# Write Caching in Distributed File Systems

A Thesis Submitted to the College of

Graduate Studies and Research

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in the

Department of Computational Science

University of Saskatchewan

Saskatoon, Saskatchewan

by

Kerhong Chen

June, 1994

## PERMISSION TO USE

# Write Caching in Distributed File Systems

*Candidate:* Kerhong Chen

*Supervisors:* Professor Rick Bunt and Professor Derek Eager

Master of Science Thesis

Presented to the College of Graduate Studies and Research

University of Saskatchewan

June, 1994

## ABSTRACT

*Disk caches have been employed at clients in distributed file systems to effectively avoid network accesses and at servers to compensate for the speed differential between main memory and disk. Typically, however, only read operations have benefitted from caching. The write policies used in most distributed file systems are based on write-through or periodic write-back approaches. There is no write caching at all in write-through, however, and data cached in ordinary memory using periodic write-back may be lost if system failures occur.*

*As the cost of non-volatile (NV) memories decreases, the uses of such memory for write caching, at the server or the client, or at both the server and the client, become increasingly attractive. Unlike ordinary (volatile) memory, the contents of NV memory are preserved even in the event of power failures. Thus, system reliability can be increased in comparison with those systems using periodic write-back with volatile memory. In comparison to write-through based systems, the response time for write operations can be greatly reduced, as data need only be written into the NV cache (at memory speed) rather than to the server and to disk, before control returns to the user program. Further, while data is resident in the NV cache, a significant fraction of it may be overwritten or deleted, thus reducing the amount of write traffic to the server.*

*In this thesis, five write caching policies for use in distributed systems with both volatile and non-volatile caches are investigated through trace-driven simulation experiments. The results indicate that policies focusing on temporal locality appear to perform well for a single-level caching system, or at the client of a multi-level caching system, while policies focusing on amortizing the cost of writing many dirty blocks over a single disk track seem to perform well at the server of a multi-level caching system. Locality characteristics in the reference stream seen at the server in a multi-level caching system may be destroyed by caching at the client with a NV cache large enough to hold the "working set of dirty blocks" for the client. A multi-level caching system using a wisely chosen combination of write caching policies seems to yield better overall performance than a single-level caching system managed by one write caching policy does, if both the single-level and multi-level caching systems have the same amount of NV memory.*

## ACKNOWLEDGEMENTS

I would like to thank my supervisors, Professor Rick Bunt and Professor Derek Eager, for their guidance and encouragement throughout my research work, and for the many hours they devoted to discussing and reading this thesis. Without their efforts, I would not have been able to complete my study in time. I would also like to thank the other members of my committee, Professor Carl McCrosky, and Professor Carey Williamson, and the external examiner – Professor Howard Hamilton, for their helpful suggestions both for proposal and for the final version of the thesis.

Many thanks to the faculty, staff, and fellow graduate students at the Department of Computational Science who were there to provide assistance and reassurance when I needed it. Thanks especially go to Doug Konkin and Tim Harrison and other DISCUS members for their help in my efforts to obtain traces and in solving the system problems.

My aunt Flora Fu-Hwa Chen supported me both emotionally and financially when I first came to Canada. If it was not for the help from her and her family, my study here could have never been started. My parents Ninghua Shen, Wensheng Chen, and my sister Kexin Chen, have always given me encouragement for each of my endeavors. Your love and care are always very important in my life.

Finally, and, most importantly, I would like to thank my husband, Xiaobo Li, for believing in me when I began to doubt. There were many hard times over the past several years, but through your aid, we made it through together.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

A computer system consists of three main subsystems: the central processing unit (CPU), memory, and the I/O system. The CPU provides control functions and performs desired arithmetic and logic operations. The memory system is an essential component in any digital computer since it is needed for storing programs and data. Information is stored in memory for current usage or for later reference. The memory system must have the capacity to read, for retrieving the information stored in memory, and the capacity to write, for storing and updating information in memory. A memory system includes main memory, which stores programs and data in current use by the CPU, and secondary memory, which provides long term storage. Memory systems can be managed by *virtual memory* method, which separates the memory address and the actual memory location. Logical memory addresses can be mapped onto the actual memory locations without letting the programmers to worry about where their programs are to be kept and about arranging for the transport of chunks of their programs between main memory and secondary memory. Commonly used secondary memory devices in computer systems are magnetic disks, magnetic tapes, and compact disks (CDs). The performance and reliability of secondary memory can be enhanced by logically grouping multiple magnetic disk drives into disk arrays. *Disk striping* is a method to map the logical disk addresses onto the actual disks in the array. The I/O system transfers information between the computer system and the external world, and between memory systems.

The I/O system presents special performance problems since there is a large disparity between secondary memory access time and CPU speeds. Over the last decade, CPU speeds have increased dramatically, but the performance of magnetic disks, still the most important secondary storage device, has not kept pace. Disk access times have improved only slowly. This trend is likely to continue in the future and this will cause more and more applications to become disk I/O bound [Rosenblum & Ousterhout 91].

Cache memory is an important component of the memory system. Information that is likely to be used in the future, but which normally resides on a slow storage medium, can be stored in caches on a faster medium to provide faster access. Two important types of caches are processor caches and disk caches. A *processor cache* is a small high-speed memory used to store instructions and operands fetched from main memory. A *disk cache* is a portion of main memory (typically) used to store data fetched from a disk. The unit of access and storage in the disk cache is the disk block (which is also the unit of access and storage on disk). There are basically two types of disk caches: the software cache, which is implemented in the operating system in main memory and the hardware cache, which is implemented by the disk controller. There are also hybrids that make use of dedicated hardware controlled through operating system modifications or extensions. The effective management of caches can help to eliminate "I/O bottlenecks" in which application performance is limited mostly by the slow speed of secondary memory unit access. This thesis focuses on disk cache management.

In a computer system, information on disk is often organized as files. A *file* is a named collection of related information defined by its creator, which exists from its explicit creation until its explicit destruction and is immune to temporary failures in the system. Files are managed by *file systems*, which are software packages that provide long-term storage and file services such as file management, file operations, and file protection to the users. A file in a file system is represented by an *i-node*. An i-node is a record that stores the information about a specific file on disk, such as the user and group identifiers of the file, the time of the last file modification and access, and pointers to the disk blocks containing the data contents of the file. The operations on files include, among others, create, write, read and delete. Files can be shared by multiple users who may, in a distributed system, be physically apart from each other and from where the file is stored.

A *distributed system* is a collection of machines interconnected by a communication network. Entities in a distributed system are often referred to as "clients" or "servers" where the terms "client", "server", and "service" are explained in [Levy & Silberschatz 90, Svobodova 84] as follows. A *client* is a process that can invoke a service using a set of operations, which forms the client interface. A *service* is a software entity running on one or more machines and providing a particular type of function to clients. An instance of the service software running on a single machine is referred to as a *server*. Often the machines on which clients and servers are running are themselves referred to as clients and servers, respectively.

A common example of a server in a distributed system is a *file server*. The machines on which client processes are running can be workstations or mainframes, and may or may not have their own local disks on which to store files. The file server manages a file system stored on one or more connected disks. An application running on a client machine waiting to access files managed by the server must access them remotely through the network. The access time for such files includes the network overhead as well as the CPU and disk service demands at the server. Figure 1.1 shows a client and a file server connected through a network, forming a simple distributed file system. There is a disk cache on both the client and the file server. The cache on the client is referred to here as the *client cache*, while the cache on the server is referred to as the *server cache*.

The goal of disk caching in a distributed file system is to reduce file access times as well as to reduce network traffic, server load, and disk I/O. By placing a cache on the client, data required by applications running on the client can be stored locally in the client cache after the data is fetched from the server. Repeated references to the same data can then be handled locally on the client without requiring additional access to the server (thereby providing faster access to the data and reducing the

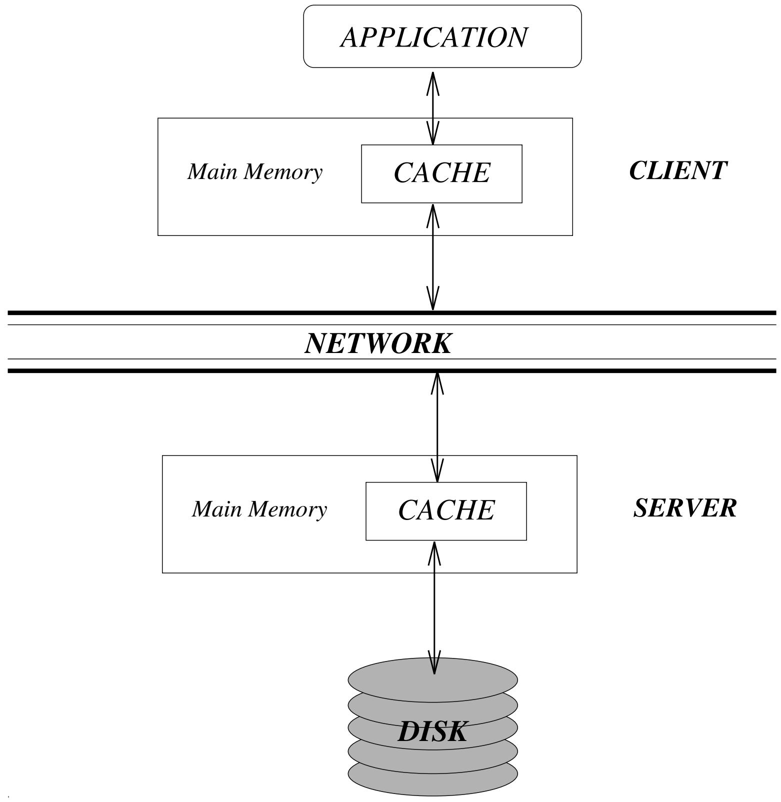



Figure 1.1: Caching in a Distributed File System

load on the server). Accesses to data not found in the client cache require at least a network request to the server, and if the data is not in the server cache, also a server disk access.

When an application running on the client generates a read to a disk block, the client cache is searched first. If the required disk block is present in the client cache, then this request can be satisfied locally. If the disk block is not present in the client cache, a request is sent to the server through the network. The server cache is then searched for the required block. If the disk block is found in the server cache, a copy of the block is sent to the client to satisfy the request. The disk block is both given to the application and placed in the client cache (in the expectation of re-reference). If the block is not found in the server cache, a disk access is performed to fetch the required block from the disk. The block is then sent through the network to the client. Typically the block will be placed in both the client cache and the server cache.

There are a variety of approaches to handling application writes (as result from modifications to existing data or creation of new data). For instance, *write-through* writes data through to the server's disk immediately, before returning control to the application making the write request [Silberschatz & Peterson 91]. *Write-back* initially records disk block modifications on the client cache only (resulting in the cache containing "dirty blocks"), and reflects these modifications later to the server [Silberschatz & Peterson 91].

Cache management is more complex in an environment such as that depicted in Figure 1.1, with caches at both the client and the server, than it is in a single cache environment. A cache management scheme in a distributed file system must address design decisions such as cache capacity, granularity of cached data, the location of the client's cache (in main memory or possibly on a local disk in the case of "whole file" caching), when and where to place data in cache, how to propagate modifications of cached copies, and how to determine if a client's cached data are consistent with other copies [Levy & Silberschatz 90] (for example, in the server's cache or in the caches of other clients). The basic idea of a cache management policy in such an environment is to keep in the caches the *working sets* of the users (the data referenced by the applications in a recent period of time), so that the number of disk accesses and the network traffic can be reduced, without endangering data reliability or consistency.

Cache management policies for a distributed system include policies of replacement, modification (write), and multi-cache consistency. The *replacement policy* decides which disk blocks should be replaced to make room for newly referenced disk blocks when the cache is full. When a cached copy of a disk block on a client is modified, the changes eventually need to be reflected on disk. How and when this is done is determined by the write policy. Maintaining consistency of multiple cached copies is the goal of a multi-cache consistency policies.

Caches have been shown to be effective in closing the gap between the speed of main memory and the speed of disk access and in reducing the number of disk accesses. Recent measurements of a distributed file system (Sprite [Nelson *et al.* 88]) have shown that about 60% of all data bytes read by applications are retrieved from client caches without contacting file servers [Baker *et al.* 91]. Typically, however, only the read operations have benefitted [Biswas *et al.* 93a]. Even though 80% of file operations are reads, writes are performance critical, because most reads are satisfied in cache, but writes are not, thus most of operations seen by server are writes (>75%). [Baker *et al.* 91] found that caches are still not very effective at reducing write traffic: Sprite's 30-second write-back policy only allows about 10% of newly written bytes to be deleted or overwritten without being written back from the client cache to the server. The measurement of file lifetimes showed that only 5 to 10% of bytes die within 30 seconds, while more than 80% die within half an hour. A 30-second write-back period is not long enough to ensure that a significantly large portion of cached data is overwritten or deleted while in the cache. Yet, increasing the length of this period increases the probability of data loss owing to system failure while dirty data is cached in main memory. As disk caches on both clients and servers continue to grow in size and satisfy an even larger fraction of the read traffic, the proportion of write traffic to the server disk will increase and become an increasingly important determinant of performance.

To ensure data integrity, newly-written data must be written from volatile cache memories to non-volatile storage as soon as possible. On the other hand, the longer dirty data is held in the cache, the more likely the dirty data is to be changed (overwritten) or deleted, thus eliminating the need to write it to disk and reducing network and disk traffic. New approaches allowing both of these conflicting goals to be met simultaneously, such as log-structured file systems [Rosenblum & Ousterhout 91, Burrows *et al.* 92], and the use of non-volatile cache memories for write caching [Baker *et al.* 92, Moran *et al.* 90, Biswas *et al.* 93a, Biswas *et al.* 93b] and write buffering [Ruemmler & Wilkes 93a], can be used to improve the performance of write operations. When a non-volatile RAM (NVRAM), such as RAM with battery backup, is used as a write cache or buffer, dirty data can be stored in NVRAM so that the system data integrity can be guaranteed without the cost of frequently transferring dirty data from the cache to disk. It has been shown that employing non-volatile write caches on clients and/or servers in a distributed system, using some write-back strategy, together with volatile read caches, can greatly reduce disk I/O traffic (up to 85%) [Baker *et al.* 92, Biswas *et al.* 93a, Biswas *et al.* 93b].

The goal of this thesis is to study performance issues relating to caching in distributed file systems, specifically when a cache system that consists of both volatile and non-volatile caches is used. Of particular interest are the policies for dealing with replacements and writes in this type of environment. Trace-driven simulation is employed to study the relative performance of various candidate policies, and the impact on performance of experimental parameters.

The remainder of this thesis is organized as follows. Chapter 2 briefly surveys several distributed file systems, specifically Sun NFS, Sprite, Andrew, Coda, and Echo. Caching in these systems is described, including policies for replacement, write, and multi-cache consistency.

Chapter 3 surveys research on write caching and buffering using non-volatile memory, and discusses write caching policy issues that arise in these previous studies.

Chapter 4 outlines the experimental design and methodology of this thesis. The detailed motivations and objectives of the thesis are discussed first, followed by the description of specific policies of interest for write caching in a system with both volatile and non-volatile caches, the description of the caching models, and the description of experimental metrics. This chapter also discusses the events recorded in the collected traces of disk block referencing activity, the system traced, the trace collection and processing, and the simulation methodology.

Chapter 5 examines the performance effects of NV caching in both the single-level caching system (a standalone workstation with a local disk), and the multi-level caching system (a client machine connected with a server machine through network with a disk connected to the server). It is observed that policies focusing on temporal locality appear to perform well in a single-level caching system, or at the client in a multi-level caching system, while policies focusing on amortizing the cost of writing many dirty blocks over a single disk track seem to perform well at the server of a multi-level caching system. In a multi-level caching system, locality characteristics seen at the server in the reference stream may be destroyed by NV caching at the client with a NV cache large enough to hold the "working set of dirty blocks" (the dirty data referenced by the application writes in a recent period of time) for the client. A multi-level caching system using a wisely chosen combination of write caching policies seems to yield better overall performance than a single-level caching system managed by one write caching policy does, if both the single-level and multi-level caching systems have the same amount of NV memory.

Chapter 6 contains a summary of the thesis, the conclusions drawn from the simulation experiments, and suggestions for future areas of research.

# Chapter 2

# Caching in Distributed File Systems

As described earlier, disk cache memories are used to hold file data from disk that is likely to be referred to in the near future, so as to overcome the access speed disparity between CPU and disk access.

This chapter gives a more detailed description of disk caching in distributed systems. Section 2.1 gives a brief survey of some existing distributed file systems, specifically Sun NFS, Sprite, Andrew, Coda, and Echo. Section 2.2 discusses the main considerations in disk cache design in these systems, including the size of the cache, the size of disk blocks, policies for replacement and modification operations, and policies for maintaining multiple cache consistency.

## 2.1 Existing Distributed File Systems

A number of distributed file systems have emerged over the last decade. This section surveys Sun NFS, Sprite, Andrew, Coda, and Echo, in Sections 2.1.1 to 2.1.5, respectively.

### 2.1.1 Sun Network File System

The Network File System (NFS) was first developed by SUN Microsystems for use in UNIX operating system environments [Sandberg *et al.* 85, Kleiman 86, Juszczak 89, Moran *et al.* 90]. NFS is a facility for sharing files in a heterogeneous environment of machines, operating systems, and networks. File sharing is accomplished by mounting a remote file system, then reading and writing files in place.

The NFS file system interface includes two parts [Sandberg *et al.* 85, Rosenthal 90]: the Virtual File System (VFS) interface defines the operations that can be done on a file system, while the v-node interface defines the operations that can be done on a file within a file system. NFS allows a variety of machines and operating systems to play the role of client and server.

NFS file systems are "stateless", which means that any state information about the file system is kept on disk and any state information about current client applications using the file service is kept on the relevant client, so that any client or server can crash and recover without causing any application on any other machine in the network to fail. A simple stateless protocol is used in NFS [Moran *et al.* 90], which requires that each client request be complete and self-contained, and that the server completely processes each request before sending an acknowledgment back to the client. If the server crashes, or an acknowledgment is lost, the client will retransmit its request to the server.

### 2.1.2 Sprite File System

The Sprite file system [Nelson *et al.* 88, Baker *et al.* 91, Levy & Silberschatz 90] was developed at the University of California at Berkeley. Sprite provides a single-system image: there is a single shared file hierarchy, and files in this shared hierarchy are equally accessible by processes on any workstation in the network.

A recent version of Sprite employs a logging scheme on the file server to improve write I/O performance [Baker *et al.* 92, Rosenblum & Ousterhout 91]. By buffering a sequence of file system changes in the file cache and then writing all the changes to disk sequentially in a log-like structure through a single disk write operation, both file writing and crash recovery can be sped up dramatically [Rosenblum & Ousterhout 91]. The log is the only structure on disk, which contains indexing information so that files can be read back from the log efficiently.

### 2.1.3 Andrew File System

The Andrew File System (AFS) [Howard *et al.* 88, Levy & Silberschatz 90, Blumson 92] is a location-transparent distributed file system that can include up to 10000 workstations on a single network. AFS has been in development at Carnegie Mellon University since 1983. Andrew presents a homogeneous, location-transparent file name space to all the client workstations by using a set of trusted servers collectively called *Vice*. A variety of operating systems can run on AFS servers, including 4.3BSD UNIX, MVS, and AIX 3.0 [Subramanian 92]. The operating system running on each workstation intercepts file system calls and forwards them to a user-level process called *Venus*, which performs caching and communication with file servers.

Venus is responsible for maintaining two client caches. The "data cache" contains files as well as the contents of directories, while the "status cache" contains status information about the files including information on file size and modification timestamps. The status cache is kept in virtual memory to allow rapid servicing of system calls. The data cache is resident on the local disk of each client so as to allow caching of an essentially arbitrary number of files with arbitrary sizes.

Andrew caches entire files in client caches so as to reduce the network traffic and to keep the cache management policies simple, and easily scalable. However, local disks are needed by the clients for this caching, which increases the system cost. Additionally, when only small portions of a large file are needed, transferring the whole file to the client, as must be done in Andrew, is wasteful of system resources.

### 2.1.4 Coda File System

The Coda file system is a descendant of the Andrew file system that was designed to be more resilient to failures [Kistler & Satyanarayanan 92, Satyanarayanan *et al.* 90a, Satyanarayanan 90b]. Coda strives for constant data availability, which allows a user to continue working regardless of failures elsewhere in the system. High availability is achieved [Kistler & Satyanarayanan 92, Satyanarayanan *et al.* 90a] by using *server replication* and *disconnected operation.* Server replication allows files to have read-write replicas at more than one server, which reduces the probability of an object becoming unavailable due to failures. Disconnected operation, which is a mode of execution in which a caching site temporarily assumes the role of a replication site, takes effect when no server is accessible. In other words, while a client is disconnected from the server, Venus services file requests by relying solely on the contents of its cache.

As a descendant of Andrew, Coda also maintains two client caches (the status cache in memory and the data cache on local disk), and caches entire files in the client data caches.

### 2.1.5 Echo File System

Echo, a distributed file system designed and developed at DEC SRC [Hisgen *et al.* 89, Siegel 92, Mann *et al.* 93, Birrell *et al.* 93], incorporates replication, caching, global naming, and distributed security. Echo replicates servers for availability and disks for data integrity, allowing a wide range of configurations and tolerating both server crashes and network faults. Data and properties for both files and directories are cached at Echo clients so that users see good performance not only when reading file data, but also when opening files and listing directories.

Echo supports a global, scalable, hierarchical name space. Echo employed two different replication techniques for availability: one at the upper levels of the hierarchical name space (the name service) and another at the lower levels of the name space (the file volume service).

In the security model of Echo, servers do not trust client machines. Instead, client machines use a cryptographic protocol to authenticate themselves as acting on behalf of the users who have logged in.

## 2.2 Cache Design Aspects

The reason why any form of caching can be successful is the fact that *locality* exists in reference patterns [Smith 85]. The principle of locality describes the reference behaviour of most programs. Locality of two forms has been identified. Information that has been used recently is likely to be reused soon (referred to as *temporal locality*) and information stored close to the information in current use is likely to be used in the near future (referred to as *spatial locality*).

A disk cache is beneficial if it improves the appropriate system performance measure (such as individual response times or system throughput) for a given system to an extent that justifies its cost. It is also possible to evaluate the effectiveness of disk caching by the change in the mean disk access time [Smith 85]. Two specific metrics that are commonly used to evaluate the effectiveness of any caching policy are *hit ratio* and *miss ratio*. The hit ratio is defined as the fraction of total references that are satisfied by searching the cache. The miss ratio, on the other hand, is defined as the fraction of total references that are not satisfied by the cache, resulting in further search. Notice that the sum of the hit ratio and the miss ratio is 1 by definition.

The design of a disk cache facility, whether volatile or non-volatile, must include consideration of the following aspects:

1. cache capacity,
2. size of cache entries,
3. cache location,
4. cache management policies
   (a) cache replacement policies,

(b) cache write policies, and

(c) multi-cache consistency.

The rest of this section discusses these disk cache design issues, using the previously surveyed distributed file systems as examples.

### 2.2.1 Cache Capacity

The required size of disk cache to achieve a given miss ratio is the most basic aspect of disk cache design. Previous studies [Smith 82, Makaroff 88, Baker *et al.* 92, Biswas *et al.* 93a] have shown that significant performance benefit can be obtained from a relatively small disk cache. Increasing the cache size further will result in a non-decreasing miss ratio curve, but eventually the cache will be large enough to contain all the active data, and the miss ratio will asymptotically approach a limiting value: the fraction of references to data that has never been previously referenced. Previous studies [Smith 85] suggest that the miss ratio curve has the form $m = a(c)^b$, where $a$ and $b$ are application dependent constants, $m$ is the miss ratio, and $c$ is the cache memory capacity. Figure 2.1 shows a typical miss ratio curve. It can be seen that as the cache size increases, the miss ratio decreases rapidly until a certain cache size is reached (at the "knee" of the cache miss ratio curve). Further increasing of the cache size decreases the miss ratio very slowly, exhibiting the phenomenon of "diminishing returns".

In a distributed file system with disk caches at both client and server machines, caching has its own unique characteristics. As indicated in [Makaroff 88], although increasing the client cache size decreases the client cache miss ratio, the server cache miss ratio may be increased. In fact, increasing the client cache size may increase the overall miss ratio (fraction of disk block references that miss in both client and server caches, and must therefore be satisfied by accessing a file server disk). Further, for small server caches, the server cache miss ratio does not drop significantly with increasing server cache size until the server cache size approaches the client cache size. Different costs and benefits may be achieved by placing a given amount of cache

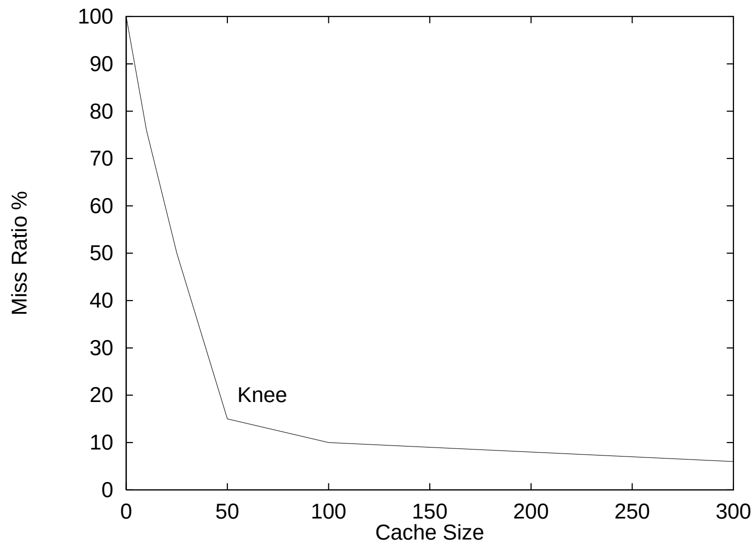



Figure 2.1: A Typical Disk Cache Cost/Performance Curve [Makaroff 88]

capacity at the workstation clients and at the server.

The remainder of this section considers how the cache size issue is handled in the example distributed file systems. This issue is handled most simply in Sun NFS, where main memory disk caches with fixed sizes are implemented [Moran *et al.* 90] at both the clients and the servers.

In the Sprite file system, the client cache changes size dynamically in response to the needs of the file and virtual memory systems [Nelson *et al.* 88, Baker *et al.* 92]. The division of main memory between that used to support virtual memory and that used for a disk cache is adjusted according to the changing usage patterns of memory. When a block of file data must be brought into the cache or a new virtual memory page is required by the virtual memory manager, the oldest page in either the client cache or main memory is chosen for replacement. In order to reduce the interference between the client cache and main memory, the client cache is always bypassed for virtual memory operations such as process swapping and paging. Substantial cache size variations over time were observed on clients that had an average cache size of about 7 megabytes out of an average of 24 megabytes of main memory [Baker *et al.* 91].

The Andrew and Coda file systems use local disks for file caching on clients

[Howard *et al.* 88, Satyanarayanan *et al.* 90a, Satyanarayanan 90b]. Status caches in Andrew and Coda reside in virtual memory. It has been claimed that a local data disk cache capacity of 100 megabytes is adequate for Coda in terms of permitting useful disconnected operation [Kistler & Satyanarayanan 92].

In Echo, most of the workstations in the system dedicate more than 16 megabytes of main memory to the cache [Mann *et al.* 93, Birrell *et al.* 93].

### 2.2.2 Size of Cache Entries

The granularity of the cached data is another important issue of cache design. One basic choice is between a whole file cache (as used in Andrew and Coda) and a disk cache that caches individual disk blocks (as used in the other example distributed file systems). The size of a block is typically some multiple of the smallest unit of information transferable from a disk [Levy & Silberschatz 90]. A disk is made up of one or more *platters* of magnetic media. There are a certain number of concentric *tracks* on each platter. The tracks at the same position on all platters together comprise a *cylinder*. A track is further divided into a number of individually addressable sectors. The block size is typically some small multiple of the sector size. One block may therefore contain information required for several individual application I/O operations (at least to the extent that spatial locality is present).

There are a number of trade-offs with respect to the choice of block size, and no single criterion dominates [Smith 82, McKusick *et al.* 84]. Increasing the block size increases the likelihood that data for the next spatially consecutive access will be found in the same block (and thus in the cache). On the other hand, the time required to read a block from disk and the potential for consistency problems are increased for larger block sizes [Levy & Silberschatz 90]. In general, with small caches, small blocks are most appropriate. Small blocks allow several pieces of distinct information to be in the cache at once, and small cache blocks are unlikely to contain unneeded information. For larger caches, however, better performance is obtained with a larger block size. More potentially relevant information can be brought into the cache by one fetch [Smith 85].

Determining the block size in distributed file systems involves parameters such as the size of the network transfer unit, which is relatively small. For example, the size of data field of an Ethernet frame is 0 for the minimum and 1500 bytes for the maximum, a valid Ethernet frame can be 64 bytes [Tanenbaum 89]. The Remote Procedure Call (RPC) protocol service unit should also be considered if an RPC protocol is used [Levy & Silberschatz 90]. In general, however, large blocks yield reduced network overheads.

Block sizes of 4 or 8 kilobytes are common [Levy & Silberschatz 90] for UNIX-like systems. HP-UX, for example, uses an 8 kilobytes block [Ruemmler & Wilkes 93b].

### 2.2.3 Cache Location

In general, a disk cache can be placed in any convenient location along the data path between the CPU and the disk surface [Smith 85]. Desired miss ratio, maintenance of cache consistency, and cost are three important issues affecting the choice of location for a disk cache. The most common location of a disk cache is in main memory. Other possible locations are the I/O channels or the disk controllers, or for a whole file cache in a distributed file system, on a local disk [Smith 85].

Main memories on client workstations are now large enough to accommodate large caches with low miss ratios; these main memories will grow to achieve even lower miss ratios in the future [Nelson *et al.* 88, Baker *et al.* 92, Biswas *et al.* 93a, Ruemmler & Wilkes 93a, Ruemmler & Wilkes 93b]. Only a single caching mechanism is needed for the use of both servers and clients if a main memory client cache is used, since the server caches are typically in main memory regardless of where client caches are located [Smith 85]. On the other hand, placing a cache on a local disk makes it easier to ensure data integrity. Modifications to cached data can be recovered after a crash if the data are kept on disk. Main memory caches allow greatly reduced access times, whereas caches on local disks allow whole file storage, and increased reliability and autonomy of single machines.

In many modern distributed file systems, recently used file data are cached in main memory, and large main memories make large file caches possible [Rosenblum & Silberschatz 90]. The Sun NFS and Sprite file systems use large disk caches in the main memories of both clients and servers [Levy & Silberschatz 90]. In addition, some Sun NFS systems use non-volatile memories as write caches on the servers to achieve high system reliability and better performance [Moran *et al.* 90]. Disk-based file caches are used in both the Andrew and Coda file systems [Nelson *et al.* 88, Kistler & Satyanarayanan 92]. Main memory is used for caching in Echo [Birrell *et al.* 93].

### 2.2.4 Replacement Policies

When a new block must be inserted into a full disk cache, one block must be removed. The policy that decides which block to replace is the *replacement policy.* The main objective of a replacement policy is to retain in the cache the blocks likely to be referenced in the near future and to discard from the cache the blocks that are no longer useful or whose next access is in the most distant future. Many replacement policies have been described in the literature, including first-in first-out (FIFO), least-recently-used (LRU), least-frequently-used (LFU), frequency-based replacement (FBR) [Robinson & Devarakonda 90], and knowledge-based replacement (KBR) [Korner 90]. The optimum algorithm (OPT) provides an upper bound on the performance of possible replacement policies, while random replacement (RAND) provides a lower bound on the performance of reasonable replacement policies.

Most existing disk cache management systems use LRU or some variant of LRU as the cache replacement policy [Stone 90]. LRU replaces the block in the cache that has not been referenced for the longest period of time. A fixed size "stack" is maintained to determine which of the blocks in the cache has been the least-recently used. On a cache hit, the referenced block is moved from its current position to the top of the stack, and the positions of all other blocks in the cache are adjusted accordingly. On a cache miss, the referenced disk block is added to the cache at the top of the stack, after removing the block at the bottom of the stack (thus, least-recently used) to make room if necessary. LRU performs well if temporal locality exists in the disk block access patterns of applications.

LFU keeps a counter of the number of references that have been made to each cached disk block. A block with the smallest count is subject to replacement when the cache is full. One problem with the use of reference counts for replacement purposes is that certain blocks may build up high reference counts and never be replaced. A solution to this problem is to introduce a maximum average value of reference count (over all blocks currently in the cache) as a parameter of this policy, and to control the average reference count dynamically. Whenever the average count exceeds the predefined maximum value, every reference count is reduced by half of its value. This scheme is often called reference count "aging".

FBR is a hybrid replacement policy that attempts to capture the benefits of both LRU and LFU without the associated drawbacks [Robinson & Devarakonda 90]. FBR maintains an LRU stack that is divided into three sections: the "new" section (consisting of the top part of the stack), the "old" section (consisting of the bottom part of the stack), and the "middle" section (consisting of the remainder of the stack), as shown in Figure 2.2. The percentage of the cache size contained in the new section ($F_{new}$) and in the old section ($F_{old}$) are parameters of the algorithm.

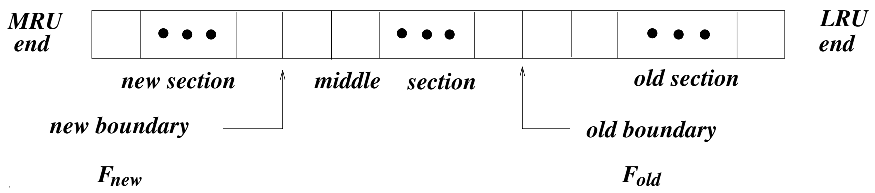



Figure 2.2: Three Sections of the FBR Cache

FBR uses the reference count scheme of the LFU algorithm (with aging), but does not increase the frequency count on a cache hit unless the referenced block is outside the new section. Further, replacements are made only from the old section. If there is only one disk block in the old section with the smallest frequency count, that block is selected to be replaced when the cache is full. If there is more than one block with the smallest frequency count, the LRU block is replaced. The new section is used to explicitly "factor out" the temporal locality that is the primary reason for the failure of frequency-based algorithms. The old section is used to "confine replacement" since blocks in the new and middle sections have not had enough time to build up their frequency counts. Note that by choosing parameters appropriately FBR can become LFU (when $F_{new} = 0$ and $F_{old} = 100$) or LRU (when $F_{old}$=1/(cache size)).

In KBR [Korner 90], disk block references are grouped into categories such as i-node references, directory references, references to executable blocks, and references to sequentially or randomly accessed file blocks. The disk cache is flexibly partitioned into a system partition and per process partitions. KBR enhances the performance of ordinary replacement policies in three areas. First, different cache management policies are used within each per process partition based on anticipated access patterns, according to the types of disk block references being made. Second, cache preloading/reloading is employed by identifying information of general utility to all processes, and preloading or reloading it during idle server periods. Finally, background read-ahead is used whenever sequential access is anticipated.

A replacement policy can be evaluated by comparing it to an optimum policy that has perfect knowledge of the future pattern of disk block references. The OPT policy provides the lowest miss ratio of all replacement algorithms, by replacing that block in the cache whose next reference is the farthest in the future, whenever a replacement must be made. It is impossible to implement OPT in practice, since it requires perfect knowledge of future disk references, although it can be simulated given a trace of disk block references.

A lower bound on the performance of reasonable replacement policies is provided by the performance of the RAND algorithm. Since RAND replaces each cache block in the cache with equal probability, no advantage is taken of locality characteristics. If a replacement policy cannot perform better than RAND in practice, then there is almost certainly some underlying flaw in the decision strategy [Willick *et al.* 93].

In a distributed system with disk caches at both clients and servers, there is a a multi-level cache hierarchy [Willick *et al.* 93]. File I/O requests generated by applications running on the clients may be satisfied at the local client cache, or if not, then possibly at the file server cache. System performance can be improved by these caches, both because they reduce the frequency at which costly disk operations are performed, and because the client cache reduces the frequency of requests from the client to the file server. Caching policies are important on both the clients and the server, although client caches are now large enough to hold much of the "working set" of a typical user [Ousterhout *et al.* 85].

Server cache performance is significantly different from client cache performance because of the capture of disk block references by client caches [Makaroff 88, Willick *et al.* 93]. The caches on the clients see all the disk block references generated at the clients, whereas the caches on the file server see only the references that are not satisfied by the client caches. Much of the temporal locality is removed from the "filtered" reference streams seen by the file server, and therefore, the performance of LRU on a server cache can degrade dramatically with even a small client cache size [Willick *et al.* 93]. LFU has been found to outperform LRU on the server in many cases [Willick *et al.* 93].

Sun NFS, Sprite, and Echo use the LRU replacement policy for both client and server caches. Andrew and Coda also use LRU as a replacement policy for both the status and the data caches on the clients [Howard *et al.* 88, Kistler & Satyanarayanan 92].

### 2.2.5 Write Policies

Caches are intended only for temporary storage, and newly created or modified data must eventually be written to disk for permanent storage.

There are two basic approaches to updating disk storage: *write-through* and *write-back* [Smith 82, Biswas *et al.* 93b, Jouppi 91]. With write-through, control does not return to an application program following a write operation until the data has been written to disk. The advantage of this policy in a distributed system is its reliability:

if a write operation completes successfully, the application program can assume that the data is safely stored on disk, regardless of any future client or server crashes. Furthermore, in a multi-client context, consistency considerations may be simplified as the disk always contains the most up-to-date information. Write-through has two disadvantages, however. First, it limits the performance benefits of client caching, since a server disk access is done for every write, and therefore only read operations can be satisfied locally by the client cache [Srinivasan 89, Biswas *et al.* 93a]. Second, an application is blocked while waiting for written data to make its way over the network, through the server queues, and onto the disk. Writes are synchronous with slow network and disk operations.

A write-back caching policy, on the other hand, stores the written data in the cache, acknowledges completion of the operation to the application, and only later performs the write to disk. In a variant of write-back termed *periodic write-back*, the cache is checked periodically and dirty blocks residing in the cache for a period longer than a certain interval are written to disk. The advantages of write-back are two-fold. First, write operations can be acknowledged more quickly, allowing applications to proceed immediately after the data is written to the cache. Second, data may be deleted or overwritten before being written back to disk, thereby reducing the disk traffic, as well as the network traffic and server load in a distributed system. The disadvantage of the write-back policy is that data not written to the disk is susceptible to loss if the cache is implemented in ordinary (volatile) memory. Maintaining multi-cache consistency is also more difficult than when write-through is used, particularly if copies of a datum may reside in multiple client caches.

Sun NFS servers are required to synchronously commit any modified data to stable storage before acknowledging completion of the operation to the application, so as to ensure the reliability of the system [Moran *et al.* 90, Juszczak 89]. For each write request, at least one, and possibly two or three, synchronous disk operations must be performed by the server before a response can be sent to the client indicating that the request has been completed. If the write increases the size of the file, or on-disk block structures have changed, or an indirect block was modified, then the blocks containing the i-node and the indirect block being modified must be written to disk before responding. It was found that over 60% of the requests for disk I/O are due to these modifications [Moran *et al.* 90]. Several methods have been employed by different vendors to achieve better NFS write performance. For instance, non-volatile memory has been used at servers in NFS for reliable write caching, the kernel NFS codes have been modified so as to avoid synchronous updates, and disk striping and disk arrays have been employed.

Sprite systems use a periodic write-back policy at both clients and servers, in which writes are cached but are held in the cache for no longer than 60 seconds before being purged [Nelson *et al.* 88] (where *purge* denotes the action of committing dirty blocks to disk). Specifically, every 30 seconds the cache is checked and those dirty blocks that have not been modified in the last 30 seconds are written back to the disk or to the server. A block written on a client will be written to the server's cache in 30-60 seconds and will be written to disk in 30-60 more seconds.

Andrew uses a variant of write-back termed *write-on-close* [Howard *et al.* 88], in which modifications to a file cached locally at a client are not reflected to the server until the file is closed, at which time the modified blocks are copied back to the server. Thus, a client's changes to a file are visible in the rest of the distributed system only after the file has been closed and the changes have been copied back to the server. Other file status and directory modifications such as protection changes are visible immediately. As a descendant of Andrew, Coda also uses a write-on-close policy [Satyanarayanan *et al.* 90a, Satyanarayanan 90b].

The Echo file system employs a variant of write-back termed *ordered write-behind*, which provides stronger reliability guarantees than ordinary write-back [Mann *et al.* 93, Birrell *et al.* 93]. Ordered write-behind is quite similar to the periodic write-back strategy used in Sprite, but it has been extended to deal with directories and to cope with replicated servers [Birrell *et al.* 93]. With this policy, updates buffered in the cache are automatically flushed after a time delay as in periodic write-back, but in such a manner that writes to any particular file or directory are usually guaranteed to be written to server disks in the same order that they were performed by the application. The only difference is that within a sequence of writes to a single file that do not alter the length of the file, the Echo client and server are allowed to re-order the write operations arbitrarily to obtain better disk access performance.

### 2.2.6 Multi-Cache Consistency

In a computer system with disk caches, it is possible for a piece of information to have multiple copies – one copy on the disk (the master copy) and one copy in each cache connected to it. Under the strongest notion of "consistency", there must be a unique value for every piece of information at any point of time, and any attempt to read that piece of information should yield the uniquely correct value at that time.

In a distributed system, files can be read shared or write shared when more than one clients read or write to the same file. Read sharing does not cause any problems to the system, because data are not changed when being read. Write sharing, on the other hand, may cause consistency problems in the system. There are two basic situations wherein files become write shared. When a file is modified by one client, closed, and then opened by some other client, then the file is said to be *sequential write shared*. When a file is opened on multiple clients, and at least one of them has it open for writing, then the file is said to be *concurrent write shared*.

A general scheme that may be used to maintain consistency under both sequential and concurrent write sharing is based on the use of version numbers [Levy & Silnerschatz 90]. The version number of a file is increased whenever a file is opened for a write operation. When a client opens a file, the client obtains the file's version number from the server and compares this number to the version number associated with any cached blocks for that file. If the version numbers are different, the client discards all cached blocks for the file and reloads its cache from the server when the blocks are needed. If the write-back policy is used, the server does not always have the current file data. In this case, it is necessary for the server to keep track of the last writer for each file. When a client other than the last writer opens the file, the server sends a request to the last writer to flush all dirty blocks of this file. The open operation blocks until the last writer has flushed all dirty blocks and the operations have been handled at the server. The server then responds to the client with the current version number of the file. When a server detects that a file is concurrent write shared, it disables client caching for that file. Caching is disabled on a file basis, and the disablement affects only clients with open files. A non-cachable file becomes cachable again when the file has been closed on all clients. All subsequent reads and writes go through the server when file caching is disabled, which serializes the accesses to the file. Notice, however, that a file may be cached simultaneously by several active readers, as long as there is no concurrent writer.

In Sun NFS, as mentioned before, file modifications are written directly through to the disk before write requests are acknowledged. The server always has an up-to-date copy of file system data. At each client, there is both a disk block cache for file data, and a "file attribute" cache in which i-node information is stored. When a file is opened, the kernel obtains the time-stamp of the last modification from the server and determines whether to fetch or revalidate the cached file attributes. The cached file blocks are used only if the corresponding cached attributes are up to date. The file attribute cache is updated whenever new attributes arrive from the server after a cache miss. Cached attributes are discarded typically after 3 seconds for files or 30 seconds for directories [Levy & Silberschatz 90]. NFS guarantees consistency under sequential write sharing, but not under concurrent write sharing.

With a write-back policy, as used in Sprite, consistency with sequential write sharing alone requires that all modified blocks of a file be written to the disk before that file can be opened by other clients. Sprite ensures the consistency of data in the caches even in the event of concurrent write sharing. It guarantees that each read will return the most recently written data for the file, even if the file is being read and written simultaneously on more than one machine [Baker *et al.* 91, Baker *et al.* 92, Nelson *et al.* 88, Srinivasan 89]. This is achieved by mechanisms similar to the general version number mechanism described previously. The Sprite server keeps track of the last client to write each file. If another client opens the file, the server recalls any dirty data not yet written back to the disk from the last writer's cache. If two or more clients have the same file open simultaneously, and at least one of them has it open for writing, the server disables client caching on the file until all the clients have closed it. The file servers are used as centralized control points for cache consistency; there are no direct client-to-client interactions.

Andrew maintains multi-cache consistency under sequential write sharing (but not under concurrent write sharing) by using a *callback* scheme [Howard *et al.* 88, Levy & Silberschatz 90], which is as follows. Venus assumes that cached entries are valid unless notified otherwise, so that Venus need not contact Vice on a file open in order to validate the cached copy. When a client caches a file or a directory, the server updates its state information recording this caching – the server has a callback on that file. The client is notified by the server before the server allows modification to the file by other clients – the server removes the callback on the file for the former client. A client can use a cached copy of a newly opened file only if the file has a callback. By reducing cache validation traffic, callback reduces the load on servers considerably [Howard *et al.* 88].

In Coda [Kistler & Satyanarayanan 92], for files and symbolic links, Venus purges object on callback break, and refetches it on demand. If a disconnection occurs before refetching, the object would be unavailable at the disconnected client. For directories, Venus does not purge on callback break, but marks the cache entry as suspicious. A callback break on a directory typically means that an entry has been added to or deleted from the directory, and often names for other directory entries and objects remain unchanged. In this way, consistency suffers a little, but availability increases significantly.

For Echo, a similar but more complex scheme to what Sprite uses is employed for consistency. Like Sprite, both sequential and concurrent write sharing is supported in Echo. At any moment, the file system data is in a single, well-defined state. The global name space is less consistent but more available, while the file volume service is more consistent but less available [Hisgen *et al.* 89]. On the global name service level, the cache consistency is weak; cache invalidation is triggered only by expiration times (say several hours). Data may be obtained from an out-of-date cache that is equivalent to data retrieved from a name service replica that is not in communication with the other replicas. On file volume service levels, however, Echo achieves its cache consistency by having the server keep track of which clients have cached which files and directories [Hisgen *et al.* 89]: the server calls back on clients at the start of an update to a file or directory by any client, telling them to discard the file or directory from their caches. The file consistency algorithm guarantees that two clients cannot see different data simultaneously .

## 2.3 Summary

Cache design issues have been discussed in this chapter using five distributed file systems as examples: Sun NFS, Sprite, Andrew, Coda, and Echo. Table 2.1 presents a brief summary of how cache design issues are handled in these five systems. In the table, "sequential" means consistency is guaranteed for sequential write sharing while "concurrent" means consistency is guaranteed for concurrent write sharing. S

Table 2.1: Summary of Surveyed File Systems

| System | Cache Location | Cache Size | Write Policies | Consistency Guarantees |
|---|---|---|---|---|
| NFS | Memory | Fixed | Write-through | Sequential |
| Sprite | Memory | Variable | 30 sec. write-back | Sequential&Concurrent |
| Andrew | Disk | Fixed | On close | Sequential |
| Coda | Disk | Fixed | On close | Sequential |
| Echo | Memory | Fixed | Ordered write-back | Sequential&Concurrent |

# Chapter 3

# Non–Volatile Caches

Large main memory disk caches reduce read traffic more effectively than write traffic. As disk cache sizes on both clients and servers continue to grow and the caches satisfy an even large fraction of the read traffic, write traffic will increase in proportion and could potentially become a bottleneck. Higher-performance and more reliable distributed computer systems, with some amount of non-volatile RAM (NVRAM) on the workstations and/or on the file servers, are increasingly feasible as the cost of NVRAM decreases [Moran *et al.* 90, Baker *et al.* 92, Biswas *et al.* 93a, Biswas *et al.* 93b, Ruemmler & Wilkes 93a]. Thus, non-volatile caches can be used to cache writes reliably for long time periods comparing to volatile caches, to allow full advantage to be taken of the temporal and spatial locality in the reference stream.

Basically, there are three ways to provide non-volatile memory: RAM with battery backup, RAM with an uninterruptible power supply (UPS), and flash EEPROM [Baker *et al.* 92]. RAM with an UPS is more expensive than RAM with battery backup, even for a small amount of memory. Flash EEPROM is significantly slower than NVRAM in write access time, and can only be written a limit of times. For the purpose of write caching, RAM with battery backup is the best choice [Baker *et al.* 92]. NVRAM is available in separate memory components (SIMMs) and on boards with redundant batteries. Today, NVRAM is still 4-6 times more expensive per megabyte than volatile DRAM, the memory access speed of board NVRAM and small SIMMs (512$\times$8RAM) can be as fast as volatile DRAM (1M$\times$9DRAM) at 70 ns [Baker *et al.* 92].

Section 3.1 of this chapter briefly surveys previous research work on the use of NV (NV) caches and buffers, including research work that has focused on use of NV caches at the clients and/or servers of a distributed file system, and work that has considered only the use of a NV cache as write buffers for disks. Section 3.2 describes a number of caching policies proposed for systems with NV caches.

## 3.1 Non–Volatile Caches and Buffers

A cache system with volatile read caches and NV write caches on servers or clients of a distributed file system can reduce the write traffic significantly, since blocks in a NV cache may be modified several times or even deleted from the cache before they are written back to the disk. Moreover, storing data in NV caches is almost as reliable as storing data on disk.

Two uses of NV caches were proposed by Baker *et al.* [Baker *et al.* 92] to improve I/O performance in distributed systems: NV disk caches on client workstations to reduce write traffic to the file server, and write buffers for file systems on the server to reduce server disk accesses. File I/O traces at the level of kernel calls of Sprite systems at UC Berkeley were gathered for a trace-driven simulation study. The NV caches were managed by an LRU replacement policy. The net write traffic (the percentage of bytes written to the client caches that eventually had to be written to the server) was studied as a function of the write-back time (the length of the time period for the periodic write-back policy) and the NV cache size. A lifetime study of cached bytes in the traces showed that normally 35-50% of written bytes die within 30 seconds, while about 60% of bytes die within a few hours. When large amounts of data were transferred, only 5-10% of bytes die within 30 seconds, while more than 80% die within half an hour. The write traffic to the server is due mainly to the cache consistency mechanism. It was found that the amount of file data written to the server from clients and the number of disk accesses by the server can be reduced considerably [Baker *et al.* 92] by using NV caching. A megabyte of NVRAM on diskless clients reduces the amount of file data written to the server by 40%. A one-half megabyte write-buffer on the server side can reduce disk accesses by about 20%. It should be noted that the logging scheme used on the Sprite file server makes the server's write operations more effective than in other file systems. Therefore, a larger percentage of disk access reduction can be expected on systems where no write optimization is performed.

Biswas *et al.* [Biswas *et al.* 93b] modeled a distributed file system with NV caches on both clients and servers. Their model uses UNIX-like semantics for write operations from multiple clients, LRU as the cache replacement policy, and a multi cache consistency policy similar to that used in Sprite. A synthetic workload was used that based on file I/O traces collected from commercial time-sharing VAX/VMS systems [Ramakrishnan *et al.* 92]. The performance of several policies for NV cache (write-back with thresholds, periodic write-back, and LRU write-back) was compared as the number of clients increased from 100 to 700 and the size of the NV cache on both the server and client was varied. On the server, it was found that the threshold policy generated fewer writes to disk than the other policies, and that 8 megabytes of NV cache at the server was adequate when the system resources were not heavily utilized. On the client, it was observed that the write response time with the NV cache at the client was substantially lower than when no client NV cache was used. Periodic write-back performed better at low loads (100 clients), while write-back with thresholds performed better at high loads (400 - 700 clients). The write response time for all load levels decreased sharply until the client cache size reached 1 megabyte.

Several previous studies indicated that the performance of the disk I/O subsystem can be improved significantly by using NV disk caches at the server. NV cache at the server can be very effective in reducing the time that clients wait for disk I/O, because the write requests from clients can be satisfied by NV cache at memory speeds rather than disk speeds. Since persistent sequences of file I/O requests are resolved in the NV cache, the total amount of disk I/O is reduced. NV write caching also permits more efficient disk I/O through more intelligent scheduling of the disk operations. Several requests can be scheduled at a time to take advantage of the position of the disk arm.

Moran *et al.* [Moran *et al.* 90] evaluated the performance of NFS when NV caches were implemented on the server. "Nhfsstone", a NFS load generating and performance measuring benchmark, was used to generate a reproducible artificial load on the server based on an NFS operation mix obtained from working systems. When a synchronous write request is issued to a disk, the request is intercepted by the device driver, which stores the data in NV memory instead of on disk. Thus, synchronous writes occur at memory speeds. As the cache fills up, cached data are sent to the disk asynchronously, in batches large enough to allow the disk drivers to optimize the order of the writes. It was observed that the response time of write operations could be acceptable with write caching, even if the synthetic load was as high as 100 NFS operations per second [Moran *et al.* 90].

Biswas *et al.* [Biswas *et al.* 93a] modeled the components of an I/O subsystem including the disk controller, the read and write caches that were postulated as part of the controller, and the magnetic media. Two write caching policies were studied for NV caches: write-back with threshold, and piggybacking (described in Section 3.2). The performance of the algorithms was analyzed by means of trace-driven simulation using file I/O traces collected from several commercial time-sharing VAX/VMS systems [Ramakrishnan *et al.* 92]. Among the findings were the following: the number of read/write operations were reduced significantly when NV caching was used; 1-2 megabytes of NV cache was adequate for write caching; an appropriate choices for the write-back thresholds were 95-99% for the high limit and 20-60% for the low limit; piggybacking performed the best; and the average read queueing time for the write caching policies was considerably improved compared to the case where there was no write caching (the baseline case). Overall, the use of NV disk cache at the server together with the appropriate write strategies reduced the number of write operations to disk by over 50%.

Ruemmler [Ruemmler & Wilkes 93a, Ruemmler & Wilkes 93b] addressed the file I/O problem in a distributed file system by adding NV memories to disk controllers as write buffers. Tracing was done on three different Hewlett-Packard computer systems running a version of the BSD UNIX fast file system [McKusick *et al.* 84]. (A BSD UNIX fast file system is a reimplementation of the UNIX file system, which clusters sequentially accessed data and provides two block sizes to allow fast access to large as well as small files. The file access rate in the reimplementation can be up to ten times faster than the traditional UNIX file system). The analysis of their traced data showed that the majority of disk accesses (57%) were write, only a small percentage of read and write accesses were logically sequential at the disk level (two requests are logically sequential if they are to adjacent disk addresses or disk addresses spaced by the file system interleave factor), most of the I/O operations (50-75%) were synchronous, about 10-18% of all write requests were overwrites of the last block written out, 25-38% of updated blocks were overwritten in less than 1 second, 45% of the blocks were overwritten within 30 seconds, 18% of the blocks were overwritten at 30-second intervals, and over 85% of all blocks written were overwritten in an hour or less. The disk I/O subsystem was modeled in detail to explore the disk design with NV RAM caches added to each disk. It was found that a small NV cache at each disk allowed writes to be serviced considerably faster than just a regular disk. The write traffic could be reduced by 10−18% by adding 8 kilobytes of NV memory per disk.

## 3.2 Caching Models and Policies

In this section, caching models and policies related to NV caches are discussed. The caching models used in the previous studies are described first, followed by the description of replacement policies.

### 3.2.1 Caching Models

Models for the relationship and cooperation of both the volatile caches (for read caching) and NV caches (for write caching) have been presented in the literature. Two such caching models used in [Baker *et al.* 92] are discussed in this section: the *write-aside* model and the *unified* model.

Under either the write-aside or the unified caching model, two separate lists of blocks are maintained for each cache: one is the free list that contains free blocks, and the other is for the blocks currently in use. Dirty blocks remain in the NV cache until replaced by other blocks or until flushed back to the server for the purpose of consistency. The term *flush* denotes the operation of writing dirty blocks to their master copy on disk. Once the dirty block is purged to disk, it is marked "clean" and the entry of that block is placed in the free block list of the NV cache. This clean block may be removed from the NV cache or remain as a clean block in either the NV cache or the volatile cache if there still is space. Subsequent reads to the clean block can still be resolved from either the volatile cache or the NV cache.

In the write-aside model, the NV cache is intended only to protect dirty data in the volatile cache. Copies of blocks that are also present in the volatile memory are held in the NV cache. The NV cache is never read, except during crash recovery. This model minimizes access to the NV cache, and so the fact that NV memory is significantly slower than volatile memory is accommodated. Writing data into both the volatile cache and the NV cache increases the traffic on the memory bus, however.

In the unified model, dirty blocks reside only in the NV cache. Clean blocks, however, can reside in either the volatile cache or the NV cache. For instance, if a read operation finds that the volatile cache is full and the NV cache still has a free block or contains the block to be replaced next according to the replacement policy in use (LRU or RAND), a clean block containing the required data may be put into the free block or replace the block to be replaced next in NV cache. Application reads can be satisfied by either the volatile cache or the NV cache, while application writes are only directed to the NV cache.

Schemes to update the volatile read cache when a block is written to the NV cache in this caching model can be [Biswas *et al.* 93a]: write allocate, write update, or write purge. In write allocate, a copy of a block written to the NV cache is also placed in the read cache if it is not already there. Write update requires that the block be updated in the read cache if it is already present, but if not, the block is not added. Write purge removes the block written to the NV cache from the read cache if it is there. Write purge was recommended in [Biswas *et al.* 93a] for all environments except those with a large amount of read-after-write activity.

### 3.2.2 Policies for Non–Volatile Cache Management

Policies that have been considered for managing NV caches in the literature include: LRU replacement on both servers and clients [Baker *et al.* 92], modified LRU replacement on servers [Moran *et al.* 90], LRU write-back on servers [Biswas *et al.* 93b], periodic write-back on both servers and clients [Biswas *et al.* 93b], write-back with thresholds on both servers and clients [Biswas *et al.* 93a, Biswas *et al.* 93b], and piggybacking on servers [Biswas *et al.* 93a].

#### LRU Replacement

With an LRU replacement policy, a free list containing free blocks and an LRU list containing blocks in use with an LRU order are maintained for each cache. When a block must be brought into the cache, a free block is chosen. If there is no free block left, the least-recently-used block is chosen for replacement, regardless of whether it is clean or dirty, and written to disk if necessary. In the study of Baker *et al.* [Baker *et al.* 92], when a block in the NV cache gets replaced, the volatile cache may be updated. When a write request comes and causes a block in the NV cache to be replaced, the removed block is written back to the server if the block is dirty. The access time of the removed block from the NV cache is compared to that of the least-recently-used block in the volatile cache. If the least-recently-used block in the volatile cache is older, the LRU block is replaced with a clean copy of the block removed from the NV cache. The RAND replacement policy was also used in their study on both servers and clients [Baker *et al.* 92].

A modified LRU replacement policy was used in the study of NFS performance by Moran *et al.* [Moran *et al.* 90] as well. In this policy, read requests not satisfied by the volatile server cache may be satisfied from the NV cache, and blocks repeatedly updated may be written to the NV cache rather than to disk. This modified LRU policy can efficiently flush the cache and keep enough free blocks available to handle burst traffic.

### Write-Back with Thresholds

Write-back with thresholds [Biswas *et al.* 93a] is a proactive purging policy for use on file servers in distributed systems. In this algorithm, thresholds are used to limit the percentage of dirty blocks in the cache. When the percentage of dirty data blocks in the NV cache exceeds a *high limit threshold* (a large percentage of the capacity of the NV cache), purging is scheduled. Purging starts at the first opportunity when the disk becomes idle (no read operations pending for processing), and continues until the percentage of dirty data blocks in the NV cache falls below the threshold. If a *low limit threshold* (a small percentage of the capacity of the NV cache) is used, the purge operation will be discontinued only when the percentage of dirty blocks falls below the low limit threshold. Write-back with thresholds attempts to clean the NV cache before the cache is full of dirty blocks by using thresholds, and tries to do the purges when the disk is quiescent by postponing purge operations if there are read operations pending for processing.

As indicated by Biswas *et al.*, write-back with thresholds may degrade to simple write-back. Write-back with thresholds is supposed to start to purge when the disk becomes idle for the first time after the purge operation is scheduled. It is possible, however, for the purge operations not to get the opportunity to proactively purge the NV cache. As a result, the cache can become full with dirty blocks. In that case, an *immediate purge*, which has the same priority as a read miss request to the disk, is scheduled. This operation is discontinued as soon as there are any free blocks in the NV cache and the system returns to proactive purging.

Purge operations for the NV cache can be optimized by taking advantage of the way the disk is managed [Biswas *et al.* 93a, Moran *et al.* 90, Ruemmler & Wilkes 93b]. A disk head is used to access any position of the disk. The access time of a disk has three principle components: seek time, rotational latency, and transfer time. *Seek time* is the time for the read/write head to travel to the required cylinder. The seek time depends on the search distance (i.e., the number of tracks over which the head must be moved). *Rotational latency* is the delay required for the correct disk block on the track to spin around to the head. *Transfer* time is the read/write time itself. According to the structure of a disk, only one seek operation is needed to write all the dirty blocks on the same disk track/cylinder to the disk. One way to optimize purging operations [Biswas *et al.* 93a] is to choose the disk track/cylinder with the most dirty blocks in the NV cache to be written back to the disk first. It was also suggested that seek distance can be used to choose the appropriate track to purge.

Write optimization can be accomplished only if the physical disk locations of the blocks in the cache are known. Otherwise, optimizations based on disk structures cannot be performed directly. In a distributed file system, since a client knows only the logical address of each request rather than its physical address, direct disk track/cylinder-based write optimization is not possible. Also, it is not possible for the client to know when the server is idle and therefore the client will purge its NV cache whenever it needs to rather than taking into account the busy/idle state of the device being purged to [Biswas *et al.* 93b].

### Piggybacking

A read miss happens when a read I/O request is not satisfied by either the client cache or the server cache, and results in a disk access operation to bring the required block into the client cache. In addition to satisfying the read request, however, the disk operation can also write back any dirty blocks in the server NV cache which belong to the same disk track/cylinder as the read request as part of the same disk access operation. Piggybacking [Solworth & Orji 90, Biswas *et al.* 93a] tries to piggyback the write back of dirty blocks from the server NV cache with read operations to the same disk track/cylinder.

As discussed in [Solworth & Orji 90, Biswas *et al.* 93a], *free* (zero-cost) track piggybacking and *full* track piggybacking are two variations of the piggybacking operation (see Figure 3.1). In free piggybacking, the read response times are not affected by piggybacking. Hence, only those dirty blocks on the target track in the NV cache that are in the region between the current head position and the position where the read operation starts can be written to the disk. Free piggybacking uses the rotational latency to write the dirty blocks from the server NV cache to the disk. In Figure 3.1, for instance, only block A is eligible for free track piggybacking while blocks A and B are both available for full track piggybacking. In full track piggybacking, all dirty blocks in the track being visited by the read miss are available for piggybacking. The read operation is not completed until the last dirty block on the track has been piggybacked out from the NV cache. Therefore, the service time of the read operation could be increased in full piggybacking.

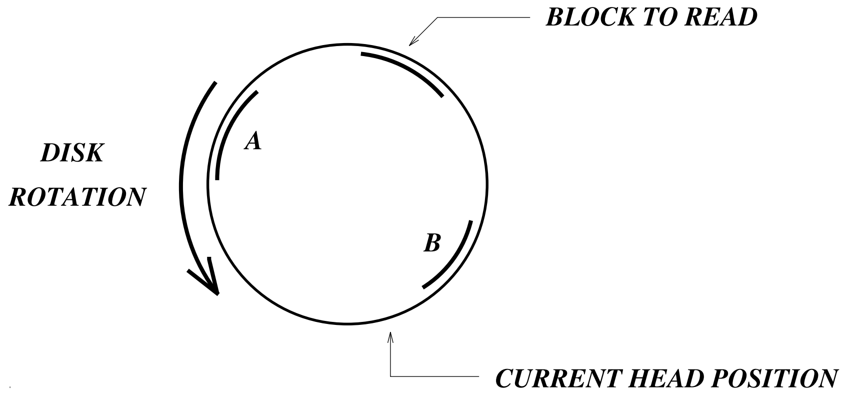



Figure 3.1: Schematic Diagram of a Disk Platter to Explain Piggybacking [Solworth 90]. Full piggybacking: dirty blocks A and B are eligible for piggybacking. Free piggyback: only dirty block A is eligible for piggybacking.

## 3.3 Summary

The use of NV memories as write caches and buffers has been studied previously and it has been determined that the performance of write operations can be significantly improved by NV caching. Two caching models for dealing with the relationship and cooperation of the volatile and NV caches have been proposed [Baker *et al.* 92]. Schemes that have been studied for updating the volatile cache in the caching model when blocks are written to the NV cache include: write allocate, write update, and write purge. Several policies for managing NV caches in the related work have been considered in this chapter, including LRU replacement, write-back with thresholds, and piggybacking.

# Chapter 4

# Experimental Design and Methodology

This chapter lays the foundations for the experimental work performed in this thesis. Section 4.1 describes the motivations and objectives of this research. Section 4.2 presents the experimental design, including the system simulated, the NV caching model for the simulation, the NV caching policies, and the combinations of NV caching policies considered. The performance metrics are also described in this section. Section 4.3 describes the traces used in the experiments, giving a description of the events traced, a description of the system traced, and a description of the method of collecting and processing the traces. Section 4.4 describes the simulator and simulation runs, as well as the factors and levels of factors considered in the experiments. Section 4.5 summarizes the chapter.

## 4.1 Thesis Motivations and Objectives

Some studies have found that write events currently account for about one-third of the total file bytes transferred between clients and the file server, while the rest are read events. Most of the write events are a result of the periodic write-back caching policy and the cache consistency policy rather than the cache block replacement [Baker *et al.* 91, Baker *et al.* 92]. Other studies have found that 57% of disk accesses at the server are writes [Ruemmler & Wilkes 93a]. To reduce the network traffic, decrease write response time, and decrease server load, non-volatile RAMs can be used at the clients for write caching, without risking data loss. To reduce the frequency of synchronous disk accesses at the server, non-volatile RAMs can also be used on the server to provide better write performance.

To determine the effectiveness of NV write caching for the purpose of reducing the network and server write traffic, the following questions about NV write caching are addressed in this thesis work:

- What fraction of writes can be saved at the clients and at the server when certain policies are used for NV write caching (how efficient is a write caching policy)?
- What NV write caching policies work best for the clients and the server? How do NV write caching policies used at the client interact with the write policies used at the the server? What combinations of NV write caching policies at the client and the server work best?
- How big should the NV caches be at the clients and at the server to reduce a significant amount of the network traffic and the server write traffic? What tradeoffs are there between the sizes of NV caches at the clients and at the server? What are the best choices for those sizes?
- How should the NV write caching on the clients and the server be coordinated to achieve better write performance?

## 4.2 Experimental Design

This section describes the distributed systems simulated, the caching policies for managing NV caches on the clients and servers, the corresponding caching models for the policies simulated, the policy combinations considered in this thesis, and the experimental metrics used in the simulation experiments.

### 4.2.1 The Simulated System

Both a single-level caching system (a standalone system with a local disk) and a multi-level caching system (a distributed system with one server and one diskless client) are simulated. The standalone system is simulated to obtain data on the performance of policies in a simple environment, which can help explain the performance of the policies in the more complex client-server environment that is really of interest.

The single-level caching system is a standalone workstation, which has a volatile cache, a non-volatile cache, and a local disk. The multi-level caching system is shown in Figure 4.1. NV caches as well as volatile caches may be present at both the client and the server. The caching models for dealing with the relationship between and the coordination of the volatile and non-volatile caches are described in Section 4.2.3. Write caching policies simulated for the NV caches are described in Section 4.2.2, followed by the policy combinations considered in this thesis and the performance metrics used for the simulation. The details of how the volatile and the non-volatile caches operate and the data structures for each NV caching policy are given in Section 4.2.3.

The input to the simulations consists of workload traces collected from the execution of real applications. The traces consist of time ordered (and time stamped) sequences of file system "events" which were captured by a tracing package. In the single-level caching system, for each traced event the simulator is given, the operations that the event would cause at the volatile and the NV caches are simulated. For the multi-level caching system, the operations that the event would cause at the client caches are simulated first, followed by the simulation of those operations (if any) that would be caused, at the server caches and the server disk. The operation of both the volatile cache and the NV cache in the single-level caching system, and the operation of both caches at both the server and the client in the multi-level caching system are simulated in detail (as indicated in Sections 4.2.2 and 4.2.3).

The time associated with network operations, disk operations, and the queueing requests is not simulated in this study. This simplification makes little difference to

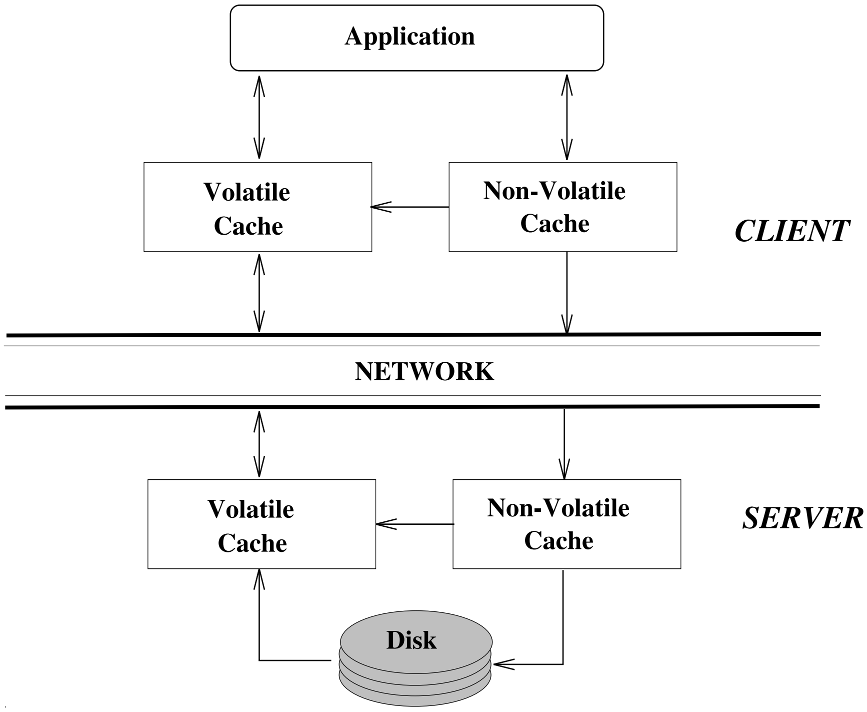



Figure 4.1: The Distributed System Simulated

the results, since the metrics described in Section 4.2.5 do not reflect the effects of time.

Multi-cache consistency is assumed to be accomplished by some consistency policy, but such a policy is not explicitly simulated, nor is the overhead of maintaining such a consistency included. This assumption should not significantly affect the performance results, since there are not many concurrent write activities. The block size in all the caches is assumed to be 8 kilobytes, justified by the fact that the cache unit size in UNIX-like systems is 4 to 8 kilobytes [Levy & Silberschatz 90].

### 4.2.2 Write Caching Policies

To focus on simple policies rather than complex compound policies such as FBR and KBR, five write policies, LRU replacement (LRU), LRU purge with thresholds (LRUPT), write-back with thresholds (WBT), write-back with thresholds with piggy-backing (WBT-PB), combined LRU replacement (COMB), and coordinated purging with thresholds (CPT), are studied in both the single-level caching system and the multi-level caching system. In a multi-level system, these policies are simulated on the client, or on the server, or on both the client and the server.

For the purging based policies (LRUPT, WBT, and WBT-PB), purging is scheduled when the percentage of dirty blocks in the NV cache exceeds a high limit threshold, and is stopped when the percentage of dirty blocks falls below a low limit threshold. The NV caches always have space for newly written dirty blocks unless the high limit threshold is set to 100%. In the multi-level caching system, purge operations from the NV cache at the client to the NV cache at the server start whenever they are scheduled, while purge operations from the server to the disk are delayed until the first time the disk is idle after the purge operation is scheduled.

In the single-level caching system or at the server of a multi-level caching system, purge operations can be optimized by writing back in one operation all the dirty blocks belonging to the same disk track/cylinder. However, for a remote diskless client in a multi-level caching system, track/cylinder based purging optimization may be impossible, since the client may not have the information of which block belongs to which disk track. In this case, file membership could possibly be used as a basis for purging optimization at the client in a multi-level caching system.

To reduce the disk seek time and latency of sequential operations on files, the operating system kernel tries to organize the list of free block numbers for a disk so that blocks dispensed to the same file are near each other [Bach 86]. Therefore, all the dirty blocks belonging to the same file can be purged together from a client's NV cache in a multi-level caching system, to optimize the purge operation, since they must be stored close to each other and the file identity of each cached block is known to the client.

In all experiments, the volatile caches in the single-level system and on the client and the server in the multi-level caching system, are managed by the LRU replacement policy for read requests. This decision was made to reduce the complexity of the experiment. For the LRU, LRUPT, WBT, and WBT-PB policies, read misses are placed only in the volatile cache. For policy COMB, read misses can be placed into both the volatile and the non-volatile caches (described in this section in more detail).

The policies considered for managing the non-volatile caches are the following:

**LRU: LRU Replacement** The NV cache is managed by the LRU replacement policy. When a write request is received, the NV cache is searched for the requested block. If the required block is in the cache (a hit), then the block is moved to the MRU end of the NV LRU list. If the required block is not in the cache (a miss), then the block is inserted into the NV cache if there are empty blocks. If the NV cache is full, the LRU replacement policy is applied to the NV cache to make room for the miss. The least-recently-referenced dirty block is flushed, (either to the disk from the NV cache in a single-level caching system, or to the NV cache at the server from the client's NV cache in the multi-level caching system), and removed from the NV cache where it was cached. Free and LRU lists of cache blocks are maintained for each NV cache managed by this policy.

**LRUPT: LRU Purging with Thresholds** LRUPT maintains the NV cache by using a LRU list for all the dirty blocks cached, and a free list for clean blocks. Blocks are removed from the NV LRU list by purging in LRU order. All the dirty blocks that are in the same file (at the client in the multi-level caching system) or on the same disk track/cylinder (in the single-level caching system or at the server of a multi-level caching system) as the least-recently-referenced block are purged in one purging operation when purging is scheduled.

**WBT or WBT-PB: Write-Back with Thresholds (with Piggybacking)** In write-back with thresholds, there is no LRU order for purging. Instead, for a NV cache in the single-level caching system or for a server NV cache in the multi-level caching system, those blocks belonging to the disk track or cylinder with the most dirty blocks are purged first, while for a client NV cache in the multi-level caching system, those blocks belonging to the file with the most dirty blocks are purged first.

Piggybacking tries to merge write requests with read requests to the same area of the disk. In the single-level caching system or on the server of the multi-level caching system, write requests to blocks on the same disk track/cylinder as the "missed" read request (missed in the caches at both client and server) can all be piggybacked in full track/cylinder based piggybacking, or can be partially piggybacked in free piggybacking.

For a diskless client in the multi-level caching system, however, the physical addresses of referenced blocks are unknown. Track/cylinder-based piggybacking is therefore impossible. For this reason, files are used as the basis for piggybacking on clients. Write requests to the same file as the "missed" read request can be sent to the server's NV cache together. By doing this, write traffic might be reduced when many files are opened for both reading and writing.

It is possible for the NV cache to become full when there are many write operations and not enough read misses to be used for piggybacking all those writes. In this case, WBT will be invoked to purge the NV cache.

**COMB: Combined LRU Replacement** In this policy, both the volatile and the NV caches are maintained as LRU lists. Clean blocks required by read operations, missed in both the volatile and NV caches, can be placed into either the volatile or the NV caches. Dirty blockse required by write operations, missed in the NV cach, can only be placed into the NV cache. When a read comes, the volatile cache is searched first. If the required block is in the volatile cache, the block is moved to the MRU ends of the volatile LRU list. If the required block is not in the volatile cache, the NV cache is then searched. If the required block is in the NV cache, the block is moved to the MRU end of the NV LRU list. If the read request is missed at both the volatile and the NV caches, the block required by the read operation will be placed into the volatile cache if the cache is not full. If the volatile cache is full, and the NV cache is not, then the block required by the read operation can be placed into the NV cache. If both the volatile cache and the NV cache are full, the LRU ends of the two caches are compared to see which one is older. The block at the older LRU end, regardless of whether it is clean or dirty, gets replaced to make room for the new read block. A dirty block that is replaced is flushed.

When a write comes, the request goes only to the NV cache managed by the LRU replacement policy.

**CPT: Coordinated Purging with Thresholds** This policy is introduced for the first time in this thesis for the multi-level caching system only. This policy tries to coordinate the purge operations on the clients and on the servers. When the server starts purging dirty blocks and the used portion of the server's NV cache is already below its high limit, the purge operations on the clients and server can be coordinated as follows. Before each purge operation, the server finds what files the dirty blocks being purged next belong to and asks each client to purge dirty blocks belonging to the identified files. The server then accepts as many dirty blocks into its NV cache as possible until its high limit threshold is exceeded again, at which time the server starts the next purge operation. The clients use purge based policies (LRUPT or WBT) when purging is needed only at the client.

This policy tries to tell the client what to purge first in order to coordinate the purge operations on both the client and the server. By doing this, further optimization of the purge operations on the server and the clients may be expected.

### 4.2.3 Caching Models

The semantics of the caching operations, between the volatile and the NV caches, and between client and server in the multi-level caching system, are defined by the particular caching model employed. Three caching models were considered in this research: the LRU purging with thresholds unified caching model (used for LRU and LRUPT), the write-back with thresholds unified caching model (used for WBT and WBT-PB), and the combined caching model (used for COMB).

The following models assume the system simulated is a multi-level caching system.

However, they can also be used to simulate the single-level caching system when caching at the client is disabled.

### The LRU Purging with Thresholds Unified Caching Model

In this model, read misses can go only to the volatile cache and write misses can go only to the NV cache. Purging operations commit dirty blocks from the NV cache at the client to the NV cache at the server or from the NV cache at the server to the disk, and keep the purged dirty blocks in the NV cache as clean blocks after purging. Blocks put in the NV cache by write misses are marked as dirty blocks. A free list and an LRU list are maintained for the volatile cache, and a free list for blocks that are not used in the NV cache, an LRU clean list for the clean blocks in the NV cache, and an LRU list for the dirty blocks in the NV cache are maintained for the NV cache. This caching model operates as follows:

*1. For read requests on the CLIENT or SERVER*

```
- search the volatile cache first
if the request is a hit in the volatile cache
        - move the block to the MRU end of the volatile LRU list
        if on SERVER
                - return a copy of the block to the CLIENT
else    /* read miss in the volatile cache */
        - search the NV clean LRU list
        if the request is a hit in the NV clean LRU list
                - save the block contents in a TMP
                - delete the block from the NV clean LRU list
                - add the block to the NV free list
                - replace the LRU block of the volatile LRU list by TMP
                if on SERVER
                        - return a copy of TMP to the CLIENT
        else /* missed on NV clean LRU list */
        - search the NV dirty list
                if the request is a hit in the NV dirty list
```

```
                        - save the block contents in a TMP
                        - replace the LRU block of the volatile
                          LRU list by TMP
                          /* now there is one copy of the block in
                             the volatile LRU list and another one in
                             the NV dirty LRU list */
                        if on SERVER
                                - return a copy of TMP to the CLIENT
                else  /* read miss */
                        if on CLIENT
                                - send request to SERVER
                                - accept the newly read block from the
                                  SERVER
                                - replace the LRU block of the volatile LRU
                                  list on the CLIENT by the new read miss
                        if on SERVER
                                - send request to disk
                                - accept the newly read block from the disk
                                - LRU replace the volatile LRU list
                                  on the SERVER by the new read miss
                                - return a copy of the block to CLIENT
```

Replacing the LRU end of the volatile LRU list by an identified block CLIENT or SERVER is performed in the following way:

```
if there is a free block in the volatile cache
        - remove the free block from the free list
        - add it to the MRU end of the volatile LRU list
else    /* the volatile cache is full */
        - find the LRU end of the volatile cache
        - delete the contents
        - move it to the MRU end
- put the contents of the identified block at the MRU end of the
```

```
  volatile LRU list
```

*2. For write requests on the CLIENT or SERVER*

```
- look into the NV cache /* purge if necessary */
if the percentage of dirty blocks > High Limit Threshold
        if on CLIENT
                while the percentage of dirty blocks > Low limit Threshold
                /* purge based on file membership */
                - search the NV dirty LRU list for all the blocks belonging
                  to the same file as the block at the LRU end
                - write all those dirty blocks back to the SERVER's NV cache
        if on SERVER
                while the percentage of dirty blocks > Low limit Threshold
                /* purge based on the disk track/cylinder membership */
                - search the NV dirty LRU list for all the blocks belonging
                  to the same disk track/cylinder as the block at the LRU end
                - write all those dirty blocks back to the disk
        - update the NV dirty LRU list
        - mark the blocks purged as clean blocks
        - add them to the NV clean LRU list in the order of purging
          /* LRU in the sense of the least-recently-purged
          not the least-recently-accessed */
        - purging stops when the low limit threshold is reached.
        if some of the blocks of the file (file base purge) being purged
        are left when the low limit threshold is reached
                   - next purging will start with this file first
- search the volatile cache
if hit in the volatile cache
        - save the block content in a TMP
        - remove the block from the volatile LRU list
        - add the block into the volatile free list
- search the NV clean LRU list
```

```
if the requested block is in the NV clean LRU list
        if not hit at the volatile cache
                - save the block content in a TMP
        - remove the block from the NV clean LRU list
        - add the block to the NV free list
- search the NV dirty LRU list
if the requested block is in the NV dirty LRU list
        - move the block to the MRU end of the NV dirty LRU list
else    /* considered to be a write miss on the NV cache */
        if there is a free block in the NV cache
                - remove the block from the free list
                - add it to the MRU end of the NV dirty LRU list
        else    /* NV cache is full of dirty/clean blocks */
                - find the LRU end of the NV clean LRU list
                - remove it from the NV clean LRU list
                - add it to the MRU end of NV dirty LRU list
        if the request is a hit in the volatile cache or a hit in the
        NV clean LRU list
                - copy TMP into the MRU end of the NV dirty LRU list
        - mark the MRU end of the NV dirty LRU list as a dirty block
- perform write on the MRU end of NV dirty LRU list
```

Notice that when the high limit threshold is the same as the size of NV cache, and the low limit threshold is one block smaller than the high limit threshold, LRU purging with thresholds becomes LRU replacement.

### The Write-back with Thresholds Unified Caching Model

This model is very similar to the unified model for LRU purge with thresholds, in the treatment of read/write misses and the determination of when a block in NV cache is marked clean or dirty. Like the previous model, a free list and an LRU list are maintained for the volatile cache. For the NV cache, a free list, an LRU clean list for the clean blocks in the NV cache, and a list for the dirty blocks with all the blocks belonging to the same file/track/cylinder stored in a sublist (See Figure 4.2),

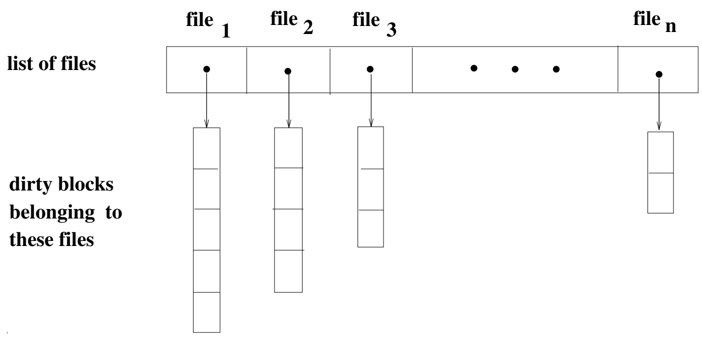



Figure 4.2: The Data Structure For NV Dirty Blocks with a Sublist for All the Dirty Blocks Belonging to Each File

are maintained. This model works as follows:

*1. For a read request on the CLIENT or SERVER*

The model works in the same way as the previous model. The LRU replacement of the volatile LRU list by an identified block is also done in the same way as in the previous model.

*2. For write request on the CLIENT or SERVER*

```
look into the NV cache
if the percentage of dirty blocks > High Limit Threshold
        if on CLIENT
                 while percentage of dirty blocks > Low Limit Threshold
                 /* purging based on file membership */
                 - search the NV dirty list for all the blocks belonging
                   to the file with the most dirty blocks in it
                 - write all these dirty blocks to the SERVER's NV cache
        if on SERVER
                 while percentage of dirty blocks > Low Limit Threshold
                 /* purging based on disk track/cylinder membership */
                 - search the NV dirty list for all the blocks belonging
                   to the disk track/cylinder with the most dirty blocks in it
                 - write all these dirty blocks to disk
```

```
        - update the NV dirty list
        - mark the blocks purged as clean blocks
        - add these new clean blocks to the NV clean LRU list
          in the order of purging /* LRU in the sense of the least
          recently purged not the least-recently-accessed */
        - purging stops when the percentage
          of dirty blocks <= Low Limit Threshold
        if some of the blocks of the file (file based purge) being purged
        still reside in the NV cache when the low limit threshold is reached
                 - next purging will start with this file first
- search the volatile cache first
if hit at the volatile cache
        - save the block contents in a TMP
        - remove the block from the volatile LRU list
        - add the block into the volatile free list
- search the NV clean LRU list
if the block is in the NV clean LRU list
       if not hit in the volatile cache
                - save the block contents in a TMP
       - remove the block from the NV clean LRU list
       - add the block into the NV free list
- search the NV dirty block list
if the block is in the NV dirty block list
        - perform write on it
          /* do not change the position of the block */
else      /* consider as a write miss on the NV cache */
        if there is a free block in the NV cache on the CLIENT
                - remove the free BLOCK from the free list
        else    /* NV cache is full of dirty/clean blocks */
                - find the LRU end of the NV clean LRU list
                - remove the LRU end BLOCK from the NV clean LRU list
        - add the BLOCK to the end of the NV dirty block list
```

```
        if hit in the volatile cache or hit in the NV clean LRU list
                - copy the content of TMP to the identified BLOCK
        - mark BLOCK as a dirty block
        - perform write on BLOCK
        if there is a sublist for the file/track/cylinder the newly
        written block belongs to
                - add BLOCK appropriately to the end of the sublist
                - update the NV dirty list
        else    /* new file has been written */
                - add the BLOCK to the end of the NV dirty list
```

### The Combined Replacement Unified Caching Model

In this model, read misses can go to both the volatile and the NV caches, while write misses can go only to the NV cache. Read misses can put clean blocks into the NV cache while write misses can put dirty blocks into the NV cache. A free list for blocks not in use and an LRU list for blocks in use are maintained for both the volatile cache and the NV cache. This model works as follows:

*1. For read requests on the CLIENT or SERVER*

```
- search the volatile cache
if request is a hit in the volatile cache
        - move the block to the MRU end of the volatile LRU list
        if on SERVER
                - return a copy of the block to the CLIENT
else    /* read miss at the volatile cache */
        - search the NV cache
        if request is a hit in the NV cache
        /* block hit can be clean or dirty */
                - move the block to the MRU end of the NV LRU list
                if on SERVER
```

```
                        - return a copy of the block to the CLIENT
        else    /* read miss on the client */
                if on CLIENT
                        - send request to SERVER
                        - accept the required block from SERVER
                        - put the newly read block into the CLIENT cache
                if on SERVER
                        - send request to disk
                        - accept the required block from the disk
                        - put the newly read block into SERVER caches
                        - return a copy of the block to the CLIENT
```

To put the newly read block into caches on CLIENT/SERVER:

```
if there is a free block in the volatile cache
        - remove the free block from the free list
        - add it to the MRU end of the volatile LRU list
        - put the contents of the newly read block into the MRU block
else    /* the volatile cache is full */
        if there is a free block in the NV cache
                - remove it from the free list
                - add it to the MRU end of the NV LRU list
                - mark the block as a clean block
                - put the newly read block into the MRU block
        else    /* the NV cache is full */
                - compare the LRU ends of the volatile and NV caches
                if the older LRU block is in the volatile cache
                        - remove the LRU end from the LRU list
                        - add it to the MRU end
                        - put the newly read block into the block
                          at the MRU end of the volatile LRU list
                else    /* the older LRU block is in NV cache */
                        if the LRU block is dirty
```

```
                                - write the dirty block to the SERVEdisk
                        - remove the LRU block from the LRU end
                        - add it to the MRU end
                        - mark the block as a clean block
                        - put the newly read block into the block
                          at LRU end of the NV LRU list
```

*2. For write requests on the CLIENT or SERVER*

```
- search the volatile cache
if the request is a hit in the volatile cache
        - save the block contents
        - remove the block from the LRU list
        - add it to the free list
- search the NV LRU list
if the request is a hit in the NV cache
/* block found can be clean or dirty */
        - move the block to the MRU end of the NV LRU list
else    /* write miss on the NV cache */
        if there is a free block in the NV cache
                - remove the block from the free list
                - add the block to the MRU end of the NV LRU list
        else    /* NV cache is full of dirty/clean blocks */
                if the LRU end of the NV LRU list is dirty
                        - write the block back to SERVER or disk
                - remove the block from the LRU end and
                  add it to the MRU end of the NV LRU list
        if the request is a hit in the volatile cache
                - copy the saved contents of block hit in the volatile cache
                  into the MRU end of the NV LRU list
        - mark the MRU end of the NV LRU list as a dirty block
- perform the write on the MRU end of the NV LRU list
```

### 4.2.4 Policy Combinations Considered

This section describes the particular policies or policy combinations investigated in this thesis.

For the single-level caching system, all five cache policies are simulated. A previous study [Willick *et al.* 93] reported that in a multi-level caching system, LRU does not perform well for read caching at the server because of the "filtering" of temporal locality by client caches. For the multi-level caching system, to see if write caching also affects or is affected by "filtering", the COMB and LRU policies are studied, not only on the client but also on the server. LRU purging with thresholds is studied on both the clients and the server. Write-back with thresholds is also studied on both the clients and the server. The purpose of piggybacking is to optimize disk accesses. If the client in a multi-level caching system is diskless, it makes sense to do piggyback only on servers. Therefore, piggybacking for WBT is simulated in the single-level caching system as well as on the server of the multi-level caching system. For the multi-level caching system, coordinated purge is studied to see if it is reasonable to coordinate the purge operations on the clients and the purge operations on the server.

Figure 4.3 shows the combinations of the write policies that were simulated at the client and the server in the multi-level caching system. During the simulation of COMB, LRU, LRUPT, and WBT at the clients and COMB, LRU, LRUPT, WBT, and WBT-PB at the server, the effects of trying to coordinate the purge operations at the server and the client were also evaluated in an attempt to determine whether or not CPT is a realistic NV write caching policy.

### 4.2.5 Performance Metrics

The purpose of the research is to evaluate the performance of different write caching policies for NV cache in a single-level caching system and NV caches on the server and client in a multi-level caching system. For each policy, it is important to know how much of the write traffic (from the NV cache to the disk in a single-level caching system, or from the client to the server or from the server to the disk in a multi-

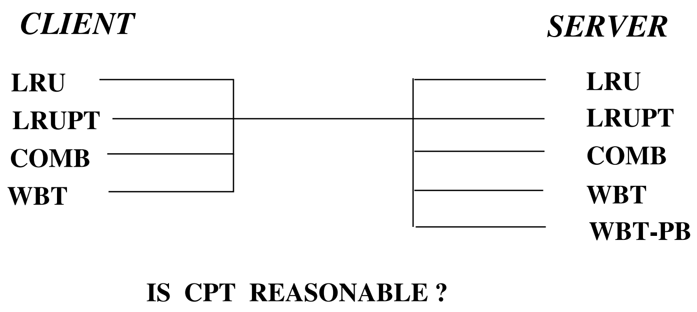



Figure 4.3: The Write Caching Policies Simulated at the Client and Server

level caching system) can be absorbed by the NV cache, and how many reads that miss in the volatile read cache can be satisfied by blocks saved in the NV cache. A set of performance metrics were devised to evaluate the effectiveness of the proposed approaches. In the following description of the metrics, an *application write* refers to a write operation generated by an application, a *write back* refers to an operation that actually goes from the NV cache to the disk in a single-level caching system, or from the client's cache to the server's cache through the network, or from the server's cache to the disk, in a multi-level caching system, and a *write miss* refers to a write request for which the dirty block the request wants to write on is not in the NV cache (in a single-level caching system, and/or at either client or server in a multi-level caching system).

The following measures were used to evaluate performance:

1. Write backs generated, expressed as a percentage of the total number of writes. A well-performing policy writes fewer blocks back to disk.

$$Write\ Backs = \frac{Blocks\ Written\ to\ Disk}{Number\ of\ Write\ Operations} \times 100\%$$

2. Dirty blocks that are overwritten while still in the NV cache, expressed as a percentage of the total number of writes. The more dirty blocks overwritten in

the NV cache before being flushed, the better the policy works.

$$Overwritten = \frac{Write\ Hits\ to\ Dirty\ Blocks}{Number\ of\ Write\ Operations} \times 100\%$$

3. Disk track accesses of write operations, expressed as a percentage of the total number of writes. This measure reflects how effective a particular write caching policy is in reducing the number of disk I/Os. The fewer disk track accesses for write operations, the more effective a purging policy is.

$$Disk\ Track\ Accesses = \frac{Disk\ Track\ Accesses\ of\ Writes}{Number\ of\ Write\ Operations} \times 100\%$$

4. Blocks in the NV cache can be removed (invalidated) before being written back. Dirty blocks that are invalidated when in the NV cache before being written back to disk, expressed as a percentage of the total number of write operations. This measure shows the fraction of dirty blocks that are removed before being flushed. More of the write back cost is saved when this measure is higher.

$$Invalidated\ Within\ NV\ Cache = \frac{Blocks\ Invalidated\ in\ NV\ Cache}{Number\ of\ Write\ Operations} \times 100\%$$

5. Read hits in the volatile cache, expressed as a percentage of the total number of reads. The higher this read hit ratio is, the better the caching policy works for read requests.

$$Read\ Hits\ in\ V\ Cache = \frac{Total\ Read\ Hits\ in\ V\ Cache}{Number\ of\ Read\ Operations} \times 100\%$$

6. Read hits in the NV cache, expressed as a percentage of the total number of reads. This measure reflects how effective the NV cache is for read requests. When this ratio is high, more read requests are being satisfied by the NV cache.

$$Read\ Hits\ in\ NV\ Cache = \frac{Total\ Read\ Hits\ in\ NV\ Cache}{Number\ of\ Read\ Operations} \times 100\%$$

7. The number of blocks at the client in the same disk track/cylinder as the blocks purged from the server's NV cache, expressed as a percentage of the total number of write backs. This measure indicates the desirability of coordinating the purging operations on the server and the client in a multi-level caching system. The more candidates for CPT, the more important CPT is.

$$Candidate\ for\ Coordinated\ Purge =$$

$$\frac{\sum_{all\ purges} Blocks\ at\ Client\ on\ the\ Same\ Disk\ Track}{Total\ Number\ of\ Writes\ to\ Disk}$$

8. The percentage of writes that are piggybacked with read misses successfully. The higher this measure is, the fewer write backs are needed, and the better the policy performs.

$$Writes\ Piggybacked = \frac{Total\ Writes\ Piggybacked}{Total\ Number\ of\ Write\ Operations} \times 100\%$$

9. The percentage of reads that can be used for piggybacking. When there are more reads that can piggyback writes, more dirty blocks can be written back together with a read miss, and more disk accesses can be saved as a result.

$$Reads\ that\ can\ have\ Writes\ Piggybacked =$$

$$\frac{Read\ Misses\ that\ can\ be\ Piggybacked}{Number\ of\ Read\ Operations} \times 100\%$$

Several other measures are included to show the overall performance effects of a multi-level caching system. These are calculated as follows:

$$Overall\ Write\ Backs = \frac{Write\ Backs\ from\ Server}{Number\ of\ Write\ Operations} \times 100\%$$

$$Overall\ Dirty\ Blocks\ Overwritten =$$

$$\frac{Write\ Hits\ on\ Dirty\ Blocks\ at\ the\ Server}{Number\ of\ Write\ Operations} \times 100\%$$

$$Overall\ Disk\ Track\ Accesses = \frac{Disk\ Track\ Accesses\ of\ Writes\ on\ Server}{Number\ of\ Write\ Operations} \times 100\%$$

$$Overall\ Invalidation\ Within\ NV\ Cache =$$

$$\frac{Blocks\ Removed\ from\ NV\ on\ Server\ +\ Blocks\ Removed\ from\ NV\ on\ Client}{Number\ of\ Write\ Operations} \times 100\%$$

$$Overall\ Read\ Hits\ in\ V\ Cache =$$

$$\frac{Read\ Hits\ in\ V\ Cache\ at\ Server\ +\ Read\ Hits\ in\ V\ Cache\ at\ Client}{Number\ of\ Read\ Operations} \times 100\%$$

$$Overall\ Read\ Hits\ in\ NV\ Cache =$$

$$\frac{Read\ Hits\ in\ NV\ Cache\ at\ Server\ +\ Read\ Hits\ in\ NV\ Cache\ at\ Client}{Number\ of\ Read\ Operations} \times 100\%$$

## 4.3 The Traces

The experiments were done by trace-driven simulation, and so the traces used are an important part of the research. The process of event tracing is discussed in this section, then the system traced is described, and finally the methodology of trace collection is outlined.

### 4.3.1 Events Recorded in the Traces

A trace is a time-ordered record of events on a real system [Jain 91]. For this study, all file I/O events that return information about file system state or change the file system state should be captured in the traces.

In a UNIX file system, the buffer cache is manipulated by a group of buffer allocation system calls, such as *getblk, brelse, bread, breada, b2reada* and *bwrite* [Bach 86]. Getblk allocates buffers from the buffer cache for reading and writing disk blocks. Brelse releases the buffer when the kernel finishes using it. To read a particular disk block, a process uses getblk to search for it in the buffer cache. If the required block is found in the cache, the kernel can return it immediately without physically reading it from the disk. Breada/b2reada reads one/two block(s) ahead into the cache. It also uses getblk to search the buffer cache for the required block. Bwrite is used to write the contents of a buffer to the disk. A set of system calls known as the lower level file system calls, which include *namei, iget, iput, ialloc, ifree, alloc, free,* and *bmap*, are built on top of the buffer allocation system calls, while file system calls that return file descriptors (*open, creat, dup, pipe,* and *close*), assign i-nodes (*creat, mknod, link,* and *unlink*), change file attributes (*chown, chmod,* and *stat*), do file I/O (*read, write,* and *lseek*), and build the file system structure (*mount,* and *umount*) are built on top of the lower level file system calls. This hierarchical structure of the UNIX file system calls is summarized in Table 4.1 [Bach 86].

Table 4.1: A Classification of File System Calls [Bach 86]

<table>
<tr><th>Return File Desc</th><th>Assign I-Nodes</th><th>File Attributes</th><th>File I/O</th><th>File Sys Structures</th></tr>
<tr><td>open<br>creat<br>dup<br>pipe<br>close</td><td>creat<br>mknod<br>link<br>unlink</td><td>chown<br>chmod<br>stat</td><td>read<br>write<br>lseek</td><td>mount<br>umount</td></tr>
<tr><td colspan="5">LOWER LEVEL FILE SYSTEM CALLS</td></tr>
<tr><td colspan="3">namei</td><td rowspan="2">ialloc<br>ifree</td><td rowspan="2">alloc free<br>bmap</td></tr>
<tr><td colspan="3">iget iput</td></tr>
<tr><td colspan="5">BUFFER ALLOCATION SYSTEM CALLS</td></tr>
<tr><td colspan="5">bread breada bwrite<br>getblk brelse</td></tr>
</table>

For the requirements of this thesis work, tracing the operation of "getblk" is sufficient since getblk is used by almost all other file manipulation system calls. Tracing the event "unlink", the system call in UNIX that removes a directory entry for a file, is also necessary to provide the simulator with enough information to invalidate the appropriate blocks in the caches when the file the block belongs to is removed.

### 4.3.2 The System Traced

Traces were gathered from the DISCUS laboratory in the Department of Computational Science at the University of Saskatchewan. The major components of the laboratory are shown in Figure 4.4. At the centre of the laboratory is a cluster of four

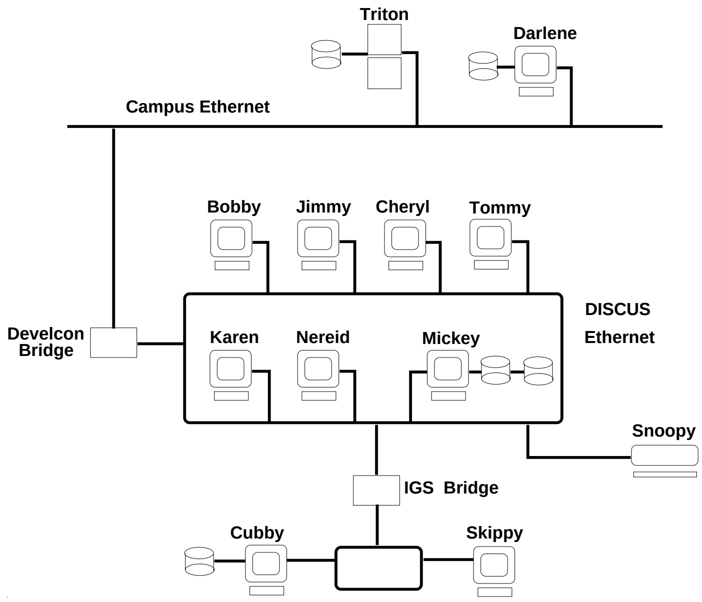



Figure 4.4: The DISCUS Experimental Laboratory

HP/APOLLO 9000/4xx series workstations (Bobby, Karen, Tommy, Mickey), two 9000/3xx series workstations (Cheryl, Jimmy), and one Silicon Graphics workstation (Nereid) connected by means of Ethernet. Triton is a 8-way SGI multi-processor used for research in parallel systems and Snoopy is a Hewlett Packard LAN protocol analyzer used for network measurement. The HP DISCUS cluster runs under the HP-UX 8.0 operating system (Hewlett Packard's UNIX variant).

The main file server for the DISCUS laboratory (Mickey) is on of the series 400 machines. It distributes over 1.2 GBytes of disk storage space to the other DISCUS machines. Access to operating system source code allows people in the DISCUS group to gather detailed traces of system events by placing measurement points strategically throughout the kernel. A set of traces collected for an earlier study [Willick *et al.* 93] was used for this research.

Two disks are connected to Mickey through the SCSI (Small Computer System Interface) interface. The descriptions of the two SCSI disks are given in Table 4.2 (information stored in file "disktab" plus information obtained by executing 'diskinfo

-v /dev+/localroot/rdsk/0s0 or 1s0').

Table 4.2: Disk Information for the DISCUS Lab

| Disks | 1s0 | 0s0 |
|---:|---:|---:|
| vendor : | FUJITSU | CDC |
| product id : | M2694ES-512 | 94191-15 |
| type : | direct access | direct access |
| size (byte) : | 1083916800 | 663475200 |
| size of physical sector (byte) : | 512 | 512 |
| space reserved for swap : | | 175 mbytes |
| number of 1 kbyte sectors per track : | 38 | 15 |
| number of tracks per cylinder : | 15 | 15 |
| number of cylinders per disk : | 1857 | 1156 |
| size of file system (kbyte) : | 1058490 | 468180 |
| size of file system block (byte) : | 8192 | 8192 |
| size of fragment (byte) : | 1024 | 1024 |
| rotational speed of platters (rpm) : | 5400 | 3600 |

Disks contain a *mechanism* and a *controller*. The mechanism is made up of the recording components (the rotating disks, the heads that read from and write to them), and the positioning components (an arm assembly that moves the heads into the correct position together with a servo system that keeps it in place). The controller manages the storage and retrieval of data to and from the mechanism, and performs mappings between incoming logical addresses and the physical disk sectors that store the information.

SCSI works by masking the internal structure of the peripherals from its host computer. A SCSI disk appears to its host computer as a linear vector of addressable blocks – device logical blocks. The address of the first device logical block number is zero, and the address of the last device logical block is N-1, where N is the number of device logical blocks available on the medium [NCR 90, GED 90]. At the server of a distributed file system with SCSI disks, a logical block number must be first converted into a device logical block number in the logical geometry representing the SCSI disk. The device logical blocks have to be in turn mapped to physical sectors (the fixed-size data layout units on the medium itself) by the disk controller

[Ruemmler & Wilkes 93b]. Blocks of data in a SCSI disk are stored on the medium along with additional information that the controller uses to manage the storage and retrieval. This additional information is used to identify the physical location of the blocks of data, the address of the device logical block, and to provide protection against the loss of user data.

It must be noted that there is no NV cache in the traced system. In a NFS-based distributed system using a write-through policy with no NV cache, all writes go to disk immediately. In the traces obtained on such a distributed system, each write request thus corresponds to a physical/logical server disk space address that is allocated for it. In contrast, in a distributed system with NV cache on both the server and the client, space on the server's disk may never be allocated for some of the write requests from a client, since some of the newly written blocks on the client's NV cache may be overwritten or deleted before being purged to the server's disk. Therefore, the disk layout seen by systems with and without NV cache may be different. For this reason, it may not be accurate to use the disk block addresses from traces collected on a system with no NV cache in a study of the write policies of NV caches.

### 4.3.3 Trace Collection and Processing

This section describes the traces that are used for this research and the methodology by which they were collected.

For the previous work of Willick *et al.* [Willick *et al.* 93], a number of traces were collected from the DISCUS laboratory, with the trace point placed prior to the client's cache primarily to record the file I/O traffic into the client's cache. The type of activity traced ranged from the execution of large file-intensive graphics/image processing applications, to program editing/debugging/compiling, to large compilations (a kernel compilation involving over 450 source files comprising more than 300,000 lines of C source code). Since this set of traces represents typical file access activity within a university research environment, traces from the set captured by Willick *et al.* [Willick *et al.* 93] formed the basis for the experiments in this thesis. Three of these traces, with the most events recorded and the most write operations, were chosen (the second and third traces were used in the experiments reported in [Willick *et al.* 93]). The characteristics of the three traces are summarized in Table 4.3.

Table 4.3: Trace Characteristics

| Trace | hours | # reads | # writes | # removes | Dominant Activity |
|---|---|---|---|---|---|
| 1 | 3.23 | 738489 | 8547 | 16 | trace-driven simulation runs |
| 2 | 1.69 | 11304 | 5498 | 383 | program development (edit,compile,debug) |
| 3 | 3.81 | 111947 | 31659 | 2981 | Kernel compiling from source code |

In preparation for the previous study [Willick *et al.* 93], two new trace types, GETBLK, and VNSREMOVE, were added to the kernel of the HP-UX 8.0 operating system running on the cluster of HPs in the DISCUS laboratory. The GETBLK trace type placed hooks into the source code so that flags can be set to determine if a reference is a read or write, in *getblk()* and *ogetblk()* routines. In the system's processing of a read request, one of the functions *bread, breada*, or *b2reada* is called, which in turn calls *getblk* to look for the requested block in the cache. For a write request, the kernel determines whether to write the entire block or only part of it. If it writes only part of a block, it must first read the block from disk so as not to overwrite the parts that will remain the same, but if it writes the whole block, it need not read the block, since it will overwrite the previous contents anyway [Bach 86]. Identification of files and storage devices, i-node pointers, and block numbers can be recorded by the GETBLK trace type.

Special attention had to be paid to the simulation of the situations where an i-node is reassigned to another file after the file it represented was deleted. A special trace type (VNSREMOVE) was added to appropriately invalidate the cached i-nodes that have been reassigned. To support this trace type, hooks were placed in *vns_remove*() and *vn_remove*() to gather information on files being removed. Since the client trace of unlink events does not contain i-node information, traces have to be collected from both the client and the server, and the information needed must be extracted by matching the unlink event on the client with the corresponding event record in the server trace. Should the client and the server be on the same machine, such as Mickey, the matching of unlink activities on the server and the client is not necessary, because complete information will be recorded for the client.

Each trace record contains a description of the file I/O activities generated by the routines traced, together with a time stamp and detailed information about the process identifier, the offset of the logical block from which the required data begins, block size, the flag for read/write on the block, the identifier of the I/O device, the identifier of the file referenced, and the identifier of the file system.

For this research, in addition to the information recorded in the traces, it is also necessary to know the physical disk address for each block referenced by the server or the clients in order to optimize the purge operations and to do the piggybacking. It was not possible to record this information directly, so a disk model was constructed to add logical disk addresses to the block referenced in each trace record. In this model, it is assumed that files are laid out contiguously on disk. If there is a small number of large files and indirect blocks are rarely used, files on disk are usually stored contiguously. The logical disk address of every i-node on a disk can be seen by reading the ./ file of each directory on the disk. To construct the disk model, the trace is first read once in its entirety in order to build a list of all the files referenced in the trace, with a sublist of all the blocks belonging to each i-node for each file. The list of files and the sublist for blocks in each file are sorted in increasing order. Assuming the first block in the first file encountered in the trace has the logical disk address of 0, and the second block has the logical disk address of 1, and so forth, the logical disk address for each block in the list can be obtained simply by counting down along the list. The trace is then read a second time to obtain the logical disk address for each block encountered by searching the file list for the particular block referenced. The physical disk address (in terms of disk track/cylinder of the block) is then calculated from each block's logical disk address by using known parameters of the device (such as the size of the disk track). This information is given in Table 4.2.

## 4.4 The Simulation Experiments

This section describes the operation of the simulator, as well as the factors of the simulation experiment and their levels.

The simulator was implemented in C. The basic requirements are to read in the processed trace, simulate the cache management operations that result from by read or write requests in the trace for different caching policies, maintain necessary performance measures, and output the results. In the implemented program, trace records were read in one at a time. For each input record, the first step is to calculate the physical disk address corresponding to the referenced block, according to the disk model (described in Section 4.3.3). Then, the cache operations on the client (and possibly on the server as well, if the reference missed on the client), are performed according to the caching model simulated, in the manner described in Section 4.2.2 and Section 4.2.3. If the trace record is a remove event, all the blocks that belong to the file being removed cached in either the volatile cache and/or the non-volatile cache are removed from their respective cache(s). The resulting performance measures were graphed using “gnuplot” (an interactive plotting program).

Each simulation run (or experimental session) consists of a particular combination of system configuration, caching policies, and an input trace. The system configuration is either a single-level caching system or a multi-level caching system. A single-level caching system was simulated to see what happens in a standalone environment where a station has its own local disk. Both read and write caching are simulated in a single-level caching system. A multi-level caching system, in which one diskless client is connected to one server with a disk through a network with read and write caching at both the server and the client, was also simulated to investigate how caching at the client and at the server affect each other. The volatile read caches and the non-volatile write caches with certain sizes are placed at the server and the client in multi-level caching systems. The combinations of NV caching policies considered are shown in Figure 4.3, while the traces used are described in Table 4.3. Read caches are always managed by LRU.

The factors of an experiment are the variables that affect the performance metrics. The values taken by factors during an experiment are referred to as levels. One of the most important factors in the simulation experiment is the write policy for the NV cache. For the single-level caching system, five levels were considered for the caching policy factor: LRU, LRUPT, COMB, LRUPT, and WBT (with/without piggybacking). For the multi-level caching system, four levels of policies (LRU, LRUPT, COMB, and WBT) were considered on the client, and five levels (LRU, LRUPT, COMB, LRUPT, and WBT (with/without piggybacking)) were considered on the server. The size of the volatile caches in the single-level caching system and on the client and the server of the multi-level caching system is always 200 blocks, this is chosen based on a previous study [Willick *et al.* 93]. The size of the NV cache is also a factor. For the single-level caching system, seven levels were considered for the NV cache size (10, 30, 50, 70, 100, 150, and 200 blocks). In the multi-level caching system, four levels were considered for the NV cache size at the client (0, 10, 50, and 100 blocks), and seven levels were considered for the NV cache size at the server (10, 30, 50, 70, 100, 150, and 200 blocks). The NV cache size is chosen at most to be the same as the volatile cache size, since it may not be realistic to have a larger NV cache.

## 4.5 Summary

Issues relating to the trace-driven simulation experiments have been presented in this chapter. These include the definition of the caching policies considered, the methods and procedures employed for trace collection and processing, the definition of the measures used to evaluate performance, and the design of the experiment.

Simulations were done of a standalone environment with one level of disk caching (single-level caching) and of a client-server environment with two levels of caching (multil-level caching). Nine metrics are used to compare the performance of five policies in a single-level caching system, as well as the performance of four write caching policies on the client and five write caching policies on the server in the multi-level caching system. Traces used in this work were taken from a set of traces collected for a previous study of read caching, and a disk model is used to add logical disk addresses for the blocks referenced in the traces. The simulation runs, the experimental factors, and the levels of each factor, were also described.

# Chapter 5

# Experimental Results

This chapter presents the results of the trace-driven simulation experiments that were performed to study the performance of non-volatile caches added on the client and the server. Section 5.1 presents and compares the performance of NV write caching policies with single-level caching. Section 5.2 presents and compares the performance of NV write caching policies with multi-level (client/server) caching. Section 5.3 studies the impact of sizes of client and server NV caches on performance, in the multi-level environment. Section 5.4 discusses the coordination of non-volatile write caching on the server and the client. Section 5.5 summarizes the major results from the experiments.

The experimental results shown in this chapter are mainly from trace 1; results for trace 2 and trace 3 can be found in Appendices B and C respectively. Results from trace 1 are more interesting because the performance differences among the policies are more clearly seen. Trace 1 (see table 4.3) is the longest of the traces in terms of the total number of references (over five times the number of references of the next largest trace), it also possesses more interesting temporal locality characteristics. Although trace 3 has more recorded write operations, a large fraction of this write activity appears to be in the form of partial writes directed to the same blocks. (For partial writes, the whole block is read first, modifications to part of that block is added, and the block is written back to the disk.) There are small performance differences between the performance of different write caching policies in the results from trace

2 and 3, as shown in Appendix B and C respectively.

For all the single-level caching simulation runs, the size of the volatile cache is 200 blocks and the disk track is the basis of purging. (In file-based purging, all the dirty blocks in the NV cache belonging to the same file chosen to be purged are purged together. In track-based purging, all the dirty blocks belonging to the same disk track as the block chosen to be purged are purged together). For all the multi-level simulation runs, the volatile cache size for both the client and server is 200 blocks, the file is the basis of purging on the client, and the disk track is the basis of purging on the server. Figure 5.1 shows the disk track access expressed as a percentage of total writes from trace 1 for the three purging policies (LRUPT, WBT, and WBT-PB), it can be seen that the thresholds for purging operations are reasonable to be chosen to be 95% for the high limit, and 65% for the low limit.

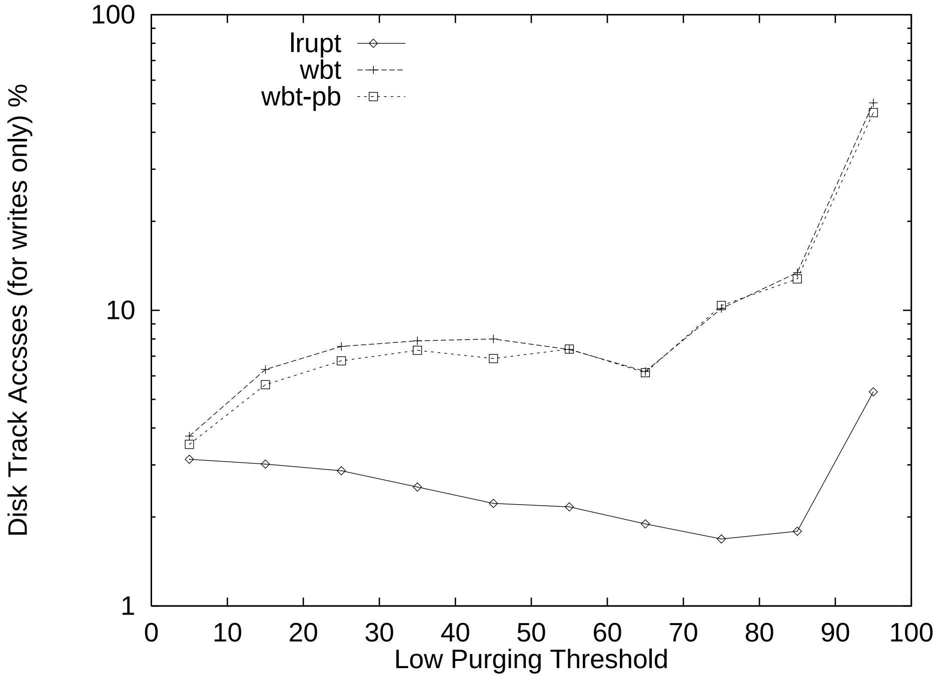



Figure 5.1: Disk Track Accesses by Write Operations as a Function of Low Limit Threshold (from Trace 1 with Single-Level Write Caching)

## 5.1 Single-level NV Write Caching

In this section, simulation results for single-level NV write caching using the five candidate policies (LRU, LRUPT, COMB, WBT, or WBT-PB) are presented. The size of the NV cache varies from 10 to 200 blocks in the experiments.

When blocks of one disk track are written to disk together, only one physical disk access is necessary. Thus, an important measure of performance is the number of required disk track accesses under each policy. Figure 5.2 shows the number of disk track accesses performed for write operations, for each policy, expressed as a percentage of the total number of write operations in the trace. The number of disk track accesses is presented in a log scale to show clearly the different performance of different policies when the number of disk access is small. As can be seen, LRUPT

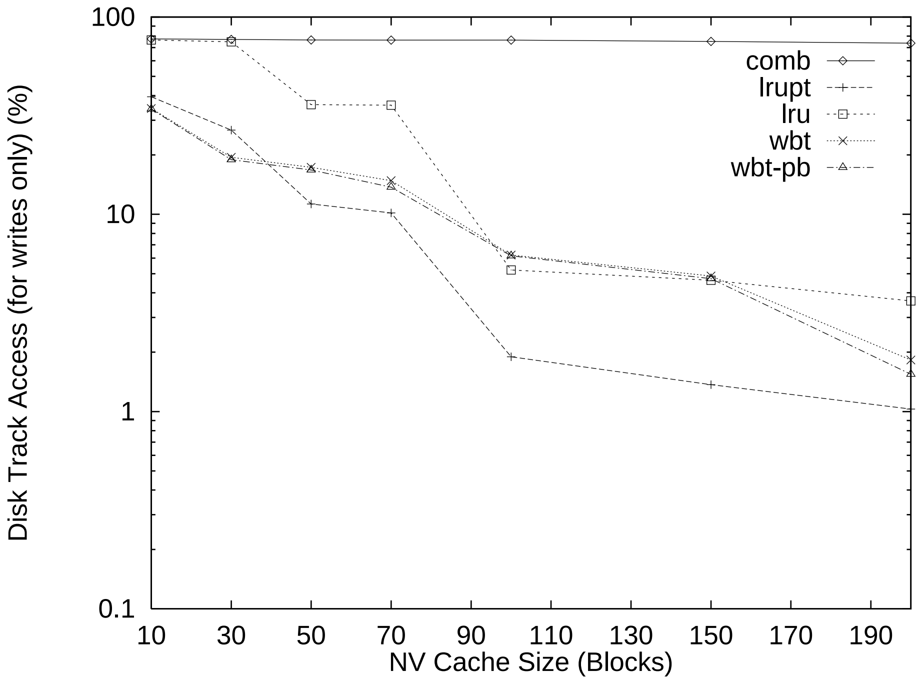



Figure 5.2: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations (Single-level NV Cache Configuration)

requires the fewest disk track accesses for write operations, when the NV cache size is greater than or equal to 50 blocks. LRUPT is a temporal locality focused policy that takes advantage of temporal locality when the NV cache size is large enough to hold the whole working set of dirty blocks. LRUPT also amortizes the cost of disk track accesses over many dirty blocks written by purging all the dirty blocks in the same disk track as the LRU-end block of the LRU list. For single-level NV caching, LRUPT performs better than the other policies for many reasons. With WBT and WBT-PT there are more disk track accesses than with LRUPT, but fewer than with LRU and COMB, when the NV cache size is greater than 40 blocks and smaller than 100 blocks. WBT and WBT-PB are designed to focus on the amortization of disk accesses over block writes as much as possible. The existence of temporal locality does not affect the performance of WBT and WBT-PB much. This may be the reason why WBT and WBT-PB do not perform as well as LRUPT in a certain range of NV cache size. When the NV cache size is in the range of 100 to 150 blocks, the performance of WBT, WBT-PB, and LRU is almost the same. When the NV cache size is greater than 150, more dirty blocks are available for purging, and WBT and WBT-PB perform better than LRU and close to LRUPT. COMB performs uniformly poorly; increasing the NV cache size has little effect on its performance. Evidently, the non-volatile cache is mostly serving the read traffic under this policy. (See the later discussions on read hit ratios shown in Figure 5.5.)

Results from trace 3 are similar in nature, except that LRU shows much less improvement as the NV cache size increases. (See Figure C.1 in Appendix C.) Evidently owing to different file reference characteristics, results from trace 2 indicate that WBT or WBT-PB is the best policy for single-level NV caching, in terms of minimizing the number of disk track accesses for writes. (See Figure B.1 in Appendix B).

Figure 5.3 shows the number of blocks that are actually written to the disk from the NV cache, expressed as a percentage of the total number of write operations, for each of the five write caching policies. The performance in terms of this measure seems to divide into three groups for all but very small caches ($\leq$ 30 blocks), with one group containing LRU and LRUPT, one containing WBT and WBT-PB, and one containing COMB. The policies in each performance group have a similar nature: LRU and LRUPT are both temporal locality focused, while WBT and WBT-PB are

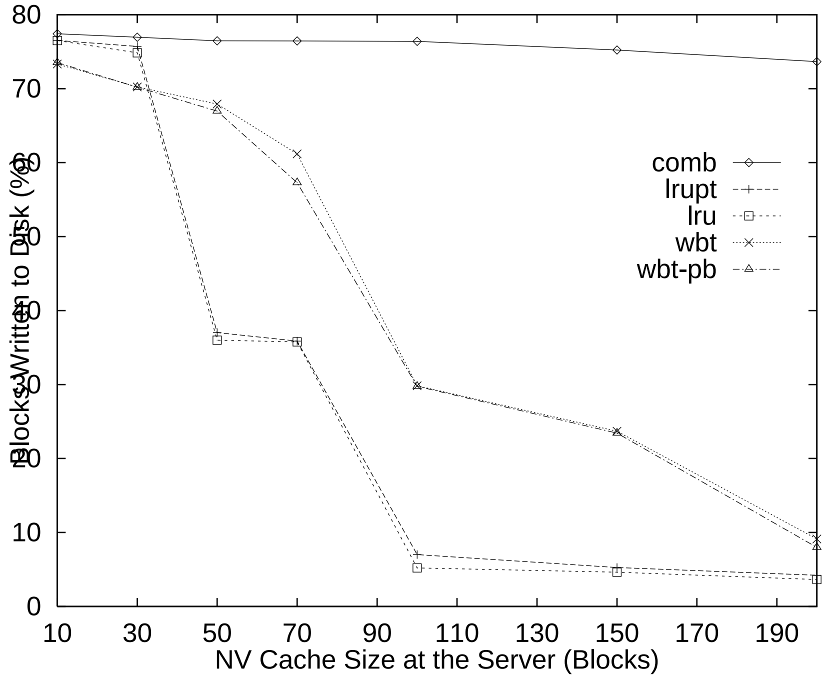



Figure 5.3: Blocks Written to Disk as a Percentage of the Total Write Operations (Single-level NV Cache Configuration)

both focused on amortizing the cost of a track access over many block writes. COMB is special itself, because clean blocks can reside in the NV cache for read requests. Comparing the LRUPT group with the other groups, LRUPT and LRU seem to be the best policies in this respect when the NV cache size is greater than 30 blocks. In fact, throughout the entire range of NV cache sizes examined, LRU is slightly better than LRUPT under this performance measure. WBT and WBT-PB are the second best to LRU and LRUPT, and they are very similar to each other (piggybacking evidently has only a very small effect on performance). COMB performs the worst. Note that the performance of COMB does not improve much even for large NV cache sizes, while the number of blocks written to disk drops from more than 70% of the total number of write operations down to less than 10% as the NV cache size increases from 10 to 200 blocks for the other four write caching policies. When the NV cache size is larger than 100 blocks, the performance of LRU and LRUPT does not change much, but that of WBT and WBT-PB continues to improve. Keeping in mind that more than one dirty block can be written back to disk in one physical disk access when purging is used, the purging policies perform dramatically better (relatively) in Figure 5.2 compared to Figure 5.3.

Figure 5.4 shows the average number of dirty blocks that are written back to disk

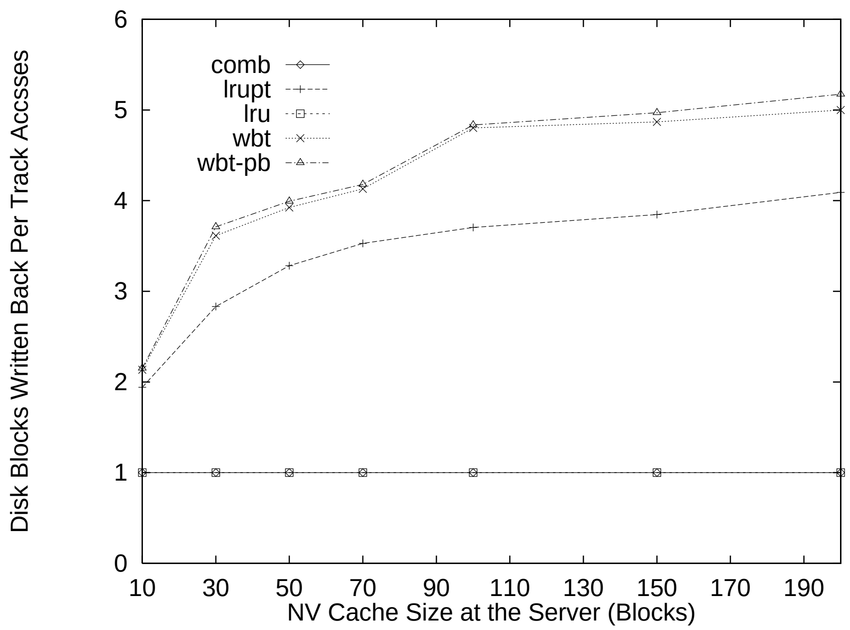



Figure 5.4: Average Number of Dirty Blocks Written Back to Disk in One Disk Track Access

in one disk access. For LRU and COMB, each disk access can write only one dirty block. For LRUPT, WBT, or WBT-PB, however, the number of dirty blocks that can be written back to disk in one disk track access is greater than one, and the number increases as the NV cache size increases. WBT and WBT-PB purge more dirty blocks per disk track access than LRUPT.

Figure 5.5 shows the number of dirty blocks overwritten within the NV cache (i.e., blocks that are written to again before they are written out to disk), expressed as a percentage of the total number of write operations. The policies divide into three groups again. LRUPT and LRU have the most dirty blocks overwritten within the NV cache, when the NV cache size is greater than 30 blocks. As the cache size increases, a very large percentage of blocks are overwritten while in the cache. Again,

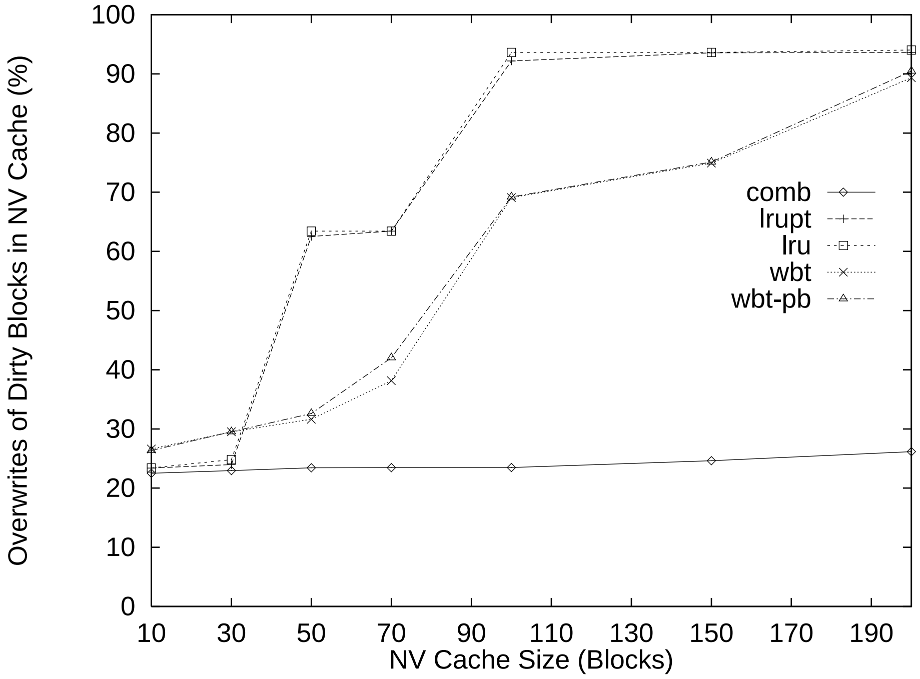



Figure 5.5: Overwrites of Dirty Blocks as a Percentage of the Total Write Operations (Single-level NV Cache Configuration)

LRU performs slightly better than LRUPT. The next best are WBT and WBT-PB; WBT-PB is again very similar to WBT, indicating that piggybacking does not help much. COMB is again the worst, with the fewest dirty blocks overwritten. Again, increasing the NV cache size does not significantly improve the performance of this policy.

The percentage of blocks written to disk should equal 100% less the sum of the percentage of dirty blocks overwritten and the percentage of dirty blocks deleted. There is no significant number of dirty blocks that are deleted from the NV cache before being written out to disk, because there is only a very small number of remove operations recorded in trace 1 (the number of remove operations evidently affects relatively few blocks, as shown in Figure 5.6). Note that traces 2 and 3 do not show such an extreme behaviour (as shown in Figures B.5 and C.5).

The read hit ratio at the volatile cache does not change when the NV cache size is increased. Table 5.1 shows the read hit ratio at the volatile cache of the three traces.

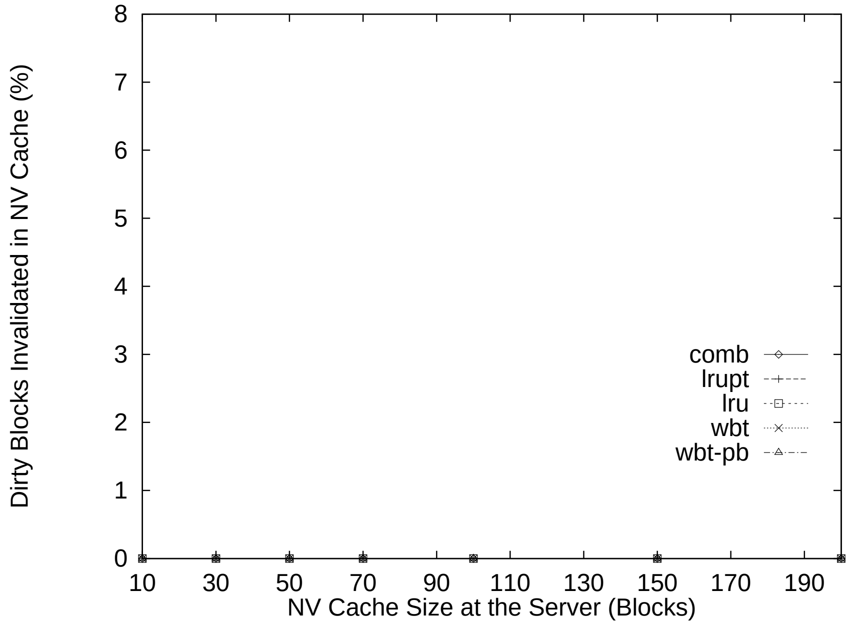



Figure 5.6: Dirty Blocks Deleted from NV Cache as a Percentage of the Total Write Operations (Single-Level NV Caching Configuration)

From the results of trace 1, for LRU, LRUPT, WBT, and WBT-PB, the read hit

Table 5.1: Read Hit Ratio at the Volatile Cache

| Trace | LRU | LRUPT | WBT | WBT-PB | COMB |
|---|---|---|---|---|---|
| 1 | 90.71 | 90.71 | 90.71 | 90.71 | 3.59 |
| 2 | 56.69 | 56.29 | 56.29 | 56.29 | 26.79 |
| 3 | 71.33 | 71.33 | 71.05 | 71.05 | 31.98 |

ratio at the volatile cache is about 91%, but only about 3% for COMB. (Note that results from traces 2 and 3 are different from those of trace 1.) On the other hand, Figure 5.7 shows that the read hit ratio at the non-volatile cache for COMB is as high as 90% when the NV cache size is greater than 30 blocks, while the read hit ratio at the NV cache for the other polices is very low. COMB uses the NV cache for read caching as well as for write caching. Note in particular that newly read blocks may be placed in the NV cache, and that blocks that experience read hits while in the NV cache (including, for example, blocks of newly written files) are not copied

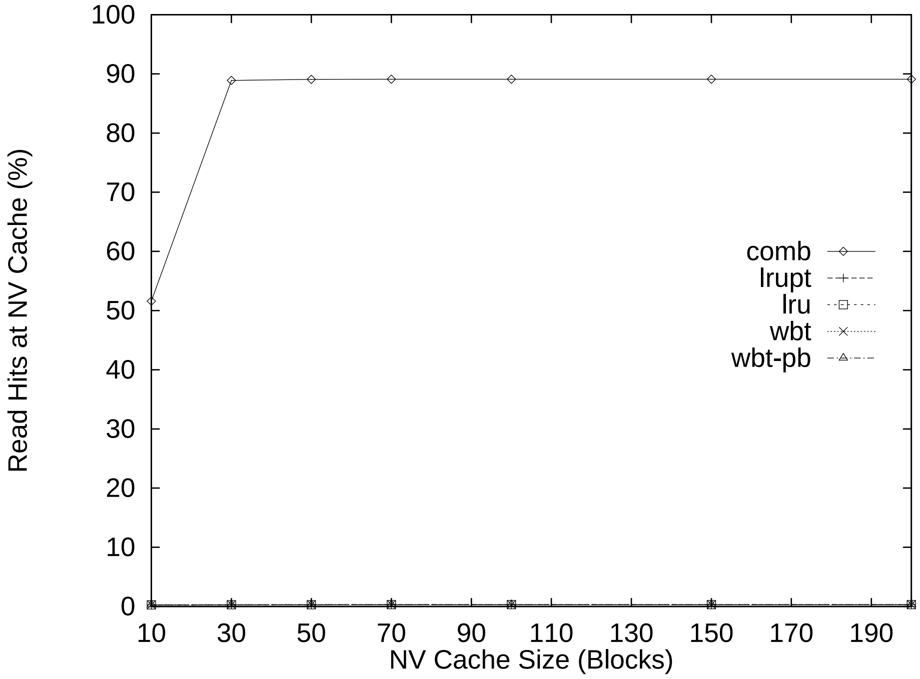



Figure 5.7: Read Hit Ratio at the NV Cache (Single-level NV Cache Configuration)

over to the volatile cache as in the other policies (in fact, such blocks are moved to the MRU end of the list of blocks, implying that they are unlikely to be replaced in the near future).

To investigate the extent to which piggybacking can improve write caching performance, Figure 5.8 and Figure 5.9 show the percentage of read operations that have writes piggybacked under the WBT-PB policy, and the percentage of writes piggybacked. In this case, only a very small fraction of reads has piggybacked writes, and only a small percentage of writes are piggybacked.

The preceding discussion suggests that LRUPT is the best write caching policy to use when there is only one level of write caching. Some conclusions about the behaviour of the policies are as follows:

⇝ LRUPT

When the disk track is used as the basis for purging, LRUPT writes all the dirty blocks that are in the same disk track as the block at the LRU-end of the NV LRU list in each purging operation. LRUPT makes use of temporal locality as

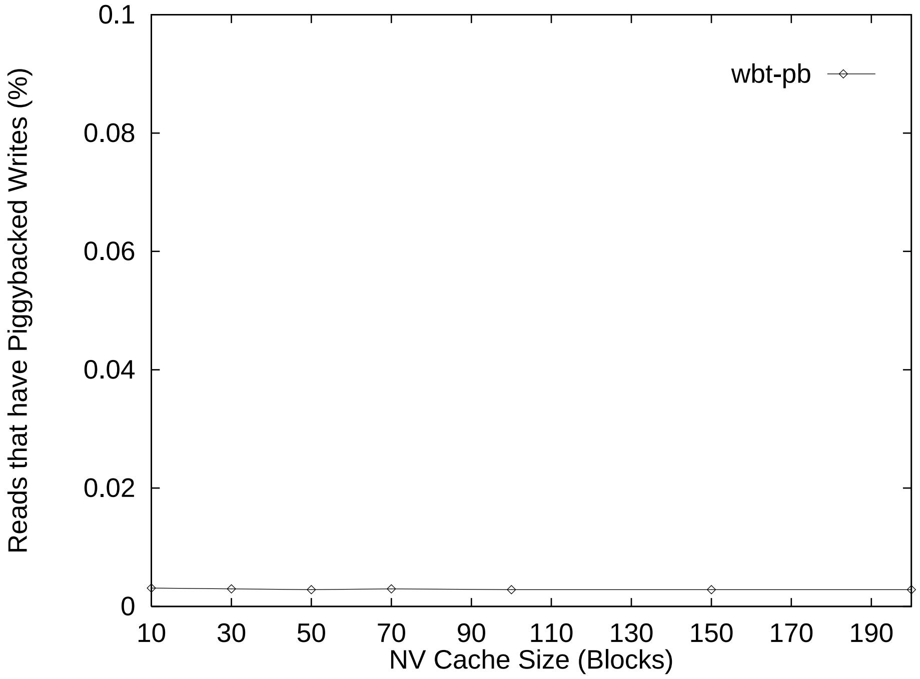



Figure 5.8: Read Misses that have Writes Piggybacked (Single-level NV Cache Configuration)

much as possible by purging in LRU order, so that there is a high percentage of overwrites of dirty blocks in the NV cache, and a small percentage of blocks written to disk. In addition, since purging is track-based, few disk track accesses are incurred.

⇝ LRU

LRU is less inclined to write dirty blocks back to disk than is LRUPT, since LRU does not purge, and therefore, will leave dirty blocks in the cache as long as possible, fully utilizing the entire NV cache. In practice, however, LRU appears to behave quite similarly to LRUPT in terms of the number of blocks written to disk. LRU does incur more disk track accesses than LRUPT, since only one dirty block is written back at a time instead of all the dirty blocks in one disk track. When the NV cache size gets large enough to retain the whole working set, LRU should present better performance.

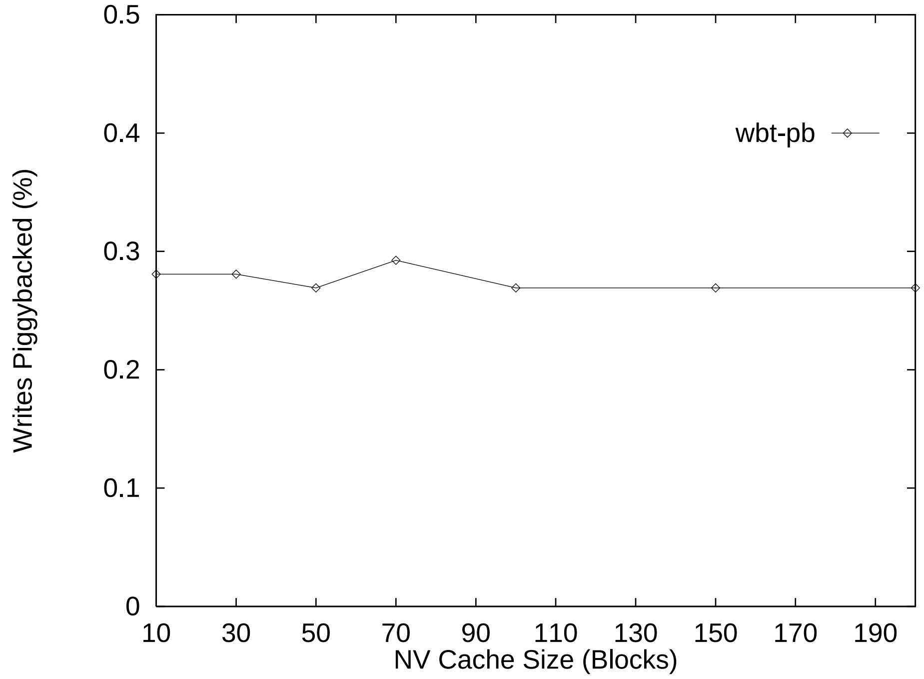



Figure 5.9: Writes Piggybacked (Single-level NV Cache Configuration)

⇝ WBT and WBT-PB

WBT and WBT-PB write back to disk, in each purging operation, all the dirty blocks that reside on the disk track that has the most dirty blocks. Temporal locality may not be respected, since more-recently-used blocks may be purged. Thus, even though WBT and WBT-PB tend to do better at amortizing the cost of a track access over many block writes, it appears that WBT and WBT-PB perform slightly worse than LRUPT in this single-level NV cache context, even in terms of the number of disk accesses. (They perform much worse in terms of the number of block writes.)

Piggybacking decreases the number of disk track accesses because dirty blocks piggybacked by a read miss can be written back to disk by the disk operation for that read miss. It is possible for piggybacking to make purging sometimes even unnecessary, since the high threshold may be reached much less often because some of the dirty blocks in the NV cache have already been written back by piggybacking. Therefore, piggybacking can potentially improve the performance of WBT significantly. The results obtained here, however, as shown in Figure 5.8 and Figure 5.9, do not show much benefit for piggybacking because of the small percentage (about 0.3%) of piggybacked writes. In fact, WBT and WBT-PB perform very similarly.

- ⇝ COMB

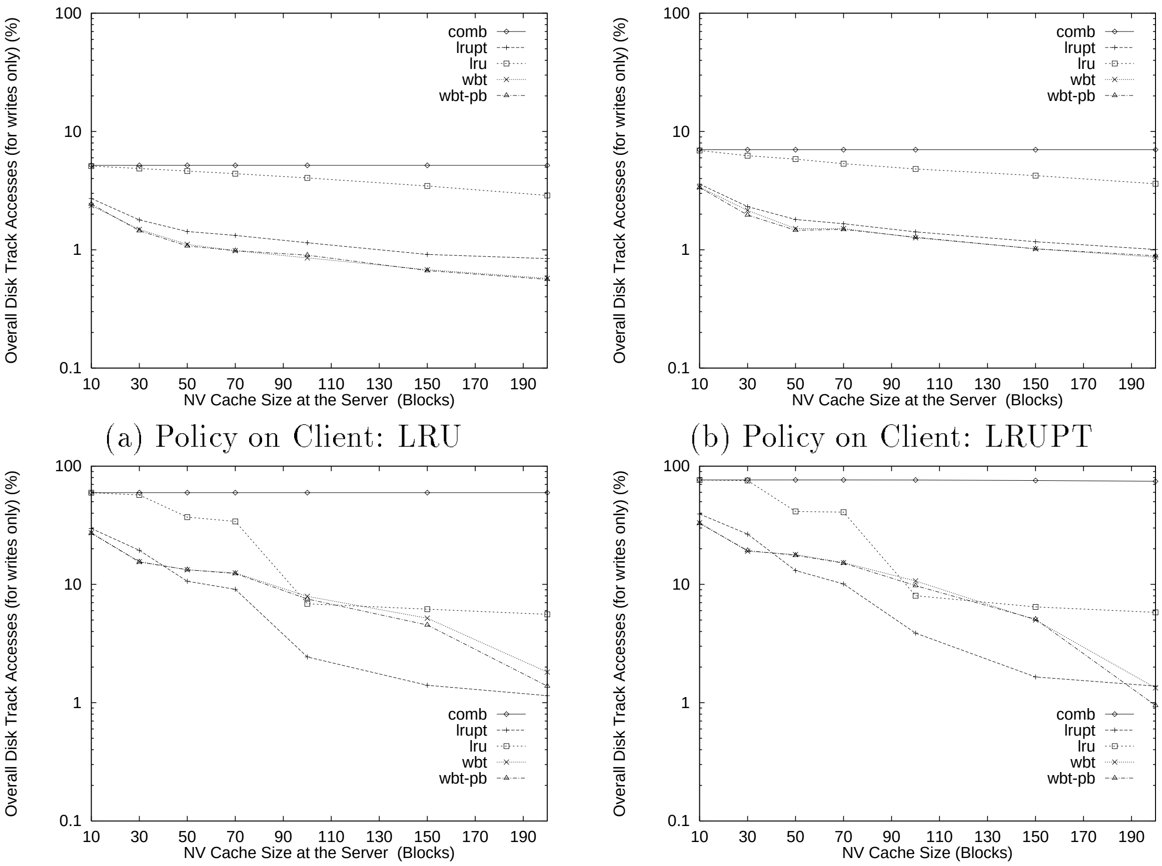
  COMB allows clean blocks required by read operations to occupy space in the NV cache. A clean block can replace a dirty block at the LRU-end of the NV cache if this dirty block is older than the block at the LRU-end of the volatile cache. Further, read hits on clean blocks in the NV cache cause the blocks to be moved to the MRU end of the list of blocks in the NV cache, implying that they will be unlikely to be replaced in the near future. Thus, the effective NV cache size for COMB for write caching may be much smaller than the actual NV cache size. Therefore, COMB may write many more dirty blocks back to disk than the other policies do. In the single-level NV write caching context, COMB is less effective than the other policies, with a high number of disk track accesses, a high number of blocks written to disk, and a small number of overwrites of dirty blocks in the NV cache.

## 5.2 Multi-level NV Write Caching

In this section, simulation results are presented for the case in which non-volatile caches are present on both the client and the server. The size of the non-volatile cache on the client takes on values of 10, 50, and 100 blocks in the experiment, while the size of the server's NV cache varies from 10 to 200 blocks. The results for a client cache size of 100 blocks are given here, followed by some results for a client cache size of 10 and 50 blocks; the remaining results for trace 1 can be found in Appendix A.

The behaviour of caching policies on the client will not be affected by the choice of caching policies on the server. The performance of write caching policies on the client should be identical to the performance of these policies with single-level caching. The performance of caching at the server is affected by caching at the client, however, and so the "overall" performance metrics introduced in Section 4.2.5 are considered in this section. Additional results focusing on the performance of the server NV cache are found in Appendix A.

Figure 5.10 shows the number of disk track accesses performed for write operations




(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

(c) Policy on Client: WBT

(d) Policy on Client: COMB

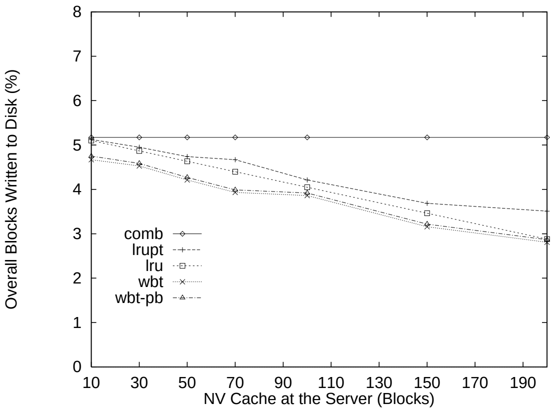
Figure 5.10: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations. Client NV Cache Size is 100 Blocks

for each policy combination at the client and the server, expressed as a percentage of the total number of write references in the trace. Because of the large variation in magnitude of the results, a log scale is employed on the vertical axis. Note that the combination of LRU on the client and WBT or WBT-PB on the server appears to be the best combination of write caching policies for multi-level caching in this case, in terms of minimizing the number of disk track accesses for writes.

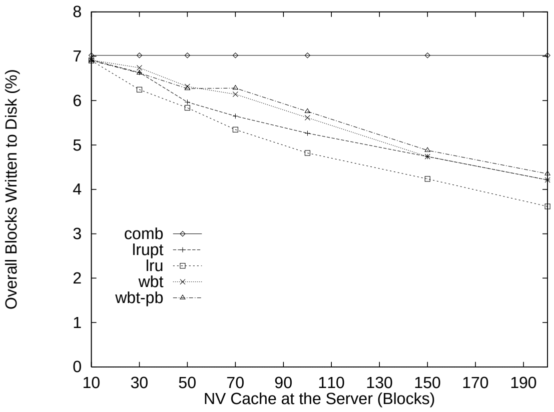
With a large NV cache size on the client (say 100 blocks), the whole client working set of dirty blocks may be held in the cache at the client, and therefore not much temporal locality is left in the write reference stream when it reaches the server. A consequence is that LRU and LRUPT do not perform as well as WBT or WBT-PB on the server, when LRU or LRUPT is used on the client. The reason why this happens is that the temporal locality characteristics seen at the server may be destroyed ("filtered out") by caching at the client. This "filtering" phenomenon of client caching has been observed previously in the context of read caching [Willick *et al.* 93]. Note that increasing the NV cache size on the server does not improve performance when COMB is employed on the server, while the performance of the other policies improves.

The number of blocks written back to disk for each of the considered combinations of client and server NV caching policies is shown in Figure 5.11, expressed as a percentage of the number of write operations. LRU on the client and WBT or WBT-PB on the server yields the fewest write backs. Notice that when LRU or LRUPT is used at the client, the overall blocks written to disk is less than 7%, while the overall blocks written to disk is higher than 60% when WBT or COMB is used at the client, as shown in Figure 5.11. Policies used at the client seem play a more important role in overall performance than policies used at the server. Comparing these results to those of Figure 5.3 also gives striking evidence of the "filtering" phenomenon in multi-level write caching. WBT and WBT-PB outperform LRU and LRUPT at the server, in terms of overall performance measures, in a multi-level caching system, while LRU and LRUPT perform much better than WBT and WBT-PB in a single-level caching system. Temporal locality focused policies (LRU and LRUPT) perform well at the client which sees temporal locality characteristics in the reference stream. LRU and LRUPT perform poorly at the server where the temporal locality characteristics in the reference stream is destroyed ("filtered out") by client caching when it reaches the server. Again, increasing the size of the NV cache on the server does not affect the performance of COMB much, while other policies perform much better when the NV cache on the server gets larger.

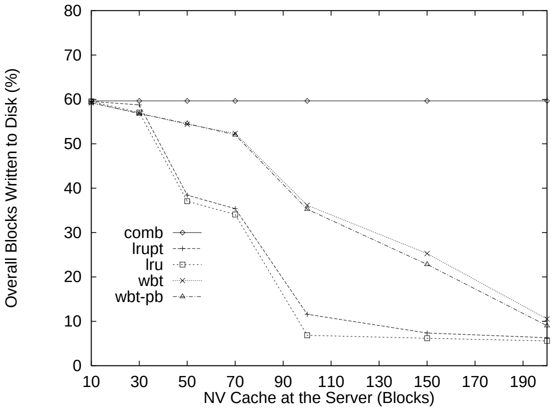
When NV write caching is used, dirty blocks can be overwritten within the NV



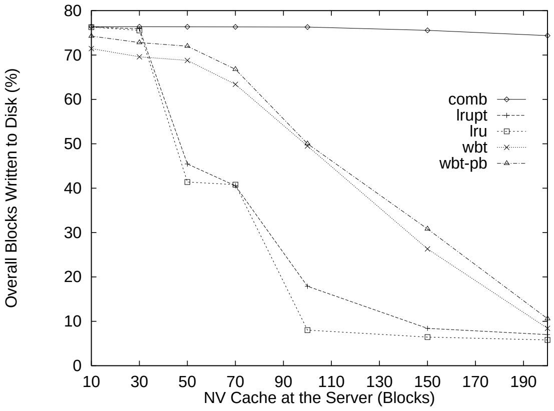

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT



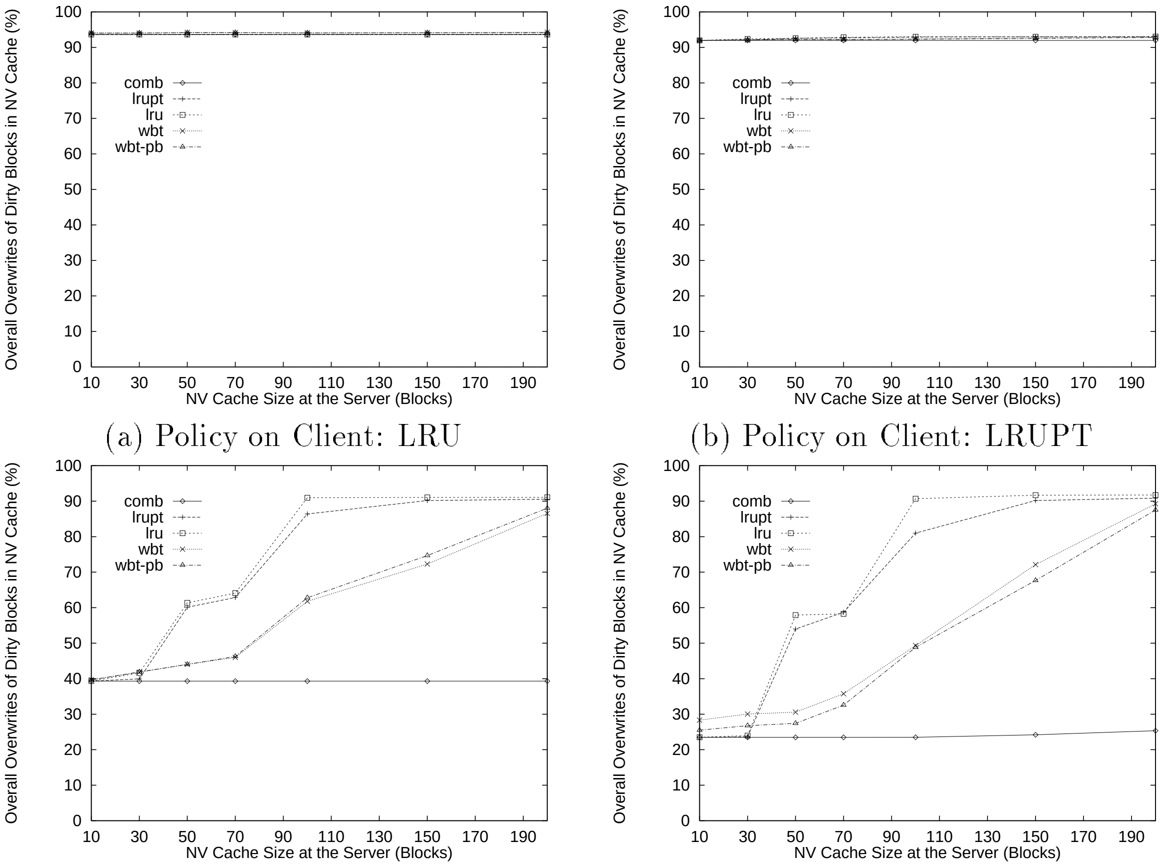

(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure 5.11: Blocks Written to Disk, as a Percentage of Total Write Operations. Client NV Cache Size is 100 Blocks

cache before being written back to the disk. The greater the number of dirty blocks that are overwritten in the NV cache, the better. Figure 5.12 shows the number of blocks overwritten in the two NV caches, expressed as a percentage of the total number of write operations. LRU or LRUPT on the client yields a high number of overwrites regardless of the policies used on the server. Evidently, the client NV cache is big enough (100 blocks) to contain the working set of dirty blocks, and also since it is managed according to the principles of locality, it does a good job of maintaining blocks likely to be overwritten in the cache. Only when the server NV cache size is similarly large to the client NV cache (90 blocks or more), does the combination of LRU or LRUPT at the server and WBT or COMB at the client yield similar




(a) Policy on Client: LRU (b) Policy on Client: LRUPT

(c) Policy on Client: WBT (d) Policy on Client: COMB

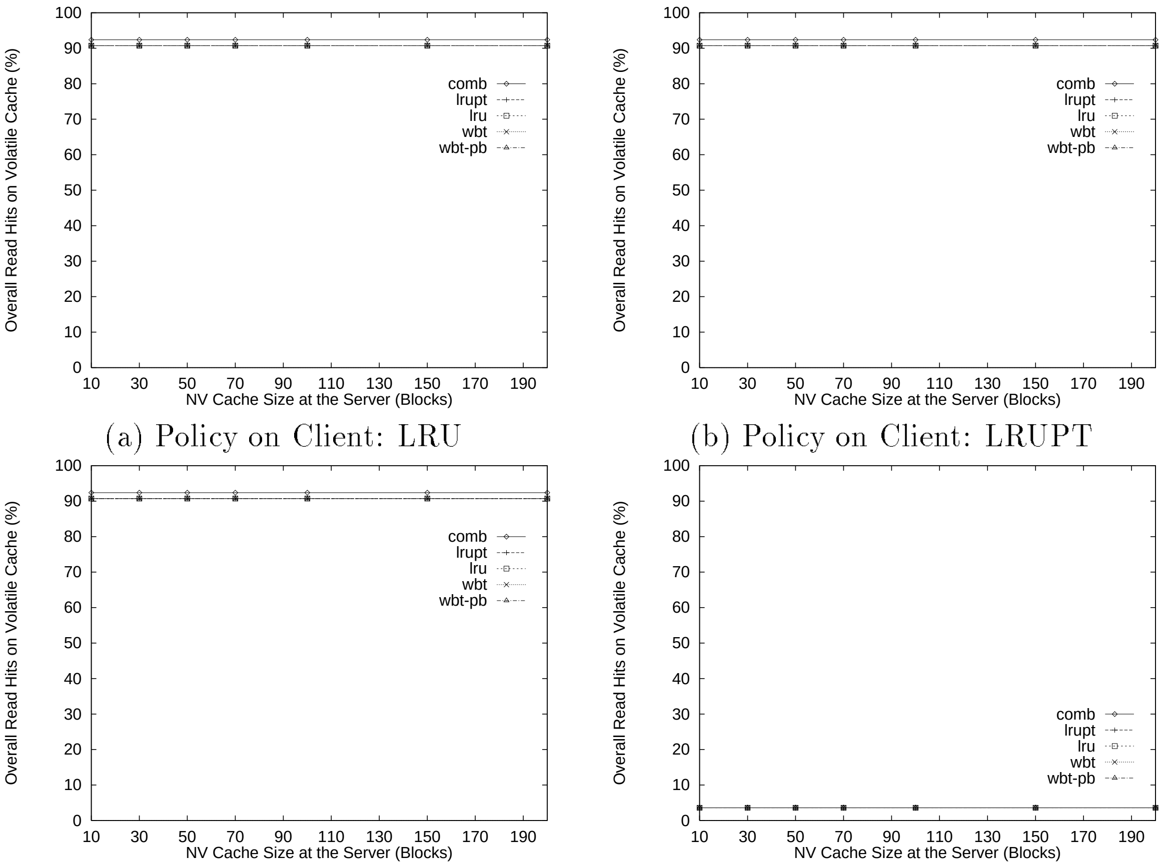
Figure 5.12: Overwrites of Dirty Blocks in NV Caches, as a Percentage of the Total Write Operations. Client NV Cache Size is 100 Blocks

performance.

The percentage of blocks written to disk should equal 100% less the sum of the percentage of dirty blocks overwritten and the percentage of dirty blocks deleted. As was observed in the context of single-level caching, there is no significant number of dirty blocks that are deleted before being written to disk.

The overall read hit ratio at the volatile caches for each of the policy combinations is shown in Figure 5.13. When COMB is used on the client, the overall read hit ratio at the volatile caches is very low (about 3%), regardless of the policy used on the server and regardless of the server NV cache size. When a policy other than COMB is used on the client, the read hit ratio on the volatile cache is higher than 90%.

The overall read hit ratio at the two NV caches for each of the policy combinations is




(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

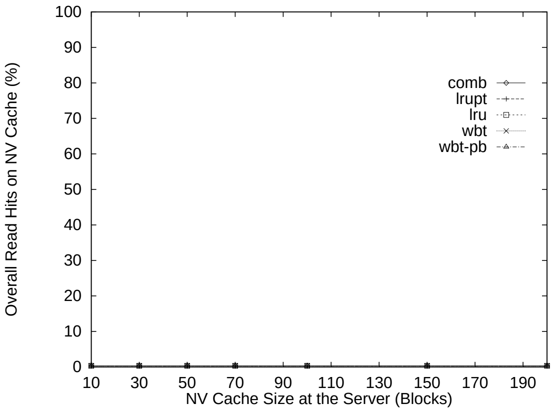
(c) Policy on Client: WBT

(d) Policy on Client: COMB

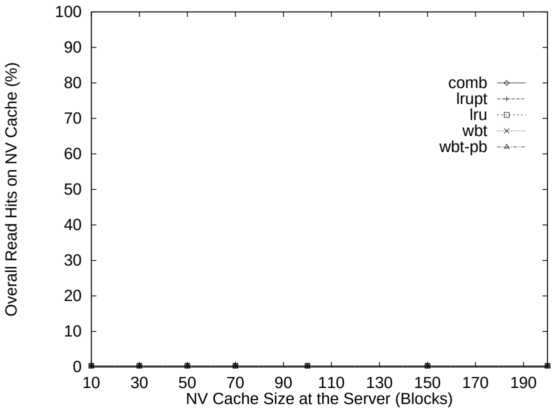
Figure 5.13: Overall Read Hit Ratio at Volatile Caches. Client NV Cache Size is 100 Blocks

shown in Figure 5.14. This read hit ratio is close to zero when LRU, LRUPT, or WBT is used on the client, regardless of the policy used on the server. When COMB is used at the client, the read hit ratio at the NV caches is about 90%. The explanation for this strikingly different behaviour of COMB is identical to that in the single-level caching context – a NV cache managed by COMB ends up serving read traffic.

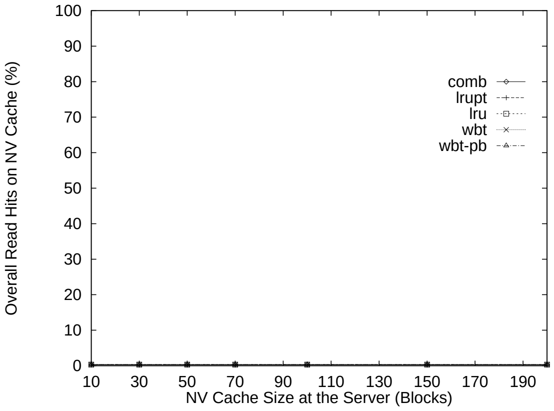
Some measures relating to piggybacking performance are shown in Figure 5.15. The percentage of writes piggybacked on read misses is shown in Figure 5.15 (a). When WBT-PB is used on the server and COMB is used on the client, 5% to 6% of writes are piggybacked. All the other choices of client caching policy result in fewer



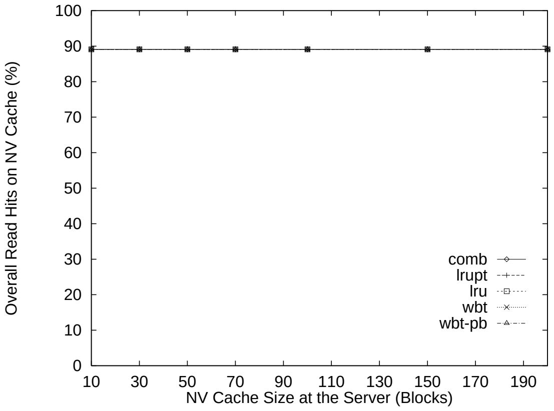

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT




(c) Policy on Client: WBT




(d) Policy on Client: COMB

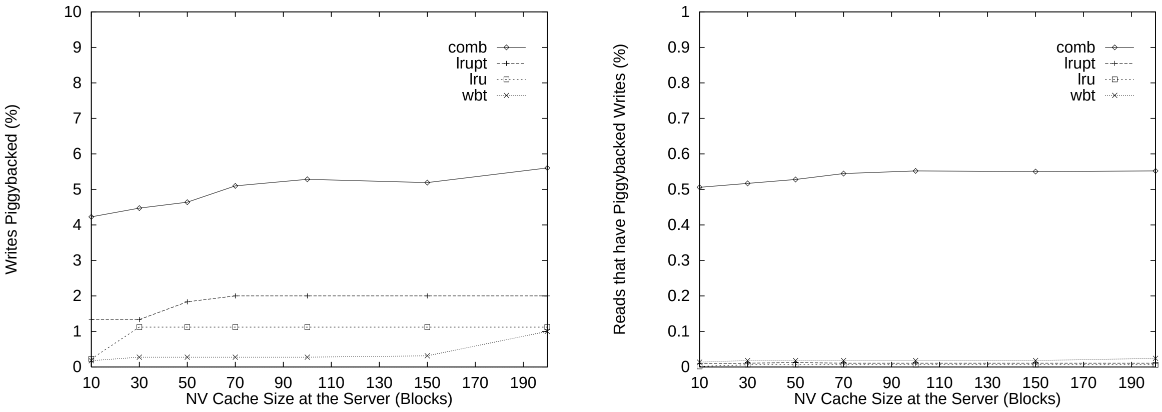
Figure 5.14: Overall Read Hit Ratio at NV Caches. Client NV Cache Size is 100 Blocks

writes that can be piggybacked. Figure 5.15 (b), showing the percentage of reads that have piggybacked writes, similarly indicates the advantage of using COMB on the client, in terms of piggybacking success. One possible explanation is simply that COMB on the client results in a greater portion of the dirty blocks residing at the server cache where they are available for purging.

Some conclusions about the behaviour of the policies considered are as follows:

⇝ LRUPT

As indicated by a previous study [Willick *et al.* 93], client caching may destroy much of the temporal locality seen in disk block reference streams at the server. Therefore, when LRUPT or LRU is used on the client, and the NV cache on




(a) Read Misses that have Writes Piggybacked

(b) Writes Piggybacked

Figure 5.15: Statistics on Piggybacking. Client policies are COMB, LRUPT, LRU, and WBT, while server policy is WBT-PB. Client NV Cache Size is 100 Blocks

the client is large enough to hold the client's working set of dirty blocks, using a temporal locality focused policy as LRUPT or LRU for write caching on the server may yield poor performance. However, if WBT or COMB is used on the client (policies that seem less effective at retaining the working set of dirty blocks locally), LRUPT can perform well on the server, achieving a low number of disk track accesses, a low number of blocks written to disk, a high number of overwrites of dirty blocks in NV cache.

Regardless of the server policy, LRUPT appears to yield good performance on the client, although not as good as that with LRU on the client. Apparently the file-based purging operations carried out by LRUPT on the client are not significantly beneficial in terms of allowing the server to do more effective track-based purging.

⇝ LRU

LRU might be the best policy for use at the client, but LRU is not a suitable policy for use at the server, since it does not amortize the cost of many block writes over a single disk track access. When the NV cache size on both the client and the server is large, however, the combination of LRU on the client (so as to take advantage of the temporal locality in the reference stream at the client) and WBT or WBT-PB on the server, appears to be the preferred combination.

⇝ WBT or WBT-PB

WBT does not take advantage of temporal locality to the same extent as do LRUPT and LRU. Thus, it does not perform as well as LRU or LRUPT on the client when the NV cache is large enough for the whole working set. It appears that WBT is not a good choice for the client NV cache in a multi-level NV caching system. (As will be seen in Section 5.3.1, this also appears to be the case for small client NV caches.) On the server, however, when temporal locality is filtered out by client NV caching, LRU and LRUPT perform poorly, but WBT and WBT-PB (as server policies) perform well. When the client has a large NV cache (as will be seen in Section 5.3.1), it appears that the combination of WBT or WBT-PB on the server, with LRU on the client, is the best policy combination. WBT-PB does not seem to offer a large performance improvement over WBT for large client NV caches, but does offer a large improvement for small client NV caches, presumably because a large portion of the client's working set of dirty blocks is present at the server, for potential use in piggybacking, when the client's NV cache is small.

⇝ COMB

COMB uses the NV cache not only for write caching, but also for read caching. In fact, under COMB the NV cache appears to serve mostly the read traffic, for the reasons noted previously. The combination of COMB on the client and COMB on the server yields the worst performance of any policy combination studied. Surprisingly, however, COMB can be a good choice for the client policy for small NV cache sizes as will be seen in Section 5.3.1.

## 5.3 The Sizes of Non-Volatile Caches on Server and Client

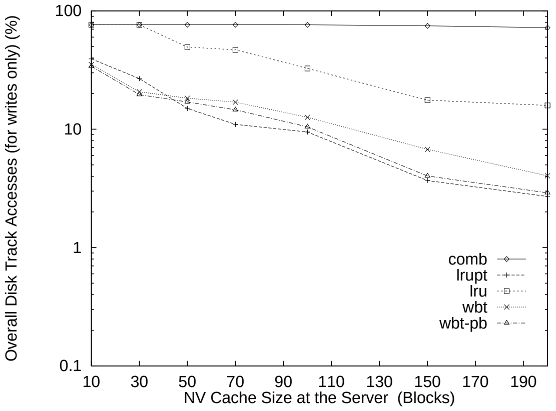
The size of any cache is an important performance determinant. This section considers both the influence of NV cache sizes on overall multi-level NV write caching performance, and the influence of NV cache size on policy comparisons with respect to this performance.

### 5.3.1 The Influence of NV Cache Sizes on Different Policy Combinations

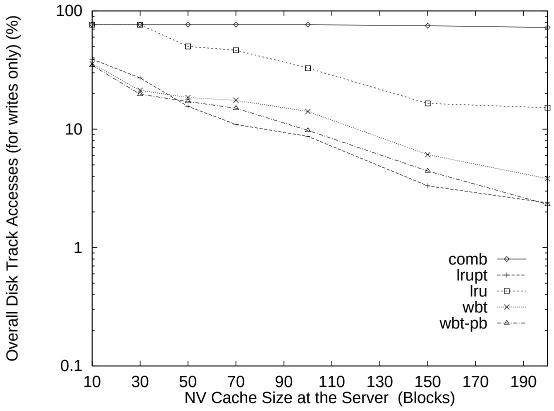
When the NV cache size on the client is small, temporal locality is not completely destroyed in the reference stream seen by the server, and the performance with policy combinations using LRU or LRUPT on the server should not be as bad as that when the NV cache on the client is large (i.e., large enough to hold the entire working set). As can be seen by comparing (a) and (b) in Figures 5.16, 5.17, and 5.18 to (a) and (b) in Figures 5.10, 5.11, and 5.12. LRUPT does better than WBT on the server when the size of the NV cache on the client is 10 blocks, while WBT does better on the server when the size of the client NV cache is 100 blocks.

When the NV cache on the client is small (10 blocks), the policy combination of COMB on the client and either WBT-PB (for a large server NV cache) or LRUPT (more moderately sized NV cache) on the server, yields the fewest disk track accesses for write operations, as shown in Figure 5.16 (d). When COMB is used on the client, with the 10 block NV cache, about 78% of writes end up going to the server (as shown in Figure 5.3). Thus, the temporal locality in the server's incoming stream of disk block writes may not be damaged severely, allowing LRUPT the opportunity to work well on the server. Further, this appears to provide significantly more choices for piggybacking (compare Figures 5.15 (a) and 5.19 (a)), at least for large server NV cache sizes, thus allowing WBT-PB to work well.

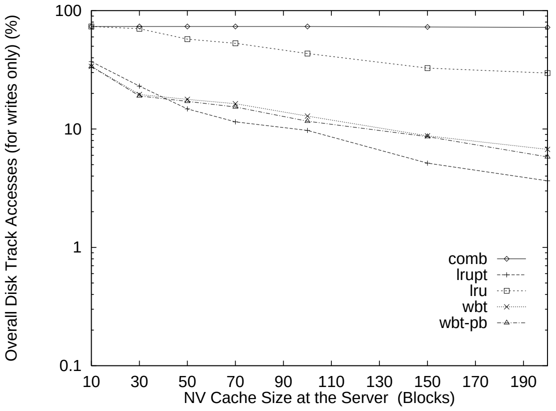
Results from traces 2 and 3 (Appendices B and C, respectively) do not show the



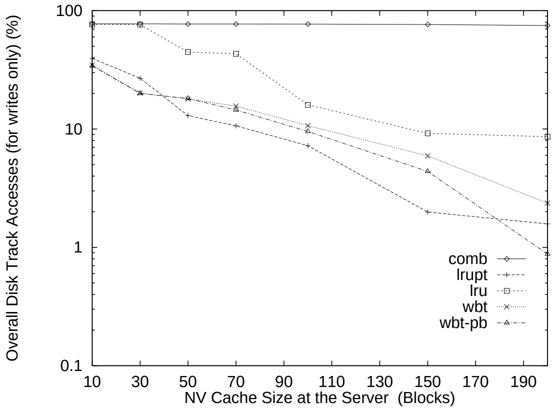

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT



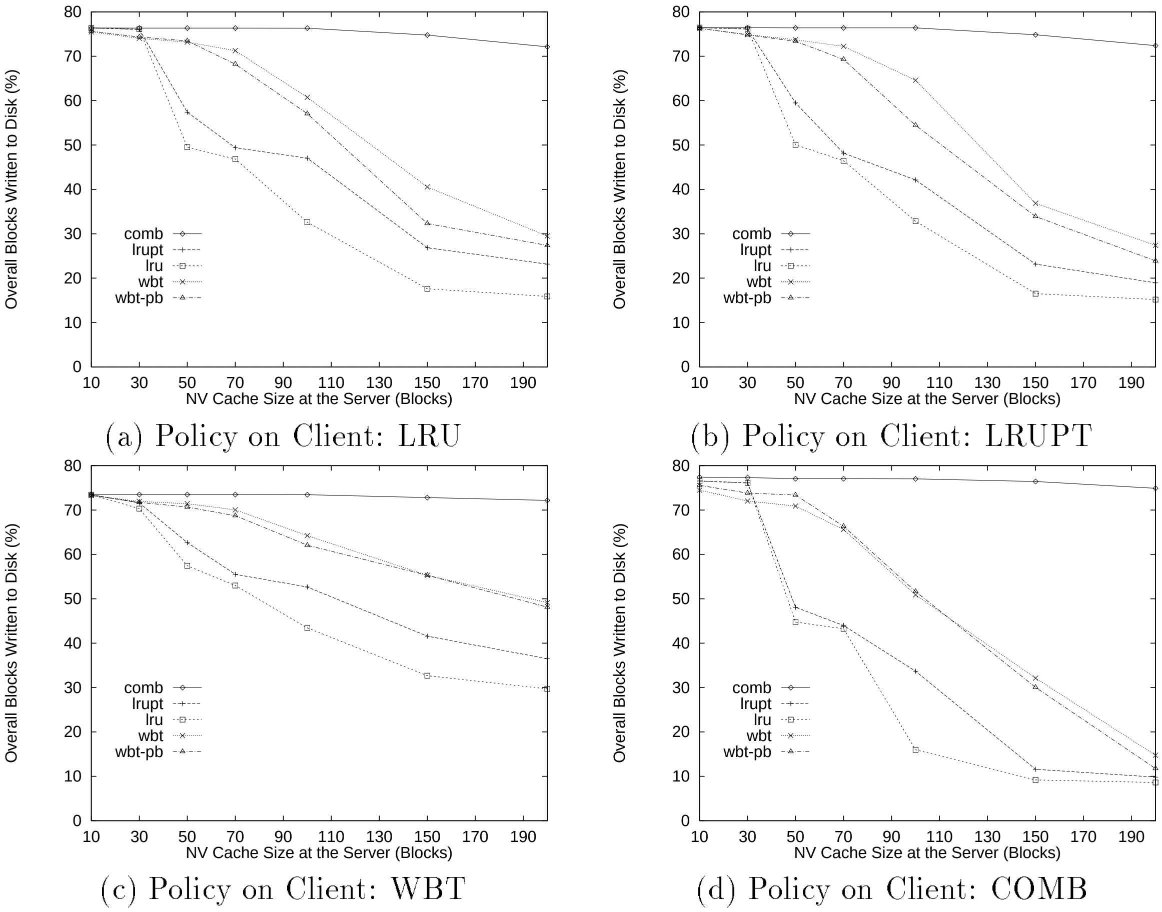

(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure 5.16: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations. Client NV Cache Size is 10 Blocks

same kind of influence of the client NV cache size on the locality characteristics seen by the server, as do the results from trace 1. In fact, the combination of COMB on the client and WBT or WBT-PB on the server is always the best policy combination for a client NV cache size of 10, 50, and 100 blocks for trace 2 (Figures B.9, B.20, and B.31 in Appendix B) and trace 3 (Figures C.9, C.20, and C.31 in Appendix C). The reason why policy combination of COMT at the client and WBT or WBT-PB at the server performs differently in traces 2 and 3 than in trace 1 might be because the "working sets of dirty blocks" of traces 2 and 3 are not as clearly defined and/or are larger than they are in trace 1, and thus cannot be contained in the client NV cache even for the largest client cache size (100 blocks) considered in the simulations.




(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

(c) Policy on Client: WBT

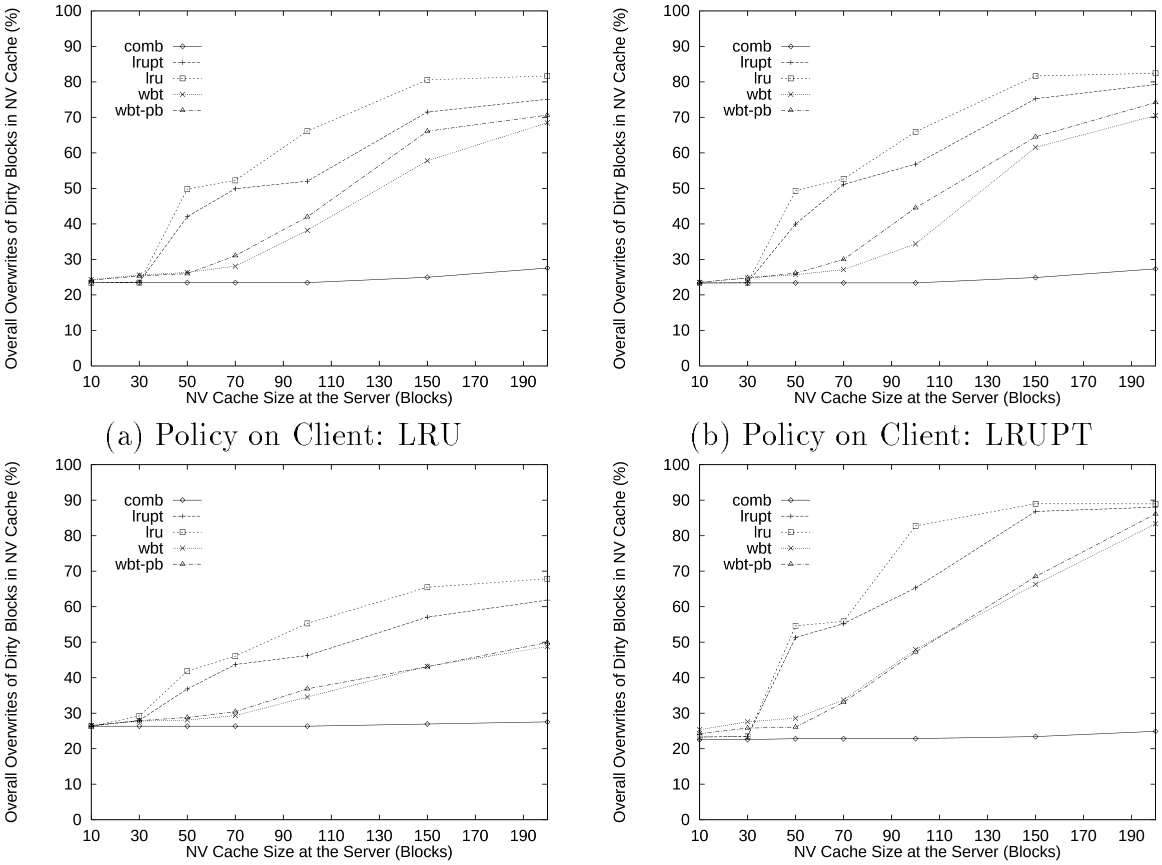
(d) Policy on Client: COMB

Figure 5.17: Blocks Written to Disk, as a Percentage of Total Write Operations. Client NV Cache Size is 10 Blocks

With larger cache sizes, results from trace 2 and 3 might be more interesting.

### 5.3.2 The Influence of NV Cache Sizes on Performance

The NV cache added to the client is used to reduce the network traffic generated by writing dirty blocks to the server or to the disk, and it can also reduce the workload at the server. Better performance can also be achieved for write operations by adding NV cache to the client, since writing to the NV cache at the client is much faster than writing things back to the server or disk with overheads of network accesses and slow disk operations. The NV cache is added to the server to save slow disk accesses, i.e., to amortize the cost of many block writes over a single disk track access. To achieve




(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

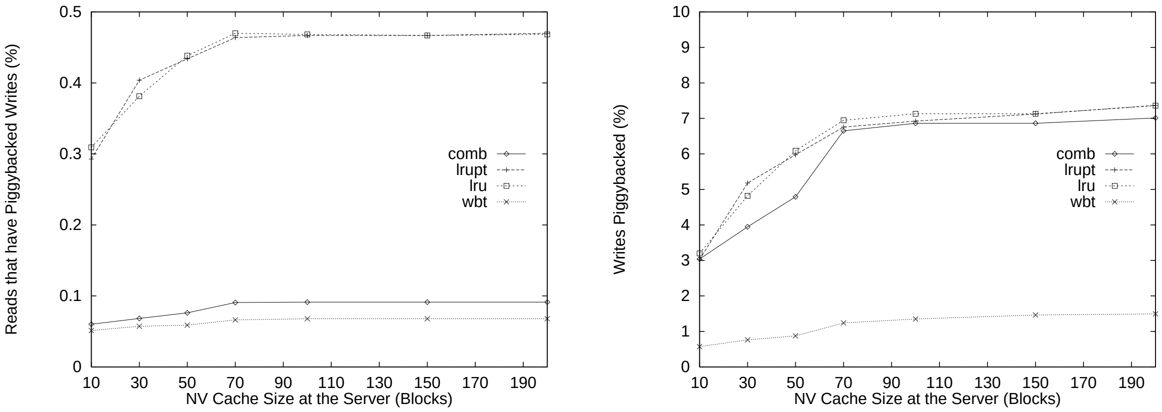
(c) Policy on Client: WBT

(d) Policy on Client: COMB

Figure 5.18: Overwrites of Dirty Blocks in NV Caches, as a Percentage of the Total Write Operations. Client NV Cache Size is 10 Blocks

better performance in terms of less network traffic, better response time, and reduced server load, a large NV cache at the client is required. On the other hand, to achieve better performance in terms of fewer disk accesses, a large NV cache is necessary at the server. The effect of NV cache size on the performance of NV caching is discussed in this section.

In a previous study of disk read cache management for distributed systems [Makaroff 88], it was observed that the server cache miss ratio does not drop significantly with increasing server cache size until the server cache size approaches the client cache size. This phenomenon can also be seen in this research for write caching. Figures 5.20, 5.21, and 5.22 show the number of disk track accesses for writes, the blocks written



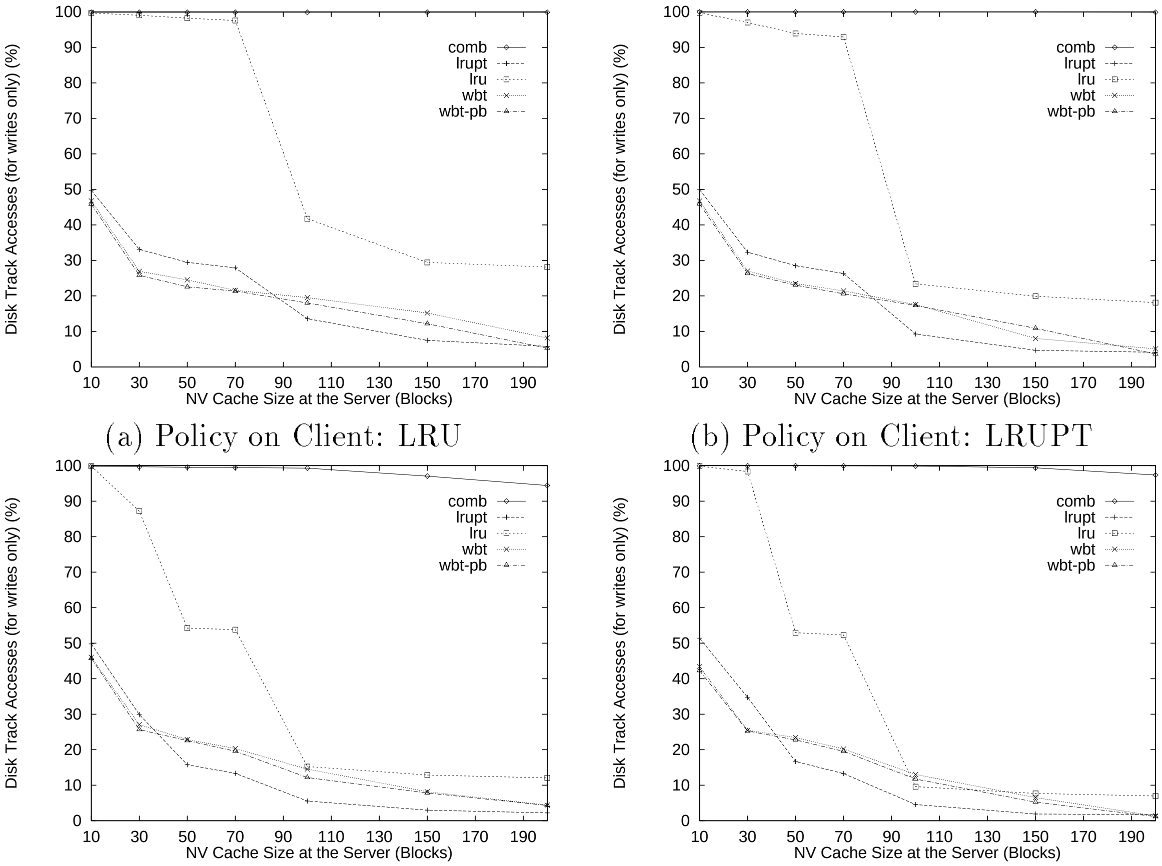

(a) Read Misses that have Writes Piggybacked

(b) Writes Piggybacked

Figure 5.19: Statistics on Piggybacking. Client NV Cache Size is 10 Blocks

to disk, and the overwrites of dirty blocks in the server NV cache, as a percentage of the total write operations seen at the server, when the NV cache at the client is 50 blocks.

When LRU or LRUPT is used at the client and LRU or LRUPT is used at the server, as shown in Figures 5.21 and 5.22, the performance does not improve much when the NV cache on the server is smaller than the client's NV cache (50 blocks), in terms of dirty blocks written back to disk and overwrites of dirty blocks in the server's NV cache. There is a dramatic decrease in the number of blocks written back to disk and a sharp increase in the number of overwrites of dirty blocks in the server's NV cache, for LRU or LRUPT at the server, when the server's NV cache becomes significantly larger than the client's NV cache (when the server's NV cache size exceeds 70 blocks). More of the frequently accessed dirty disk blocks are retained in the client's NV cache when it is large enough, so the server's NV cache does not see many repeated accesses to these blocks. The dirty blocks that are in the server's NV cache are placed there by the initial client requests, but repeated accesses at the client for those blocks do not keep them at the top of the LRU stack in the server's NV cache. Only when the NV cache at the server is larger than the client NV cache can the blocks accessed less frequently be held in the server's NV cache.




(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

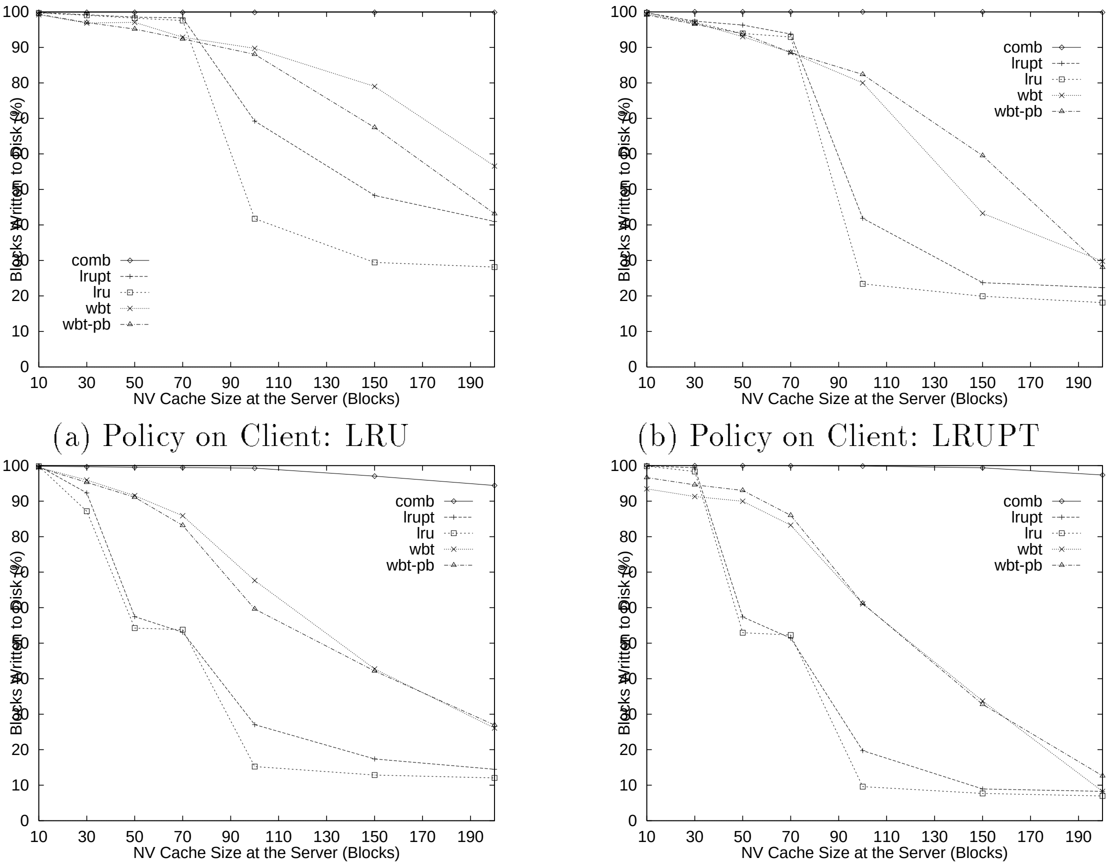
(c) Policy on Client: WBT

(d) Policy on Client: COMB

Figure 5.20: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache is 50 Blocks

Unlike policies focused on temporal locality characteristics, when policies focused on amortizing the cost (e.g., WBT, and WBT-PB) are used at the server, increasing the server's NV cache size even by a small amount (e.g., from 10 to 30 blocks), improves performance in terms of disk track accesses, when there is a moderate sized NV cache at the client (e.g., 50 blocks), as shown in Figure 5.20. There is no dramatic change for policies of WBT or WBT-PB at the server, in terms of dirty blocks written back to disk and overwrites of dirty blocks in the server's NV cache, when the server's NV cache size exceeds the client's NV cache size. Temporal locality characteristics in the reference stream may not be so crucial to the performance of WBT and WBT-PB.

The effective client NV cache size for write caching when COMB is used at the



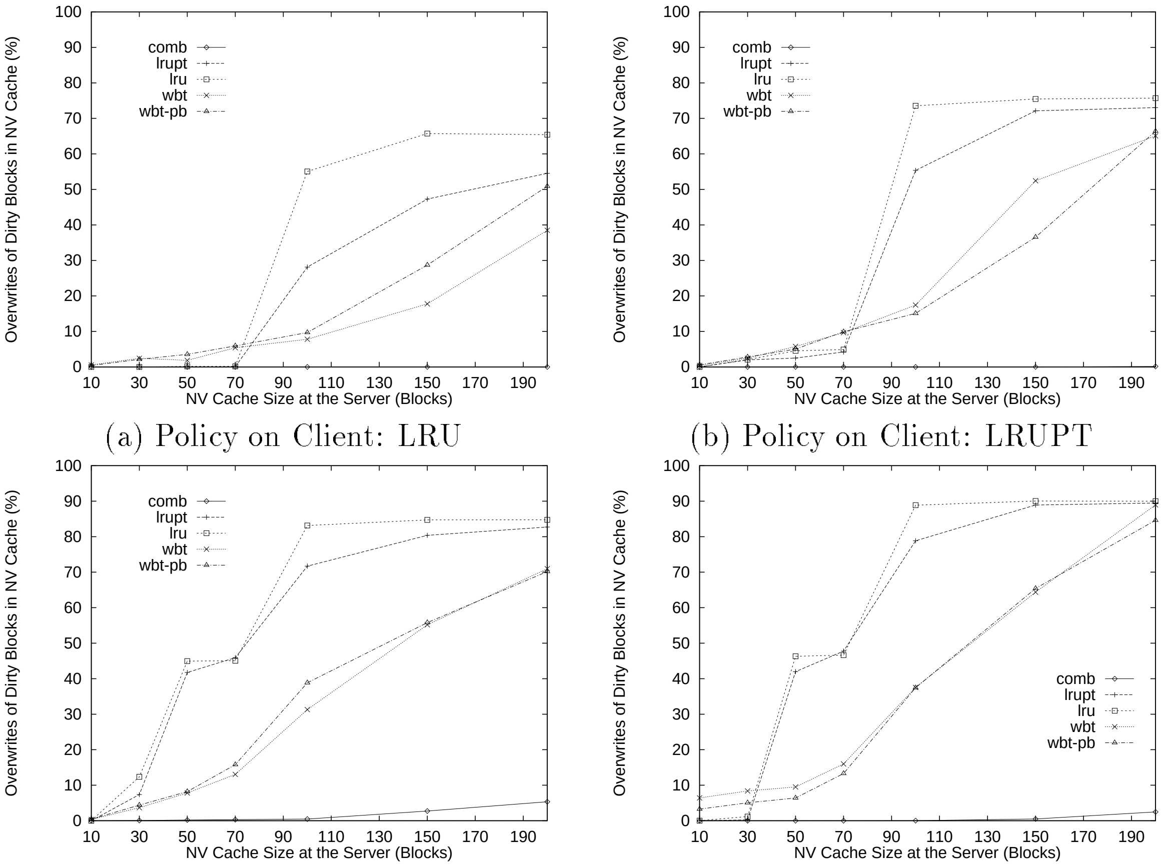

(a) Policy on Client: LRU (b) Policy on Client: LRUPT

(c) Policy on Client: WBT (d) Policy on Client: COMB

Figure 5.21: Blocks Written to Disk, as a Percentage of Total Write Operations Seen at the Server. Client NV Cache Size is 50 Blocks

client is smaller than what it appears to be because of the residence of clean blocks required by reads in the NV cache. The phenomenon that increasing the server's NV cache size does not improve the performance in terms of dirty blocks written back to disk until the server's NV cache size is greater than the client's effective NV cache size can also be seen in Figure 5.21 (d).

When there is a moderate-sized NV cache at the server (e.g., 50 blocks), adding a small NV cache at the client (e.g., 10 blocks) does not improve the overall performance in terms of disk track accesses. In fact, it makes the performance even worse than with no client NV caching at all. This can be seen by comparing Figure 5.16 with Figure 5.2. Increasing the client's NV cache size (from 10 to 100 blocks) improves




(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

(c) Policy on Client: WBT

(d) Policy on Client: COMB

Figure 5.22: Overwrites of Dirty Blocks in Server NV Cache, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache is 50 Blocks

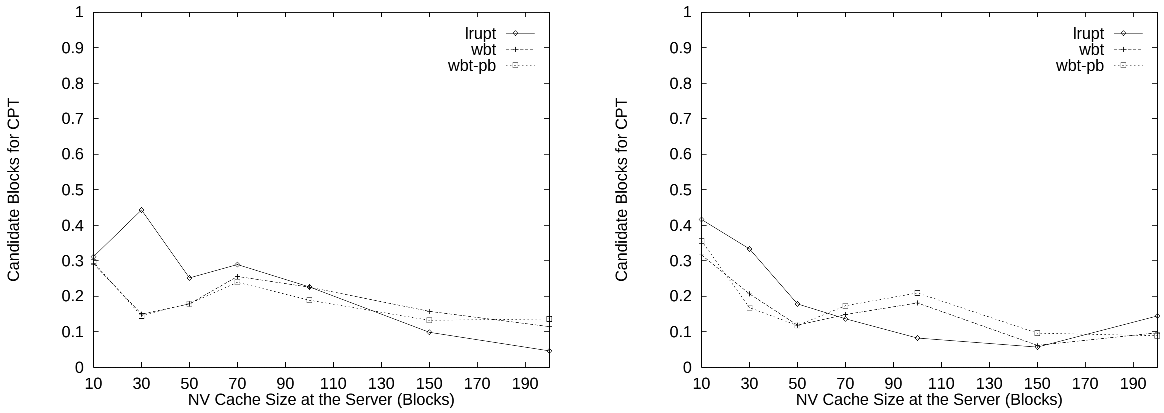
the performance of policy combinations in terms of disk track accesses (as shown in Figures 5.16 and 5.10). For a 100 block NV cache at the client and a 50 block NV cache at the server, and for the policy combination of LRU at the client and WBT or WBT-PB at the server, performance in terms of disk track accesses is better than having a 150 block NV cache at the server and no NV caching at the client. However, with an appropriate hybrid policy, a single-level NV cache with 150 blocks may perform better than a caching system that partitions the 150 blocks of NV cache into 100 blocks at the client and 50 blocks at the server. For example, one might attempt to capture the benefits of both LRU and WBT, by dividing the NV cache into two partitions: a LRU partition and a WBT partition. The LRU partition would be managed by the LRU replacement policy, with aged blocks being moved from the LRU partition into the WBT partition. Purging operations would be performed only from the WBT partition. On the other hand, reduced network traffic, better response time, and reduced server load can be achieved by putting a NV cache at the client.

For a certain amount of NV memory, if the sharing of the NV cache at the server in a multi-client environment is the issue to be considered (ignoring the extra client delay in going through the network and the server load), large NV cache at the server and small NV cache at each client is required. Amortizing the cost of writing many dirty blocks over a single disk track can be more effective when the server's NV cache is large, because each client with a small NV cache may write many dirty blocks to the server's NV cache. Meanwhile, the network traffic and server load can be reduced to some extent with the small NV cache at the clients. On the other hand, if better performance, less network traffic, and reducing the server load are the issues to be considered (ignoring the sharing at the server), large NV cache at each client and a small NV cache at the server is desired. Fewer dirty blocks will be written back to the server when the NV cache of each client is large enough to hold the "working set of dirty blocks" of that client, and the network traffic and the server load can be reduced. At the same time, the small NV cache at the server amortizes the cost of writing some dirty blocks over a single disk track.

## 5.4 Coordinating NV Write Caching on the Server and Client

For each combination of purging policies used on the server and client (LRUPT on the server and LRUPT or WBT on the client; WBT on the server and LRUPT or WBT on the client), each time a new disk track is used as the basis for purging operations on the server, the NV cache on the client is searched for all the blocks that belong to this disk track. The number of blocks belonging to the same disk track as the purging base on the server are averaged over all the blocks that are written back to disk by the server, to illustrate whether or not coordinating the purging operations on the server and client is necessary. If the average number of these blocks on the client is large, that may indicate that it is reasonable to have purge operations on the client coordinated with the purge operations on the server.

Figure 5.23 (a) and Figure 5.23 (b) show the number of CPT candidate blocks as an average over blocks written back to disk, when the client NV cache size is 100




(a) Candidate Blocks for CPT, Client Policy is LRUPT

(b) Candidate Blocks for CPT, Client Policy is WBT

Figure 5.23: Statistics on CPT. Client policies are LRUPT, WBT, and WBT-PB. Client NV Cache Size is 100 Blocks

blocks. For each block written back to disk from the server, there appears to be less than 0.5 blocks in the client's NV cache able to be a candidate for CPT, regardless of the policies at the client and regardless of the NV cache sizes at the server. This might indicate that CPT is of questionable value, considering the overhead and the benefit of CPT.

## 5.5 Summary

In this chapter, the performance of five NV write caching policies at the server and four NV write caching policies at the client are studied and compared. The effects of the NV cache sizes at both server and client on the performance of different policy combinations in multi-level NV caching, as well as on the performance issues, are investigated. The measures of NV caching performance considered are those described in Section 4.2.5.

Several interesting results are observed for the limited selection of workloads studied. These results confirm some of the results obtained in previous research, but also point out some unique characteristics of a distributed non-volatile write caching system. Continued study is necessary in order to establish that the results obtained hold for other workloads on similarly configured systems.

With single-level NV caching, LRUPT appears to be the best choice of write caching policy, as observed and explained in Section 5.1. With multi-level NV write caching, COMB on the client and WBT-PB on the server provides the best combination of policies for smaller client NV cache sizes (10 blocks), and LRUPT on the client and WBT or WBT-PB on the server seems be the best combination of policies for larger client NV cache sizes (100 blocks), as shown in Section 5.2.

When the NV cache size on the client is large enough to hold the entire working set, NV write caching on the client may filter out temporal locality characteristics from the reference stream that the server sees. Also, if temporal locality focused policies are used at both the client and the server, the performance of NV caching on the server does not improve much until the server NV cache size approaches the client NV cache size. For purging-focused policies at the server and temporal locality focused policies at the client, increasing the server's NV cache size does improve the performance in terms of disk track accesses. Adding a small NV cache at the client may make the performance even worse than having no NV caching at the client, in terms of disk track accesses, as shown in Section 5.3.2.

Coordinating the purging operations on the server and the client may be unnecessary as indicated by the results from trace 1, since there are very few blocks that can be chosen to be CPT candidate blocks.

/

# Chapter 6

# Summary and Conclusions

The goal of this thesis was to evaluate policies for managing non-volatile caches in a distributed file system. Section 6.1 provides a summary of what has been done in the thesis. The major results of the research are outlined in Section 6.2. In Section 6.3, possible directions for future research are suggested.

## 6.1 Thesis Summary

In Chapter 1, the performance and reliability benefits made possible by using non-volatile memory write caching are described. Network traffic from client to server, and disk accesses at the server, can be reduced by adding non-volatile caches on client and server for write caching. Such caches also greatly reduce the latency of write operations as the data need only be placed in the cache, rather than synchronously written to disk. Dirty data stored in the non-volatile cache(s) will often be overwritten or deleted before being flushed back, thus reducing the network traffic and the number of required disk I/Os. Further, unlike write caching using conventional memory, system data integrity can be guaranteed while the dirty data resides in cache, even in the event of power failures.

Chapter 2 discusses previous research in disk cache management schemes. First, five different file systems are overviewed: Sun Network File System [Sandberg *et al.* 85], Sprite [Nelson *et al.* 88], Andrew [Howard *et al.* 88], Coda [Kistler *et al.* 92], and

Echo [Hisgen *et al.* 89]. Cache design considerations are then detailed, using the systems outlined previously as examples.

Chapter 3 surveys previous research work concerning the use of non-volatile memories as caches and as buffers on servers. Two non-volatile (NV) caching models used in [Baker *et al.* 92] are discussed in detail, as well as some NV caching policies used in previous studies, including modified LRU replacement, write-back with thresholds, and piggybacking.

Chapter 4 presents the caching models and the five NV caching policies for which experiments are performed: COMB, LRU, LRUPT, WBT, and WBT-PB. The system from which disk block reference traces are obtained is described, as well as the required post-processing of these traces. The experimental factors chosen include the NV write caching policies on the client and server, and the NV cache sizes on the client and the server. Nine performance metrics are used in the simulation, including measures of the number of disk track accesses required by the write operations, the dirty blocks written back to disk, the overwrites of dirty NV cached blocks, the dirty blocks deleted from the NV cache, the read hit ratio on the volatile cache and the read hit ratio on the NV cache, the number of candidate blocks for CPT, the number of writes piggybacked, and the number of reads that have piggybacked writes.

In Chapter 5, results from trace-driven simulation experiments of single-level and multi-level NV write caching are presented and analyzed. The influences of the experimental factors upon the nine performance metrics are evaluated.

## 6.2 Results

The major results of this study can be divided into two categories: the performance of different NV write caching policies in single-level NV caching, and the performance of different policy combinations in multi-level NV write caching. Results are also presented concerning the effect of the size of the NV cache on performance and whether a coordinated purging policy such as CPT is reasonable or not.

The results obtained in this thesis suggest that in a single-level NV caching system, LRUPT can perform very well. The existence of temporal locality in the reference stream from an individual application makes LRUPT a successful NV write caching policy in this context. Recently referenced blocks that may be overwritten or deleted in the near future remain in the cache, since LRUPT writes blocks to disk through purging in LRU order. WBT and WBT-PB also perform well, but COMB appears to be ill-suited to single-level NV caching since an excessive fraction of the NV cache appears to be used in serving the read traffic.

In a multi-level NV caching system with a client NV cache large enough to hold the client's working set of dirty blocks, LRU on the client and WBT or WBT-PB on the server is a combination of NV caching policies that performs very well, as discussed in Section 5.2. LRU as used on the client can take advantage of the existence of temporal locality in the disk block reference stream seen at the client. On the server, however, temporal locality is at least partly destroyed by client NV caching. Thus, on the server, policies such as WBT and WBT-PB, which do not take particular advantage of temporal locality but are designed to minimize the number of track accesses by purging blocks belonging to the track with the most dirty blocks, appear to perform best. The combination of LRU on the client and WBT or WBT-PB on the server exploits both temporal locality (on the client) and the economy of writing multiple blocks per track on the server.

As mentioned previously, NV caching on the client may filter out the temporal locality from the reference stream that is seen by the server. If the NV cache on the client is large enough to hold the entire working set, the temporal locality in the reference stream may be almost destroyed. That is not the case when the client NV cache is small. The performance of NV caching policies on the server in terms of disk block writes does not improve much when the NV cache size on the server is increased until the NV cache size on the server approaches the effective NV cache size on the client. Purging-based server cache policies, however, benefit from increasing the server cache size even for smaller sizes, in terms of disk track accesses, owing to greater numbers of blocks that can be found sharing the same tracks.

The overall performance for a moderate-sized NV cache at the server, in terms of disk track accesses, appears to be better when there is no NV caching at the client than when there is a small NV cache at the client. When different NV caching policies are used at the client and at the server, the performance can be improved by increasing the NV cache size at the client if the right policy combination is chosen. When LRU is used for a small NV cache at the client and WBT or WBT-PB is used for a medium-sized NV cache at the server, the performance measured by the number of disk track accesses can be better than the performance obtained by putting all the NV caches together at the server and nothing at the client. It appears that a multi-level caching system with the "right" combination of policies at the client and the server performs better than a single-level caching system managed by a single policy, when they both have the same amount of NV memory. A policy combination for a multi-level caching system allows one to take the advantage of both the policies used at the client and at the server, while a single policy at the server cannot overcome its own limitations.

COMB is not recommended for use on the server under any circumstances, for similar reasons as in single-level NV caching. With a small NV cache on the client that holds only part of the working set, however, the combination of COMB on the client and WBT-PB on the server works very well in terms of minimizing disk accesses (but not in terms of minimizing network traffic). It is unclear as to why this is the case, but presumably COMB must do well at keeping the "right" part of the client "dirty block working set" at the server NV cache, so as to allow most effective management of that cache in terms of piggybacking and in terms of blocks per track access.

Piggybacking can be more effective when the NV cache size is small on the client and large on the server. Piggybacking is more effective when COMB is used on the client than it is when other policies are used on the client.

Attempting to coordinate purging operations on the server and client may not be desirable. The results obtained in this thesis indicate that only a small number of candidate blocks in the client NV cache are candidates for use by a CPT-type policy.

## 6.3 Suggestions for Future Research

Since NV write caching is a developing research area, there are many possible avenues of extension to this work. The rest of this section outlines some future research which can be done to expand on and confirm the results of this thesis.

Further investigations of the nature of the workload in distributed systems of the type considered in this thesis would be beneficial to NV caching performance analysis. The disk block references used for the simulation are obtained from a live workload. Different types of user activities may generate quite different disk block reference patterns. These differences may lead to variations in the performance of NV caching policies. For example, for single-level NV caching, the results of trace 2 indicate that WBT-PB performs better than the other policies while the results of trace 1 and trace 3 show that LRUPT works well. When multi-level NV caching is used, and the client NV cache size is large, results from trace 1 suggest that LRU on the client and WBT or WBT-PB on the server is the policy combination that works well (Figure 5.10), but results from trace 2 and trace 3 show that COMB on the client and WBT-PB on the server is a good combination of NV write caching policies (Figures B.31 and C.31). Capturing trace data in a real industrial environment may provide a greater understanding of disk block referencing behaviour in these systems. Finally, if the physical disk address for each block referenced in the traces can be recorded in each trace record, mapping between logical and physical disk addresses can be more realistic than the disk model used for this mapping task in the thesis.

There are some other limitations of this thesis. For instance, the systems studied are always single client systems. Exploring NV write caching in systems with multiple clients is an important extension. As indicated in a previous study [Willick *et al.* 93], some frequency-based replacement (FBR) [Robinson & Devarakonda 90] policies perform better on the server than pure LRU, particularly when there is temporal locality "filtering" by client caching. For write NV caching, policies such as FBR with purging at server, or a hybrid policy of LRU and WBT (in which the server's NV cache is partitioned into two partitions, one for LRU replacement and one for WBT), might perform well for reasons similar to those that make FBR perform well in read caching at the server. Therefore, a study of the performance of FBR with purging at the server, or a hybrid policy, can be one extension to this thesis. In the purging based policies used in this thesis, it is assumed that writes are going to a single disk, and that by writing multiple contiguous blocks together at once, multiple seeks can avoided. With disk striping, consecutive logical data units (stripe units) are distributed among the disks in a round-robin fashion, so that multiple disk address spaces can be folded into a single, unified space seen be by the host. For purging based write caching policies, there are still considerable benefits to write multiple contiguous blocks (e.g. within a stripe unit) at once rather than individually. Although the benefits for NV caching with disk striping technology are expected to be quantitatively similar to the benefits found in the thesis, the benefits may be quantitatively different (perhaps greater). Investigation of benefits of NV caching with disk striping can be a possible area for future work. Another extension could be to analyze the locality characteristics of the traces, before and after the effects of client caching, to support the conjectures made in this thesis.

# Reference


[Bach 86] Bach, M.J., *The Design of the UNIX Operating System*, Prentice-Hall, 1986.

[Baker *et al.* 91] Baker, M.G., Hartman, J.H., Kupfer, M.D., Shirriff, K.W., and Ousterhout, J.K., "Measurements of a Distributed File System", *Proceedings of the 13th Symposium on Operating System Principles*, Monterey, CA, October 1991, pp.198-212.

[Baker *et al.* 92] Baker, M., Asami, S., Deprit, E., Ousterhout, J., and Seltzer, M., "Non-Volatile Memory for Fast, Reliable File Systems", *Proceedings of the 5th International Conference on Architecture Support for Programming Languages and Operating Systems*, Boston, MA, October 1992, pp.10-22.

[Birrell *et al.* 93] Birrell, A.D., Hisgen, A., Jerian, C., Mann, T., and Swart, G., "The Echo Distributed File System", *SRC Research Report 111*, Digital Equipment Corporation, September 1993.

[Biswas *et al.* 93a] Biswas, P., Ramakrishnan, K.K., and Towsley, D., "Trace Driven Analysis of Write Policies for Disk", *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Santa Clara, CA, May, 1993, pp.13-23.

[Biswas *et al.* 93b] Biswas, P., Ramakrishnan, K.K., Towsley, D., and Krishna, C.M., "Performance Analysis of Distributed File Systems with Non-Volatile Caches", *Proceedings of the 2nd International Symposium on High Performance Distributed Computing*, Spokane, Washington, July, 1993, pp.252-262.

[Blumson 92] Blumson, S., "AFS Write Performance – A Campaign Paper", *Center for Information Technology Integration Technical Report 92-7*, University of Michigan, December, 1992.

[Burrows *et al.* 92] Burrows, M., Jerian, C., Lampson, B., and Mann, T., "On-line Data Compression in a Log-structured File System", *Proceedings of the 5th International Conference on Architecture Support for Programming Languages and Operating Systems*, Boston, MA, October 1992, pp.10-22.

[Copeland *et al.* 89] Copeland, G., Keller, T., Krishnamurthy, R., and Smith, M., "The Case for Safe RAM", *Proceedings of the 15th International Conference on Very Large Data Bases*, Amsterdam, August, 1989, pp.327-335.

[GED 90] American National Standard for Information Systems – SMALL COMPUTER SYSTEM INTERFACE - 2 (SCSI-2), Global Engineering Documents, Revision 10c, Irvine, CA, 1990.

[Hisgen *et al.* 89] Hisgen, A., Birrell, A., Mann, T., Schroeder, M., and Swart, R., "Availability and Consistency Tradeoffs in the Echo Distributed File System", *Proceedings of the Second Workshop on Workstation Operating Systems*, Pacific Grove, CA, September 1989, pp.49-54.

[Howard *et al.* 88] Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., and West, M.J., "Scale and Performance in a Distributed File System", *ACM Transactions on Computer Systems*, Vol.6, No.1, February 1988, pp.51-81.

[Jain 91] Jain, R., *The Art of Computer Systems Performance Analysis*, Wiley, 1991.

[Jouppi 91] Jouppi, N.P., "Cache Write Polices and Performance", *Digital Western Research Laboratory Research Report 91/12*, December 1991.

[Juszczak 89] Juszczak, C., "Improving the Performance and Correctness of an NFS Server", *Proceedings of the Winter 1989 USENIX Conference*, San Diego, CA, February 1989, pp.10-22.

[Kistler & Satyanarayanan 92] Kistler, J.J, and Satyanarayanan, M., "Disconnected Operation in the Coda File System", *ACM Transactions on Computer Systems*, Vol.10, No.1, February 1992, pp.3-25.

[Kleiman 86] Kleiman, S.R., "Vnodes: an Architecture for Multiple File System Types in Sun UNIX", *Proceedings of the Summer 1986 USENIX Conference*, Atlanta, GA, June 1986, pp.238-247.

[Korner 90] Korner, K., "Intelligent Caching for Remote File Service", *The 10th International Conference on Distributed Computing Systems*, Paris, France, May 1990, pp.220-226.

[Lazowska *et al.* 86] Lazowska, E.D., Zahorjan, J., Cheriton, D.R., and Zwaenepoel, W., "File Access Performance of Diskless Workstations", *ACM Transactions on Computer Systems*, Vol.4, No.3, August 1986, pp.238-268.

[Leffler *et al.* 89] Leffler, S.J., McKusick, M.K., Karels, M.J., and Quarterman, J.S., *4.3BSD UNIX Operating System*, Addison-Wesley, 1989.

[Levy & Silberschatz 90] Levy, E., and Silberschatz, A., "Distributed File Systems: Concepts and Examples", *ACM Computer Surveys*, Vol.22, No.4, December 1990, pp.321-374.

[Makaroff 88] Makaroff, D., *Disk Cache Management for Distributed Systems*, M.Sc. Thesis, Department of Computational Science, University of Saskatchewan, Saskatoon, Saskatchewan, 1988.

[Mann *et al.* 93] Mann, T., Birrell, A., Hisgen, A., Jerian, C., and Swart, G., "A Coherent Distributed File Cache with Directory Write-behind", *SRC Research Report 103*, Digital Equipment Corporation, June 1993.

[McKusick *et al.* 84] McKusick, M.K., Joy, W.N., Leffler, S.J., and Fabry, R.S., "A Fast File System for UNIX", *ACM Transactions on Computer Systems*, Vol.2, No.3, August 1984, pp.181-197.

[Moran *et al.* 90] Moran, J., Sandberg, R., Coleman, D., Kepecs, J., Lyon, B., "Breaking Through the NFS Performance Barrier", *Proceedings of the European Unix Users Group Spring 1990 Conference*, Munich, Germany, April 1990, pp.199-206.

[Morris 93] Morris, R.J., "Analysis of Superposition of Streams into a Cache Buffer", *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Santa Clara, CA, May 1993, pp.229-235.

[NCR 90] NCR Corporation, *Scsi – Understanding the Small Computer System Interface*, Prentice Hall, 1990.

[Nelson *et al.* 88] Nelson, M.N., Welch, B.B., and Ousterhout, J.K., "Caching in the Sprite Network File System", *ACM Transactions on Computer Systems*, Vol.6, No.1, February 1988, pp.134-154.

[Ousterhout *et al.* 85] Ousterhout, J.K., Costa, H., Harrison, D., Kunze, J.A., and Thompson, J.G., "A Trace-Driven Analysis of the UNIX 4.2 BSD File System", *Proceedings of the 10th Symposium on Operating System Principles*, Orcas Island, WA, December 1985, pp.15-24.

[Ramakrishnan *et al.* 92] Ramakrishnan, K.K., Biswas, P., and Karedla, R., "Analysis of File I/O Traces in Commercial Computing Environments", *Proceedings of the 1992 ACM SIGMETRICS and PERFORMANCE '92 International Conference on Measurement and Modeling of Computer Systems*, Newport, R.I., June 1992, pp.78-90.

[Rangan & Harrick 91] Rangan, P.V., and Harrick, M.V., "Designing File Systems for Digital Video and Audio", *Proceedings of the 13th ACM Symposium on Operating System Principles*, Pacific Grove, CA, October 1991, pp.81-94.

[Robinson & Devarakonda 90] Robinson, J.T., and Devarakonda, M.V., "Data Cache Management Using Frequency-Based Replacement", *Proceedings of the 1990*

*ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Boulder, Colorado, May 1990, pp.134-142.

[Rosenblum & Ousterhout 91] Rosenblum, M., and Ousterhout, J.K., "The Design and Implementation of a Log-Structured File System", *Proceedings of the 13th ACM Symposium on Operating System Principles*, Pacific Grove, CA, October 1991, pp.1-15.

[Rosenthal 90] Rosenthal, D.S., "Evolving the Vnode Interface", *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, CA, June 1990, pp.107-117.

[Ruemmler & Wilkes 93a] Ruemmler, C., and Wilkes, J., "UNIX Disk Access Patterns", *Proceedings of the Winter 1993 USENIX Conference*, San Diego, CA, January 1993, pp.405-420.

[Ruemmler & Wilkes 93b] Ruemmler, C., and Wilkes, J., "Modeling Disks", *HP Laboratories Technical Report HPL-93-68*, July 1993.

[Sandberg *et al.* 85] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B., "Design and Implementation of the Sun Network Filesystem", *Proceedings of the Summer 1985 USENIX Conference*, Portland, OR, June 1985, pp.119-130.

[Satyanarayanan *et al.* 90a] Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, W.H., and Streere, D.C., "Coda: A Highly Available File System for a Distributed Workstation Environment", *IEEE Transactions on Computers*, Vol.39, No.4, April 1990, pp.447-459.

[Satyanarayanan 90b] Satyanarayanan, M., "Scalable, Secure, and Highly Available Distributed File Access", *IEEE Computer*, Vol.23, No.5, May 1990, pp.9-21.

[Seltzer *et al.* 90] Seltzer, M., Chen, P., and Ousterhout, J., "Disk Scheduling Revisited", *Proceedings of the Winter 1990 USENIX Conference*, Washington, DC, January 1990, pp.313-324.

[Siegel 92] Siegel, A., *Performance in Flexible Distributed File Systems*, Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1992.

[Silberschatz & Peterson 91] Silberschatz, A., and Peterson, P.G., *Operating System Concepts*, (3rd ed.) Addison-Wesley, 1990.

[Solworth & Orji 90] Solworth, J.A., and Orji, C.U., "Write-Only Disk Caches", *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, Atlantic City, NJ, May, 1990, pp.123-132.

[Smith 82] Smith, A.J., "Cache Memories", *Computing Surveys*, Vol.14, No.3, September 1982, pp.473-530.

[Smith 85] Smith, A.J., "Disk Cache – Miss Ratio Analysis and Design Considerations", *ACM Transactions on Computing Systems*, Vol.3, No.3, August 1985, pp.161-203.

[Srinivasan 89] Srinivasan, V., "Spritely NFS: Experiments with Cache-Consistency Protocols", *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, Litchfield Park, Arizona, December 1989, pp.45-57.

[Stone 90] Stone, H.S., *High-Performance Computer Architecture*, Addison-Wesley, 1990.

[Subramanian 92] Subramanian, R., "Workload Characterization of AFS File Servers", *Center for Information Technology Integration Technical Report 92-6*, University of Michigan, November 23, 1992.

[Svobodova 84] Svobodova, L., "File Servers for Network-Based Distributed Systems", *Computing Surveys*, Vol.16, No.4, December 1984, pp.353-398.

[Tanenbaum 89] Tanenbaum, A.S., *Computer Networks*, Prentice Hall, 1989.

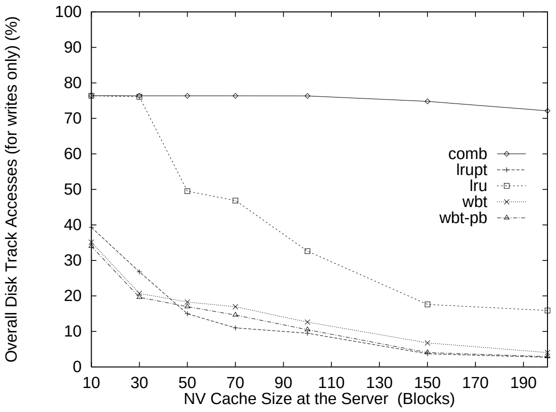
[Thekkath *et al.* 92] Thekkath, C.A., Wilkes, J., and Lazowska, E.D., "Techniques for File System Simulation", *HP Laboratories Technical Report HPL-92-131*, October 1992.

[Willick *et al.* 93] Willick, D.L., Eager, D.L., and Bunt, R.B., "Disk Cache Replacement Policies for Network Fileservers", *Proceedings of the 13th International Conference on Distributed Computer Systems*, Pittsburgh, PA, May 1993, pp.2-11.

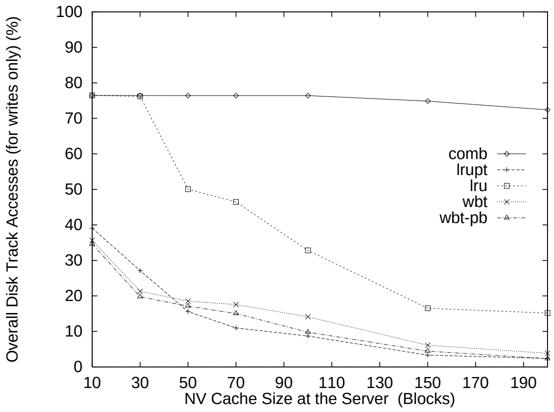
# Appendix A

# Additional Simulation Results from Trace 1

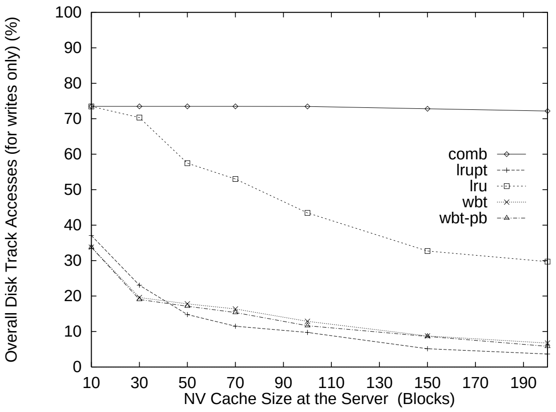
Additional results, for multilevel NV caching obtained from trace 1, are shown in this appendix. Results concerning NV cache performance on the server, and overall results, when the client NV cache size is 10 and 50 blocks, as well as results concerning server cache performance when the NV cache size on the client is 100 blocks, are shown.



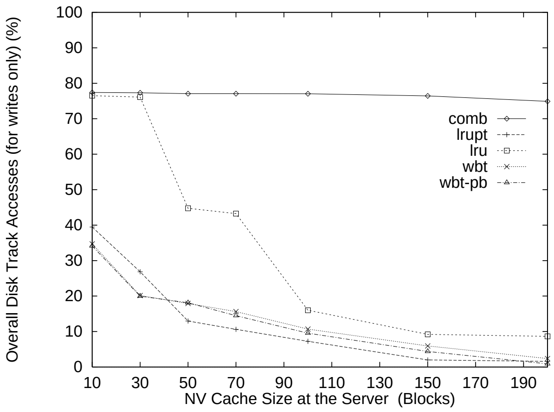

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

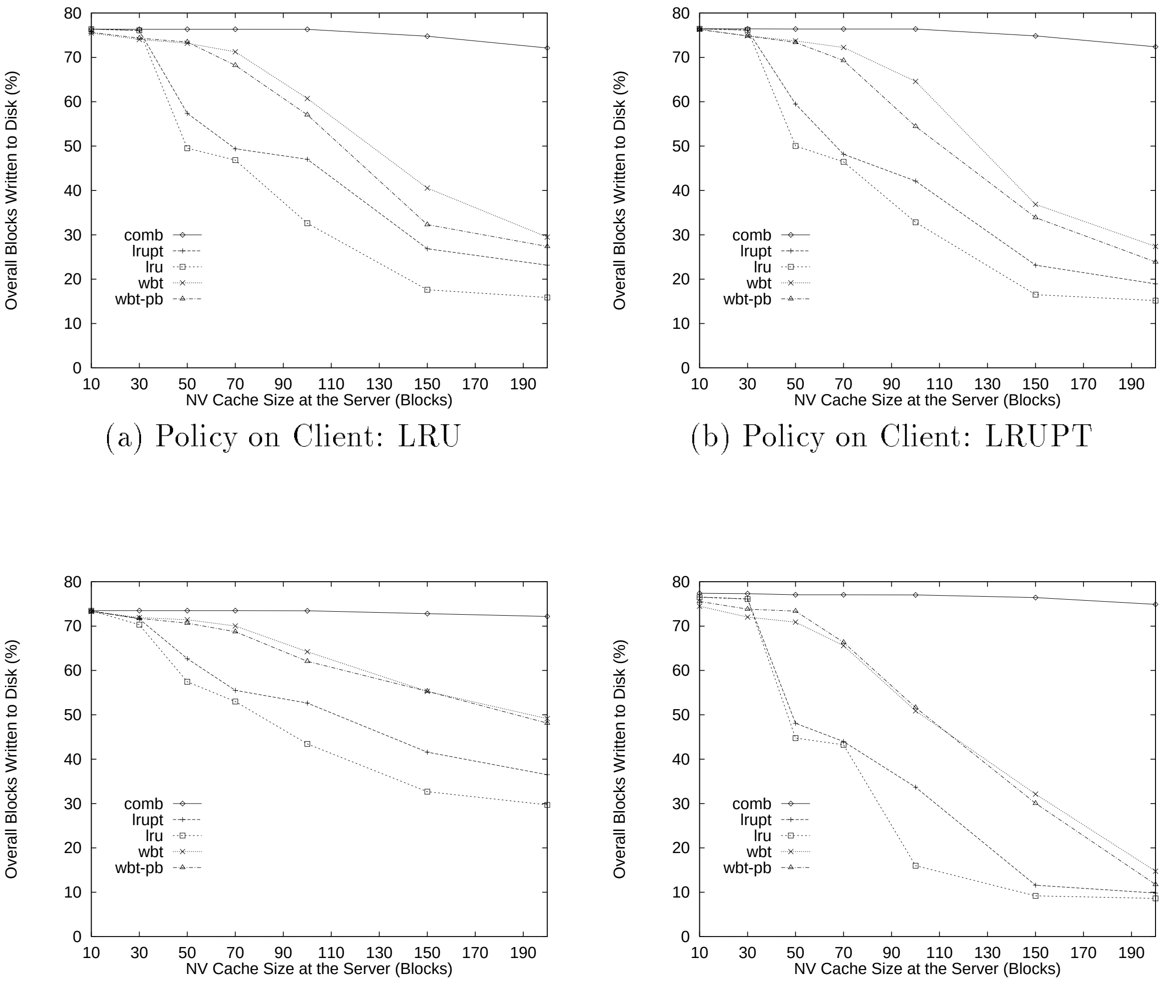



(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure A.1: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations. Client NV Cache Size is 10 Blocks



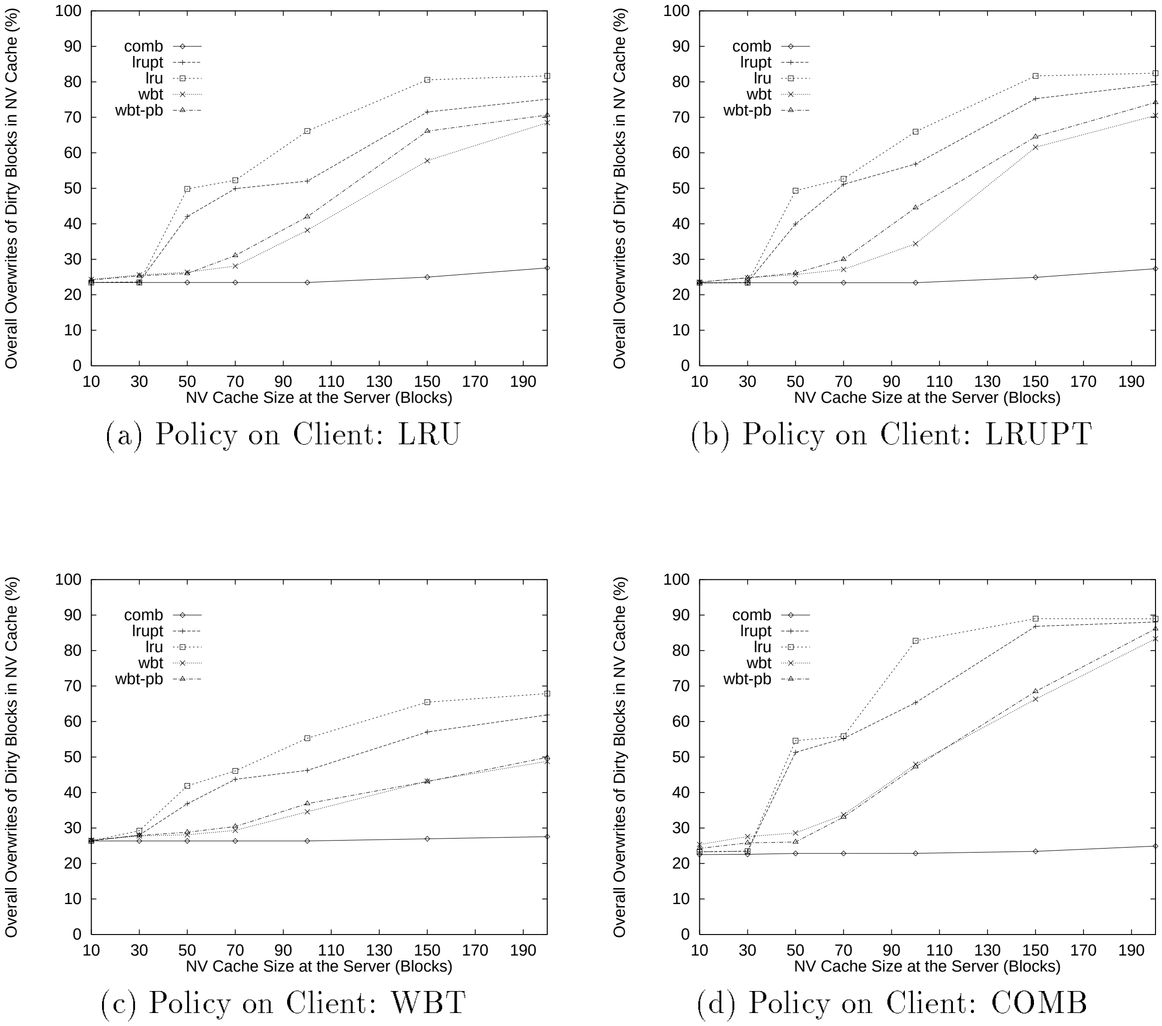

(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

(c) Policy on Client: WBT

(d) Policy on Client: COMB

Figure A.2: Blocks Written to Disk, as a Percentage of Total Write Operations. Client NV Cache Size is 10 Blocks



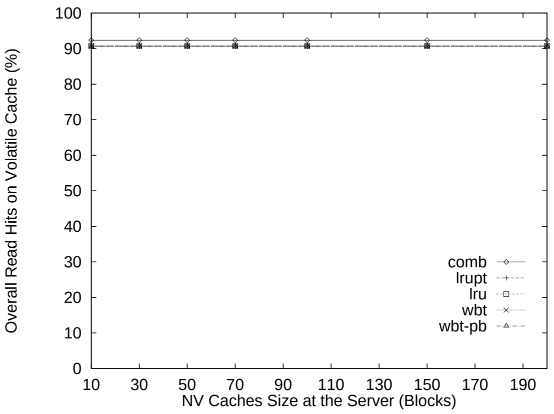

(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

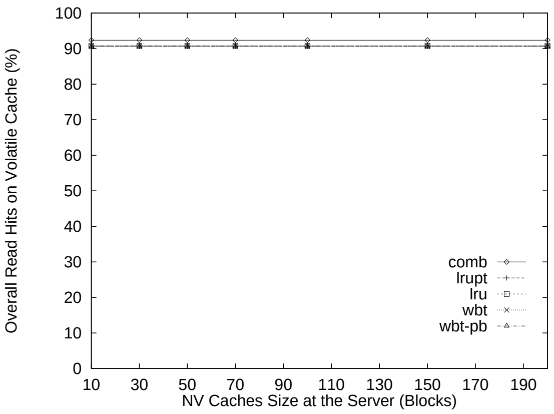
(c) Policy on Client: WBT

(d) Policy on Client: COMB

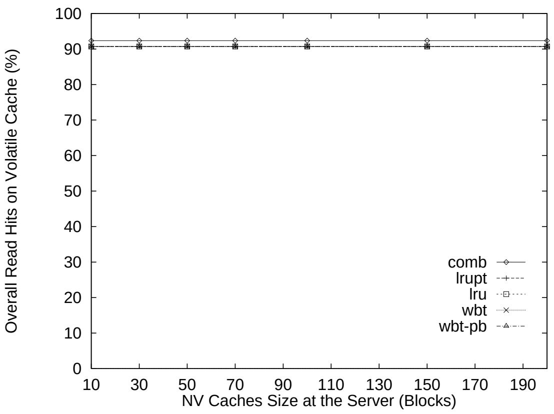
Figure A.3: Overwrites of Dirty Blocks in NV Caches, as a Percentage of the Total Write Operations. Client NV Cache Size is 10 Blocks



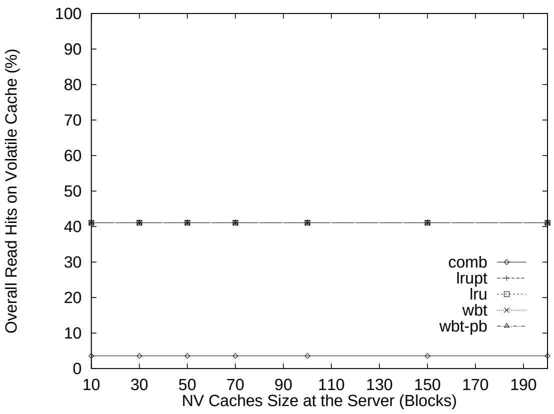

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

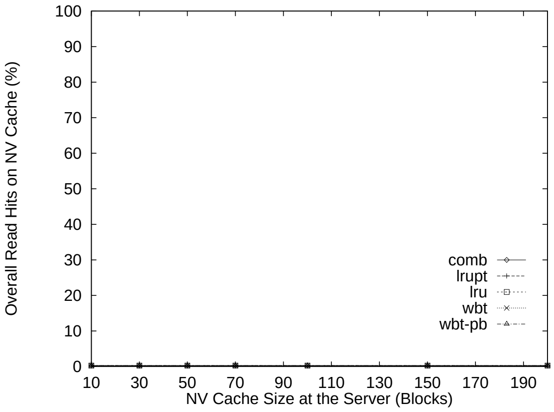



(c) Policy on Client: WBT

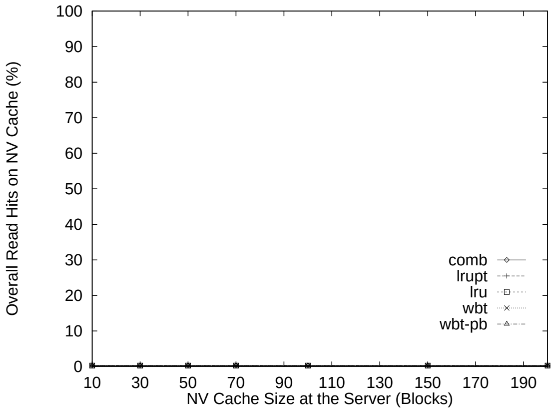



(d) Policy on Client: COMB

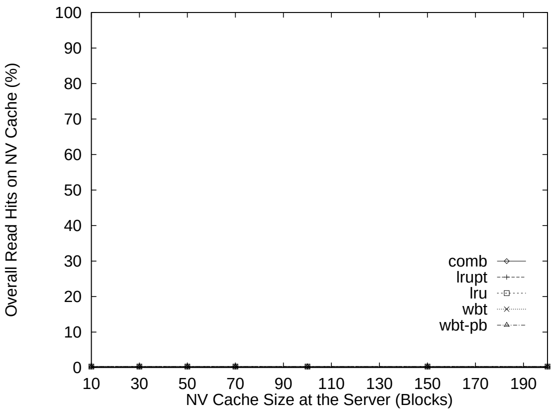
Figure A.4: Overall Read Hit Ratio on Volatile Caches. Client NV Cache Size is 10 Blocks



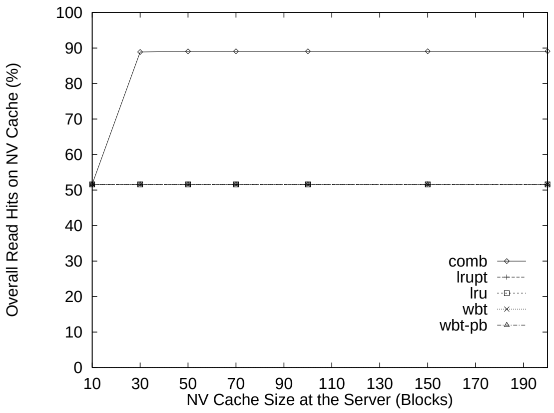

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

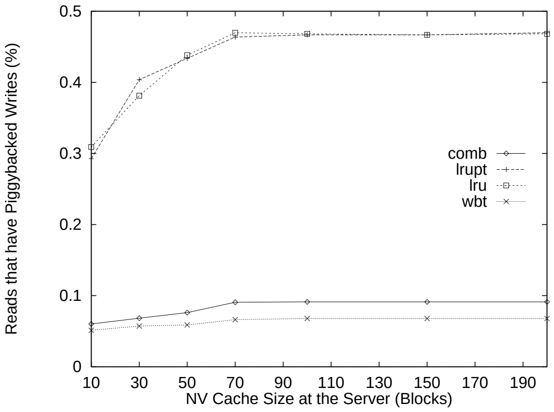



(c) Policy on Client: WBT

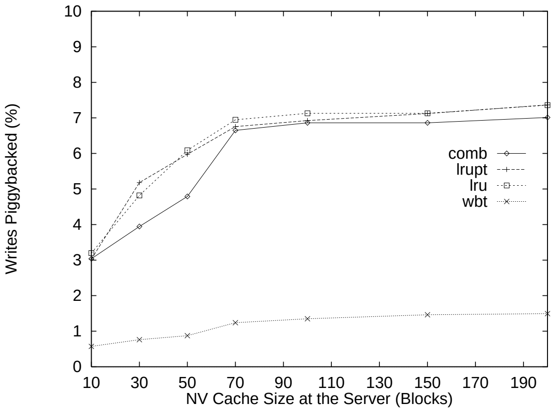



(d) Policy on Client: COMB

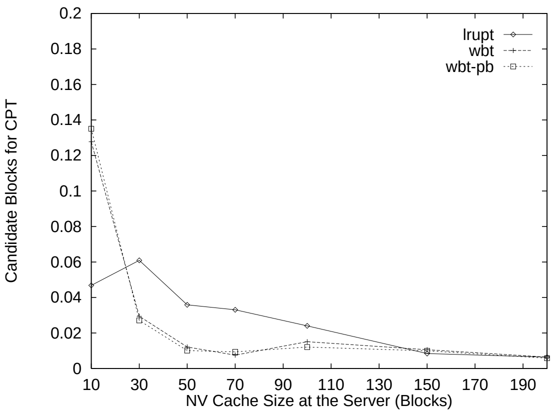
Figure A.5: Overall Read Hit Ratio on NV Caches. Client NV Cache Size is 10 Blocks



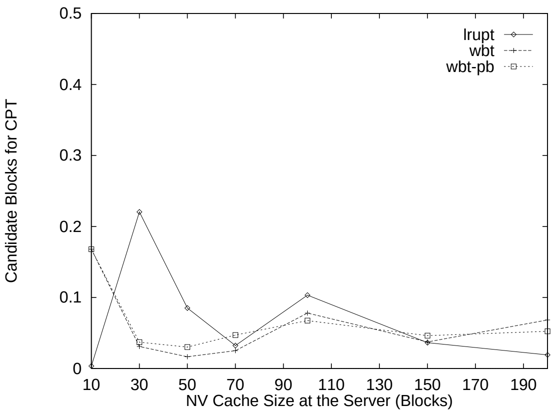

(a) Reads Misses that have Writes Piggybacked




(b) Writes Piggybacked

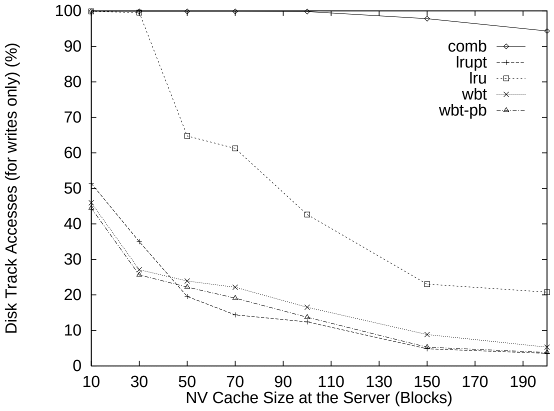



(c) Candidate BLocks for CPT, Client Policy is LRUPT

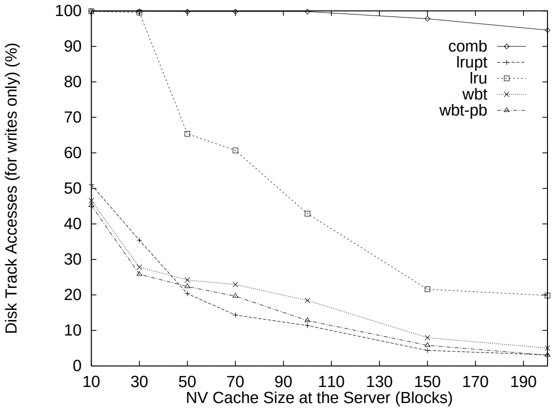



(d) Candidate for CPT, Client Policy is WBT

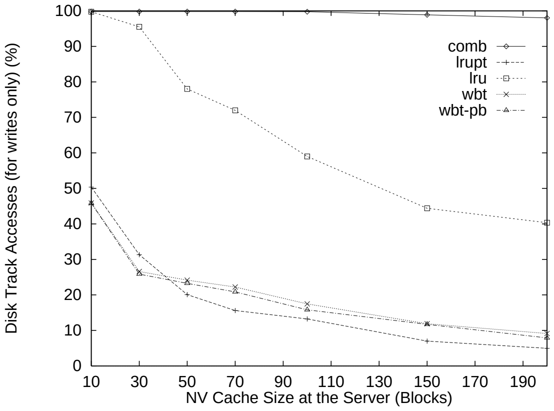
Figure A.6: Other Statistics. Client NV Cache Size is 10 Blocks



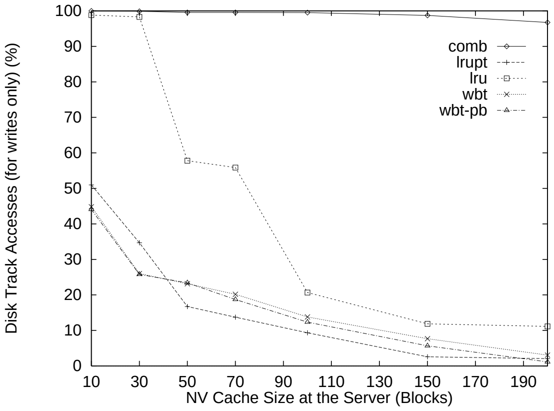

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

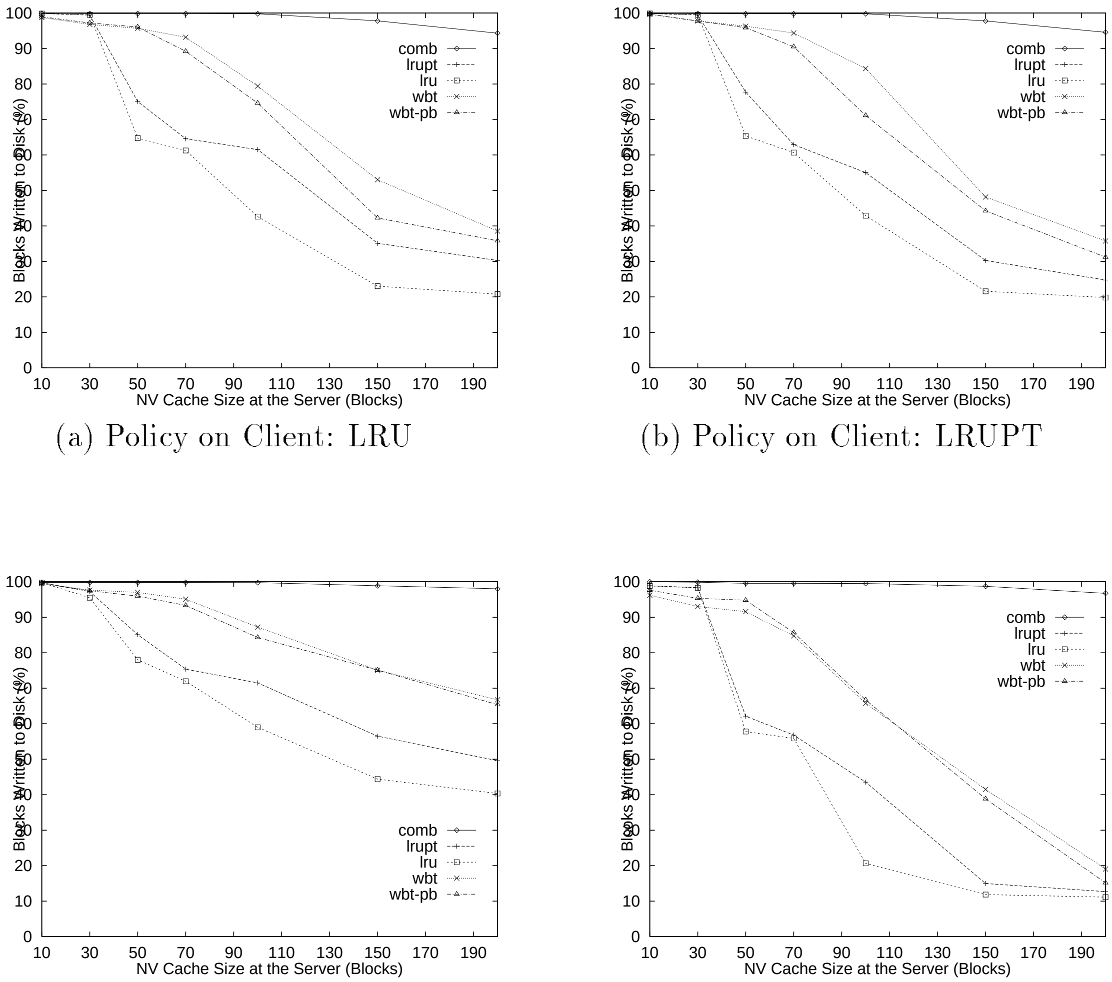



(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure A.7: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache Size is 10 Blocks



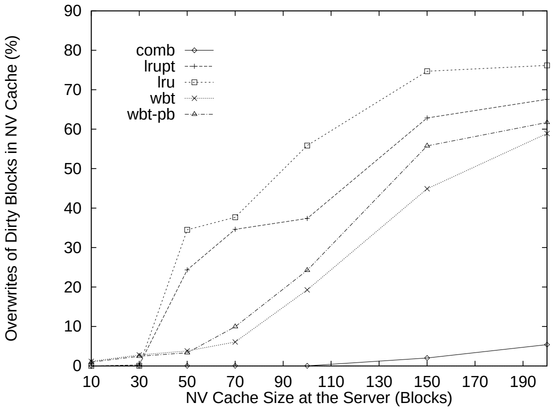

(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

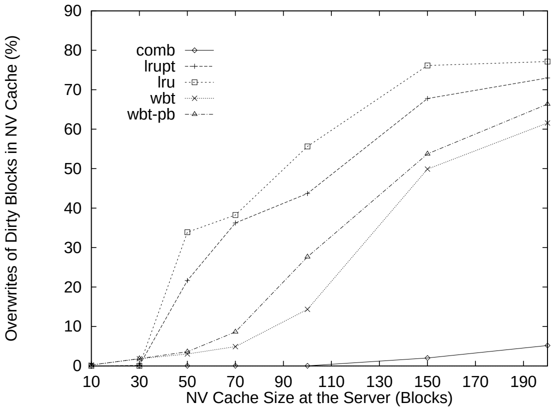
(c) Policy on Client: WBT

(d) Policy on Client: COMB

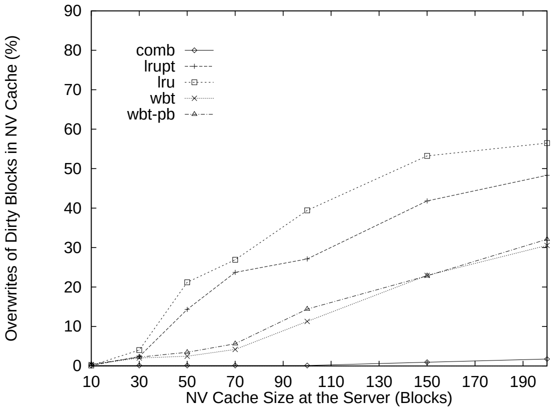
Figure A.8: Blocks Written to Disk, as a Percentage of Total Write Operations Seen at the Server. Client NV Cache Size is 10 Blocks



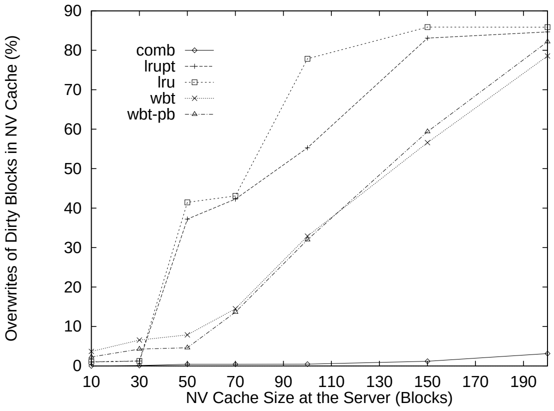

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

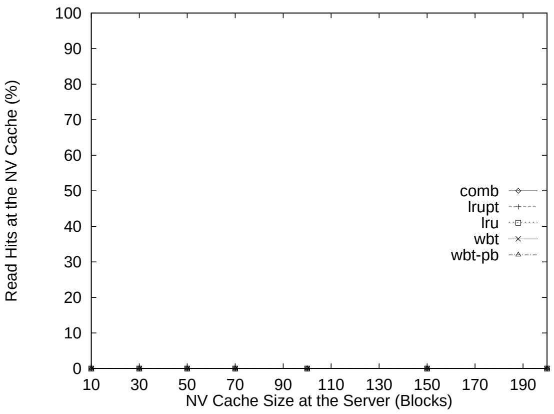



(c) Policy on Client: WBT

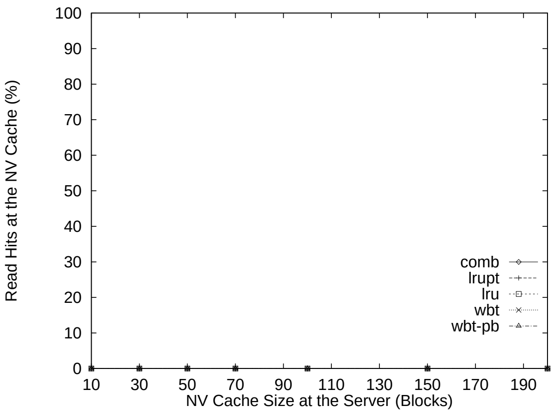



(d) Policy on Client: COMB

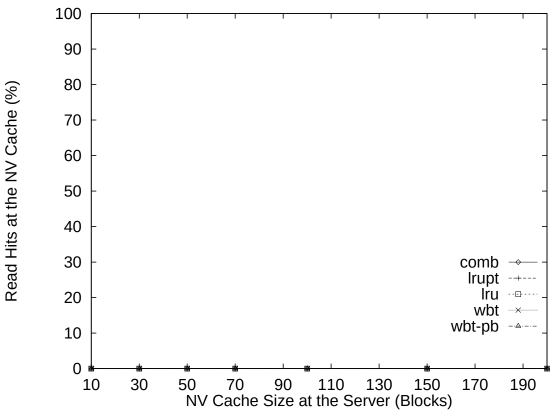
Figure A.9: Overwrites of Dirty Blocks in Server NV Cache, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache Size is 10 Blocks



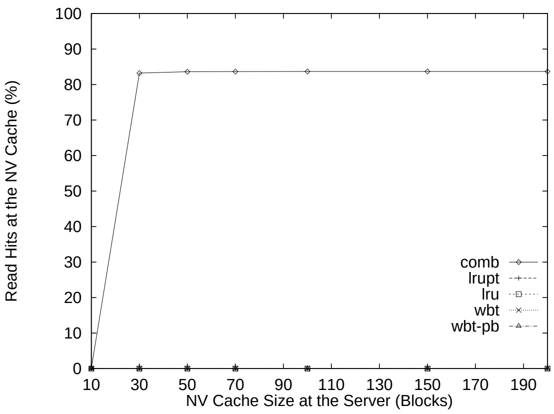

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

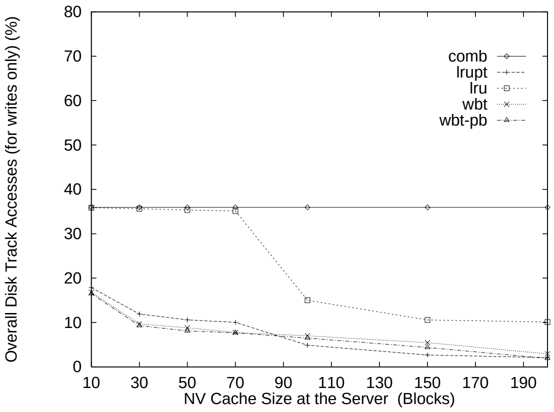



(c) Policy on Client: WBT

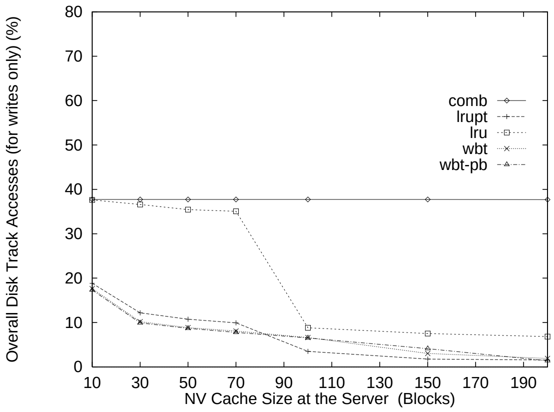



(d) Policy on Client: COMB

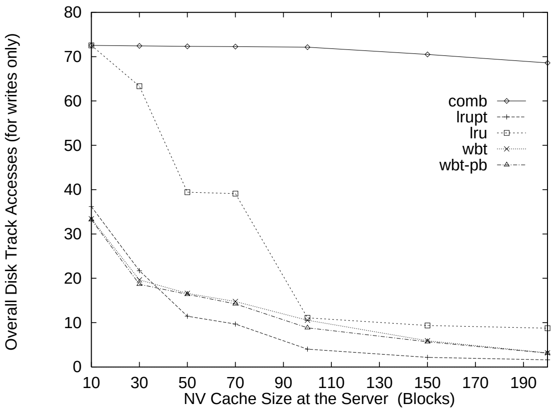
Figure A.10: Overall Read Hit Ratio on Server NV Cache. Client NV Cache Size is 10 Blocks



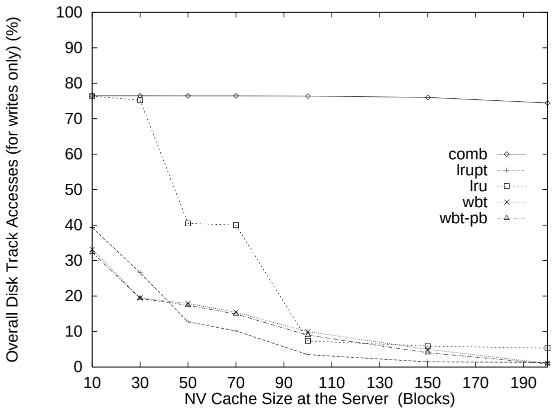

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

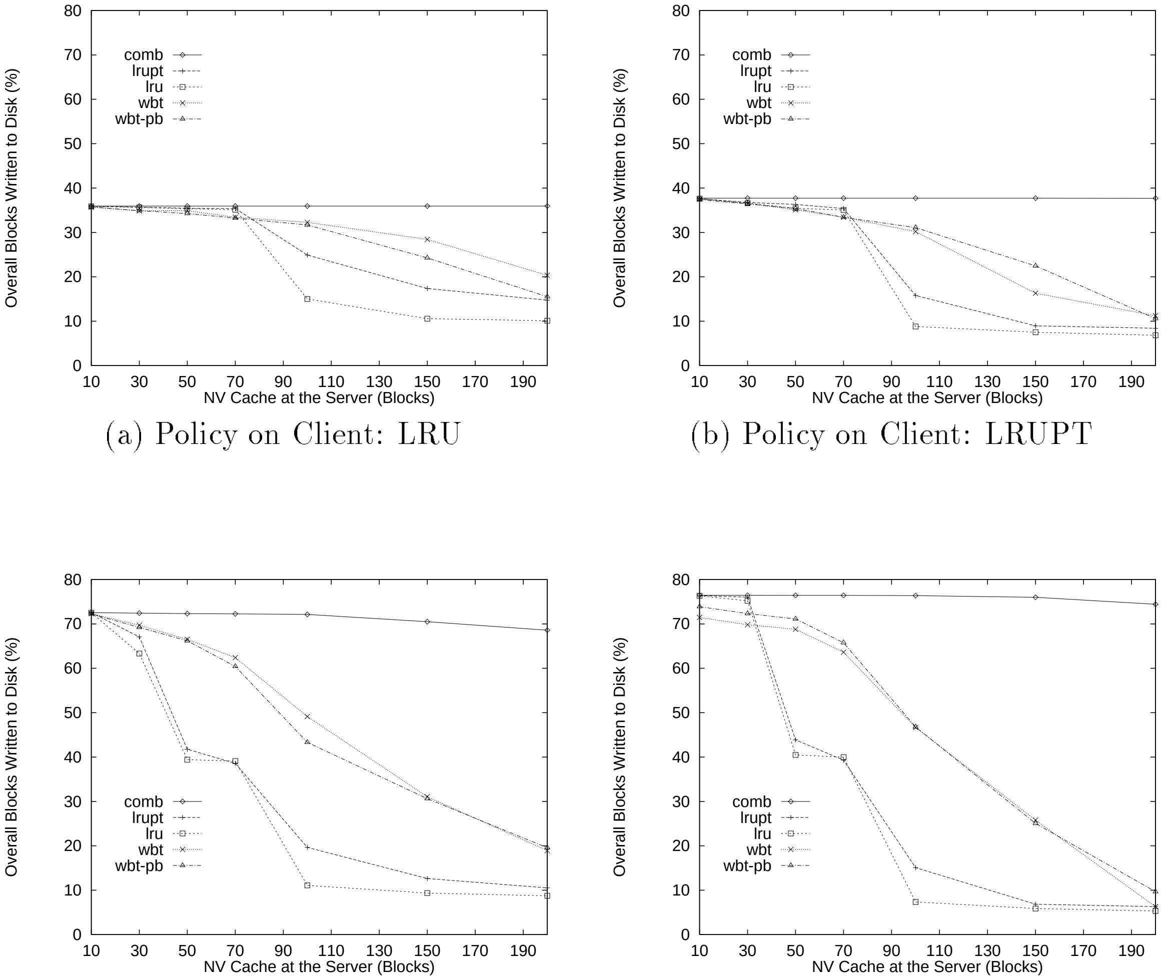



(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure A.11: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations. Client NV Cache Size is 50 Blocks



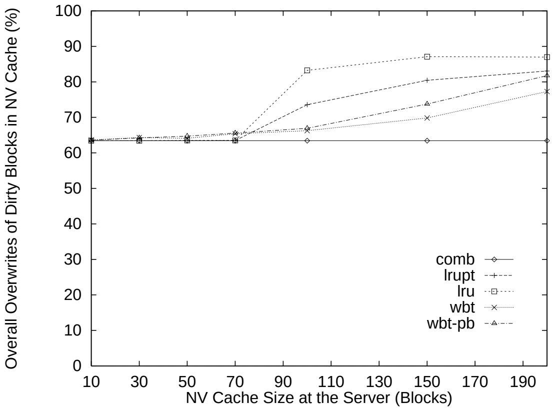

(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

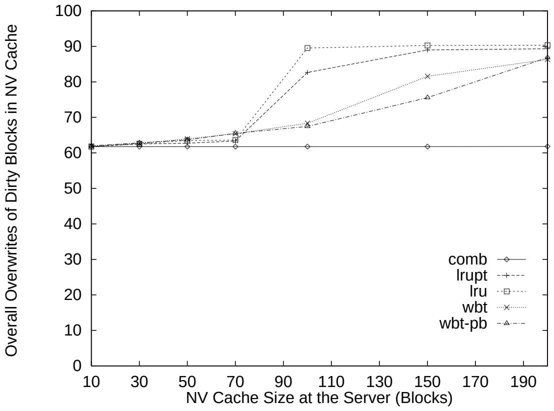
(c) Policy on Client: WBT

(d) Policy on Client: COMB

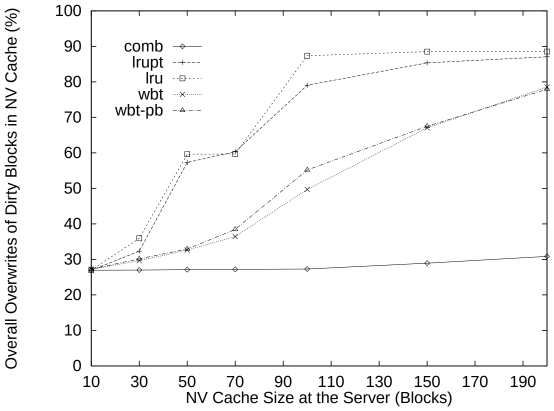
Figure A.12: Blocks Written to Disk, as a Percentage of Total Write Operations. Client NV Cache Size is 50 Blocks



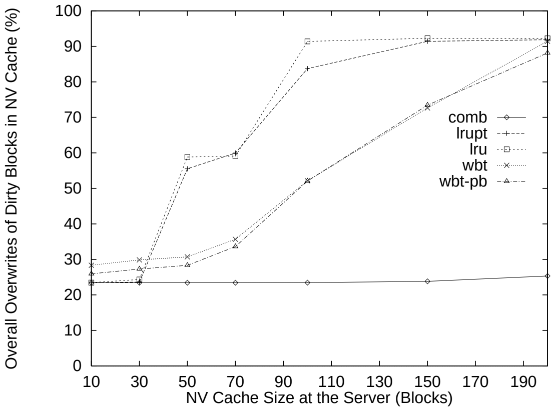

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

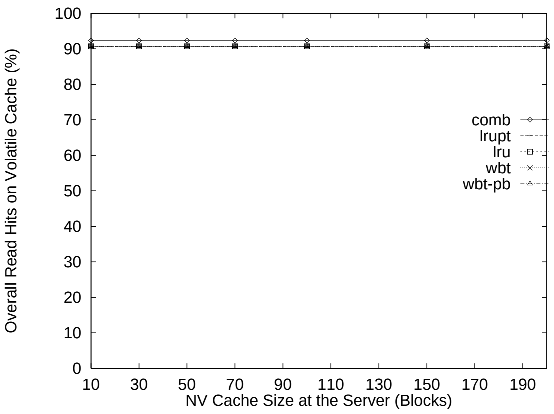



(c) Policy on Client: WBT

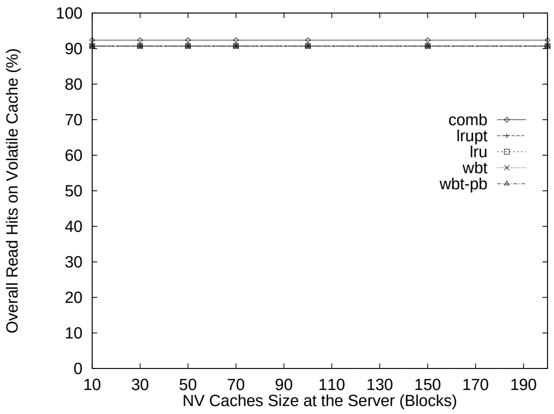



(d) Policy on Client: COMB

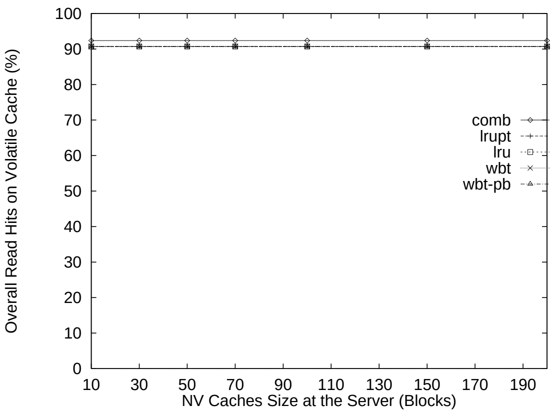
Figure A.13: Overwrites of Dirty Blocks in NV Caches, as a Percentage of the Total Write Operations. Client NV Cache Size is 50 Blocks



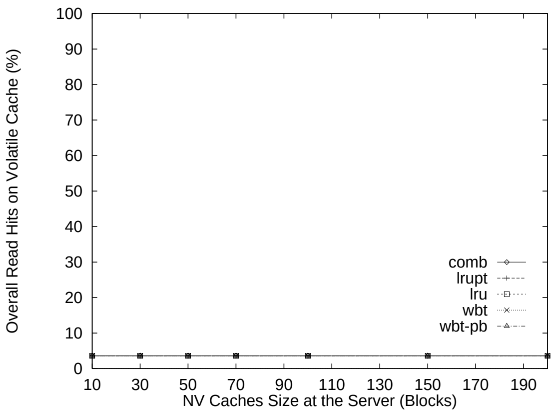

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

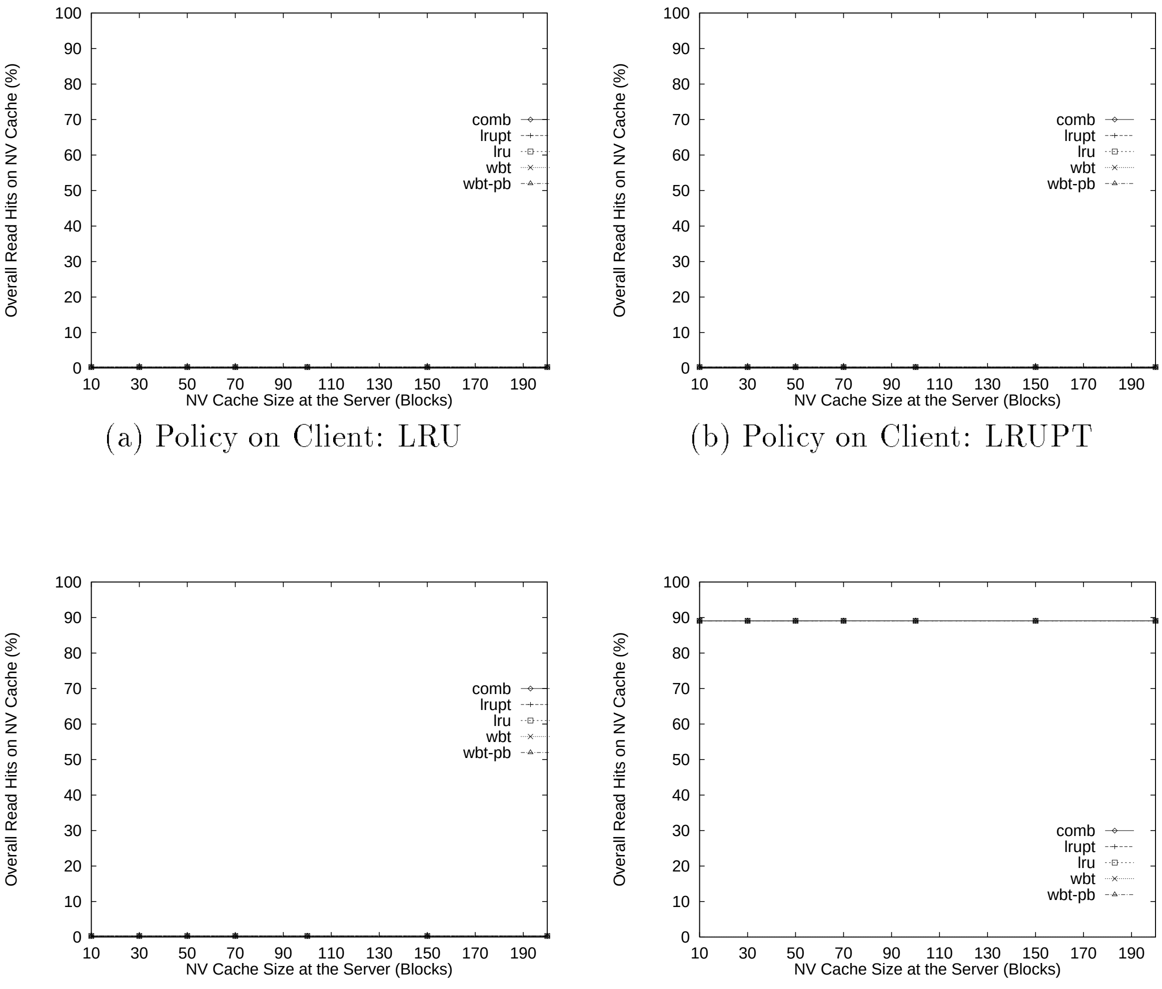



(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure A.14: Overall Read Hit Ratio on Volatile Caches. Client NV Cache Size is 50 Blocks



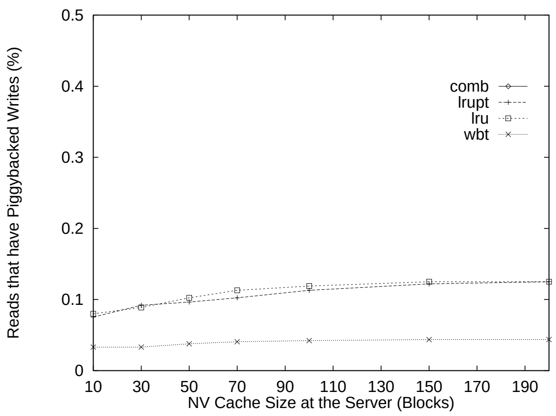

(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

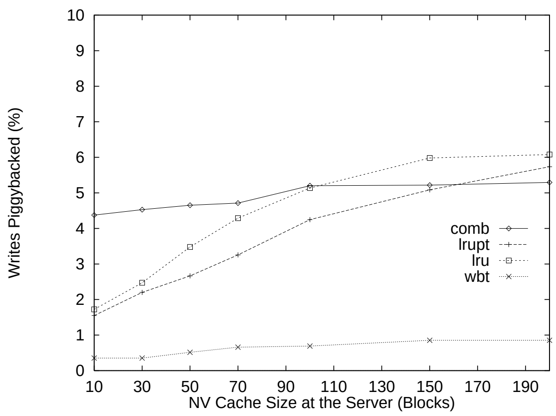
(c) Policy on Client: WBT

(d) Policy on Client: COMB

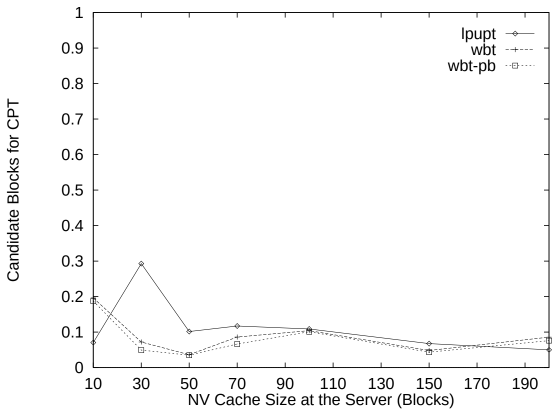
Figure A.15: Overall Read Hit Ratio on NV Caches. Client NV Cache Size is 50 Blocks



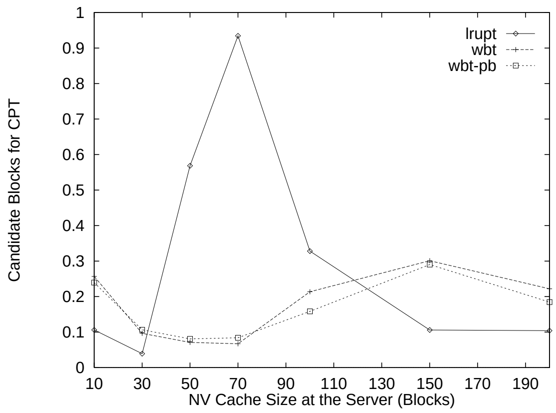

(a) Reads Misses that have Writes Piggybacked




(b) Writes Piggybacked

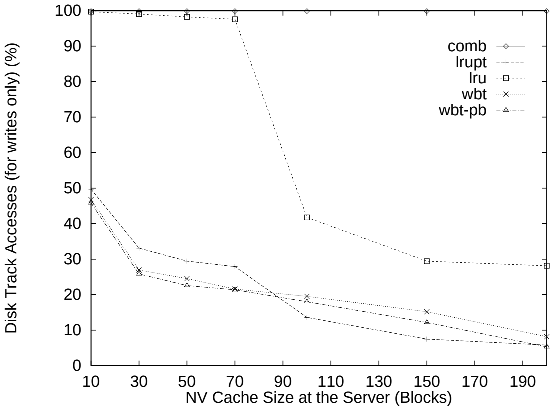



(c) Candidate BLocks for CPT, Client Policy is LRUPT

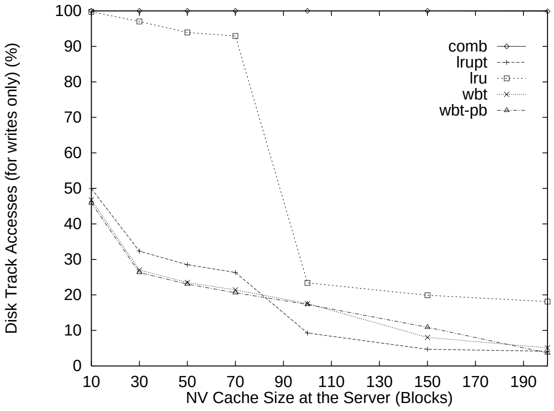



(d) Candidate for CPT, Client Policy is WBT

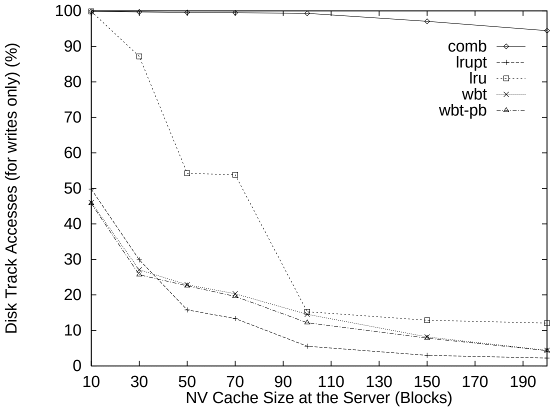
Figure A.16: Other Statistics. Client NV Cache Size is 50 Blocks



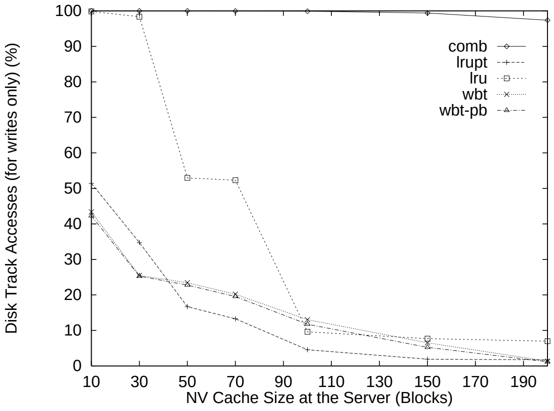

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

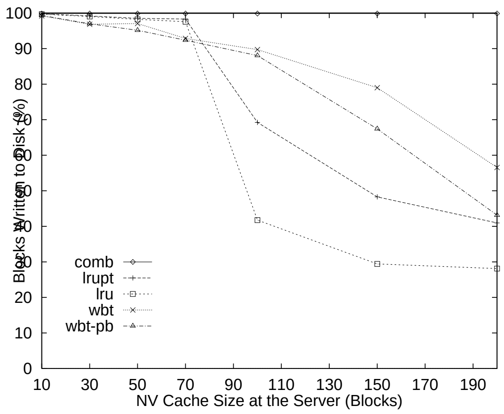



(c) Policy on Client: WBT

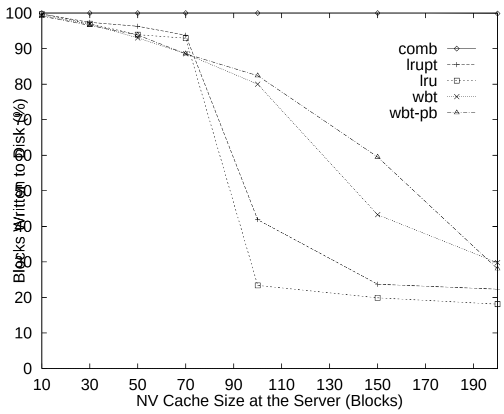



(d) Policy on Client: COMB

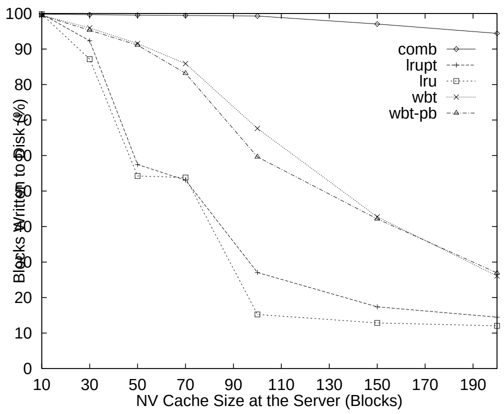
Figure A.17: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache Size is 50 Blocks



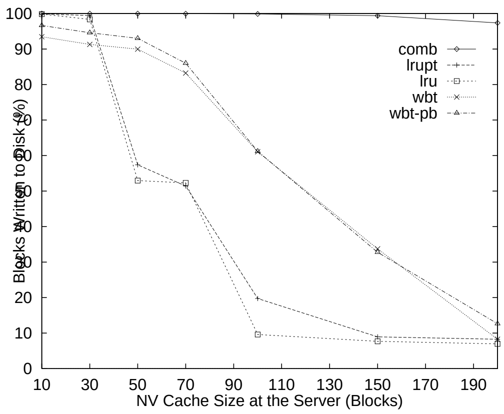

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

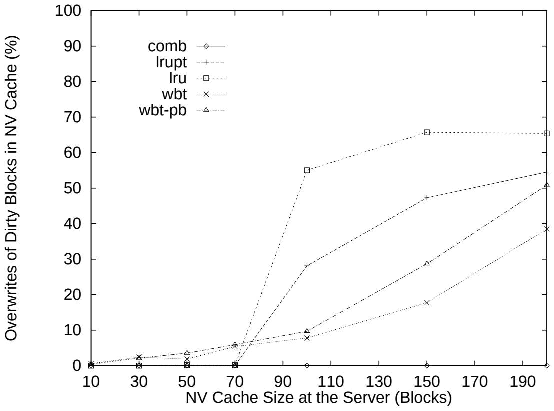



(c) Policy on Client: WBT

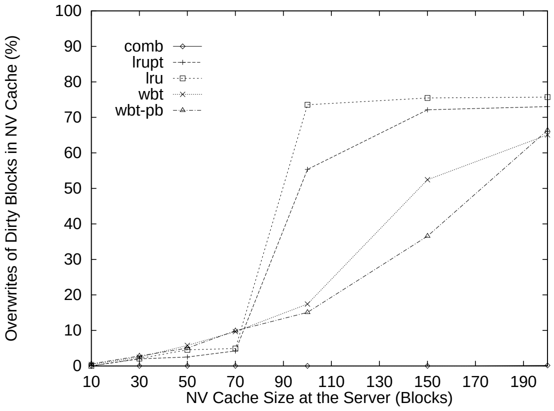



(d) Policy on Client: COMB

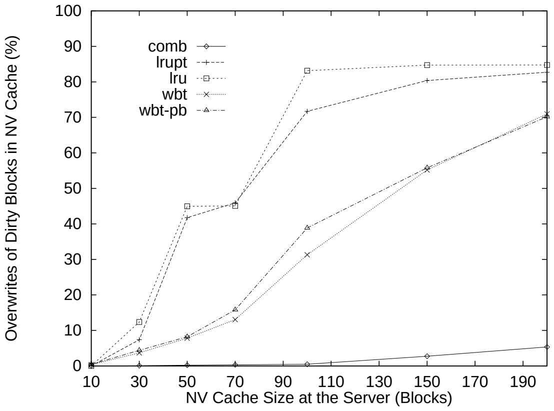
Figure A.18: Blocks Written to Disk, as a Percentage of Total Write Operations Seen at the Server. Client NV Cache Size is 50 Blocks



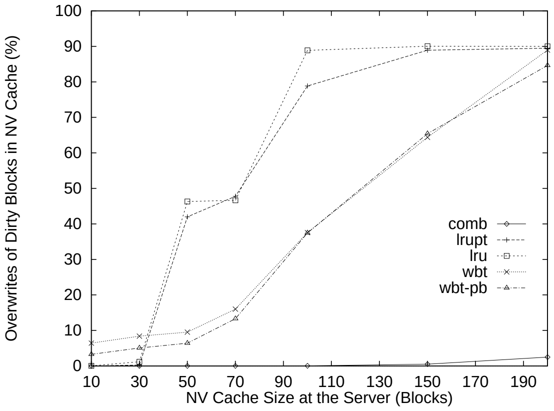

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

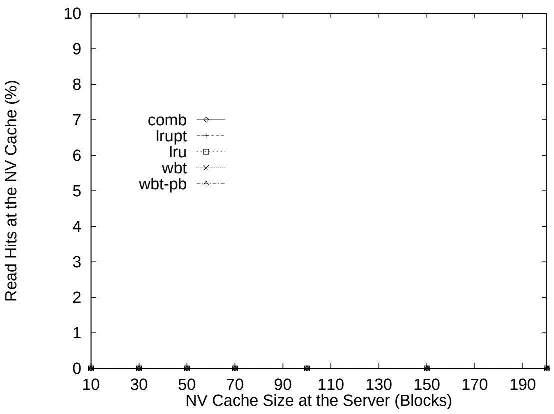



(c) Policy on Client: WBT

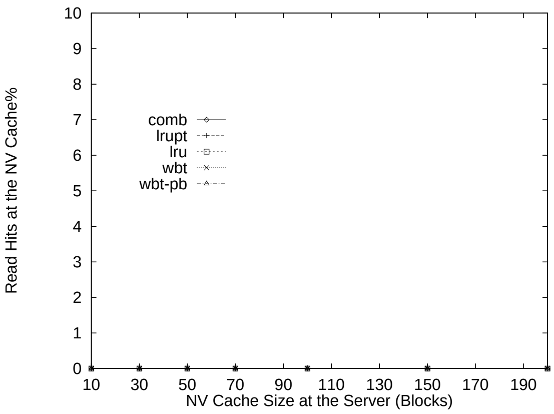



(d) Policy on Client: COMB

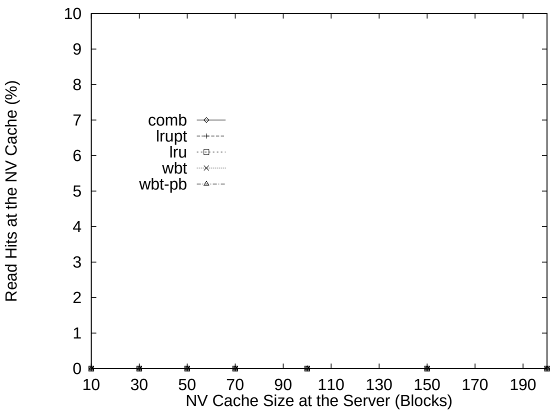
Figure A.19: Overwrites of Dirty Blocks in Server NV Cache, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache Size is 50 Blocks



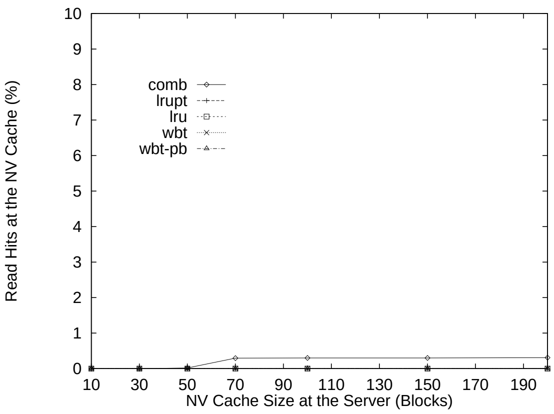

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

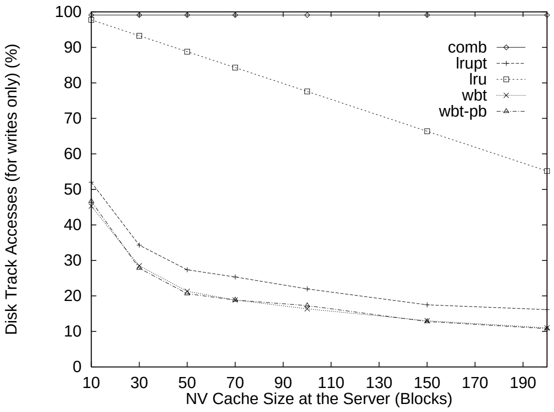



(c) Policy on Client: WBT

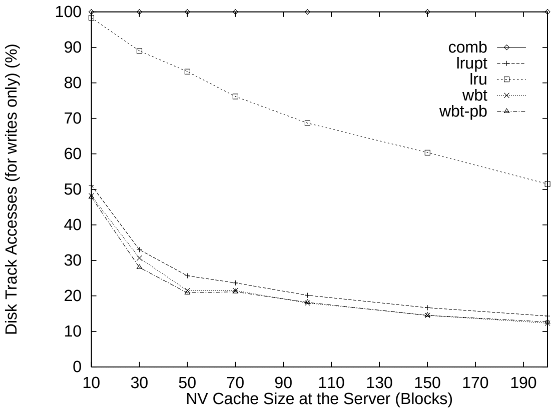



(d) Policy on Client: COMB

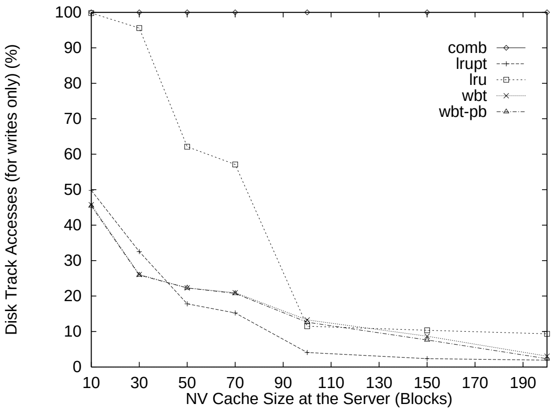
Figure A.20: Overall Read Hit Ratio on Server NV Cache. Client NV Cache Size is 50 Blocks



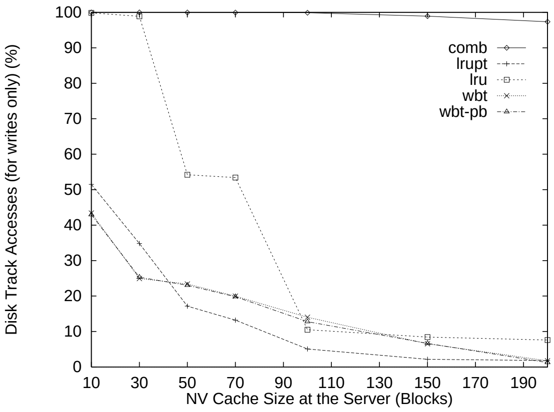

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

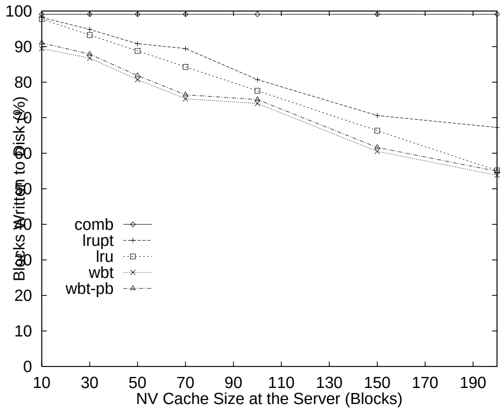



(c) Policy on Client: WBT

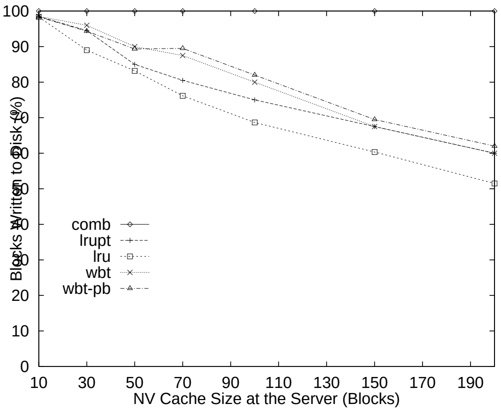



(d) Policy on Client: COMB

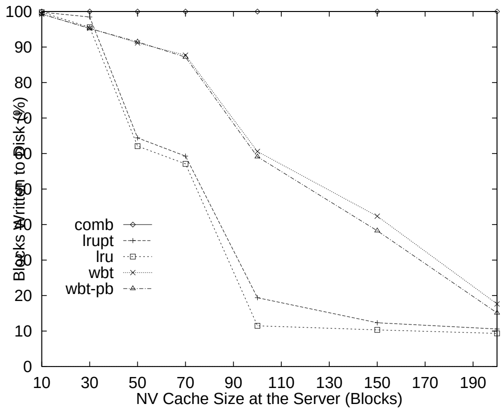
Figure A.21: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache Size is 100 Blocks



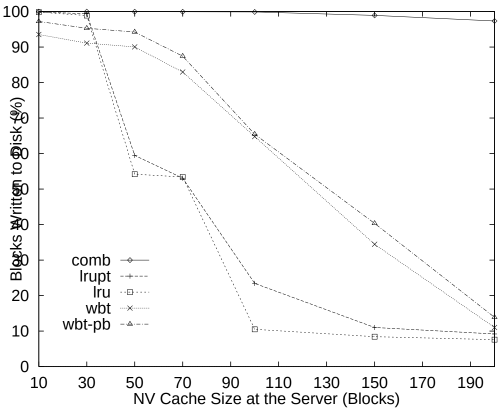

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

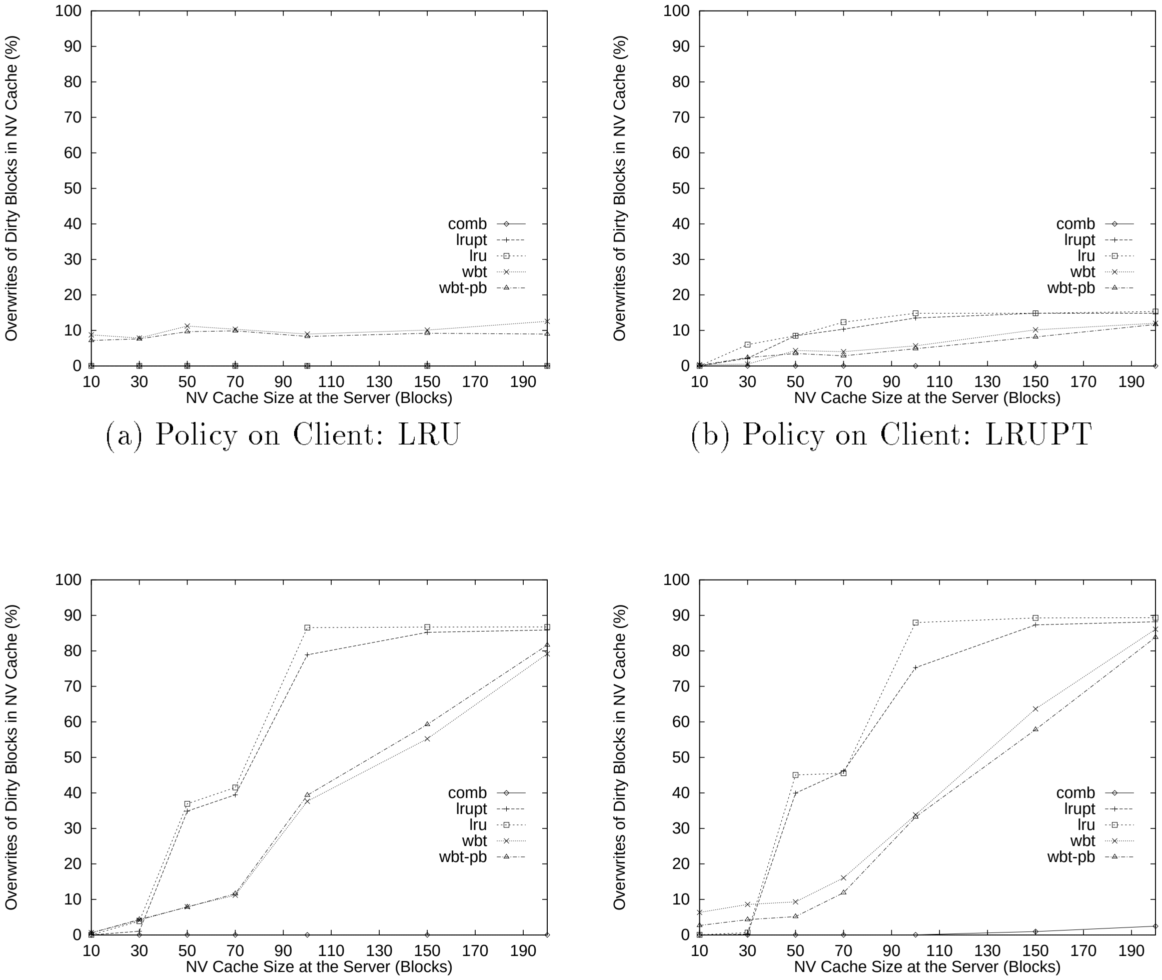



(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure A.22: Blocks Written to Disk, as a Percentage of Total Write Operations Seen at the Server. Client NV Cache Size is 100 Blocks



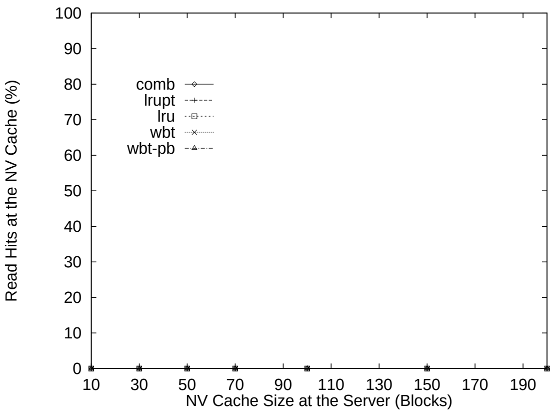

(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

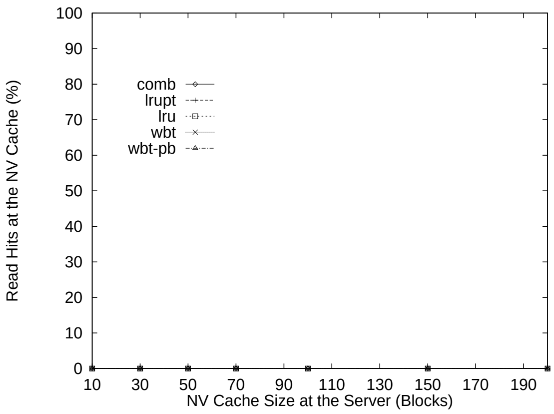
(c) Policy on Client: WBT

(d) Policy on Client: COMB

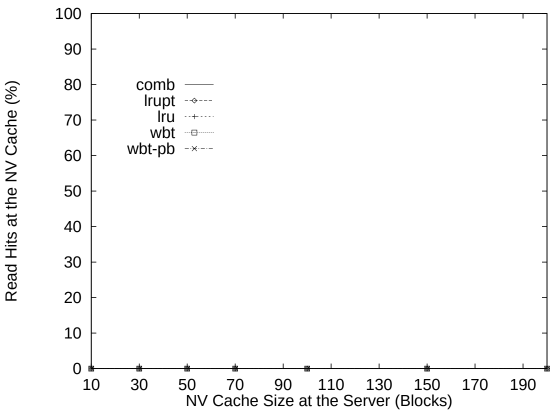
Figure A.23: Overwrites of Dirty Blocks in Server NV Cache, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache Size is 100 Blocks



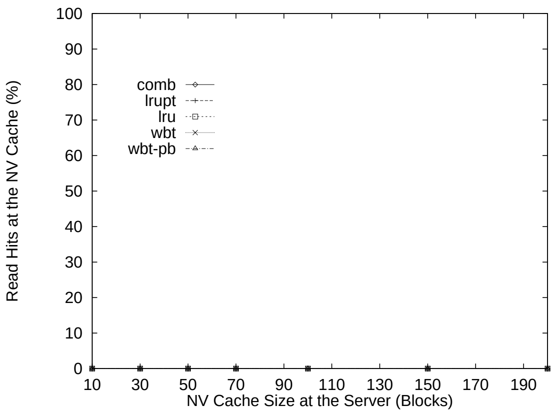

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT




(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure A.24: Overall Read Hit Ratio on Server NV Cache. Client NV Cache Size is 100 Blocks

# Appendix B

# Simulation Results from Trace 2

All results obtained from trace 2 are shown in this appendix.

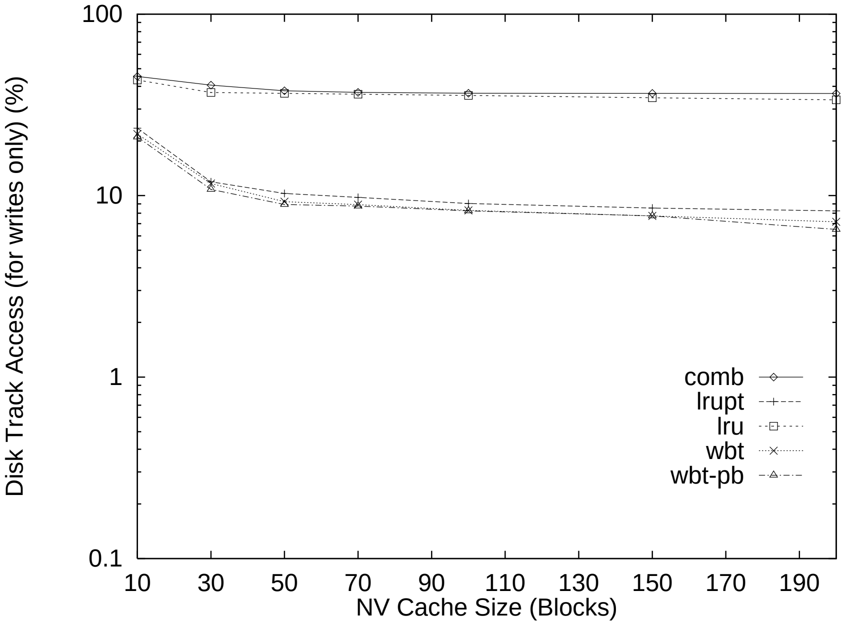



Figure B.1: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations (Single-Level NV Cache Configuration)

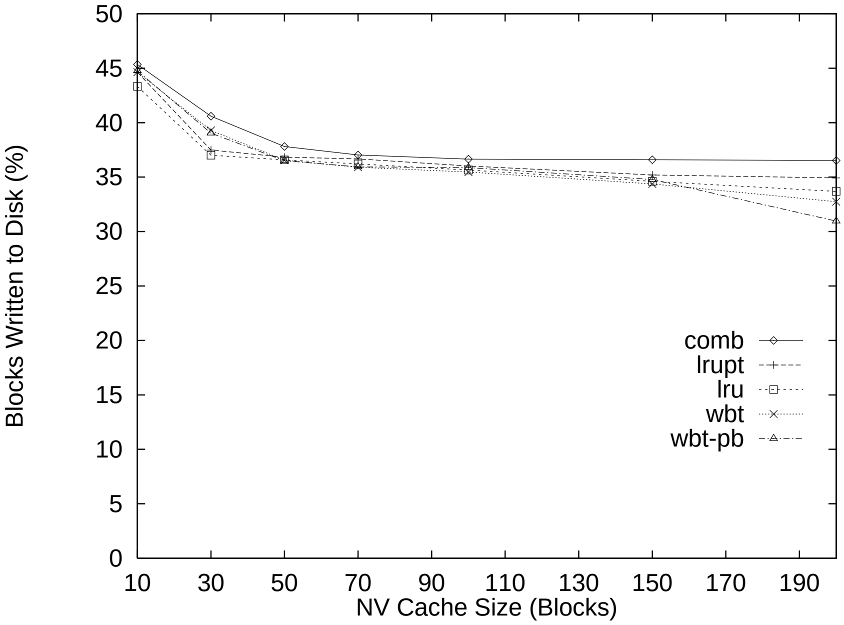



Figure B.2: Blocks Written to Disk as a Percentage of the Total Write Operations (Single-Level NV Cache Configuration)

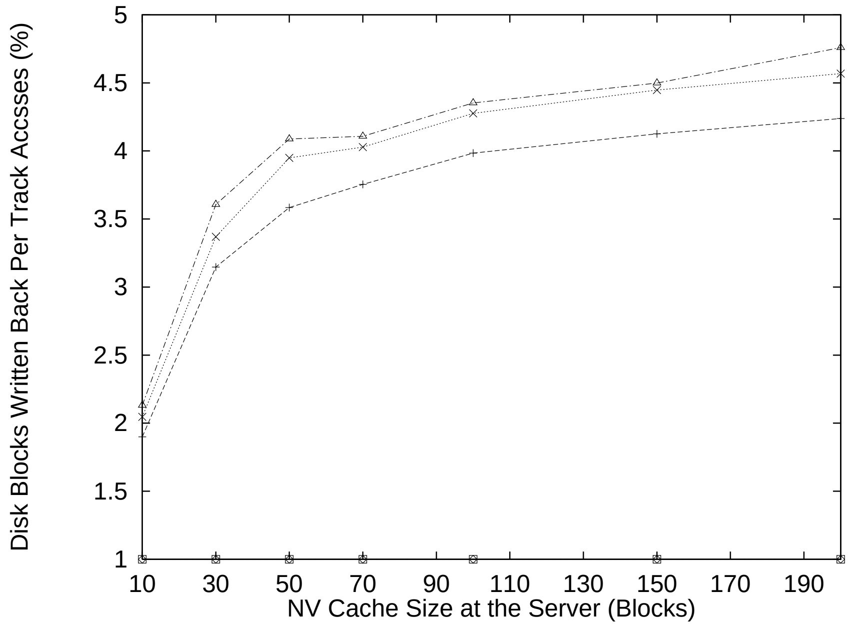



Figure B.3: Average Number of Dirty Blocks Written Back to Disk in One Disk Track Access

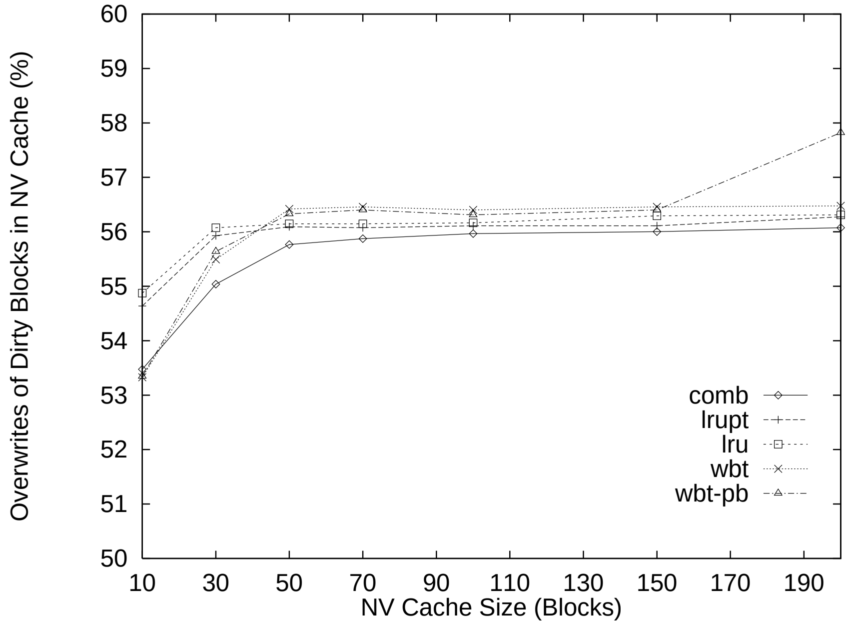



Figure B.4: Overwrites of Dirty Blocks as a Percentage of the Total Write Operations (Single-Level NV Cache Configuration)

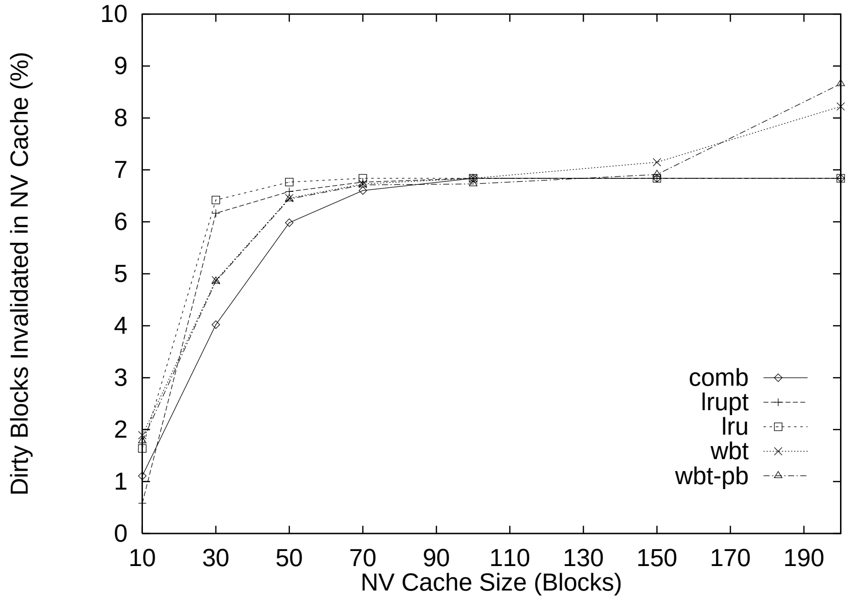



Figure B.5: Dirty Blocks Deleted from NV Cache as a Percentage of the Total Write Operations (Single-Level NV Caching Configuration)

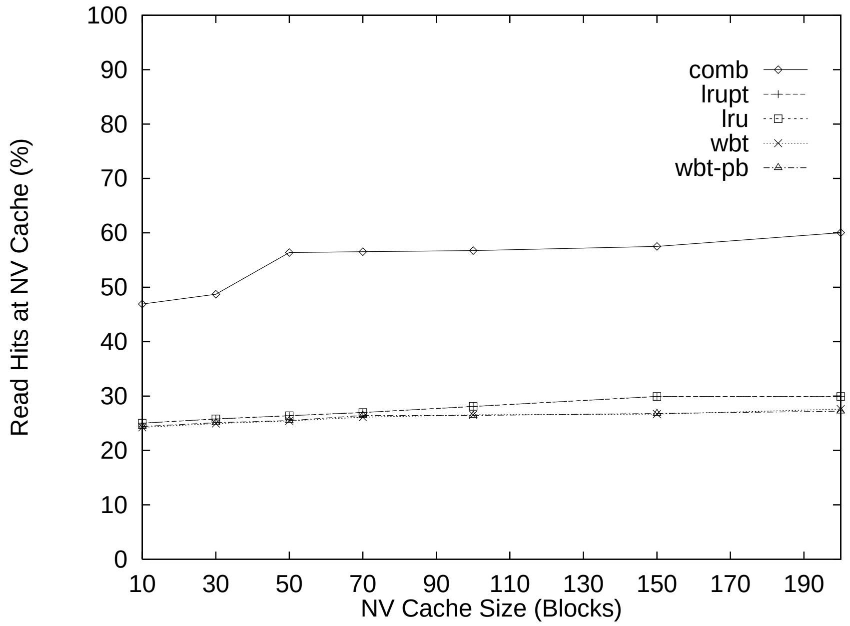



Figure B.6: Read Hit Ratio on NV Cache (Single-Level NV Cache Configuration)

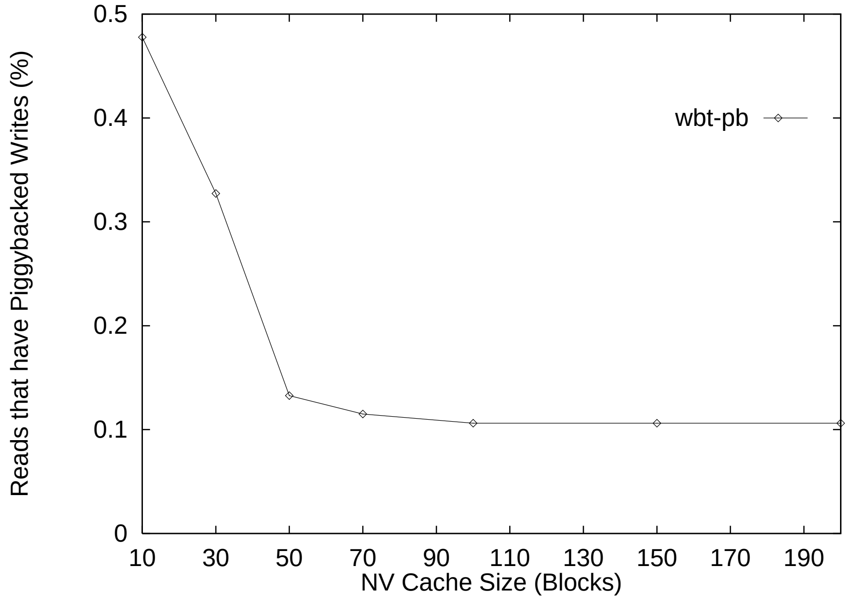



Figure B.7: Read Misses that have Writes Piggybacked (Single-Level NV Cache Configuration)

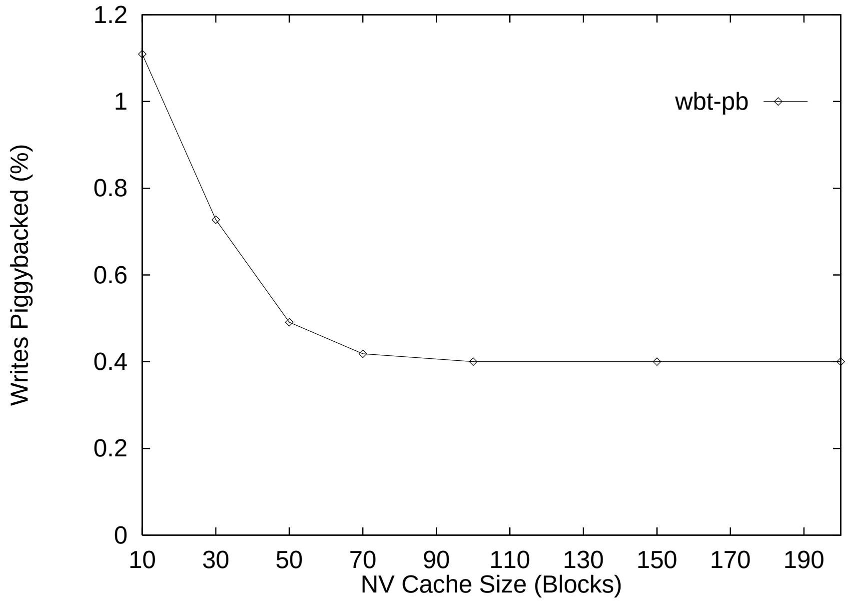
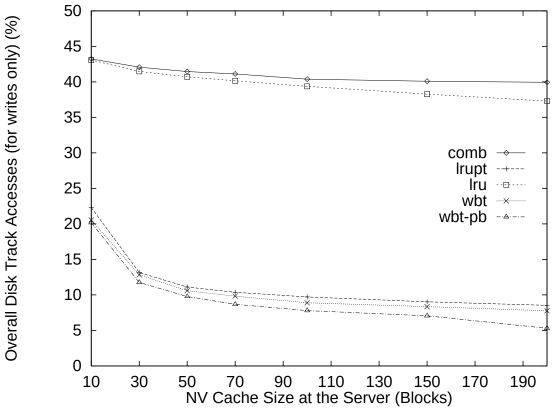
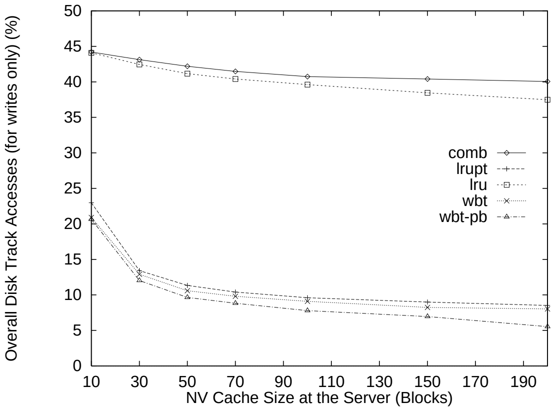


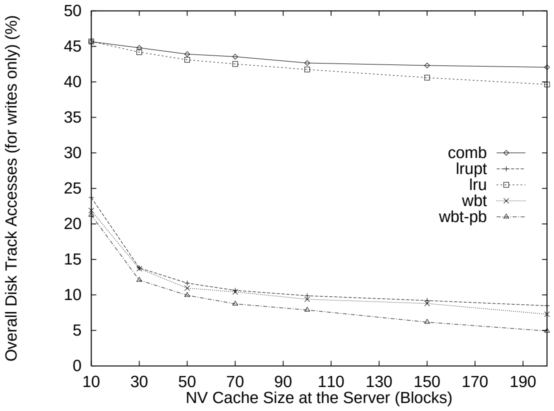

Figure B.8: Writes Piggybacked (Single-Level NV Cache Configuration)



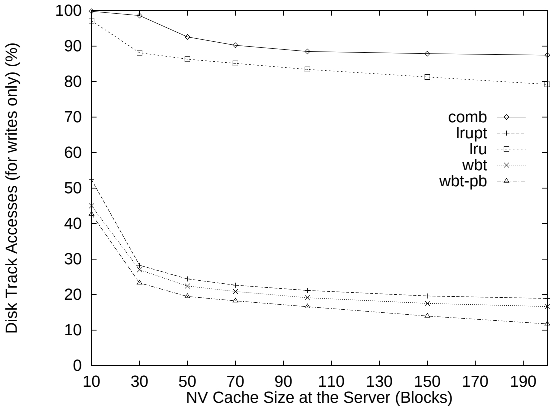

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

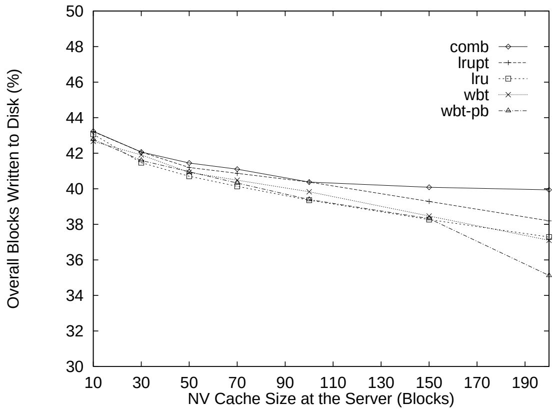



(c) Policy on Client: WBT

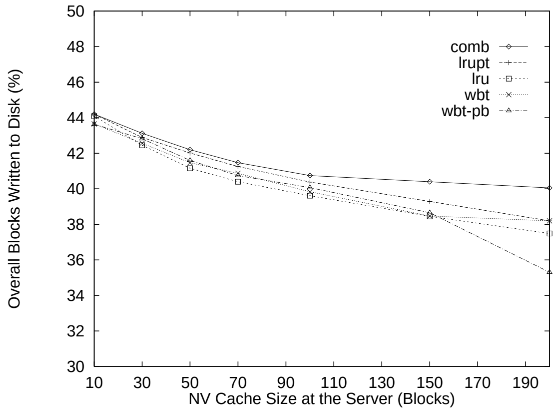



(d) Policy on Client: COMB

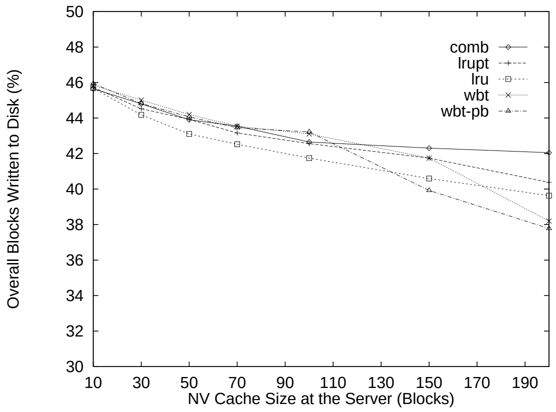
Figure B.9: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations. Client NV Cache Size is 10 Blocks



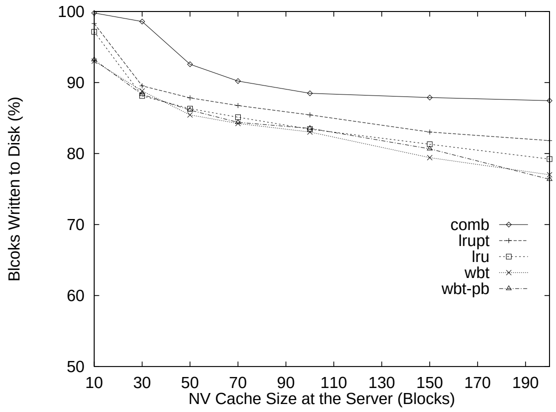

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

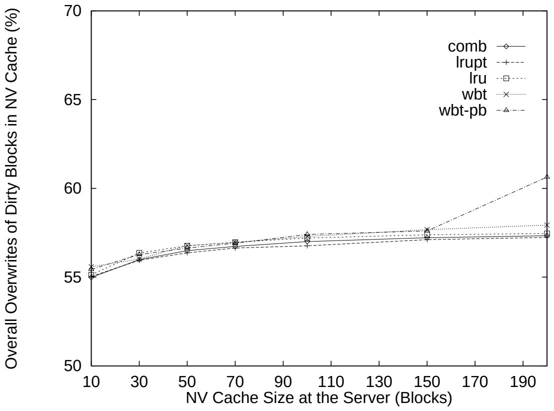



(c) Policy on Client: WBT

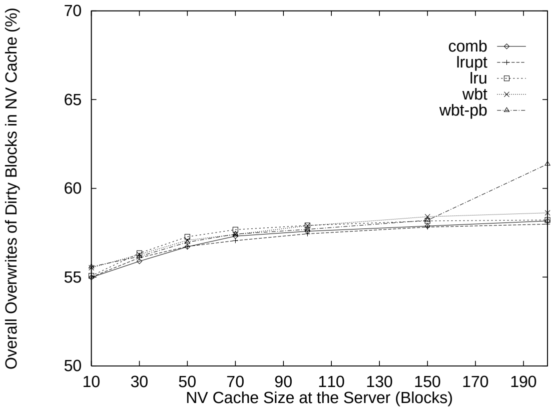



(d) Policy on Client: COMB

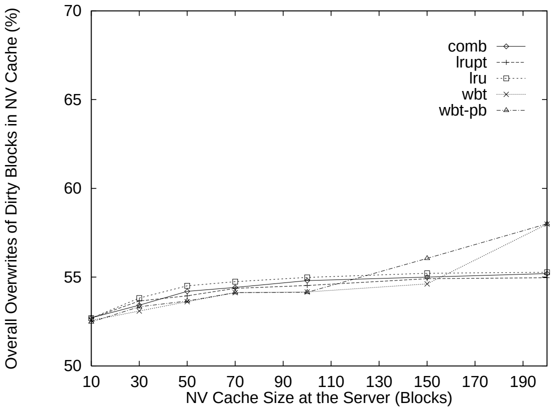
Figure B.10: Blocks Written to Disk, as a Percentage of Total Write Operations. Client NV Cache Size is 10 Blocks



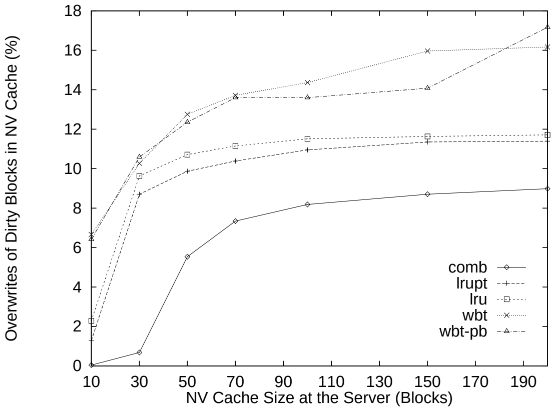

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

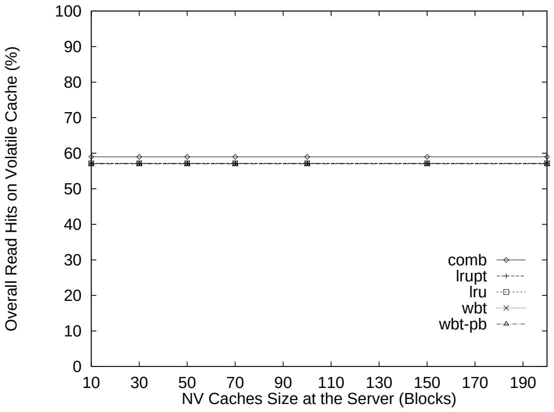



(c) Policy on Client: WBT

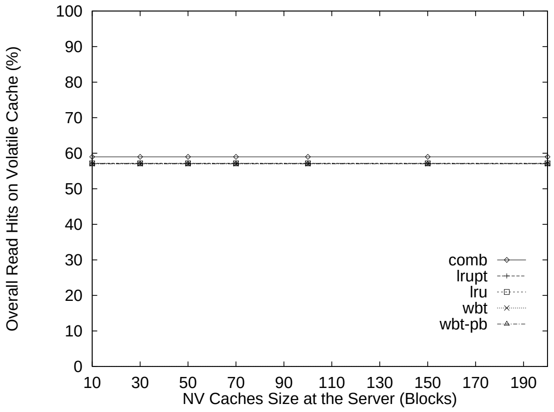



(d) Policy on Client: COMB

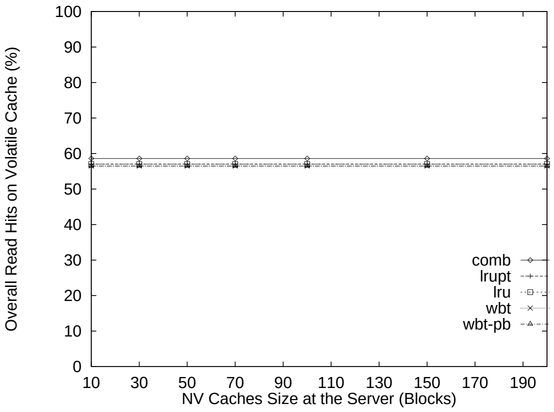
Figure B.11: Overwrites of Dirty Blocks in NV Caches, as a Percentage of the Total Write Operations. Client NV Cache Size is 10 Blocks



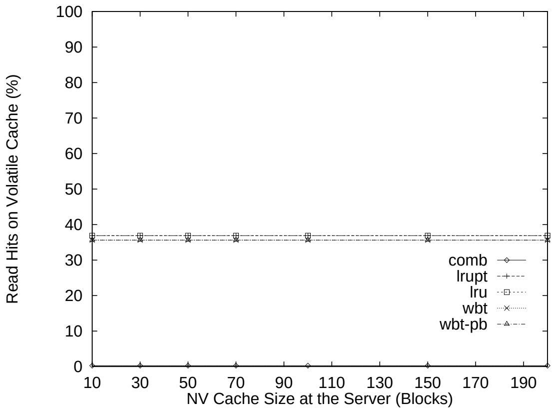

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

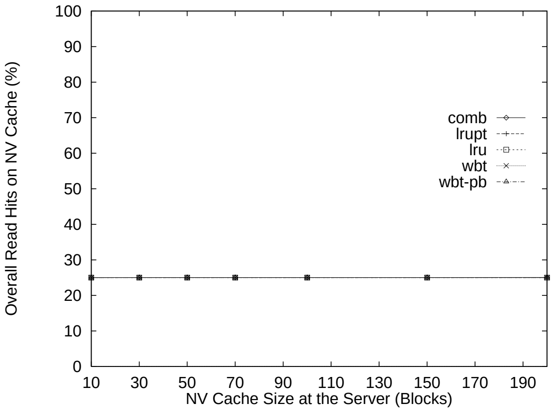



(c) Policy on Client: WBT

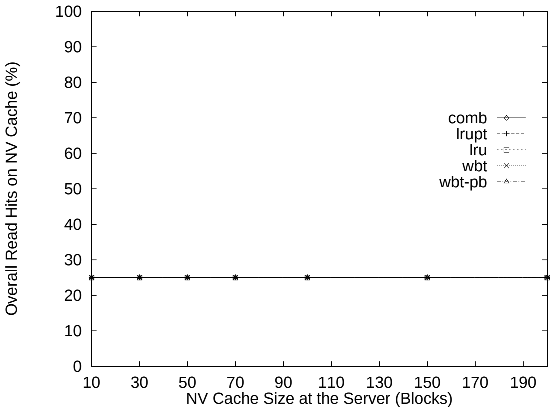



(d) Policy on Client: COMB

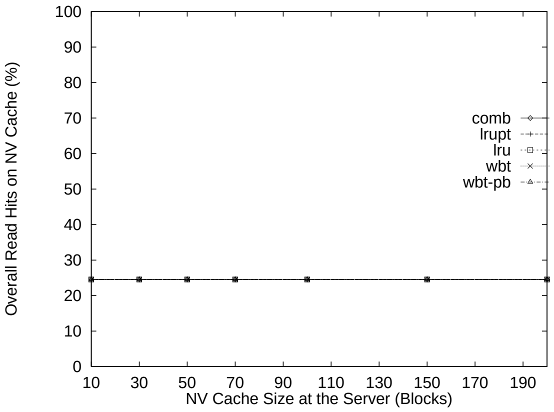
Figure B.12: Overall Read Hit Ratio on Volatile Caches. Client NV Cache Size is 10 Blocks



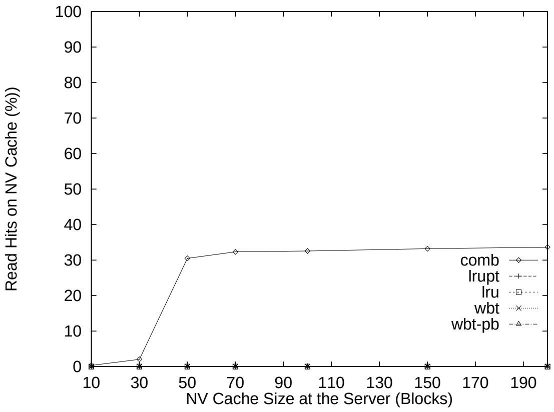

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

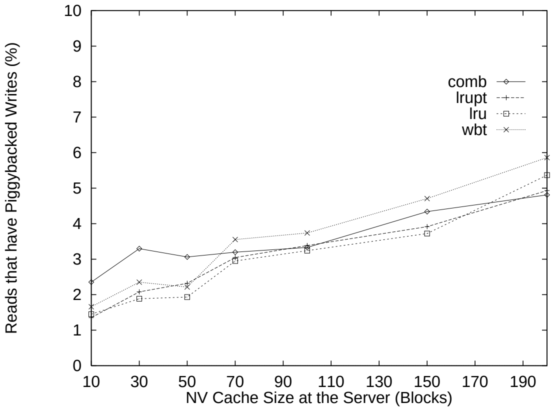



(c) Policy on Client: WBT

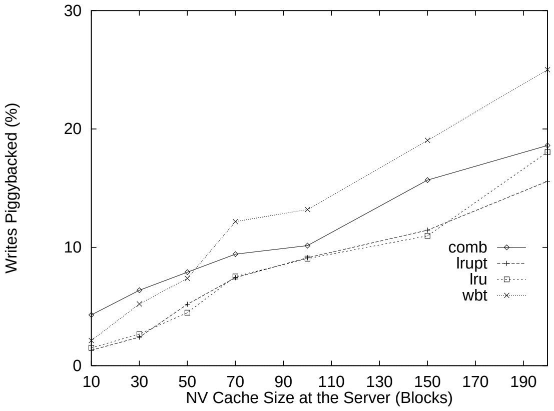



(d) Policy on Client: COMB

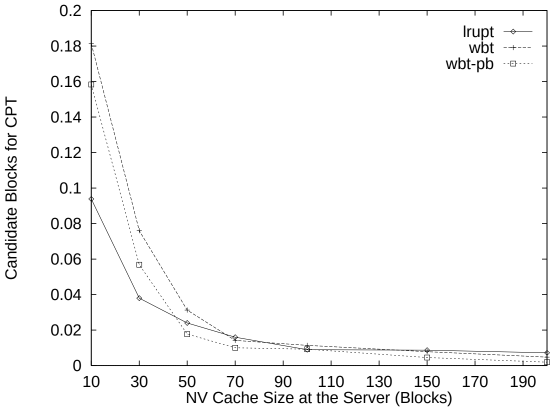
Figure B.13: Overall Read Hit Ratio on NV Caches. Client NV Cache Size is 10 Blocks



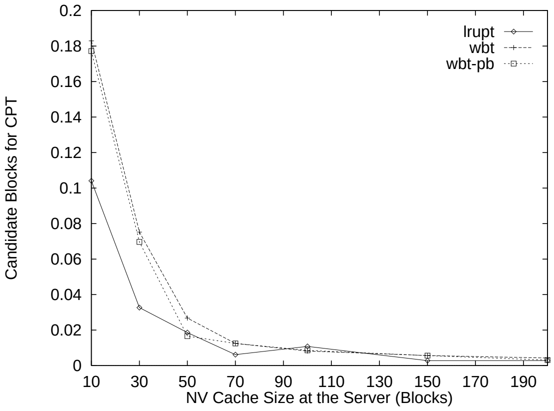

(a) Reads Misses that have Writes Piggybacked




(b) Writes Piggybacked

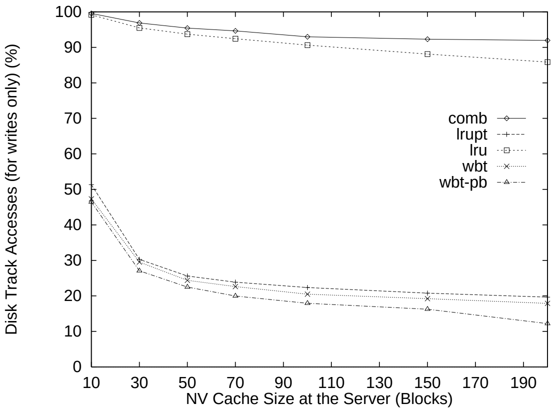



(c) Candidate Blocks for CPT, Client Policy is LRUPT

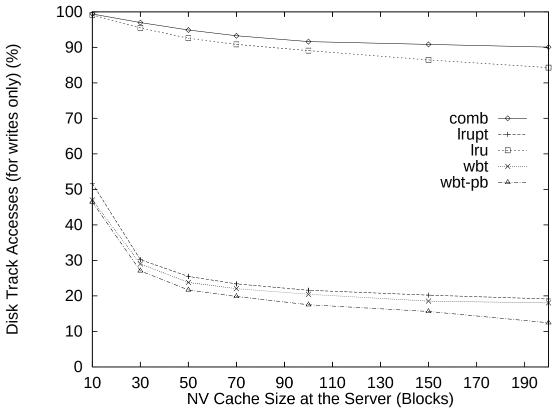



(d) Candidate for CPT, Client Policy is WBT

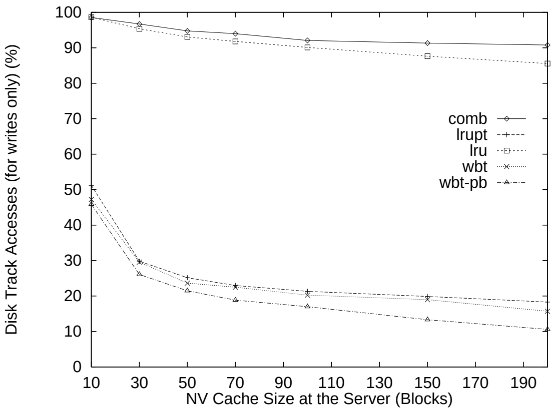
Figure B.14: Other Statistics. Client NV Cache Size is 10 Blocks



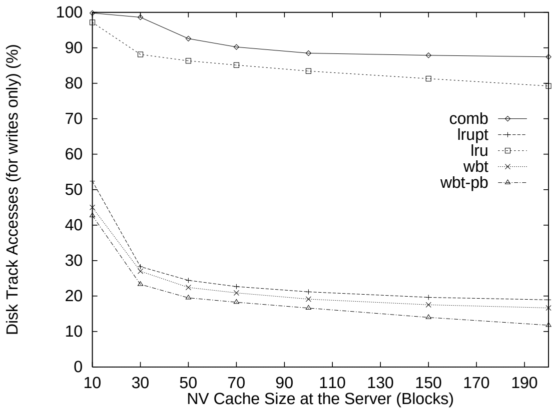

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

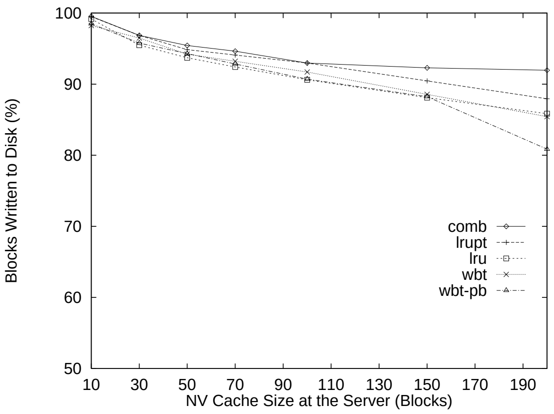



(c) Policy on Client: WBT

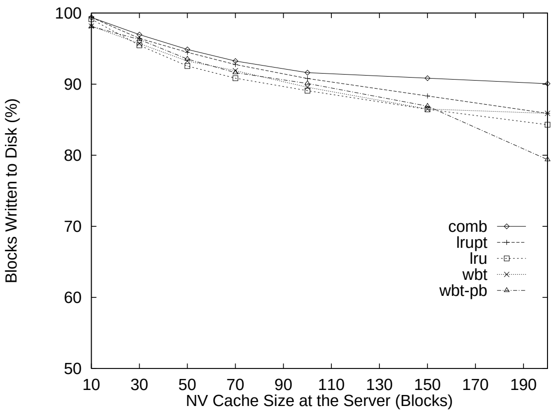



(d) Policy on Client: COMB

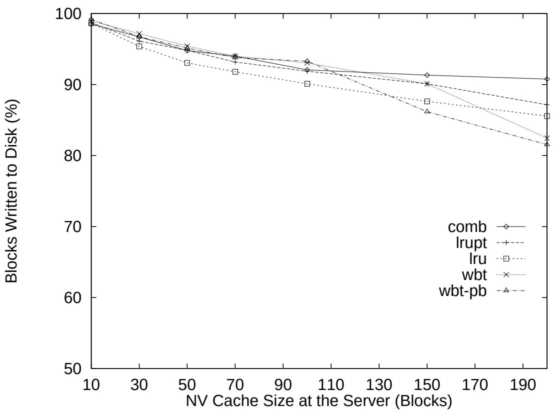
Figure B.15: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache Size is 10 Blocks



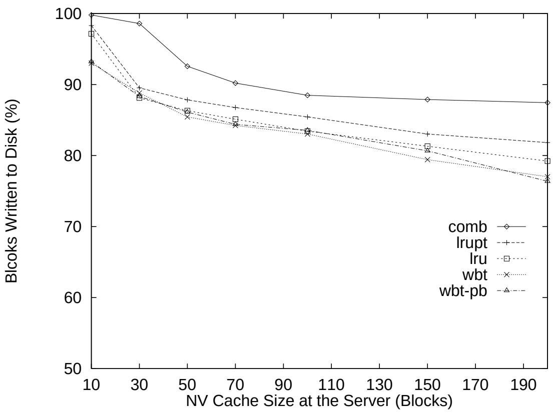

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

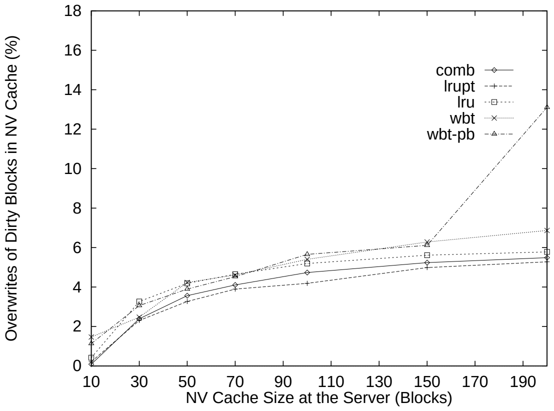



(c) Policy on Client: WBT

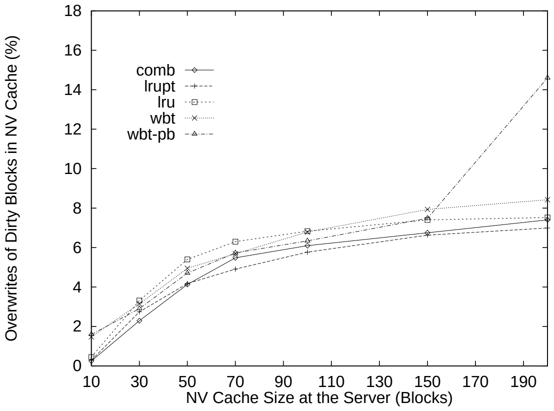



(d) Policy on Client: COMB

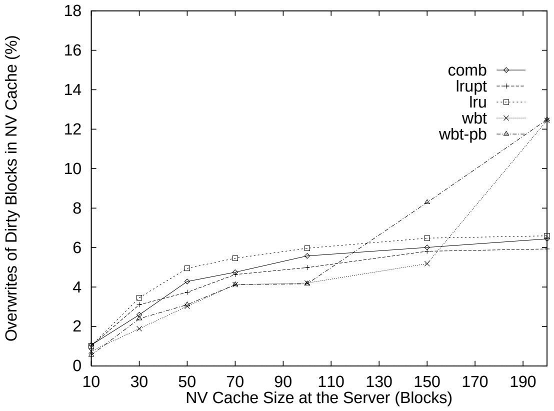
Figure B.16: Blocks Written to Disk, as a Percentage of Total Write Operations Seen at the Server. Client NV Cache Size is 10 Blocks



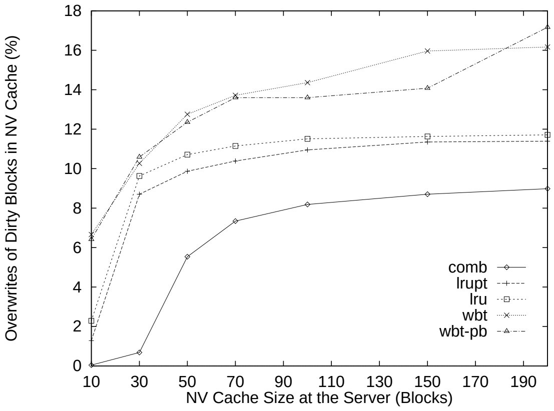

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

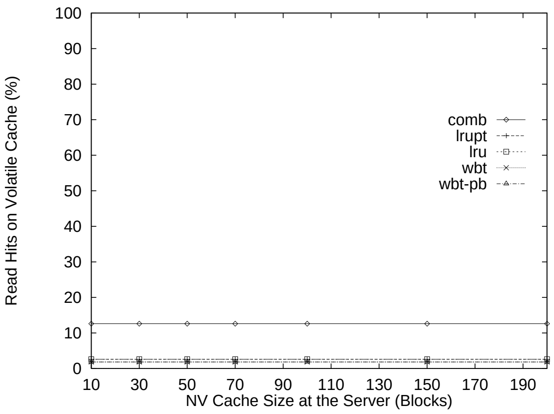



(c) Policy on Client: WBT

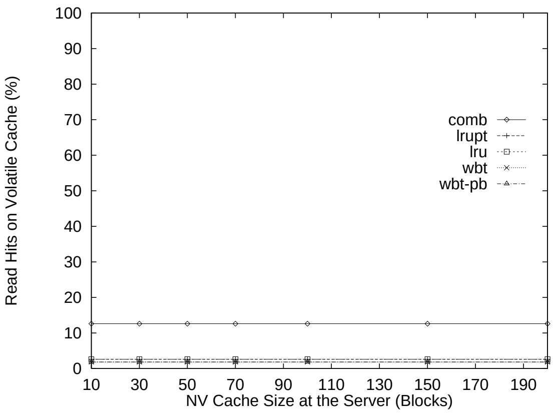



(d) Policy on Client: COMB

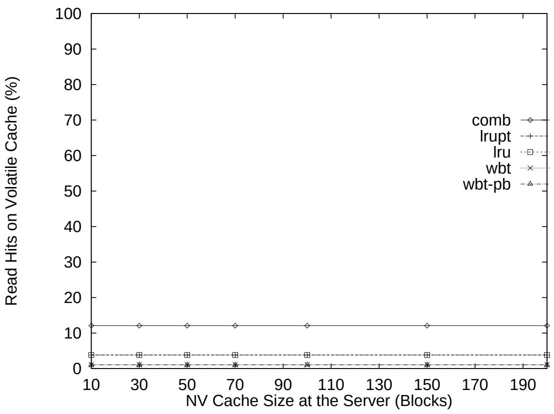
Figure B.17: Overwrites of Dirty Blocks in Server NV Cache, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache Size is 10 Blocks



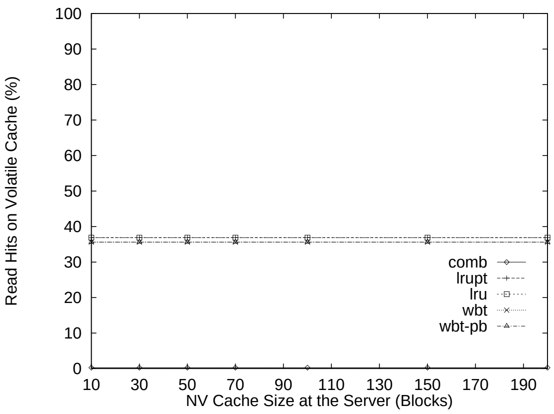

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

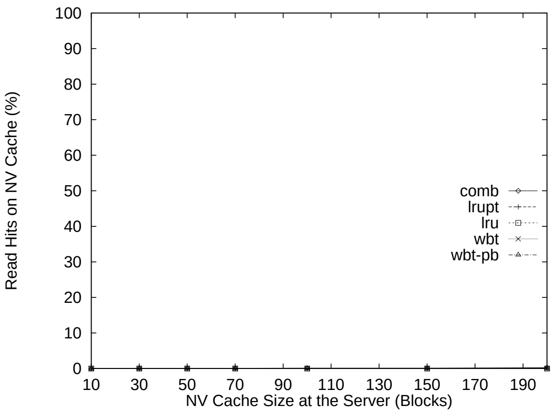



(c) Policy on Client: WBT

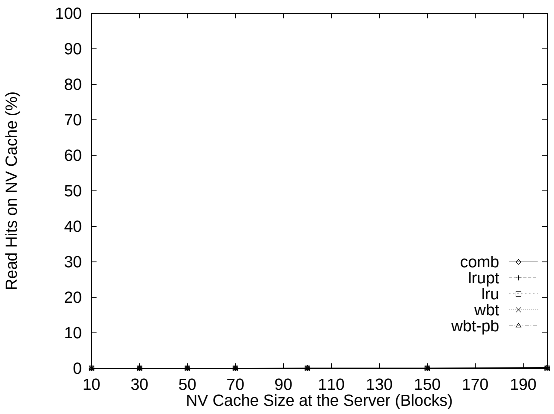



(d) Policy on Client: COMB

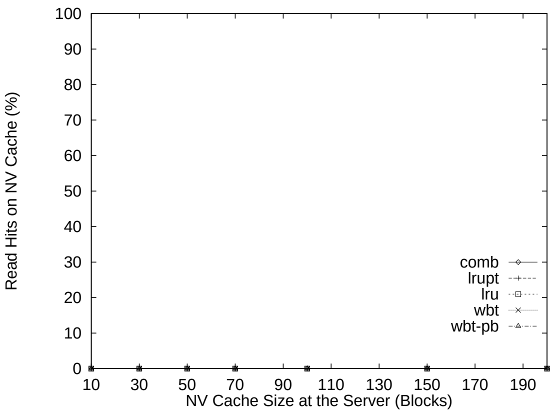
Figure B.18: Overall Read Hit Ratio on Server Volatile Cache. Client NV Cache Size is 10 Blocks



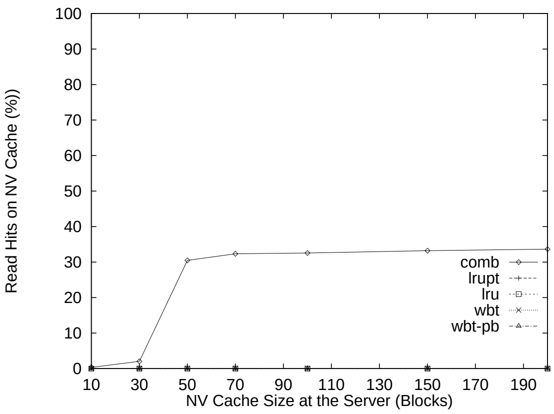

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

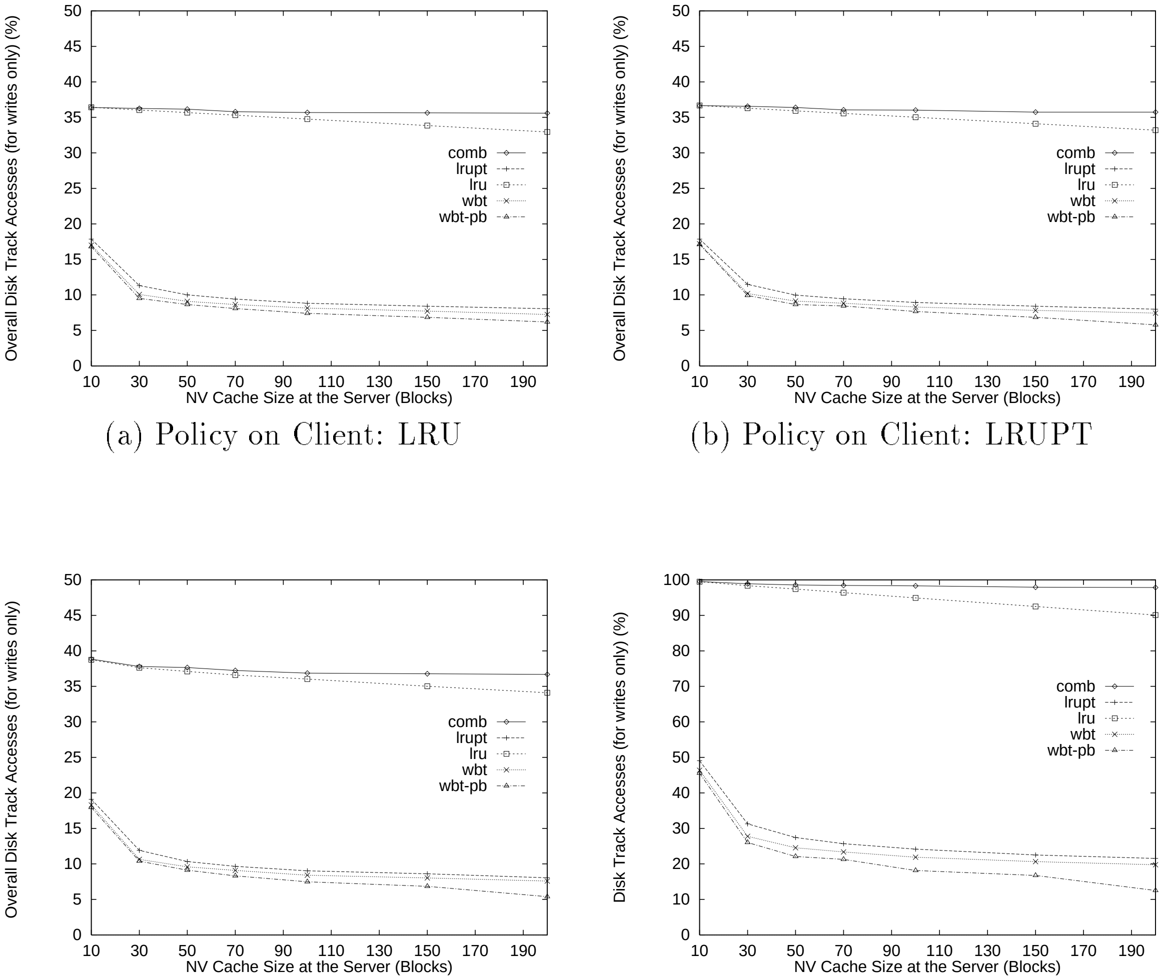



(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure B.19: Overall Read Hit Ratio on Server NV Cache. Client NV Cache Size is 10 Blocks



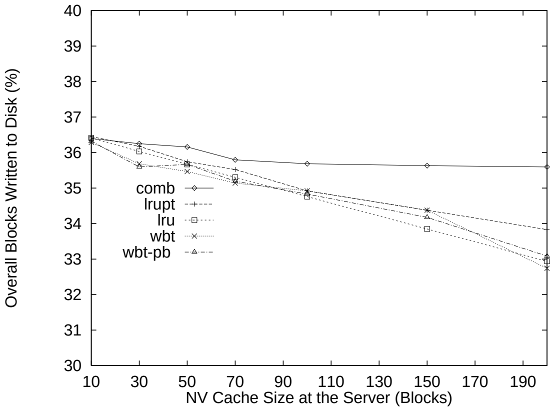

(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

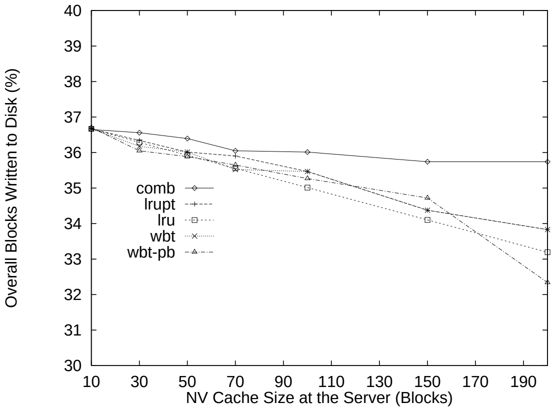
(c) Policy on Client: WBT

(d) Policy on Client: COMB

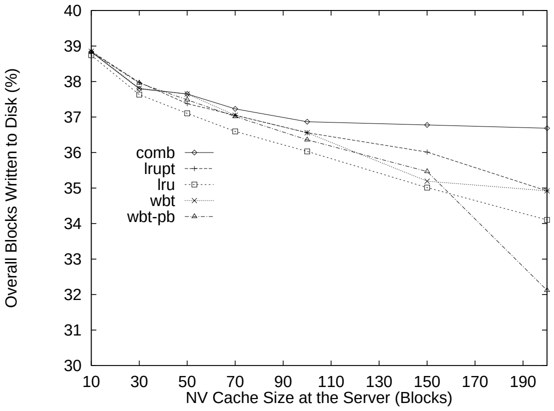
Figure B.20: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations. Client NV Cache Size is 50 Blocks



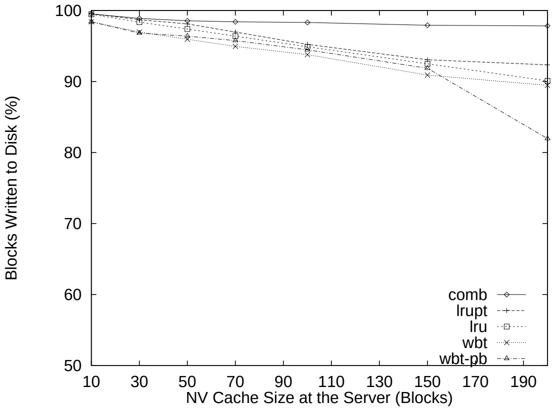

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

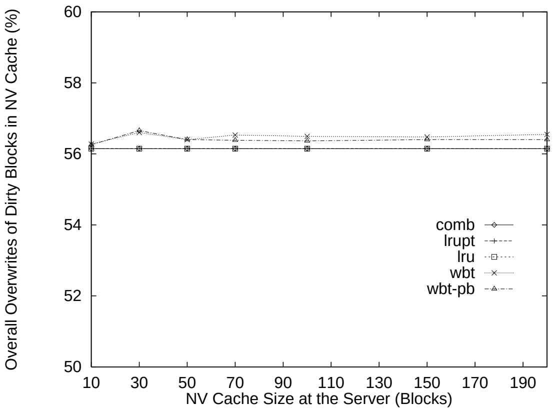



(c) Policy on Client: WBT

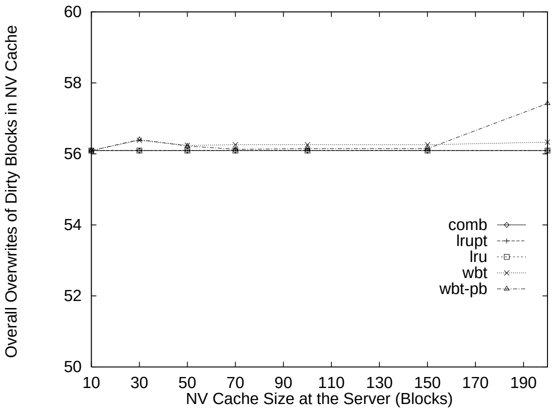



(d) Policy on Client: COMB

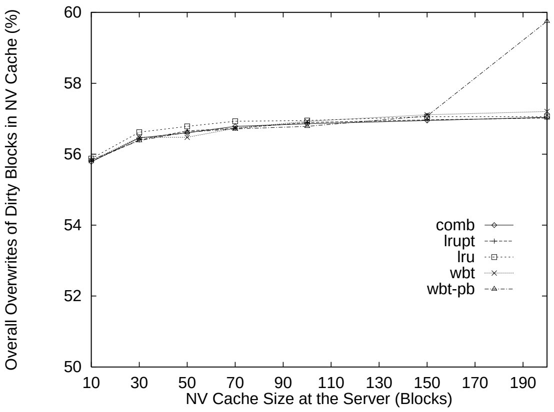
Figure B.21: Blocks Written to Disk, as a Percentage of Total Write Operations. Client NV Cache Size is 50 Blocks



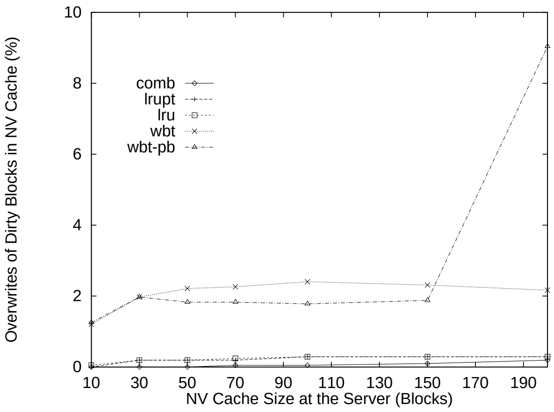

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

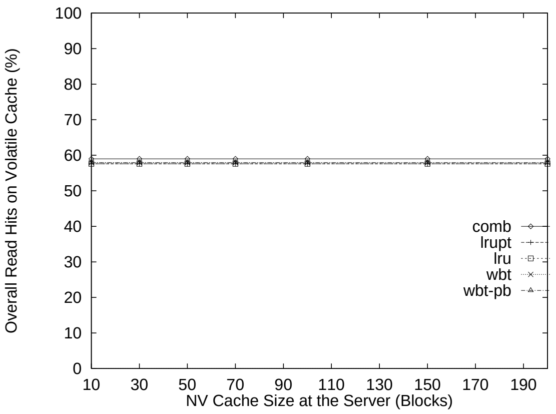



(c) Policy on Client: WBT

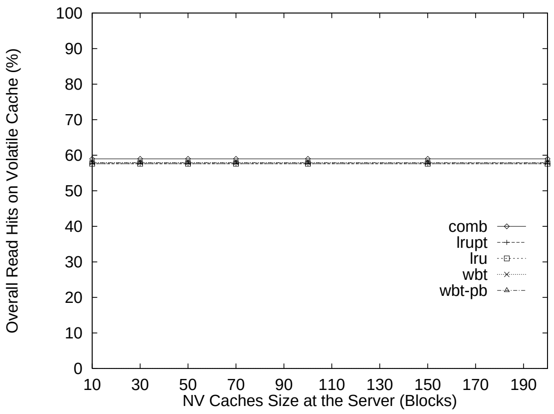



(d) Policy on Client: COMB

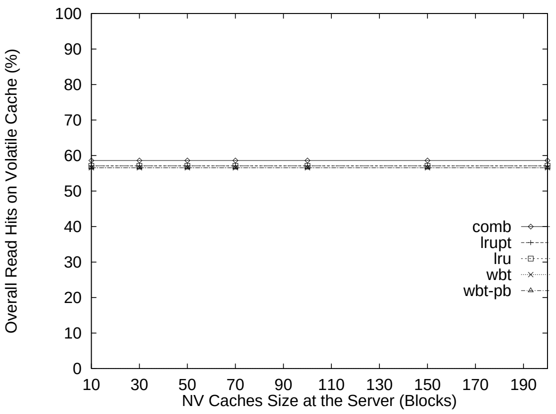
Figure B.22: Overwrites of Dirty Blocks in NV Caches, as a Percentage of the Total Write Operations. Client NV Cache Size is 50 Blocks



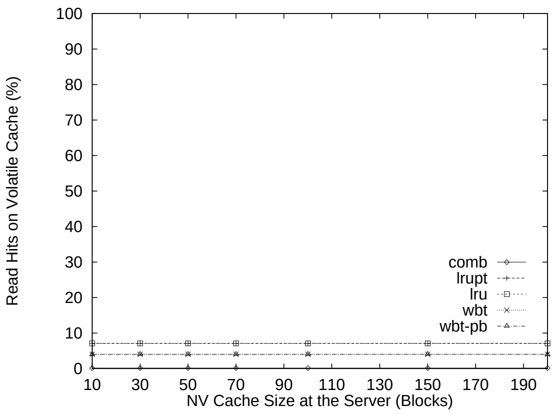

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

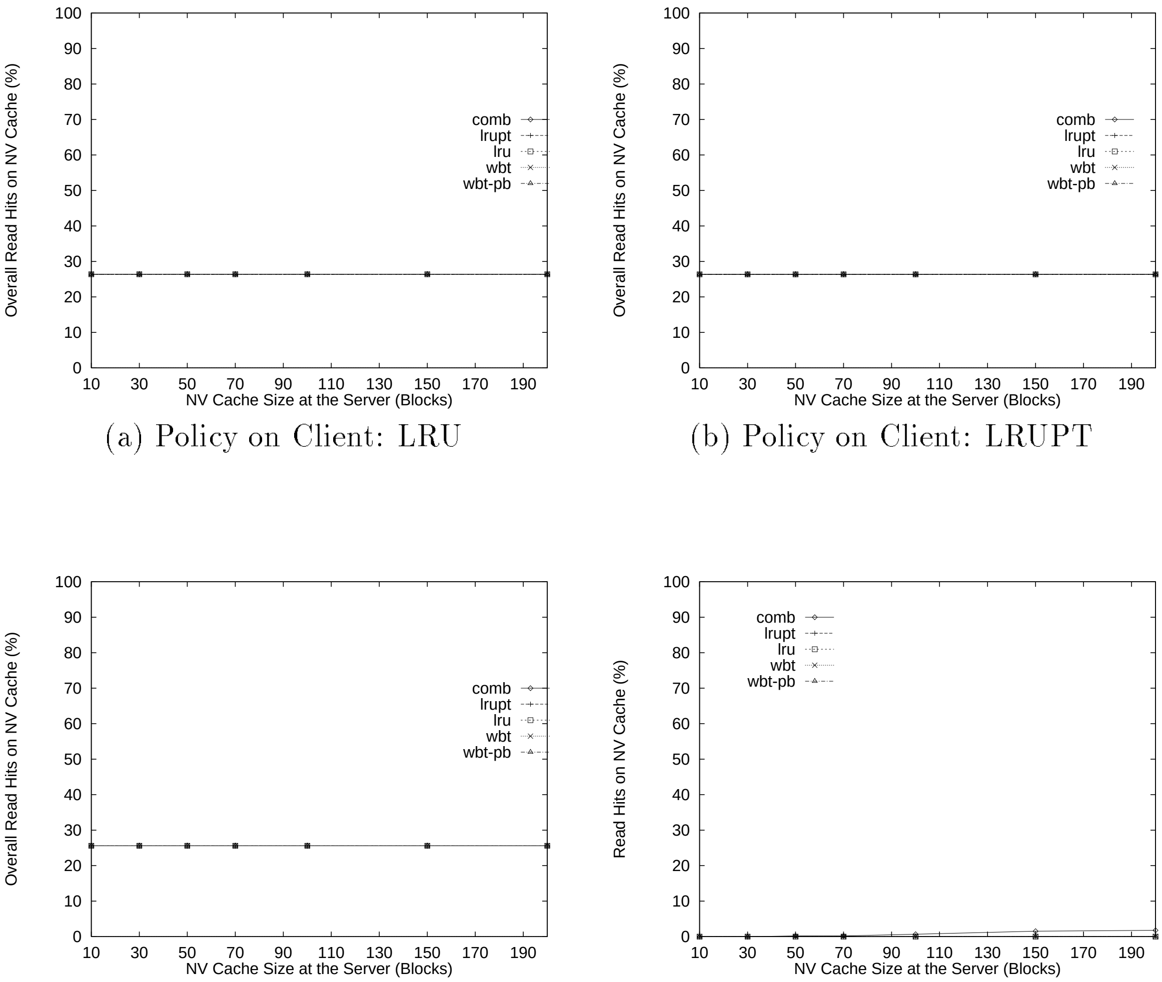



(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure B.23: Overall Read Hit Ratio on Volatile Caches. Client NV Cache Size is 50 Blocks



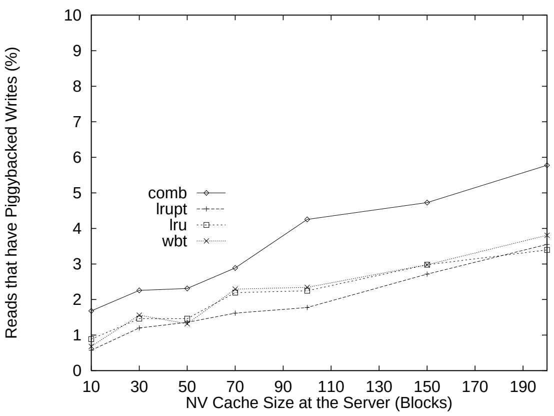

(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

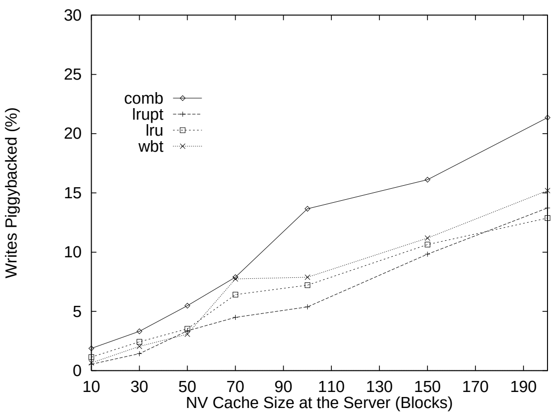
(c) Policy on Client: WBT

(d) Policy on Client: COMB

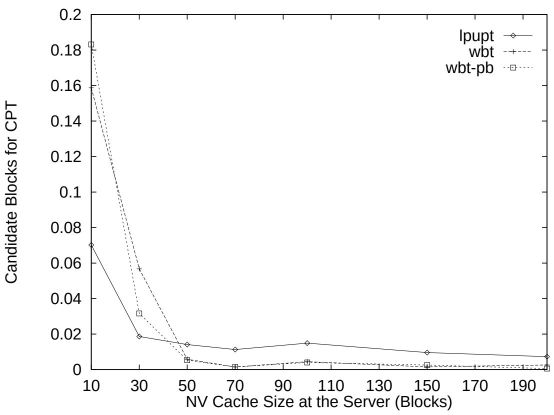
Figure B.24: Overall Read Hit Ratio on NV Caches. Client NV Cache Size is 50 Blocks



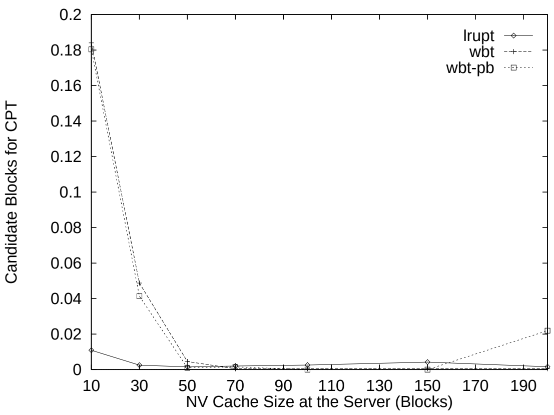

(a) Reads Misses that have Writes Piggybacked




(b) Writes Piggybacked

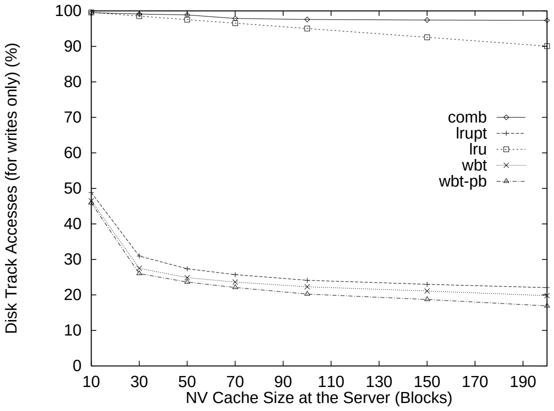



(c) Candidate Blocks for CPT, Client Policy is LRUPT

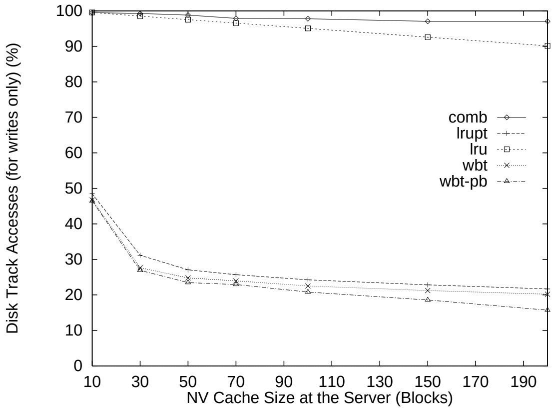



(d) Candidate for CPT, Client Policy is WBT

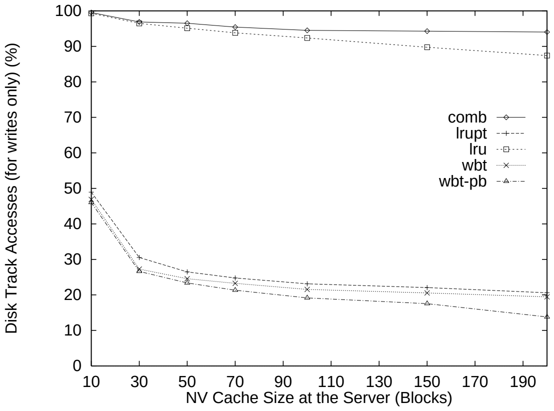
Figure B.25: Other Statistics. Client NV Cache Size is 50 Blocks



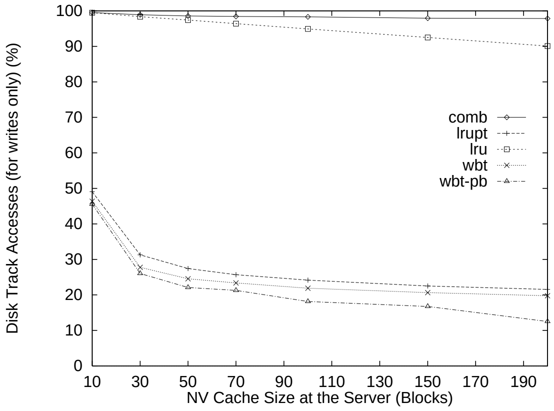

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

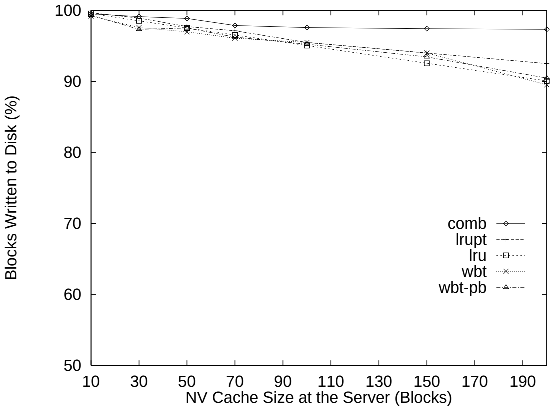



(c) Policy on Client: WBT

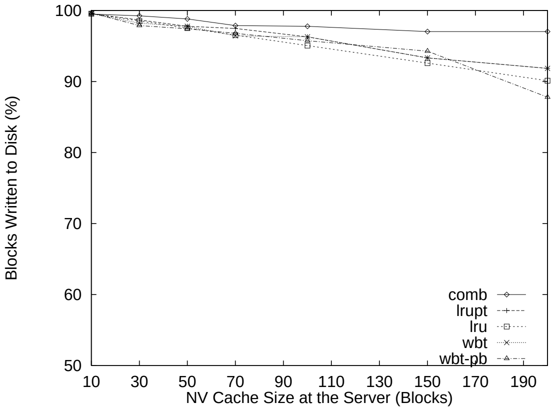



(d) Policy on Client: COMB

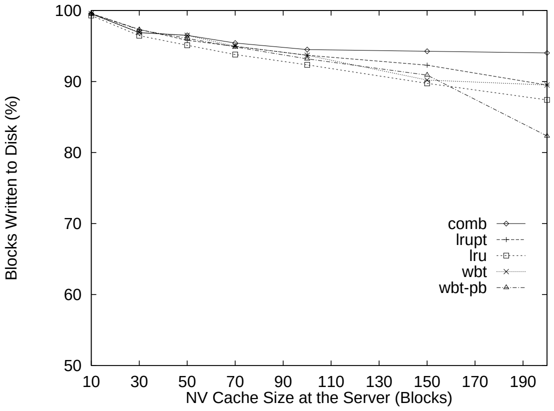
Figure B.26: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache Size is 50 Blocks



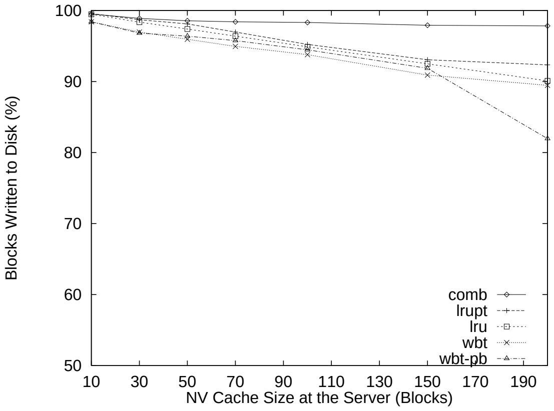

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

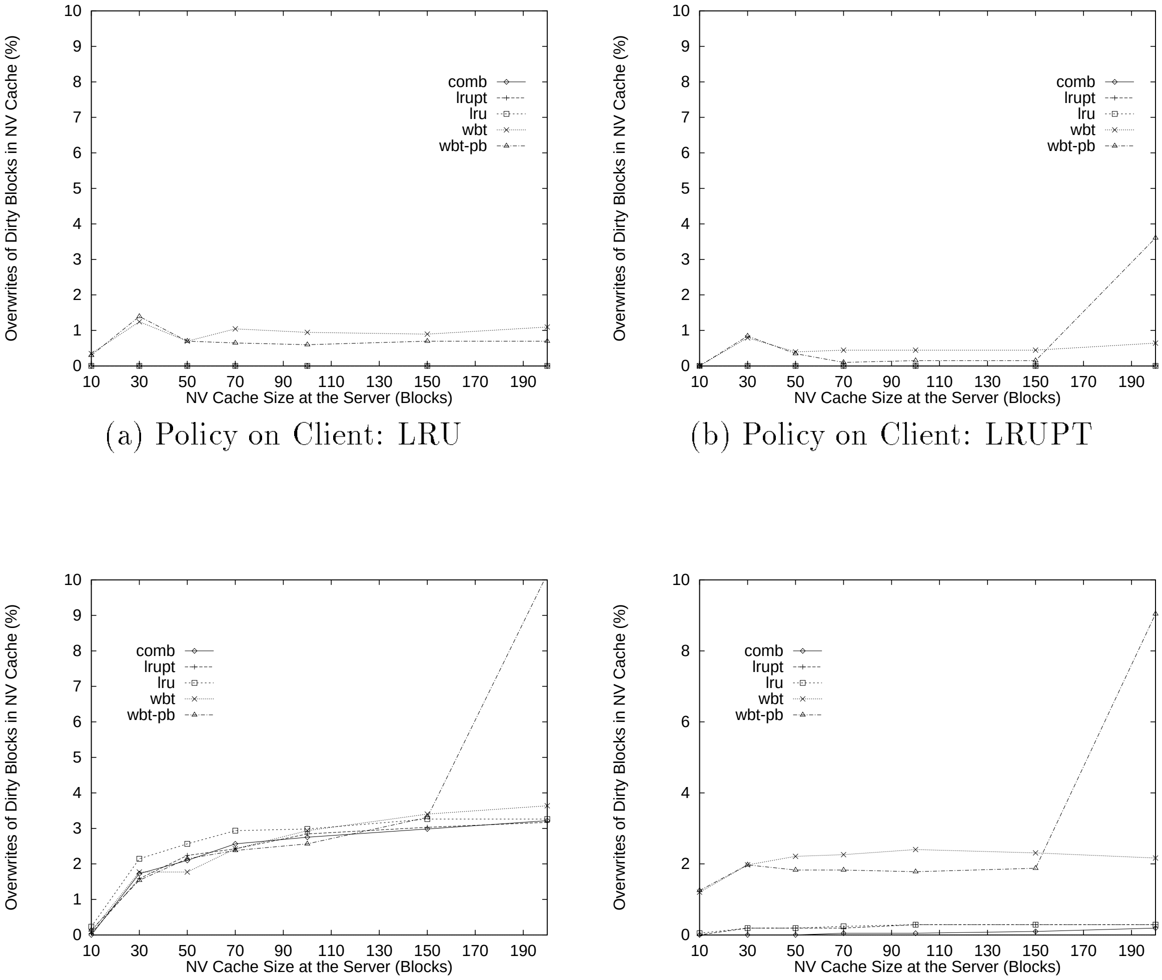



(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure B.27: Blocks Written to Disk, as a Percentage of Total Write Operations Seen at the Server. Client NV Cache Size is 50 Blocks



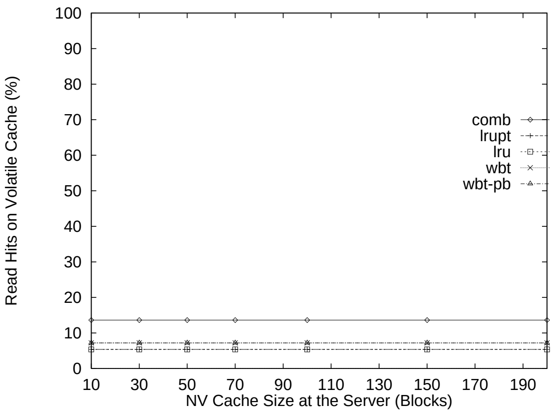

(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

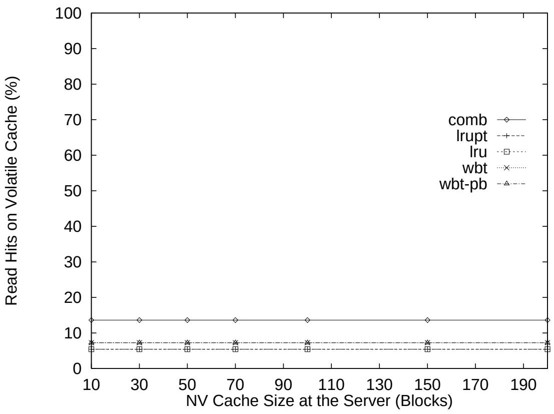
(c) Policy on Client: WBT

(d) Policy on Client: COMB

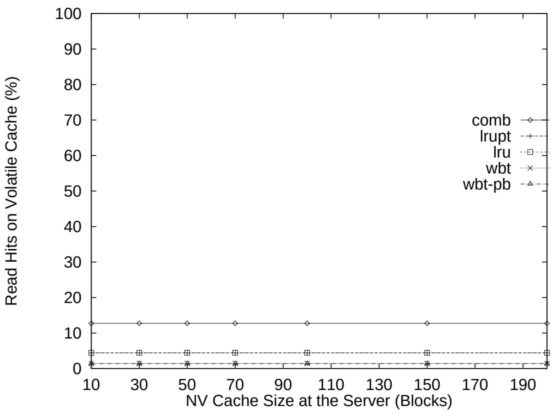
Figure B.28: Overwrites of Dirty Blocks in Server NV Cache, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache Size is 50 Blocks



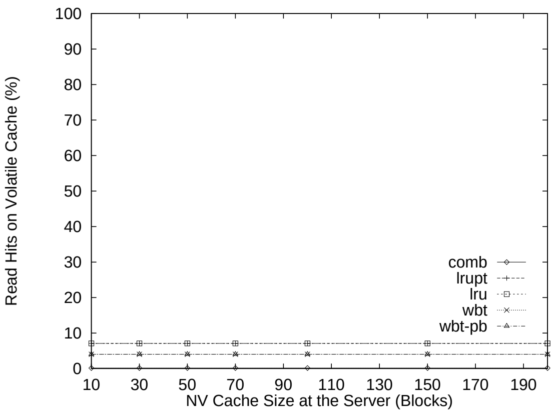

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

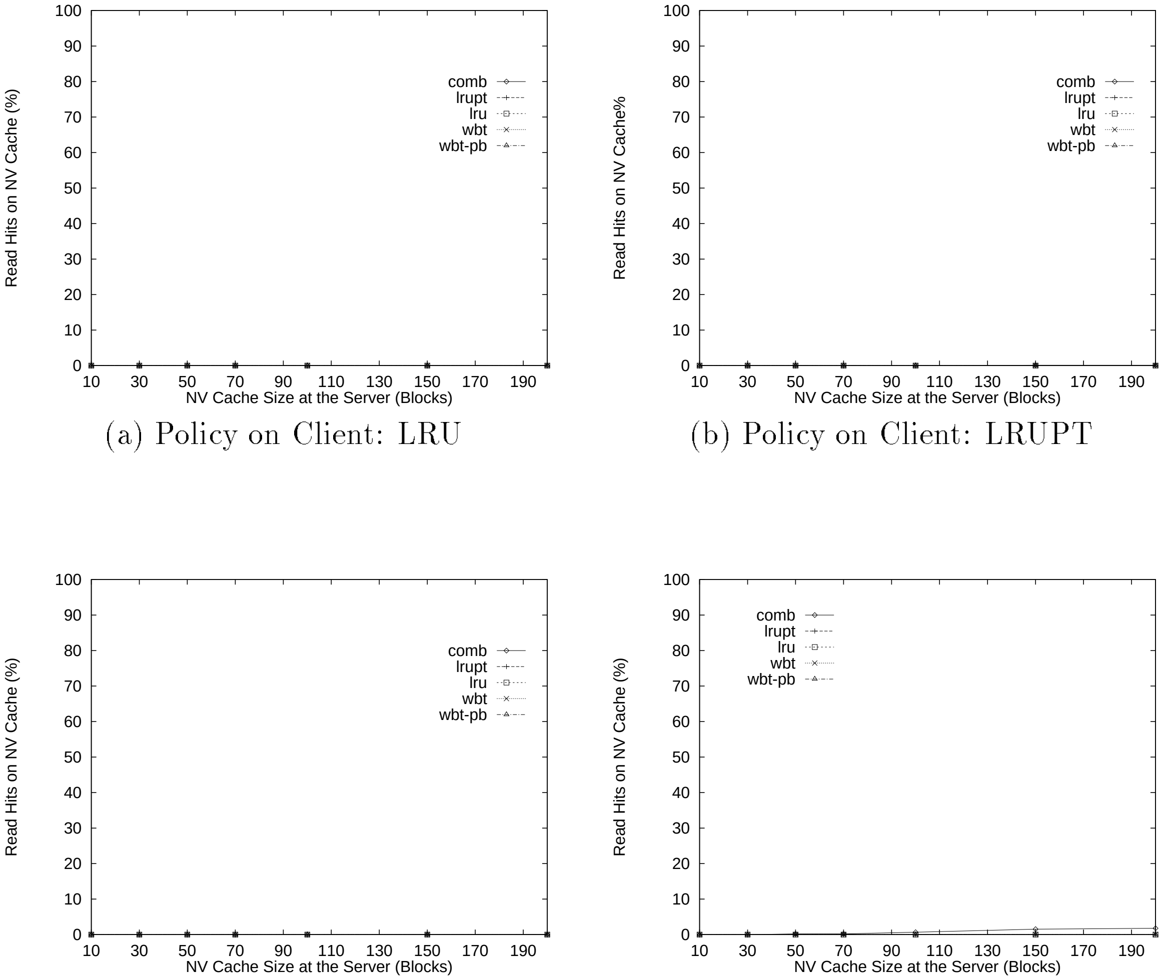



(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure B.29: Overall Read Hit Ratio on Server Volatile Cache. Client NV Cache Size is 50 Blocks



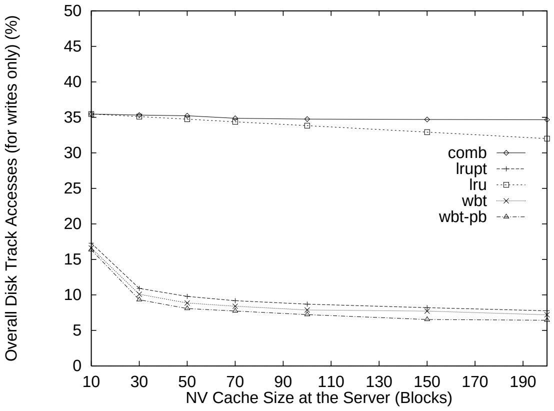

(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

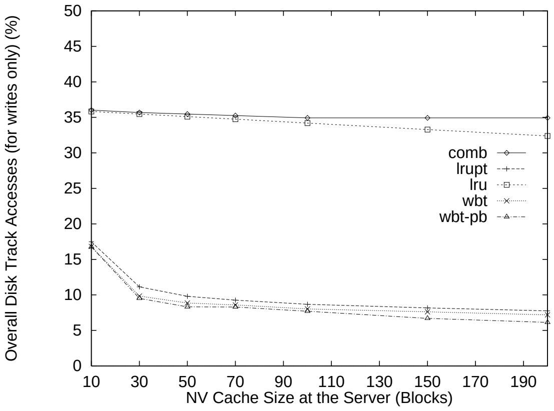
(c) Policy on Client: WBT

(d) Policy on Client: COMB

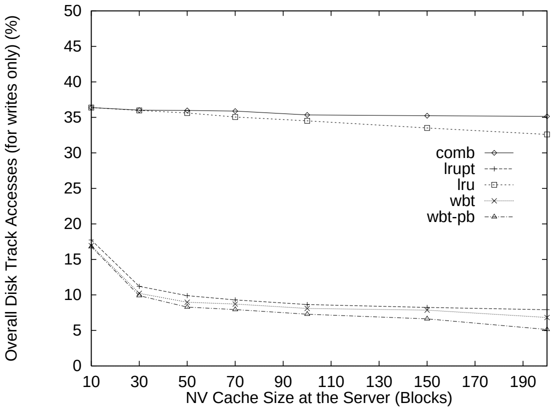
Figure B.30: Overall Read Hit Ratio on Server NV Cache. Client NV Cache Size is 50 Blocks



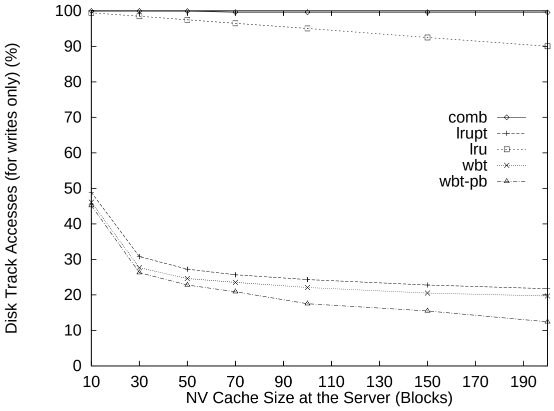

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

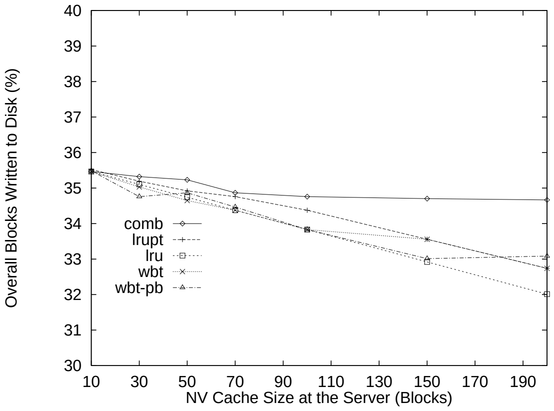



(c) Policy on Client: WBT

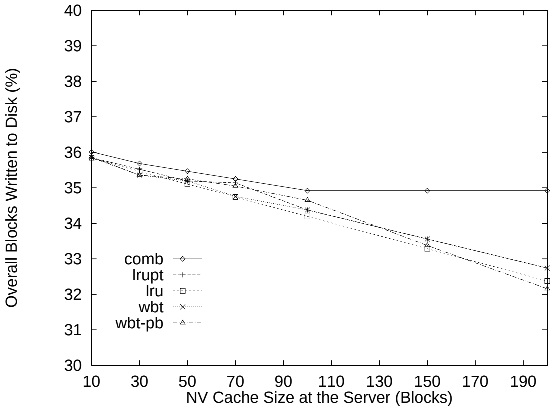



(d) Policy on Client: COMB

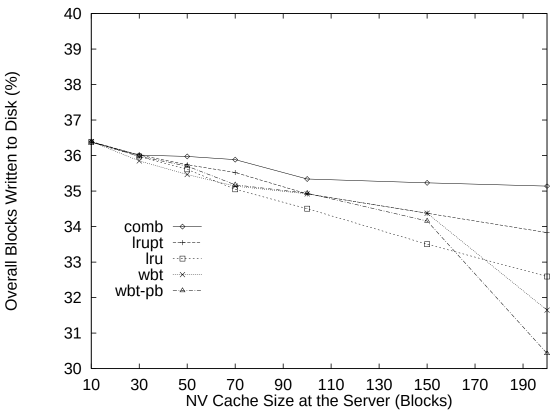
Figure B.31: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations. Client NV Cache Size is 100 Blocks



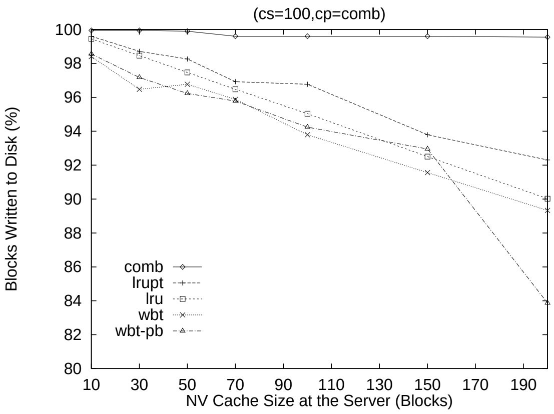

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

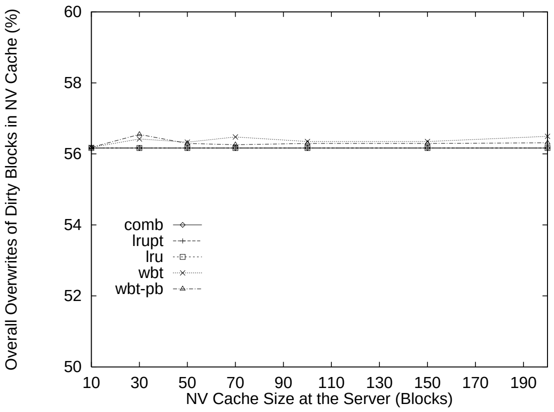



(c) Policy on Client: WBT

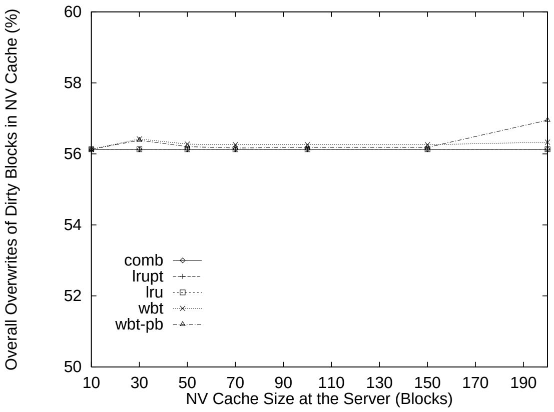



(d) Policy on Client: COMB

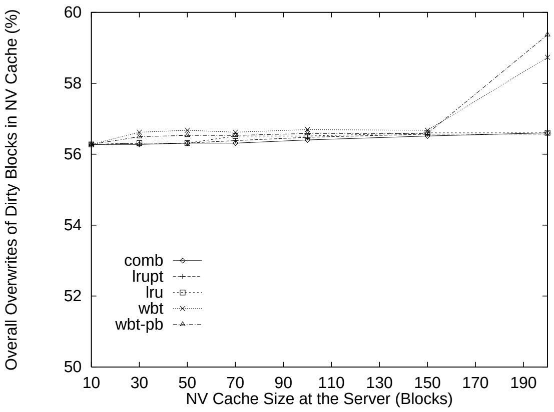
Figure B.32: Blocks Written to Disk, as a Percentage of Total Write Operations. Client NV Cache Size is 100 Blocks



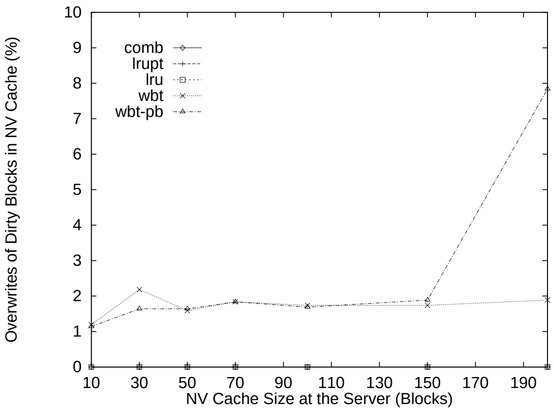

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

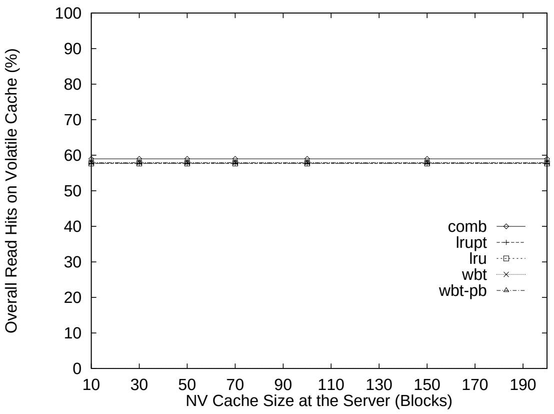



(c) Policy on Client: WBT

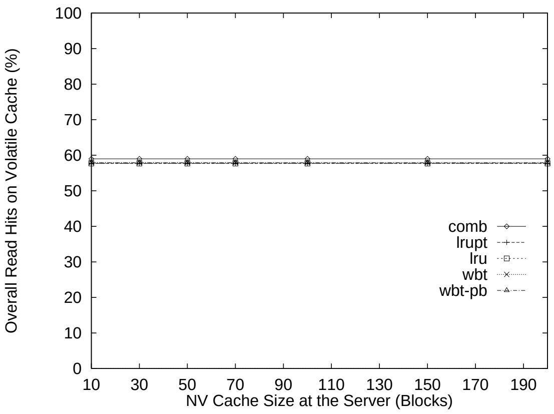



(d) Policy on Client: COMB

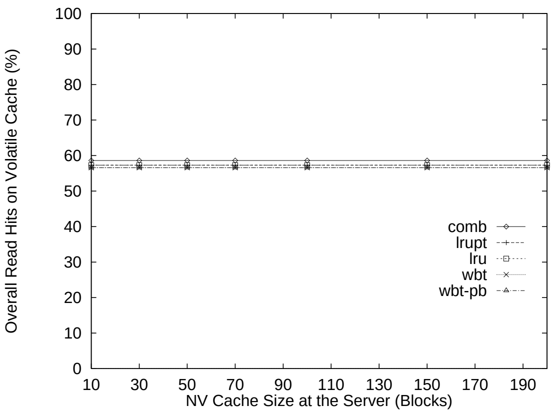
Figure B.33: Overwrites of Dirty Blocks in NV Caches, as a Percentage of the Total Write Operations. Client NV Cache Size is 100 Blocks



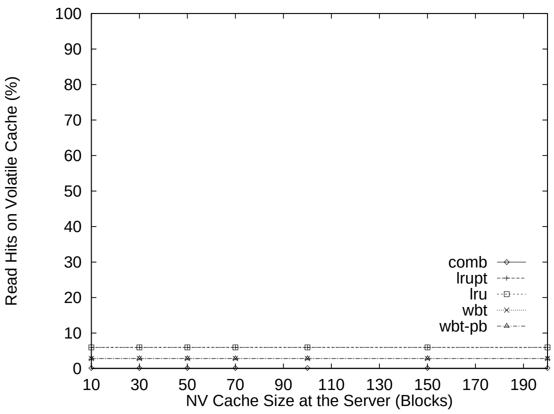

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

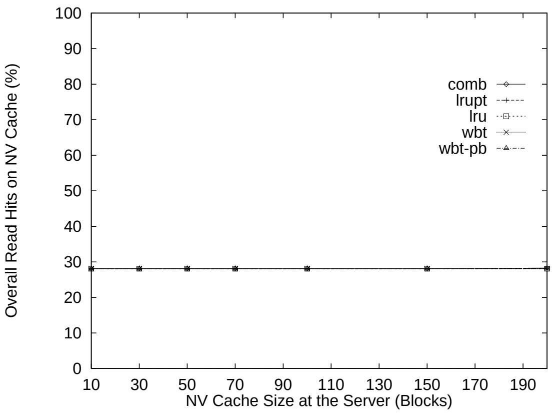



(c) Policy on Client: WBT

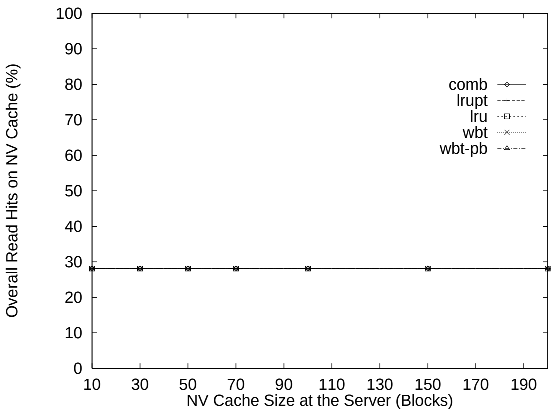



(d) Policy on Client: COMB

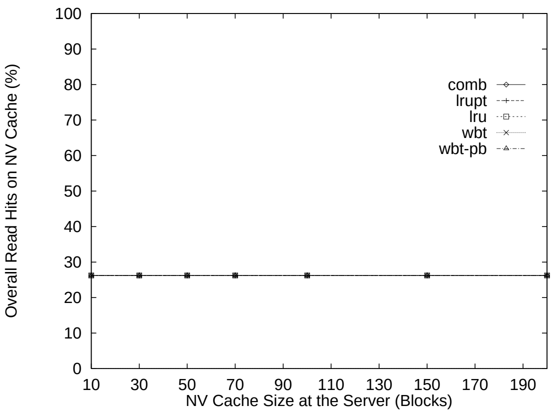
Figure B.34: Overall Read Hit Ratio on Volatile Caches. Client NV Cache Size is 100 Blocks



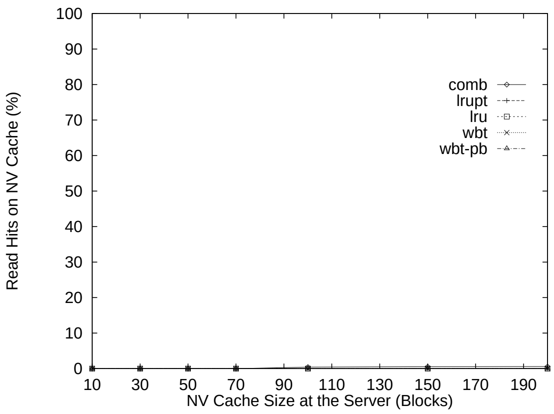

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

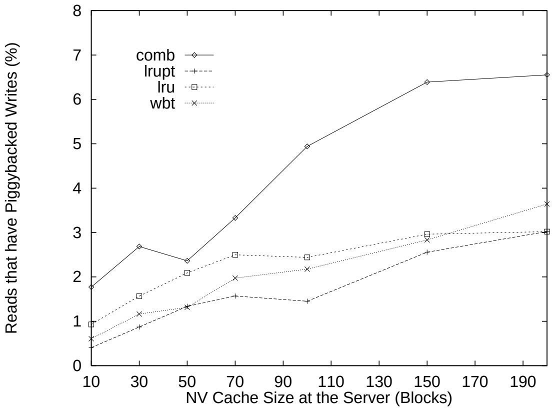



(c) Policy on Client: WBT

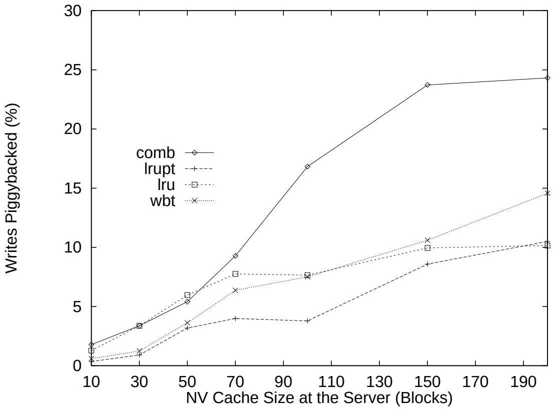



(d) Policy on Client: COMB

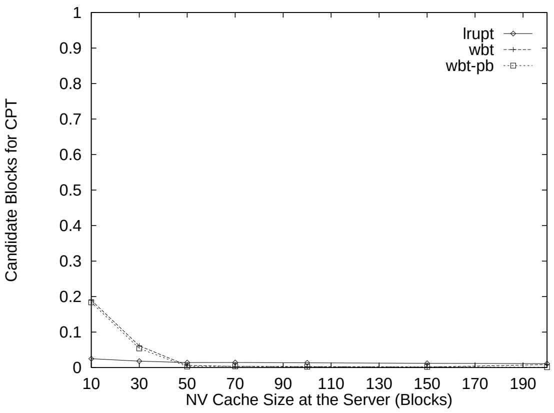
Figure B.35: Overall Read Hit Ratio on NV Caches. Client NV Cache Size is 100 Blocks



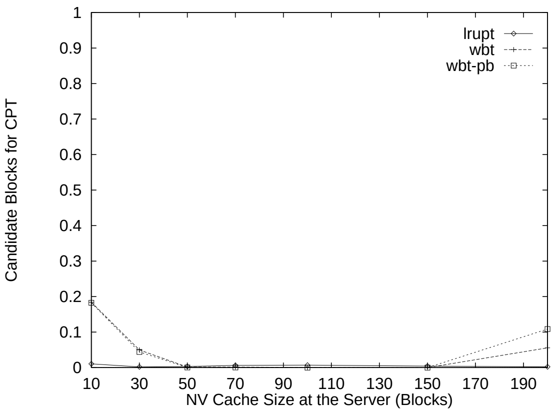

(a) Reads Misses that have Writes Piggybacked




(b) Writes Piggybacked

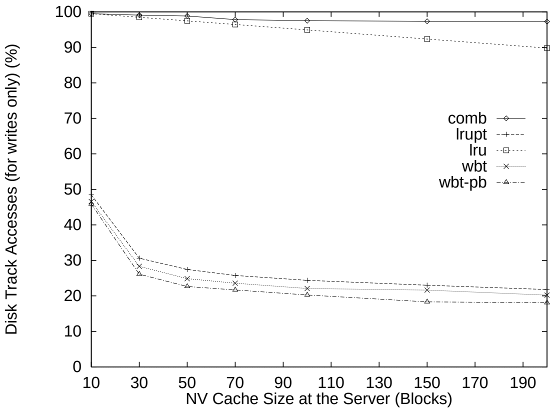



(c) Candidate Blocks for CPT, Client Policy is LRUPT

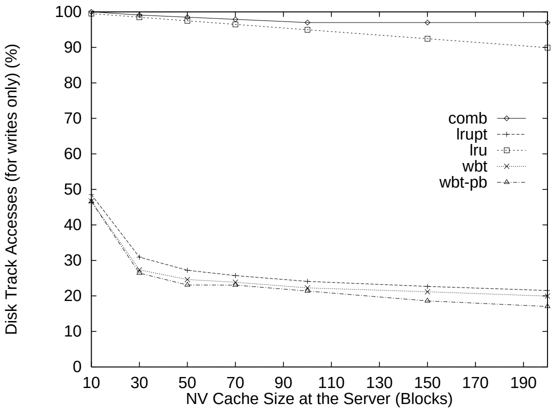



(d) Candidate for CPT, Client Policy is WBT

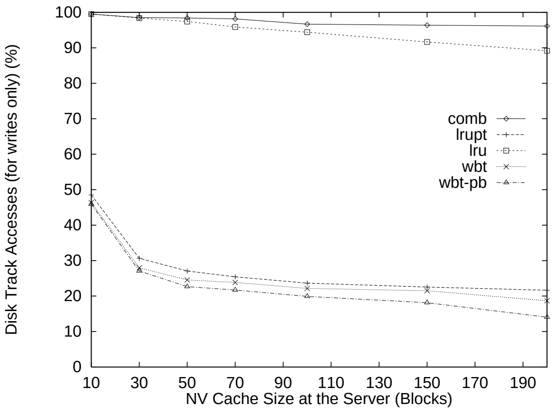
Figure B.36: Other Statistics. Client NV Cache Size is 100 Blocks



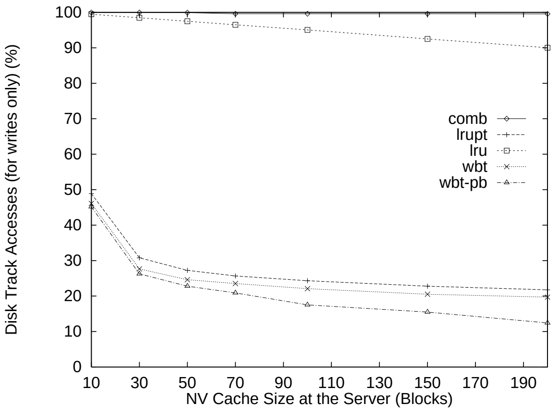

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

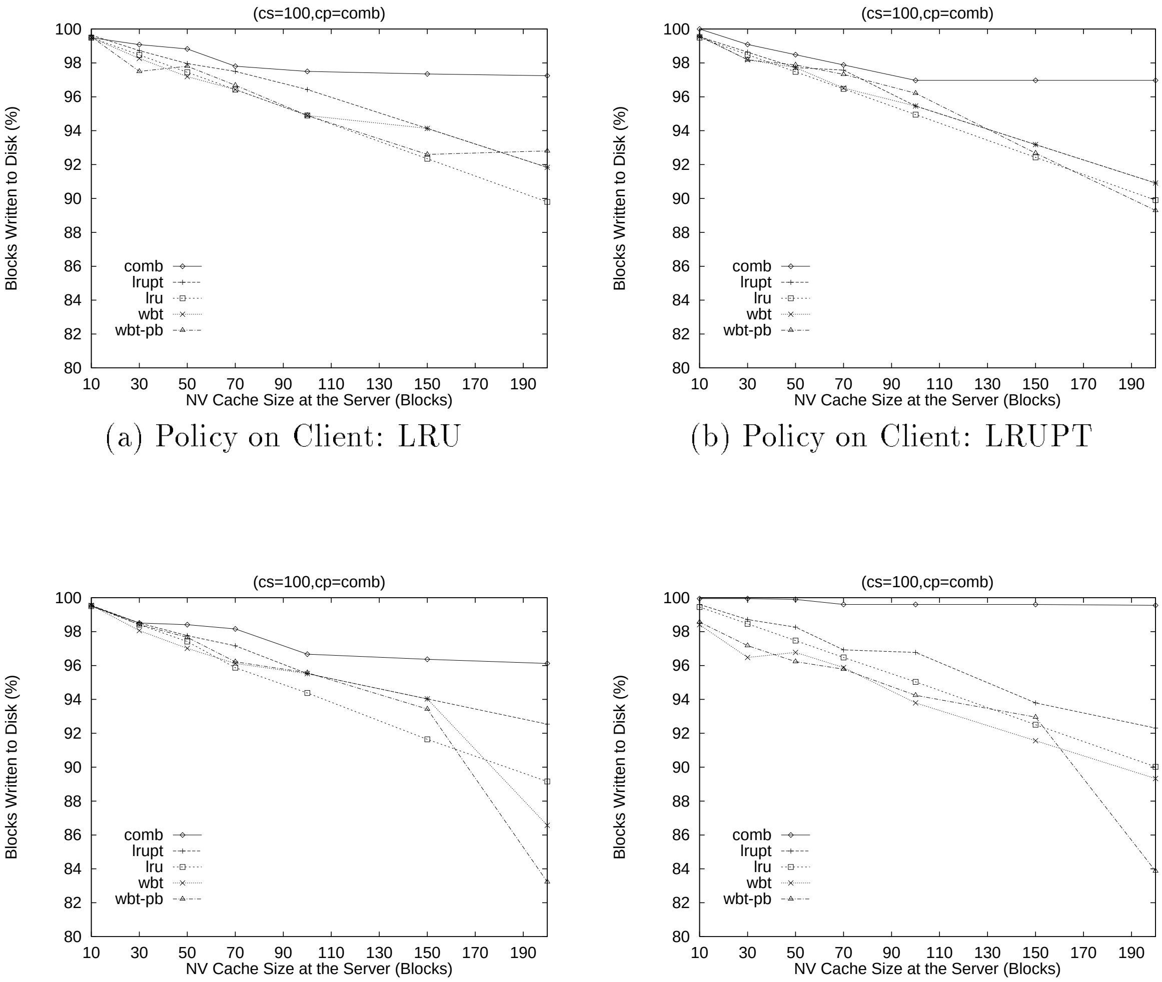



(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure B.37: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache Size is 100 Blocks



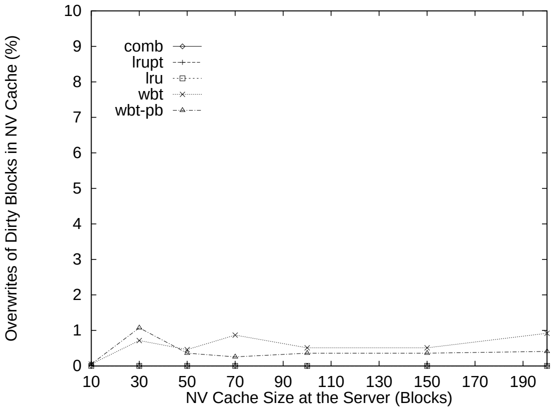

(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

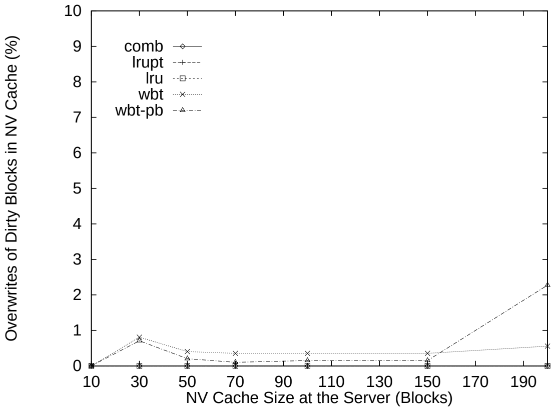
(c) Policy on Client: WBT

(d) Policy on Client: COMB

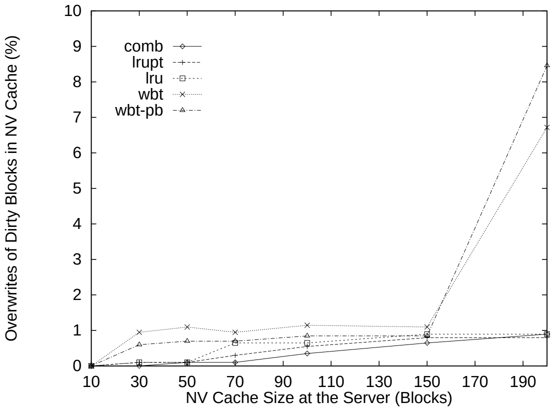
Figure B.38: Blocks Written to Disk, as a Percentage of Total Write Operations Seen at the Server. Client NV Cache Size is 100 Blocks



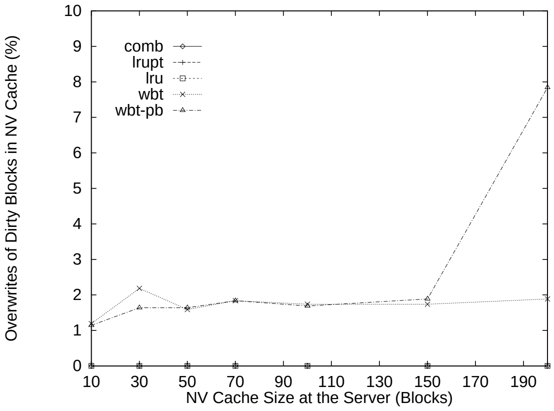

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

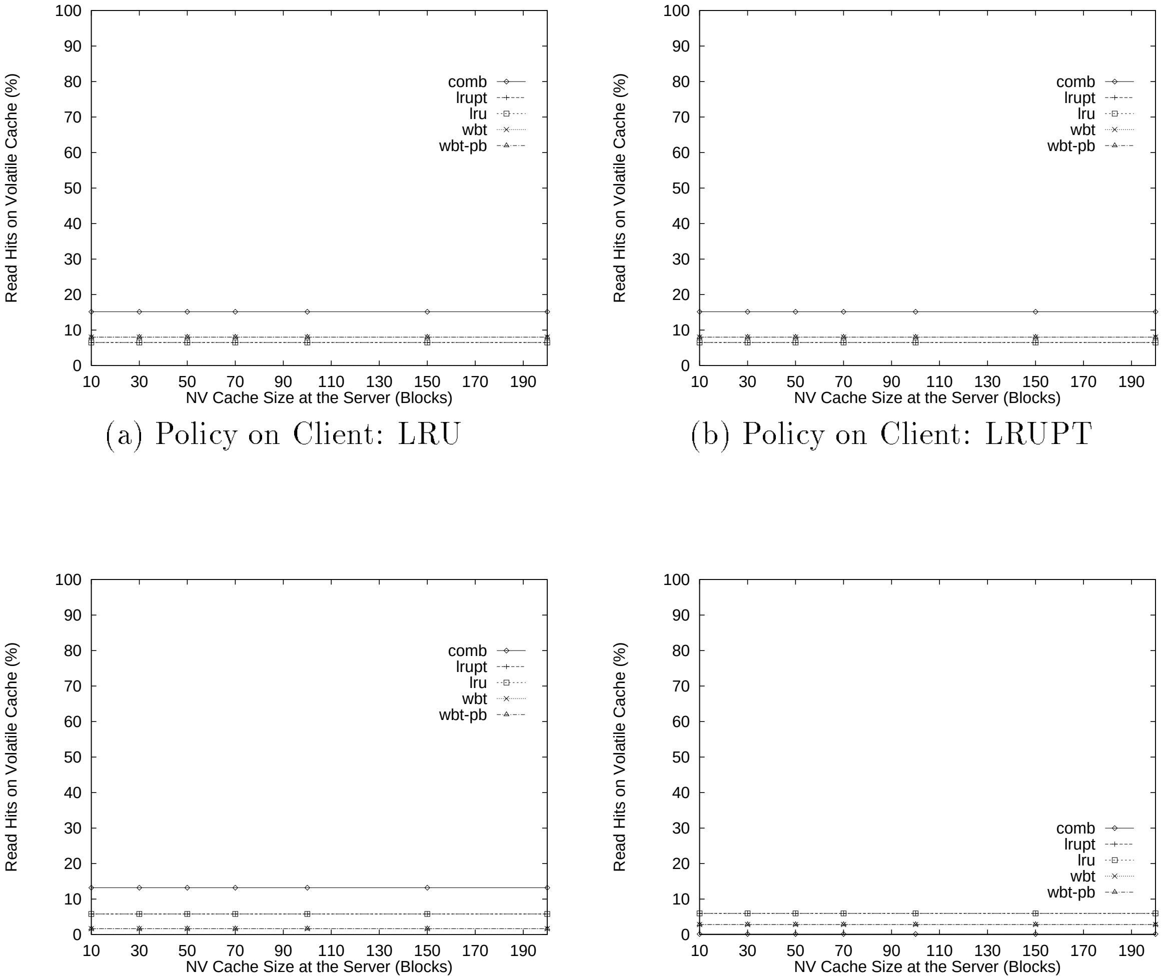



(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure B.39: Overwrites of Dirty Blocks in Server NV Cache, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache Size is 100 Blocks



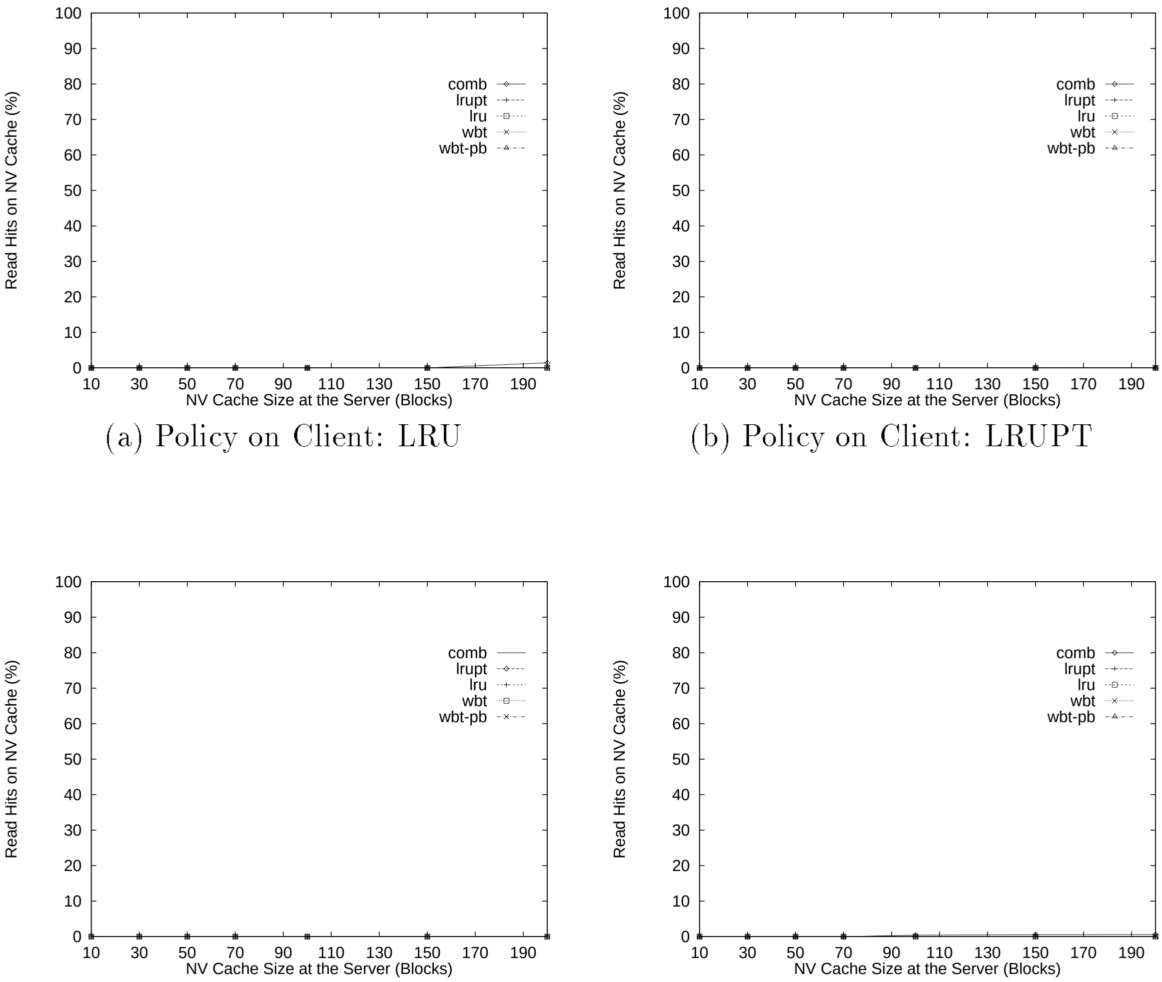

(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

(c) Policy on Client: WBT

(d) Policy on Client: COMB

Figure B.40: Overall Read Hit Ratio on Server Volatile Cache. Client NV Cache Size is 100 Blocks




(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

(c) Policy on Client: WBT

(d) Policy on Client: COMB

Figure B.41: Overall Read Hit Ratio on Server NV Cache. Client NV Cache Size is 100 Blocks

# Appendix C

# Simulation Results from Trace 3

All results obtained from trace 3 are shown in this appendix.

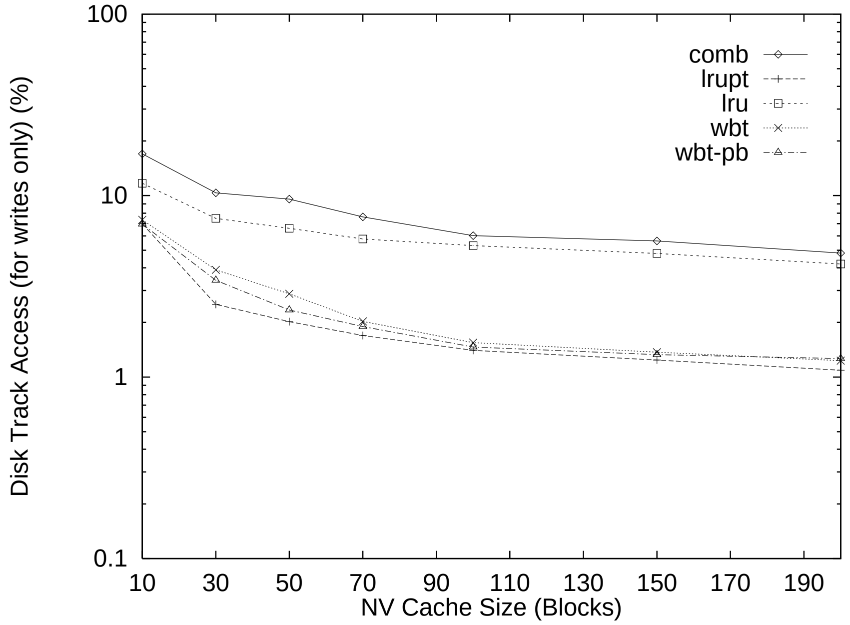



Figure C.1: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations (Single-Level NV Cache Configuration)

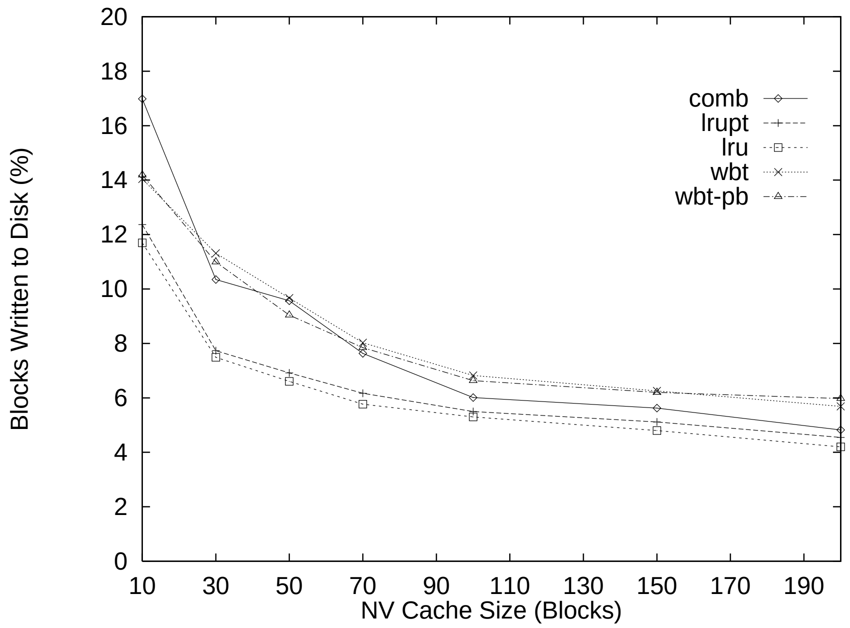



Figure C.2: Blocks Written to Disk as a Percentage of the Total Write Operations (Single-Level NV Cache Configuration)

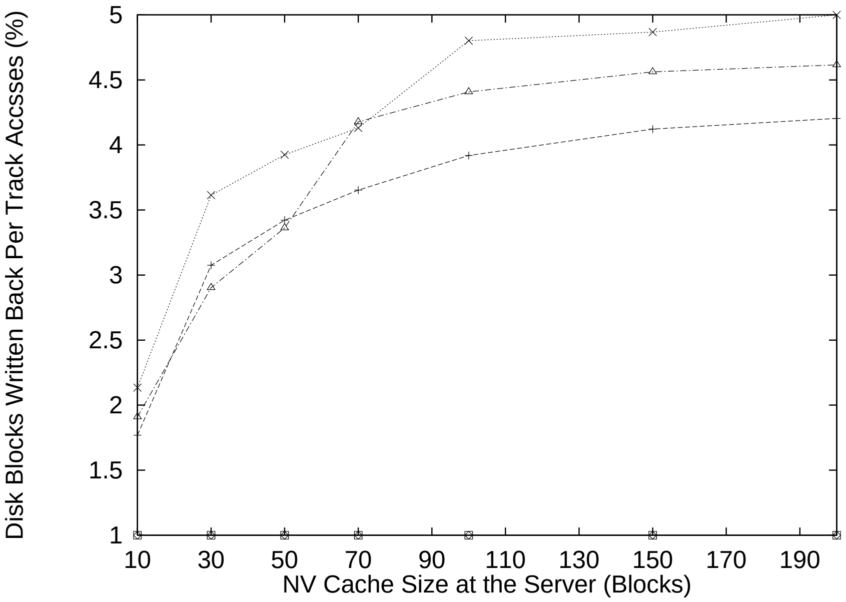



Figure C.3: Average Number of Dirty Blocks Written Back to Disk in One Disk Track Access

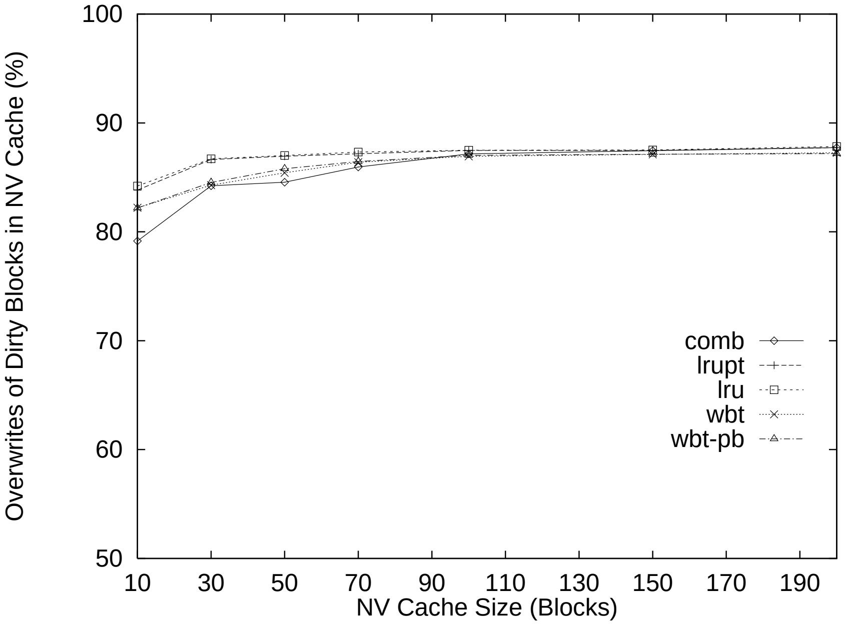



Figure C.4: Overwrites of Dirty Blocks as a Percentage of the Total Write Operations (Single-Level NV Cache Configuration)

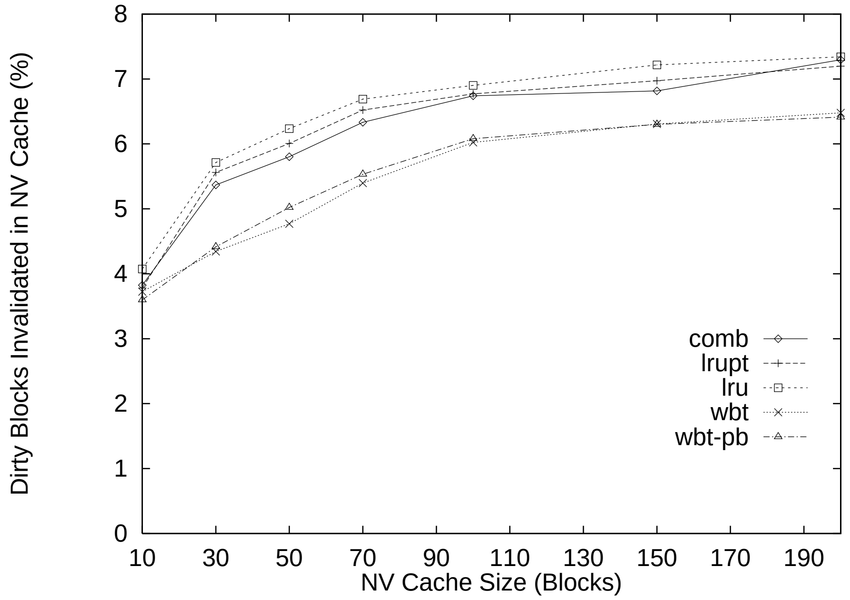



Figure C.5: Dirty Blocks Deleted from NV Cache as a Percentage of the Total Write Operations (Single-Level NV Caching Configuration)

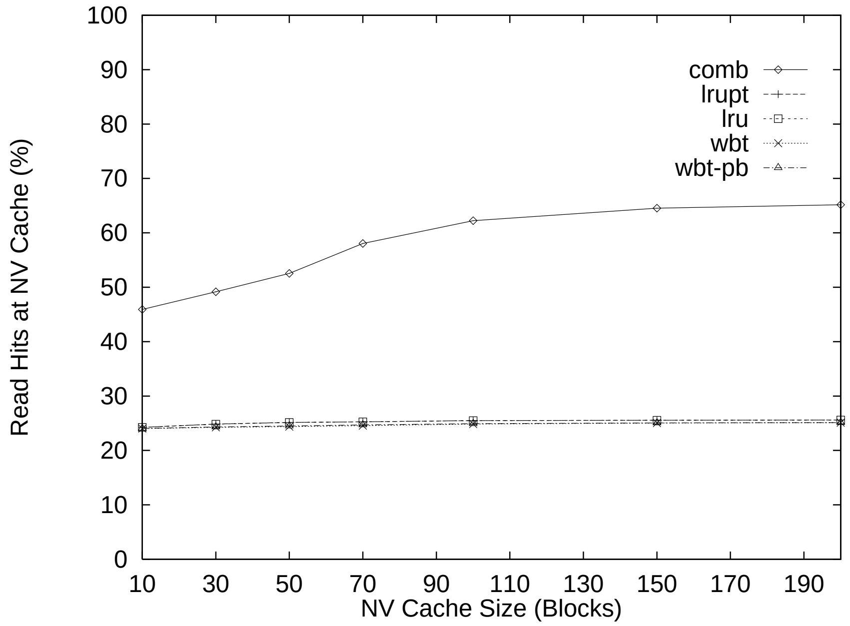



Figure C.6: Read Hit Ratio on NV Cache (Single-Level NV Cache Configuration)

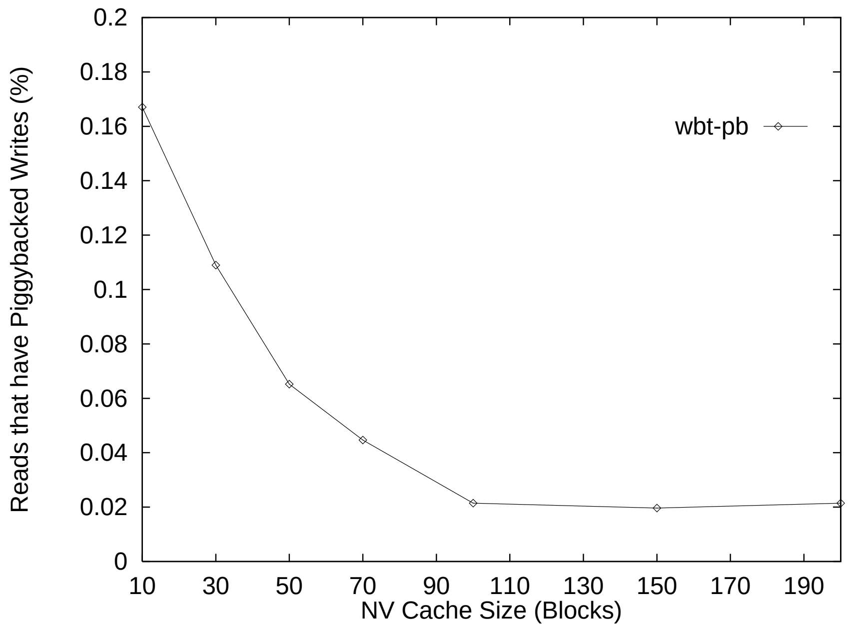



Figure C.7: Read Misses that have Writes Piggybacked (Single-Level NV Cache Configuration)

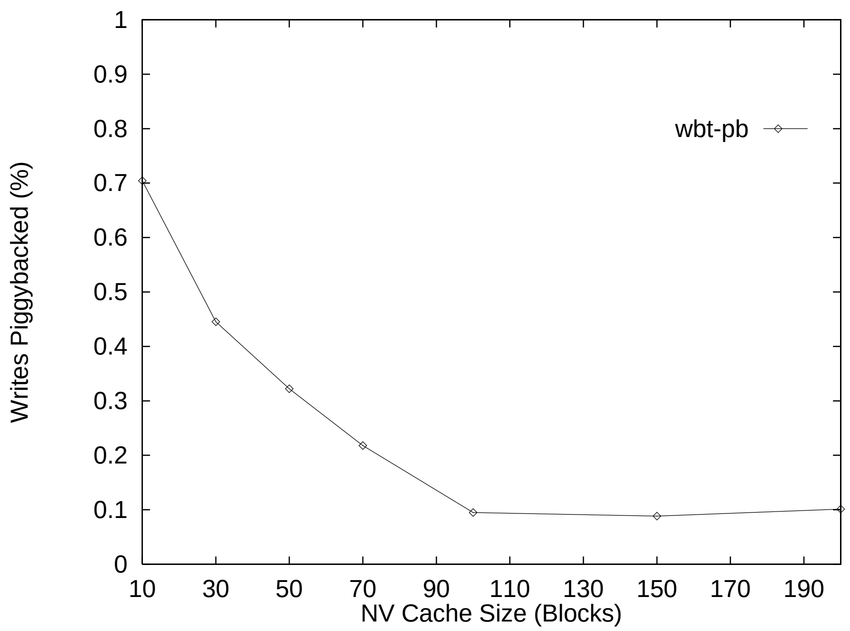
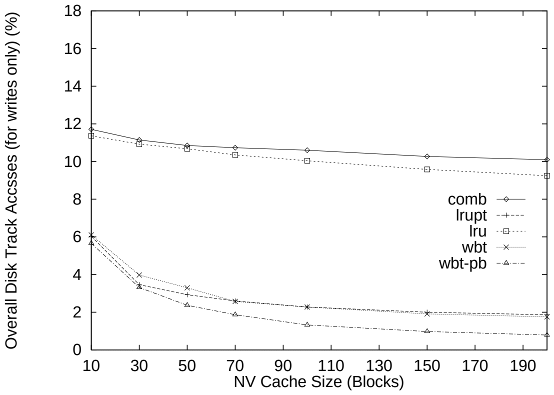
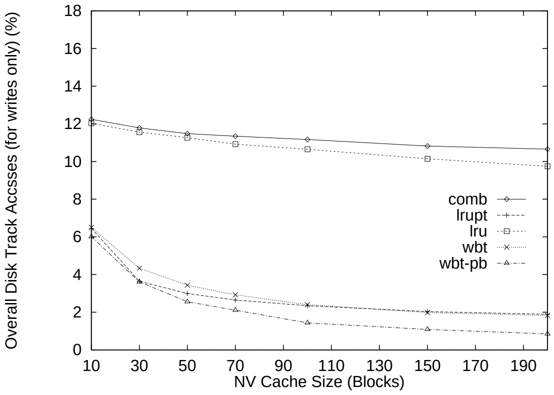


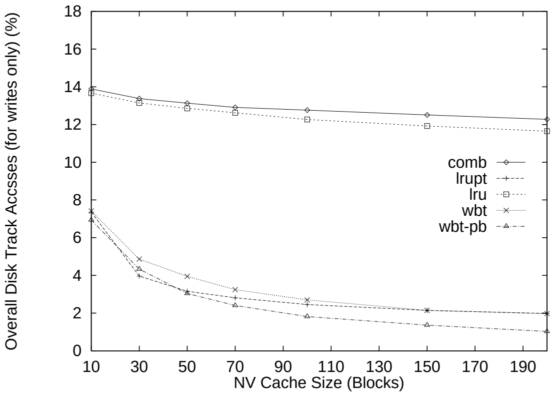

Figure C.8: Writes Piggybacked (Single-Level NV Cache Configuration)



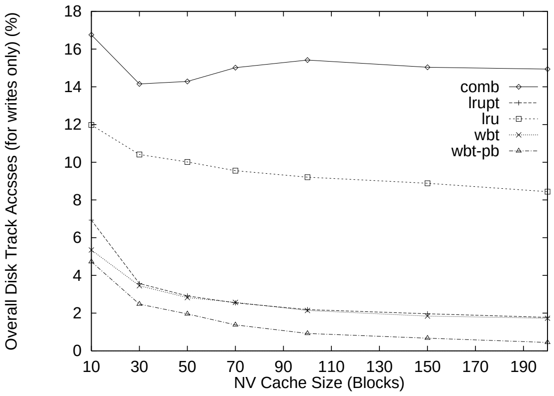

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

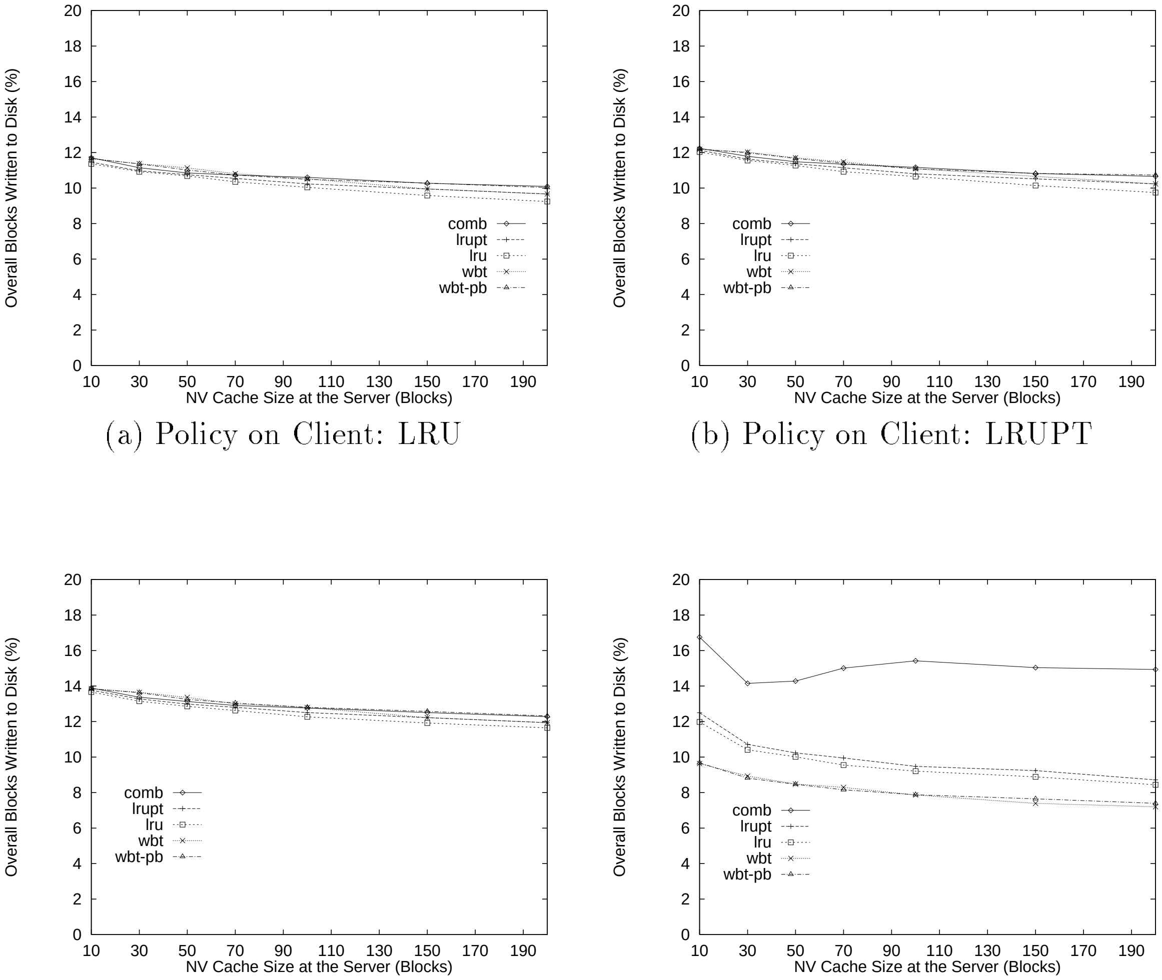



(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure C.9: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations. Client NV Cache Size is 10 Blocks



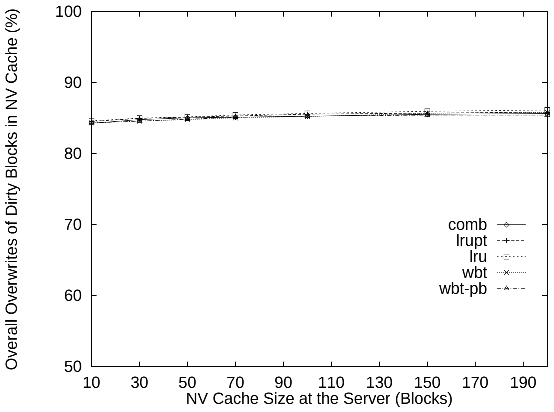

(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

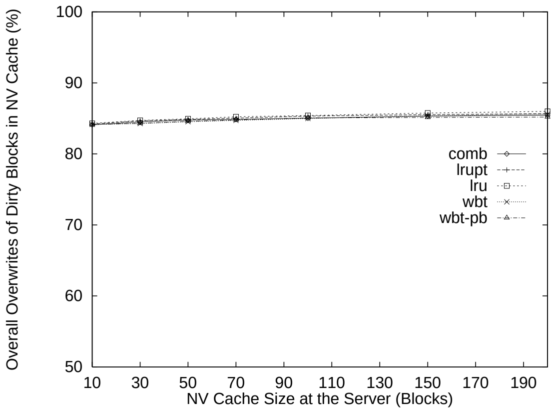
(c) Policy on Client: WBT

(d) Policy on Client: COMB

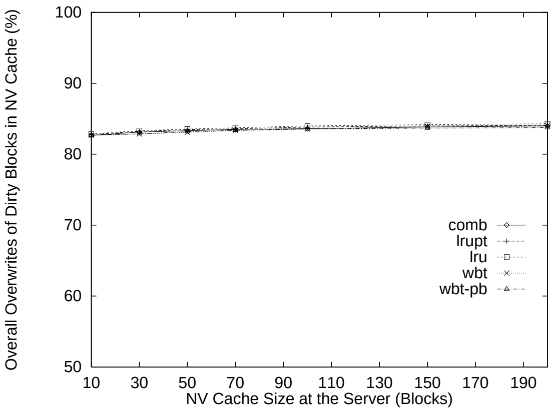
Figure C.10: Blocks Written to Disk, as a Percentage of Total Write Operations. Client NV Cache Size is 10 Blocks



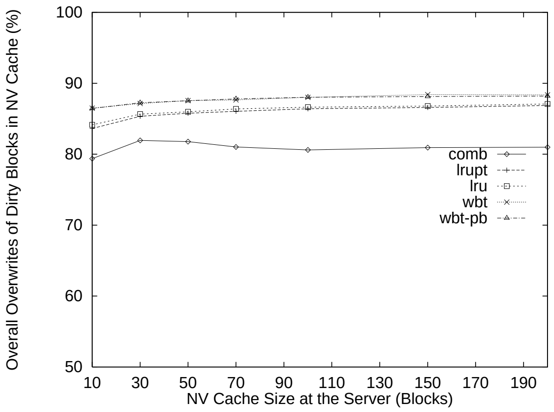

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

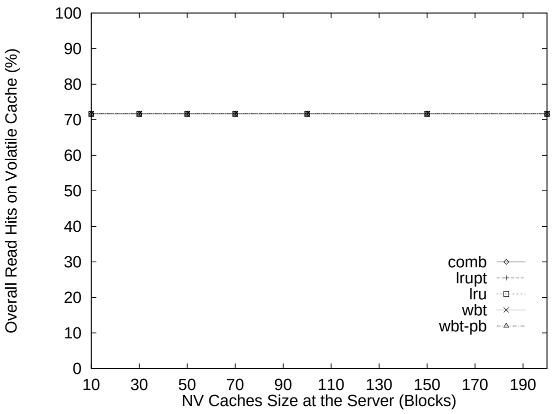



(c) Policy on Client: WBT

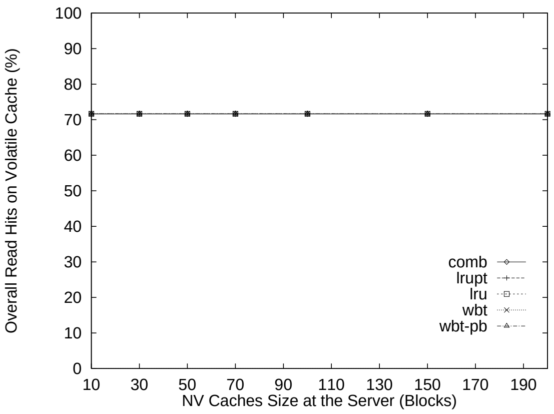



(d) Policy on Client: COMB

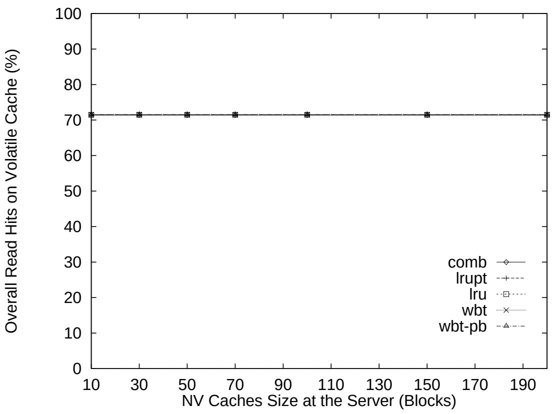
Figure C.11: Overwrites of Dirty Blocks in NV Caches, as a Percentage of the Total Write Operations. Client NV Cache Size is 10 Blocks



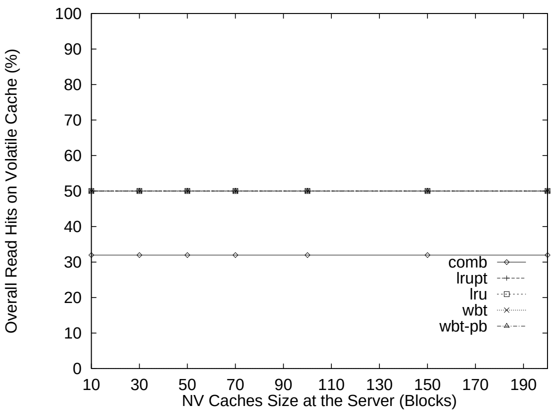

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

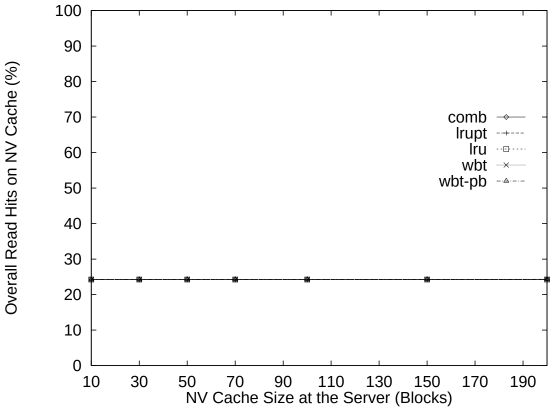



(c) Policy on Client: WBT

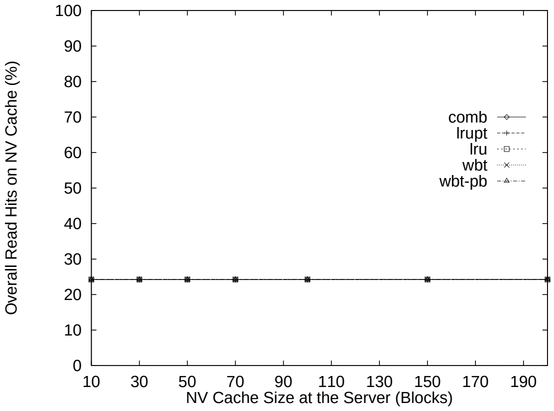



(d) Policy on Client: COMB

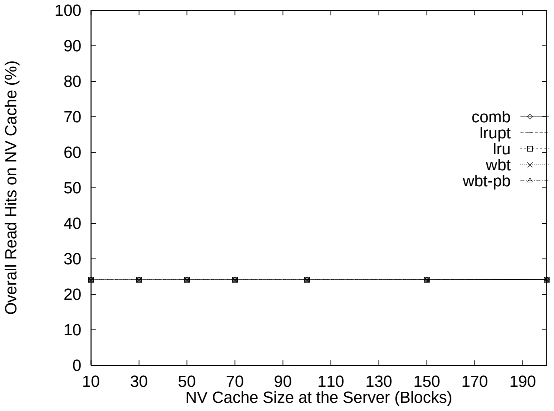
Figure C.12: Overall Read Hit Ratio on Volatile Caches. Client NV Cache Size is 10 Blocks



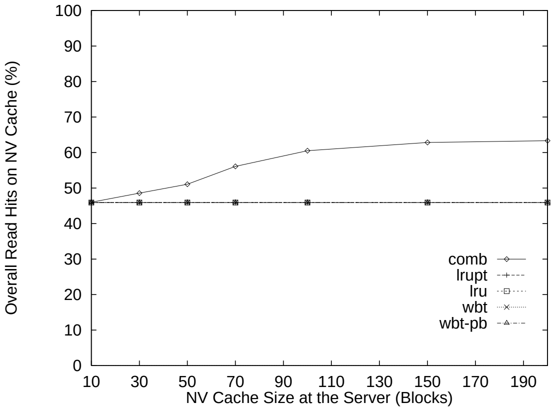

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

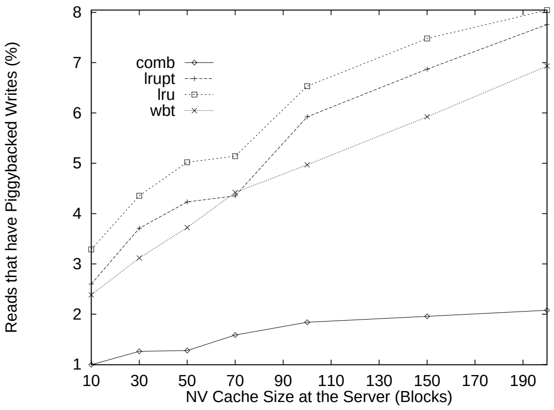



(c) Policy on Client: WBT

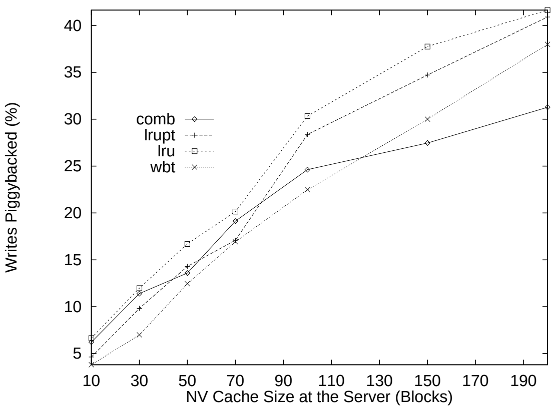



(d) Policy on Client: COMB

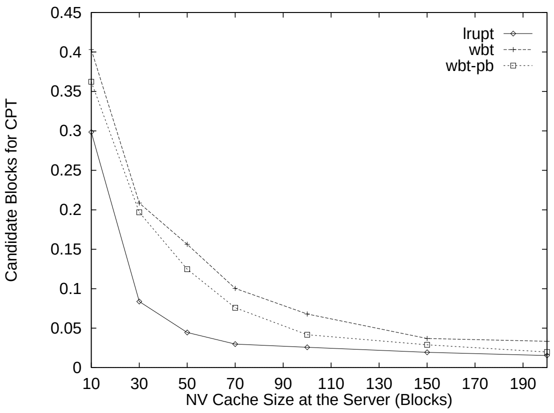
Figure C.13: Overall Read Hit Ratio on NV Caches. Client NV Cache Size is 10 Blocks



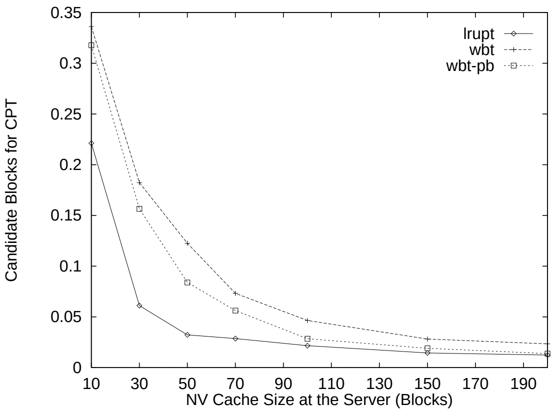

(a) Reads Misses that have Writes Piggybacked




(b) Writes Piggybacked

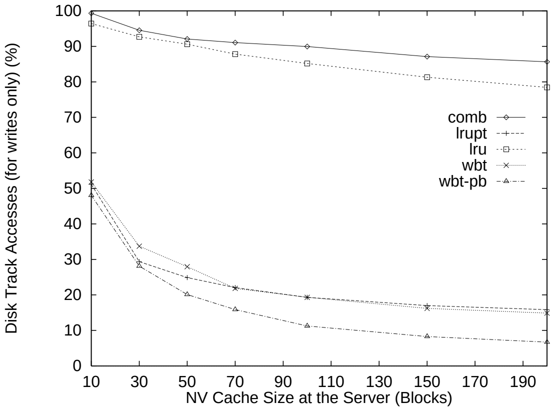



(c) Candidate Blocks for CPT, Client Policy is LRUPT

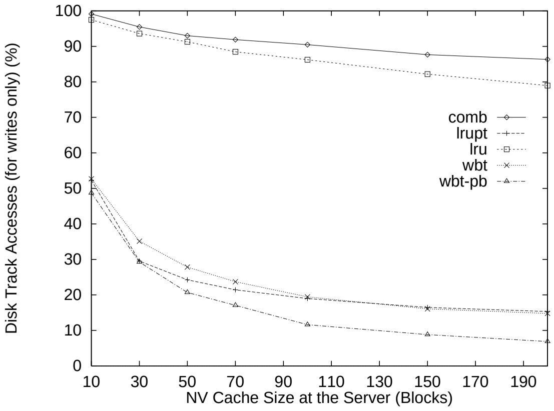



(d) Candidate for CPT, Client Policy is WBT

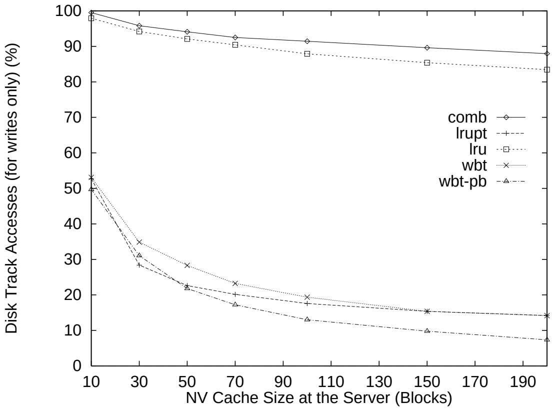
Figure C.14: Other Statistics. Client NV Cache Size is 10 Blocks



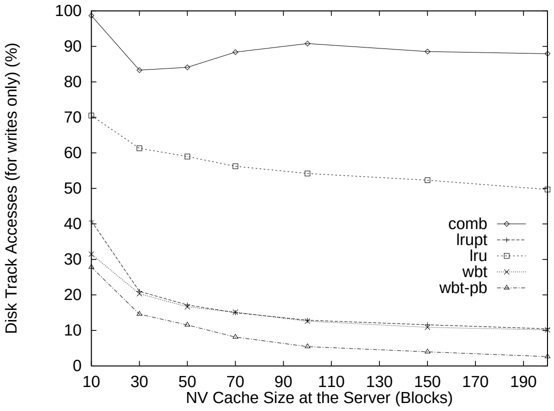

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT




(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure C.15: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache Size is 10 Blocks

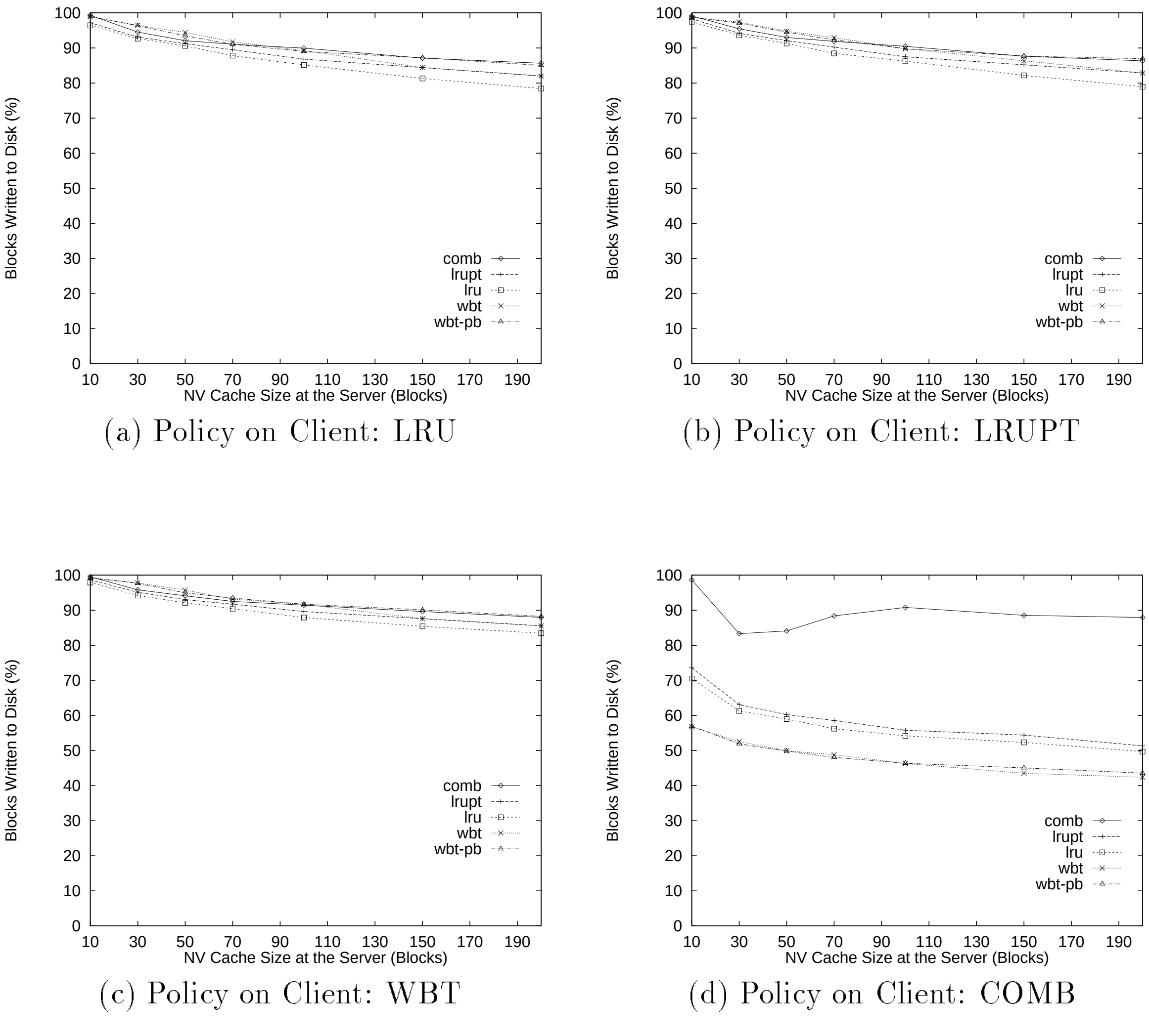
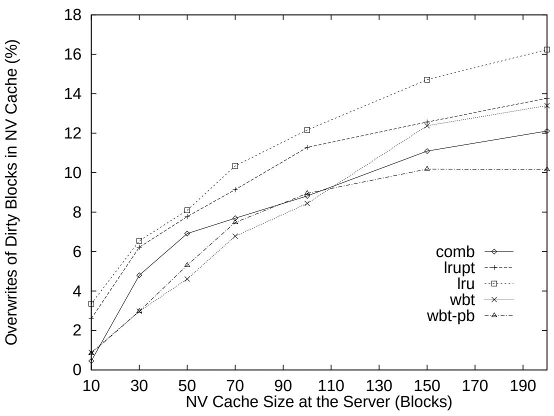
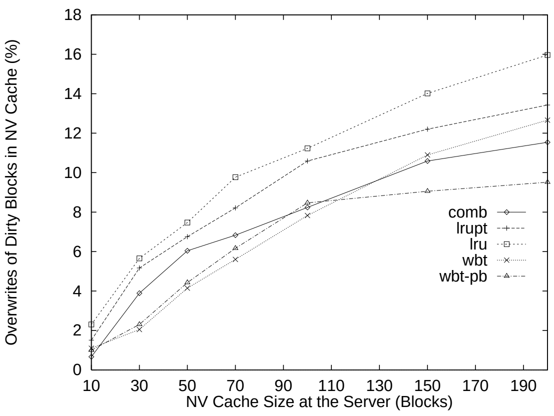


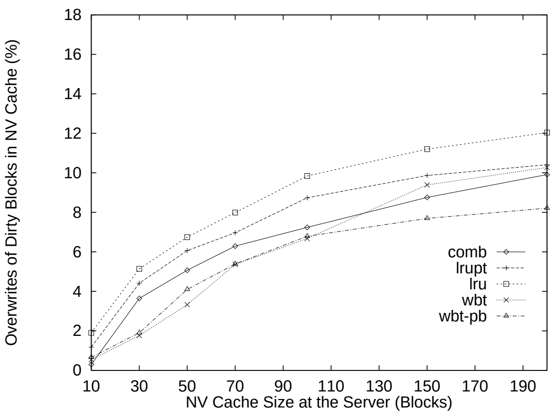

Figure C.16: Blocks Written to Disk, as a Percentage of Total Write Operations Seen at the Server. Client NV Cache Size is 10 Blocks



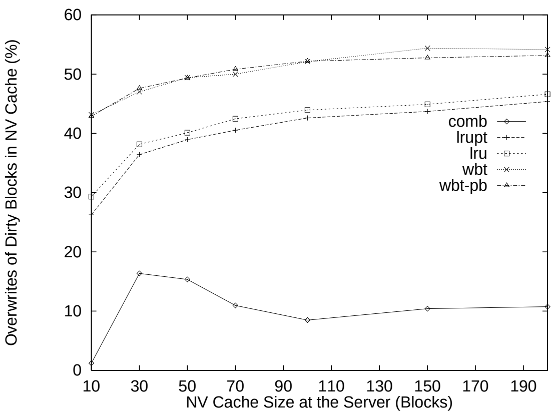

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

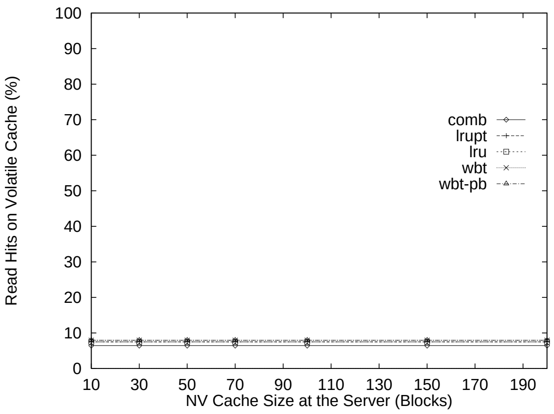



(c) Policy on Client: WBT

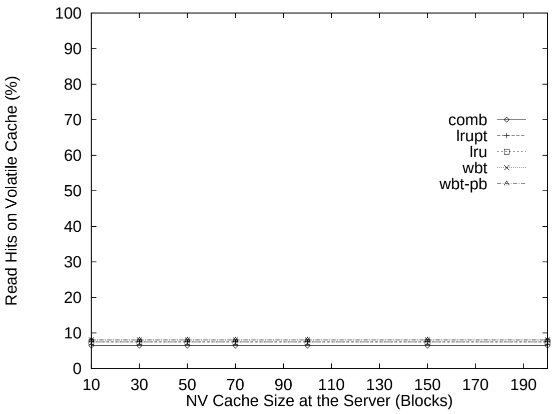



(d) Policy on Client: COMB

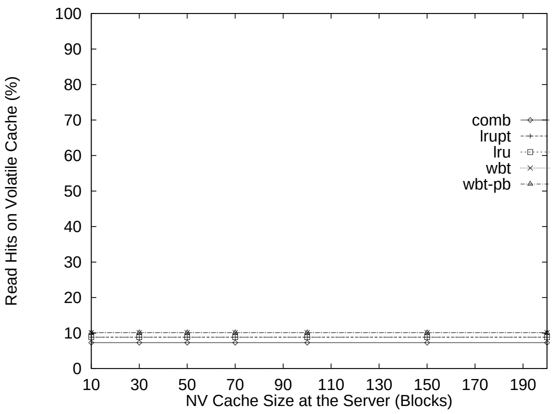
Figure C.17: Overwrites of Dirty Blocks in Server NV Cache, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache Size is 10 Blocks



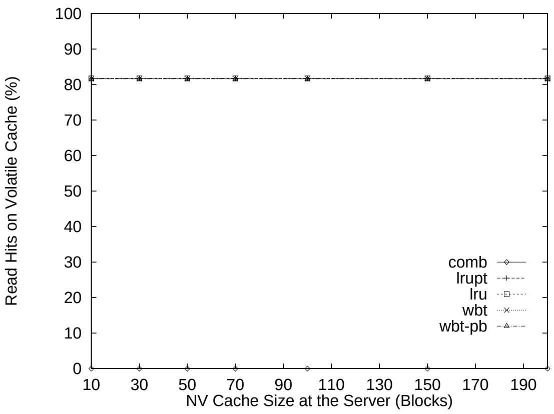

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

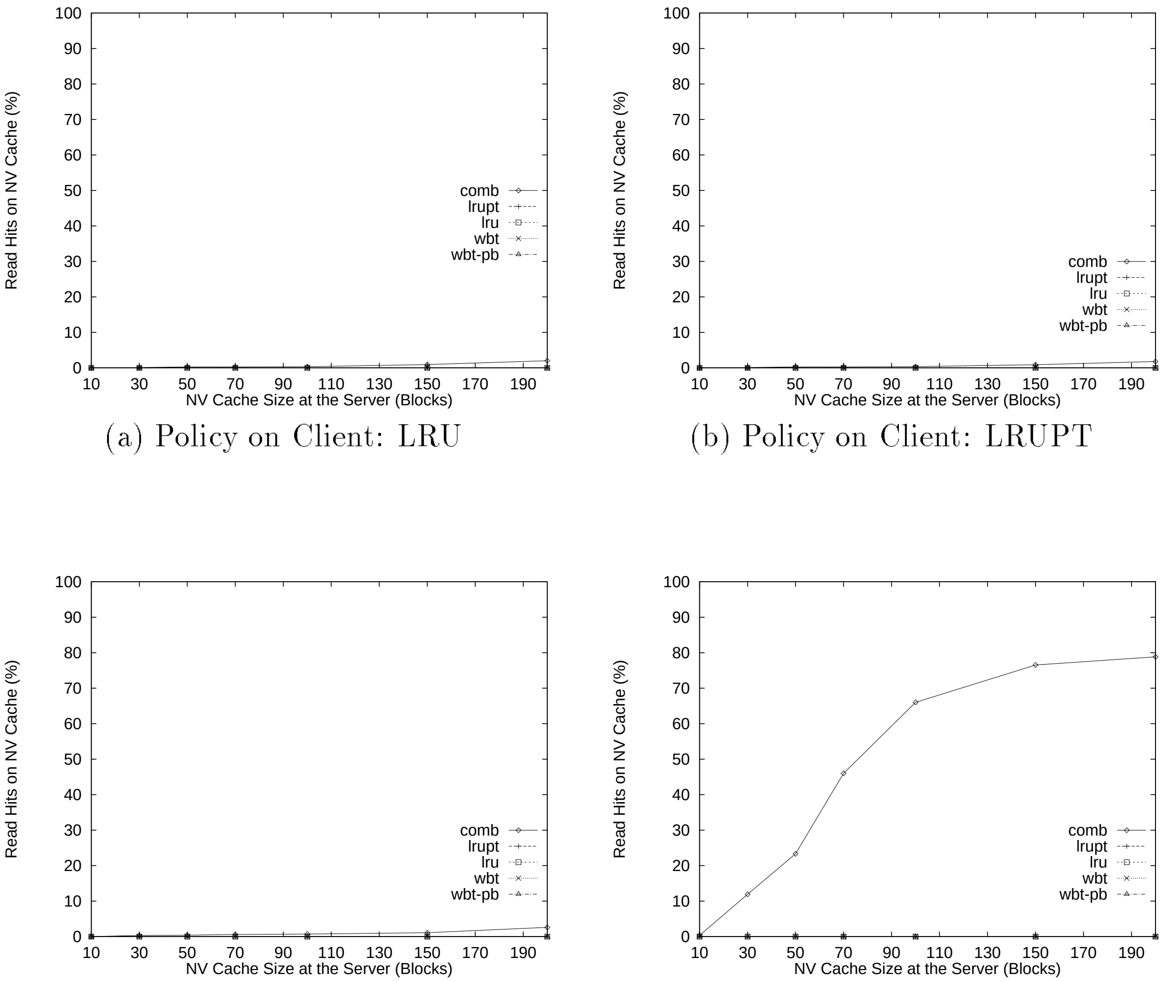



(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure C.18: Overall Read Hit Ratio on Server Volatile Cache. Client NV Cache Size is 10 Blocks



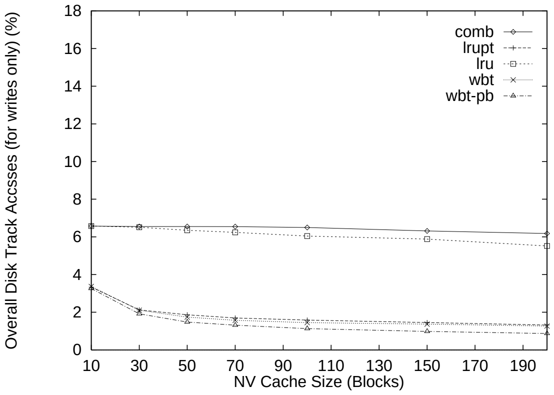

(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

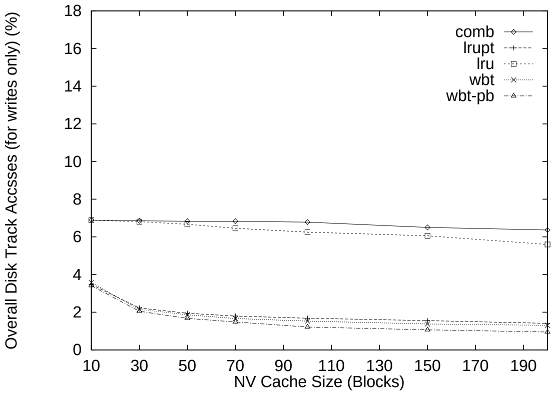
(c) Policy on Client: WBT

(d) Policy on Client: COMB

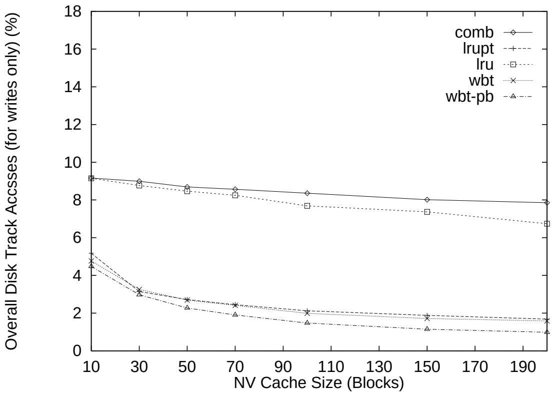
Figure C.19: Overall Read Hit Ratio on Server NV Cache. Client NV Cache Size is 10 Blocks



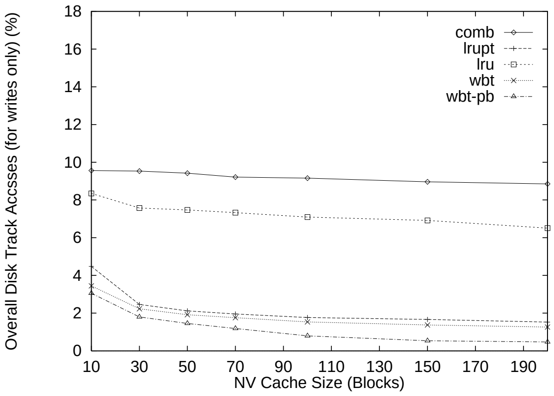

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

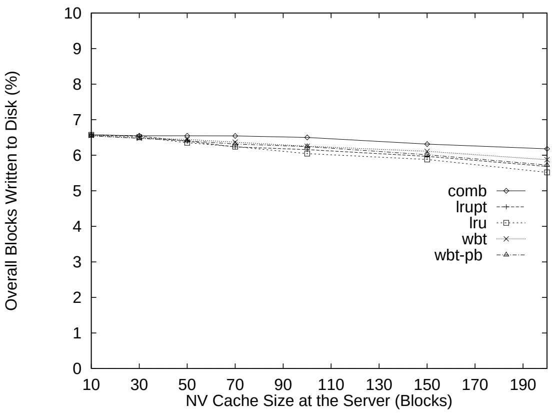



(c) Policy on Client: WBT

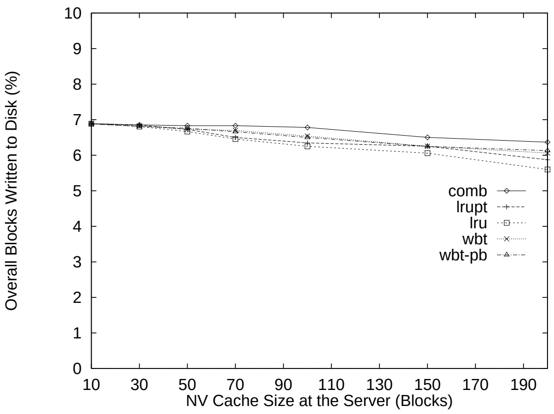



(d) Policy on Client: COMB

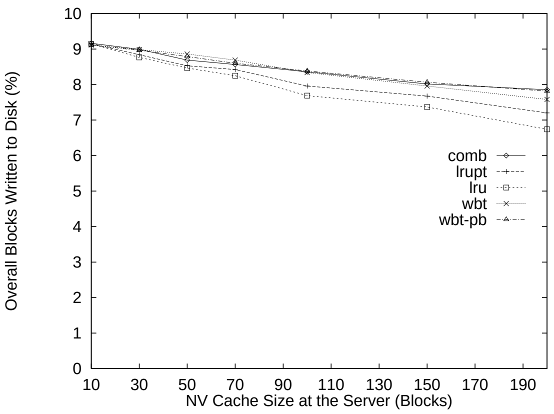
Figure C.20: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations. Client NV Cache Size is 50 Blocks



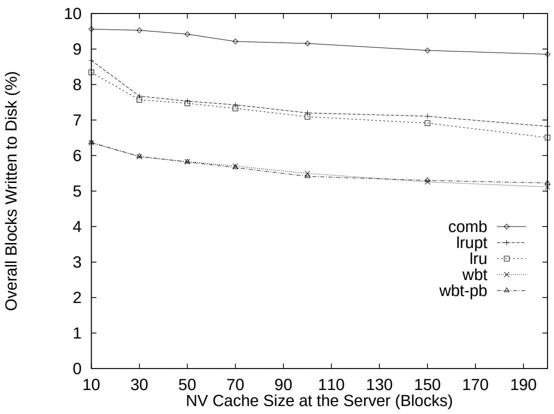

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

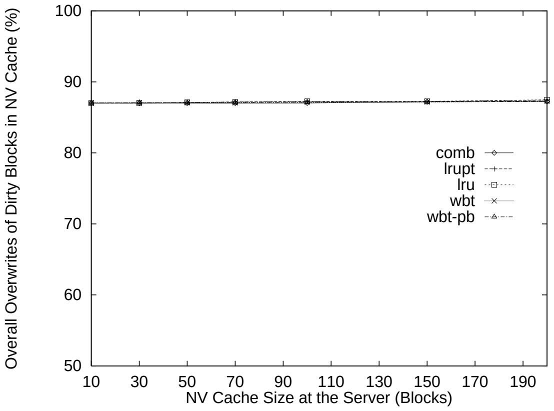



(c) Policy on Client: WBT

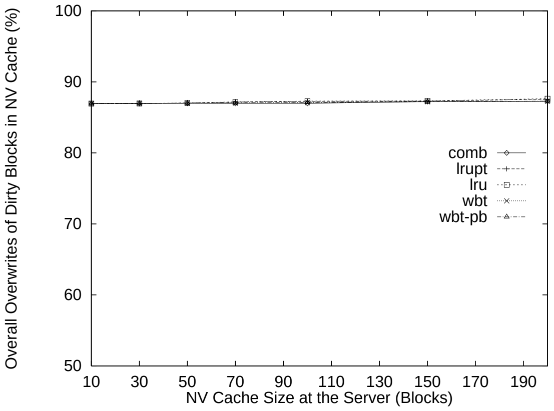



(d) Policy on Client: COMB

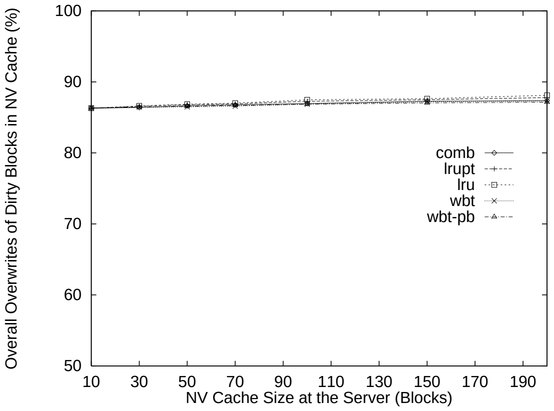
Figure C.21: Blocks Written to Disk, as a Percentage of Total Write Operations. Client NV Cache Size is 50 Blocks



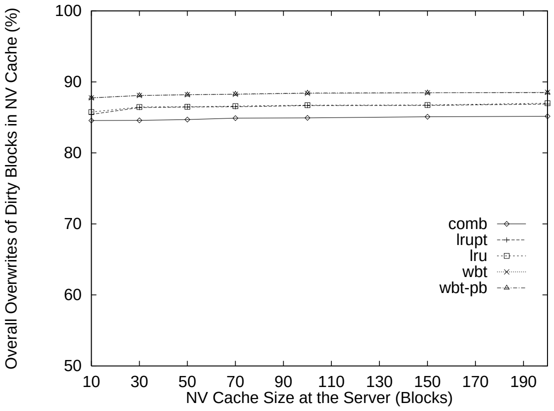

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

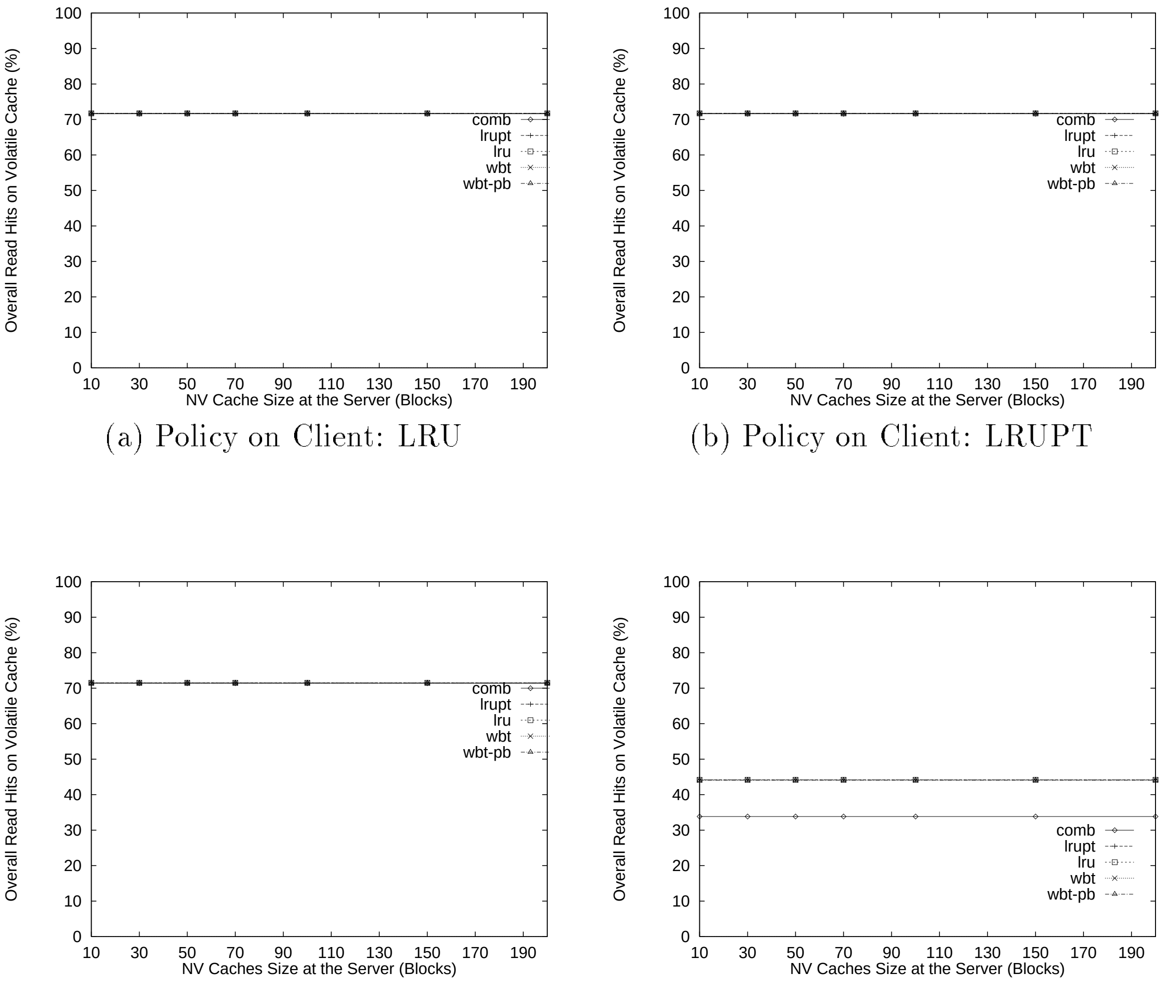



(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure C.22: Overwrites of Dirty Blocks in NV Caches, as a Percentage of the Total Write Operations. Client NV Cache Size is 50 Blocks




(a) Policy on Client: LRU



(b) Policy on Client: LRUPT

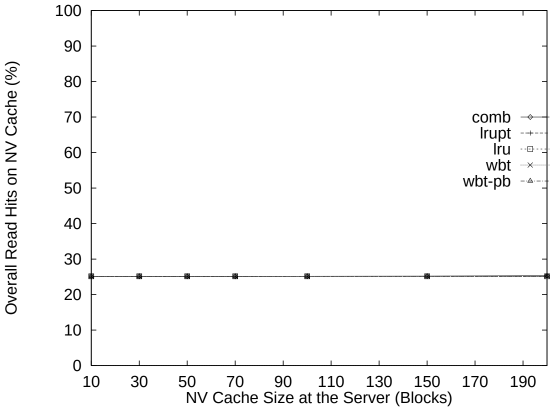


(c) Policy on Client: WBT

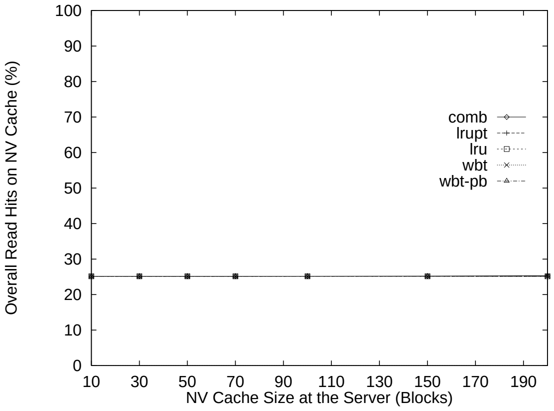


(d) Policy on Client: COMB

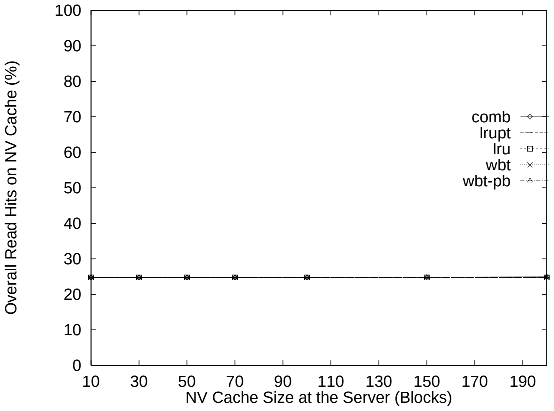
Figure C.23: Overall Read Hit Ratio on Volatile Caches. Client NV Cache Size is 50 Blocks



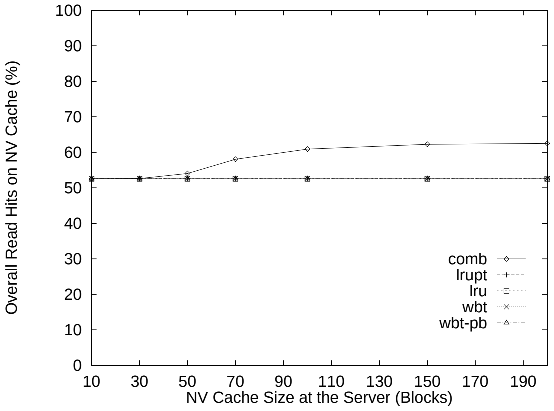

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

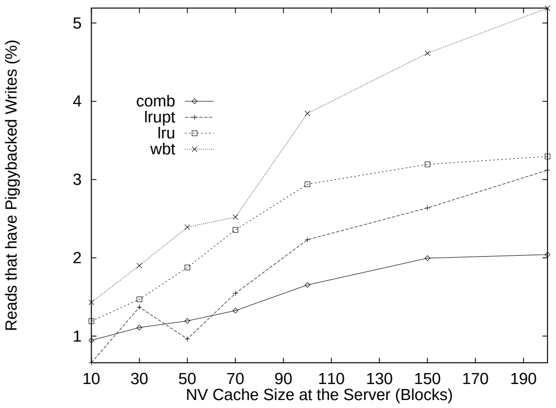



(c) Policy on Client: WBT

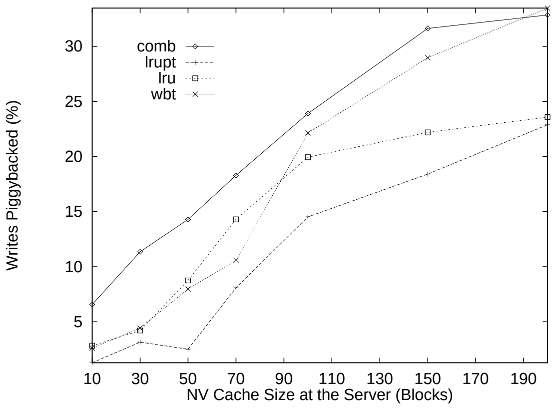



(d) Policy on Client: COMB

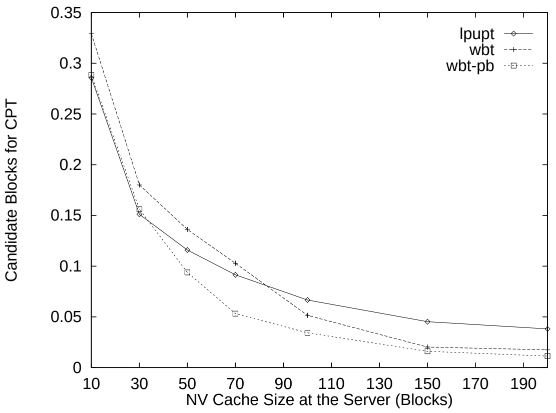
Figure C.24: Overall Read Hit Ratio on NV Caches. Client NV Cache Size is 50 Blocks



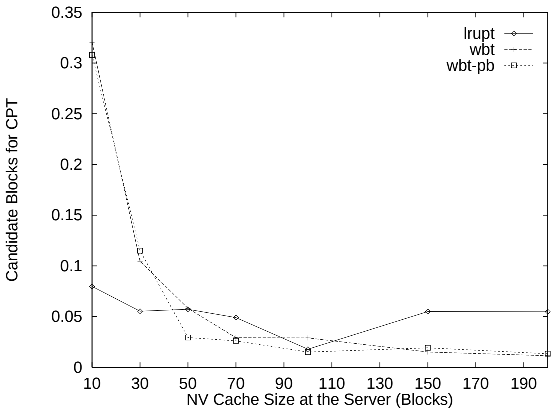

(a) Reads Misses that have Writes Piggybacked




(b) Writes Piggybacked

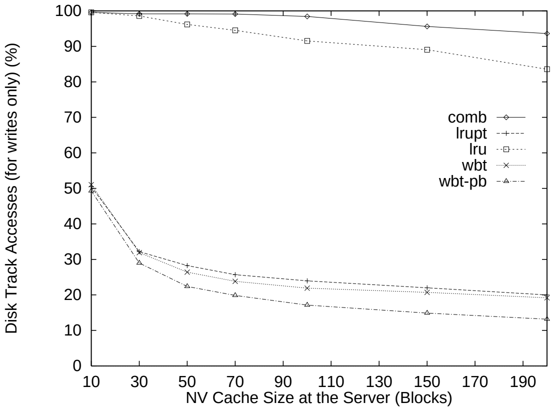



(c) Candidate BLocks for CPT, Client Policy is LRUPT

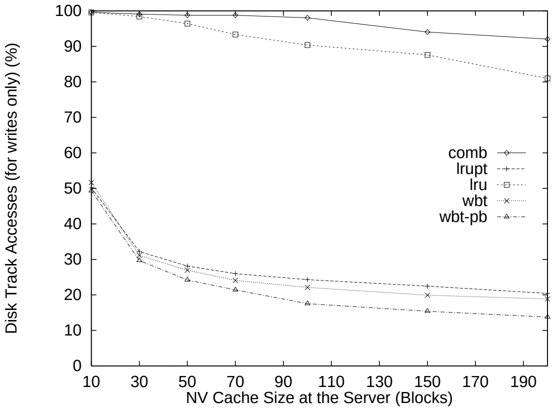



(d) Candidate for CPT, Client Policy is WBT

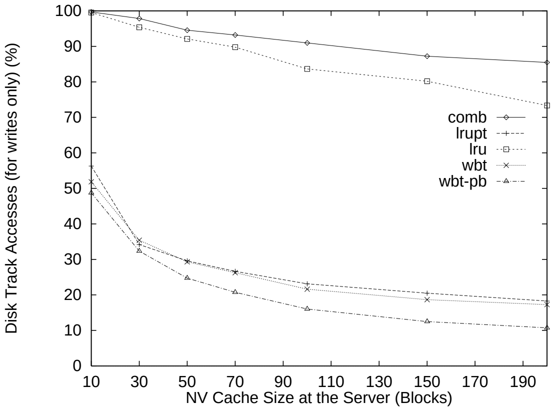
Figure C.25: Other Statistics. Client NV Cache Size is 50 Blocks



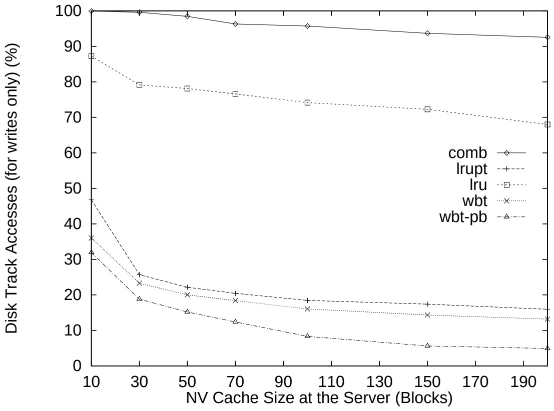

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

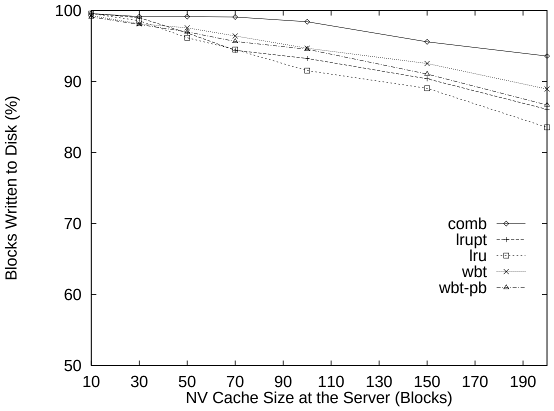



(c) Policy on Client: WBT

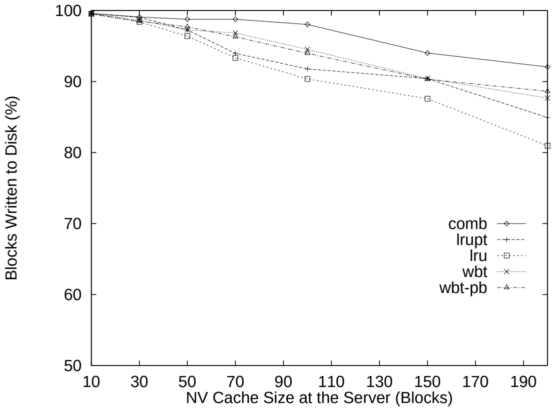



(d) Policy on Client: COMB

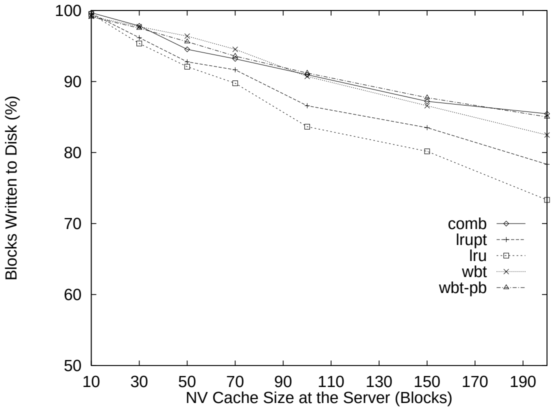
Figure C.26: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache Size is 50 Blocks



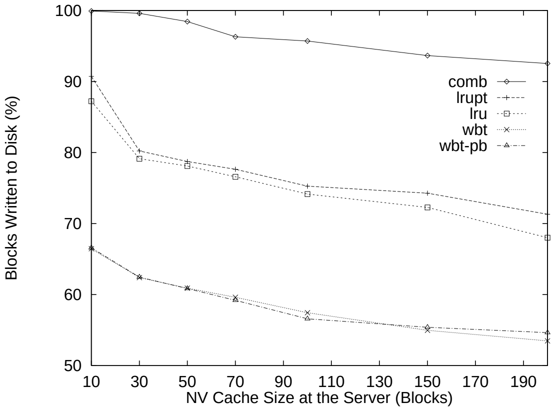

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

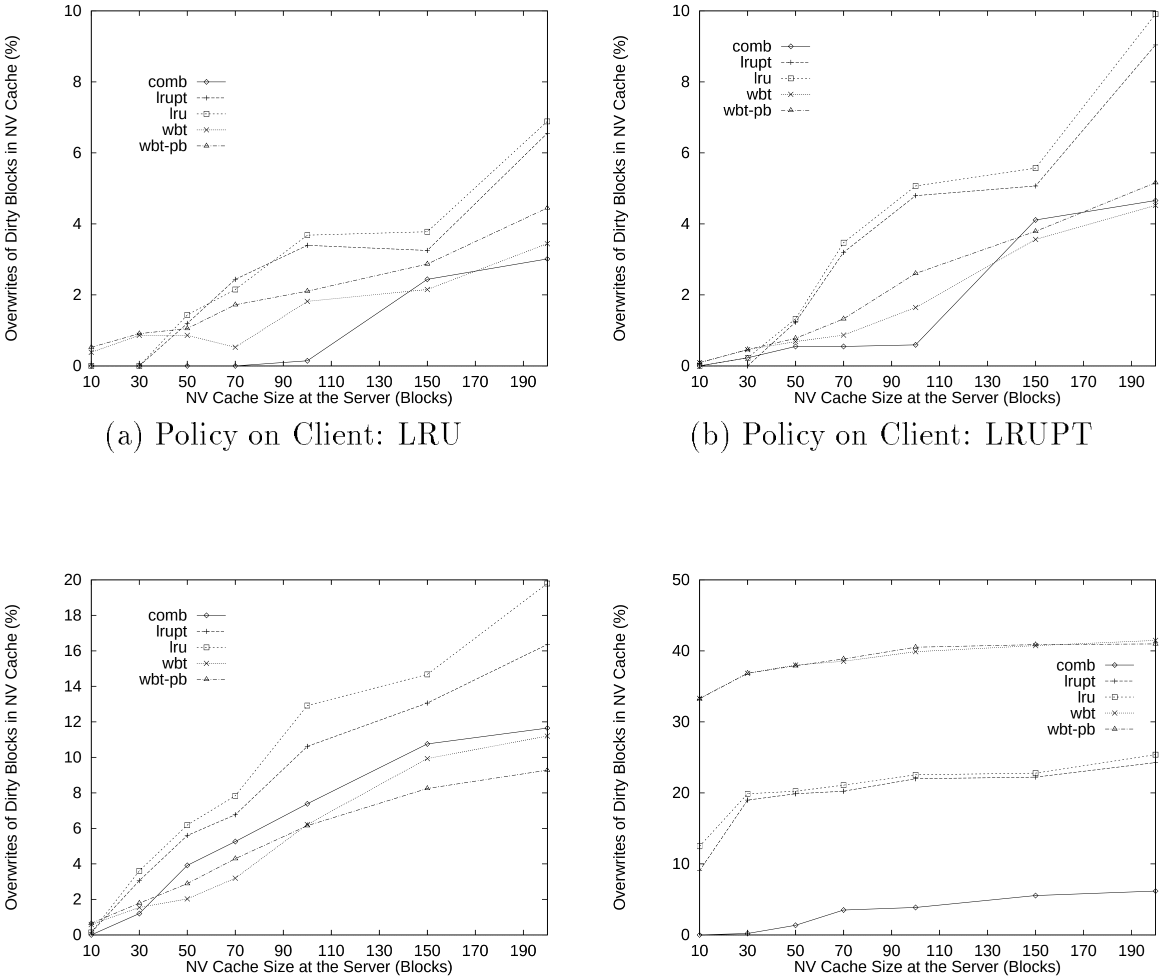



(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure C.27: Blocks Written to Disk, as a Percentage of Total Write Operations Seen at the Server. Client NV Cache Size is 50 Blocks



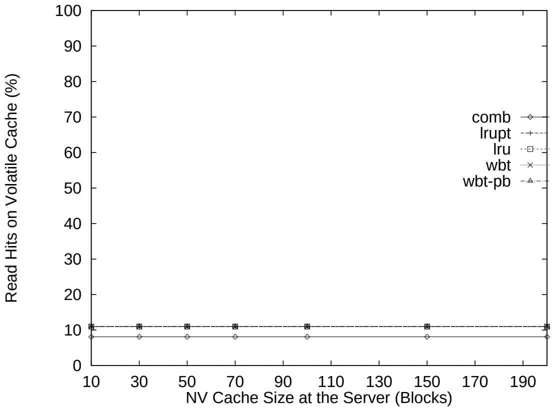

(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

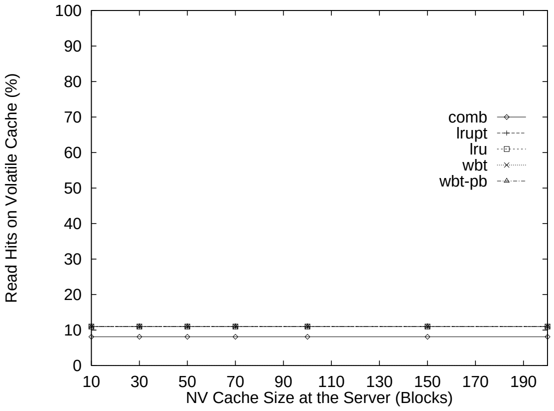
(c) Policy on Client: WBT

(d) Policy on Client: COMB

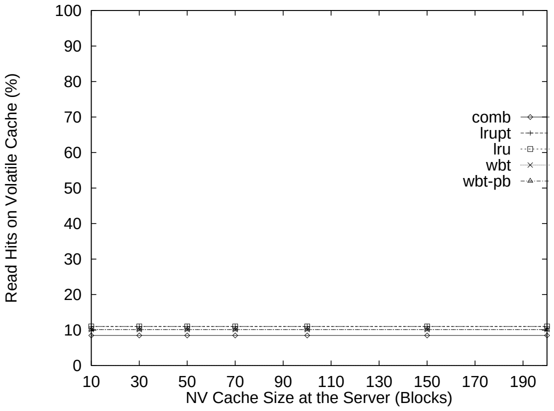
Figure C.28: Overwrites of Dirty Blocks in Server NV Cache, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache Size is 50 Blocks



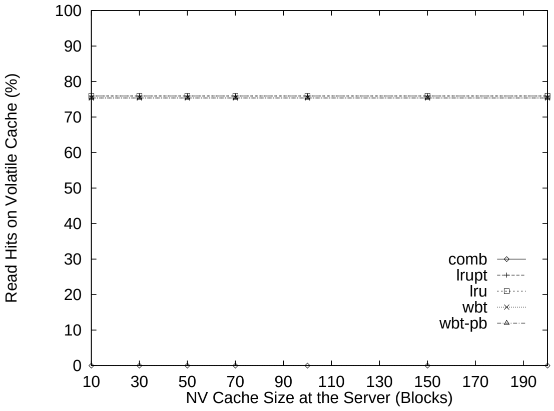

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

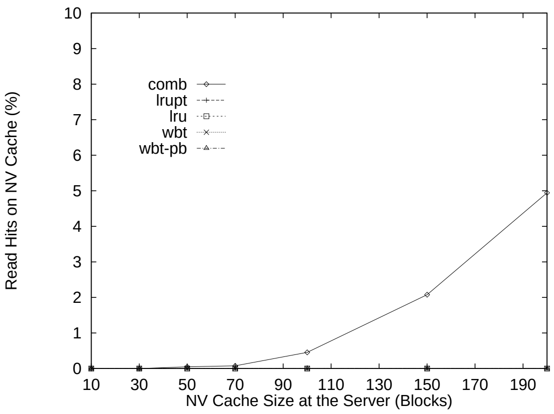



(c) Policy on Client: WBT

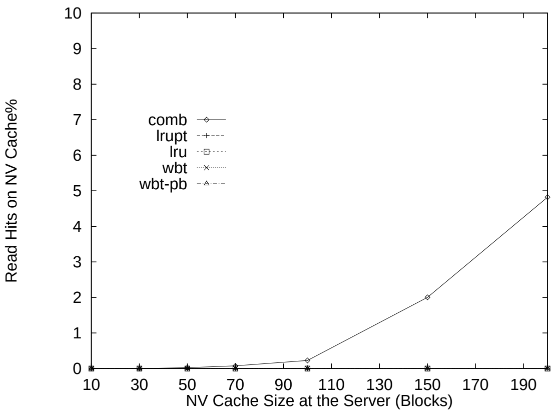



(d) Policy on Client: COMB

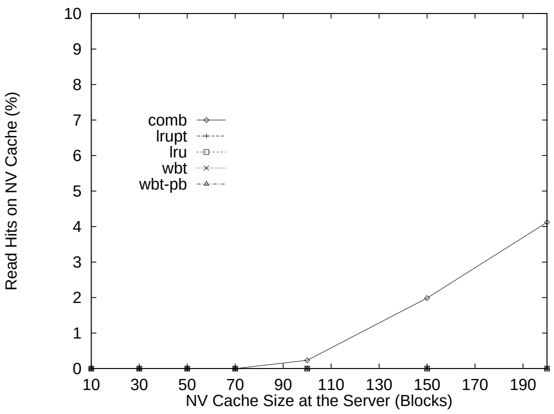
Figure C.29: Overall Read Hit Ratio on Server Volatile Cache. Client NV Cache Size is 50 Blocks



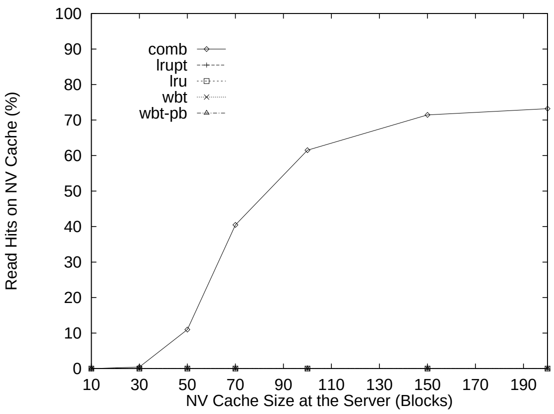

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

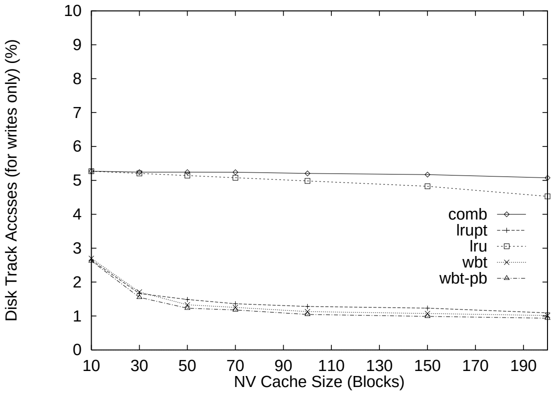



(c) Policy on Client: WBT

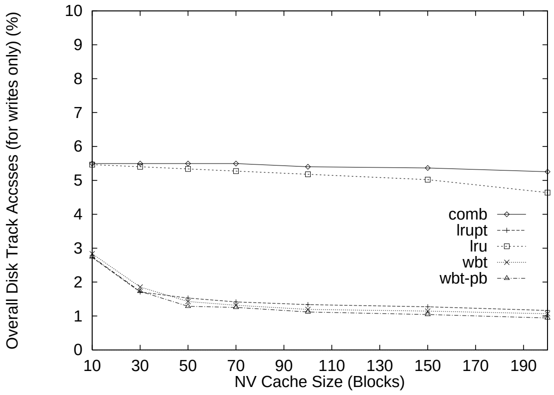



(d) Policy on Client: COMB

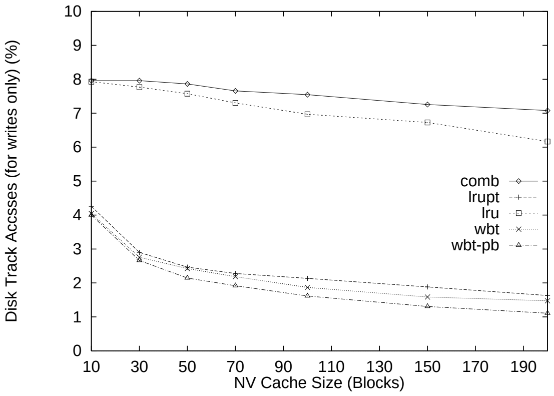
Figure C.30: Overall Read Hit Ratio on Server NV Cache. Client NV Cache Size is 50 Blocks



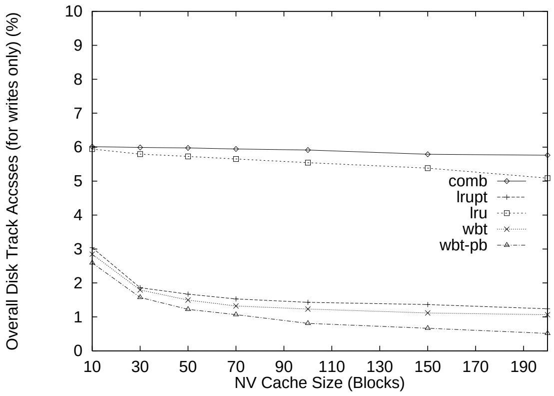

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

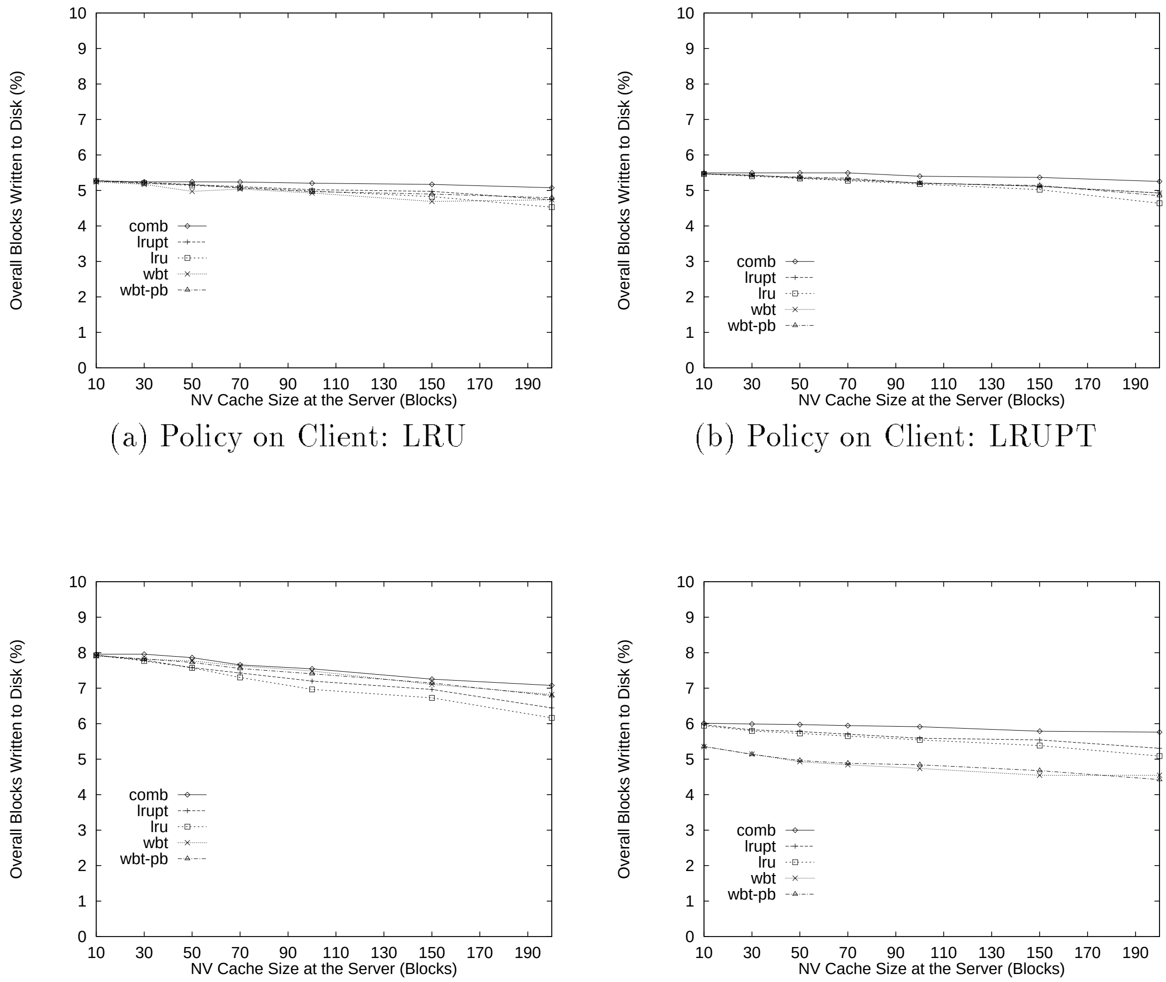



(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure C.31: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations. Client NV Cache Size is 100 Blocks



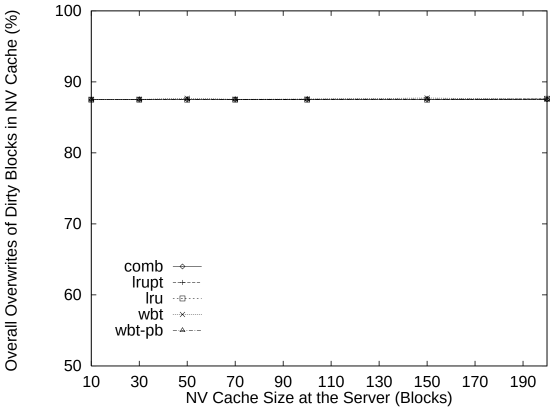

(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

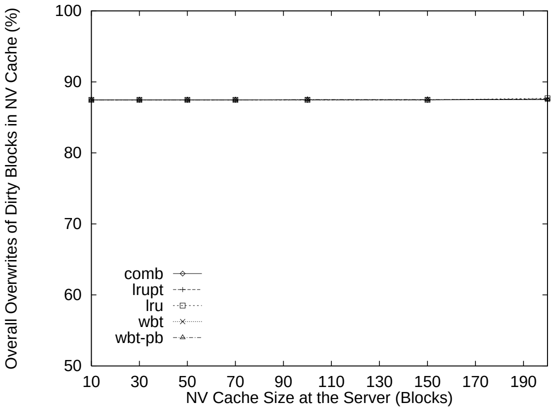
(c) Policy on Client: WBT

(d) Policy on Client: COMB

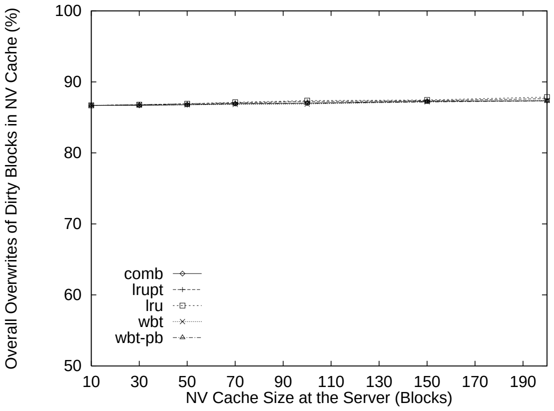
Figure C.32: Blocks Written to Disk, as a Percentage of Total Write Operations. Client NV Cache Size is 100 Blocks



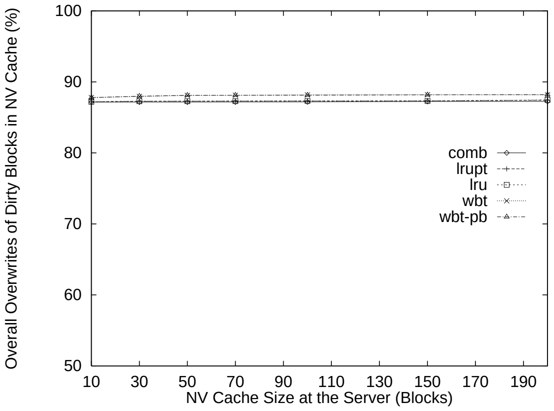

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

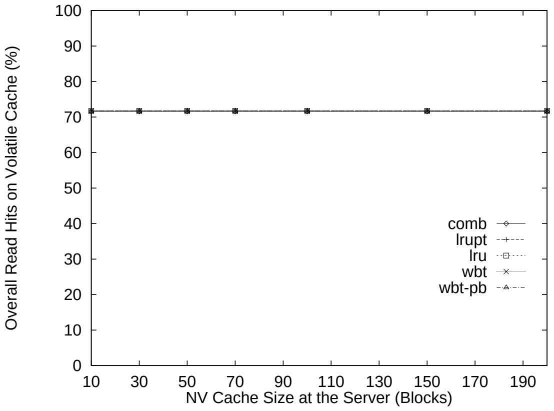



(c) Policy on Client: WBT

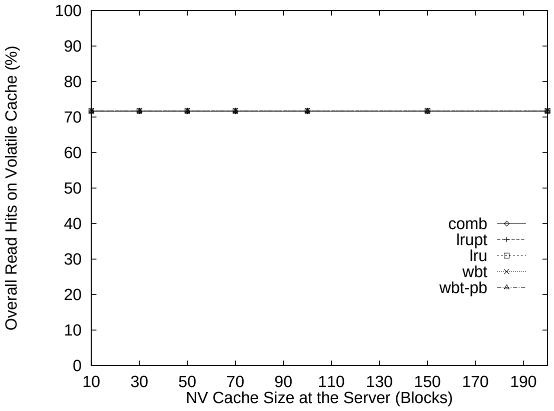



(d) Policy on Client: COMB

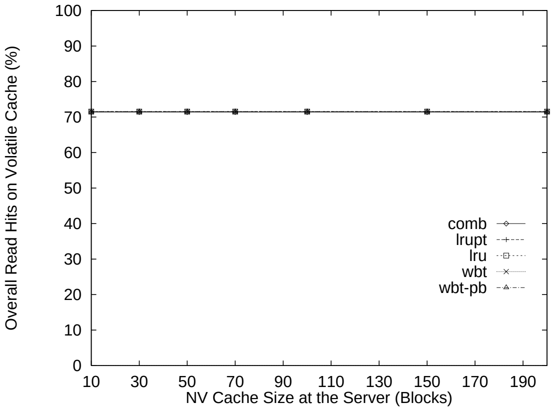
Figure C.33: Overwrites of Dirty Blocks in NV Caches, as a Percentage of the Total Write Operations. Client NV Cache Size is 100 Blocks



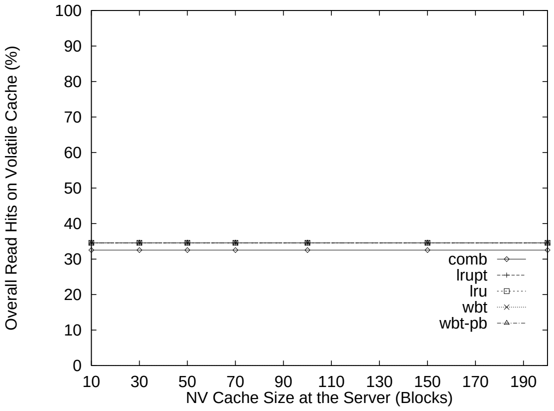

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

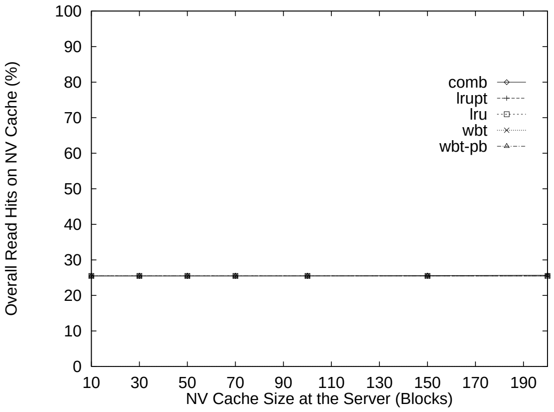



(c) Policy on Client: WBT

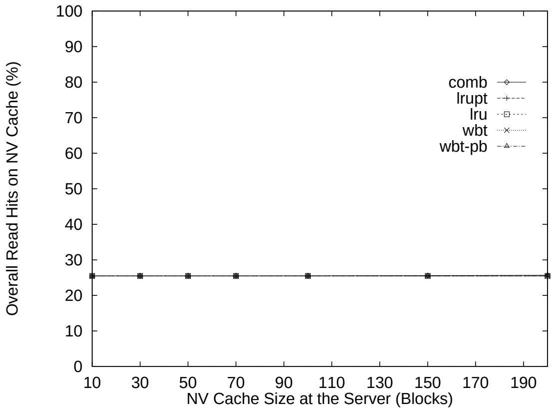



(d) Policy on Client: COMB

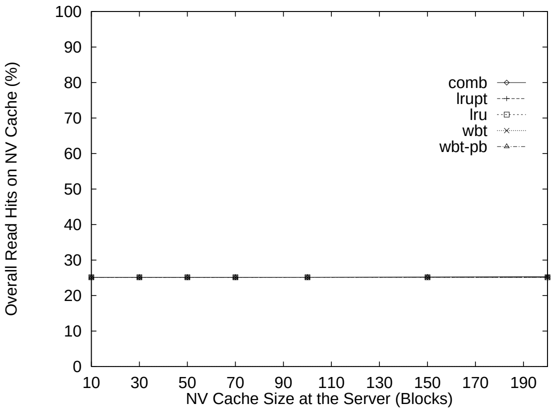
Figure C.34: Overall Read Hit Ratio on Volatile Caches. Client NV Cache Size is 100 Blocks



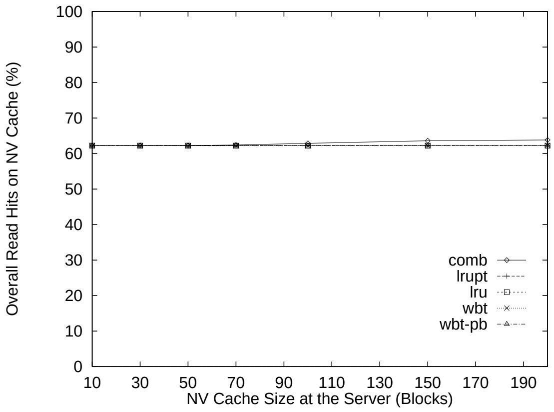

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

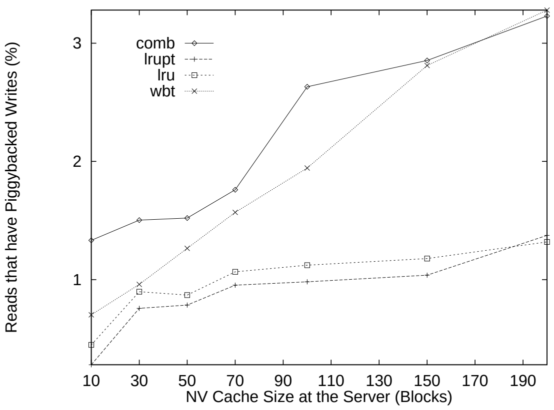



(c) Policy on Client: WBT

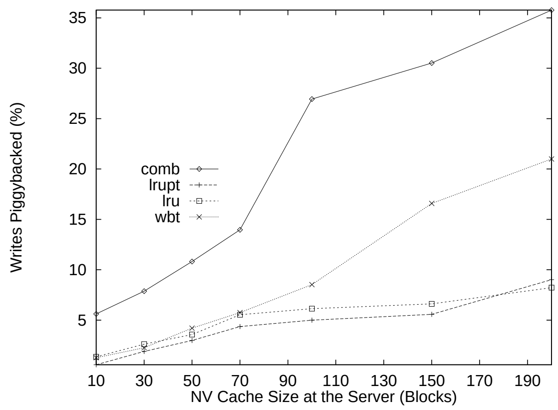



(d) Policy on Client: COMB

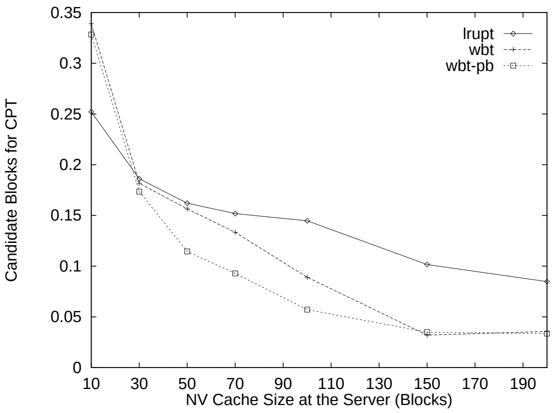
Figure C.35: Overall Read Hit Ratio on NV Caches. Client NV Cache Size is 100 Blocks



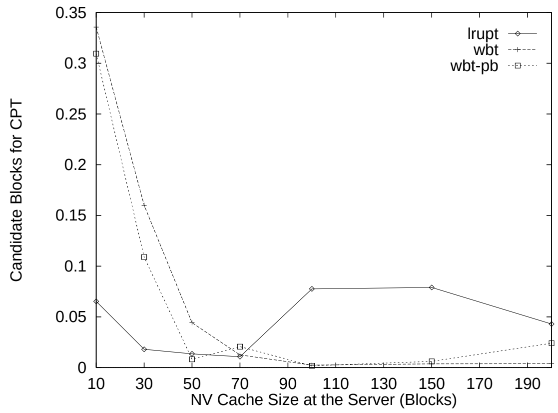

(a) Reads Misses that have Writes Piggybacked




(b) Writes Piggybacked

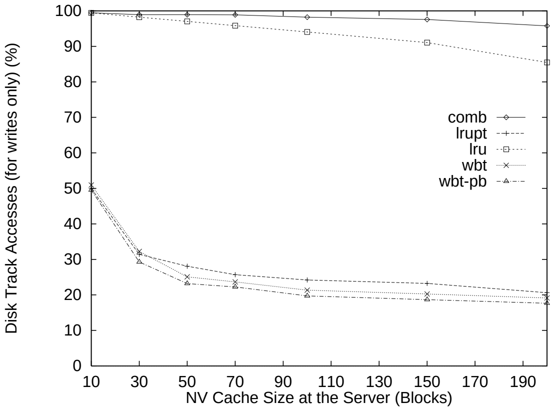



(c) Candidate Blocks for CPT, Client Policy is LRUPT

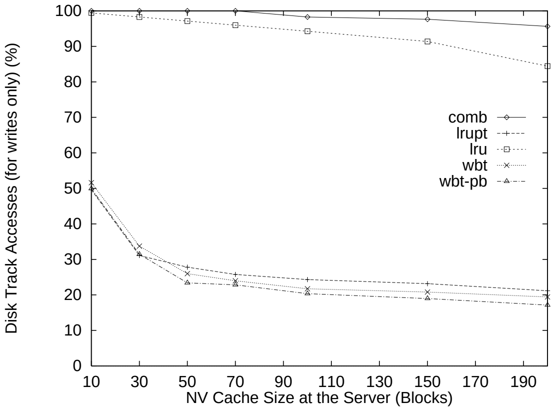



(d) Candidate for CPT, Client Policy is WBT

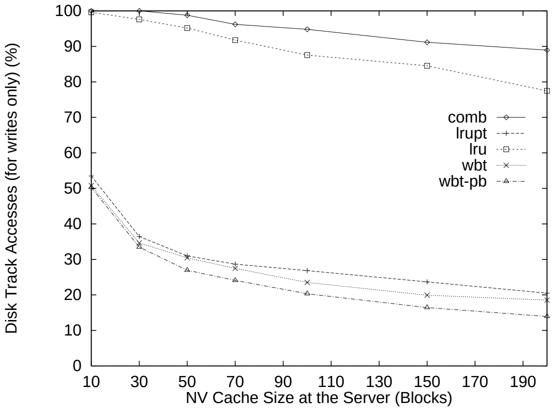
Figure C.36: Other Statistics. Client NV Cache Size is 100 Blocks



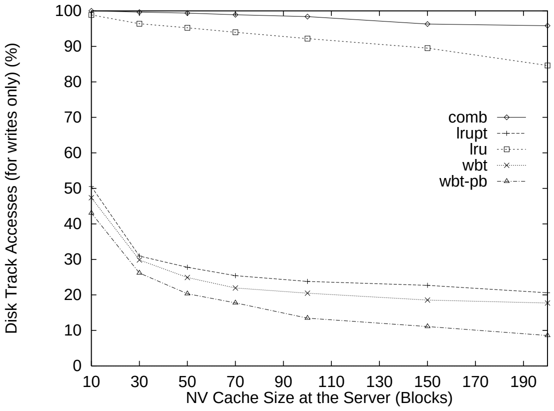

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

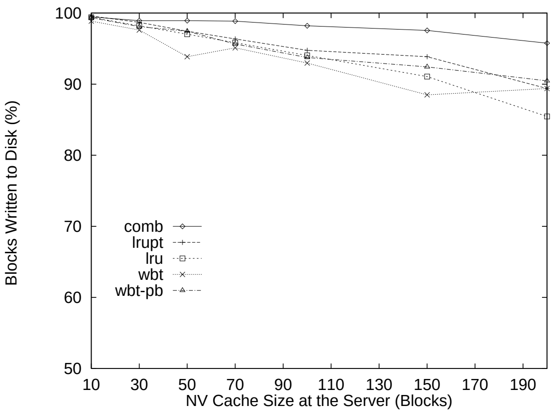



(c) Policy on Client: WBT

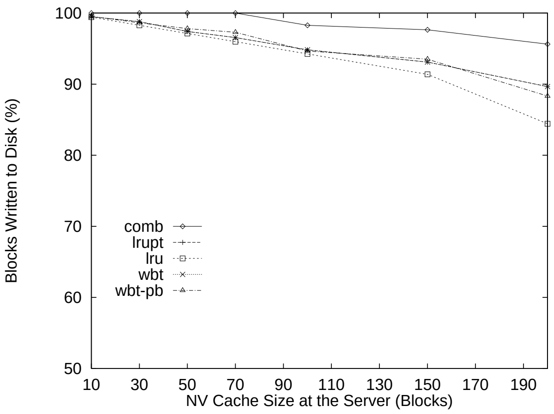



(d) Policy on Client: COMB

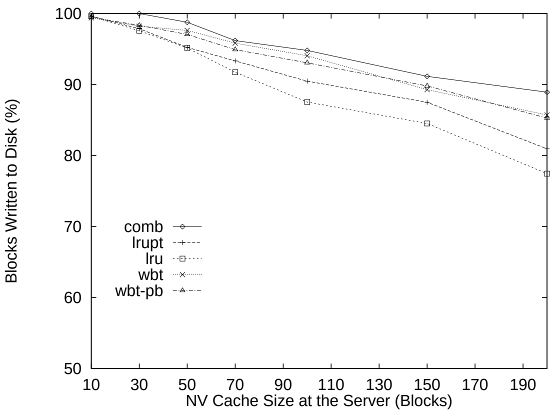
Figure C.37: Disk Track Accesses by Write Operations, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache Size is 100 Blocks



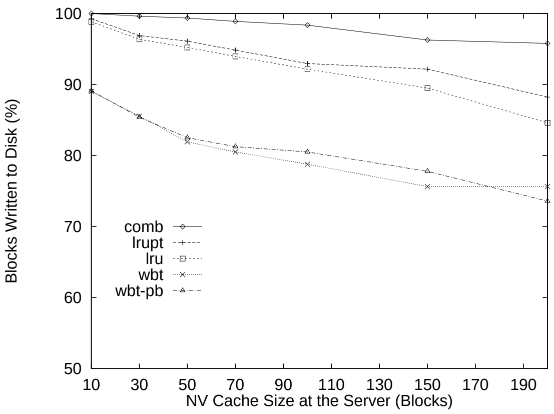

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

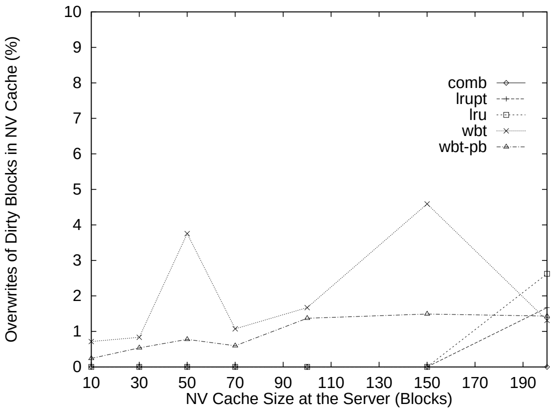



(c) Policy on Client: WBT

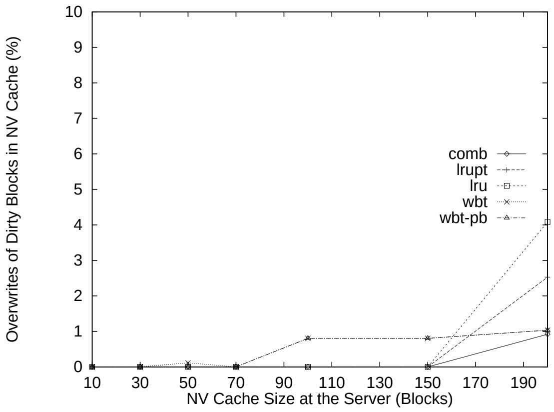



(d) Policy on Client: COMB

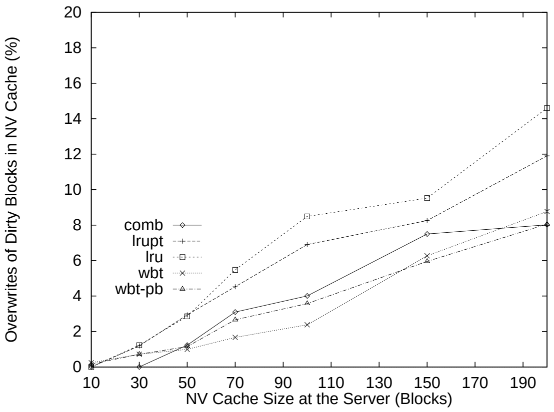
Figure C.38: Blocks Written to Disk, as a Percentage of Total Write Operations Seen at the Server. Client NV Cache Size is 100 Blocks



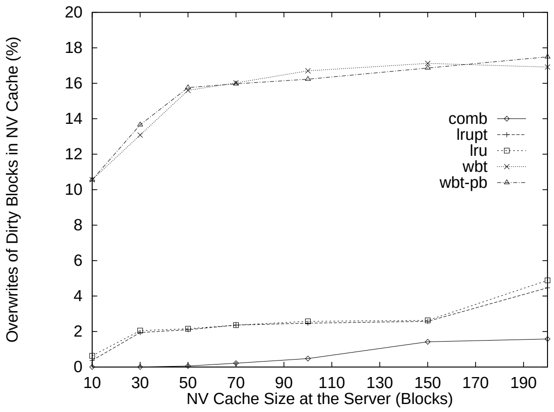

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

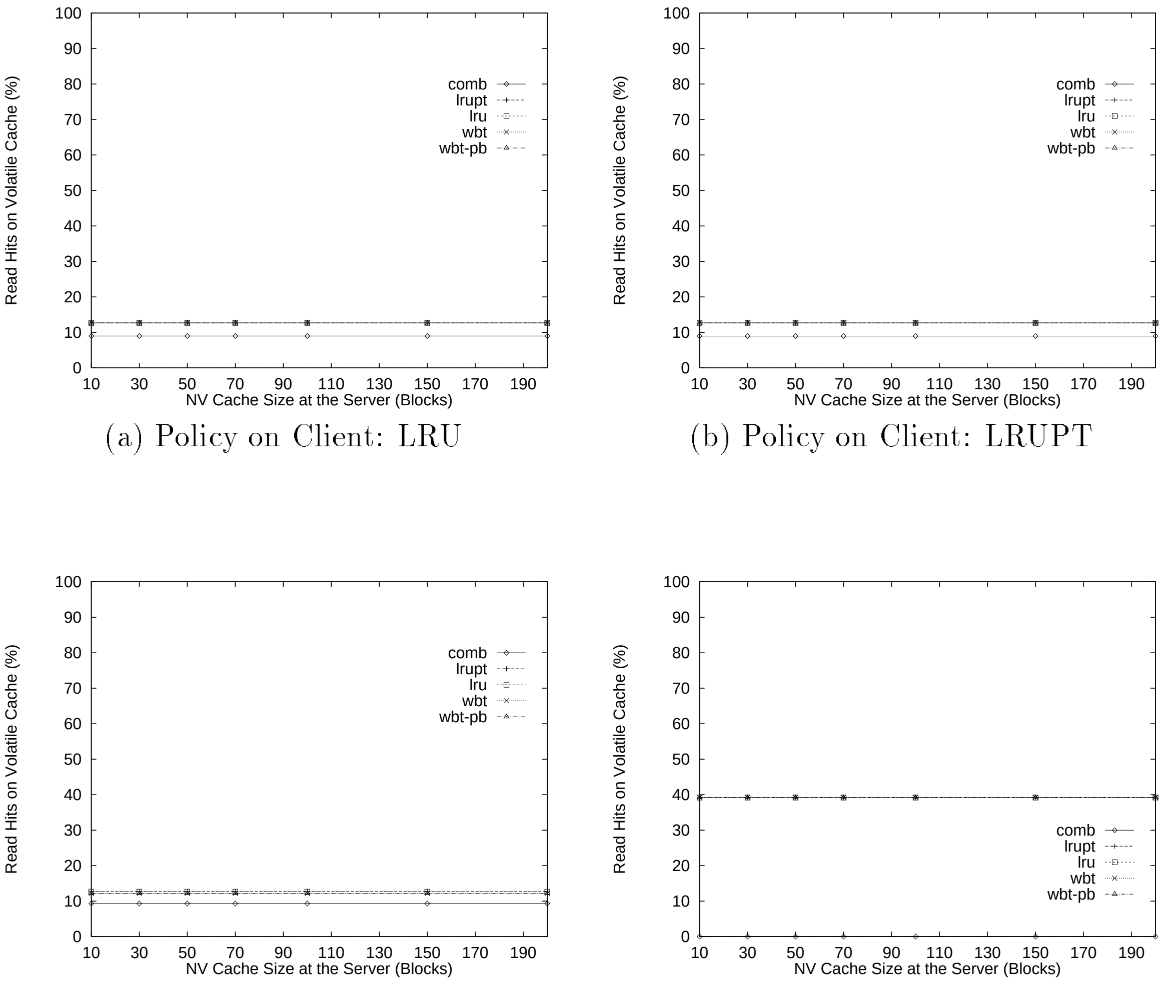



(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure C.39: Overwrites of Dirty Blocks in Server NV Cache, as a Percentage of the Total Write Operations Seen at the Server. Client NV Cache Size is 100 Blocks



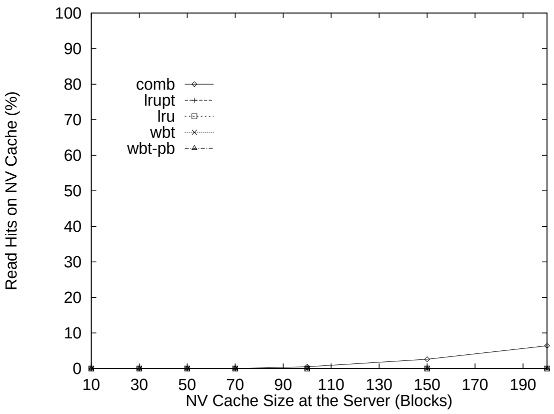

(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

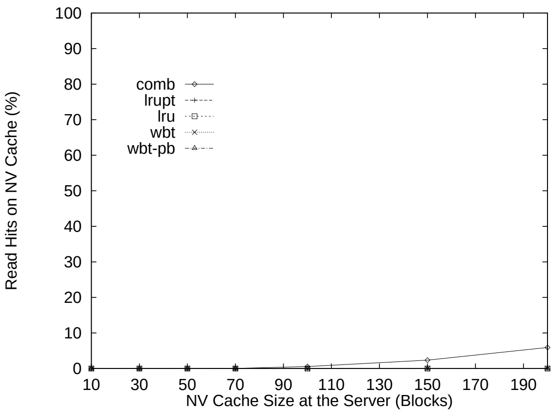
(c) Policy on Client: WBT

(d) Policy on Client: COMB

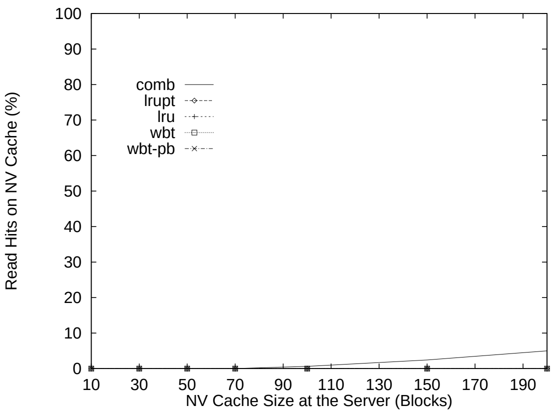
Figure C.40: Overall Read Hit Ratio on Server Volatile Cache. Client NV Cache Size is 100 Blocks



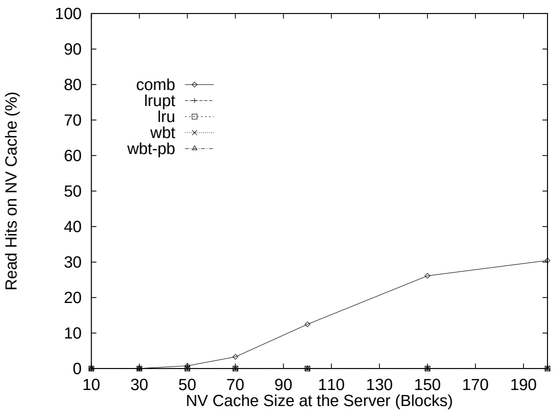

(a) Policy on Client: LRU




(b) Policy on Client: LRUPT




(c) Policy on Client: WBT




(d) Policy on Client: COMB

Figure C.41: Overall Read Hit Ratio on Server NV Cache. Client NV Cache Size is 100 Blocks