# Write Caching in Distributed File Systems

Kerhong Chen, Richard B. Bunt and Derek L. Eager
Department of Computational Science
University of Saskatchewan
Saskatoon, Canada S7N 0W

October 4, 1994

Disk caches are employed in distributed file systems to avoid network accesses at clients and to compensate for the speed differential between main memory and disk at file servers. Because of concerns about volatility, however, write requests have typically not benefitted from the presence of caches. Instead, they have been processed with some sort of write-through or periodic write-back approach to ensure the integrity of the stored data. The introduction of reasonably priced non-volatile (NV) memories has prompted interest in the use of such memory for write caching, at the server and/or at the client. This paper describes an investigation of several approaches to write caching in distributed systems with both volatile and non-volatile caches through trace-driven simulation experiments. The results support the findings of earlier work that suggests important differences between caching in the traditional single-level caching environment and caching in a two-level caching environment. While policies focusing on temporal locality perform well for a single-level caching system, or at the client of a two-level caching system, they may not be suitable for use at the server in a two-level caching system. This is because locality characteristics in the reference stream seen at the server in a two-level caching system may be destroyed by caching at the client with a NV write cache large enough to hold the client's working set of dirty blocks. Policies focusing on amortizing the cost of a disk seek operation over multiple write-back operations perform better at the server of a two-level caching system.

## 1 Introduction

Caches are known to be effective in reducing the performance impact of the considerable gap between the speed of main memory and the speed of disk access, by reducing the frequency of disk accesses [Smith 85]. In a client-server environment disk caches can be placed both at the client, to avoid network access to the file server, and at the file server, to avoid physical disk accesses. Such a two-level cache environment has been the subject of a number of recent investigations [Makaroff and Eager 90, Willick *et al.* 93]. These studies identified some important differences between the two-level cache environment and the more familiar single-level cache environment, differences that speak to the suitability of various approaches to cache management. These studies focused only on read performance, however, as typically only read operations benefit from caching. The write policies used in traditional file system caches most commonly use write-through or periodic write-back approaches to ensure the integrity of the stored data, and thus writes benefit little from caching.

As the cost of non-volatile random access memory (NVRAM) becomes more reasonable, it becomes feasible to use NVRAM to cache dirty data, so that data integrity can be guaranteed without the cost of frequently transferring dirty data from the cache to disk. The use of NV write caches in client-server environments and policies for managing them have been studied by several authors. Policies that have been studied include LRU replacement on servers [Moran *et al.* 90], LRU replacement on clients [Baker *et al.* 92], LRU write-back on servers [Biswas *et al.* 93b], periodic write-back on both servers and clients [Biswas *et al.* 93b], threshold-based write-back on both servers and clients [Biswas *et al.* 93a, Biswas *et al.* 93b], and piggybacking on servers [Biswas *et al.* 93a].

This paper considers systems with both client and server NV caches (as well as volatile caches), and studies the performance characteristics of various combinations of distint client and server NV caching policies, using trace-driven simulation. Results are also presented for various policies in single-level caching systems (stand-alone environments) for comparative purposes. The results suggest that policies focusing on temporal locality perform well in a single-level caching system, or at the client in a two-level caching system, while policies focusing on amortizing the cost of a disk seek operation over many dirty block writes perform well at the server of a two-level caching system. In a two-level caching system, locality characteristics seen at the server in the reference stream may be destroyed by write caching at the client with a NV write 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.

The remainder of this paper is organized as follows. A description of the experimental methodology is given first, followed by a discussion of issues of interest for write caching in a system with both volatile and non-volatile caches, a description of the caching models studied, and a description of the performance metrics used in this evaluation. Then, the results of the trace-driven simulation experiments that were carried out to study the performance effects of write caching in both a single-level caching system (a stand-alone workstation with a local disk) and a two-level caching system (a single client workstation connected to a file server over a network, with a disk connected to the server) are presented and discussed. Finally, conclusions are drawn from the results of the experiments.

# 2 Methodology

## 2.1 Simulation Model

Both a single-level caching system (a stand-alone workstation with a local disk) and a two-level caching system (a distributed system with one file server and one diskless client) were simulated. The single-level caching system has a disk which is accessed through both a volatile cache for read requests and a non-volatile cache for write requests; the two-level caching system (shown in Figure 1) has non-volatile caches for read requests and volatile caches for write requests at both the client and the server. Read misses are placed only in volatile cache while write misses are placed in non-volatile cache. Since some data is both read and written there may be some movement between the two caches, but all service of write requests goes through the non-volatile cache.

The input to the simulations is in the form of workload traces that are collected from the execution of representative applications. The traces were captured by a kernel-based event tracing package and consist of time-ordered (and time-stamped) sequences of file system "events" (reads, writes, removes, etc.). In the single-level caching simulator, each traced

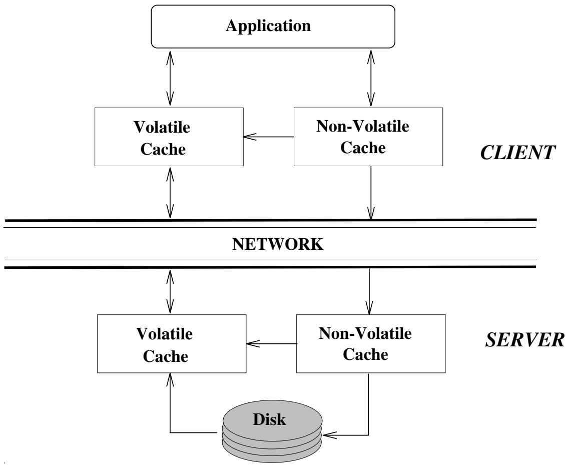



Figure 1: The Distributed System Simulated

event initiates the simulation of the operations that the event would cause at the volatile and the non-volatile caches. In the two-level caching simulator, the operations that the event would cause at the client caches are simulated first, followed by the operations (if any) that would be caused at the server caches and the server disk. The model is very similar to the one used in [Willick it et al. 93].

The performance measures of interest focus on the amount of input/output activity associated with the management of the caches and with the hit/miss behaviour of the caches. The time associated with network operations, disk operations, and queueing was not simulated. Multi-cache consistency is assumed to be accomplished by some consistency policy, but the policy is not explicitly simulated, nor is the overhead of a consistency protocol. This assumption should not significantly affect the performance results, since there are few concurrent write activities. Finally, the block size in all caches is assumed to be 8 kilobytes, a common choice for UNIX-like systems [Levy and Silberschatz 90].

## 2.2 The Caching Policies

The caching policies simulated represent several fundamental approaches to the problem of managing the non-volatile write cache. Locality-based policies make replacement decisions based on the anticipated likelihood of subsequent re-reference to blocks already held in the cache. Well-known policies such as LRU (least recently used) work well to the extent that locality is present in the stream of cache references, and **LRU** was selected as a representative of such policies. Blocks are written out to disk one at a time, in LRU order, whenever it is necessary to make way for an incoming block. Purging-based policies, on the other hand, schedule write-backs in groups of related blocks whenever a prespecified cache occupancy high-limit threshold is exceeded, and continue to write out dirty blocks until a prespecified low-limit threshold is reached. Individual policies differ in their definition of "related blocks": for example, related blocks might be those belonging to the same file or those stored on the same disk track or cylinder (as suggested in [Biswas *et al.* 93a]. For these experiments, a policy referred to as **WBT** (write-back with thresholds) was chosen as a representative purging-based policy. In WBT a block purge is scheduled whenever the cache occupancy exceeds a given high-limit threshold. The choice of blocks to purge is different depending on whether the cache is at the client or the server. In the single-level caching environment, blocks belonging to the disk track[1] with the most dirty blocks are purged first. (It is assumed that all these blocks can be written to disk in a single disk "access", i.e. seek, and that this is substantially less costly than writing out a sequence of spatially unrelated blocks). In the two-level caching environment, dirty blocks in the server cache are replaced on the same basis, while dirty blocks in the client cache are replaced on a file basis (since disk addresses are unavailable); those blocks belonging to the file with the most dirty blocks are chosen first. We also experimented with an intermediate policy, **LRUPT** (LRU purge with thresholds), that combines these two approaches. In LRUPT, cached blocks are maintained in LRU order and are purged according to this order. In the single-level caching environment, all the dirty blocks on the same disk track as the least-recently-referenced block are purged when purging is scheduled; in the two-level caching environment, purging is done on the same basis at the server cache, while file membership is used as the basis for purging at the client cache – the dirty blocks belonging to the same file as the least-recently-referenced block are selected for purging.

In more detail, the three policies operate in the following way:

- *LRU:* Blocks in the NV cache are maintained in LRU order. When a write request is received, the cache is searched for the requested block. If the block is in the cache (a hit), then the block is moved to the MRU end of the LRU list. If the block is not in the cache (a miss), then the block is inserted into the cache if there are empty blocks. If the cache is full, the LRU replacement policy is applied 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 cache in the two-level caching system) and removed from the cache. Free and LRU lists of cache blocks are maintained for each NV cache managed by this policy.

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

- *WBT:* 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 two-level caching system, those blocks belonging to the disk track with the most dirty blocks are purged first, while for a client NV cache in the two-level caching system, those blocks belonging to the file with the most dirty blocks are purged first.

In all experiments, the volatile caches are managed by the LRU replacement policy. This decision was made since LRU is currently the standard policy for such caches and to reduce the complexity of the experiment. Only read requests are served through the volatile caches.

[1] This could equally be done on a cylinder basis

## 2.3 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 two-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 two-level caching system) can be absorbed by the NV cache. The following set of performance metrics was used to evaluate the effectiveness of the proposed approaches.

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

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

2. 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 is the purging policy.

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

3. Dirty blocks that are overwritten while still in a NV cache, expressed as a percentage of the total number of writes. The more dirty blocks overwritten in a NV cache before being flushed, the more effective is the write-back policy.

$$Overwrites\ in\ NV\ Cache\ (\%) = \frac{Write\ Hits\ to\ Dirty\ Blocks}{Total\ Number\ of\ Write\ Operations} \times 100\%$$

## 2.4 Trace Selection

A set of file system event traces had been previously collected by Willick *et al.* [Willick *et al.* 93] from the Distributed Systems Performance Laboratory in the Department of Computational Science at the University of Saskatchewan. Applications running on a client workstation form the basis for the tracing. The trace point is placed prior to the client's cache, primarily to record the file I/O traffic into the client's cache. A wide spectrum of activities is represented in this collection of traces, ranging 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). We believe this to represent typical file access activity within a university research environment. In the interests of conserving space while, at the same time, presenting comparable results, results from only a single trace are presented in this paper. Results from a wider set of traces can be found in [Chen 94]. The trace selected came from tracing a set of simulation experiments used to generate the results in [Willick *et al.* 93]. This particular trace was chosen because it involved a good deal of file activity (738,489 read events and 8,547 write events over a span of 3.23 hours). Each time-stamped trace record contains a description of the file I/O activities generated by the routines traced, along with the process identifier, the offset of the logical block from which the requested data begins, the block size, a read/write flag for the block, the I/O device identifier, the file identifier, and the file system identifier.

For this research, in addition to the information already recorded in the traces, it is also necessary to know the physical disk address for each block referenced in order to optimize the purge operations. At the server, modern disk devices may hide the true physical track and cylinder numbers from the operating system, but we assume that the address available to the operating system can be used to estimate track membership. At the client, where we do our tracing, no disk address information is available at all. For this reason, a secondary 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 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 (the block's track/cylinder) 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). For a block with logical block address *Logical Block Address* and the disk track size *Disk Track Size* in Kbytes, the physical disk track number *Physical Disk Track Num* for the block is computed as:

$$Physical\ Disk\ Track\ Num\ = \lfloor \frac{(8K \times Logical\ Block\ Address)}{Disk\ Track\ Size} \rfloor .$$

In the experiments reported here, it is assumed that a track can hold 5 blocks.

### 2.5 Experimental Design

The most important factor in this experiment is the write policy for the NV cache. Three levels for the caching policy factor on each of the client and the server were considered: LRU, LRUPT, and WBT. The size of the volatile cache in the single-level caching system and on the client and the server of the two-level caching system was constant (200 blocks – a figure chosen based on the results in [Willick *et al.* 93]).

The size of the NV cache is also a factor in the experiment. 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 two-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 to be at most the same as the volatile cache size, since it may not be realistic to have a NV cache larger than the volatile cache.

In all the simulation runs reported in this paper, the thresholds for purging were 95% for the high limit and 65% for the low limit. These particular values were chosen on the basis of the results of some preliminary experiments.

# 3 Results and Discussion

## 3.1 Write Caching in the Single-level NV Cache

Since the primary motivation for introducing a disk cache is to reduce the frequency of disk accesses, an important measure of performance is the number of disk track accesses required under each policy. Track-based purging policies should perform well under this measure since only one physical disk access is necessary when blocks on the same disk track are written to disk together. Figure 2 shows the number of disk tracks accessed to perform the required write operations plotted as a function of NV cache size, for each of the three caching policies considered. The number of disk track accesses is given as a percentage of the total number of write operations in the trace and is expressed in logarithmic form. All three policies considered show a sharp reduction in track accesses as the NV cache size increases, with the two purging policies (WBT and LRUPT) outperforming LRU. LRUPT requires the fewest track accesses for its write operations, when the NV cache size is greater than or equal to 50 blocks. LRUPT is able to take advantage of temporal locality in the reference stream when the NV cache size is large enough to hold the entire working set of dirty blocks, but LRUPT is also able to amortize the cost of disk access over many dirty block writes by purging all the dirty blocks on the same disk track as the least-recently-referenced block on the LRU stack. WBT yields more disk track accesses than LRUPT (but fewer than LRU) when the NV cache size is greater than 40 blocks and smaller than 100 blocks. Since WBT is designed to amortize the cost of disk access over multiple writes as much as possible, the presence of temporal locality in the reference stream has little effect. The "hybrid" nature of LRUPT may explain why it outperforms WBT for an intermediate range of NV cache sizes. When the NV cache is large (greater than 150 blocks), however, more dirty blocks are available for purging, and the performance of WBT begins to improve relative to LRUPT.

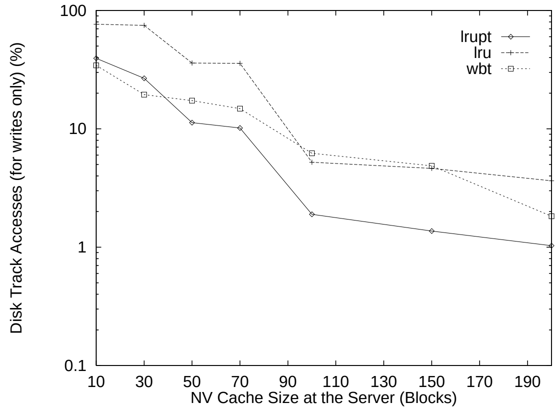



Figure 2: Disk Track Accesses to Process Write Operations (Single-level NV Cache Configuration)

Figure 3 shows the percentage of dirty blocks that are actually written to disk from the NV cache. The two locality-based policies, LRU and LRUPT, show almost identical behaviour, and result in fewer actual writes than WBT, which attempts by design to do more physical writes on each disk access. Under all three policies the number of physical writes is reduced from more than 70% of total write operations down to less than 10% as the NV cache size increases from 10 to 200 blocks. When the NV cache size is larger than 100 blocks, the performance of LRU and LRUPT does not change much, but WBT 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 (in relative terms) in Figure 2 compared to Figure 3.

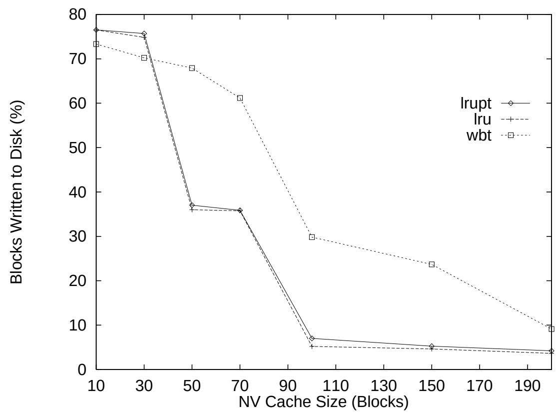



Figure 3: Blocks Written to Disk (Single-level NV Cache Configuration)

One of the benefits of delaying the write-back of a dirty cache block is that, should a further write be made to it before it is written back, the need for the first write-back is eliminated altogether. Figure 4 shows the percentage of dirty blocks overwritten while in the NV cache (i.e., blocks that are written to again before they are written out to disk). As the cache size increases, the percentage of blocks overwritten while in the cache increases sharply, exceeding 90% in many cases. LRUPT and LRU yield the highest percentage of overwrites, when the NV cache size is greater than 30 blocks. Again, LRU performs slightly better than LRUPT.

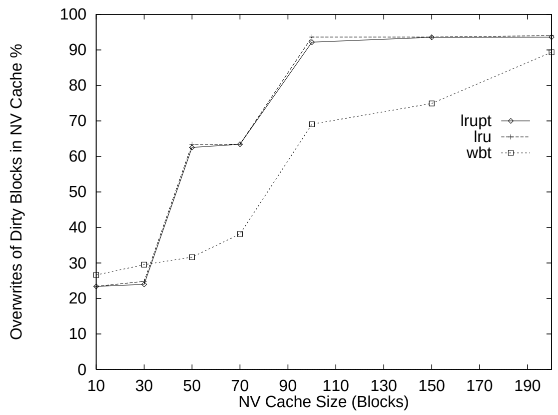



Figure 4: Overwrites of Dirty Blocks (Single-level NV Cache Configuration)

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. The number of dirty blocks deleted from the NV cache before being written out to disk is insignificant since there is only a very small number of remove operations recorded in the trace used for this experiment.

The results of this experiment suggest that write caching can be effective, and that the hybrid policy LRUPT is a good choice when there is only one level of write caching. Some conclusions about the behaviour of the three write caching policies considered 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 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.

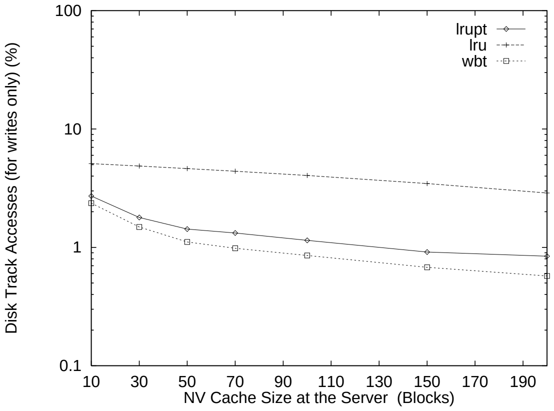
- LRU
  LRU is less inclined to write dirty blocks back to disk than is LRUPT, since LRU does not purge, and therefore, will leave dirty blocks in the cache as long as possible, fully utilizing the entire NV cache. In practice, however, LRU appears to behave quite similarly to LRUPT in terms of the number 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.

- WBT
  WBT writes back to disk, in each purging operation, all the dirty blocks that reside on the disk track that has the most dirty blocks. Since recently-referenced blocks may be included in the purge group, temporal locality may not be respected, Thus, even though WBT does better at amortizing the cost of a track access over many block writes, it appears that WBT performs slightly worse than LRUPT in this single-level NV cache context, even in terms of the number of disk accesses. (It performs much worse in terms of the number of block writes.)

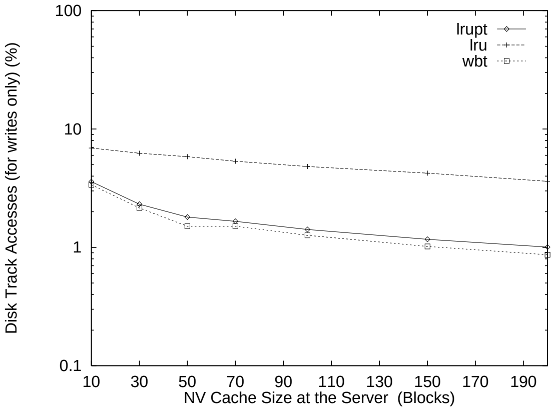
### 3.2 Write Caching in the Two-level NV Cache

The performance of write caching policies on the client should be identical to the performance of these policies with single-level caching, since the behaviour of caching policies on the client is not be affected by the presence of a cache at the server. The performance of caching at the server, however, is affected by caching done at the client. In this way, the presence of a cache at *both* client and server adds a significant new dimension to the problem of cache management. In this section, simulation results are presented for the case in which non-volatile write caches are present on both the client and the server, under various combinations of caching policies.

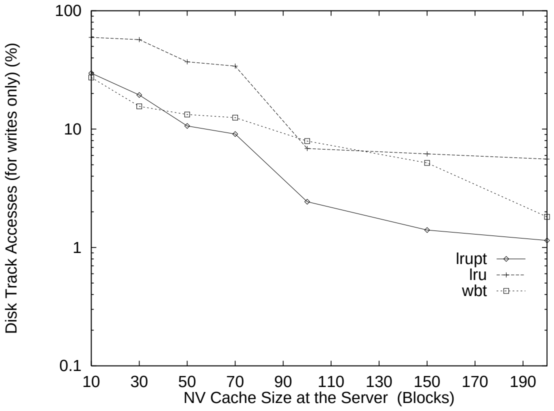
Figure 5 shows the number of disk track accesses performed for write operations for each combination of caching policies at the client and the server. LRU appears to be the worst choice for a server cache management policy, regardless of the policy employed at the client, and WBT appears to be a poor choice at the client. With a large NV cache on the client (for example, 100 blocks), the entire working set of dirty blocks can be held in the client cache, and therefore not much temporal locality remains in the write reference stream by the time it reaches the server. A consequence is that LRU and LRUPT do not perform as well as WBT on the server, when LRU or LRUPT is used on the client. This "filtering" phenomenon of client caching was observed previously in the context of two-level read caching [Willick *et al.* 93] and these results suggest that it also an issue in two-level write caching. The combination of LRU on the client and WBT on the server appears to provide good overall performance.




(a) Policy on Client: LRU



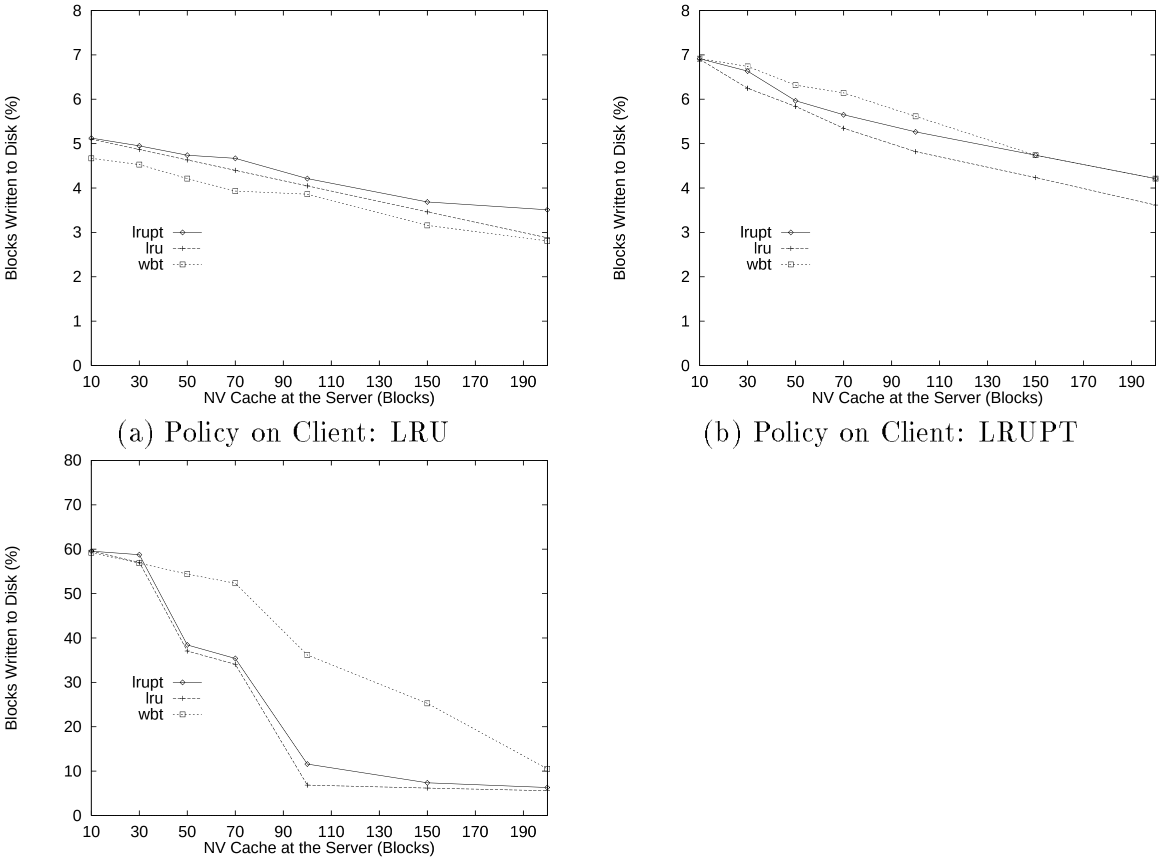

(b) Policy on Client: LRUPT




(c) Policy on Client: WBT

Figure 5: Disk Track Accesses by Write Operations (Client Write Cache is 100 Blocks)

The number of blocks written back to disk for each combination of client and server NV caching policies is shown in Figure 6, expressed as a percentage of the number of write operations. The combination of LRU on the client and WBT on the server yields the fewest write backs. When LRU or LRUPT is used at the client, fewer than 7% of write operations actually result in blocks being written back to disk, while more than 60% of write operations result in disk writes when WBT is used at the client. This reult suggests that the choice of policy for the client cache plays a more important role in overall performance than the choice of policy for the server cache. Comparing these results to those of Figure 3 also gives striking evidence of the "filtering" phenomenon in two-level write caching. In the two-level caching system, WBT outperforms LRU and LRUPT at the server, while LRU and LRUPT perform much better than WBT in a single-level caching system. In other words, locality-focused policies (such as LRU and LRUPT) perform well at the client, which sees temporal locality characteristics in the reference stream, but they perform poorly at the server, where the temporal locality characteristics in the reference stream have been filtered out by client caching.

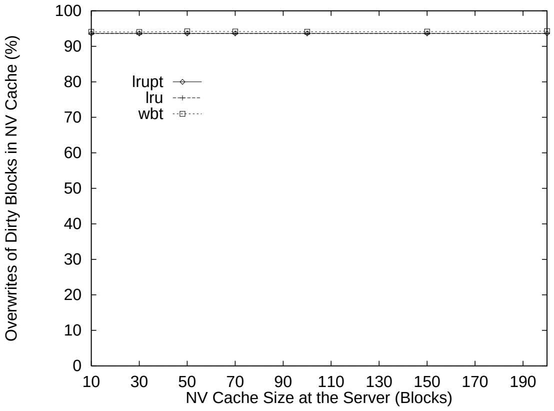



(c) Policy on Client: WBT

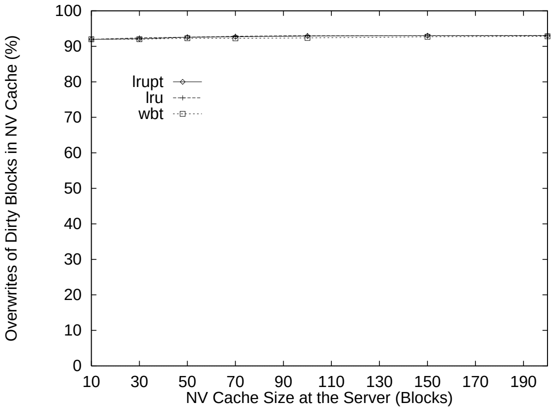
Figure 6: Blocks Written to Disk (Client Write Cache is 100 Blocks)

The use of NV write caching allows the possibility that dirty blocks can be overwritten within the NV cache before being written back to the disk. Figure 7 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 percentage of overwrites, regardless of the policy used for the server cache. Evidently, at 100 blocks, the client's write cache is sufficiently large to contain the working set of dirty blocks, and since it is managed according to the principles of locality, it does a good job of maintaining in the cache those blocks that are likely to be overwritten. Only when the server's write cache is similarly large (90 blocks or more), does the combination of LRU or LRUPT at the server and WBT at the client yield similar performance.

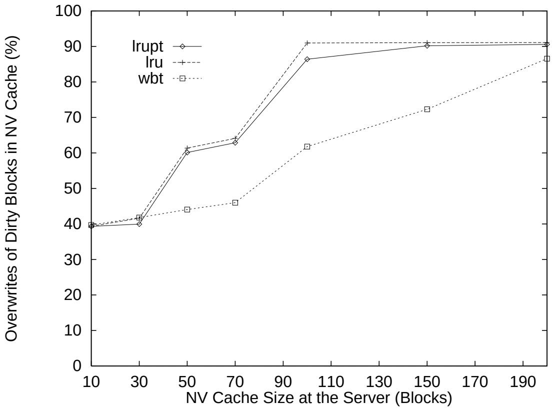
Once again, the percentage of blocks written to disk should equal 100% less the sum of the percentage of dirty blocks overwritten and the percentage of dirty blocks deleted. As was the case in the single-level cache experiment, the number of dirty blocks deleted before being written to disk was not significant.




(a) Policy on Client: LRU




(b) Policy on Client: LRUPT




(c) Policy on Client: WBT

Figure 7: Overwrites of Dirty Blocks in NV Caches (Client Write Cache is 100 Blocks)

Some conclusions about the behaviour of the policies considered for managing NV write caches in a two-level caching environment are as follows:

- Client caching can destroy the temporal locality seen in disk block reference streams at the server. Therefore, when LRUPT or LRU is used on the client, and the write cache on the client is large enough to hold the client's working set of dirty blocks, a locality-based policy such as LRU or LRUPT may not perform well at the server write cache. If a policy that is less effective at caching the working set of dirty blocks (such as WBT) is used on the client, LRUPT can perform well on the server, achieving a low number of disk track accesses, a low number of blocks written to disk, and a high number of overwrites of dirty blocks in the write cache.

- LRU might be the best policy for use at the client, but it is not a suitable policy for use at the server.

- Since it does not take advantage of temporal locality, WBT does not perform as well as LRU or LRUPT on the client when the write cache is large enough to hold the working set of dirty blocks.

- When the write cache on both the client and the server is large, the combination of LRU on the client (to take advantage of the temporal locality in the reference stream at the client) and WBT on the server (to amortize I/O costs), appears to be the best combination.

# 4 Other Approaches

Several other approaches to writing back cached data are possible, and are considered in [Chen 94]. Two of these are reviewed briefly in this section.

## 4.1 Piggybacking

Since read requests that miss at both the client cache and the server cache necessitate immediate disk access, it makes sense to "take out" any writes that might be in the same area of the disk at the same time. By "piggybacking" write requests in this way, write traffic might be reduced when many files are opened for both reading and writing.

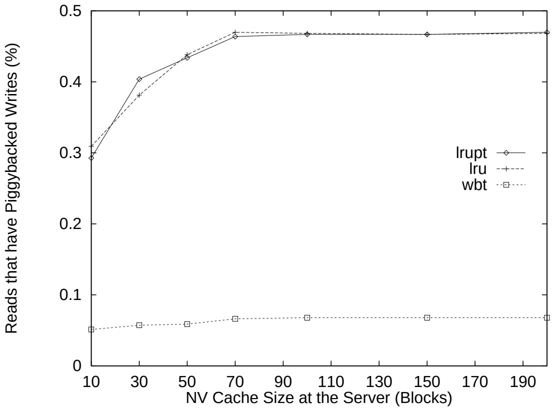
Two forms of piggybacking were investigated as a modification to WBT, referred to as **WBT-PB**. When the physical disk addresses of the referenced blocks are known, write requests to blocks on the same track as the missed read request can all go out in the same disk access operation, thereby eliminating the need for separate disk operations to write back these dirty blocks at a later time. This "track-based piggybacking" can be done in a single-level caching system or at the server in a two-level caching system. For a diskless client in a two-level caching systems, however, track-based piggybacking is not possible since the physical addresses of referenced blocks are not known. For this reason, files are used as the basis for piggybacking on clients. In "file-based piggybacking", write requests to the same file as the missed read request can be sent to the server's write cache together.

Some results relating to piggybacking performance when WBT-PB is used at the server NV cache are shown in Figure 8. Figure 8 (a) shows the percentage of write-back operations that were done using piggybacking, and Figure 8 (b) shows the percentage of read misses that resulted in piggybacked writes. The results obtained suggest that piggybacking is unlikely to yield any significant performance improvement for this particular workload trace. On all the other performance measures considered in this investigation, the performance of WBT-PB was virtually identical to that of WBT.

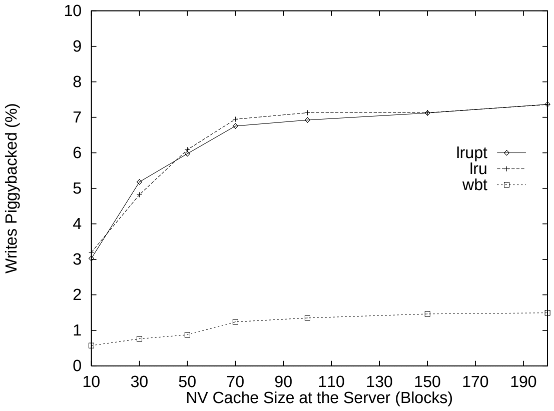
## 4.2 Coordinated Purging

A new policy was introduced for the two-level caching system only. The CPT policy (Coordinated Purging with Thresholds) tries to coordinate the purge operations on the server and client. When a purge of dirty blocks begins at the server (triggered by the cache occupancy exceeding its high-limit threshold) the server determines what files the dirty blocks being purged belong to and asks the client to purge dirty blocks belonging to the identified files as well. The server then accepts as many dirty blocks into its NV cache as possible (i.e. until its high-limit threshold is exceeded again), at which time the server starts the next purge operation. In other words, the server tries to advise the client on the choice of blocks to purge. When purging is needed only at the client the client is free to choose any block according to the purging policy being used.

To determine whether or not coordinating the purging operations on the server and client should be beneficial, the number of blocks belonging to the same disk track as the



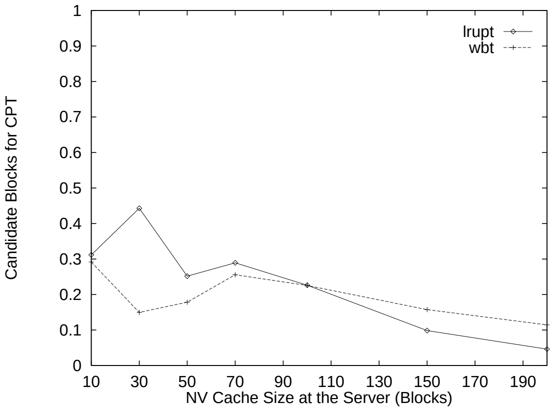

(a) Writes Piggybacked



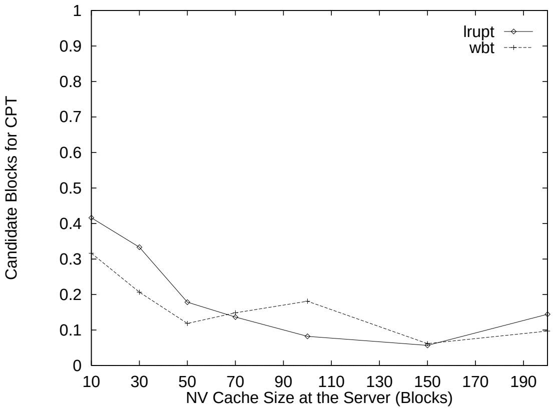

(b) Read Misses that have Writes Piggybacked

Figure 8: Piggybacking Performance (Client Write Cache is 10 Blocks)

purging base on the server (averaged over all the blocks that are written back to disk by the server) is of interest. If many of these blocks are also present in the client cache, then it might be reasonable to have purge operations on the client coordinated with the purge operations on the server. Figure 9 shows the number of blocks that would be CPT candidates, when the client NV cache size is 100 blocks. For each block written back to disk from the server, there appears to be few blocks in the client's NV cache eligible to be candidates for coordinated purging, regardless of the client caching policy and regardless of the size of the NV cache at the server. Again, these results are based on one particular workload trace, but they suggest that coordinated purging is of limited benefit.




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




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

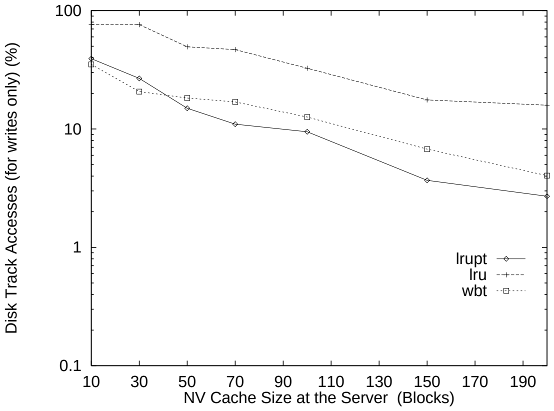
Figure 9: Candidates for Coordinated Purging (Client Write Cache is 100 Blocks)

# 5 The Effect of Cache Size

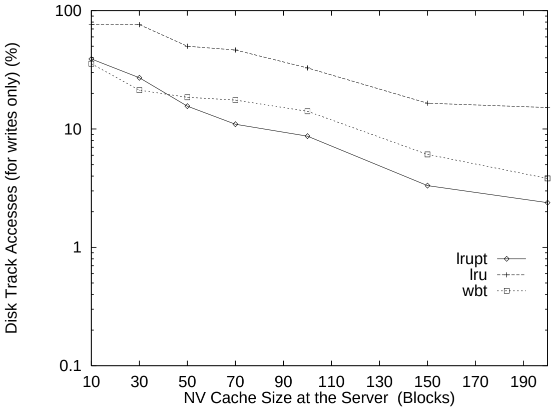
The size of any cache is an important performance determinant. This section examines the influence of NV cache size on both overall two-level NV write caching performance and policy performance.

## 5.1 The Influence of Cache Size on Choice of Policy Combination

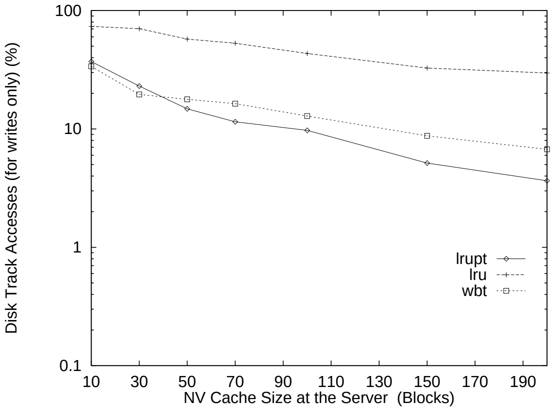
When the NV write cache on the client is small, the entire working set of dirty blocks cannot be kept in it and thus temporal locality is not completely destroyed in the reference stream seen by the server. As a result, the performance with policy combinations using LRU or LRUPT on the server should not be as bad as that when the NV cache on the client is large. As can be seen by comparing (a) and (b) in Figures 10, 11, and 12 to (a) and (b) in Figures 5, 6, and 7. LRUPT does better than WBT as a server cache management policy when the client cache is small (10 blocks), but WBT does better when the client cache is large (100 blocks).




(a) Policy on Client: LRU

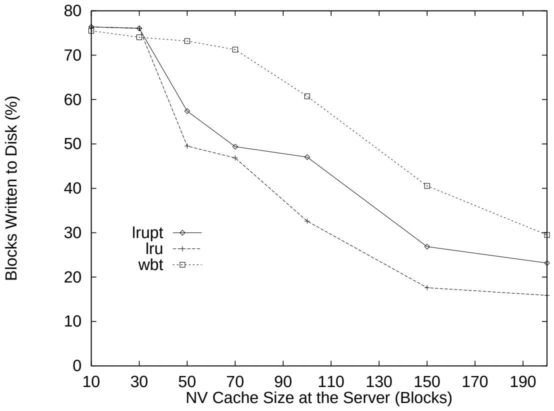



(b) Policy on Client: LRUPT

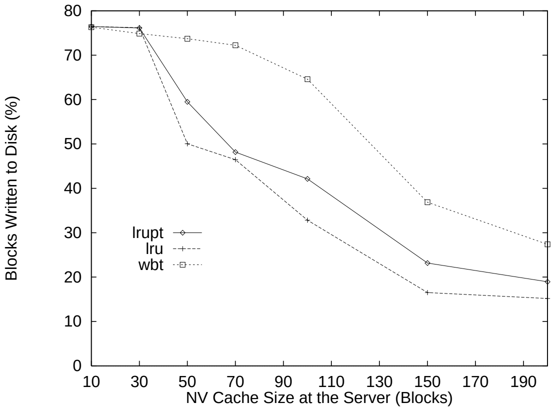



(c) Policy on Client: WBT

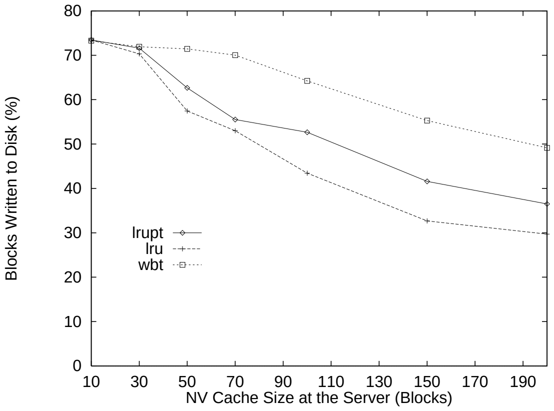
Figure 10: Disk Track Accesses by Write Operations (Client Write Cache is 10 Blocks)




(a) Policy on Client: LRU




(b) Policy on Client: LRUPT



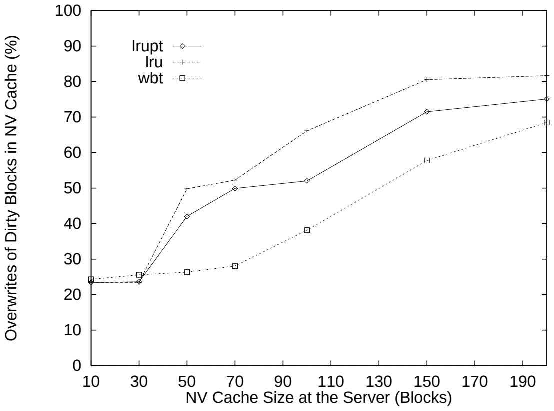

(c) Policy on Client: WBT

Figure 11: Blocks Written to Disk (Client Write Cache is 10 Blocks)

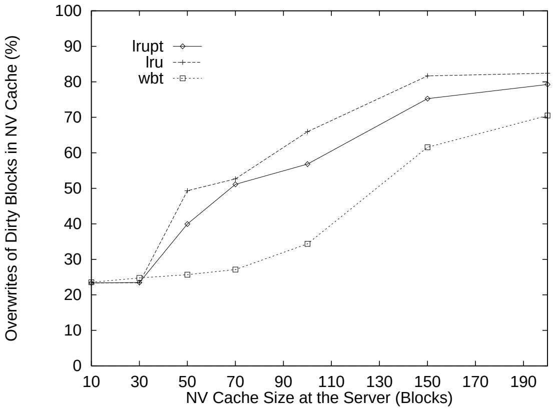
## 5.2 The Effect of Cache Size on Performance

A write cache is placed at the client to reduce the network traffic generated by the need to write dirty blocks to the server; thus it can also reduce the workload at the server. Better performance is achieved since writing to the client cache is much faster than writing back to the server or disk with overheads of network access and disk operations. This is of particular importance when applications are doing synchronous writes. A write cache is placed at the server to save slow disk accesses, i.e., to amortize the cost of disk accesses over many dirty block writes. A large write cache at the client should result in better performance, in terms of reduced network traffic, faster response time, and reduced server load, while a large write cache at the server should result in better performance in terms of fewer disk accesses. The effect of cache size on the performance of write caching is discussed in this section.

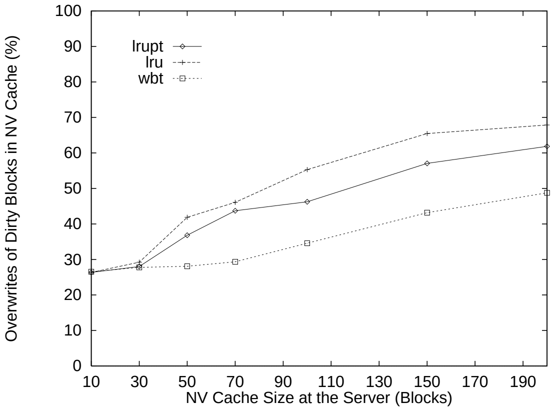
In an earlier study, Makaroff and Eager [Makaroff and Eager 90] observed that the read miss ratio at the server cache does not drop significantly with increasing server cache size until the size of the server cache size approaches the size of the client cache. The results of this research suggest that the same phenomenon holds for write caching as well. Figures 13, 14, and 15 show results for disk track accesses for writes, blocks written to disk, and overwrites of dirty blocks in the server's write cache, when the client's write cache is 50




(a) Policy on Client: LRU




(b) Policy on Client: LRUPT



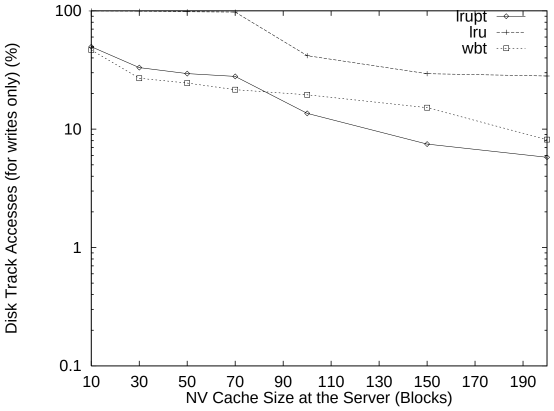

(c) Policy on Client: WBT

Figure 12: Overwrites of Dirty Blocks in NV Caches (Client Write Cache is 10 Blocks)

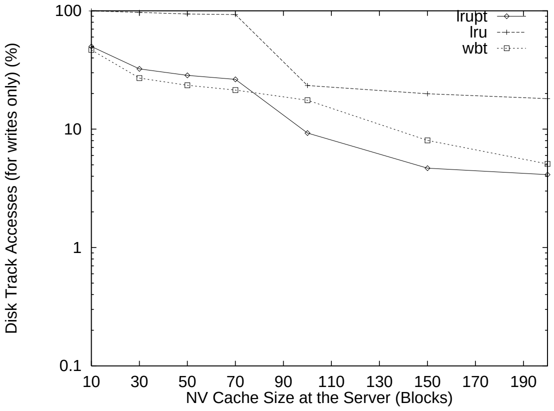
blocks.

When locality-based policies (LRU or LRUPT) are used at both client and server caches, there is little performance improvement in terms of dirty blocks written back to disk and overwrites of dirty blocks in the server's write cache. when the write cache at the server is smaller than the write cache at the client (50 blocks), as is shown in Figures 14 and 15. 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 write cache, for LRU or LRUPT at the server, when the server's write cache becomes significantly larger than the client's write cache (when the server's write cache exceeds 70 blocks). More of the frequently accessed dirty disk blocks are retained in the client's write cache when it is large, so the server's write cache sees fewer repeated references to these blocks. The dirty blocks that are in the server's write cache are placed there by the initial client misses, but repeated references to these blocks at the client do not keep them at the top of the LRU stack in the server's write cache. Only when the write cache at the server is larger than the write cache at the client can the blocks referenced less frequently be held in the server's write cache.

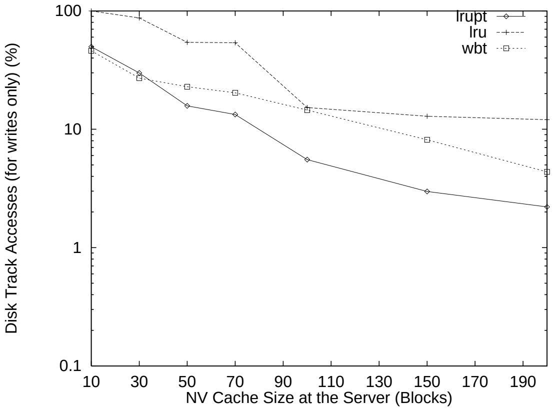
Unlike locality-based policies, when purging-based policies (such as WBT) are used at the server, increasing the server's write cache size even by a small amount (e.g., from 10 to 30




(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

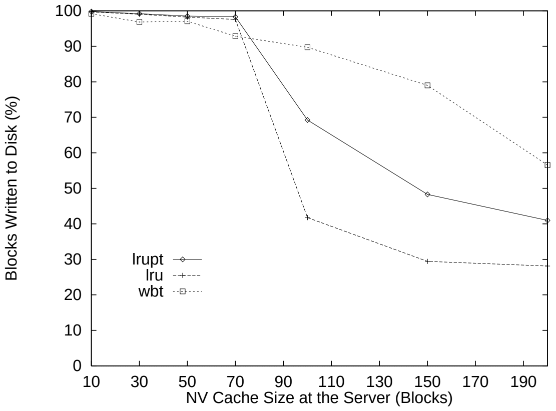



(c) Policy on Client: WBT

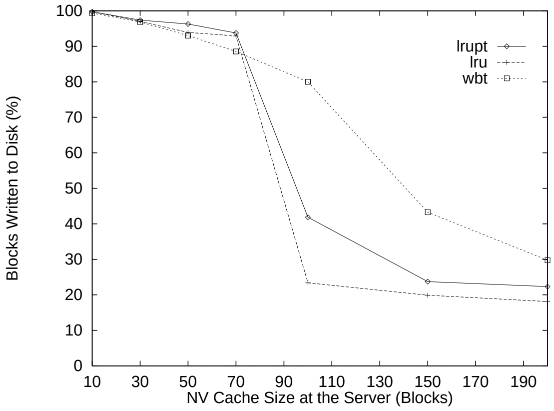
Figure 13: Disk Track Accesses by Write Operations (Client NV Cache is 50 Blocks)

blocks) improves performance in terms of disk track accesses, when there is a moderate sized write cache at the client (e.g., 50 blocks), as is shown in Figure 13. There is no dramatic change when purging-based policies are used at the server, in terms of dirty blocks written back to disk and overwrites of dirty blocks in the server's write cache, when the server's write cache size exceeds the client's write cache size. Temporal locality characteristics in the reference stream appear to be less important to the performance of WBT.

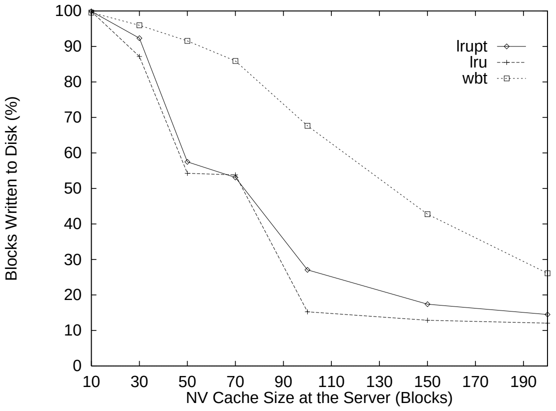
When there is a moderate-sized write cache at the server (e.g., 50 blocks), adding a small write cache at the client (e.g., 10 blocks) does not improve the overall performance in terms of disk track accesses. In fact, it makes the performance even worse than with no client write caching at all. This can be seen by comparing Figure 10 with Figure 2. Increasing the client's write cache size (from 10 to 100 blocks) improves the performance of policy combinations in terms of disk track accesses (as shown in Figures 10 and 5). For a 100 block write cache at the client and a 50 block write cache at the server, and for the policy combination of LRU at the client and WBT at the server, performance in terms of disk track accesses is better than having a 150 block write cache at the server and no write caching at the client. With an appropriate hybrid policy, however, a single-level write cache with 150 blocks may perform better than a caching system that partitions the 150 blocks of write cache into 100 blocks at the client and 50 blocks at the server. For example, one




(a) Policy on Client: LRU




(b) Policy on Client: LRUPT

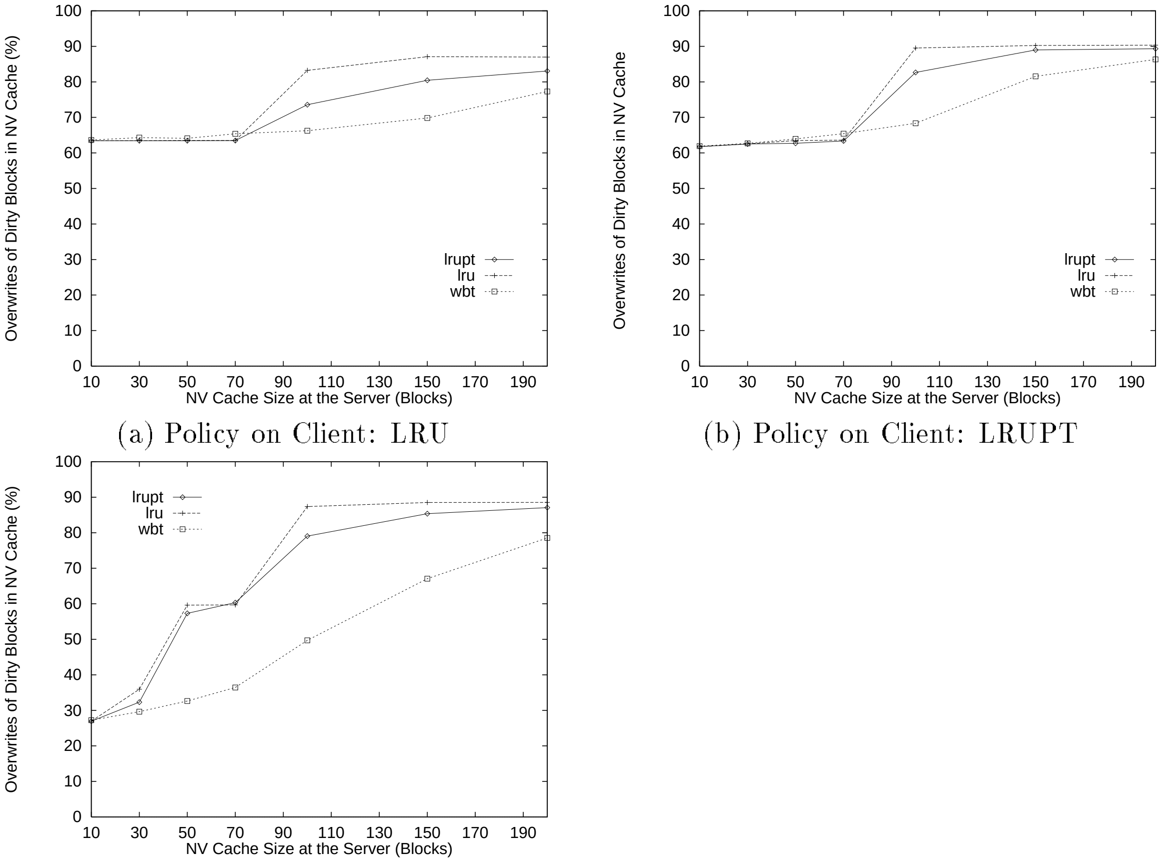



(c) Policy on Client: WBT

Figure 14: Blocks Written to Disk (Client Write Cache is 50 Blocks)

might attempt to capture the benefits of both LRU and WBT, by partitioning the cache into an 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 write cache at the client.

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




(a) Policy on Client: LRU

(b) Policy on Client: LRUPT

(c) Policy on Client: WBT

Figure 15: Overwrites of Dirty Blocks in Server NV Cache (Client NV Cache is 50 Blocks)

and the server load can be reduced. At the same time, the small write cache at the server can still be used to amortize I/O costs.

# 6 Summary and Conclusions

The emergence of affordable non-volatile memory technologies has prompted system designers to consider using them to provide write caches in a distributed client-server file system. This paper reports on an investigation of policies for the management of non-volatile write caches in a client-server environment. The central conclusion is that write caching at both the client and the file server can have a significant impact on overall performance. The major results of this study can be divided into two categories: the performance of different policies to manage a non-volatile write cache in a single-level cache environment, and the performance of different policy combinations in a two-level cache environment. Two classes of write caching policy were studied: locality-based policies that focus on exploiting temporal locality in the request stream to make write-back decisions, and purging policies that focus on amortizing the cost of disk access over multiple dirty block write-backs. Results are also presented concerning the effect of the size of the write cache on performance and whether or not a policy to coordinate purging between server and client caches might be reasonable.

The results suggest that locality-based write caching policies perform well in a single-level write caching environment, or at the client cache in a two-level write caching client-server environment. In a two-level write caching environment with a client write cache large enough to hold the client's working set of dirty blocks, the combination of a locality-based policy at the client and a purging-based policy at the server is a better choice. At the server, temporal locality is at least partly destroyed by client caching and so locality-based policies are less effective.

If the write cache on the client is large enough to hold the client's entire working set of dirty blocks, the temporal locality in the reference stream may be eliminated almost entirely. This does not happen when the non-volatile cache at the client is small. For locality-based policies at the server, performance in terms of disk block writes does not improve much as the server cache size is increased until the size of the server cache size approaches the effective size of the client cache. Purging-based policies at the server, however, benefit from increasing the server cache size even for smaller sizes, because of greater numbers of blocks that can be found sharing the same tracks.

With the technology in place to provide affordable and safe write caches, and policies to manage them effectively in a client-server environment, it appears that the benefits of disk caching that were once limited to read requests can now be extended to write requests as well.

## References

[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.

[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.

[Chen 94] Chen, K., *Write Caching in Distributed File Systems*, M.Sc. thesis, Department of Computational Science, University of Saskatchewan, August 1994.

[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 and Eager 90] Makaroff, D.J. and Eager, D.L., "Disk Cache Performance for Distributed Systems", *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, France, May 1990, pp.212-219.

[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.

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

[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.