ENWRICH: A Compute-Processor Write Caching Scheme for Parallel File Systems*

Apratim Purakayastha
Department of Computer Science
Duke University
Durham NC 27708-0129
ap@cs.duke.edu

Carla Schlatter Ellis
Department of Computer Science
Duke University
Durham NC 27708-0129
carla@cs.duke.edu

David Kotz
Department of Computer Science
Dartmouth College
Hanover NH 03755-3510
dfk@cs.dartmouth.edu

Abstract

Many parallel scientific applications need high-performance I/O. Unfortunately, end-to-end parallel-I/O performance has not been able to keep up with substantial improvements in parallel-I/O hardware because of poor parallel file-system software. Many radical changes, both at the interface level and the implementation level, have recently been proposed. One such proposed interface is *collective I/O*, which allows parallel jobs to request transfer of large contiguous objects in a single request, thereby preserving useful semantic information that would otherwise be lost if the transfer were expressed as per-processor non-contiguous requests. Kotz has proposed *disk-directed I/O* as an efficient implementation technique for collective-I/O operations, where the compute processors make a single collective data-transfer request, and the I/O processors thereafter take full control of the actual data transfer, exploiting their detailed knowledge of the disk layout to attain substantially improved performance.

Recent parallel file-system usage studies show that writes to write-only files are a dominant part of the workload. Therefore, optimizing writes could have a significant impact on overall performance. In this paper, we propose ENWRICH, a compute-processor write-caching scheme for write-only files in parallel file systems. ENWRICH combines low-overhead write caching at the compute processors with high performance disk-directed I/O at the I/O processors to achieve both low latency and high bandwidth. This combination facilitates the use of the powerful disk-directed I/O technique independent of any particular choice of interface. By collecting writes over many files and applications, ENWRICH lets the I/O processors optimize disk I/O over a large pool of requests. We evaluate our design via simulated implementation and show that ENWRICH achieves high performance for various configurations and workloads.

1 Introduction

Many parallel scientific applications need high-performance I/O [17, 30]. The computational performance of multiprocessors has leaped far ahead of their I/O performance, making I/O the bottleneck for many of these applications. Although raw parallel-I/O hardware capabilities have improved substantially in recent years, they rarely lead to substantially faster application I/O. This shortfall is largely due to poor performance of the file-system software. In this paper we propose one technique that can dramatically improve file-system performance.

Most current multiprocessor file systems are derivatives of Unix file systems. Typical Unix workloads [29, 11], however, differ significantly from scientific multiprocessor workloads [24, 31]. Scientific programs use files for checkpointing, application-controlled virtual memory [9, 12], and visualization output, which are not common in Unix workloads. Furthermore, parallel scientific programs exhibit patterns that are more complicated than simple sequential patterns observed in vector scientific or Unix workloads. For example, many exhibit "forward-jumping" sequential patterns [24, 31], many of which are actually complex strided patterns [26, 27]. Clearly, parallel file systems must be redesigned to fit these common access patterns.

Several recent works have proposed changes to the file-system interface [3, 7, 8, 10, 12, 18, 19, 26]. One such proposed interface is *collective I/O*. In a traditional file-system interface, processes within a parallel job often have to express the transfer of a large object (e.g., a large matrix) as small, non-contiguous, per-processor requests, thereby losing valuable semantic information that a large contiguous object is being transferred. A collective interface allows all processes to make a single large request, preserving semantic information and thereby making it possible for the I/O subsystem to better coordinate the actual data transfer.

Disk-directed I/O (DDIO) is one efficient implementation technique for collective I/O [19]. In DDIO, the I/O processor (IOP) directs the order in which disk I/O operations should occur, after the job as a whole makes a collective I/O request. The IOP uses its intimate knowledge of the disk subsystem to optimize performance. Compared to a traditional-caching system, in which each IOP manages a buffer cache dynamically in response to many uncoordinated requests from all compute processors, DDIO has been shown to provide much higher throughput. It is not often easy, however, for a program to form its I/O requests as large, collective requests. Even in data-parallel programs, where all I/O was collective, we saw a surprising abundance of small requests [31]. These patterns and the difficulty of rewriting code to fit a collective interface can dilute the benefits of DDIO.

The CHARISMA workload studies— which spanned two systems, two sites, and two programming models [24, 31, 27]— exposed certain trends in multiprocessor file access. First, write traffic was high; the number of bytes written was almost double the number read, and the number of files written was almost double the number read. Though not generalizable to an arbitrary multiprocessor workload, it appears that optimizing writes may result in high performance for many applications. Second, bytes within write-only files were almost never shared across processors, although blocks

*This work was supported in part by the National Science Foundation under grant number CCR-9113170 and CCR-9404919.

From the Fourth Workshop on I/O in Parallel and Distributed Systems, May 1996.
THIS COPY IS THE AUTHORS' PRE-PUBLICATION VERSION; it may differ slightly from the official published version.
doi:10.1145/236017.236034

were often shared, resulting in false sharing. Third, there were many small writes. Even collective writes from many programs were limited to a few hundred bytes. These small writes tended to overload the I/O subsystem with many small requests, each of which had appreciable overhead, resulting in poor disk throughput.

In this paper we introduce ENWRICH (Efficient compute-Node WRIte caCHes)— a system that combines compute-node write caches for write-only files with disk-directed I/O for high-performance writes. Kotz explores ways to adapt many of the existing interfaces to DDIO [22], but, as proposed so far, DDIO enhances performance over *one* collective request from *one* job on *one* file. Moreover, to accommodate a set of generic patterns, the IOP has to maintain a sizable library of those pattern mappings. ENWRICH allows the powerful disk-directed I/O technique to be used for writes *irrespective* of any specific interface, and allows IOPs to optimize disk I/O not over just one collective request, but over many requests collected from many files and applications. Our simulations show that ENWRICH achieves high performance for various configurations and workloads.

In Section 2 we outline related work in this area. Section 3 outlines the design of ENWRICH and its operation. Section 4 describes our simulated implementation and evaluation methods, and Section 5 presents the results of our experiments. Finally, in Section 6 we conclude with our main observations and discuss possible future work.

2 Background

In this section we outline our model of a multiprocessor and its I/O subsystem, briefly survey write-caching studies, and summarize some recent relevant work in parallel I/O.

2.1 Model

In this work we focus on a MIMD multiprocessor model. We assume that there are, in general, two classes of processors: Compute Processors (CP) and I/O Processors (IOP). The CPs mainly run application code that involves computation, message passing, and file-system requests. The IOPs mainly service file-system requests from the applications. The processors are connected via an interconnection network. Each IOP may have one or more disks attached to it. We use a traditional file abstraction (a file is an addressable sequence of bytes), though we assume an underlying implementation that stripes file blocks across all disks. We assume the Parallel Independent Disk (PID) model, which means that each disk can be accessed independently and that different blocks can be read from different disks at the same time. Existing multiprocessors like the Intel iPSC/860, Intel Paragon, Thinking Machines CM-5, KSR/2, IBM SP-2, nCUBE, and the Meiko CS-2 are based on this model.

2.2 Write Caching

Write caches in multiprocessors have been limited to IOP caches at the file-block level. Kotz compared several IOP write-caching policies on a shared-memory multiprocessor. He concluded that a strategy in which a disk write was issued only after n bytes were written to an n-byte buffer (the WriteFull strategy) performed the best for his test workload [23]. Huber *et al.* propose centralized caching agents for write-shared files [16], a technique which still suffers network latency on every write request.

Compute-processor write caching in multiprocessors has been unpopular because of anticipated consistency overhead. Indeed, the amount of block sharing between compute nodes clearly predicts high consistency overhead for a block-level cache at the compute nodes [24, 31].

For uniprocessors, however, there have been numerous studies on various kinds of write-caching. In the log-structured file system [32], for example, file system changes are buffered and periodically written to disk in a single, big disk-write operation.

In Zebra [14], the log-structured file system is combined with a distributed striped file system. In the Scotch Parallel File System, targeted for a network-of-workstations supercomputing environment, client write caches provide weakly consistent file-sharing with the help of *propagate* and *expunge* mechanisms and explicit barriers to avoid read/write data hazards [13]. The xFS [1] exploits high speeds offered by ATM networks by using a technique called *cooperative-caching*. This technique allows a client to access file data directly from another client's memory. Cache consistency in xFS is maintained by a token-based scheme.

With emerging memory technology, the use of non-volatile, battery-backed-up RAMs (NV-RAMs) for write caching is becoming attractive. NV-RAMs allow write caches to use write-behind strategies, thereby reducing the number of disk-write operations. Biswas *et al.* show that a small amount of NV-RAM (about 1 MB) is sufficient to provide large performance gains [4]. Baker *et al.* show similar gains using a persistent write cache with a log-structured file system [2]. In ENWRICH, we draw on some of these results in the uniprocessor domain and apply them to multiprocessors.

2.3 Recent Advances in Parallel I/O

Earlier generations of parallel file systems like Intel CFS extended the Unix interface with file-pointer modes for parallel access [5, 28]. The Scalable File System (SFS) on the CM-5 also provides a Unix-like interface, but has an additional collective-I/O interface [3]. Among more recent parallel file systems, Vesta [8] allows users to specify both the logical partitioning of file data among the CPs and the physical partitioning of file blocks across the disks.

Some recent proposals focus on devising and implementing new and meaningful interfaces. Corbett *et al.* propose *MPI-IO*, which models I/O as message passing and allows programmers to express I/O with program datatypes rather than byte offsets within a file [7]. Nieuwejaar and Kotz propose a *nested-batched* interface for complex access patterns [26].

Two-phase I/O, proposed by Del Rosario *et al.*, is an efficient implementation of a large transfer operation where data is permuted in CP memory before a collective I/O operation so that the I/O operation conforms to the actual file layout for better I/O performance [10]. Disk-directed I/O (discussed in Section 1), proposed by Kotz, efficiently implements collective I/O, out-of-core computations, data-dependent distributions, and both regular and irregular requests [20, 21].

3 Design and Operation of ENWRICH

Write caches are normally used to delay writes, avoiding extraneous disk I/O for blocks that are overwritten in pieces. Although ENWRICH achieves that, its primary motivation is different. ENWRICH's caches hoard a large number of writes, and then flush them periodically, simultaneously, with disk-directed I/O. The main focus is not to *reduce* the number of block I/Os but to accumulate a large number of them and then perform them in an order dictated by DDIO. Thus we optimize disk access over a large number of writes involving many jobs and files. Putting the caches at the compute processor rather than on the I/O processor has other advantages: first, it avoids network delay for every write, and second, it relieves the IOP of managing a cache that is shared by many CPs. This technique of batching writes, although for somewhat different reasons, has been used successfully in the log-structured file system [32] and in the Zebra file system [14]. Note that ENWRICH only caches writes to files opened only for writing (writes to write-only files were dominant in CHARISMA studies; the number of files that were both read and written was tiny). Read-only files could be cached in separate per-CP caches and read-write files could be cached in IOP buffer caches. As outlined later, this selective compute-node write caching allows us to solve the cache-consistency problem with minimal overhead. The consistency check is actually overlapped with useful work that eliminates extra block I/O at the IOPs.

3.1 The Cache Organization

Each compute processor has a cache that is divided into two sections, a directory section and a data section (Figure 1). For each write request on that particular processor, the data is appended to the data section and a new entry is appended to the directory section. The directory entry consists of a compute-processor number,[1] a unique file id, a timestamp, an offset into the file where the write has occurred, and the number of bytes written. Note that the cache is not aware of file blocks. This log-based cache design was inspired by the presence of false-sharing of file blocks in our traces. A CP write cache based on file blocks would waste space through replication and cause excessive consistency traffic. Our system only incurs minimal cache-management overhead for each write. When the caches are emptied out in a flush phase (discussed below), the CP can easily calculate an offset in the data section corresponding to a particular directory entry by summing the byte counts in the preceding directory entries. Note that the ENWRICH design does not specify how the physical cache is logically divided into a data section and a directory section. An implementation may choose to divide it into two static parts, which implies that the cache is full when either section is full, or it may choose to use the same physical memory for both sections but have them grow in opposite directions, which implies that the cache is full when the two sections meet.

3.2 The Flush Operation

Each write appends data to the CP cache. Obviously, the caches occasionally need to be written out to the disk (called a *cache flush*). Figure 2 summarizes the events in a flush operation. When the cache in one CP becomes full, it broadcasts interrupts to all processors, triggering a flush operation. Each flush operation starts with a barrier involving all the processors. The barrier does not halt user computation, unless the computation attempts to write to the file system. It is merely a means of synchronizing the flush activity across all CPs. After the barrier, the CPs send relevant parts of their directories to different IOPs and then wait for the IOPs to request data from them. IOPs start to process the directories after they have received them from all the CPs.

Each IOP checks *consistency* as follows: it first sorts the directory entries by the unique file-id field; then it sorts the entries in increasing file-offset order; then it traverses down the sorted list checking for byte-sharing between any two entries; if there is byte-sharing, it is resolved using the timestamp values associated with the conflicting entries (the older entries are ignored). This operation not only ensures consistency but also saves extra disk I/O that may have occurred in a traditional-caching system. We assume adequately synchronous CP clocks. To be consistent with causal ordering, the clock drift should be less than the message-passing delay between any two processors. This is a reasonable assumption for current-generation multiprocessors like the SP-1, SP-2, and CM-5. In a cluster of workstations, clocks drift by larger amounts; as long as the clock drift is less than the message-passing delay, our technique works. Systems that have hot replacement of processors must adapt by blocking on cache writes (continuing all other operations), until the new processor is in adequate synchrony with all the other processors.

After the above phase, the IOPs perform pure DDIO: each IOP constructs a list of unique blocks that it needs to write in this flush; then it re-orders the block list according to actual disk layout; then it requests data corresponding to each block from CPs that have directory entries for this block. When CPs receive a request for data, they look up their directory entries and return the relevant portion of the data cache to the requesting IOP. When all data corresponding to a block has arrived, the IOP issues a disk write request for that block and starts the above procedure for the next block.

When an IOP has finished writing all its blocks, it broadcasts "done" messages to all the CPs and ends the flush operation. When a CP has received "done" messages from all the IOPs, it clears its cache for the next flush operation and exits the current flush operation.

The time interval between cache flushes depends on the particular flush policy. The flush policy may be a static policy based on a timeout, or it may be a dynamic policy based on cache fullness. A dynamic policy that potentially holds write-data in the cache for a long time would likely result in more throughput because the IOP would be able optimize over a larger number of writes; it also would be more adaptive to system load than a static policy. It could also lead to longer stalls during the flush phase, however, since no new writes could be added to a full cache, while in a static policy data could be written to empty parts of the cache while a flush is in progress. A well-tuned ENWRICH implementation would likely have a policy that flushes the cache when either a timeout occurs or more than a threshold of the cache is full, thereby allowing computation and new writes to happen while the cache is flushed in the background. Any policy may also result in write data being cached for a time long enough to violate read-after-write consistency. An ENWRICH implementation would likely use a per-file flush-on-read policy when a file is re-opened for reading after having been written.

We claim that combining compute-node write caches with disk-directed I/O in this way has several benefits. First, it enables programs to use the power of disk-directed I/O irrespective of a particular interface. Second, the caches collect writes over many files and applications, which let the IOPs optimize disk I/O over a large pool of requests rather than a single collective request. This capability is especially important since we observed even collective requests to be small. Third, the consistency scheme is not purely overhead since it also eliminates unnecessary disk I/O in the process. Moreover, the consistency mechanism is implemented without inter-CP communication, which saves message-passing overhead. Fourth, this approach results in low-latency writes. Since the flush operation offers high throughput via DDIO, this approach combines low latency and high throughput for writes, a desired but so far unreached goal in existing parallel file systems.

4 Evaluation Method

We have implemented ENWRICH on top of the STARFISH[2] simulator, which is further based on the Proteus parallel architecture simulator [6]. Proteus is an execution-driven simulator that provides a

[1] The CP number is not strictly necessary, but it simplifies processing at the IOP.

[2] More information about STARFISH, including source code, can be found at the URL http://www.cs.dartmouth.edu/research/starfish.

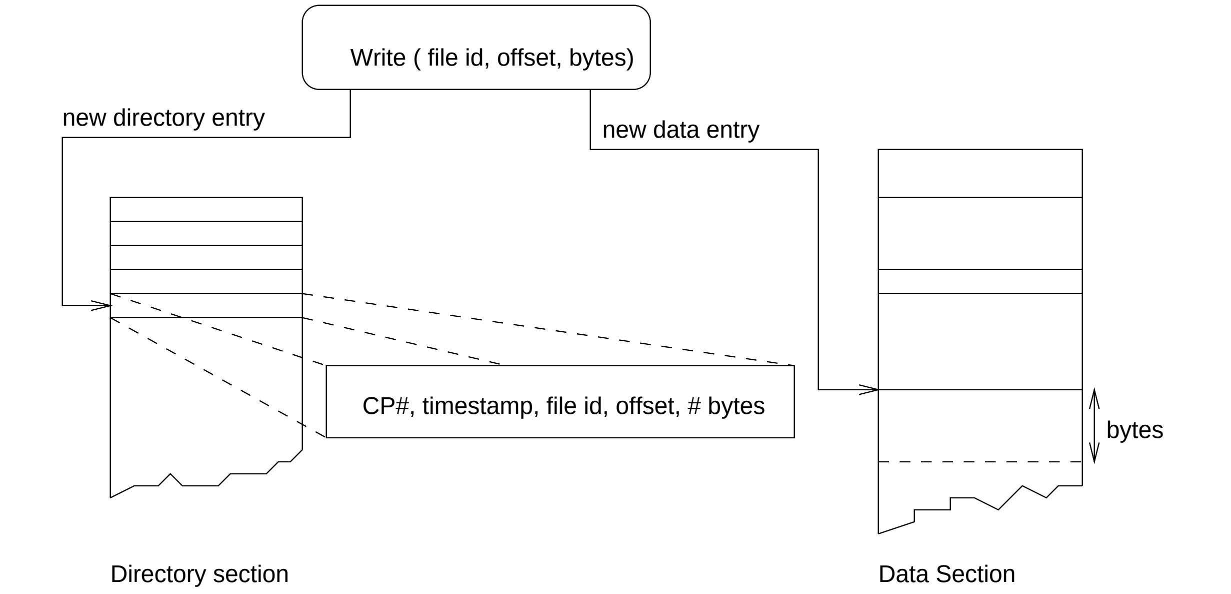

Figure 1: The log-structured write cache at each compute processor.

generic framework with basic message-passing capabilities, interprocessor interrupts, thread operations, and shared memory. Proteus runs as a single multithreaded process on a uniprocessor.

The STARFISH simulator is essentially a Proteus application, but it provides various higher level abstractions to facilitate parallel-I/O simulation. It includes a validated disk model [25] based on Ruemmler and Wilkes' HP97560 model [33]. It provides the framework of a multiprocessor system with CPs and IOPs, and IOPs having one or more disks and one I/O bus attached to each IOP. Some file systems, like the traditional-caching system and the disk-directed system, are also implemented as part of STARFISH.

ENWRICH Implementation. We have implemented ENWRICH as a new file system on STARFISH. We use the STARFISH disk model with minor modifications. Since we focus on write caching, the current implementation of ENWRICH does not have a read interface at the API level.

Each CP has a *main* thread that generates file system requests according to a pattern, manages the local write cache, and broadcasts flush interrupts according to the flush policy. The CP write caches are statically divided into a data section and a directory section, both of which have fixed maximum sizes. The maximum size of the directory section is a simulator parameter, *directory fraction*, that determines what percentage of the total cache is to be used to store directory metadata. In our experiments, flush interrupts are generated by a CP when either the directory part or the data part of its cache is full.[3] In response to a flush interrupt, each CP wakes up a *flush* thread that coordinates the actual flush operation (Figure 2).

The IOP is essentially idle except for the flush phase. The flush interrupts broadcast from a CP are handled by an interrupt handler at the IOP that wakes up the IOP flush thread. The IOP flush thread coordinates the actual flush operation (Figure 2).

Traditional Caching. We have borrowed the traditional-caching implementation directly from STARFISH and extended it for multiple files. In this system, the CP simply forwards each I/O request to the IOP. Each IOP manages a cache that has two buffers per CP per local disk, large enough to double-buffer an independent stream of requests from each CP to each disk. The IOP write cache uses a WriteFull policy (Section 2). More details about the traditional-caching implementation can be found in [19].

4.1 Experimental Design

Most of our experiments had 32 processors (16 CPs and 16 IOPs). Each CPU was a generic RISC with a 50 MHz clock. Each IOP had one SCSI bus with 10 MB/s peak bandwidth. Most experiments had 16 disks (one per IOP); each disk had a maximum capacity of 1.3 GB and a peak transfer rate of 2.34 MB/s. The interconnect for most experiments was a 6-by-6 bidirectional torus with wormhole routing. The interconnect latency was set to 20 ns per router with a peak bandwidth of 200 MB/s. Appropriate message-specific software overhead is accumulated within the simulator. For ENWRICH experiments, the per-CP write-cache size was 1 MB with the directory fraction set to one-tenth of the cache size. Some of these *base-configuration* parameters were varied to conduct sensitivity studies; we discuss our variation of these parameters in relevant sections.

We ran experiments to compare ENWRICH with traditional caching, using data-transfer throughput as a metric.[4] Each experiment was repeated five times, and results averaged, to account for disk-layout and network randomness.

Workload. Most of our experiments used synthetic access patterns that were derived from different modes of distributing two-dimensional matrices in CP memories, as suggested by those available in High Performance Fortran [15, 10] (Figure 3, adapted from [19]). Elements in each dimension of a multidimensional array could be mapped fully in one CP (denoted as NONE), distributed in contiguous segments among the CPs (BLOCK[5]), or distributed

[3] A flush operation is also forced when each experiment terminates, to write out cached data to disks, as in the traditional caching system.

[4] ENWRICH cache size was larger than that of TC. TC would not benefit from a larger cache due to lack of temporal locality in our workload. Moreover, Figure 6 shows that ENWRICH outperformed TC even when the total cache sizes were almost equal.

[5] "BLOCK" here represents simply a name for a mapping, and is distinct from a file-system block.

Compute Processor

```
barrier ();

for each (IOP)
  send relevant part of
     the directory;
endfor;

while (not received "done" messages
          from all IOPs)
   service data requests from
      different IOPs;
endwhile;

reset data and directory pointers
   in cache;
```

I/O Processor

```
barrier ();

while (not received directories from all CPs)
    accumulate directories received from the CPs;
endwhile;

Perform consistency check by sorting and
    comparing directory entry timestamps;

Form a list of blocks that need to be written to the
    disk and re-order them according to physical
    layout;

for each (block in the list)
      process relevant directory entries;
      request data from relevant CPs;
      when all data is received issue disk write;
endfor;

send "done" message to all CPs;
```

Figure 2: Events during a flush operation at the Compute and I/O processors. The flush operation begins in response to interrupts broadcast from the CP that is causing the flush.

in a round-robin fashion among the CPs (CYCLIC). We named the patterns in a shorthand notation starting with *w* for *write*, followed by the first letter of the mapping pattern for the first dimension, and then the first letter of the mapping pattern for the second dimension (e.g., in this system a 2-d array mapped NONE in the first dimension and CYCLIC in the second dimension would be called *wnc*).

For some experiments we wrote a single file that contained a single 2-d array mapped in the CPs in one of the above patterns. For some experiments we used another pattern named *wss* (for Write Synchronous Sequential). For this pattern, the array layout was NONE-CYCLIC, but the CPs requested corresponding chunks synchronously. This pattern was abundant in CMMD programs [31].

Our studies show that some jobs write to two files at the same time [31]. To capture the flavor of such a scenario, we devised patterns for experiments where CPs simultaneously wrote two different arrays with different mappings that were output to two different files on disk. We experimented with a few combinations of simple patterns, and the order in which one CP wrote chunks belonging two arrays was arbitrarily set to round-robin. The shorthand names we use for these patterns start with *w* for *write*, followed by *r* for *round-robin*, followed by the abbreviations for the simple patterns (e.g., a pattern with two arrays mapped in NONE-BLOCK and BLOCK-NONE fashion that were output to two different files was named *wrnbbn*). The patterns that we actually used for this paper were wrnnbb, wrnncc, wrnbbn, and wrbccb.

It is also common for multiprocessors to have separate partitions that run different jobs at the same time. These partitions often share the same I/O subsystem that services requests from jobs running in different partitions simultaneously. Again, to roughly model such a scenario, we devised patterns in which we essentially had half the CPs writing out an array in some simple pattern to one file, and another half writing out yet another array in a possibly different pattern, to a different file. The actual interleaving of requests generated was determined by the way Proteus scheduled the CP threads to run. We named the patterns like the round-robin ones, but simply changed the letter *r* to *p* to indicate *partition* (e.g., if two different CP groups wrote two arrays mapped in NONE-BLOCK and BLOCK-NONE fashion, the pattern was called *wpnbbn*). The patterns that we actually used were wpnnbb, wpnncc, wpnbbn, and wpbccb.

We chose three different record sizes. Experiments with an 8-byte record size (size of a double-precision floating point number) investigated system performance at one extreme with tremendous contention and interprocess locality. Experiments with an 8192-byte record size (size of a file-system block) exercised the system at another extreme where there was minimal contention and interprocess locality. Experiments with a 512-byte record size reflect a "popular" range of request sizes [31].

The synthetic patterns described above helped us explicitly control the experiments and answer questions such as why performance of both TC and ENWRICH varied from pattern to pattern in different configurations. For a more realistic comparison we also experimented with a workload derived from some of the traces we collected in [31]. Section 5.3 describes the trace-driven experiments in more detail.

File and Disk Layout. We used an 8-KB file-system block size in all the experiments. Each file was striped, at block granu-

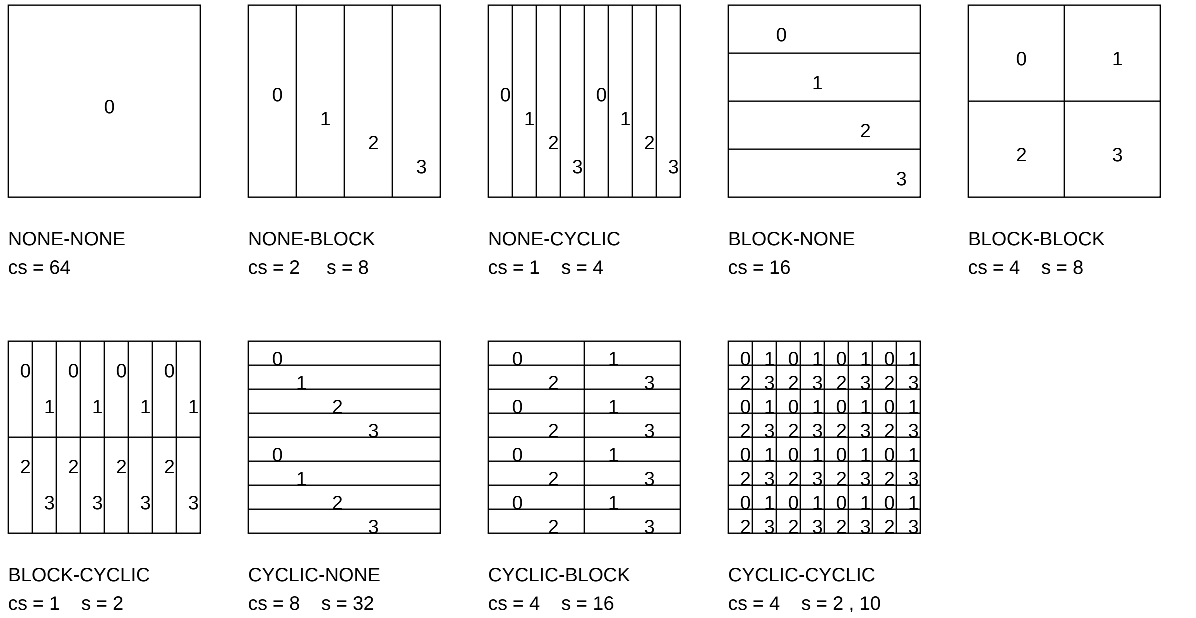

Figure 3: 2-d array patterns that represent several ways of distributing an 8x8 matrix over four processors as suggested in HPF. Patterns are named by the distribution method (NONE, BLOCK, or CYCLIC) in each dimension. Each region of the matrix is labeled with the number of the CP that it gets mapped to. The *chunk size* (cs) is the largest contiguous chunk of the file that is sent to a single CP (in units of array elements), and the *stride* (s) is the file distance between the beginning of one chunk and the beginning of the next chunk mapped to the same CP, where relevant.

larity, across all disks. The arrays were stored in row-major order within the files. The total amount of data written in most synthetic experiments was 64 MB. For two-file experiments we wrote two files of 32 MB each. We experimented with larger and smaller files but found no qualitative changes in our results. The 64 MB data size appeared to be a good compromise that saved us simulation time but still caused a significant number of cache flushes in ENWRICH experiments. For some patterns with 8-byte records we used file sizes of 8 MB for both TC and ENWRICH experiments to save excessive simulation time.

We experimented with two extreme disk layout policies: *contiguous*, where logically consecutive file blocks were actually mapped on the disk in a physically consecutive manner; and *random*, where files blocks were placed on random physical locations on the disk. Kotz in [19] asserts that a real system would have a layout intermediate between the two and would have intermediate performance.

5 Results

In this section we first study ENWRICH and Traditional Caching (TC) performance on the base configuration described in section 4.1. We then investigate how the systems respond to various changes to the parameters in our base-configuration. Finally, we compare the systems using some traces collected from a production workload.

5.1 Base-configuration Experiments

Figures 4 and 5 compare the performance of ENWRICH with the traditional caching method for 8192-byte, 512-byte and 8-byte record sizes. ENWRICH was almost always faster than TC. Indeed in one case it was up to *13 times faster.*

Figure 4 compares TC and ENWRICH in a random-blocks disk layout. For 8192-byte and 512-byte record sizes ENWRICH almost always reached throughputs between 7 and 8 MB/s, while the highest TC could reach was 5.3 MB/s. The only pattern where TC was faster than ENWRICH was the NONE-NONE pattern, which was heavily biased against ENWRICH. In this pattern, all data is mapped to one CP. Thus, even with 16 CPs and a 1 MB cache per CP, the effective total ENWRICH write cache size for the whole system was reduced to just 1 MB. Hence ENWRICH caused a whopping 72 cache flushes to write out the 64 MB file, instead of just 5 flushes as in most other cases. On the other hand, when we diluted the pure NONE-NONE pattern with another pattern like BLOCK-BLOCK or CYCLIC-CYCLIC in two-array experiments such as *wrnnbb* and *wpnncc* — where still half the data (32 MB) is mapped to one CP and the other half mapped to all CPs — ENWRICH performed better than TC. Hence, even in a scenario where one CP monopolized half the data in the system, ENWRICH performed better than TC. We also found that both ENWRICH and TC had difficulty with some patterns for 8-byte record sizes. For patterns like *wnc, wcc*, and *wpnncc*, I/O took place in small 8-byte chunks. In such cases, while TC suffered badly due to intense buffer contention problems, ENWRICH inherited DDIO overhead for transferring 8-byte chunks from CPs [19] and also caused excessive *directory flushes*. A *directory flush* was an undesirable situation where a flush occurred because there was no space to write metadata. With 8-byte I/O, the CP stored 24 bytes of directory metadata for each 8 bytes of real data. Hence, each flush actually transferred little real data to disk, requiring many directory flushes. Even with such disadvantages, ENWRICH almost always performed at par or better than TC, although not quite as well as pure DDIO [19] wherever DDIO could be applied.

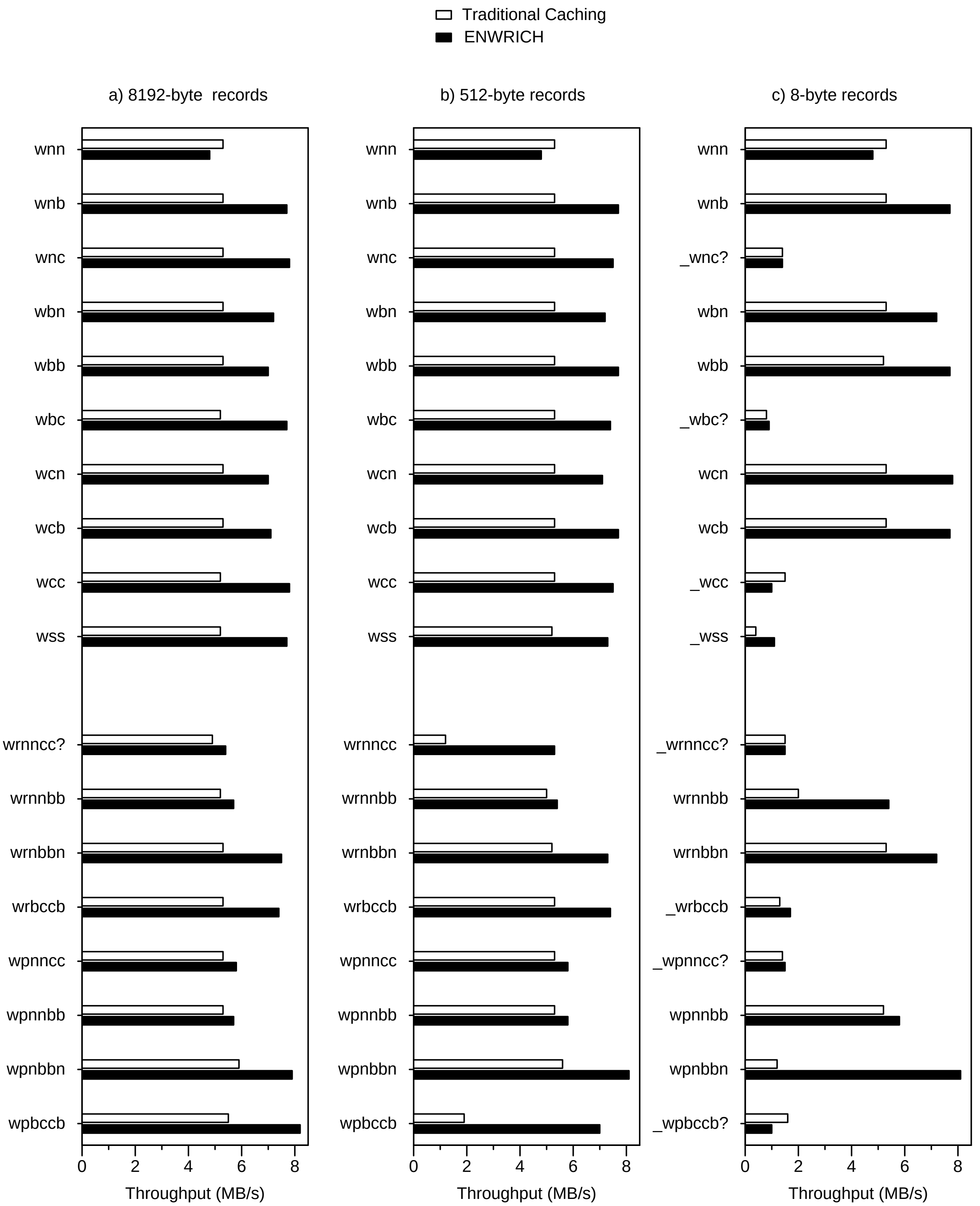

Figure 4: Graphs comparing the throughput of ENWRICH to that of traditional caching with a **random** disk layout for three record sizes. The 8-byte patterns starting with an underscore used a file size of 8 MB instead of 64 MB. Each bar represents the average of five independent trials (maximum coefficient of variation was 0.16). Results for all patterns except the ones ending with a question mark were statistically significant at the 95% confidence level.

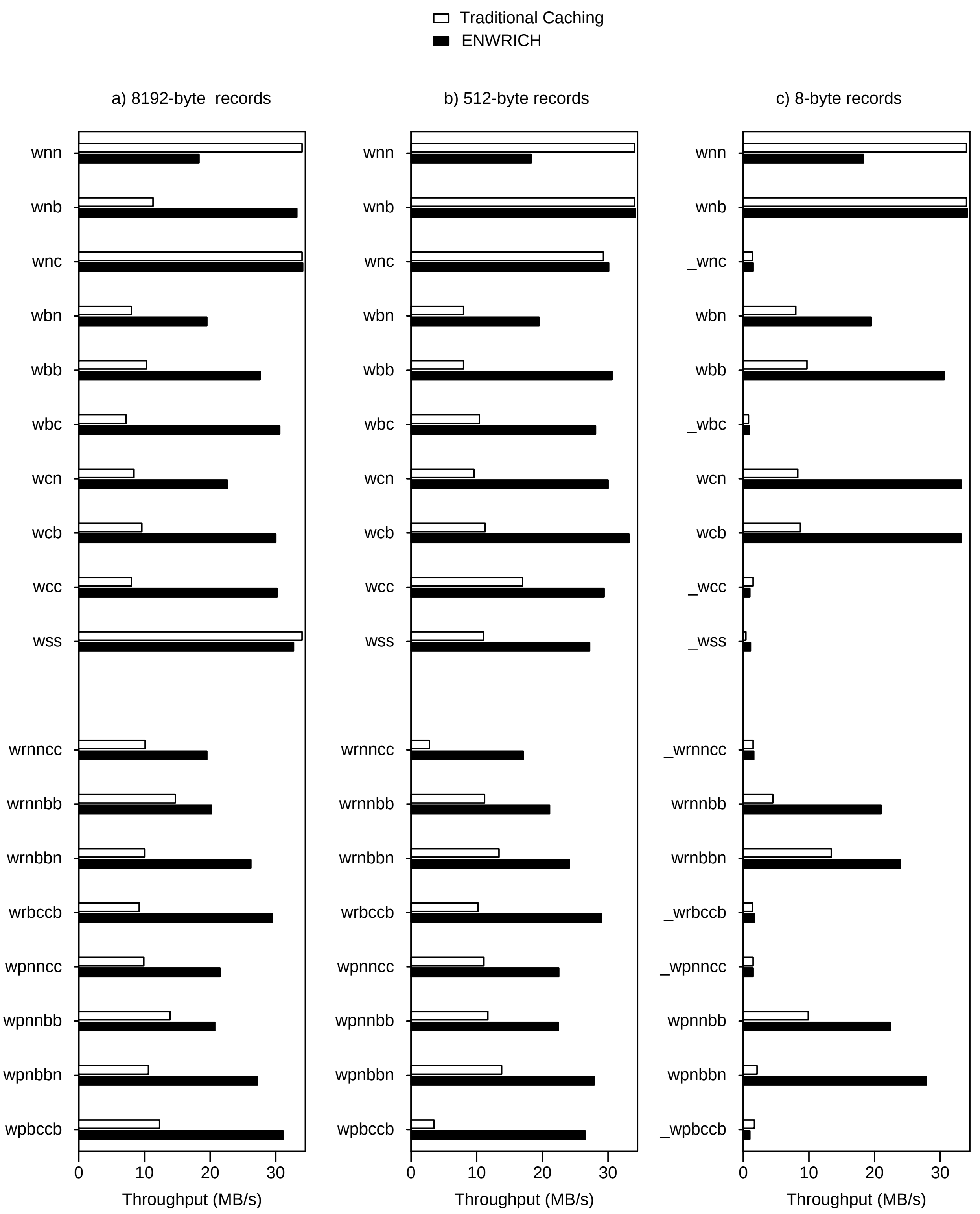

Figure 5: Graphs comparing the throughput of ENWRICH to that of traditional caching with **contiguous** disk layout for three record sizes. The 8-byte patterns starting with an underscore used a file size of 8 MB instead of 64 MB. Each bar represents the average of five independent trials (maximum coefficient of variation was 0.14). All results were statistically significant at the 95% confidence level.

Although disk-layout strategies are not the focus of this work, we wanted to test system performance with a contiguous layout, the opposite extreme of a random-blocks layout. Figure 5 summarizes the results of our experiments. With this layout, higher throughputs were obtained for both TC and ENWRICH, but ENWRICH exploited the advantages of a contiguous disk layout much more than TC could. For a large majority of cases, ENWRICH reached throughputs in the range of 30 MB/s (80% of the peak disk throughput of 37.5 MB/s), and for some favorable cases it reached 34.1 MB/s (91% of the peak disk throughput). For a few tests with favorable patterns and record sizes, TC also reached 34.1 MB/s, but for a large majority of patterns and record-sizes, TC throughput hovered around 9-10 MB/s (25% of the peak disk throughput). As in the random layout, TC performed markedly better than ENWRICH for the *wnn* pattern. Again, when the *wnn* pattern was diluted with another array distributed as BLOCK-BLOCK or CYCLIC-CYCLIC (bringing the data distribution a little closer to reality), ENWRICH outperformed TC comfortably. For reasons discussed in the random-blocks layout case, for some patterns such as *wcc, wpbccb*, with 8-byte record sizes, ENWRICH lost performance significantly. It still performed at par with TC but worse than DDIO [19]. Except for the *wnn* pattern, in the few other cases where ENWRICH performed worse than TC, it was never worse by more than 5%.

TC lost performance for a few important reasons. For patterns like *wcn*, there was little interprocess spatial locality. For 8-byte cyclic patterns, buffer contention increased tremendously and brought TC performance down appreciably. In general, higher data rates in the contiguous layout exposed the cache-management overhead as a bottleneck in TC. TC appeared to have suffered most with two-array patterns. For all two-array patterns and all record sizes, its throughput never exceeded 12 MB/s, and the average was less than 9 MB/s. Multiple per-file localities hurt IOP caching and disk-access locality in these cases.

For reasons quite different than the above, ENWRICH also failed to deliver high-performance in some cases. For *wnn* ENWRICH failed due to effective reduction of the overall write-cache size to just 1 MB. For certain 8-byte patterns, ENWRICH failed due to metadata overhead. For some patterns, like *wbn,* ENWRICH performed twice as well as TC but could not deliver more than 20 MB/s because of an imbalance in the per-IOP block distribution during each flush. In ENWRICH, the delay in a flush phase was determined by the slowest IOP: if all IOPs had n blocks to flush but one had $n + 4$ blocks to flush, the flush operation did not end until that IOP had transferred $n + 4$ blocks. Additional IOP buffering could eliminate the impact of some load imbalances.

ENWRICH succeeded in a critical way. Without new and unusual interfaces, an issue that the parallel-I/O community is still debating, ENWRICH often delivered more than 80% of the disk-bandwidth in our experiments with contiguous disk layout. The performance upper bound for ENWRICH was DDIO, wherever DDIO was applicable. This makes sense, because ENWRICH incurs all the DDIO overheads as well as the CP cache-management and flush overheads.

5.2 Sensitivity Study

Figure 6 shows the performance variation of ENWRICH with different sizes of the per-CP cache. We kept the parameters in our base configuration fixed except for the per-CP cache size, which was set at 0.25 MB, 0.5 MB, 1 MB, and 2 MB for four sets of experiments. The directory fraction (part of the total cache reserved for metadata) was kept constant at one-tenth. Even with a 0.25 MB per-CP cache, ENWRICH performance was markedly better than TC performance in the same configuration for most patterns. ENWRICH performance increased with increasing per-CP cache size mainly because fewer flushes were caused and because there were more blocks per flush operation over which each IOP could optimize disk-I/O. For most patterns, the percent performance increase with increasing cache size declined as cache sizes were successively doubled from 0.25 MB to 2 MB. This was caused by an increase in per-flush directory overhead on the IOPs. Patterns like *wnc, wbc, wcc*, and *wss*— which generated more directory entries for the same amount of data than patterns like *wbn, wcn*, and *wbb*— exhibited diminishing returns with increasing cache size.

Figure 7 shows performance of ENWRICH and TC when varying the number of CPs. The numbers of IOPs and disks were kept constant at 16. The IOP cache size for traditional caching was two buffers per CP per disk. The ENWRICH cache size was 1 MB per CP. The total cache size increased with the number of CPs in both systems. We used the contiguous disk layout and 8192-byte record sizes. We chose to illustrate only selected patterns from different sets such as 1-array patterns, 2-array round-robin patterns, and 2-array partition patterns. A sensitivity study on all patterns was not feasible, but we believe that these patterns, in general, were able to capture all the nuances. With TC, performance in general fell off with an increasing number of CPs (there were some exceptions that showed a slight performance increase from 4 to 8 CPs). For two patterns, TC showed tremendous performance for a 1-CP system, but they degraded sharply as the number of CPs increased. The general degradation of TC performance with increasing number of CPs can be attributed to greater contention involving multiple CPs and greater cache-management overhead with increasing cache size on the IOP and less disk locality. ENWRICH performance usually fell from a 1-CP to a 2-CP system. Apparently, the benefit obtained by having a bigger overall cache with the 2-CP system was offset by increased synchronization overhead in the flush phase with the 2-CP system compared with the 1-CP system. For larger systems, however, ENWRICH performance scaled fairly well with an increasing number of CPs, and hence increasing overall cache size. The benefit from increasing total cache size and the resulting decreasing number of flushes seemed to outweigh the increased cost of synchronization at each flush in systems with more than 2 CPs.

We also varied the number of IOPs, keeping the number of CPs and disks fixed at 16, and keeping the total cache size for both TC and ENWRICH constant (Figure 8). For a 1-IOP system with 16 disks serviced by just the one SCSI bus, ENWRICH and TC performed at par in the 8-9 MB/s range (peak throughput was limited to 10 MB/s by the bus bandwidth). With more IOPs and hence fewer disks per IOP, performance of both ENWRICH and TC improved but ENWRICH improved much more dramatically over TC. ENWRICH seemed to take better advantage of the reduced bus contention than TC. Apparently, TC performance was limited by the multiple disk localities seen in most of these patterns. For patterns like *wnc*, which did not have multiple localities, TC performance scaled with the number of IOPs.

Finally, we varied the number of disks, keeping the number of CPs fixed at 16 and the number of IOPs fixed at 8 (Figure 9). We tried configurations with 8, 16, 24, 32, and 40 disks, that is, the number of disks per IOP varied from 1 to 5. The IOP cache size for TC was 2 buffers per CP per disk, which increased with the number of disks in the system. The per-CP cache size for ENWRICH was kept constant. Performance of both systems scaled with number of disks, but ENWRICH achieved throughputs as high as 46 MB/s with 5 disks/IOP, where TC could only achieve about 25 MB/s for the same configuration. The ENWRICH performance gain for the pattern *wpnncc* was worse than other patterns because the benefit from a larger number of disks was offset by mapping 32 MB of data to just one CP. Similar results were obtained using the random layout (Figure 10), although the throughput values were much lower.

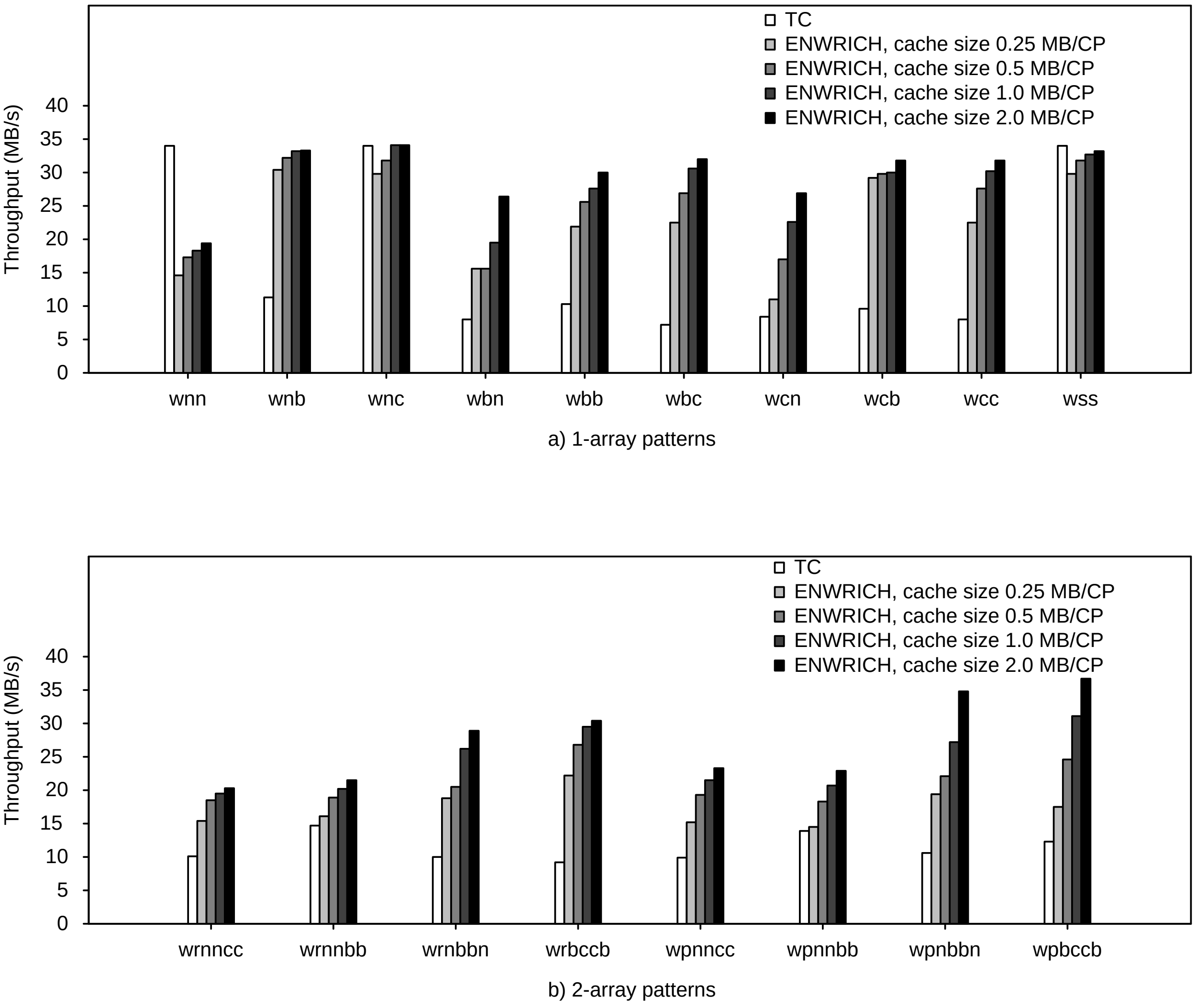

Figure 6: Graphs showing ENWRICH performance for various per-CP cache sizes for 1-array and 2-array patterns, compared with TC performance. All experiments used contiguous disk layout, 8192-byte records, 64 MB files, and 16 each of CP, IOP, and disks.

5.3 Trace-driven Experiments

In this section we describe results obtained by driving both ENWRICH and TC with traces obtained from a workload study on the CM-5 at the NCSA [31]. We had to make some significant compromises in using the traces. First, due to hardware limitations we could not simulate multiprocessors with a large number of nodes, which limited us to using traces only from jobs that ran on 32-node partitions (the next bigger partition was 64 nodes, which was too big for our simulations). Second, our current ENWRICH implementation only has a *write* interface, limiting us to use traces from jobs that only wrote to files. Third, we had to exclude jobs that wrote files larger than 2 GB because our simulator could only handle 4-byte file offsets. Write requests were sorted in increasing time order, and a proportional number of simulator cycles were added between two write requests to represent computational delay that was derived from the traces.

The traces included several self-selecting CMMD jobs, and most CMF jobs. For logistical convenience, traces were preprocessed and coalesced from selected CMMD jobs into one trace called *cmmd*, and from selected CMF jobs into three traces called *cmf1, cmf2,* and *cmf3*. These jobs mirrored certain observations made in [31]: the average write-request size was small (they varied between 500 and 1200 bytes for the four files), and CMMD files were, on average, an order of magnitude larger than CMF files. We used 32 CPs, 16 IOPs, 16 disks, and a 7-by-7 bidirectional torus interconnect in all the experiments. The write-cache size per CP for ENWRICH was fixed at 1 MB. ENWRICH outperformed TC comfortably in all the cases (Table 1). ENWRICH performance was as high as 28.1 MB/s (about 74.9% of peak disk throughput), and in the worst case was 20.5 MB/s (57.6% of peak disk throughput). In contrast, the best performance of TC was only 13.8 MB/s (36.8% of peak disk throughput). There were no directory flushes with ENWRICH, like those we experienced for certain 8-byte patterns in synthetic experiments. The average request size was small (a few hundred bytes), which seemed to hurt TC performance much more than ENWRICH performance. The request sizes were not so small as to cause excessive metadata overhead in ENWRICH.

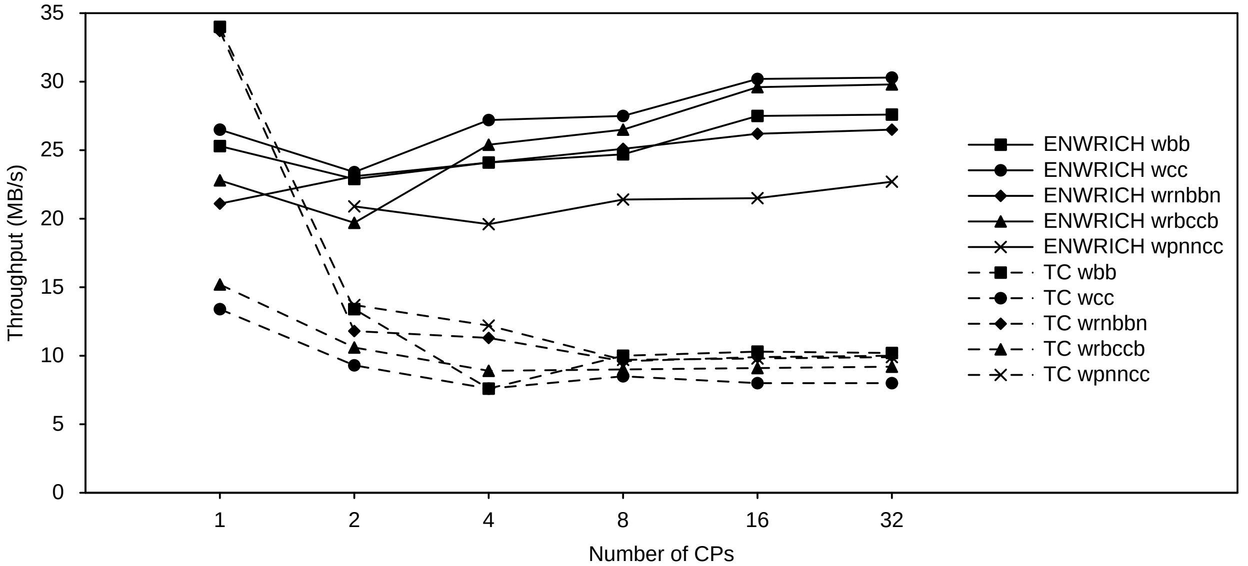

Figure 7: Graph showing performance variations of ENWRICH and TC with a changing number of CPs for some patterns. The number of IOPs and disks were kept at 16. All experiments used contiguous disk layout, 8192-byte records, and 64 MB files. Per-CP cache size for ENWRICH experiments was fixed at 1 MB.

Trace name	TC throughput in MB/s	ENWRICH throughput in MB/s
cmmd	13.8	28.1
cmf1	10.5	20.5
cmf2	9.6	22.3
cmf3	8.0	21.6

Table 1: Table showing performance of ENWRICH and TC driven by traces of some CMMD and CMF jobs that ran on a CM-5.

6 Conclusions and Future Work

Writes to checkpoint files, output files and intermediate result files are common and important activities in a multiprocessor file-system workload. We propose a compute-processor write-caching technique that essentially collects a large number of writes and flushes them to the disk intermittently using the high-performance disk-directed I/O technique. Qualitatively, our design allows disk-directed I/O to be used with any interface for writes. Quantitatively, it provides an opportunity for disk-directed I/O to optimize disk-write operations not only over one large request but over many requests from one or more jobs. Also, as positive side-effects, our design dramatically reduces per-write latency compared to a traditional-caching system, and overlaps coherency checking with useful work that eliminates extra disk I/O.

A smart disk scheduler could improve TC performance, but it is still unlikely to match ENWRICH because the scheduler could only make optimization decisions over a few blocks. Disk-directed I/O increases the I/O processor's knowledge about the sequence of blocks involved in one collective request, and ENWRICH goes further by increasing the knowledge of the IOP to span many requests. In ENWRICH, once the compute-processor (CP) caches accumulate a lot of data and decide to do a flush, the "disk-directed" I/O processors (IOPs) form a distributed disk scheduler. ENWRICH is also scalable, in that it avoids pulling all the data over to the IOPs until it is needed, which is critical if IOP memories are smaller than the total CP cache, a likely scenario in systems with many CPs.

Our experiments show that, overall, ENWRICH yields high performance. In many cases it yielded two to three times the throughput of TC and in one case it yielded 13 times as much. It was substantially slower than TC only in one pattern when using the contiguous layout (*wnn*). ENWRICH did, however, suffer from metadata overhead for some 8-byte patterns that brought its performance down to the level of TC. A more tuned implementation of ENWRICH with dynamically partitioned caches, however, should improve ENWRICH performance for such small writes. It is also noteworthy that writes so small are rare in representative workloads, and when they do exist they usually transfer a small overall fraction of the data, unlike our experiments where they transferred *all* the data. Our trace-driven experiments, although derived from only a portion of the workload, do indicate that ENWRICH can sustain high performance under a realistic workload with a mix of files and jobs.

Here we have designed and evaluated only a first-generation system. We need more thorough evaluation that can simulate bigger systems to ascertain the scalability issue. Currently we can only simulate 32-CP systems, due to memory limitations. Implementation of ENWRICH on a real machine and evaluation of its performance under real applications are important future goals. The design of ENWRICH is certainly optimizable in many ways that include dynamic distribution of per-CP cache space between data and metadata, allowing other concurrent activities at the CP during a flush, and processing received directories at the IOP while waiting for outstanding directories. Integration of ENWRICH with read caches is also necessary for its use in production systems. Gather/scatter message passing, by reducing the number of network messages, might help ENWRICH to adapt to cluster file systems with slower interconnects.

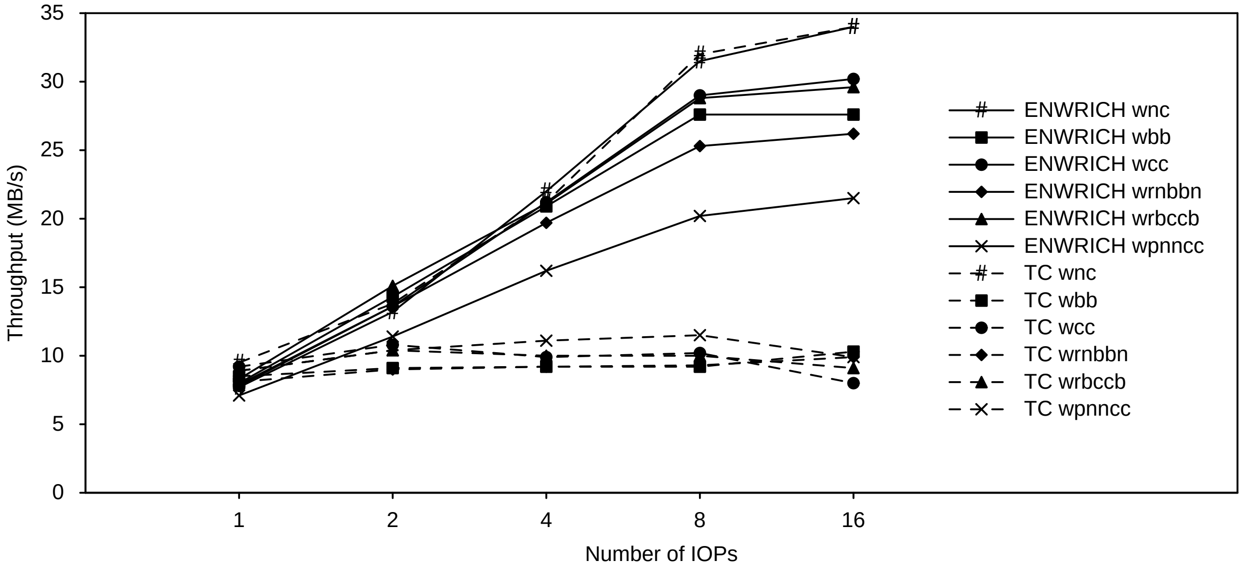

Figure 8: Graph showing performance variations of ENWRICH and TC with changing number of IOPs for some patterns. The number of CPs and disks were kept at 16. All experiments used a contiguous disk layout, 8192-byte records, and 64 MB files. Per-CP cache size for ENWRICH experiments was fixed at 1 MB.

Acknowledgments

Many thanks to Sid Chatterjee at UNC for allowing the use of their DEC 5000 compute server; to Jeff Chase at Duke for allowing us to test the early versions of our simulator on his lab machine; to Song Bac Toh and Sriram Radhakrishnan at Dartmouth for implementing and validating the disk model; to Gershon Kedem, Tom Alexander, and Surendar Chandra at Duke for many helpful discussions; and to our shepherd Denise Ecklund for guiding the paper through its final stages.

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *Proceedings of 15th ACM Symposium on Operating Systems Principles*, pages 109–126. Association for Computing Machinery SIGOPS, December 1995.

[2] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-Volatile memory for fast, reliable file systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 10–22, 1992.

[3] Michael L. Best, Adam Greenberg, Craig Stanfill, and Lewis W. Tucker. CMMD I/O: A parallel Unix I/O. In *Proceedings of the Seventh International Parallel Processing Symposium*, pages 489–495, 1993.

[4] Prabuddha Biswas, K.K. Ramakrishnan, and Don Towsley. Trace driven analysis of write caching policies for disks. In *Proceedings of the ACM SIGMETRICS*, pages 13–23, 1993.

[5] Rajesh Bordawekar, Alok Choudhary, and Juan Miguel Del Rosario. An experimental performance evaluation of Touchstone Delta Concurrent File System. In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 367–376, 1993.

[6] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. Proteus: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, MIT, September 1991.

[7] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO parallel I/O interface. In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 1–15, April 1995.

[8] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, and Sandra Johnson Baylor. Parallel access to files in the Vesta file system. In *Proceedings of Supercomputing '93*, pages 472–481, 1993.

[9] Thomas H. Cormen and David Kotz. Integrating theory and practice in parallel file systems. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 64–74, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies. Revised from Dartmouth PCS-TR93-188.

[10] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[11] R. Floyd. Short-term file reference patterns in a UNIX environment. Technical Report 177, Dept. of Computer Science, Univ. of Rochester, March 1986.

[12] N. Galbreath, W. Gropp, and D. Levine. Applications-driven parallel I/O. In *Proceedings of Supercomputing '93*, pages 462–471, 1993.

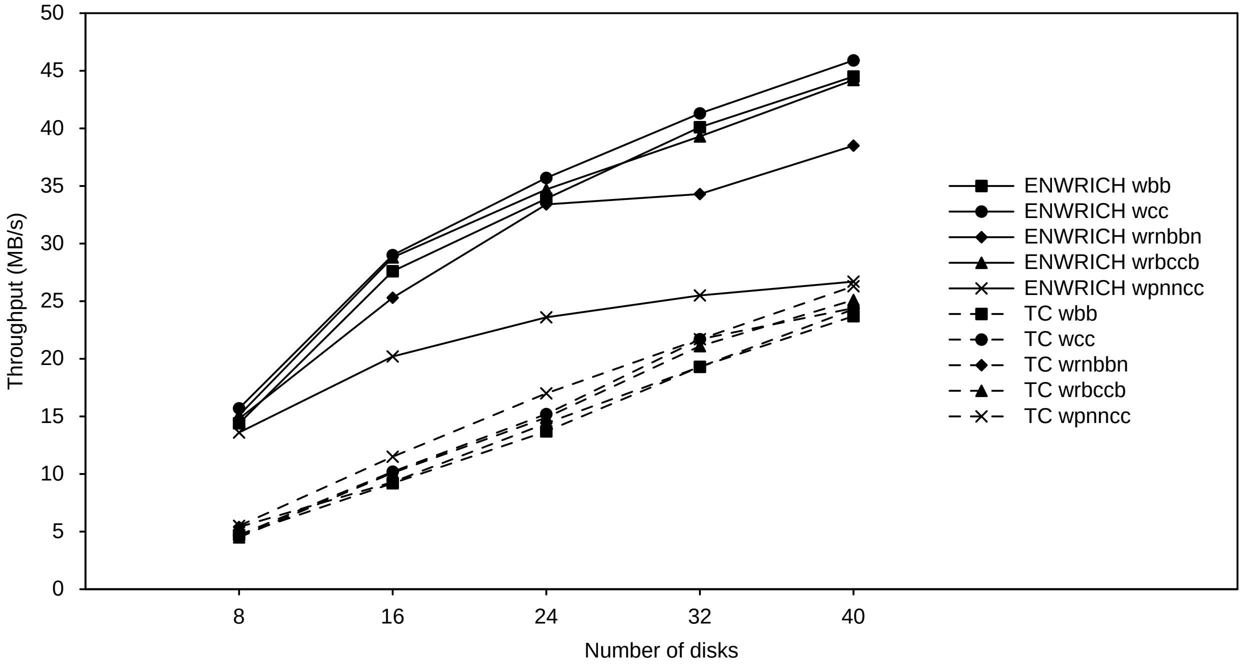

Figure 9: Graph showing performance variations of ENWRICH and TC for some patterns, varying the number of disks and using a **contiguous** layout. There were 16 CPs and 8 IOPs in all the experiments. The ENWRICH cache size was fixed at 1 MB per CP.

[13] Garth A. Gibson, Daniel Stodolsky, Pay W. Chang, William V. Courtwright II, Chris G. Demetriou, Eka Ginting, Mark Holland, Qingming Ma, LeAnn Neal, R. Hugo Patterson, Jiawen Su, Rachad Youssef, and Jim Zelenka. The Scotch parallel storage systems. In *Proceedings of 40th IEEE Computer Society International Conference (COMPCON 95)*, pages 403–410, San Francisco, Spring 1995.

[14] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 29–43, 1993.

[15] High Performance Fortran Forum. *High Performance Fortran Language Specification, 1.0 edition*, May 1993.

[16] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, July 1995.

[17] Concurrent I/O application examples. Intel Corporation Background Information, 1989.

[18] David Kotz. Multiprocessor file system interfaces. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 194–201, 1993.

[19] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[20] David Kotz. Disk-directed I/O for an out-of-core computation. In *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing*, pages 159–166, August 1995.

[21] David Kotz. Expanding the potential for disk-directed I/O. In *Proceedings of the 1995 IEEE Symposium on Parallel and Distributed Processing*, pages 490–495, October 1995.

[22] David Kotz. Interfaces for disk-directed I/O. Technical Report PCS-TR95-270, Dept. of Computer Science, Dartmouth College, September 1995.

[23] David Kotz and Carla Schlatter Ellis. Caching and writeback policies in parallel file systems. *Journal of Parallel and Distributed Computing*, 17(1–2):140–145, January and February 1993.

[24] David Kotz and Nils Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of Supercomputing '94*, pages 640–649, Nov 1994.

[25] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. A detailed simulation model of the HP 97560 disk drive. Technical Report PCS-TR94-220, Dept. of Computer Science, Dartmouth College, July 1994.

[26] Nils Nieuwejaar and David Kotz. Low-level interfaces for high-level parallel I/O. In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 47–62, April 1995.

[27] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best. File-access characteristics of parallel scientific workloads. Technical Report PCS-TR95-263, Dept. of Computer Science, Dartmouth College, August 1995. Submitted to IEEE TPDS.

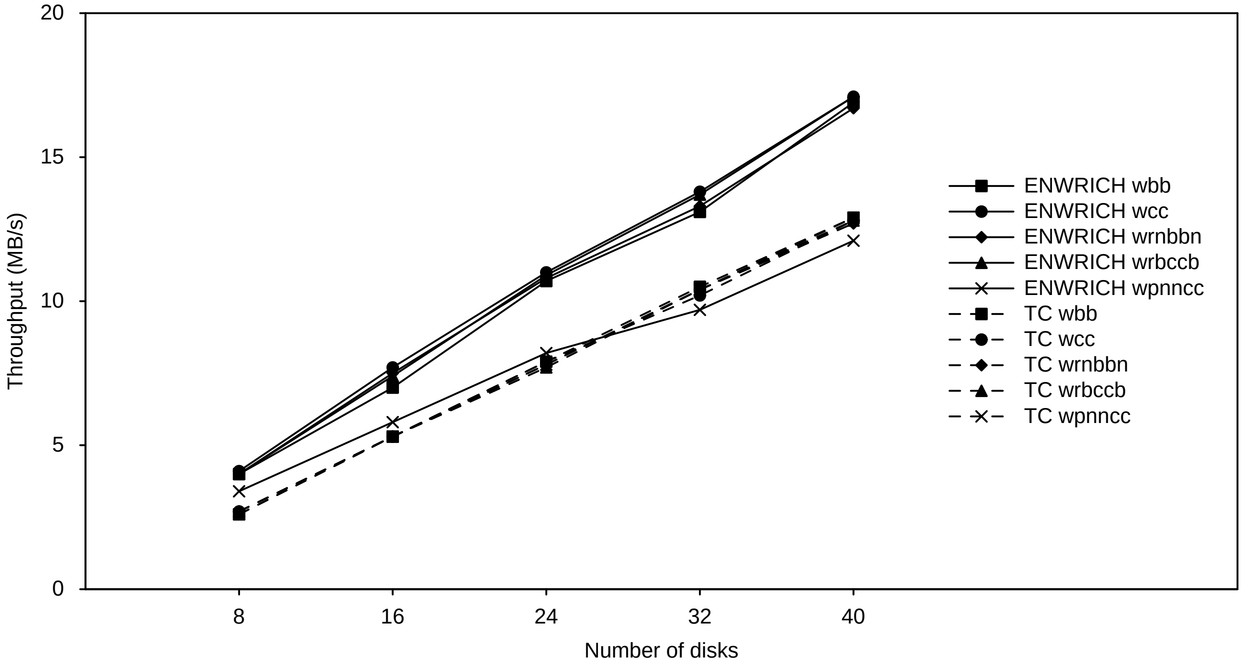

Figure 10: Graph showing performance variations of ENWRICH and TC for some patterns, varying the number of disks and using a **random-blocks** layout. There were 16 CPs and 8 IOPs in all the experiments. The ENWRICH cache size was fixed at 1 MB per CP.

[28] Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992.

[29] J. Ousterhout, H. DaCosta, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A trace driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of 10th Symposium on Operating System Principles*, pages 15–24, December 1985.

[30] James Pool. Preliminary survey of I/O intensive applications. Technical Report CCSF-38, Caltech Concurrent Supercomputing Facilities, January 1994.

[31] A. Purakayastha, Carla S. Ellis, David Kotz, Nils Nieuwejaar, and Michael Best. Characterizing parallel file-access patterns on a large-scale multiprocessor. In *Proceedings of the International Parallel Processing Symposium*, pages 165–172, April 1995.

[32] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[33] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, March 1994.