Appears in SIGMETRICS Performance Evaluation Review, volume 25, number 3, (pp 29-36).

Prefetching Over a Network: Early Experience With CTIP

David Rochberg Garth Gibson

Computer Science Department
Carnegie Mellon University Pittsburgh PA 15213
{david.rochberg,garth.gibson}@cs.cmu.edu

1 Abstract

We discuss CTIP, an implementation of a network filesystem extension of the successful TIP informed prefetching and cache management system. Using a modified version of TIP in NFS client machines (and unmodified NFS servers), CTIP takes advantage of application-supplied hints that disclose the application's future read accesses. CTIP uses these hints to aggressively prefetch file data from an NFS file server and to make better local cache replacement decisions. This prefetching hides disk latency and exposes storage parallelism. Preliminary measurements that show CTIP can reduce execution time by a ratio comparable to that obtained with local TIP over a suite of I/O-intensive hinting applications. (For four disks, the reductions in execution time range from 17% to 69%). If local TIP execution requires that data first be loaded from remote storage into a local scratch area, then CTIP execution is significantly faster than the aggregate time for loading the data and executing. Additionally, our measurements show that the benefit of CTIP for hinting applications improves in the face of competition from other clients for server resources. We conclude with an analysis of the remaining problems with using unmodified NFS servers.

2 Introduction

Our motivation is a familiar one: processor speeds are increasing faster than peak disk bandwidths, which in turn are improving much faster than disk drive seek times. Storage parallelism [6] can provide more storage bandwidth, and aggressive prefetching can hide latency [2][4] but still, the fraction of time that applications spend waiting for disk drive activity continues to increase. CMU's informed prefetching and caching system (TIP [7]) exploits knowledge of application workloads to perform aggressive prefetching. In TIP, applications disclose their future I/O accesses as *hints*, and the TIP system uses this information both to manage the system's file cache more effectively and to prefetch data from disk into the cache before the application requests the data. When the storage subsystem has hints for many future disk requests, it can make better use of I/O parallelism. Consider a single-threaded application that makes a series of nonsequential reads one at a time in a large file striped over several disks: without knowledge of the upcoming reads, the storage system cannot start other disks seeking for the targets of future reads while it waits for the current read in the sequence to complete. Through disclosure, TIP can take advantage of such parallelism. Figure 1 shows that TIP is able to reduce application stall times even with small numbers of drives; with 10 disks, application runtime is reduced by 20% - 83% over a suite of I/O-intensive benchmarks.

To date, TIP has been confined to direct-attached storage. This is a natural domain for computation requiring high-performance I/O, but we would like to get TIP's benefits without having spend time and money making local copies of datasets on dedicated parallel storage arrays within a 25m differential-SCSI-bus-length of our machines. Distributed filesystems free us from these constraints: they allow us to share data, amortize storage costs, and centralize management. Hence, in this work, we explore client-driven remote TIP, or CTIP, a modification to the TIP system that prefetches data from NFS file servers. CTIP applies TIP's cost-benefit model to networked storage by treating the remote storage NFS server simply as a disk with higher latency and increased CPU driver cost.

In the remainder of the paper, we give an overview of the TIP system and discuss how it is modified to create CTIP. We then analyze the performance of CTIP on the TIP benchmark suite, both with and without competition for file server resources, and discuss the limitations of our initial CTIP approach.

3 Extending TIP to the network

3.1 An overview of TIP

In TIP, applications supply *hints* that disclose their future behavior to the cache-management and prefetching

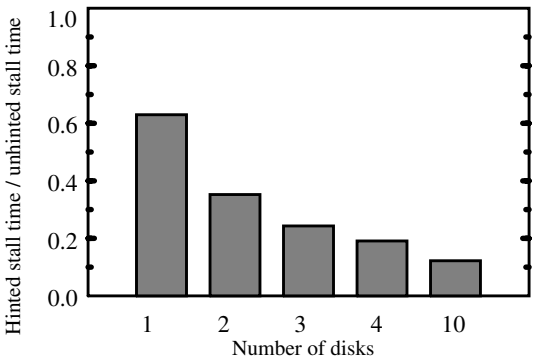

Figure 1. Average stall time for the hinting version of the benchmark suite described in Section 4.2 divided by stall time for the unhinting version of the suite running under the same number of disks. Application hints allow TIP to overlap computation and I/O on multiple disks and to manage its cache better. This results in a substantial reduction in stall time, even for small numbers of disks.

This research is sponsored by DARPA/ITO through DARPA (order D306), under contract N00174-96-0002. Additional support was provided by an NSF graduate fellowship and the member companies of CMU's Parallel Data Consortium (including Hewlett-Packard Laboratories, Symbios Logic Inc., Data General Clariion, Compaq, Quantum Corporation, Seagate Technology, Wind River Systems, and Storage Technology Corporation). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any supporting organization or the U.S. Government.

system. Each application hint specifies a file, either by file descriptor (for files the application has already opened) or by name, and an ordered list of offsets within the file. Applications deliver hints (through a special system call) for data in the order that they expect to read the data. Applications are free to read blocks that they have not hinted (and almost all applications do so) without negating the benefit of the hints they give. However, the benefits of TIP depend on applications issuing all the reads they have hinted; the current implementation makes little effort to improve the performance of applications that issue incorrect hints. When TIP receives no hints or inaccurate hints, it falls back on traditional cache-management and prefetching techniques. In these cases, our implementation uses a least-recently-used (LRU) cache replacement scheme and Digital Unix's aggressive sequential readahead policy (which prefetches up to 64KB ahead for sequential reads of files on local disks).

Given information about the state of the system and good hints describing the future I/O plans of running applications, TIP attempts to allocate buffers in a way that minimizes the I/O stall time, or the time spent servicing I/O requests and waiting for I/O completion. To do this, it maintains a set of estimators for the benefit of buffer allocation actions, like caching a block that is the target of a future hint or devoting a block to the prefetcher. The estimators compute the cost or benefit of their corresponding buffer allocation decision in a common currency: change in application I/O stall time per buffer access. For tractability, the estimators base their calculations on a simple model of disk and application behavior. This model assumes that there are "enough" disks arms available, so that TIP does not need to model queue depth at the disk drives and it pessimistically assumes that applications may not spend any time computing between adjacent requests. Three factors potentially contribute to the stall time for a given request: T_{disk} is the latency for actually retrieving a block from disk once the command to read the block has been sent. T_{driver} is the amount of CPU time required to issue and receive a disk request, and the amount of time to service a hit in the cache is T_{hit}. A complete description of the estimators and their implementation is beyond the scope of this paper (a detailed exposition is in [7]), so we give an intuitive overview here.

The four cache actions that TIP can choose from, and their associated estimators are:

- **Allocating a buffer for prefetching:** Just as for a regular read, TIP must allocate a buffer to hold the results of prefetching a hinted block before it issues the prefetching read to storage. Since applications hint accesses in the order that the corresponding reads will occur, it makes sense to talk about an application's (read) position in its hint stream. The prefetching estimator, then, estimates the benefit of allocating an additional buffer so that the TIP system can prefetch one buffer further ahead of the application's current position. The benefit of adding prefetching buffers starts out high, because if a hinted block is not prefetched in time for its consumption, the application will stall for more than T_{disk}. As the prefetching depth increases, the benefit of increasing it further diminishes quickly. Finally, since we are modeling disk accesses as constant-time operations (where that constant is T_{disk}), and it takes at least T_{hit} to read a block, a prefetched block will always arrive in time if it is prefetched T_{disk}/T_{hit} accesses before it is needed. We call this quantity the *prefetching horizon* and assume that there is no benefit to prefetching beyond it.
- **Allocating cache blocks for demand reads:** An unhinted read or a hinted read whose data has not yet been prefetched is a *demand* read. Because the application will stall indefinitely unless it receives resources to

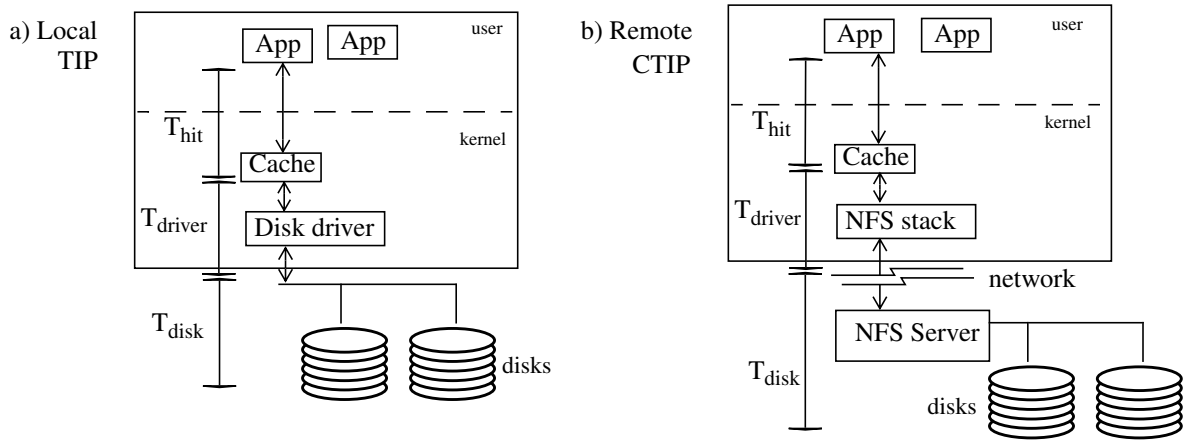

Figure 2. The parameters for the TIP and CTIP models. Part a) shows the parameters for the local case and part b) the parameters for the remote case. All the time spent in filesystem code or waiting for the disk is stall time. We split that time into three components: T_{hit} is the CPU time to retrieve a block from the cache, T_{driver} measures the CPU time required to issue and receive a one-block read from storage, and T_{disk} measures the latency of retrieving a block from storage.

satisfy the demand read, there is no estimation involved with demand reads—it always makes sense to allocate blocks for them.

- **Shrinking or growing the LRU cache:** TIP maintains a traditional LRU cache to satisfy many unhinted accesses without fetching from storage. The LRU estimator keeps track of what the hit rate in the LRU cache would be for several LRU cache sizes, and uses this information, coupled with the disk-model cost of an additional cache miss (T_{disk} +T_{driver}), to estimate the benefit or cost of making the LRU cache larger or smaller, respectively.
- **Shrinking or growing the hinted cache:** TIP also tracks cache blocks for which future accesses are hinted. If TIP allows such a cached and hinted block to be ejected (because it falls off the tail of the LRU queue), it will later have to stall the application for at least T_{driver} to read the block back into memory. Since the benefit of avoiding this driver work is amortized over all accesses until the block is fetched, the cost of allowing such a block to be ejected decreases from T_{driver} as the number of accesses before the block will be fetched back in increases.

Given these estimations, TIP uses a straightforward greedy algorithm to manage the cache. On every access, it (efficiently) finds the buffer whose ejection has the least cost. If the cost of ejecting this least valuable buffer is less than the benefit of adding to the prefetching depth, or if there is an outstanding demand read, it ejects the buffer.

3.2 CTIP

CTIP is a minimal extension of the TIP model to networked storage; it treats the file server as a disk with longer latency. To do this, we make two modifications to the local TIP system: the first is the addition of a set of routines for prefetching from an NFS server and the second is a new set of estimates for the time required to retrieve data from storage.

The added prefetching routines are similar to those used by local filesystems. Digital's original NFS (version 3) code has a one block sequential readahead routine. When CTIP has hints to act on, we disable this default readahead, but, for comparison, we leave it enabled in our unhinted runs. Since disclosure exposes more parallelism, CTIP tends to issue more concurrent requests than an unhinted NFS system does. To make sure that there are contexts available to handle these requests promptly, we substantially increase the number of NFS I/O daemon processes and threads on both the client (from the suggested 7 to 64) and on the server (from the suggested 16 to 70, enough to handle all the asynchronous threads on the client plus a few additional synchronous requests).

Our measurements for the model parameters are similar to those in [7]. We used a microbenchmark that repeatedly issued a series of nonsequential 8KB reads to a large file. When the series does not fit into the cache, the CPU time per access is T_{driver} + T_{hit}, and the idle time per access is T_{disk} (we ran the tests in single-user mode, so there were no other processes available to use the CPU when the microbenchmark was waiting for the disk) When the series does fit into the cache and the program is run with a warm cache, the CPU time per access is T_{hit} (and there is no significant idle time). Table 3 shows the values for these parameters for the system described in Section 4.1

Parameter	Local (TIP)	Remote (CTIP)
T_{disk}	13.6ms	15ms
T_{driver}	580µs	877µs
T_{hit}	190µs	190µs

Table 3. Storage model parameters for TIP and CTIP. Because of our fast network, T_{disk} for the remote case is surprisingly close to T_{disk} for the local case. CTIP must send its request through an expensive set of protocol stacks (ONC RPC over UDP) while TIP uses a faster SCSI stack. Because of this difference, T_{driver}, the client CPU time cost to retrieve an 8KB block, is substantially higher for CTIP. This increased cost translates into longer run times for remote applications.

4 Experiments

4.1 Experimental setup

To evaluate the performance of the CTIP system, we ran the benchmark suite on a DEC 3000/500 workstation (with an Alpha 21064 CPU running at 150MHz, 128MB of RAM, 12MB of which was dedicated to the buffer cache) running our TIP-modified version of Digital Unix 3.2g. Since TIP shows most of its possible performance benefit with arrays as small as four drives, we ran the benchmark on a filesystem striped over 4 HP 2247 1GB drives with a striping psuedo-device driver. For the CTIP runs, we also used a DEC 3000/600 (with an Alpha 21064 running at 175MHz and a 12MB buffer cache) server running NFSv3. The server exported a local filesystem striped over 4 HP 2247 drives. To connect the machines, we used 155 Mbit/s OC-3 ATM links through a DEC Gigaswitch.

For this configuration, the values for the parameters discussed above (in section 3.1) are in Table 3. While the T_{disk} costs in the two systems are similar, the CTIP spends more CPU time for network storage accesses than TIP pays for local disk accesses.

For all the benchmarks, we stored the application executables in the same filesystem that we ran the benchmarks from, and we ran the benchmarks from cold caches with the client machine in single-user mode. The figures in the graphs are the mean of five trials, and the standard deviation of the run times over those trials was less than 2% of the mean in every case but one (unhinted agrep, where the deviation was still under 1 second).

4.2 Benchmark Suite

We use the benchmark suite from Patterson's TIP paper [7] to measure the macro performance of CTIP. The benchmark suite includes seven I/O-intensive applications that we have modified to be able to disclose hints:

- XDataSlice, a scientific-visualization application that displays arbitrary planar slices taken from a 3-D dataset. Our benchmark takes 25 random slices out of a 512x512x512 cube of 64-bit values. For each slice, XDataSlice hints all of the blocks it will need to display the slice before it starts reading the slice.
- Gnuld, the GNU link editor. Gnuld makes several passes over its object files. Parts of the first (table-building) pass are unhinted, but after that, Gnuld's reads are largely nonsequential and hinted. We time Gnuld as it links approximately 64MB of OSF/1 v2.0 kernel objects, producing an 8.8MB kernel.
- Agrep, a text searching program. We time agrep as it searches for (and does not find) a simple string in 11 MB of OSF/1 v2.0 kernel sources scattered over 1349 files and 2922 disk blocks. Agrep hints all the files as it starts up, and then it reads the files sequentially.
- Sphinx, a speech-recognition program that we have modified to use an out-of-core language model. After a table-building start-up phase, Sphinx makes several passes through the utterance to be recognized. For each 10ms acoustical frame of the utterance, Sphinx hints and then reads as many as 100 short selections from its 200MB language model. Sphinx has an effective internal cache, so these reads result in only a few disk accesses. Our benchmark tests the time it takes for Sphinx to recognize an 18-second utterance.
- Davidson, a computational physics application that sequentially reads the same 16.3MB matrix 63 times. Since the reads are sequential, traditional sequential readahead in the non-TIP case can substantially reduce stall time waiting for disk latency, but the traditional LRU caching strategy means that the entire matrix is read into and ejected from the cache on each iteration. TIP's informed caching keeps much of the matrix in cache, and hence substantially reduces the number of reads (and hence the amount of T_{driver} stall) required.
- Postgres, a relational database executing two joins. Each run joins an outer relation with 20,000 tuples (3.2 MB) and an inner relation with 200,000 tuples (32 MB). The inner relation is indexed by the join field (5MB). The first join benchmark matches 20% of tuples and the second matches 80%. We modified Postgres to split its computation for these joins to allow hint generation and provide better cache locality for the index. This modification improves Postgres performance even in the unhinted case. Postgres first makes a (hinted) sequential read of the outer relation. For each tuple in the outer relation, it looks the tuple up in the index for the inner relation (with unhinted accesses), and stores the address of a matching inner tuple in memory. It then uses this array of tuple addresses to generate hints and read addresses for the matching tuples in the inner relation.

This benchmark suite is diverse; it includes sequential, non-sequential, cache-intensive, and small-file workloads. The average reduction in I/O stall time provided by TIP over these applications is shown in Figure 1.

4.3 Results without server competition

Figure 4 shows the execution times of each benchmark in four cases: hinted and unhinted in the remote filesystem (CTIP) and in the local filesystem (TIP). Each execution time is divided into CPU busy time and idle time. Since no significant tasks were running during the benchmark runs, the idle time closely approximates time spend waiting for storage. Although our TIP experiments repeat work presented in Patterson's 1995 paper, we re-measured them on our (slightly different) system with little change.

4.3.1 CTIP improves remote performance

The most important, if expected, result in Figure 4 is that disclosing and acting on hints always helps. For remote storage, the hinted versions of the applications experience reductions in elapsed execution time of 17%-62% (or speedups of 1.2 to 2.6). The magnitude of these savings suggests that informed prefetching and caching is worthwhile, even if application data must be accessed over a network.

4.3.2 CTIP and TIP speedups are comparable

We can compare CTIP to TIP in many ways: raw execution time, raw reduction in execution time from unhinted to hinted, and percentage savings in execution time from unhinted to hinted. We consider the last two comparisons first.

We expect CTIP to perform slightly better in terms of raw execution time differential; for every I/O whose latency TIP hides, CTIP should be hiding an I/O of slightly higher latency. Table 5 reports the differences in execution time and the corresponding speedups and percentage reductions that come from exploiting hints in both the remote and local cases. For each benchmark, the absolute time difference between unhinted and hinted runs is better (larger) for CTIP.

Comparing the multiplicative speedups (or percentage reductions), however, shows both cases where CTIP is notably more and less effective than TIP. To quantify these, we compute the geometric mean of the speedups: on average, CTIP provides a speedup of 2.0 and TIP provides a speedup of 2.2, which suggests that hints benefit local storage about 10% more than they do remote storage.

4.3.3 Why is CTIP still slower than local TIP?

We would like to be able to say that CTIP manages to overlap the additional latency of remote storage access with local computation, and that it thus allows applications to run

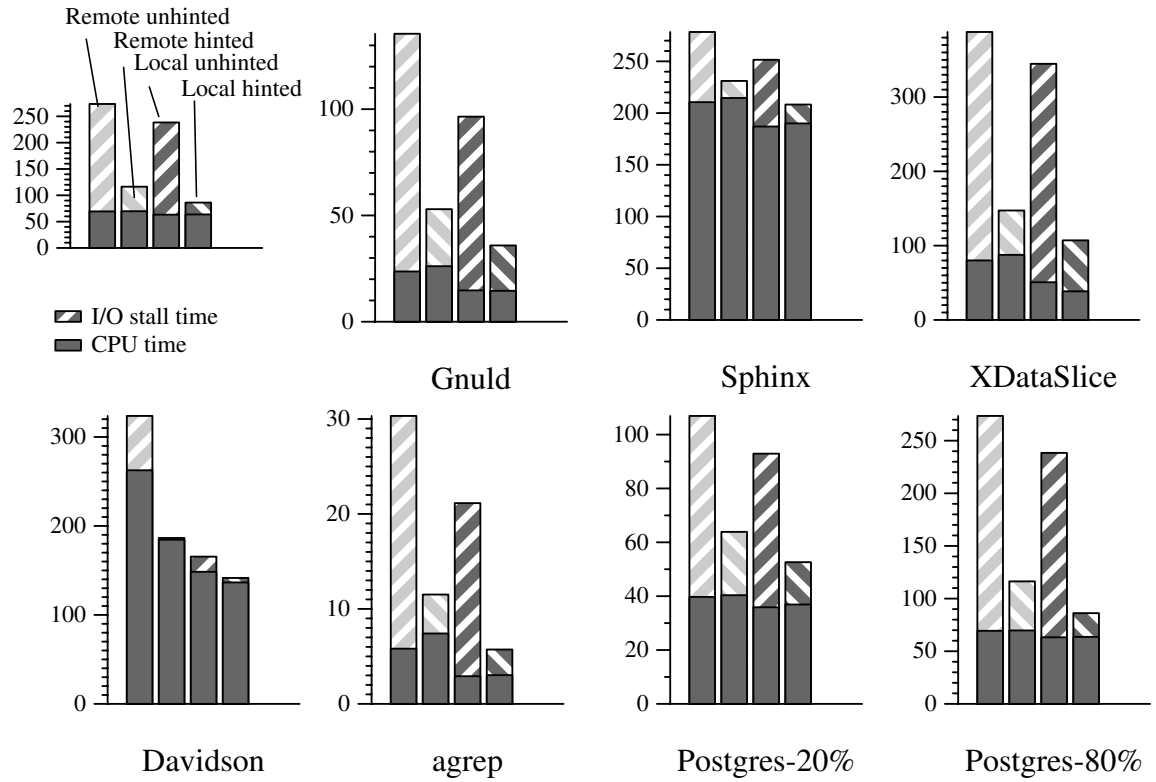

Figure 4. A comparison of the run times (in seconds) of our benchmarks running under CTIP (remote), TIP (local), unhinted local, and unhinted remote accesses. Each bar shows total run time for the corresponding benchmark, and the darker bar at the bottom represents total CPU consumption (by the application and the I/O system). All application code and data was striped over four disks.

as fast under CTIP as they do under local TIP. Unfortunately, our benchmarks still do not run as fast under CTIP as they do on a local system. There are several reasons for this disparity.

The first is the high CPU cost of accessing networked storage. Applications under CTIP must pay the higher T_{driver} costs of accessing network storage through multiple protocol stacks (see Figure 3) for each I/O access. This difference is apparent in Figure 4; remote experiments use more CPU time than do the corresponding local experiments.

A second reason for the disparity is increased latency for unhinted accesses. In addition to the increased CPU time, CTIP must pay increased stall time on applications' remaining unhinted accesses as it waits for data to traverse the server's CPU and the network between the client and server.

	CTIP (Remote)			TIP (local)		
Benchmark	**difference**	**speedup**	**reduction**	**difference**	**speedup**	**reduction**
Davidson	137.2s	1.74	42%	23.9s	1.16	14%
Gnuld	82.5s	2.56	61%	60.5s	2.69	63%
Postgres 20%	43.1s	1.67	40%	40.3s	1.77	43%
Postgres 80%	157.2s	2.35	57%	152.2s	2.76	64%
Sphinx	47.3s	1.20	17%	43.3s	1.20	17%
agrep	18.8s	2.64	62%	15.4s	3.69	73%
XDataSlice	240.3s	2.63	69%	237.6s	3.21	62%

Table 5. Comparing the benefits of TIP and CTIP. The geometric mean of the multiplicative speedups is 2.0 for CTIP and 2.2 for TIP, and CTIP reduces runtime by more than TIP for every benchmark.

Parameter (per 8k block)	Local (TIP)	Remote (CTIP)
T_{driver} (8K req)	580µs	877µs
T_{driver} (64K req)	198µs	1158µs

Table 6. Storage model parameters for TIP and CTIP and large requests. Remote performance suffers because remote filesystems cannot take advantage of clustering.

A third reason for this disparity, particularly evident in the Davidson benchmark, is clustering. In the local case, the I/O system coalesces, or clusters, requests for contiguous blocks into larger requests. This clustering has two effects: it allows the drive to transfer data more efficiently (without extra rotational delays), and it allows the client to amortize some of the driver cost over more data. The Digital Unix NFS code (like many NFS implementations) makes only single-block requests, and hence it cannot make the larger requests necessary to take advantage of this I/O clustering. Under NFS, the I/O subsystem must make eight separate 8KB requests and pay T_{driver} 8 times when the local filesystem could make a single 64KB request.

To quantify the extent of this clustering disparity, we repeated the experiments we used to compute T_{driver}, but with a 64KB requests (the maximum cluster size for the local filesystem). The results are shown in Table 6. The local system is amortizing part of the driver work over multiple blocks while the remote system is doing even more work per block. We believe this increase in work results from the overhead of context switching between the multiple NFS threads that handle the multiple blocks requested. Clearly, the NFS implementation is wasting performance relative to the more sophisticated local-disk management code.

4.3.4 For remote data, CTIP is faster

In some sense, our model is too pessimistic. Many users must run their jobs against shared data on storage attached to a machine where they cannot run their applications (either because policy prohibits it, or because their storage servers do not support general-purpose computation). Before these users can use the faster TIP system, they must copy the data to a temporary local store. Figure 7 shows how much time this "load" phase adds on to the execution of a single TIP run. For every benchmark except Davidson, the load cost must be amortized over several runs before using copy-and-TIP becomes faster than using CTIP.

4.4 CTIP with competition for server resources

Since file servers are shared resources, clients cannot always expect to be the sole users of their storage servers. To investigate the effect of competition for file service, we ran our benchmarks with a competing load on the file server. We induced the competing load with a process on the server that issued random 8KB reads into the same storage array that the

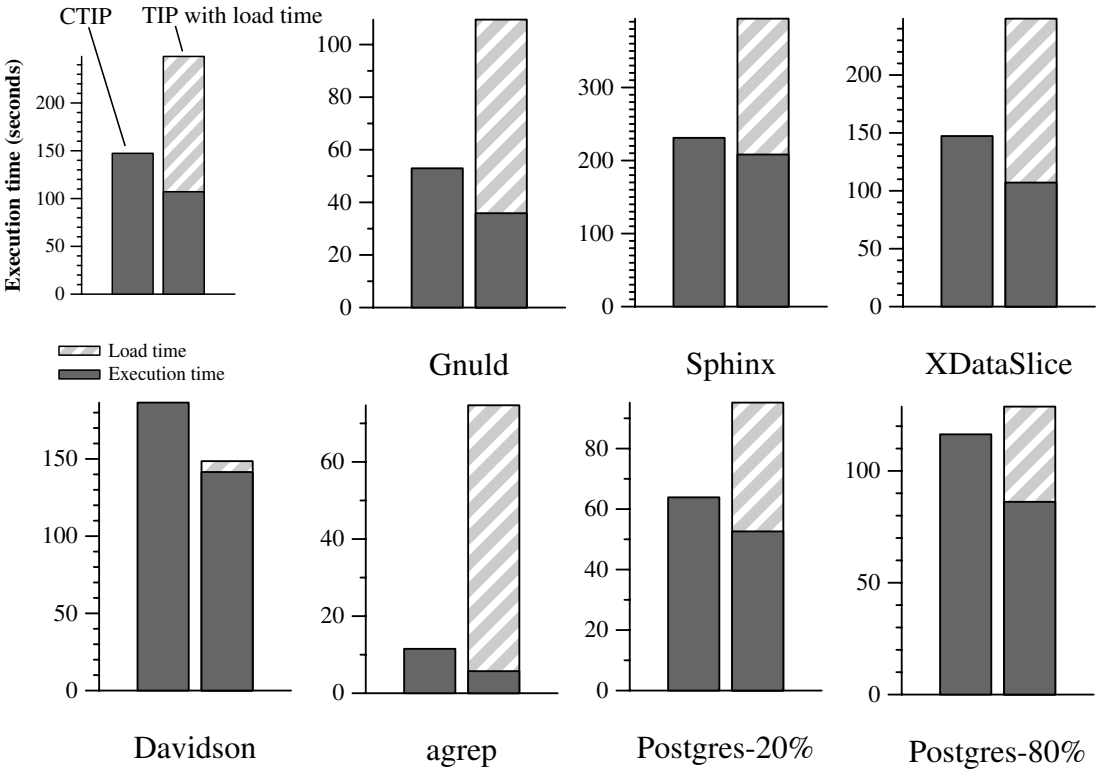

Figure 7. Reprise of hinted runs from Figure 4 with execution times for the local case enlarged by the time required to load the application and its data from remote (NFS) storage into local storage.

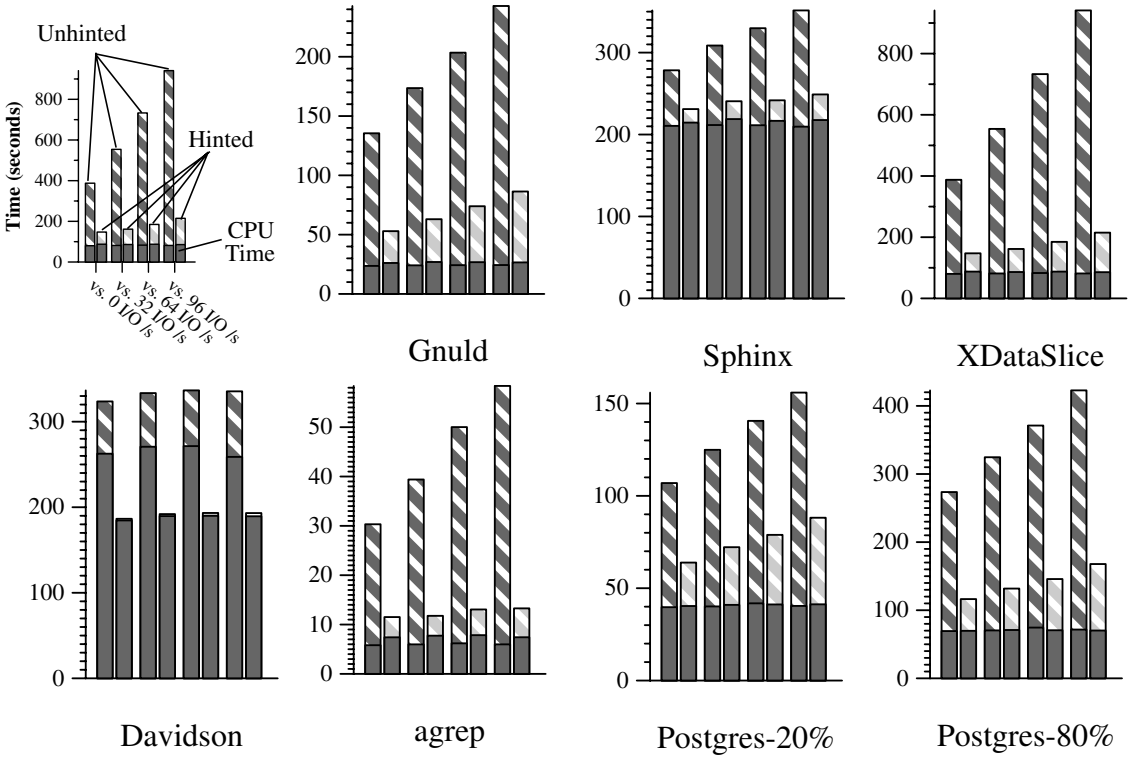

Figure 8. CTIP running in the face of competition. We present an increasing load on the file server and measure execution times for both hinted and unhinted runs. For some benchmarks (like XDataSlice and Agrep), the speedup for CTIP increases with the competing load.

client was making requests from. We used a process on the server so that we could easily make the reads span the entire array.

Figure 8 shows the time required to run our benchmarks as we varied the applied load from no competition to 96 reads/s (.75MB/s). As the competing load increases, queue depths at the server drives increase, and most applications experience significantly more wait time. The exception is Davidson, where aggressive sequential readahead on the server suffices to hide the additional latency. Notice that for applications other than Davidson, the increase in execution time due to this competition is smaller for the CTIP case than it is for the unhinted case.

This latency hiding comes at a cost. Although CTIP can reduce an application's bandwidth requirements through effective caching, it is often the case that CTIP applications issue more I/Os per second than their unhinting counterparts while they execute. When this happens, the competing process sees an increase in the response time for the reads that it issues. The read response time for the competing process started out between 20ms and 25ms with no CTIP benchmarks running against the server. Running an unhinted benchmark increased this latency by 5ms-20ms. For Davidson, this increase went down when hints were turned on, because CTIP substantially reduces the number of reads Davidson must make, but for other applications, response times got substantially worse. Gnuld is an extreme case: the 96 I/O/s competition process experiences a 27ms read response time against unhinted gnuld and a 53ms read response time against hinted gnuld.

5 Discussion

CTIP provides useful speedup, but its performance suffers in several cases because the client and storage do not share information effectively.

Although the network storage system has a potentially large cache (whatever cache is available on the server in addition to the cache already available on the client), CTIP does not currently make efficient use of the server cache; in fact, blocks recently read by the client are likely to be cached in both the server and the client. Cooperative caching techniques [3] [1] [8] hold promise in this area.

Our experience with clustering suggests that it is important to add enough richness to the network client-storage interface that the clients can (and do) adequately express clustered read requests.

In Section 4.4, we see that aggressive prefetching to a server can hurt performance for non-prefetching users of the same server. More intelligent access scheduling may be effective at ameliorating this problem. At the very least, servers should be able to distinguish between low-priority prefetching requests and regular demand reads.

Finally, if the client models the array of drives at the storage as a single black box, it cannot apply more advanced

techniques like TIPTOE and FORESTALL [9] [5] to modify its prefetching and compensate for load imbalances. A client-server interface that exposed this information to the client could help fix this problem.

6 Conclusion

We have examined CTIP, a straightforward extension of the TIP system based on treating remote storage as a reparameterized local disk. Though CTIP stretches TIP's simplistic storage model, TIP's cost-benefit buffer management strategies are robust enough to cope: CTIP reduces run times by from 17% to 62% over our benchmark suite, and in every benchmark, the amount of time saved under CTIP exceeds the amount of time saved under local TIP. The geometric mean of CTIP speedups is only 10% less than that for TIP. For TIP to be used on shared data stored on servers that do not support computation, that data must first be copied to local storage on the client. In almost every case, the time required for this copy overwhelms the differential advantage of local TIP.

Finally, hinting applications under CTIP perform robustly in the face of competition for server resources, slowing down much more slowly in the face of load than do unhinting applications. The principal limitations of CTIP are the increased CPU cost of going through the longer code path required to access networked storage, the increased latency of unhinted network accesses, and the standard NFS interface, which makes it difficult or impossible to express information about hints, I/O priorities, clustering, and load balancing.

7 Acknowledgments

We profited greatly from our discussions of this work with Jim Zelenka and Hugo Patterson. We would like to thank them, as well as David Kotz, who provided valuable feedback on this paper.

8 References

[1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang "Serverless Network File Systems" Proc. of the 15th ACM Symposium of Operating Systems Principles, Copper Mountain Resort, CO, December, 1995.

[2] Cao, P., Felten, E.W., Li, K., "Implementation and Performance of Application-Controlled File Caching," *Proc. of the First USENIX Symp. on Operating Systems Design and Implementation*, Monterey, CA, Nov., 1994, pp.165-178.

[3] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy, and Chandramohan A. Thekkath.Implementing Global Memory Management in a Workstation Cluster. Proc. of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain Resort, CO, December 1995.

[4] Garth A. Gibson, R. Hugo Patterson, and M. Satyanarayanan. Disk Reads with DRAM Latency. Third Workshop on Workstation Operating Systems, April 1992.

[5] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. Felten,G. Gibson, A. Karlin, and K. Lee. "A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching," Proc. of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA, October 28-31, 1996, pp. 19-34.

[6] Patterson, D., Gibson, G., Katz, R., A, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proc. of the 1988 ACM Conf. on Management of Data (SIGMOD)*, Chicago, IL, Jun. 1988, pp. 109-116.

[7] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. "Informed Prefetching and Caching," Proc. of the 15th ACM Symposium of Operating Systems Principles, Copper Mountain Resort, CO, December, 1995.

[8] Prasenjit Sarkar and John Hartman, "Efficient Cooperative Caching Using Hints," Proc. of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI),Seattle, WA, October 28-31, 1996.

[9] Andrew Tomkins, R. Hugo Patterson, and Garth Gibson. "Informed Multi-Process Prefetching and Caching," Proc. of the ACM International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '97), Seattle, Washington, June 15-18, 1997.