
</tr>
<tr>
<td>SMV</td>
<td>305</td>
<td>305</td>
<td>305</td>
<td>sequential write, single access</td>
</tr>
<tr>
<td>SMT1</td>
<td>94330880</td>
<td>471654400</td>
<td>962560</td>
<td>random access, read and write, multi-access</td>
</tr>
<tr>
<td>SMT2</td>
<td>240504</td>
<td>481008</td>
<td>172032</td>
<td>sequential write, followed by sequential read</td>
</tr>
<tr>
<td>SMO6</td>
<td>1925</td>
<td>1925</td>
<td>1925</td>
<td>sequential write, single-access</td>
</tr>
</tbody>
</table>

**Table 9.9**: Files accessed by SMS.

SMS opens five files. Table 9.9 shows how SMS accesses these files. The block size column indicates the amount of data transferred by each read or write system call when sufficient data is available, i.e., when not at the end of file. SMT1 dominates the workload’s file accesses. It is the largest file and it accounts for 99.88% of the total amount of file data accessed. Figure 9.2 illustrates SMT1’s file access behavior. Figure 9.3 illustrates how SMS accesses its files.

**9.1.3 The Experimental Computing Environment**

The experimental computing environment consists of a pair of identically equipped Sun SPARCstation 2 workstations. Each workstation has a single Micropolis 4110 hard drive. All files reside on BSD
SMT1 File Access Behavior

Figure 9.2: SMT’s file access behavior.
open SMI41
open SMV
read 32768 bytes from SMI41
read 32768 bytes from SMI41
read 11015 bytes from SMI41
open SMT1
open SMT2
open SMO6
write 172032 bytes to SMT2
write 68472 bytes to SMT2
a lot of reads and writes on SMT1 ...
read 1702032 bytes from SMT2
a lot of reads and writes on SMT1 ...
read 68472 bytes from SMT2
a lot of reads and writes on SMT1 ...
write 1925 bytes to SMO6
close SMO6
write 305 bytes to SMV
close SMV
close SMI41
several writes to SMT1
close SMT1

Figure 9.3: SMS’s file access behavior.
<table>
<thead>
<tr>
<th>Strategy</th>
<th>Runtime (secs)</th>
<th>Cache miss indicator (4 kb pages)</th>
<th>Network load (Mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>without delay</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LRUCLK</td>
<td>251</td>
<td>7281</td>
<td>33.60</td>
</tr>
<tr>
<td>ZOMBIE</td>
<td>188</td>
<td>6787</td>
<td>31.93</td>
</tr>
<tr>
<td>SMART</td>
<td>182</td>
<td>6642</td>
<td>30.95</td>
</tr>
<tr>
<td>LCACHE 16 Mb</td>
<td>184</td>
<td>8582</td>
<td>36.81</td>
</tr>
<tr>
<td>ADAPTIVE 16 Mb</td>
<td>167</td>
<td>6589</td>
<td>27.92</td>
</tr>
<tr>
<td>LCACHE 32 Mb</td>
<td>168</td>
<td>7329</td>
<td>17.71</td>
</tr>
<tr>
<td>ADAPTIVE 32 Mb</td>
<td>160</td>
<td>6486</td>
<td>15.84</td>
</tr>
<tr>
<td>LCACHE 64 Mb</td>
<td>161</td>
<td>7512</td>
<td>14.85</td>
</tr>
<tr>
<td>ADAPTIVE 64 Mb</td>
<td>159</td>
<td>6471</td>
<td>14.85</td>
</tr>
</tbody>
</table>

Table 9.10: Building the *Choices* kernel with different caching strategies.

file systems created under SunOS 4.1.3. Each workstation has 32 megabytes of memory. Ethernet (10 megabits per second) connects the two workstations.

### 9.1.4 Summary

In summary, these experiments replay two different trace-driven workloads to evaluate several caching strategies.

### 9.2 Observations

This section presents the experimental data and key observations from each workload.

#### 9.2.1 Building the *Choices* kernel

Table 9.10 compares how five different caching strategies perform in building the *Choices* kernel. The first column of the table enumerates the caching strategies. LCACHE and ADAPTIVE employ secondary caches that store remote file data on the client’s local disk. In order to investigate the effects of different local disk cache sizes on LCACHE and ADAPTIVE, this workload is replayed three times with three different local disk cache sizes for each strategy. The suffix after LCACHE or ADAPTIVE indicates the maximum amount of local disk storage that secondary caches may use. For example, LCACHE 16 Mb indicates LCACHE with secondary caches that will consume up to 16 Mb of local disk space.

The second and third column of this table present two different runtimes. Runtime with delay is the amount of time required to complete the workload with time delays between file accesses included. These delays represent time spent on CPU computation between file accesses. In this workload, these time delays add up to approximately 450 seconds.

Runtime without delay is the amount of time required to complete the workload ignoring time delays between file accesses. It is the minimum runtime that can be achieved. It assumes that the CPU is
ininitely fast. It shows how caching strategies affect runtimes on faster CPUs. With faster CPUs, the workload becomes increasingly I/O bound. The SPARC CPUs in the SPARCstation 2’s used in these experiments are no longer the fastest. The latest DEC Alpha and UltraSPARC are already approximately five times faster. Hence, the total computational delay for these CPUs is only 90 seconds. However, the network bandwidth, disk bandwidth, and disk access time of the latest workstations equipped with these CPUs have only improved marginally.  

ZOMBIE versus LRUCLK

The first important observation is that ZOMBIE has much lower runtimes than LRUCLK. This workload consists of large number of small files, but only a small number of files are active at any one time (refer to Section 9.1.2). Hence, the least recently used units are usually found in oldest (least recently active) zombie. Therefore, ZOMBIE provides much better approximation of LRU than LRUCLK’s two-handed clock for this workload. This reduces cache misses by 6.8%. Since neither LRUCLK nor ZOMBIE employs secondary caches, reducing main memory cache misses also reduces the network load. In this case, ZOMBIE reduces network load by 13.9% (4.63 Mb).

However, ZOMBIE’s 6.8% reduction in cache misses and 13.9% reduction in network load do not account totally for the 25.1% reduction in runtime without delay. ZOMBIE’s improved I/O efficiency also contributes to the reduced runtime. Flushing a stale zombie instead of individual units preserves the spatial relationship between units in the cache (refer to Section 6.3.10). This reduces the number of write requests and improves storage layout efficiency. LRUCLK generates 8800 write I/O requests while ZOMBIE generates 6302 write I/O requests. ZOMBIE’s 28.4% reduction in write requests contributes significantly to the 25.1% runtime improvement over LRUCLK.

SMART versus ZOMBIE

In addition to ZOMBIE, SMART adds the read-ahead, write-behind, and free-behind data transfer strategies. SMART has marginally better runtimes than ZOMBIE. It reduces cache misses by 2.1% and network load by 3.5%. SMART does not perform significantly better than ZOMBIE because 92.3% of all files accessed are less than 32 kb (refer to Table 9.7). Since 32 kb is less than the cluster size for remote file objects, these data transfer strategies benefits only a small number of files (refer to Section 6.3.7.4). More precisely, read-ahead paged-in only 12.6% of all pages read, write-behind wrote only 6.8% of all pages written, and free-behind reclaimed approximately 10.1% of all pages reclaimed.

---

1 The standard network is still Ethernet. The Micropolis 4110 1 Gb disks used in this experiment are still one of the fastest available. Their access time of 8.5 milliseconds is still comparable with the access time (9 milliseconds) of the 1 Gb disks used in Sun’s latest SPARCstation 20.

2 This also indicates that LRU is appropriate as a main memory replacement policy for this workload.
<table>
<thead>
<tr>
<th>Strategy</th>
<th>Write-behind (4 kb pages)</th>
<th>Free-behind (4 kb pages)</th>
<th>VM requested I/O’s (4 kb pages)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCACHE 16 Mb</td>
<td>4686</td>
<td>783</td>
<td>588</td>
</tr>
<tr>
<td>ADAPTIVE 16 Mb</td>
<td>3880</td>
<td>4742</td>
<td>4885</td>
</tr>
</tbody>
</table>

Table 9.11: Differences between LCACHE 16 Mb and ADAPTIVE 16 Mb.

Free-behind is the only data transfer strategy that may reduce cache misses. This explains the modest reduction in cache misses.

SMART also reduces runtime without delay more than runtime with delay. This is because write-behind reduces memory reclamation latencies. On the other hand, computational delays between file accesses, present in runtime with delay, hide memory reclamation latencies.

**LCACHE 16 Mb versus non-local disk caching strategies**

Another important observation is that caching remote file data on a local disk does not always reduce network load and runtimes. This workload requires 64 Mb of local disk storage to cache all remote file data that will be accessed, 32 Mb of local disk storage to hold the working set. 16 Mb is insufficient.

LCACHE 16 Mb performs poorly. It requires more time to complete the workload than SMART. It also generates more network load than both SMART and ZOMBIE. This suggests that LCACHE manages a small 16 Mb local disk cache badly. LCACHE caches every remote file accessed in the client’s local disk cache. This is very inefficient because many of the remote files cached in the local disk cache will not be accessed again before they are flushed from the cache. Table 9.11 shows that only 5.44% of all data written to the local disk cache is accessed again. This indicates that LRU is not an appropriate cache replacement policy for the local disk cache.

LCACHE 16 Mb also generates the more main memory cache misses than any other caching strategy. This suggests poor main memory utilization. Two independent factors contribute to LCACHE 16 Mb’s poor main memory utilization. First, objects and data structures that maintain the state of the local disk cache consumes main memory. This reduces the amount of main memory available for caching file data. Second, transient data buffers increase memory contention. There are two main kinds of transient data buffers. The first kind holds unacknowledged RPC requests that may have to be retransmitted. The second kind provides a staging area for transferring data from the local disk cache to the server (refer to Section 6.2.5). Poor local disk cache management increases the need for transient data buffers. Transient data buffers increases memory contention resulting in less main memory available for caching file data. Therefore, poor local disk cache management increases cache misses.
In summary, local disk caches may adversely affect performance because they consume main memory to maintain their state and poorly managed local disk caches increases network load and main memory contention.

**ADAPTIVE 16 Mb versus LCACHE 16 Mb**

ADAPTIVE 16 Mb has significantly shorter runtimes, fewer cache miss ratios, and a lower network load than LCACHE 16 Mb. Its lower runtimes and fewer cache misses suggest ADAPTIVE 16 Mb has reduces latencies and improves main memory utilization. Its lower network load indicates better local disk cache utilization.

Table 9.11 shows that ADAPTIVE 16 Mb addresses the poor local disk cache management problem of LCACHE 16 Mb. In general, ADAPTIVE is more selective. Unlike LCACHE, ADAPTIVE caches only those files that are likely to accessed again in the future. For example, ADAPTIVE 16 Mb does not cache source files that are only accessed once by the compiler. As a result, ADAPTIVE 16 Mb reduces the amount of data written to the local disk cache by 52.3% when compared to LCACHE 16 Mb. More significantly, 95.0% of all data written to the local disk cache were accessed again. Because of better local disk cache utilization, ADAPTIVE 16 Mb reduces network load by 24.15% when compared to LCACHE 16 Mb. Its network load is also lower than without local disk caching, *i.e.* LRUCLK, ZOMBIE and SMART.

In general, ADAPTIVE ensures that local disk caching does not result in higher network loads than without local disk caching. ADAPTIVE becomes more selective with smaller local disk cache sizes.

ADAPTIVE 16 Mb also has much better runtimes than LCACHE and SMART. This is primarily because ADAPTIVE is more eager to flush from main memory data that is unlikely to be accessed again. This reduces cache misses and latencies. ADAPTIVE encourages eager flushing of data belonging to single-access files from main memory by minimizing these files’ *freeBehindLag* and lifetime (refer to Section 8.4.1). ADAPTIVE 16 Mb uses free-behind to flush six times more pages from main memory than LCACHE 16 Mb (refer to Table 9.11).

ADAPTIVE 16 Mb also writes less out of main memory via write-behind than LCACHE 16 Mb. This because more data is written out as part of free-behind operations. Free-behind is more likely to be activated to write dirty data than write-behind.

**ADAPTIVE 16 Mb versus SMART and ZOMBIE**

Unfortunately, ADAPTIVE 16 Mb has more cache misses than SMART and ZOMBIE. This is due to the increased main memory requirements of local disk caching. Fortunately, ADAPTIVE makes up for

---

3 Assuming appropriate file access behaviors have been detected earlier (refer to Section 8.4.3).
the additional cache misses by reducing network load and latency. As a result, ADAPTIVE 16 Mb still completes the workload in less time than SMART and ZOMBIE.

**LCACHE 32 Mb versus LCACHE 16 Mb**

LCACHE 32 Mb has lower runtimes and a lower network load than LCACHE 16 Mb. However, it still has a more cache misses than caching strategies that do not employ local disk caches, *i.e.* LRUCLK, ZOMBIE, and SMART. As mentioned earlier, this is due to increased main memory requirements of local disk caching.

LCACHE 32 Mb’s larger local disk cache is more effective. It reduces network load by 51.9% to 17.71 Mb which is much closer to optimal network load⁴ of 14.85 Mb. It also reduces cache misses since an effective local disk cache reduces memory contention from transient data buffers. As a result, LCACHE 32 Mb reduces cache misses significantly when compared to LCACHE 16 Mb. LCACHE 32 Mb’s lower network load and less cache misses allows LCACHE 32 Mb to complete the workload in less time than LCACHE 16 Mb.

**LCACHE 32 Mb versus ADAPTIVE 16 Mb**

Even though LCACHE 32 Mb has a significantly lower network load than ADAPTIVE 16 Mb, it does not complete the workload faster. This is because ADAPTIVE 16 Mb is better at managing main memory. ADAPTIVE 16 Mb has 7.1% fewer cache misses and is more aggressive in reclaiming memory from single-access files than LCACHE 32 Mb.

**ADAPTIVE 32 Mb versus LCACHE 32 Mb**

ADAPTIVE 32 Mb has lower runtimes, fewer cache misses, and a lower network load than LCACHE 32 Mb. Its network load of 15.84 Mb is only 1 Mb more than optimal. ADAPTIVE 32 Mb performs better than LCACHE 32 Mb since it manages main memory and local disk cache storage better than LCACHE 32 Mb.

**ADAPTIVE 32 Mb versus ADAPTIVE 16 Mb**

As expected, ADAPTIVE 32 Mb also performs better than ADAPTIVE 16 Mb on every performance criteria since it benefits from having a larger local disk cache. A larger disk cache reduces memory contention for transient data buffers. This indirectly reduces cache misses.

---

⁴This is the least amount of new or dirty data that must to transferred between the server and the client.
<table>
<thead>
<tr>
<th>Strategy</th>
<th>Run time (secs)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>without delay</td>
<td>with delay</td>
</tr>
<tr>
<td>SunOS NFS</td>
<td>366</td>
<td>803</td>
</tr>
<tr>
<td>SMART local</td>
<td>149</td>
<td>583</td>
</tr>
<tr>
<td>ADAPTIVE local</td>
<td>145</td>
<td>580</td>
</tr>
<tr>
<td>SunOS local</td>
<td>237</td>
<td>631</td>
</tr>
</tbody>
</table>

Table 9.12: Time needed to build the *Choices* kernel on SunOS and with local files.

**ADAPTIVE 64 Mb versus ADAPTIVE 32 Mb**

64 Mb of local disk storage is large enough cache every file accessed. Increasing the size of the local disk cache to 64 Mb from 32 Mb barely affects the performance of ADAPTIVE. In other words, ADAPTIVE 64 Mb and ADAPTIVE 32 Mb perform almost equally well. ADAPTIVE manages local disk caches so well that ADAPTIVE 64 Mb i only reduced the network load by 1 Mb. This modest reduction in network load results in very modest reductions in runtimes.

**LCACHE 64 Mb versus LCACHE 32 Mb**

On the other hand, 64 Mb of local disk storage benefits LCACHE more than ADAPTIVE. It results in a greater reduction in network load from LCACHE 32 Mb to LCACHE 64 Mb than from ADAPTIVE 32 Mb to ADAPTIVE 64 Mb. Unfortunately, LCACHE 64 Mb has more cache misses than LCACHE 32 Mb. LCACHE 64 Mb caches more files in the local disk cache than LCACHE 32 Mb. Hence, it also consumes more main memory to maintain state associated with each remote file cached on the local disk. Therefore, LCACHE 64 Mb has less main memory available for caching file data than LCACHE 32 Mb. This increases cache misses.

**Choices versus SunOS**

Table 9.12 presents SunOS runtimes and single-host runtimes for reference. *Choices* consistently outperform SunOS and SunOS NFS. This shows that the *Choices* distributed file system and its file system caching service are highly optimized. Except when LRUCLK is used, *Choices* completes this workload with remote file accesses in less time than SunOS with local file accesses.

Since *Choices* is highly optimized, runtime improvements resulting from fewer cache misses and a lower network load tend to be minimized as well. In particular, the performance impact of poor local disk cache management would be much more exaggerated if the local disk cache subsystem is less efficient. For example, the Andrew File System’s local disk cache subsystem is less efficient than the *Choices* secondary cache subsystem. It fetches and caches an entire file in its disk cache before making the data in the file available to the main memory file cache. *Choices* makes new file data available to the main memory file cache immediately after the data has been fetched from the file server. Unlike *Choices*, the
Andrew File System also cannot issue concurrent local disk and remote file server write requests. As a result of these deficiencies, the Andrew File System would experience a larger increase in runtime than Choices would when network load increases as a result of poor local disk cache management.

Summary

In summary, the key observations from this workload are:

1. Efficient I/O affects runtime more than cache misses. ZOMBIE improves I/O efficiency and provides a better approximation of LRU than LRUCLK.

2. A local disk cache consumes main memory to maintain the state of the local disk cache. It also allocates transient data buffers that consume main memory.

3. A poorly managed local disk cache adversely affects performance. For example, caching every file accessed in a local disk is inappropriate for a small local disk cache. LCACHE and the Andrew File System does this.

4. ADAPTIVE tries to cache, in the local disk cache, only files that are likely to be accessed again. In other words, it dynamically determines the set of files that will benefit from local disk caching according to the amount of local disk storage available for caching.

5. ADAPTIVE also manages main memory better than the other non-adaptive strategies. It eagerly frees main memory allocated to single-access files. This reduces cache misses.

6. ADAPTIVE’s better main memory utilization effectively offsets the increased main memory demands of local disk caching.

In conclusion, ADAPTIVE consistently performs better than the other non-adaptive caching strategies.

9.2.2 SMS

Table 9.13 compares how the various caching strategies perform in running the SMS workload. For this workload, two different local disk cache sizes will be evaluated. 128 Mb of local disk cache storage is sufficient to hold all data accessed, and 64 Mb is not sufficient.

ZOMBIE versus LRUCLK

ZOMBIE has a minimal impact on performance when compared with LRUCLK. This is because ZOMBIE is only efficient when there are zombies to reclaim. Unlike building the Choices kernel, SMS accesses very
<table>
<thead>
<tr>
<th>Strategy</th>
<th>Run time without delay (secs)</th>
<th>Cache miss indicator (4 kb pages)</th>
<th>Network load (Mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRUCLK</td>
<td>719</td>
<td>47624</td>
<td>453</td>
</tr>
<tr>
<td>ZOMBIE</td>
<td>716</td>
<td>47555</td>
<td>453</td>
</tr>
<tr>
<td>SMART</td>
<td>531</td>
<td>47570</td>
<td>453</td>
</tr>
<tr>
<td>LCACHE 64 Mb</td>
<td>487</td>
<td>47536</td>
<td>260</td>
</tr>
<tr>
<td>ADAPTIVE 64 Mb</td>
<td>442</td>
<td>47595</td>
<td>256</td>
</tr>
<tr>
<td>LCACHE 128 Mb</td>
<td>438</td>
<td>47562</td>
<td>155</td>
</tr>
<tr>
<td>ADAPTIVE 128 Mb</td>
<td>417</td>
<td>47550</td>
<td>155</td>
</tr>
</tbody>
</table>

**Table 9.13:** Replaying the SMS workload with different caching strategies.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Read-ahead (4 kb pages)</th>
<th>Write-behind (4 kb pages)</th>
<th>Free-behind (4 kb pages)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMART</td>
<td>24690</td>
<td>6081</td>
<td>64760</td>
</tr>
<tr>
<td>LCACHE 64 Mb</td>
<td>24696</td>
<td>6506</td>
<td>64760</td>
</tr>
<tr>
<td>ADAPTIVE 64 Mb</td>
<td>28246</td>
<td>6651</td>
<td>64760</td>
</tr>
<tr>
<td>LCACHE 128 Mb</td>
<td>24685</td>
<td>6107</td>
<td>64760</td>
</tr>
<tr>
<td>ADAPTIVE 128 Mb</td>
<td>28224</td>
<td>6436</td>
<td>64760</td>
</tr>
</tbody>
</table>

**Table 9.14:** The SMS workload and different caching strategies.

few files. A single 90 Mb file (SMT1) dominates the file accesses of this workload (refer to Section 9.1.2). Hence, there is very little opportunity for ZOMBIE to reclaim memory from stale files.

**SMART versus LRUCLK and ZOMBIE**

SMART improves runtimes significantly when compared with LRUCLK and ZOMBIE. This is because it is optimized for large sequential data transfers (refer to Section 6.3.7.4) and SMS reads and writes file data in large SMART’s read-ahead, write-behind, and free-behind data transfer strategies are most effective on large sequential data transfers such as the 940 kb transfers that dominate the SMS workload (refer to Figure 9.2). These strategies improve concurrency and reduce latencies. Since SMART has little impact on cache misses and network load, its significantly better runtimes are mostly the result of increased concurrency and reduced latencies achieved by these data transfer strategies.

**ADAPTIVE versus LCACHE**

When local disk caches are employed, LCACHE and ADAPTIVE have similar cache misses and network loads for each local disk cache size. However, ADAPTIVE consistently has lower runtimes than LCACHE. This is because ADAPTIVE adapts to the workload and is more aggressive in its optimizations. Table 9.14 shows that ADAPTIVE transfers more data in and out of memory via read-ahead and write-behind than LCACHE and SMART.
<table>
<thead>
<tr>
<th>Strategy</th>
<th>Run time without delay (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SunOS NFS</td>
<td>2007</td>
</tr>
<tr>
<td>SMART local</td>
<td>239</td>
</tr>
<tr>
<td>ADAPTIVE local</td>
<td>229</td>
</tr>
<tr>
<td>SunOS local</td>
<td>224</td>
</tr>
</tbody>
</table>

**Table 9.15**: Time needed to complete the SMS workload on SunOS and with local files.

ADAPTIVE 64 Mb also has a slightly lower network load than LCACHE 64 Mb. As discussed earlier, LCACHE always attempts to cache every file accessed in on local disk. ADAPTIVE is more selective. It correctly identifies SMT1 as the only file to cache. This reduces network load by 1.5%.

**Choices versus SunOS**

Table 9.15 presents the SunOS runtimes and single-host runtimes for reference. It shows that *Choices* completes the workload 2.8 to 4.8 times faster than SunOS NFS. This is because SunOS NFS limits each network read or write operation to 8 kb and limits the number of concurrent asynchronous requests to 8. On the other hand, *Choices* uses much larger packets (up to the 64 kb limit imposed by UDP). Larger packets reduce network latencies and improve network throughput. *Choices* also supports asynchronous operations through the use of continuations (refer to Section 7.5.1). This allows *Choices* to have more concurrency with little additional main memory overhead. *Choices*’ good performance shows that large network transfers and a high level of concurrency are crucial to distributed file system performance.

Unfortunately, *Choices* does not perform as well as SunOS in the single-host case. The primary performance bottleneck in the local case for this workload is the disk throughput. SunOS performs better than *Choices* because its low level device driver has better throughput than *Choices*’ disk driver when there are large I/O transfers.

In summary, the key observations from this workload are:

1. The read-ahead, write-behind, and free-behind data transfer strategies are important when there are large files and large data transfers. As a result, SMART has much better runtimes than ZOMBIE and LRUCLK.

2. ADAPTIVE correctly detects the need to cache the dominant file (SMT1), when there is insufficient local disk cache storage. This reduces network load.

3. ADAPTIVE adjusts caching parameters to suit the workload. In particular, it adjusts read-ahead and write-behind parameters. This improves runtimes.

4. To achieve good performance when large I/O requests are present, a distributed file system must be able generate large network transfers and support many concurrent asynchronous I/O requests.
In conclusion, ADAPTIVE consistently performs better than the other non-adaptive caching strategies.

9.3 Summary

The experimental results show that Choices’ adaptive file system caching strategy consistently outperforms the other non-adaptive file system caching strategies. The distinct file access characteristics of the different workloads demonstrates that Choices’ adaptive caching strategy can adapt to the different requirements of the two different workloads. For the Choices kernel build workload, the adaptive caching strategy improves local disk cache utilization, especially when the local disk cache is small. It also reduces cache misses by eagerly removing from main memory file data that are unlikely to be accessed again, i.e. data that belong to single-access files. For the SMS workload, the adaptive caching strategy aggressively tunes the the read-ahead and write-behind parameters to suit the file access characteristics of the workload. This reduces latency and improves runtime.

The experimental results also validate the performance benefits of the two other significant contributions of the Choices file system framework and distributed file system. One of these contributions is zombie reclamation. Zombie reclamation improves I/O efficiency and provides a better LRU approximation. The other significant contribution is the use of continuations to support asynchronous remote file accesses. A continuation-based distributed file system permit more concurrent asynchronous remote file accesses with less main memory overhead than a thread-based distributed file system.
Chapter 10

Conclusion

The goal of this thesis is to build a distributed file system with a flexible and adaptive caching service, and to demonstrate the benefits of such a distributed file system. This goal is met by defining and implementing an object-oriented framework that permits multiple caching strategies and supports multiple secondary caches. In addition, an adaptive caching strategy has been implemented in this framework. This adaptive caching strategy consistently outperformed non-adaptive caching strategies.

This adaptive caching strategy selects caching strategies according to the file access characteristics of each workload and the amount of local disk storage available for caching remote file data. The experiments described in the previous chapter show that the adaptive caching strategy adapted differently to two distinct workloads with very different file access characteristics.

The contribution of this adaptive caching strategy is most significant when the local disk cache is small. A small local disk cache can increase runtime, network load, and cache misses if poorly managed. In the case of Choices kernel build workload, LRU is inappropriate as the cache replacement policy for the local disk cache, and resulted in very poor local disk cache utilization. This increases cache misses and runtimes. On the other hand, the adaptive caching strategy correctly adapted to the workload and cached only those files that are likely to be accessed again before being flushed from the local disk cache. The adaptive caching strategy should always realize the primary expected benefit of local disk caching which is reduced network load.

This adaptive caching strategy also improves main memory utilization. It detects files that are only accessed once and eagerly reclaims main memory occupied by these files’ data. It’s better main memory management effectively offsets the main memory overhead introduced by local disk caching.

This adaptive caching strategy also reduces latency. It aggressively adjusts read-ahead, write-behind, and free-behind parameters of each file to suit the needs of each workload. This improves runtime.
Finally, comparisons with SunOS show that the flexible object-oriented file system caching framework described in this thesis does not adversely impact performance. Two features of this framework actually improve performance. The first feature is zombie reclamation. It improves I/O efficiency. The second feature is the continuation model for distributed file access. It improves concurrency for asynchronous remote file accesses.

In summary, flexible file system caching is practical and adaptive file system caching improves performance.
Bibliography


182
Vita

Swee Boon Lim was born in 1966. He completed his primary and secondary education in Singapore.

He started his undergraduate education at the George Washington University in 1983. He received his B.S. in Computer Science in 1987. He was also a research assistant from 1984 to 1987, where he developed a 4GL compiler, a SQL interpreter, and worked on an expert system inference engine.

He served his national service in the Singapore Army from 1987 to 1989.

He enrolled in the computer science graduate program at the University of Illinois in 1989. As one of Professor Michael Condry’s research assistant, he worked on high-speed networks and supercomputing applications. He earned his M.S. in 1991. He joined the Choices project in 1992 and worked as a research assistant to Professor Roy Campbell.

Upon completion of his doctoral studies, he accepted a position at SunSoft, Inc. in Mountain View, California in 1994, where he worked on an object-oriented distributed system.